Nios II Embedded Processor Design Contest

Outstanding Designs 2005

Foreword

Since its introduction in June 2000, Altera's Nios® and Nios II soft-core processors have rapidly been integrated in a wide range of commercial applications. From video broadcast systems and WiMAX base stations using Stratix II FPGAs to SOHO networking equipment and automotive telematics using Cyclone® II devices, academic and professional designers worldwide have discovered the advantages of using Nios processors. Today, more than 15,000 development kits have been shipped, and over 5,000 companies—including the world's top 20 original equipment manufacturers (OEMs)—are licensed to use Nios processors.

The versatility of the Nios processor is a big factor in our success. Designers can tailor Nios systems with the exact peripherals, memory, and interfaces required, and add their own proprietary functions—their own "secret sauce"—to create a unique competitive advantage. Altera solves the IP integration problem with a tool that allows designers to drag and drop the exact mix of functions required, so they can focus on higher, system-level requirements instead of mundane, error-prone, manual tasks. Plus, Altera® tools work seamlessly with other industry-standard tools, minimizing training time. The soft-core implementation enables easy software and design upgrades, effectively making the design obsolescence-proof. With the added advantages of FPGA flexibility, fast time-to-market, and system integration, designers have a risk-free path to a custom embedded solution.

Altera continually cultivates designers in Asia and around the world through our University Program. As part of that program, the Nios II Embedded Processor Design Contest aims to increase student interest in embedded processors, improve their design and creative abilities, and ultimately motivate the continued development of FPGA-based embedded designs. This year, we received 578 qualified entries—nearly three times last year's number of entries. The significant growth of the program is yet more proof of how rapidly the Nios design community is expanding.

The 20 winning entries presented in this book showcase not only the breadth of possibilities that can be addressed using Altera's embedded solutions—including everything from digital imaging, watermarking and storage centers to an electric network monitoring system and a mechanical control system—but also technology trends in the industry. When designers have the tools and flexibility they need, there are no limits to what they can create.

Congratulations to all the Nios II Design Contest winners and their professors. Keep the fires of innovation burning!

Jordan S. Plofsky
Senior Vice President, Marketing
Altera Corporation

Preface

In today's competitive environment, the integration of multiple complex computational and processing units to build a System on a Chip (SoC) is an increasingly challenging task. The task of building a SoC involves creating an exact-fit embedded processor with the inclusion of user-defined instructions to accelerate the execution of time-critical software algorithms. Moreover, the SoC needs to be flexible enough to cater for rapid adaptation to design requirements or changing standards, and the ability to update the features and product enhancements through remote deployment.

The continual technology advancements in programmable logic, through increased performance and higher density devices, has created an opportunity to design cost effective scalable processor systems for SoC applications. These scalable "soft processors," implemented using general logic primitives rather than a hard, dedicated block in the programmable logic devices, provide a superior alternative for real-time processing needs in most applications. The SoC designers can choose from any combination of highly customizable features that will bring their products to market faster, extend their products' life cycle, and avoid processor obsolescence.

Altera provides one such flexible soft processor (Nios II). The Altera Nios II processor is a 32-bit RISC soft core with a rich instruction set optimized for embedded applications. It can achieve a performance of over 200 MHz. The Nios II processor is designed to be flexible, giving the user control of a number of features such as the execution units, cache sizes, memory and I/O interfaces, and a flexible bus architecture. The configurability aspect of the Nios II processor allows the user to trade-off features for size to achieve the necessary performance for the target application.

The Nios II solution comes with a complete development suite. Tools include Eclipse-based integrated development environment (IDE), embedded software, operating systems, middleware, and debuggers. It is also supported by SOPC Builder, the system-level design and integration tool.

To increase the understanding of SoC design methodology and promote its take up by the design engineers and researchers early in their careers, Altera has been successfully conducting an annual Nios Student Design Contest for the Asia-Pacific region. This positive initiative from Altera has encouraged some of the best young talents in the educational institutions in this region to produce exceptionally innovative designs for many challenging DSP and embedded systems problems requiring SoC solutions.

Dr. Saeid Nooshabadi
Senior Lecturer
University of New South Wales, Australia

Contents

Consumer Applications

Industrial Applications

Communications Applications

Appendix

First Prize

High-Speed Image Evidence Collector Based on Dual Nios II Soft Core Processors

Institution: **School of Communication and Information Engineering, Shanghai University**

Participants: **Lu Xiaofeng, Wu Bainian, and Huang Yan**

Instructor: **Lu Hengli**

Design Introduction

Currently, the laser velocity measurement forensics system used in road transport systems includes a velocimeter, charge-coupled device (CCD) camera, image card, and PC. Among these, the CCD camera, image card, and PC are components of car image forensics systems. Because car information collection and the forensics of road transport systems are carried out on site, timely communication with the traffic authority and cooperating institutions is important. Therefore, the system must be able to send road transport system data accurately, in real time. A system's real-time performance enables quick data collection and processing. Accurate data ensures clear images, which can be used by law enforcement agencies, and the convenience of data exchange enables effective cooperation between agencies. Because the system needs a human/machine interface, the system is too complex for portable, real-time operation. The solution is a system that makes it easy to collect and transmit transport system data in real time.

Keeping in mind the need for a compact system, we focused on the miniaturization of the original car image forensics system comprising a CCD camera, image card, and PC. We wanted to design a single image processing system with functions like multipath image collection, intelligent processing, intelligent display, and wireless transmission with a laser velocimeter to fulfill the need of monitoring forensics tasks for a road transport system. By adopting Altera's system-on-a-programmable-chip (SOPC) solution, our forensics system contains the functions of dual-way asynchronous image collection, pre-storage, ASIC high-speed JPEG compress/uncompress, compact flash (CF) card picture storage, picture searching, multiple display modes such as single-way image and picture-in-picture, real-time character overlay and OSD, and remote wireless transmission.

With the Nios® II soft core processor, we can overcome design issues such as low-speed motion compensation unit (MCU) processing, limited peripheral resources, difficult I/O configuration, complex hardware design, and software programming. Using two Nios II soft core processors, we can distribute control and access to multiple peripherals logically, so that communication between them are coordinated. This design also meets the requirements of time sequence and functions, makes the best use of the processor's resources, and greatly improves the overall operational efficiency of system. Because the system has external memory and I/O, memory access is frequent. Through the Nios II processor's user-defined peripherals, user-defined logic, and direct memory access (DMA), memory access and data movement are simplified when accessing SDRAM, SRAM, and flash memory. Additionally, we can implement real-time data overlapping and OSD. In practice, the road transport situation is subject to frequent changes requiring constant system upgrades. However, while making system upgrades, you cannot change the system hardware due to cost and time issues. By combining the requirements of both software and hardware in a coordinated development process, Altera's SOPC solution offers the best choice for the team because this solution can fully showcase the advantages of a multiple soft core processors in combination with an FPGA's logic control and data processing capabilities. This design approach allows for a flexible system configuration and simple and convenient development, supports various processing modes, and offers powerful data processing capacity at low cost.

With China's auto industry forecasted to post high growth, a highly integrated and portable forensics system for cars has a bright market future. Because the system can fulfill mobile random forensics and offer a real-time position fix, it is very useful for public security and road transport departments. Additionally, the system can perform functions such as information collection, processing, and transmission of data on legally registered automobiles, thereby enhancing timely and safe operation of the road transport system. Our design results in an effective monitoring and forensics system for cars, which is a crucial component of an intelligent transport system.

Function Description

Our system collects, synchronizes, displays, and stores dual-camera asynchronous images. These functions include the collection of images based on long shot and close shot distances, and close shot images with detailed features. All recorded images are transformed based on modulus 8-bit YUV (CCIR-656) format.

Customized I²C bus peripherals with the Nios II soft core processor control the two-way video ADC's initialization. The Altera® FPGA's on-chip hardware logic elements handle image collection, synchronization, and memory read/write functions. The two-way asynchronous images are synchronized using two blocks of 8-Mbyte SDRAM cache. These SDRAMs store each field of two-way images within the three-field image time, respectively, and then alternate read-write of two blocks' memory cache. At the same time, the program advances to the next memory location, that is, by writing memory cache B while reading memory cache A, and writing memory cache A while reading memory cache B. While writing into memory cache, each field of two-way images is written to within a three-field time, but only one-way image data is read while reading the cache.

We can display these two-way images separately or within a picture-in-picture (PIP) configuration. We display these images in real time and skip caching image data when it is one-way image display. Data is updated and displayed in real time while it is being written into storage memory cache by the method previously described. When the desired main image display format is displayed as PIP, the image data of the main picture is not cached and the sub-picture data is the field extraction image by alternate read operations from the SDRAM cache memory. According to the control mode, if you consider the close shot image as the main picture, the sub-picture data is read from cache B and the main picture image is computed from all updated fields while the sub-picture image is updated for every three fields. In contrast, if you consider long shot image as the main picture, the sub-picture data is read from cache A using the mode previously discussed.

2

Because each sequential burst maximum read/write capability is limited to 512 bytes, the external SDRAM cache selected cannot write the interlace field data of 720[x]288 resolution into SDRAM in real time. We solved this problem by designing two dual-port RAMs (using FPGA logic) to act as a cache memory block of external SDRAM. This scheme ensures that SDRAM continuously reads/writes a whole-line of image field data in real time.

We implemented the dual-port RAM design through logic control elements in the Nios II soft core processor and the first block of the FPGA.

Real-Time Character Overlapping

When the system monitor starts to capture image data manually, the system starts processing data on the spot. If the system monitor uses an automatic process, the system performs a fixed-site execution. The system monitor communicates with the laser velocimeter through an RS-232 interface. When a speeding vehicle is monitored, the laser velocimeter provides information such as vehicle speed, distance from the velocimeter, and current time by series. The monitoring system compares this information in real time against the stored images database for evidence purposes used by law enforcement agencies. All of this data is displayed on the LCD.

Before storing the compressed images for display purposes, we used the Nios II soft core processor to translate it into a regional code character library and stored it in the external flash memory. We then stored the images in the dual-port RAM of the FPGA. Next, using suitable logic circuits, we were able to read this character lattice from the RAM and overlap it on the image to be compressed and stored. This form of the DMA mode improves the operational efficiency of the system. During overlapping, the image color is pre-defined and overlapping characters are changed into grayscale (because only letters containing one type of color are overlapped), and 1-bit character information is read every time. The character information corresponds to a byte of image data. Due to the overlapped grayscale image, Y is assigned the least grayscale value if some characters must be overlapped. If they do not overlap, Y is reduced by a 32-bit grayscale grading to highlight overlapped characters.

We created this grayscale overlapping function using logic control elements of the Nios II soft core processor in the first FPGA. In doing so, we bypassed the costly traditional character-overlapping function module. After this operation, the system passes image data that have been synchronized and overlapped with necessary characters to the FPGA for further processing and display.

Early Image Memory

The system monitor must support an early image capture memory function because of the laser velocimeter instrument and the time delay generated during RS-232C communication between the laser velocimeter and the system, as well as requirement of law enforcement in traffic system. In other words, the system must be able to store image information of a T-1 period or even earlier according to T-period requirements. Keeping this crucial design requirement in mind, we designed our system to control the image cache using a large external SDRAM to ensure enough image storage precision to meet the law enforcement requirement when storing images. The system applies cache A and B during a two-way asynchronous synchronizing process, during which time the early memory storage function is also completed. The early image capture memory process is related to the size of the cache and the address position data read from the cache. In theory, the system can implement an intense early memory function if the cache is large enough. For the current design, because the image time comprises three fields of storage on cache plus the expected execution time of the Nios II soft core processor during control compression and read/write into CF card memory, the system can guarantee an early memory storage requirement of greater than 0.3 seconds.

Image Compression/Decompression

Because the images are available in the CCIR-656 format, we directly adopt a scheme turning one line into two lines, instead of computing direct interpolation values for mobile images to change the image format from 640[x]240 to 640[x]480 to ensure image-memory quality. Although, the scheme of one line turning into two lines adds extra edges to the image, it keeps the definition of the compressed image. In addition, the size of every image can be up to 600 Kbytes or more, which is a great load for memory and radio transmission. Hence, we must compress the images in advance. The system adopts JPEG compression based on the ZR36060 device with about 83% compressibility (affected by image data). The compressed JPEG picture occupies only 100 Kbytes per image or more with higher compressibility requirement. However, the reserved images comprise one data field of the interlineations scan odd/even field. These compressed images, improve memory speed and efficiency of image processing and meet the requirement of image radio transmission. The chip also supports a decompression function.

The Nios II soft core processor performs initialization and compression/decompression timing, and commands the ZR36060 compression chip by way of customized peripherals. The compression chip operates under the control of the Nios II processor and logic elements, and the compression function is realized by two blocks of external SDRAM (cache A and B) as well as RAM within the FPGA. Cache A and B, respectively, store a field of distant and close shot images during asynchronous image synchronizing. If a close shot image is required for compression, the system reads data of the close shot image from cache A to FPGA RAM, and then the logic elements send data to the ZR36060 for JPEG compression. In this case, cache B only stores one field of distant image and close shot image in a six-field data time. By contrast, if a distant image is compressed, the system reads the distant image data from cache B to FPGA RAM for compression. If PIP image is required for compression, e.g., a close shot image of the main picture, the system first reads the close shot image from cache A to interior RAM, and then reads the distant image of extraction frame from cache A to the specified place of RAM. When logic elements read and compress data from the dual-port RAM, the system reads the scrambled image in PIP so as to implement the PIP image compression. In this case, cache B only stores one field of distant image and close shot image data within six-field data time.

Note that image decompression of stored images in the CF card is a reversible process of the previously described compression process.

When the system is compressing images, no real-time image data is sent to the FPGA and cache C and D for processing. This process enables the FPGA to repeatedly send image data in cache C to cache D and LCD to freeze the display picture. Then, the system decompresses the latest data from CF card, delivers it to the FPGA by data bus transfer, and writes to cache C. The FPGA later reads the latest image data from cache D and freezes them onto LCD. So every time image memory data is processed manually or automatically, cache C sends the latest decompressed image data to cache D. The images are frozen on the LCD from the latest image data of cache D to avoid frame-skipping and black screen during compression/decompression and image storage processes, thus guaranteeing a continuous and smooth image display. In addition, this operation also involves updating and intercommunication with the FPGA RAM when the Nios II processor operates cache C and D.

The function was realized by the control and peripheral access functions of data and command communication between two Nios II soft core processors in two blocks of the FPGA. Accessing the compressed chip was performed through the customized parallel input/output (PIO) mode of the Nios II processor.

CF Card File Management System

Because images must be stored in real time during image collection, and a great number of images need to be stored while monitoring speeding instances, the system needs substantial memory. Therefore, the monitoring system must support the right kind of memory medium featuring large capacity, low cost,

and easy operation. Our system uses a commonly available CF card with a 256-Mbyte capacity, which can store more than 1,000 groups of pictures. Combined with the LCD display, the CF card can be used on our system for browsing and searching images or on a PC for easy operation.

The Nios II processor starts initializing both the CF card and compression chip, and builds the catalog contents and file allocation tables (FAT) of the currently stored files in the CF card. Next, compressed images can be stored onto to the CF card under the control of Nios II logic elements. However, the Nios II processor must update catalog contents and FAT of the second picture before the second picture is stored.

CF card file management involves communication between the two Nios II soft core processors and the CF card during the storage and read of CF card data. The data channel is connected directly, and the command channel depends on I^2C bus. The communication mode responds according to the compression process for access and display of external memory data. All keystrokes used by the control process adopt the I^2C bus access mode.

According to the requirements of the traffic system, the monitoring system must be able to store data similar to the size of an image under question to track aberrant behavior on the spot. This feature helps the inspector working with the system to track down an over-the-limit speeding vehicle with high-quality pictures. Using this principle, we designed two memory modes in our system: one mode for two pictures and the other for three pictures. The system can randomly select two or three pictures to store in a single image or PIP image under these two memory modes. In addition, the monitoring system can browse, search, cancel, and format the stored pictures on the CF card. The system LCD display is user friendly in that it hints all possible operations (including OSD display mode features), its visualization, and possible shortcuts.

File management on the CF card is realized through control and peripheral accessing functions of the Nios II soft core processor as well as the command communication function of the FPGA. We have implemented the I^2C bus and CF card interface through a customized PIO mode.

GPRS Wireless Transmission of Images

Because a road traffic system involves intense communication between law enforcement units and related locations, our system features a peripheral general packet radio service (GPRS) module. This module communicates with the RS-232 serial port and the Nios II soft core processor of the first FPGA, and enables wireless transmission of images involving traffic violation (venues and vehicles). The module allows users to browse the pictures stored on the CF card directly, and transmit them as forensic evidence to fixed IP addresses on the Internet through a GSM network. This function was mainly realized through accessing serial port peripherals by the Nios II soft core processor in the first FPGA device.

OSD Display Mode

Because we need to display status information of the CF card, GPRS module, and user menu on the LCD screen along with the image, we adopted the OSD display mode for easy operation.

The OSD overlapping character comprises two parts. The first part deals with the law enforcement location and the enforcement code number on the upper screen. This data is sent to the first FPGA via a serial port connected to the front-end laser velocimeter to obtain information. Then the main processor in the first FPGA sends this data to the subordinate FPGA via the I^2C bus to implement the OSD display. The second part deals with the status and menu prompt information. Information in this mode is obtained using direct data transmission between on-chip and off-chip memory under the Nios II soft core processor control. This operation is similar, in theory, to DMA mode, but it does not use DMA peripheral control directly. Instead we use the external SRAM's custom-defined write logic. When data

is translated into RGB format from YUV format in the second FPGA, the Nios II processor begins to write OSD content stored in the external flash memory to an external SRAM chip via the internal RAM of the FPGA. Then, the logic control unit takes in OSD data read in the external SRAM to the transmitted RGB image, and displays it in OSD mode. The Nios II processor modifies the OSD content character bitmap in the external SRAM via the internal RAM of the FPGA during each data access.

The OSD data identifies an overlapped word data with a 16-bit word. Because the text displayed in our current system is black and white, we only use three bits. If necessary, we can use all 16 bits to display the text in color. The Nios II processor uses custom logic to write the external SRAM. We have used a DMA-like mode when data in the external SRAM is transmitted to internal RAM and overlapped under Nios II processor control.

The logic unit translates the image format, from YUV to RGB, before OSD affected in the second FPGA. During this process, we implemented line interpolation to create smoother images for easy viewing of the display.

Performance Parameters

These parameters are the actual performance parameters of the system, which were obtained after implementing the system design. The actual system performance parameters are:

■ *Power supply*—DC voltage: 9 – 12 V; operating current: 600 mA (motherboard current is about 200 mA); power consumption: 10 W

■ *Environment*—Operating temperature: 0° C – 40° C; relative humidity: 8% – 95%

■ *Input image*—Mode: two-channel asynchronous image; data format after digitalization: CCIR-656

■ *Display*—LCD resolution: 640[x]480; LCD display area: 16 cm² or 6.4 inch²; LCD display color depth: 6-bit/RGB

■ *Storage*—Storage image resolution: 640[x]480; storage image color depth: 8-bit, JPEG; storage image capacity: about 100 Kbytes/image; image storage total cycle: < 0.5 s/2 images

■ *Transmission time*—About 1 minute/image (depending on local network condition)

We have used two Nios II soft core processors in this design to manage the FPGA's internal resources, define the time sequence requirements for data processing, and handle display and control of the system. In addition, we needed to make frequent access to multiple peripherals from the main system and the Nios processor helped us to improve the overall system operational efficiency. The dual-core system fully utilizes all features and performance of the Nios II soft core processor. The following functions were included in the design:

Functions of the Nios II soft core processor in the first FPGA:

■ Main processor, controlling logic unit for the whole system.

■ Performs intra-soft core communication with Nios II soft core processor in the second FPGA via user-defined I²C bus peripheral.

■ Handles data and command communication with front-end laser velocimeter through RS-232 serial port peripheral.

■ Handles all operation instructions to FPGA internal logic circuits through user-defined PIO peripherals.

■ Acts as the main component of I^{2}C bus in the whole system, controlling all subcomponents on the I^{2}C bus in the system, such as image analog-to-digital (A/D) conversion chip, user buttons, and Nios II soft core processor in the second FPGA.

■ Communicates with the GPRS module via RS-232C serial port peripheral.

■ Converts the standard region code character library in flash memory into character dot matrix.

■ Issues partial control instructions during image compression and CF card storage while initializing the compressed chip and CF card at the same time through user-defined peripherals.

■ Performs file allocation table (FAT) file management system of CF card via user-defined peripheral.

Functions of the Nios II soft core processor in the second FPGA:

■ Subprocessor.

• Handles intra-core communications with main Nios II soft core processor in the first FPGA via I^{2}C bus.

• Issues all operation instructions to FPGA internal logic via user-defined PIO peripheral.

• Implements self-defined logic to write into external SRAM.

• Handles DMA data access mode between external SRAM data and on-chip logic.

• Converts standard character library in Flash memory into character dot matrix.

■ Implements OSD display.

Design Architecture

Figure 1 shows the system block diagram.

Figure 1. System Block Diagram
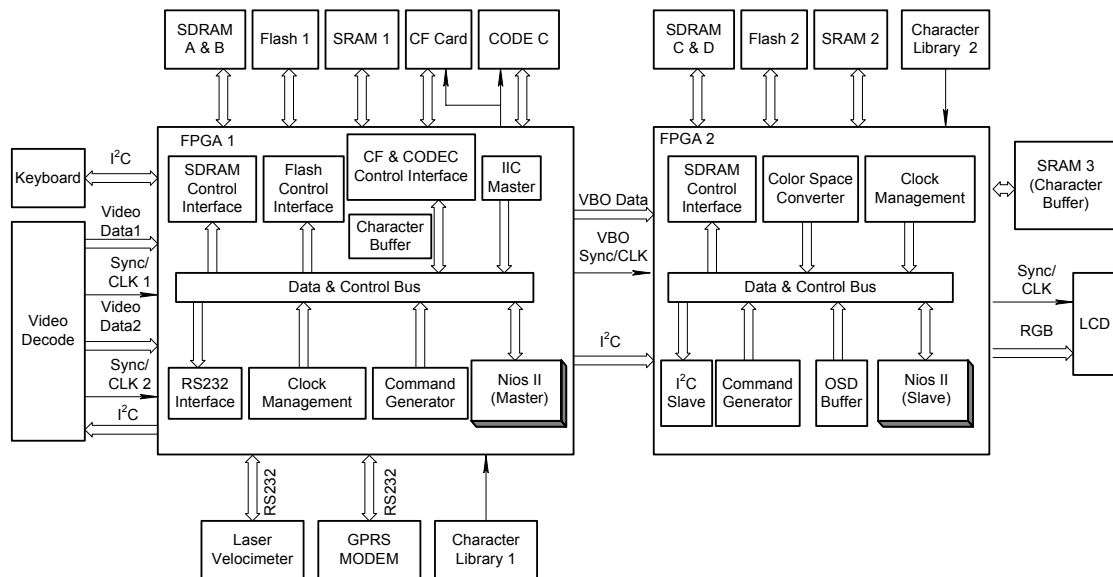
Design Methodology

This section describes the design methodology we used for the system.

System Function Design & Test

Using Altera's UP3 development board, we were able to design most of the system functions, test, and simulate all functions. In addition, we designed a few circuit boards for design and test. We did our system design and testing as outlined in the steps below:

1. Design the technical parameter and function indexes that need to be implemented by high-speed image storage.

2. Refer to examples and testing documents on UP3 development board, understand Nios II soft core architecture, software programming using C language under IDE environment, and online program debug and programming for the device.

3. Adopt SOPC Builder to implement peripheral storage access, RS-232C serial port, and PIO on UP3 development board.

4. Implement access of hardware through IDE interface and setup file management system on UP3 development board (the IDE interface on UP3 development board also defines the sequence error of address pinout definition).

5. Implement user-defined peripherals and logic on UP3 development board.

6. Implement collection, compression, decompression, and CF card storage for a single processor image.

7. Implement real-time character overlapping for single processor image.

8. Realize LCD OSD display.

9. Realize synchronization of dual-processor asynchronous images.

10. Realize picture in picture, collection, compression, decompression, and CF card storage for dual-processor asynchronous images.

Hardware Implementation

Our hardware design task was to combine the above-mentioned testing methods realized on UP3 development board and our circuits, then design the necessary hardware modules, build a high-speed image forensics system based on dual Nios II soft core processors, and implement the above functions.

Using the PROTEL 99SE tool, we were able to partition the design using the UP3 development board and our own circuit modules. This enabled us to decide upon the usage of two Cyclone® EP1C6Q240 FPGAs as controlling devices. All other functional modules in the system were designed based on these two FPGAs, which represents a high SOPC-integrated design concept and principle in hardware. Because the Nios II soft core processor is available to be designed in both FPGAs, only peripherals such as SRAM and flash devices needed to be configured. Due to the design requirements of image buffer and freeze, a high-capacity SDRAM needed to be configured as a buffer. We also needed to perform JPEG image compression; so we decided to use an ASIC to perform compression. Our design treats CF card as storage media in order to implement image storage, and we chose the CF card to make the storage media easy to use. The dual-processor simulated image had to be digitized to achieve effective processing. Therefore, the system had to configure the A/D module for both single and dual-processor images. Other system peripherals include a power management unit, LCD interface, two serial ports (for communication with front-end laser velocimeter and GPRS module), and operational buttons.

To make it easy to understand the system hardware design process, we have split the design into the schematic diagram, functional verification of the schematic diagram, component purchasing, and PCB development, as described below.

1. *Schematic diagram design*—Because most of the functional testing and simulation has been completed on UP3 development board and self-designed circuits, the schematic diagram design mainly refers to our own circuit designs that effectively combine these into the most representative of the circuit system schematic diagram. Thanks to the adoption of Altera SOPC solution, all processors and functional control units were integrated into two FPGAs, further simplifying the design structure.

2. *Functional verification of the schematic diagram*—Although most of the functional testing has been completed on testing board, hardware integration needs functional verification. The functional verification of the schematic diagram is primarily to demonstrate the proof-of-the-concept in consideration with the guidance of the tutor. This process led to the confirmation of a complete circuit design.

3. *Component purchase*—We purchased the necessary components while designing the schematic diagram and its verification. All component packages were clearly marked on the schematic, which made PCB development easier.

4. *PCB development*—Our circuit board design uses a four-layer PCB scheme, which is capable of handling multiple signal wires and earth wire to ensure the normal functioning of the circuit board and perform all system functions.

Software Implementation

The software design task is to migrate the VHDL and C language program of the above-mentioned functions realized on UP3 development board and self-designed circuit to two Cyclone EP1C6Q240 FPGAs with two Nios II soft core processors. The design of all software modules was based on highly integrated hardware modules. We used the Altera Quartus® II software, SOPC Builder, and Nios II IDE to build the Nios II soft core processor and to develop the controlling program of the system, highlighting the SOPC solution's highly integrated and programmable concepts. Our software design was based, but did not completely rely, on hardware modules so that we could complete core modification and be able to make upgrades. The design also made it possible to easily implement, add, and remove multiple peripherals, access user-defined peripherals, and design user-defined instructions. Generally, the system is under control of the dual Nios II soft core processors. However, most of the software modules were implemented based on the cooperation between the Nios II soft core processor with the FPGA logic.

We used the VHDL and C languages for the software design. We wrote the logic control and data processing program in VHDL, and used C language routines for the control program of the main and sub Nios II soft core processors. Based on the functional tasks, the system software is divided into system initialization, direct image display, character overlapping of image data, image compression and decompression, image storage, GPRS wireless transmission, and OSD display. The implementation method and steps are shown below:

1. *System initialization*—The system is initialized for all parts in the system by the main Nios II soft core processor via the I²C bus, which includes video A/D module and user buttons, initialization operation for the CODEC chip during compression and decompression, FAT file management system during CF operation, and directory entry and FAT modification.

2. *Image display*—The image is displayed on the LCD screen before it is captured and stored. The image can be displayed in single-processor mode or picture-in-picture mode. Although the image data seems to be directly available, the system is writing this data into external SDRAM cache unceasingly and repeatedly to meet the requirements of compression and storage in advance. It also refers to the access to external storage operations in user-defined peripheral mode by the Nios II soft core processor. Meanwhile, logic is needed to do a great deal of work for clock management, controlling command generation, and data channel handling to carry out functions set by the system.

3. *Real-time character overlapping*—The character overlapping software design includes translating standard characters into binary lattice information by the Nios II soft core processor, the access time sequence design of the lattice information, and reading/writing of external storage with internal RAM in the FPGA by the logic unit, which is performed by the Nios II processor and FPGA logic.

4. *Image compression and decompression*—Image compression and decompression refer to the initialization of a compression chip by the Nios II soft core processor, generation of control signal for compression and decompression is also provided. Logic handles the processing and transmission of other data.

5. *Image storage*—The Nios II processor plays an important role during the access of the CF card by the system. It creates a complete FAT file management system, initializes the CF card, and issues

control commands for storing and reading processes. Other data processing is performed in cooperation with the FPGA logic.

6. *GPRS wireless transmission*—Pictures stored in the CF card need to be transmitted wirelessly to a fixed IP address at any time based on the users' requirements. In this way, all law enforcement units of the whole road traffic system can share the latest data resources. The Nios II serial port peripheral accesses the GPRS module and to transmit the image data. This implementation provides more convenience for operating and programming the system.

7. *OSD display*—This function is mainly implemented by the sub Nios II soft core processor, including bitmap translation from OSD data, writing of bitmap data to the external SRAM, reading to the internal RAM of FPGA from data of external SRAM, and overlapping of OSD data to real-time image. We used the Nios II soft core processor to access the peripheral, write to the external SRAM in user-defined logic mode, and read/write external SRAM data in DMA mode. In addition, FPGA logic translates the image format, converges image and OSD data, and interpolates the image and data.

Design Features

This section describes the system's design features.

Dual Nios II Soft Core Processors

The master and slave Nios II soft core processors were implemented in two Cyclone FPGAs. The processors implement communication with Nios II-defined I^{2}C bus peripherals. This design takes full advantage of the dual-core cooperation to coordinate system data processing and display timing requirement, and utilize FPGA internal logics and storage device resources to share data processing, peripheral access, peripheral and internal logic control, enabling CF card file management system, user-defined peripheral access and control, user-defined logic, DMA, and OSD display. With this implementation, we could add extra system functions and control methods, while reducing the overhead of extra control units such as MCU, hardware circuits, and design complexity. This implementation led to lower software and hardware design costs.

Synchronization of Dual-Camera Asynchronous Images

We used I^{2}C bus peripherals defined by the Nios II processor to control the dual-channel video ADC initialization and schedule the FPGA logic to control image collection, synchronization, and storage. While synchronizing the two asynchronous images, the off-chip and on-chip cache can be used to synchronize the dual-channel asynchronous images, store image information in advance, and realize multiple display methods such as dual-channel and PIP. The direct access to multiple storage devices using FPGA often makes the programming task difficult and takes plenty of on-chip resources. We solved this design issue by using the Nios II processor's fast access to multiple storage device peripherals, which helped to reduce the design cycle time and expense.

Real-Time Character Overlapping

During the image capture process, the capturing system and laser velocimeter communicate through the RS-232 serial port. When detecting a speeding vehicle, the laser velocimeter provides the current speed of the vehicle, its distance from the velocimeter, and the precise time to the capturing system through serial ports. Then, this information is displayed through OSD on top of the screen by the system. Meanwhile, the character information, such as the execution site and force number, are overlapped into the saved pictures in real time to be used as evidence for police action. This design makes full use of the Nios II-defined peripheral functions to enable the system to communicate with the front-end

velocimeter in a timely fashion to obtain character information and instructions needed without having to develop a dedicated serial port driver.

ASIC Compression & Decompression

The system adopts an ASIC-based, ZR36060 device JPEG compression, delivering 83% compression ratio. The device can compress JPEG images down to 100 Kbytes, which not only improves the image storage speed and efficiency but also meets the requirements for image wireless transmission. The chip is also equipped with a decompression function. Its implementation relies mainly on a user-defined peripheral of the Nios II software to initialize the compressed chip and issue control instructions for compression and decompression tasks in coordination with the logic unit.

CF Card File Management System

The Nios II processor starts to initialize the CF card while initializing the compressed chip, and further creates a directory entry and FAT for the files being stored. Then, the compressed images can be saved to the CF card. The Nios II processor configures the CF card for saving every image. We have designed an image storage format of 2 or 3 images in a group. The Nios II software user-defined peripheral function is fully utilized when accessing or storing images onto the CF card, which allows the Nios II processor instructions to complete the complex operations on CF card.

Image GPRS Wireless Transmission

We devised externally connected GPRS modules to connect with the Nios II software and the first FPGA through serial ports to wirelessly transmit speeding vehicle images to the law enforcement, which facilitates communication between the execution units and sites. Users can view the images stored in CF card directly and then transmit these images to a fixed IP address on the Internet through a GSM network. The monitoring execution sites can communicate with each other through GPRS and quickly read the information on condition of the scene and vehicle in question. This communication allows the police to intercept these vehicles quickly, and further monitor the road traffic in real time.

OSD

The system uses OSD to facilitate user operation. It is needed because there are many prompts during system operation, status information of the CF card, GPRS, and various menus that need to be synchronized and displayed on the LCD along with images.

The characters overlapped on OSD include two parts. One is the execution site and police number on the top of the screen. The information is sent by the front-end laser velocimeter through the serial port to the first FPGA. The FPGA's master processor then sends the data to the FPGA for OSD. The other is the status and menu prompts. This information is obtained by transmitting data directly between on-chip and off-chip storage devices under the control of the Nios II soft core processor. This operation follows the same principle as DMA but does not use DMA peripheral control directly. Additionally, this function writes to the external SRAM using user-defined logic. When the data format is transformed from YUV to RGB in the second FPGA, the Nios II processor starts to write the OSD contents stored in external flash into an external SRAM chip via the RAM in the FPGA. The logic control unit overlaps the OSD data read from external SRAM into the RGB image whose format has been transformed for OSD. Then, the Nios II processor modifies the OSD content character matrix in external SRAM through the RAM in the FPGA during each blanking interval of the data field.

OSD data indicates an overlapped data point with one character (16-bit data). Now, only three conditions are used for black-and-white display. If necessary, all 16 bits can be used for color display. The Nios II processor uses user-defined logic to write to the external SRAM chip, while the data in external SRAM under the control of Nios II soft core processor is transmitted into RAM for overlapping through a DMA-like method.

Highly Integrated SOPC Solution

The whole system has been designed around two FPGAs. Making full use of multiple functions and features of two Nios II soft core processors, the control and data processing are managed by Nios II soft core processors in the FPGAs. While the design is processed with the FPGA software, the hardware system and software system can be combined, casting off the traditional reliance of software development on hardware. Additionally, the multiple soft core processors can be used to select appropriate peripherals, storage devices, and I/O interface to build a system that is well-tailored to match customer demands. This design approach also yields low price, simple design, high integration, and low risk.

Easy System Upgrades

Due to the nature of the user requirements and road traffic conditions, the system requires quick and frequent upgrades. The Altera SOPC solution frees users from upgrading hardware for software upgrades. Even if the product has been delivered to a customer, the software can be updated regularly. Users can add new features to the hardware continuously to reduce risks possibly caused by changes in standards and add functions, thus simplifying hardware repairs. They can also avoid processor obsolescence.

Collaborative Software/Hardware Development

Because the hardware and software of an SOPC system can be combined, the design of both can be jointly performed during the system development process. The development of software and hardware can begin and end almost simultaneously, saving a lot of time. Additionally, the SOPC design approach accelerates the pace of product launches and enables a longer product life cycle. You only need to generate a new Nios II kernel if you need to modify some definitions during development process, exerting no impact on other peripherals or other Nios II programs.

Lower System Cost

The Nios II soft core processor solution can help to achieve a good balance between system performance and system cost when designing systems involving higher integration, optional CPU (high-speed, economical, and standard), a multi-CPU system, and no fixed requirements for a processor implemented on an FPGA. Using the Nios II soft core processor, we designed an economical and practical monitoring solution for road traffic systems, while also taking into consideration the need for flexible system functions.

Conclusion

Before participating in the contest, we had some prior experience with Altera devices and possessed some development experience. When we started using Altera's SOPC solution incorporating FPGAs and the Nios II soft core processor, our original understanding of the FPGA changed. The FPGA functions that are single, parallel, and high speed with strong logic, simple time sequence, and complex operation were greatly expanded. We integrated designs that could be implemented by an external control unit into one FPGA, and removed/added the peripherals and interfaces of an internal Nios II soft core processor. Additionally, the Nios II soft core processor can be adapted easily for low-speed control, awkward peripheral access, and massive storage peripheral devices, all of which need frequent interface modifications. Furthermore, in accordance with many features of the FPGA I/O, there is always a problem of operation coordination and interface between the FPGA and control units. You need to take this coordination into account during software/hardware design, e.g., for data processing, command control, time sequence coordination, parallel/serial, and high/low speed. Between the Nios II software processor and FPGAs you can simplify system design and software programming. By migrating the OS into the Nios II soft core processor, you can combine the FPGA and traditional embedded CPU. At the same time, the kernel of the SOPC solution is the most effective.

Accordingly, an integrated SOPC solution and the Nios II soft core processor have totally overthrown the old embedded system design and greatly boosted the hardware/software design gains. Based on this evolution, an embedded system's hardware circuit is simpler and more effective and the software design can be visualized and migrated easily. It is certain that hardware circuit design is not as complex as before and "processor + memory + peripheral" is easy to understand. However, the software design is more complex than before. The hardware/software coordinated development technology of SOPC solution has solved the problem and enabled the synchronous development of FPGA logic and the Nios II soft core processor inner program to enhance the design efficiency.

In the contest, we created the dual Nios II soft core processor control, but did not migrate a real-time OS into the processor due to time limitations and difficult technical challenges. Additionally, our Nios II evaluation is deficient in this aspect. According to the current requirements of embedded systems, the realization of real-time OS migration shall be based on the improvement of system integrated efficiency and stability, which is also the direction of our efforts.

Because the SOPC solution and Nios II soft core processor were launched not long ago, some problems occur in real applications such as process speed deficiency, shortage of IP cores, an internal resource insufficiency of general mid- to low-level FPGAs, and expensive high-level devices, all of which have restrained the application of SOPC solutions in some fields. But we believe that with the growth of related technology, as well as innovations from Altera, system designers will be sharing many practical SOPC solutions before long. When that happens, by virtue of advanced tools, we will then design the most profound system in the most concise way.

First Prize

Passive Digital Camera

Institution: **Hanyang & Yonsei University**

Participants: **Ji Won Kim, Doe-Hoon Kim, and Shin Seung-Chul**

Project Leader: **Min-Chul Kwon**

Design Introduction

Today we live in a society where digital devices and related software tools enable us to capture videos. Thanks to advances in semiconductors, it has become affordable to own personal video equipment such as video and digital cameras. People use video cameras to store and share their experiences, and blogs on the Internet to share their thoughts.

Because of popular interest in image acquisition and its storage, a unique set of problems in managing these images has been created. Users often need to selectively access images; therefore, they need tools that can consistently enable them to select, sort, and manage the images. In the past, image collections were small, and therefore easy to manage, but with the ever-growing library of images, it is becoming increasingly difficult to classify and locate these images. To solve this problem, you need a tool that can classify and search images automatically. The image compression standard, MPEG-7, was developed based on this need. However, it is not useful for personal video users because digital camera manufacturers supply software that provides only the acquisition date and time stamp, and classifies images based on these parameters. Unfortunately, searching for an existing image without the time stamp is impossible based on this technique.

To solve this problem, we would like to propose a device called the passive digital camera featuring image-classification software. We call this device a passive digital camera because the digital camera, not the user, decides the shot time. The passive digital camera determines the acquisition time of the image by using various sensors. The camera recognizes the local conditions such as temperature, velocity, and tilting angle.

We have developed a software module for image analysis based on the image classification descriptor found in the MPEG-7 standard. The HW-IP module, which analyzes images, does so by considering the ability of the device and transferring it to a PC. The module also makes software control easy by loading the uCLinux real-time operating system (RTOS). Our digital camera is compact and lightweight. Being

a portable device, it is best if it can be implemented as a one-chip solution. Altera's FPGA and Nios® II embedded processor provide a huge advantage in this type of design.

Function Description

Our Personal Black Box (PB[2)] design collects image data that is beyond the user's cognition capability. We tried to use a grouping algorithm, which sorts collected images automatically, but we had to discard this approach because it needed a more powerful computing device. Instead, we tried to perform image classification in hardware. To enable this methodology, PB[2] would include a global positioning system (GPS) module in which it stores location information for individual images. We had to discard this approach also, and try a different method.

Using an acceleration sensor and an infrared ray sensor, we thought that we could fix the shaking noise in the image acquisition. This method worked partially because it enabled us to determine how long it took to acquire images. The image module in our design features more than a 300-Kbyte pixel capability and can control 640 x 480-size 24-bit color images. For this project, however, we used 320 x 240 image sizes.

Our PB[2] supports 1 Gbyte or more of flash memory and an SD card interface, using serial peripheral interface (SPI). We have also implemented the FAT32 file system. Because the PB[2] operates in real-time, it offers enhanced reliability in image acquisition timing. To achieve this reliability, we developed two software modules: the first module features single-threaded firmware, and the second features a multi-threaded program on µC/OS-II. These modules work identically.

With our power management circuitry we thought we could guarantee up to 16 hours of operation. (however, we have not been able to meet this expectation). Our product was implemented on the UP3 development board, but power management needs its own full custom design. On the UP3 development board the 1.2-V 900-mAh Ni-MH x 5 starts at 6.6 V, and an hour later it falls to 5.4 V and disables the UP3 board. We suspect this may be due to the power consumption of the LCD backlight.

Performance Parameters

The image control module uses a probability algorithm, the analysis of which leads to the dispersion comparison of images.

To refrain from acquiring unnecessary images, the module compares the head of the most recent two images. It then extracts the sample from the center of the image and compares it with the other image. To make the operation more efficient, it extracts the minimum sample for a comparison. See Figure 1.

Figure 1. Distribution of Each Image

The statistical tests were carried out on the PC as shown in Figure 2. We then used the Nios II processor to perform the tests and found that our operation efficiency was equal to the PC environment by a significant level (0.10).

Figure 2. Testing of Statistical Hypothesis

$$F = \frac{S_1^2/\sigma_1^2}{S_2^2/\sigma_2^2} = \frac{\frac{1}{n_1-1}\sum_{i=1}^{n_1}(X_i - \overline{X})^2/\sigma_1^2}{\frac{1}{n_2-1}\sum_{i=1}^{n_2}(Y_i - \overline{Y})^2/\sigma_2^2} \sim F(n_1-1, n_2-1)$$

Testing of statistical hypothesis :

Null hypothesis $H_0 : \dfrac{\sigma_1^2}{\sigma_2^2} = 1$ and

Testing statistics : $F = \dfrac{S_1^2}{S_2^2}$

Alternative hypothesis $H_1 = \sigma_1^2/\sigma_2^2 > 1$

Significant level : 0.10

Critical region : $F \geq F(n_1-1, n_2-1; \alpha)$

Design Architecture

While developing our design, we found that the MPEG-7 standard implementation posed many constraints that were too big to handle. We had a very small-capacity FPGA device, and we needed to integrate several peripherals. Hardware acceleration was almost impossible to achieve under these conditions. Also, we needed to develop many device control software routines, which required a simpler algorithm. When we have a more powerful FPGA available to us, we can reconsider hardware acceleration.

The image matching algorithm flowchart is shown below in Figure 3. We contrasted and compared two images based on a randomly chosen pixel's distribution ratio. If the ratio is over a specific limit, we deem the two images to be different, and then store the new image.

Altera Corporation **17**

Figure 3. Testing Algorithm Flow Chart
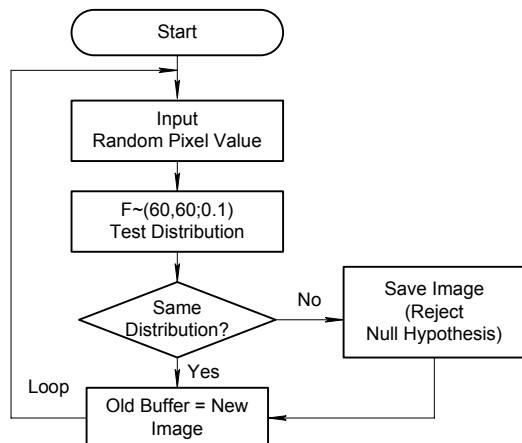
Hardware Design

We used an Altera® FPGA in our system design, where it functions as the main controller, handles all sensor functions, and manages all peripherals. The sensor module determines the image acquisition time. The peripherals in our design consist of a mass storage device and a battery module. Figure 4 shows the PB² hardware block diagram, including logic element (LE) usage. Later, we plan to add a battery module to the design.

We designed the image analysis function using the Nios II processor. The image analysis function's performance can be enhanced with statistical data analysis software such as the MATLAB software; or you can use a generic C program module.

Figure 4. Hardware Block Diagram
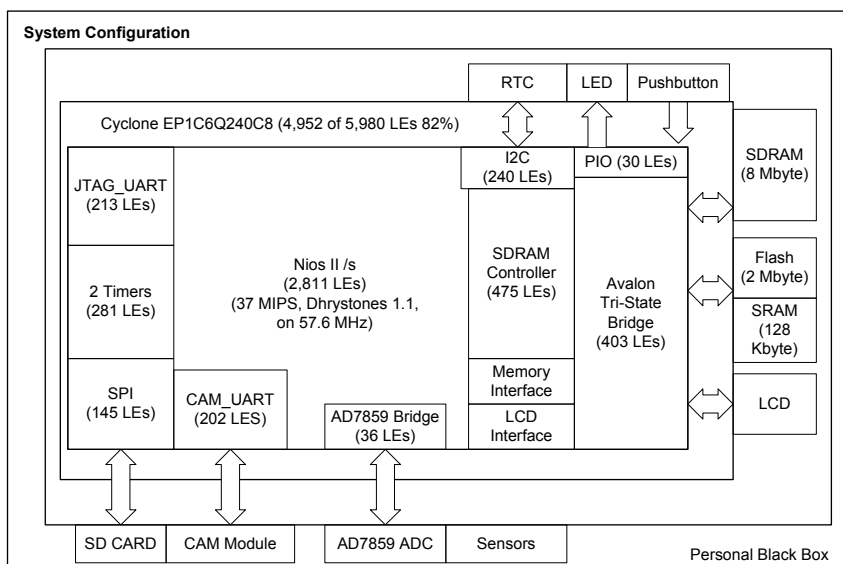
Software Design

Figure 5 shows the software flow chart for image acquisition, excluding the image processing. Each sensor helps determine the image acquisition time. The system uses a heat sensor, speed sensor, and tilt sensor. The image processing function starts before image storage and is determined by the available power and storage parameters. The image processing module comprises two parts: an image description extraction and classification of captured images. We use a PC to perform image classification. Therefore, each processing module task can be performed independently of the other.

Figure 5. Software Process Flowchart
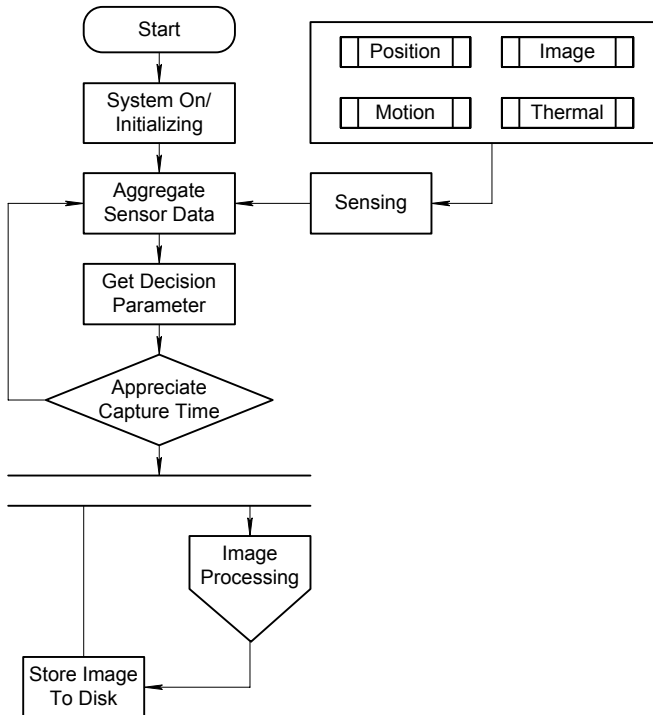
Figures 6 and 7 show the single-threaded and multi-threaded state diagrams, respectively.

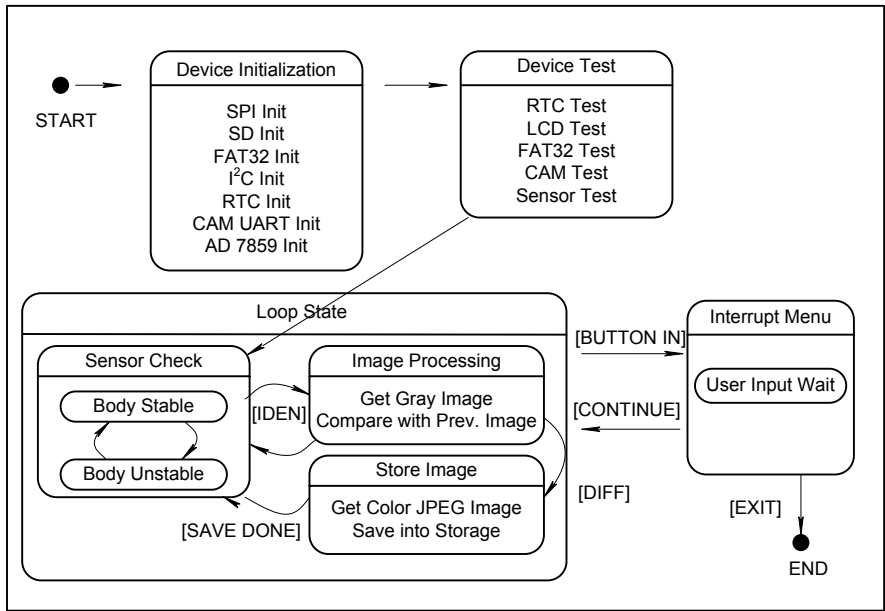*Figure 6. Single-Threaded Software State Diagram*

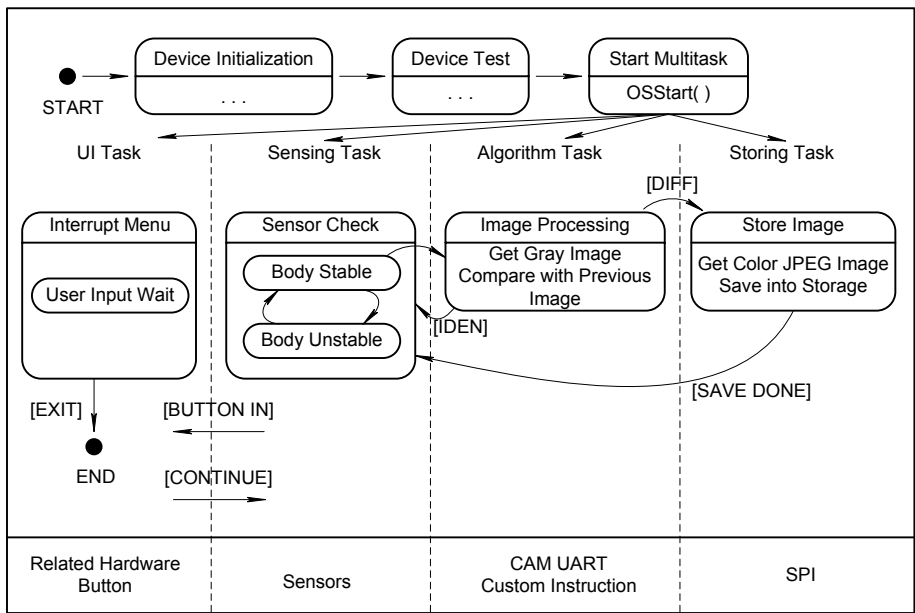*Figure 7. Multi-Threaded Software State Diagram*

Although the multi-threaded software module looks good, it was not much help in our design application. A slow image sensor and a fast processor for the image algorithm made it difficult to partition tasks. We feel, however, that the multi-threaded software module works well when all of the peripherals are reliable and the processing speed is uniform.

Design Methodology

Because our system clock was set at 57.6 MHz, it caused peripheral communication errors, especially with the UART baud rate. By keeping the board clock at 48 MHz, we managed to get 57.6 MHz using a phase-locked loop (PLL) adjustment. The main modules in our system were:

- 57.6 MHz Nios II/s processor with JTAG level 1 and a 14.4-MHz SPI bus for SD card control (with wide clock range support). We used SOPC Builder to calculate the clock latency, which was very helpful.

- I^2C Core (from opencores.org) for real-time clock (RTC) implementation. We found that Altera's OpenCore® Plus I^2C core required too many constraints for our system.

- 115.2-kbps UART for CAM module communication. Because this was a critical module, we changed and tuned the system clock. The CAM integrated circuit (IC) supports SPI mode for up to 3 MHz, but had some restrictions when used as a UART-type module. We could have selected the raw CAM module without using the UART interface, but at 115.2 kbps it was too slow to obtain the image.

- SDRAM memory controller. This module used a 57.6-MHz clock, but needed some phase-shift adjustments. We set this shift to -82.0 degrees, which was determined by board-specific characteristics.

- SRAM and CFI. This module is a tri-state bridge between memories.

- AD7859 user logic bridge. The AD7859 has control and data pins. We made AD7859 a bridge component using the software's new component feature. Because we had good knowledge of the Avalon bus interface, connecting peripherals was easy.

- Keypad and LED parallel (I/O) (PIO). This module comprised a 16 x 2-character LCD interface, buttons, and LEDs.

The SOPC Builder settings are shown in Figure 8.

Figure 8. SOPC Builder Settings
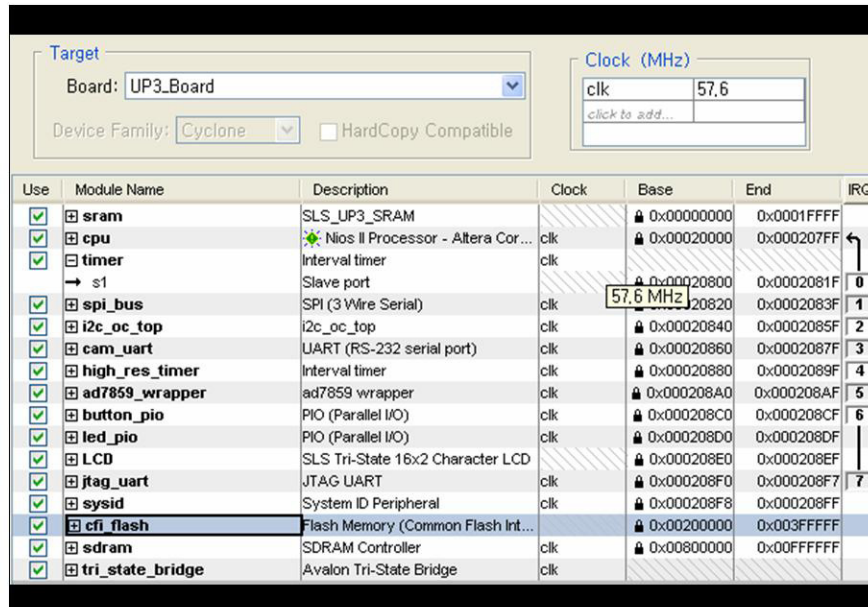
Using SOPC Builder, we designed a general microcontroller like AVR or ARM. This approach saved us additional hardware design work. Then we developed device drivers. We found that the Nios II Hardware Abstraction Layer (HAL) is good enough for normal operation, but we needed to customize it for better performance. We found some compilation problems as well. The memory alignment for structures was not documented.

Design Features

Our design had the following features.

Performance

Image processing requires many functions, and image processing resources are limited in mobile devices. Consequently, hardware/intellectual property (IP) implementation can help improve image processing efficiency.

Portability

The design must be flexible. Because the whole system is complex, you can get around this requirement by using a PC-based camera system to perform software conversion easily. We used the Nios II embedded processor for a hardware implementation of software functions, which is more convenient.

Low Power Consumption (Abandoned)

Our original design had low power consumption as one of the performance parameters, which we could not accomplish. Power consumption is determined by the operation time of the hardware module. Also, mobility considerations and related sensor operations have to be taken into account to create a low-power design.

Integration

The image processing unit put a strain on integration. From image acquisition to sorting to storage, we needed to have the hardware and software modules working together in the design. Thanks to the Nios II processor, we managed to accomplish that in our design.

Conclusion

Using Altera's system-on-a-programmable-chip (SOPC) solution, we learned a new way to solve system design problems. By employing an SOPC design in our system, we learned its advantages and disadvantages. Because SOPC designs use multiple IP modules to optimize the hardware design, this technique allowed us to simplify the system revision and debug process. This approach also allowed us to simultaneously design software and hardware modules.

In this design contest, we acquired hands-on experience and were able to use some excellent hardware development tools. The Quartus® II and SOPC Builder software made it easy for us to modify and change hardware, depending on the application. For example, we could easily add and change the SPI peripherals that are needed by the SD card control and transmission modules. We also feel that taking this design approach is economical—you do not need to buy additional hardware; you just change the Nios II soft core configuration. You can easily build new functions by adding related hardware based on the changed Nios II processor.

However, we do feel that the Nios II software development tools can be further improved. For example, the coding, compiling, and debugging tools are separate, making it difficult for the designer to work efficiently. Integrating these tools would greatly improve the design efficiency.

Using the Nios II development kit, we found it very easy to connect the custom logic components. Additionally, using the tool, we could easily tune several clocks in the system. However, we found it difficult to find a matched system clock. We think it would be a good idea to provide a table of several matched clocks. The Avalon bus interface was simple to understand. The UP3 Education Kit was great. We could do many things on UP3 board. But we hungered for a small LE. – *Jiwon, Kim*

We found that using the Nios II development kit in our design made it easy for us to develop our design with greater efficiency. – *Doehoon, Kim*

We had planned a bigger design at first, but were not able to implement it given time and resource constraints. We hope to do better next time - *SeungChul, Shin*

Second Prize

Nios II Processor-Based Hardware/Software Co-Design of the JPEG2000 Standard

Institution: **University of New South Wales**

Participants: **Mike Dyer, Amit Kumar Gupta, and Natalie Galin**

Design Introduction

JPEG2000 is a recently standardized image compression algorithm that provides significant enhancements over the existing JPEG standard. JPEG2000 differs from widely used compression standards in that it relies on discrete wavelet transform (DWT) and uses embedded bit plane coding of the wavelet coefficients [1]. Due to the bit-oriented processing techniques used in the standard, full implementation via software is inefficient, making embedded processing slow on standard microprocessors. Possible applications, such as scanners and printers, require a reasonable processing speed, which may be difficult to achieve using existing embedded processors. On the other hand, a full hardware implementation may not utilize the flexibility available in the standard. To improve the speed of the JPEG2000 algorithm while maintaining flexibility, we investigate the use of a co-design approach, using hardware acceleration for the bit-oriented and digital signal processing (DSP) tasks while leaving packet formation, code-stream formatting, and manipulation to software.

The Nios® II processor provides an ideal platform for implementing a co-design solution. The customizable arithmetic logic unit (ALU) allows for the addition of DSP-style instructions, which will improve the wavelet transform speed and code size. By adding custom peripherals to the system, the bit-oriented functions can be moved outside of the software into dedicated hardware. The provided real-time operating system (RTOS) (µC/OS-II) allows for parallel processing using multiple custom peripherals.

A software implementation of JPEG2000, called Kakadu [2], is used as the implementation framework and baseline for our design. Our proposed design adds the following features to Kakadu: multithreading with RTOS, custom instructions, and custom peripherals.

Function Description

Our system is a JPEG2000 encoder based on a Kakadu software framework. Fully compliant with Part 1 of the JPEG2000 standard, the main features of the system are:

■ Lossless and lossy compression

■ Region of interest (ROI) coding

■ Compression of color and gray-scale images
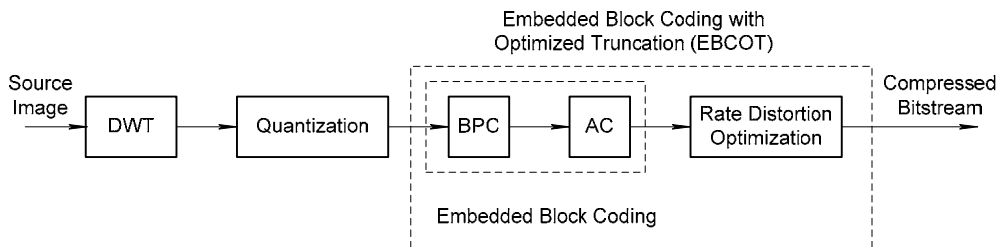
Figure 1. JPEG2000 Encoding Flow

Figure 1 illustrates the compression system of the JPEG2000 algorithm. The image is compressed in the following steps:

1. Image samples are separated into color components (if any).

2. Image color components are optionally decomposed into rectangular tiles, with each tile to be compressed independently.

3. DWT is used to decompose each tile into four frequency subbands. JPEG2000-Part 1 specifies two wavelet kernels for lossy and lossless compression, 9/7 and 5/3 wavelet kernels respectively.

4. The output from the wavelet transform is quantized and separated into rectangular `code-blocks', to be processed by EBCOT unit.

5. Each code-block is processed independently by the block coder (BC). The BC may be subdivided into bit-plane coder (BPC) and arithmetic coder (AC) modules. The BPC encodes a code-block in bit-plane by bit-plane order generating context-data (C x D) pairs. C x D pairs are then encoded by the AC module to generate the compressed bitstream.

6. Rate-distortion optimization selects optimal contribution of a code-block to the compressed bitstream for a given target bit rate such that the reconstructed image has minimum distortion. Kakadu uses the post compression rate distortion (PCRD) optimization algorithm [3].

7. Markers are added to the output bitstream to increase error resilience and packed into the JPEG2000 compressed bitstream.

The DWT and BC are two most resource intensive components of JPEG2000. A detailed study and analysis is performed to determine the strategy to best partition JPEG2000 into software and hardware components to optimize the compression stream using the rich feature set of the Nios II processor. We made the following changes to Kakadu:

■ Custom instructions used to implement DWT

■ Implementation of EBCOT in hardware, with BPC, AC, and the distortion estimation module implemented as hardware peripherals

■ RTOS (µC/OS-II) used to instantiate multiple block-coders to increase throughput

Performance Parameters

Table 1 presents the test environment used for comparison between the baseline and the proposed Kakadu implementation. It is to be noted that modules (DWT custom instructions, BPC, and AC) implemented in hardware are bit-exact with respect to the baseline, and thus do not alter the output bitstream. However, the hardware version of the module Distortion Estimation is not, and experimental results show that this change results in an average 0.02-dB PSNR difference between the baseline implementation and our proposed design when compressed for a given target rate.

Table 1. Test Environment Parameters

Property	Value
Image	café (ISO test image)
Image dimensions	2,560 × 2,048
Image format	pgm; 8-bit samples
DWT kernel	CDF 9/7
DWT levels	5
Block coder mode	Normal
Code-block size	64×64

Profile Results

The profile results for a purely software implementation of Kakadu is presented in Figure 2. From the profile, we note that block coding accounts for 103.02 of the total 167.08 seconds (64.01%) used to compress the cafe test image. DWT, on the other hand, accounts for 11.36 seconds (6.89%) of computation time.

Figure 2. Kakadu Profiling Results

```
Flat profile:
 Each sample counts as 0.001 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 28.80    48.13     48.13     1286    0.04     0.08   encoder()
 17.23    76.92     28.79     9420    0.00     0.00   encode_cleanup_pass()
 15.62   103.02     26.10     8134    0.00     0.00   encode_mag_ref_pass()
  3.89   109.52      6.51     4960    0.00     0.00   perform_vertical_lifting_step()
  3.54   115.44      5.92                             __floatsidf()
  3.48   121.25      5.82     2560    0.00     0.00   transfer_bytes()
  3.12   126.47      5.22                             __pack_d()
  2.90   131.32      4.85     4960    0.00     0.02   horizontal_analysis()
  2.89   136.16      4.83      122    0.04     0.92   encode_row_of_blocks()
  2.69   140.64      4.49                             __unpack_d()
  2.47   144.77      4.13     4960    0.00     0.00   push(kdu_line_buf&)
  0.00   167.08      0.00        1    0.00   133.29   main()
index % time    self  children    called     name
               48.13    56.70    1286/1286        encode_row_of_blocks()
[10]    62.7   48.13    56.70    1286             encode()
               28.79     0.00    9420/9420        encode_cleanup_pass()
               26.10     0.00    8134/8134        encode_mag_ref_pass()
                1.51     0.00    1286/1286        find_convex_hull()
                0.21     0.00   23116/25688       find_truncation_point()
                0.06     0.03    1286/1286        terminate(bool)
                0.00     0.00    1286/1286        start(unsigned char*, bool)
                0.00     0.00       2/2           set_max_bytes(int, bool)
                0.00     0.00       1/1           set_max_contexts(int)
                0.00     0.00       1/1           set_max_passes(int, bool)
```

Block Encode Time

On average, the hardware implementation of the BC (BPC combined with an AC), will take:

$$T_{BPC} = \frac{CTX_{blk}}{CTX_{cyc}} \cdot \frac{1}{F_{clk}}$$

Where CTX_{blk} is the average number of C x D pairs produced per block, CTX_{cyc} is the average number of C x D pairs produced per cycle and F_{clk} is the system clock frequency. For a system running at 50 MHz and processing 64 sample code-blocks, the average code-block processing time is:

$$T_{BPC} = \frac{23 \times 10^6}{1.1} \cdot \frac{1}{50 \times 10^6} = 4.182 \times 10^{-4} \text{ sec}$$

For each code-block, the internal code-block RAM must be loaded via direct memory access (DMA). This time will accumulate in systems that use multiple BCs.

$$T_{DMA} = \frac{N_s W_s}{W_b F_{clk}}$$

N_s is the number of samples in the code-block, W_s is the width of the sample in bits, W_b is the width of the bus in bits, and F_{clk} is the system clock frequency. This equation assumes that the DMA has exclusive access to the bus, as it will in our system. Using 64 sample code-blocks, where each sample is 16 bits, the system bus is 32 bits and the 50-MHz clock gives:

$$T_{DMA} = \frac{64 \times 64 \times 16}{32 \times 50 \times 10^6} = 4.096 \times 10^{-6} \text{ sec}$$

The average time taken to code a code-block in a system is approximately:

$$T_{blk,avg} = \frac{1}{N}(T_{DMA} + T_{BPC})$$

It should be noted that this equation is only accurate while the utilization of the data bus is low. When the time spent performing DMA transfers is greater than the time required to code a single code-block, the bus will become the limiting factor. This would indicate that the number of BCs should be

$N \leq \left\lfloor \dfrac{T_{BPC}}{T_{DMA}} \right\rfloor = 10$. Proper simulation is invaluable when determining the true value of N.

Because block coding accounts for 103.02 of the total 167.08 seconds used to compress the cafe test image (Table 2), we expect that the implementation of block coding in hardware will give the highest performance improvement. We also note that the BC must process 1,286 blocks. Table 2 shows the average block coding time when using N parallel hardware BCs, and the time taken to code 1,286 blocks, while Table 3 shows the speed-up factor using multiple BCs.

Table 2. Average Block Coding Times for Parallel BCs

N	1	2	3	4
T_{blk,avg}	4.5916×10^{-4}	2.2958×10^{-4}	1.5305×10^{-4}	1.1479×10^{-4}
T_{blk,total}	0.59	0.30	0.20	0.15

Table 3. Speed-Up Achievable With Multiple BCs

N	1	2	3	4
Speedup	2.58	2.59	2.60	2.60

DWT Flow

Amdahl's Law provides us with an estimate of the speed-up achieved from an improvement to a computation that affects a proportion P of that computation where the improvement has a speedup of S. (For example, if an improvement can speed up 30% of the computation, P will be 0.3; if the improvement makes the portion affected twice as fast, S will be 2.) Amdahl's law states that the overall speed-up of applying the improvement will be:

$$speed - up = \frac{1}{(1-P) + \dfrac{P}{S}}$$

From the profiling analysis performed, we estimate that the DWT processing consumes approximate 6.78% (P=0.068) of the total image compression time. If we achieved a 5/2 improvement in the lifting step, by applying Amdahl's Law we estimate an approximately 1.0425 (4.3%) improvement in the processing speed overall.

Design Architecture

The system structure is illustrated by Figure 3.

Figure 3. JPEG2000 Co-Design Configuration on Altera® EP1S40 FPGA
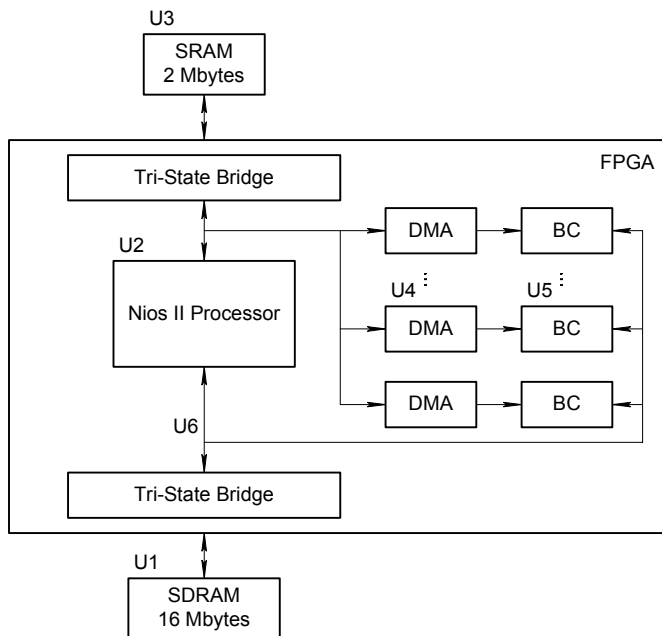
U1: 16 Mbytes of SDRAM containing the modified Kakadu program and image to be compressed

U2: Nios II processor

U3: 2 Mbytes of SRAM, which is loaded with code block data as it is created by Kakadu

U4: DMA controller configured to feed code block data at the rate of 16-bits per clock cycle to the BPC (U5)

U5: Block encoder hardware peripheral as described in detail later.

The number of DMAs and BPCs determined by the available bandwidth on the Avalon® bus. We chose to load the image directly onto the SDRAM so that the speed of our system is not limited by data transfer rates of external data I/Os. In this way, a fair comparison is made between the baseline and our proposed system. In the future, it will be possible to integrate the system with a fast bus fabric that will not saturate the speed advantages it provides.

Figure 4. BC Detailed System Configuration

Figure 4 shows a detailed block diagram for the BC hardware peripheral, where:

U6: 4,096-word RAM to buffer the code-block data (entire code-block, if necessary)

U7: Reorganizes sub-band samples in a bit-plane by bit-plane fashion to feed the BPC

U8: 16 x 4-bit FIFO buffer to store the data bits from U7

U9: 16 x 4-bit FIFO buffer to store the sign bits from U7

U10: BPC module

U11: 22 x 13-bit FIFO buffer used to store distortion estimation data

U12: AC module

U13: 16 x 17-bit FIFO buffer used to store compressed data from the AC module

U14: Finite state machine to control the information flow between the various components of the system

U15: Register file containing 32 x 32-bit registers, 16 for read and 16 for write, to interface between the BPC (U5) system and the control bus

U16 : Distortion estimation module

Design Methodology

The hardware/software co-design of the JPEG2000 followed these design steps:

1. Alteration to Kakadu to support multithreading.

2. Development of bit-accurate software to verify the functionality of the proposed hardware peripherals.

3. Implementation of hardware peripherals using hardware description languages (HDLs).

4. Use of the ModelSim® tool to verify the hardware peripherals' functionality. The testbench vectors are generated using the bit-accurate software.

5. Use of the LeonardoSpectrum™ tool to synthesize the hardware peripherals.

6. Use of the Quartus® II development suite to produce the layout and timing analysis of the hardware peripherals.

7. Post-synthesis simulation in the ModelSim software using the Quartus II timing results.

8. Load Kakadu software into the Nios II integrated development environment (IDE) and develop glue software to interface it to the hardware peripherals. Custom instructions were also added to Kakadu at this point.

9. Build and load of the Quartus II project onto the FPGA.

To improve the performance of the DWT function in Kakadu, we augmented the instruction set in the Nios II processor with two new custom instructions. The block diagram for the custom instructions is shown in Figure 5.

Figure 5. Nios II Arithmetic Logic Unit (ALU) [4] & Custom Logic Layout to Perform a Lifting Step
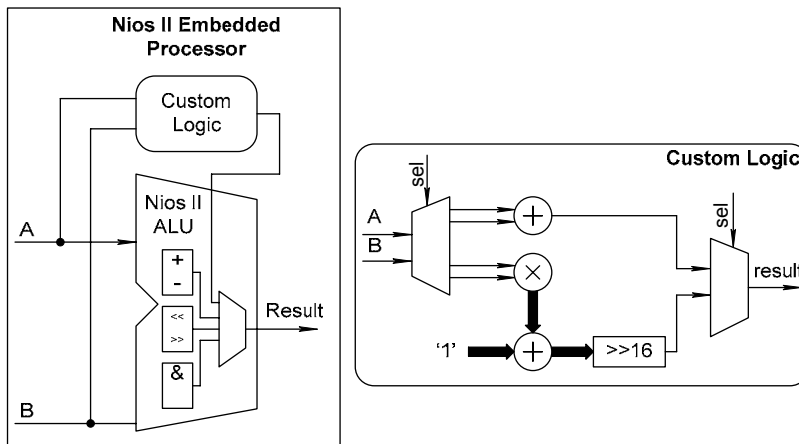
Design Features

This section describes the project's design features.

DWT Custom Instructions

As can be seen from Figure 6 of the state machine for the CDF 9/7 Lifting DWT implementation, the four lifting steps are very similar; the only difference is the value of the multiplier coefficient. Therefore, it was clear that the DWT would best be implemented using the Nios II processor's ability to add custom logic to its ALU. To perform a lifting step, two custom instructions were needed: one to augment two 16-bit samples into one, and one to perform the lifting step, shown in Figure 6.

Upon compilation, the number of assembly instructions to perform the lifting step decreased from five (in the pure C implementation) to two (using Nios II custom instructions).

Figure 6. 9/7 DWT Lifting State Machine
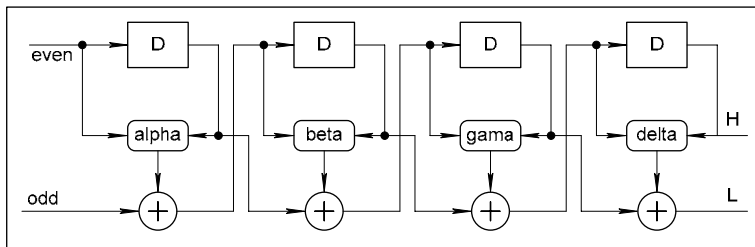
Multi-Threading of Kakadu

The Kakadu software library was originally written as a single-threaded library. While this is adequate for single CPU systems, it makes dispatching multiple code-blocks to multiple BCs quite challenging. To utilize the availability of multiple BCs, the Kakadu library was modified to support threads. The use of threads requires using the µC/OS-II real-time operating system, a useful feature integrated into the Nios II IDE.

When enough data has been generated by the DWT, a row of code-blocks is dispatched via the function call 'encode_row_of_blocks()' [2]. At this point, the library was modified to support threads. The library was supplied with a thread pool, where each thread is capable of encoding a single code-block. Each thread is attached to a single BC hardware resource and is responsible for initiating the DMA transfer and for collecting the compressed data and rate distortion information. When a thread completes, it is restarted with a new code-block until the row of code-blocks is exhausted.

Hardware Peripheral

The main reason for the high computational cost of JPEG2000 on a general-purpose processor is due to the bit-oriented processing during block coding. This motivates the use of a custom hardware accelerator for the BC. A major feature of the Nios II SOPC Builder is that it supports the creation and utilization of custom hardware peripherals. Thus, it presents an ideal platform for our design.

The BC peripheral consists of two Avalon slave interfaces and four sub-modules. The first slave interface is used to receive block samples via DMA to remove the processor overhead involved in transferring sample data into the BC. The second slave interface accesses a register file, and is used to control the peripheral as well as access status information and compressed data. The four sub-modules perform bit-plane reorganization, bit-plane coding, distortion estimation, and AC. They are outlined below.

Bit-Plane Reorganization

As the BPC operates on bit planes, sample data must be converted to this format before being sent to the BPC. The bit-plane reorganizer scans through stripe columns and forms a 4-bit word that contains the bit value of the four samples in that stripe column for the current bit plane. These are then stored in a FIFO buffer ready for sending to the BPC. This system must operate at twice the rate of the BPC to ensure data is always available.

BPC Module

■ Generic: Handles all modes of BPC operation for all nominal code-block dimensions

■ High processing throughput: Generates an average of 1.1 C x D/clock-cycle (Existing generic BC architectures only generate 0.7CxD/clock-cycle) [5]

■ Based on two-state memory system [6]

■ Uses proposed optimal two sub-bank memory architecture for internal memories [7]

■ Minimum memory cost (16 Kbits dual port RAM) currently reported for a generic BPC architecture

■ Efficient intermediate buffer: The BPC has a varying C x Ds per clock-cycle output (anywhere between 0 to 10 C x Ds per clock-cycle) depending on image statistics. Since our AC module has a maximum input rate of two C x Ds per clock-cycle, we use an intermediate buffer [8] to integrate the BPC and AC module. The buffer is optimized for its hardware cost versus throughput performance using real image statistics.

AC Module

The BPC is capable of producing multiple C x D pairs per clock cycle. Although an AC capable of coding a single pair per cycle can have enough throughput to cope with the C x D rate of the BPC, this would require the AC to have a separate, faster clock domain. To mitigate this complexity, an AC was designed that could consume two C x D pairs per cycle, while operating at the same frequency as the BPC [9].

Distortion Estimation

The PCRD algorithm requires estimated distortion values associated with each truncation point (coding passes) [9]. The distortion estimation for a truncation point depends on the sample values and their distribution among coding passes, a factor that cannot be simply pre-calculated.

We designed a novel hardware module for distortion estimation that uses one fractional bit (in comparison to five fractional bits as used by Kakadu). Our simulation results show that this results in average 0.02-dB PSNR degradation for a given target rate (in comparison to Kakadu's reported architectures, which achieve an average 0.3-0.7 dB PSNR degradation [10]).

Implementation Results

Figures 7 and 8 show simulation waveforms of our system. The following points in time are of particular interest:

1. 82,100 ns: Sample DMA finishes.

2. 82,200 ns: Register file access asserts operating parameters and starts system.

3. 82,400 ns: Bit plane reorganization started.

4. 132,550 ns: Register file access checking status and reading compressed data bytes.

5. 132,500 ns: Onwards demonstrates normal operation with BPC providing contexts to the AC.

Figure 7. The Simulation Flow Showing BC-Avalon Bus Transfer

Conclusion

The profile of the pure software implementation justifies our decision to provide dedicated hardware for the BPC and DWT, as these functions are at the top of the profile list. The Nios II processor provides an ideal platform for integrating dedicated hardware, as it provides the ability to include both custom instructions and peripherals. Our implementation results show that the inclusion of a DWT step instruction will improve the speed by a factor of 1.04, while the inclusion of dedicated block coding hardware can improve speed by a factor of 2.6. It is interesting to note that providing multiple BCs in parallel provides only minimal improvement over a single hardware BC as a consequence of Amdahl's law.

Figure 8. Simulation Flow Showing DMA Transfer in Place between Bit-Plane Reorganizer & BPC
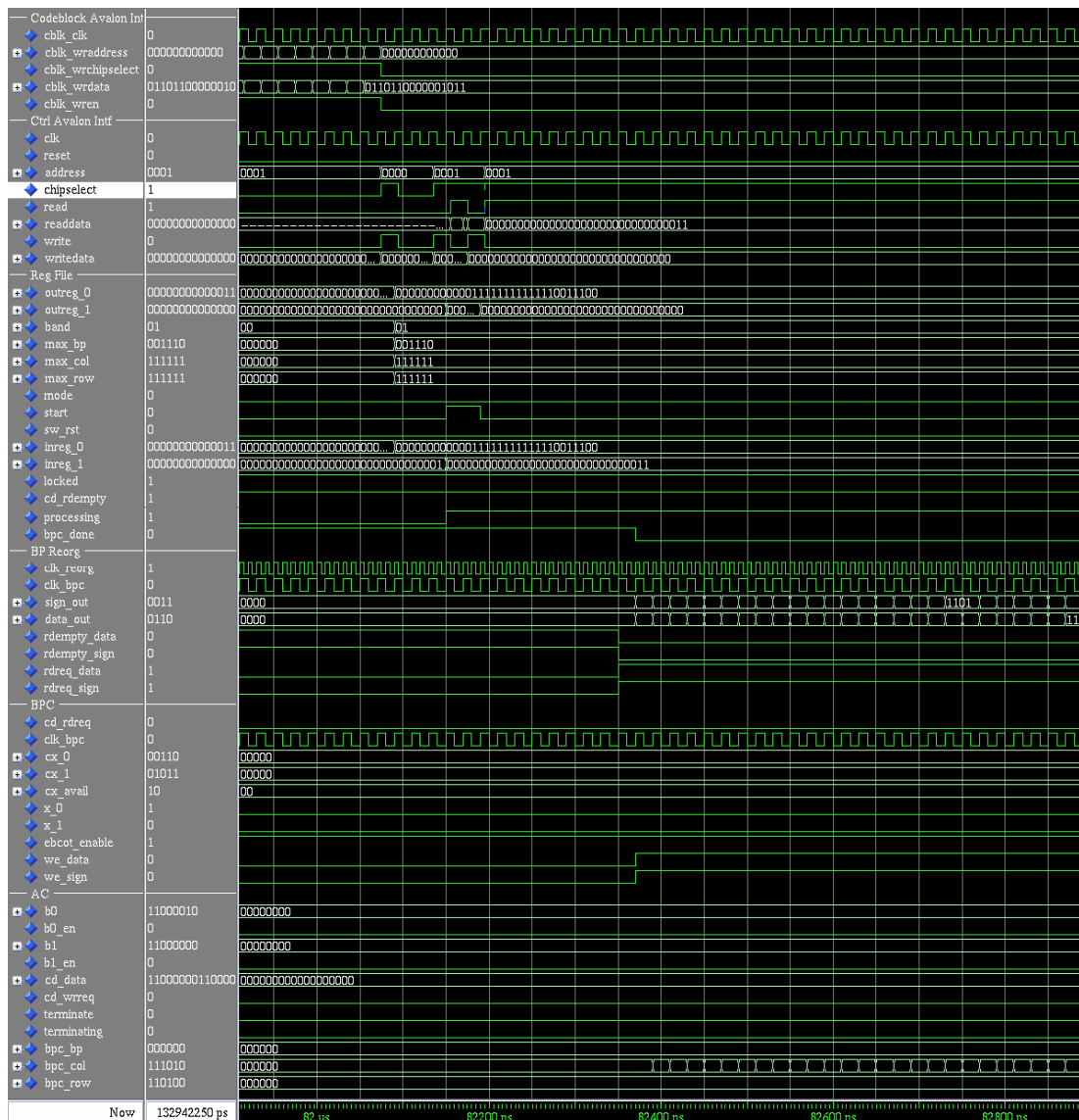
References

[1]   "JPEG2000 part i final committee draft version 1.0 ISO/IEC JTC1/SC29/WG1N1646R," March 2000.

[2]   D. Taubman, "Kakadu software- a comprehensive framework for JPEG2000." http://www.kakadusoftware.com/.

[3]   D. S. Taubman and M. W. Marcellin, *JPEG2000 Image Compression Fundamentals, Standards and Practice*. Norwell, Massachusetts 02061 USA: Kluwer Academic Publishers, 2002.

[4]   "Nios documentation." http://www.altera.com/literature/manual/mnlniossft.pdf.

[5]   A. K. Gupta, S. Nooshabadi, and D. Taubman, "Concurrent symbol processing capable VLSI architecture for bit plane coder of JPEG2000," *IEICE Transactions on Information and Systems, Special Section on Recent Advances in Circuits and Systems*, vol. E88-D, pp. 1878 – 1884, 2005.

[6]   Y.-T. Hsiao, H.-D. Lin, K.-B. Lee, and C.-W. Jen, "High-speed memory-saving architecture for the embedded block coding in JPEG2000," *IEEE International Symposium on Circuits and Systems*, vol. 5, pp. V–133 – V–136, May 2002.

[7]   A. K. Gupta, S. Nooshabadi, and D. Taubman, "Optimal 2 sub-bank based memory architecture for bit plane encoder of JPEG2000," *IEEE International Conference of Circuits and Systems (ISCAS'05)*, 2005.

[8]   A. K. Gupta, S. Nooshabadi, and D. Taubman, "Efficient VLSI architecture for buffer used in EBCOT of JPEG2000 encoder," *IEEE International Conference of Circuits and Systems (ISCAS'05)*, 2005.

[9]   M. Dyer, D. Taubman, and S. Nooshabadi, "Improved throughput arithmetic coder for JPEG2000," *IEEE international conference on Image Processing (ICIP'04)*, 2004.

[10]   Y. Chang, H. Fang, C. Lian, and L. Chen, "Novel pre-compression rate distortion optimization algorithm of JPEG2000," *SPIE Proc. of Visual Communication and Image Processing*, vol. 5308, pp. 1353–1361, 2004.

Second Prize

Embedded Network MP3 Playing System

Institution: **Southern Taiwan University of Technology**

Participants: **Cai Suwei, Xiao Xingjie, Zhang Jiahao**

Instructor: **Dr. Wei Zhaohuang**

Design Introduction

We designed an embedded Netware MP3 player system that consolidates both software and hardware, and is based on system-on-a-programmable-chip (SOPC) design principles. The product finds use in the following applications.

Public Broadcasting System

Many public audio-broadcasting systems have transmitted audio signals that suffer from weak audio quality and complicated cabling, and are confined to individual or limited area broadcasts. In contrast, our system broadcasts MP3 audio via Ethernet to solve these problems. Our design not only improves the broadcasting quality, but also extends the life of the broadcasting device, reducing installation and maintenance costs for the entire system.

CD Audio Player

In music stores and supermarkets, customers are provided with preset audio CD players on which they may sample the music. Frequent customer usage can cause the CD players to malfunction. Additionally, the CD audio content must be updated manually. Using our design, you can download MP3 archived audio from a server into the system. This approach needs no mechanical operation and the MP3 server can update the audio data at any time.

Recently, the electronics industry has been shifting away from PC-centric devices to multi-functional Internet appliance (IA) applications, leading to a boom in rapid market development of multimedia and consumer electronics such as MP3, DVD players, TV game consoles, consumer electronics devices, and mobile phones. This trend complicates the system design and shortens the product development cycle. Therefore, using FPGAs has helped product designs become powerful, multi-featured, consistent, low

power, and highly integrated. Programmable logic devices greatly assist designers in planning and customizing the systems they want to build. FPGAs help keep costs down and reduce marketing time, important factors in the electronic industry. Also, programmable logic components play a key role in system integration.

Figure 1 shows the SOPC hardware platform used in our system design. The SOPC hardware features The Altera® 32-bit Nios® II processor, which is the intellectual property (IP) that can be embedded into many FPGA devices. This design approach enables the developer to build systems without worrying about development costs. Using the FPGA, we implemented a serial port, timer, boot ROM, Avalon® bus bridge, and PIO, which connects these interface devices.

Figure 1. Internal Architecture of SOPC Development Platform
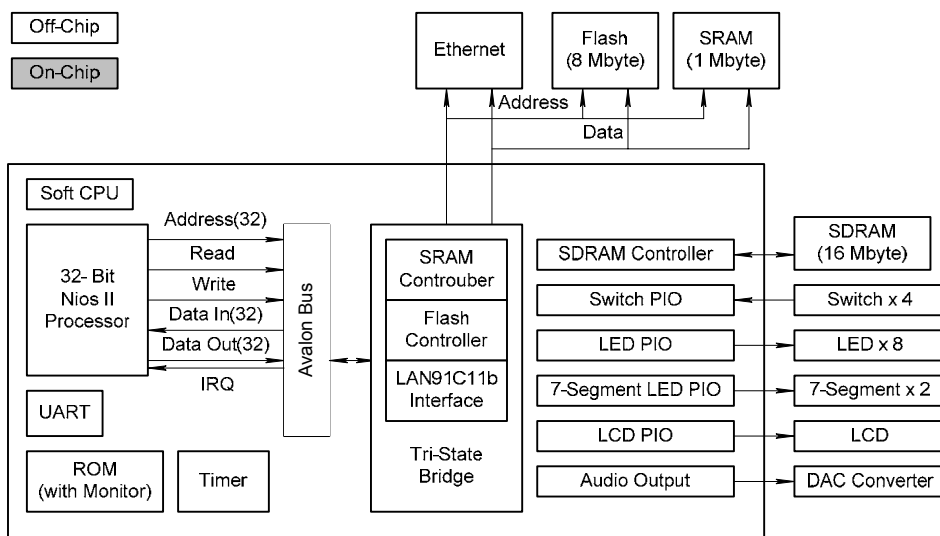
The Nios II CPU can be optimized in three ways:

■ For maximum system performance.

■ For minimum logic use.

■ For a mix of system performance and logic use.

Because the program machine codes for all these optimized CPUs are 100% compatible, designers can easily modify the CPU performance according to changes in system requirements. The 32-bit RISC embedded processor has been designed specifically for the FPGA architecture, and features a performance of greater than 200 MIPS (Dhrystones 1.1). Additionally, the development costs are low when you use an Altera FPGA, making it a good choice for consumer electronics applications that demand low price and mass production.

The Nios II processor continues to use the Avalon bus structure introduced by the first-generation Nios processor. This structure provides a set of pre-defined signal types, which can be used to connect more than 60 peripherals, including Ethernet, USB, and memory controllers. SOPC Builder and related tools, such as the Altera Quartus® II software, generate the Avalon bus structure logic automatically. The structure includes data channel reuse, address decoding, waiting period generation, dynamic bus size,

and interrupt assignments. Designers can use the SOPC Builder Import Guide to integrate their own IP modules with other peripherals into the Nios II project.

The Avalon bus structure provides flexible interconnection, simultaneously permitting multiple cores (CPU and accelerator) to communicate on dedicated channels while reading/writing data. This design scheme helps increase the volume of system data transmission. Therefore, the hardware accelerator often used in network communications to compute cyclic delay code can increase its performance two-fold, compared to software processing. For instance, using software, we need millions of clock cycles to process a 64-kilobyte data block. If we used Nios II custom instructions, this could be accomplished with hundreds of thousands of clock cycles. Using the hardware accelerator, it takes only tens of thousands of clock cycles.

Unlike the Nios processor, the Nios II processor does not restrict you to five custom instructions, and allow you to use unequal clock cycles. Furthermore, the Nios II processor supports a maximum of 256 user-defined instructions with fixed or variable frequency period operations. Designers can use these instructions to accelerate the program code and meet an application's strict timing requirements. Additionally, they can implement large and complex algorithms and call them as subroutines in the C language.

Altera provides the Quartus II software, a complete software development tool for the Nios II processor, which includes a Compiler, integrated development environment (IDE), JTAG debugger, and TCP/IP protocol stack:

■ *Nios II IDE*—The Nios II IDE is an Eclipse project that opens with the original code and provides a complete C/C+ software development kit, including a compiler, project manager, building tools, debugger, and flash programmer complying with Common Flash Interface (CFI). The IDE supports connection with target hardware via the JTAG port, and supports a connection between the Nios II instruction set simulation and Mentor Graphics' ModelSim hardware simulation tools.

■ *IP TCP/IP protocol stack*—A lightweight IP TCP/IP protocol stack provides a Berkeley socket application program interface (API), supporting IP, ICMP, UDP, TCP and RTT estimation, fast recovery, and fast re-transmission.

Function Description

The system comprises an MPEG-1 Layer III (MP3) archive server, MP3 decoder, and Ethernet connection. The design is implemented using software and hardware with an embedded SOPC platform, and uses the embedded uClinux as its real-time operating system (RTOS). The system design principles and detailed description are described in the following sections.

MPEG-1 Layer III Coder Architecture

The MP3 coding principle should be understood before creating the MP3 decoder. Its coding architecture is shown in Figure 2, which includes a psychoacoustics model, hybrid filter bank, and quantization/Huffman coding. Quantization and distortion-free code are mainly used for the rate and distortion control loop in the MP3 architecture.

Taking monophonic data as an example, one MP3 frame contains 1,152 sound samples (a frame equals two granules, a granule contains 576 sound samples); each sample is 16-bits of data. Using the filter bank analysis, the originally entered 16-bit PCM audio is transformed into 32 sub-band signals with the same bandwidth. Then, each sub-band signal is subdivided into 18 hypo-band signals using the modified discrete cosine transform (MDCT). Next, bit assignment and quantization coding are made for each sub-band signal according to the signal-to-mask ratio (SMR) provided by the psychoacoustic model II. At last, this coded data emerges in bit serial mode defined by MPEG-1.

MP3 Decoder Architecture

Figure 3 shows the MP3 decoding architecture. The MP3 bitstream uses a demultiplexer to perform the header and side information decoding. It then implements Huffman Decoding, a descaler, and a dequantizer using the header and side information. Next, it performs the inverse modified discrete cosine transform (IMDCT) using dynamic windows and outputs PCM data through the filter group.
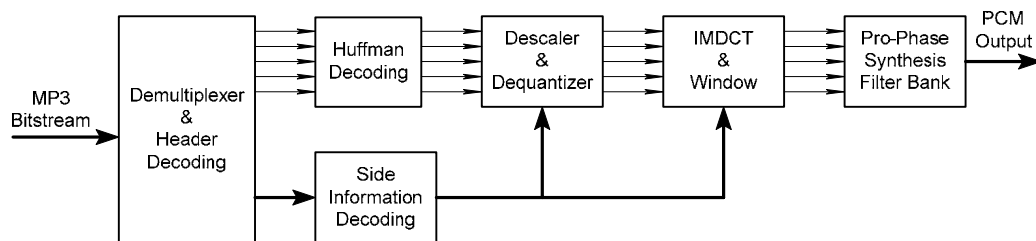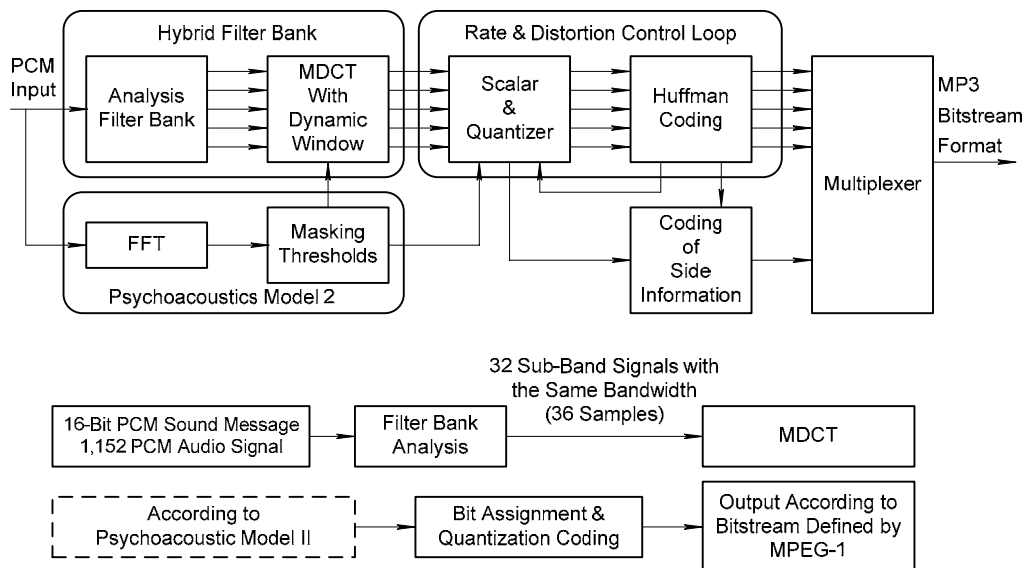
Figure 2. MP3 Decoding Data Stream

Figure 3. MPEG-1 Layer III Coding Architecture

MP3 Archive Description

Related information about archive decoding should be obtained before implementing the MP3 decoding. This data will be used in the corresponding encoding. You need to know the data's bitstream format, frame format, header file format, and side information format.

Bitstream Format

The MP3 bitstream is composed of many frames, with the slot size as its basic unit. The MP3 standard defines a slot size as 1-byte of data, and therefore, the whole bitstream is made up of slots of integer numbers. Every frame contains a great deal of decoding information. The starting point of each frame is a set of sync words, and the ending point is before the sync word of the next frame. All frame sizes are 1,152 samples, although their frame length may vary. The length varies because Huffman decoding is

used during MP3 decoding, which results in a non-identical decoding length and a variable frame length.

Frame Format

The MP3 frame is composed of four parts: the header, the cyclic redundancy code (CRC), side information, and the main data (see Figure 4). The header contains sync words and other important system information used to detect the new frame's starting point, such as layer, bit rate, sampling frequency, and number of channels. The CRC is used in error correction and is 16-bits of data. The side information includes information needed for main data decoding, and the main data section contains data that is needed during Huffman decoding and scale factor rebuilding.

Figure 4. MP3 Frame Format
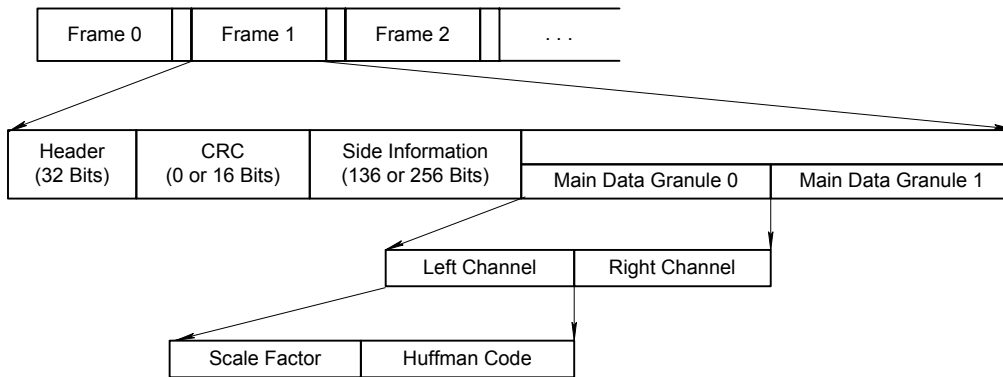
Header Format

The first 32 bits of a frame hold the header file information, which is divided into 13 fields, and which records important frame information (see Figure 5).

Figure 5. MP3 Header Format
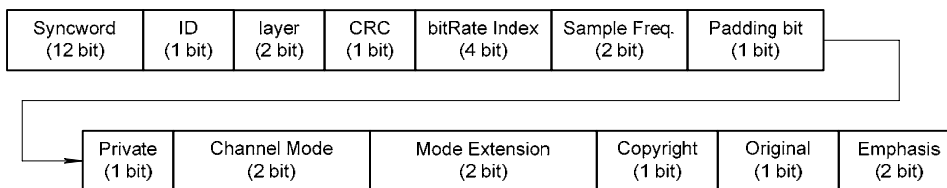
A sync word identifies the start of the frame and detects the frame length, i.e., the distance between the sync word and the next group of sync words (also referred to as a frame). Sync words comprise 12-word groups of 1s. However, ID information is used to tell the decoding end which algorithm needs to be adopted to decode the current bitstream. The decoding end should use this algorithm to decode while the ID field is indicated using a single bit. When the ID is '1', it indicates the adopted algorithm to be the MPEG-1 audio standard (ISO/IEC11172-3); when the ID is '0', it indicates the algorithm to be the MPEG-2 audio standard (ISO/IEC 13818-3). The layer field is shown in two-bit format, and indicates which layer of the audio standard is used. The protection bit indicates whether or not there is an added 16-bit CRC in the bitstream.

The output bit rate index indicates the bit rate of the bitstream with 4 bits. Layer 3 ranges from 32 kbps to 320 kbps. The MPEG-1 audio defines three kinds of sampling rates: 48, 44.1, and 32 kHz. Two bits

are used to indicate the sampling frequency in the header files, while another two bits are used to support four channel modes and indicate the channel mode of the bitstream in the header files. When the mode is set at "01" it is in the joint stereo channel mode. Therefore, the joint stereo processing step must be added to the decoding. Additionally, the two flags—copyright and original—indicate whether the MP3 archive owns copyright or is original, respectively.

Side Information Format

Side information records the information needed in audio data decoding. It is 17 bytes long in a single channel frame and 32 bytes in dual channel or stereo channel (see Figure 6). The starting index of the main data in the side information indicates the starting address of the main data. The Part2_3_length field indicates the length of the scale-factor data. The scalefac_compress[gr] table shows how many bits need to be read each time. Because transform coding is used in MP3, it must transform according to the window size. In addition to the long window used for a stable signal and the short window used for an unstable signal, there are two relay windows used for when it changes from a long window to a short window or vice versa. We use window_switch_flag to indicate which tables are being used. The advantage of using four different windows is that frequency resolution can be added to maintain good audio quality. The block_type field indicates which kind of window each granule uses. There are 32 Huffman tables in all, so table_select in the side information indicates which table is used to perform the Huffman decode.

Figure 6. MP3 Side information format

Single-Channel Mode 17 Bytes

Main_data_begin (9 Bits)	Private_bit (5 Bits)	Scfsi[ch][scfsi_band] (4 Bits)	Gr0 Side Information (59 Bits)	Gr1 Side Information (59 Bits)

Dual-Channel Mode 32 Bytes

Main_data_begin (9 Bits)	Private_bit (5 Bits)	Scfsi[ch][scfsi_band] (4 x 2 + 8 Bits)	Gr0 Side Information (59 x 2 = 118 Bits)	Gr1 Side Information (59 x 2 = 118 Bits)

MP3 Decode Operation

The MP3 decoding operation includes the Huffman decoder, dequantizer, reordering, IMDCT, and synthesis filter bank, which are described in the following sections.

Huffman Decoder & Dequantizer

To obtain high compression rates during MP3 decoding, the spectrum coefficient transformed via the MDCT is divided into 3 regions—the big_value region, the count1 region, and the rzero region from low frequency to high frequency values. Data needs to be recovered from each region according to different Huffman code tables during decompression.

Before entering data into the synthesis filter bank, the value after the Huffman decoding should be dequantized.

Long window:

$$xr_i = sign(is_i) * |is_i|^{\frac{4}{3}} * 2^{\frac{1}{4}(global\_gain[gr]-210)}$$

$$* 2^{-(scalefac\_multiplier*(scalefac\_l[gr][ch][sfb][window]+preflag[gr]*pretab[sfb]))}$$

Short window:

$$xr_i = sign(is_i) * |is_i|^{\frac{4}{3}} * 2^{\frac{1}{4}(global\_gain[gr]-210-8*subblock\_gain[window][gr])}$$

$$* 2^{-(scalefac\_multiplier*scalefac\_s[gr][ch][sfb][window])}$$

Where is_i indicates audio cable undequantized, and xr_i represents that of dequantized; sign() takes the positive signal; global_gain and preflag are obtained from side information.

Reordering

The MDCT adopts two block modes: the short window and the long window, which have different spectrum types, after the MDCT transform. For greater efficiency in Huffman decoding, the short-window spectrum is deployed prior to the Huffman decoding. This reordering step recovers the original sorting sequence.

IMDCT

The IMDCT operation is performed on the data frame as the standard unit. There are 576 points in the data of a frame, which are further divided into 32 sub-bands. Each sub-band has 18 points whose data are IMDCT-transformed into 36 points. Each frame thus transforms into 1,152 points, and this value is multiplied with appropriate window functions depending on the type of window. IMDCT definition is (2-1) mode where *N=36* in the long window and *N=12* in the short window.

$$X_i = \sum_{k=0}^{\frac{n}{2}-1} X_k \cos(\frac{n}{2\pi}(2i+1+\frac{n}{2})(2k+1)) \quad for \ i=0 \ to \ n-1$$

After the IMDCT transform, the synthesis filter bank data enters into the polyphase filter to synthesize the audio signal in PCM format. The synthesis actions in the polyphase filter are remove, IMDCT, and matrix-multiply. IMDCT transforms 32 samples to get a 64 V vector and takes 512 samples to the synthesis filter (W window) to form the W vector (512 elements). The 512 elements are placed into 16 groups of 32 elements, whose summation of the vectors is the final rebuilt audio signal in PCM format (see Figure 7).

Figure 7. Synthesis Filter Bank Flow

Preamble	SFD	DA	SA	LN	Data	PAD	FCS
7	7	2/6	2/6	2	0~1500	0~46	4
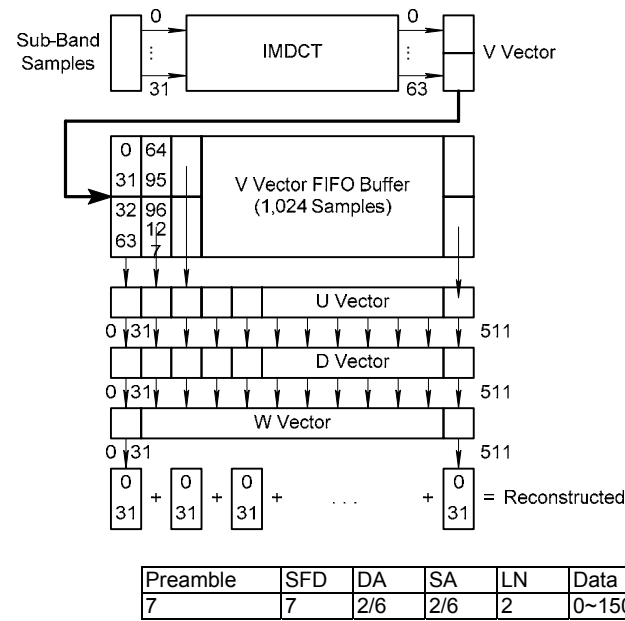
- Preamble: Synchronization function

- SFD: 10101011, starting byte

- DA, SA: Destination and starting address

- LN: Data length

- Data: At most 1500 bytes

- PAD: Ensure one frame has at least 64 bytes

- FCS: Error checking code

Ethernet Protocol

Ethernet uses a carrier sense multiple access/collision detection (CSMA/CD) for data transfer. It must ensure that no online signal transfer is made before the data transfer. The data transfer needs to be halted in case of conflict, and retried after some random delay.

Data on Ethernet is transferred in frames. The format is shown in the following section. Each frame has 64 bytes of data as the minimum frame length, and 1,518 bytes as the maximum frame length. PAD is used to fill the packets to 64 bytes in case no data exists or the data is less than 64 bytes.

The Ethernet addressing mode is set by the DA/SA in the packet, whose value could be two bytes (for local supervision) or six bytes (for global addressing). The first byte determines transfer to the individual address or the group address. If all values are '1', it is a multicast or a broadcast address.

Using these addressing modes, the MP3 broadcaster can transfer audio to specified receivers or multicast the data.

Data in the packet includes TCP/IP information. TCP/IP is the best protocol available today for use in Internet or Intranet applications. The protocol is briefly described as follows.

The network protocol is built layer by layer. Each layer is responsible for a certain network function. The TCP/IP four-layer network communications architecture includes the network application layer, transport layer, network layer, and data link layer. Programs implemented on the application layer are HTTP, telnet, e-mail FTP, etc; TCP and UDP are implemented on the transport layer; IP and ICMP are carried out on the network layer; the Ethernet driver and PPP protocols are implemented in the data link layer. Figure 8 shows the network protocol.

Figure 8. Network Protocol
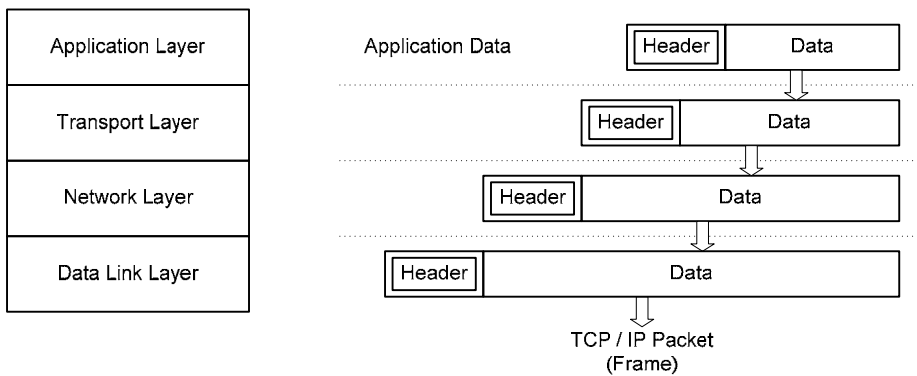
Single channel mode transfers the network data with TCP/IP, whose operation for "additional Header" in each layer is shown as follows.

The data architecture of each layer of the TCP/IP network communication is shown below. AP Data is added in the header of each layer from the TCP segment to the Ethernet frame, and forms network packet data. Figure 9 shows the data architecture.

Figure 9. Data Architecture

TCP Segment
Network Application Layer ---
AP Data = AP Layer + (Essence) Data

TCP Layer ---
TCPH = TCP Header
TCP Segment = TCPH + TCP Data
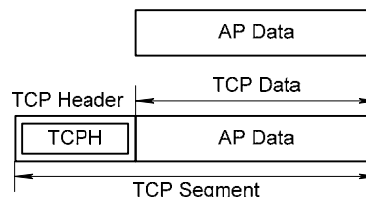
IP Datagram
IP Layer ---
IPH = IP Header
IP Datagram = IPH + IP Data
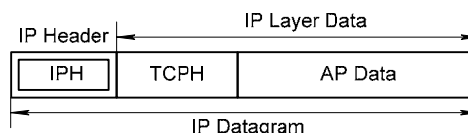
Ethernet Frame
Ethernet Layer ---
EH = Ethernet Header
Ethernet Frame = EH + Ethernet Layer Data
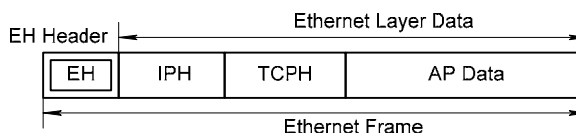
Embedded Operating System Analysis & Selection

An efficient RTOS needs to be supported by the system to enable the embedded products' multifunction features and shorter development time. In the past, embedded systems have implemented fewer functions and therefore did not need operating systems. However, as more and more consumer electronics devices integrate multiple functions, many embedded systems are more complex.

Among the many available RTOS for embedded systems, the µC/OS and uClinux are the most popular, offering excellent performance and open source code. µC/OS is suitable for small control systems because of its high efficiency, small size, and strong scalability. For example, the smallest core of the µC/OS can be compiled into a 2-Kbyte space. The uClinux, adapted from the standard Linux OS, is designed according to the feature of the embedded processor. With built-in network protocols, uClinux supports many file systems.

Comparison between the µC/OS & uClinux Embedded Operating Systems

The embedded operating system is the control center of the embedded system software and hardware resources. It organizes and manages multiple user needs. Task scheduling, file system support and system migration are general tasks in embedded operating system applications. This section compares how the µC/OS and uClinux RTOSs handle these tasks.

Task Scheduling

Task scheduling arranges the use of system resources (memory, I/O devices, and CPU). Task scheduling—also referred to as CPU scheduling—allocates the CPU for tasks in ready status based on two basic modes: preemptive and non-preemptive scheduling. In non-preemptive scheduling, a task is implemented once it is scheduled, unless it gives up the CPU time and enters into wait status, in which case the CPU is reallocated to other tasks. Preemptive scheduling identifies the current task, and is preempted to ready status once a task with higher priority exists in ready status or the running program has used up the specified time slice, in which case the CPU will be allocated to other tasks.

Being an RTOS, µC/OS adopts preemptive real-time multi-task core, i.e., the core always runs the task with the highest priority in ready-status. µC/OS supports a maximum of 64 tasks, which correspond to a priority status of 0~63, 0 being the highest priority. Scheduled tasks can be divided into two parts: search and switch, of the task with the highest priority.

Search of the task with the highest priority is performed by setting up the task in ready status. All tasks in µC/OS have an independent stack. They also have a data structure called tcb (task control block) with a stack index. The task scheduling module first records the tcb address of the task in ready status with the highest priority, currently with the OStcbHighRdy variant, then invokes the os_task_sw() function to realize the task switch.

The uClinux task scheduling adopts the traditional Linux mode. The system starts the task in certain intervals, generates fast and periodic clock-timing interrupt, and decides when the program could possess its time slice by using the scheduling function (timer processing function). It then makes the related task switch by invoking the fork function with the parent task.

After the invoking the fork function in the uClinux system, the subtask substitutes the parent task to realize the implementation. This time, an executable file is generated, even if this task is a copy of the parent task. After the subtask has exited or executed, it awakes the parent task using a wakeup to continue the parent task implementation.

Because uClinux does not have a memory management unit (MMU), its access to the memory is direct, and the accessed addresses in all programs are real physical addresses. The operating system does not protect the memory space, so all tasks actually share the same running space. In this case, data protection needs to be made during the multi-task implementation, which may also result in the user's program taking up the system core space. All these problems should be addressed during the design stage.

From the above analysis, we deduced that µC/OS is best suited our real-time system requirements.

File System

The file system handles the access and management of file information. These operations include file creation, reading/writing, modifying, copying, and software programs that manage resources (directory table, storage media, etc.).

µC/OS is used for medium and small-sized embedded systems. Because the core of µC/OS is only 6 to 10 Kbytes after compilation, the system itself does not support the file system.

uClinux inherits the perfect Linux file system performance. It adopts the romfs file system, which requires less space than the general ext2 file system. The space savings is made up of two aspects. The core needs less code when supporting the romfs file system than supporting the ext2 file system. On the other hand, romfs file system is easier to implement, and requires less storage space when establishing the superblock file system. The romfs file system does not support dynamic erase saving. It adopts a virtual RAM mode to process data that needs to be dynamically saved by the system (RAM would use ext2 file system).

uClinux also inherits the advantages of the Linux network operating system, which supports the network file system and embeds TCP/IP protocol that simplifies development of the network embedded devices of uClinux RTOS.

Based on the file-system support, we decided that uClinux is better suited for complex embedded systems that need many file processes. In contrast, µC/OS is suitable for smaller control system applications.

Migration of the Operating System

The basic idea of migrating an embedded operating system is to run the OS on a certain microprocessor or microcontroller. μC/OS and uClinux are operating systems with open original code whose structural designs make it simple to separate processor-related segments. Therefore, they can migrate to the new processor easily.

The target processor has to meet the following requirements to migrate μC/OS:

■ The C program compiler of the processor must generate re-settable machine code and open and close the interrupt routine using the C language.

■ The processor supports interrupt and can generate interrupts periodically.

■ The processor supports abundant RAM (several Kbytes) as the task stack under a multi-tasking environment.

■ The processor features instructions to read or store the stack index and other CPU registers into the stack or memory.

When compared with μC/OS, migrating the uClinux RTOS is more complex. Generally speaking, the target processor should also support external ROM and RAM with abundant capacity in addition to the above-mentioned μC/OS requirements.

The migration of uClinux can be split into three hierarchies:

■ *Migration of hierarchy*—If the structure of the processor to be migrated is not the same as any supported processor structure, then the relative processor structure archive in the linux/arch directory must be modified. Although most of the machine code of the uClinux core is independent of the processor and its architecture, the machine code in the lowest hierarchy is not the same in different systems because of their unique interrupt process program, memory map, task process invoking, and initialization process. These routines are in the linux/arch/directory. Because Linux supports various architectures, to handle a new architecture, its low-level program should be compiled by copying the architecture similarly.

■ *Platform level migration*—If the processor to be migrated is the branch processor of an architecture that has been supported by uClinux, the corresponding directory must be established under a relative architecture directory and the corresponding machine code must be compiled. For example, the migration in this case must establish the linux/arch/nioscpu/platform/nios directory and compile the tracking program (to realize the user program to core function interface and some other functions), the interrupt control program, and the vector initialization program in the directory.

■ *Board level migration*—If the processor is supported by uClinux, then board level migration will suffice. The board level migration needs a corresponding board directory established in the linux/arch/platform/ directory, and it should contain the corresponding starting machine code crt0_rom.s or crt0_ram.s and link description file rom.ld or ram.ld. The board level migration also includes driver compilation and environment variables setup.

Comparing μC/OS and uClinux, we can see that the two operating systems have their strengths and weaknesses. μC/OS takes up less space, implements with high efficiency, offers real-time performance, and migrates to a new processor relatively easily. uClinux takes up more space, implements with weak real-time performance, and migrates to the new processor in a relatively complex way. However, with its embedded TCP/IP protocol, uClinux supports various file systems, and benefits from abundant open Linux program resources. uClinux is stronger when used in some more complex applications.

Performance Parameters

The system features the Nios II /s CPU, peripheral compact flash (CF) card, Ethernet network chip, MP3 decoding chip, ROM, SRAM, SDRAM, and other peripherals. The hardware design is shown in Figure 10, and the hardware resource utilization in Table 1. Figures 11 and 12 show the finished MP3 player development platform.

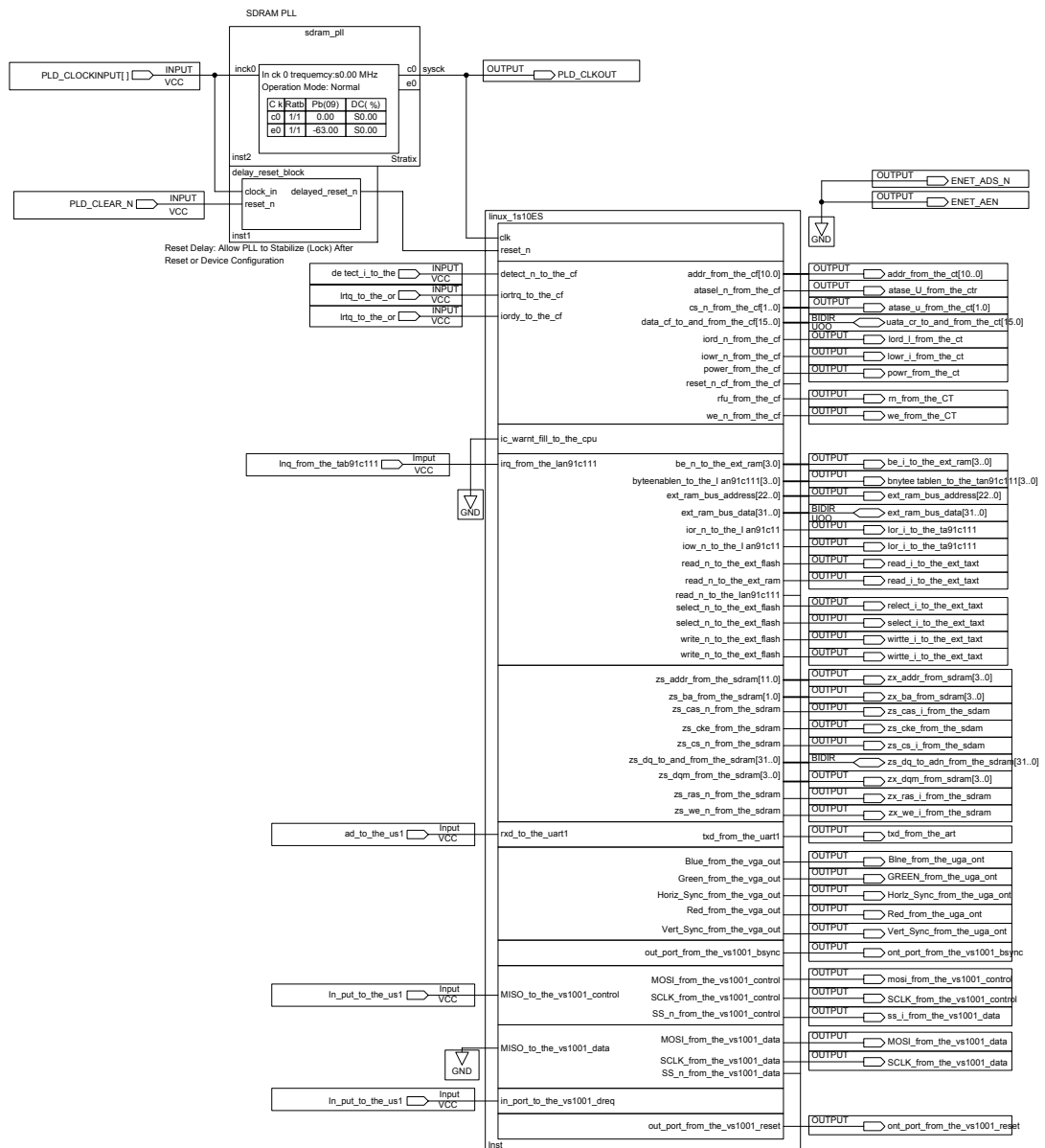*Figure 10. Nios II CPU & Peripheral Component Design*

Table 1. MP3 Chip Used by Network Player Hardware & Performance

Item	Description		
Function	Nios II MP3 Player System		
Hardware Device	Altera EP1SF780C6ES FPGA		
	Available	Usage	
LEs	10,570	6,575 *	
9-Bit DSP Blocks	48	9	
On Chip Memory (Bits)	920,448	801,536	
		CPU	On Chip RAM	VGA
		45,824	524,288	230,400
I/O Pins	427	191	
Performance	61.63 MHz (fmax)		

*Using the Quartus II software version 5.0 group compiling and integration; Set options together: the least logic

As for software resources, the uClinux kernel and file system take up 1,784 Kbytes and 2,417 Kbytes of external flash ROM, respectively. The application takes up about 30-Kbytes space, and the MP3 files are stored on the CF card.

Figure 11. Outside View of MP3 Player LCD

Figure 12. MP3 Player Development Platform
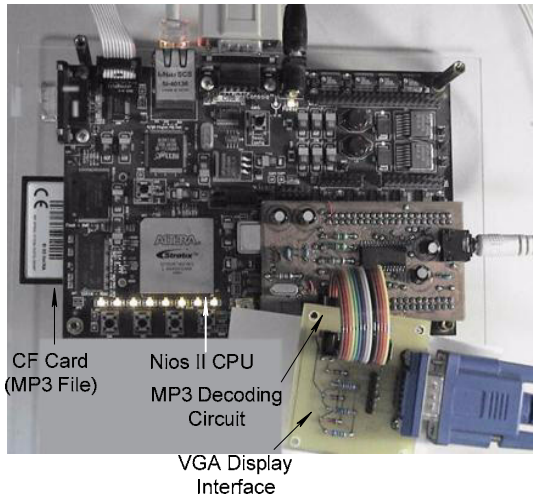
Design Architecture

This section describes the project's design architecture.

MP3 Broadcasting Network

This system architecture will be used for music audio/broadcasting or music audition (see Figure 13). MP3 audio is provided by the MP3 broadcasting server and is downloaded to each MP3 receiver from the Ethernet. The broadcasting server selects the audio actively, and the receiver plays or audits the downloaded music. In this system, the broadcasting server is based on a PC because of the large hard disk, which is easy to store, extend, and maintain. Additionally, the hard disk offers abundant operating system support. If the system is used for business purposes, the Linux/uClinux is preferable because of the server operating system and receiver operating system, considering system stability and costs.

Figure 13. Ethernet MP3 Broadcasting System
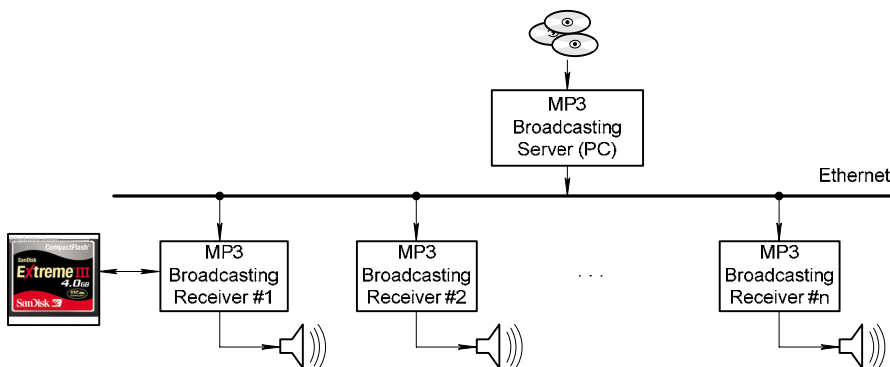
MP3 Broadcasting Receiver

Figure 14 shows the circuit function block of the MP3 broadcasting receiver. After receiving the Ethernet data, the MP3 data is first prepared for processing by the Nios II CPU, which also performs

MP3 decompression. Then, the audio is output via a dual-channel amplifier. The received MP3 data can also be stored in flash memory for playback.

Figure 14. Ethernet MP3 Receiver
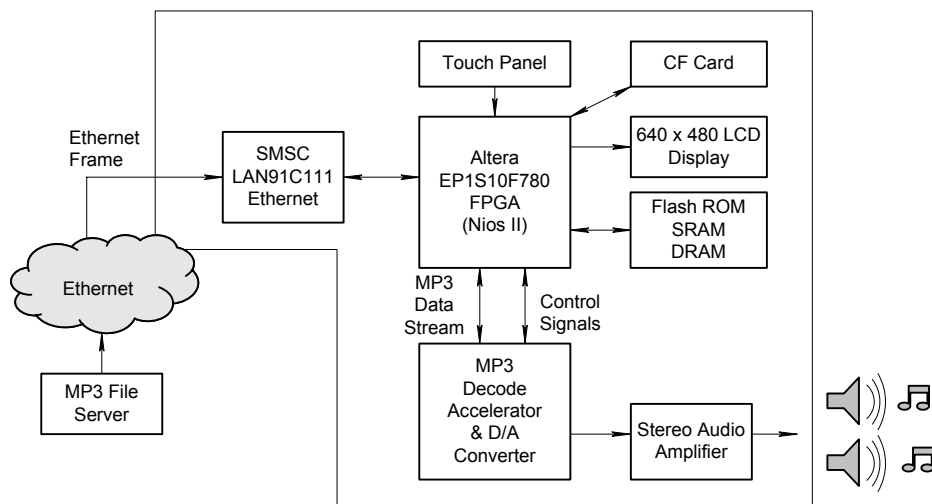
MP3 Decoder

From the previous MP3 coding theory, we found that you needed a huge amount of data packet composing or decomposing, and a great deal of repeated numerical operations in both the coding and decoding processes. Additionally, in a real-time system, a large volume of data needs to be processed in every clock cycle. If the processor speed is low in an embedded system, then you cannot play MP3 audio. Our system has been designed to accelerate MP3 decoding with the help of a peripheral decoding circuit. Although the Nios II processor in an FPGA can perform the MP3 software decoding function, it cannot reach the real-time processing speed. Therefore, the MP3 data stream decoding is implemented using an additional decoding chip to reduce the burden on the CPU.

The system adopts a special chip to implement MP3 decoding. Other design considerations include:

■ MP3 audio play needs to be transformed into simulation signals towards the end of the process. However, the FPGA currently cannot implement this simulation circuit; in contrast, the dedicated chip features a simulation circuit. Taking this design approach, you can save on components of the simulation circuit.

■ Because MP3 audio players are very popular, you can easily get cheap, reliable, and low-power integrated circuits.

Figure 15 shows the block diagram of internal functions of the VS1001, an MP3 decoding chip developed by VLSI Solution Oy from Finland. It features a low-power DSP chip, which could be used to implement user programs to process special audio effects. It also operates as the MP3 decoder and has a 16-digit dual channel digital-to-analog converter (DAC) with no phase difference and a simulation earphone amplifying circuit. These features could significantly simplify the MP3 decoder production.

Figure 15. Block Diagram of Internal Functions in VS1001 MP3 Decoding Chip
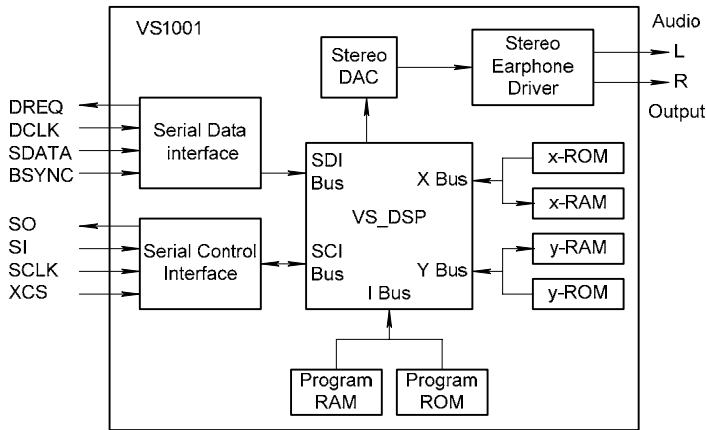
Figure 16 shows the circuit connections between this chip and the Nios II development platform. The 3.3-V working power is obtained from P17 (on the board), the controlled serial peripheral interface (SPI) signal and MP3 data's SPI signal are connected with to the development board at P11. The two SPI signals are controlled by the independent SPI interface in the FPGA (see Figure 16).

The VS1001 chip uses two SPI buses, the serial data interface (SDI) that transmits MP3 compressed data and serial control interface (SCI) that implements control instructions. By reading/writing the 16-bit register of the SCI interface, the following operations can be implemented:

■ Operation mode control

■ Loading user program

■ Reading header data

■ Reading status information

■ Accessing decompressed digital data

■ Feed-in entry data

Figure 16. MP3 Decoding Chip Circuit
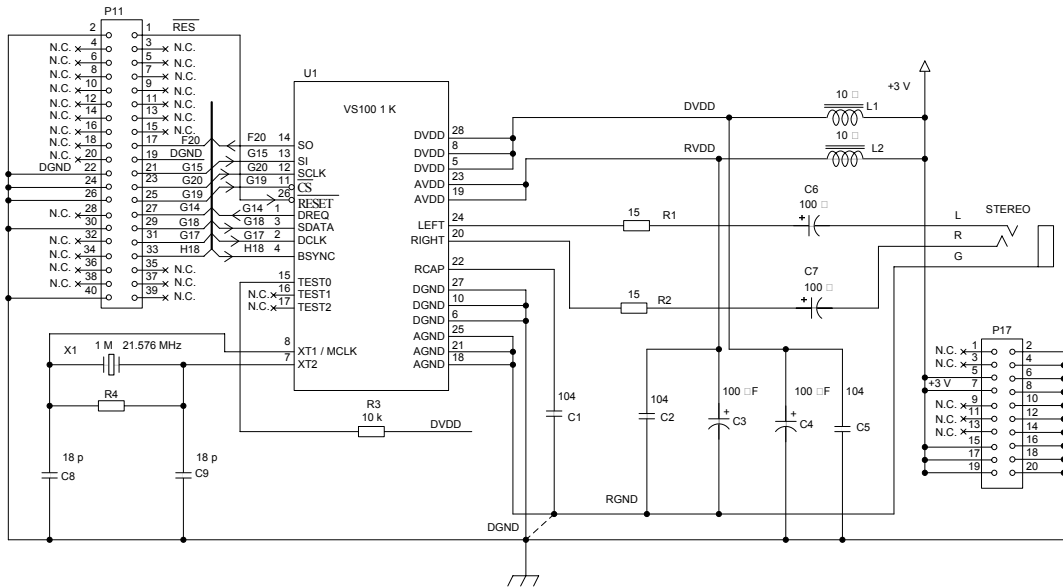
The SCI instruction read/write waveforms are shown in Figures 17 and 18, respectively. The data is read or written to at the SCK rising edge.

Figure 17. Reading of SCI Word
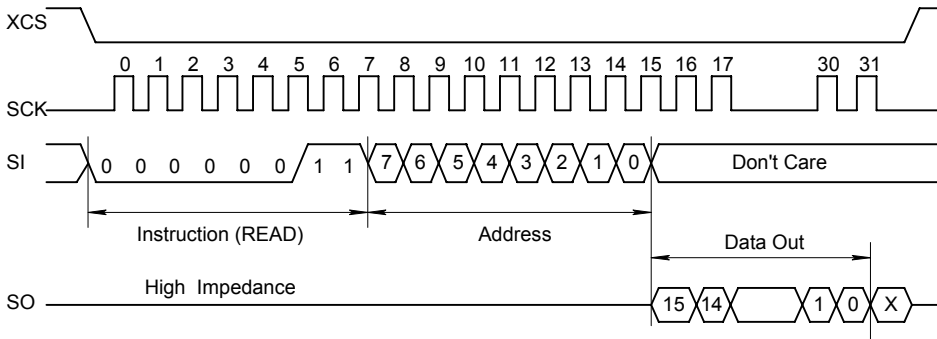
Figure 18. Writing of SCI Word
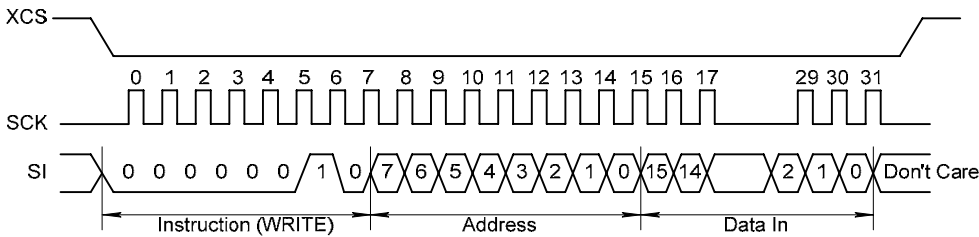
Figure 19 shows the MP3 decoding flow in the VS1001 device and its program steps are as follows:

1. MP3 data is entered through the SDI bus.The data is delivered to the bass/tenor enhancing circuit, which is controlled by the SM_BASS of the SCI register, after MP3 decoding.

2. If A1ADDR of the SCI register is not zero, the application code set by the user is implemented. The starting address is specified by A1ADDR.

3. The digital PCM audio data is then sent to the volume control unit. The output signal is temporarily stored in a FIFO buffer, and is then transformed into simulation audio by the DAC for output according to the sampling frequency. The FIFO buffer can save 512 stereo audio (2 x 16 bits).

Figure 19. VS1001 MP3 Decoding Flow
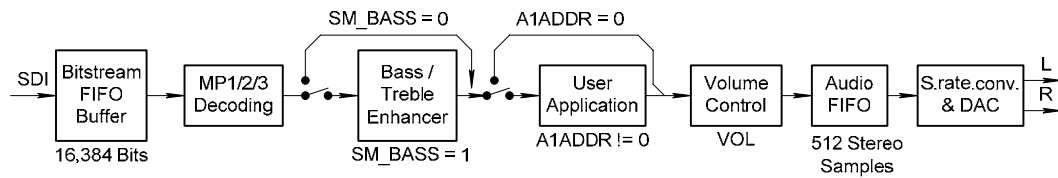
Figure 20 shows the connection of the microprocessor to the VS1001 chip. The basic settings are listed for the microprocessor interface as follows:

■   SO and DREQ are inputs, and the rest of the signals are outputs.

■   When the SPI control is idle, the PI clocks should be set to low power.

■   If the microprocessor has no SPI signal interface, the SO, SI and SCK signals could be implemented by the general I/O; however, the microprocessor must have a fast enough operation speed.

Figure 20. VGA Display Control Circuit
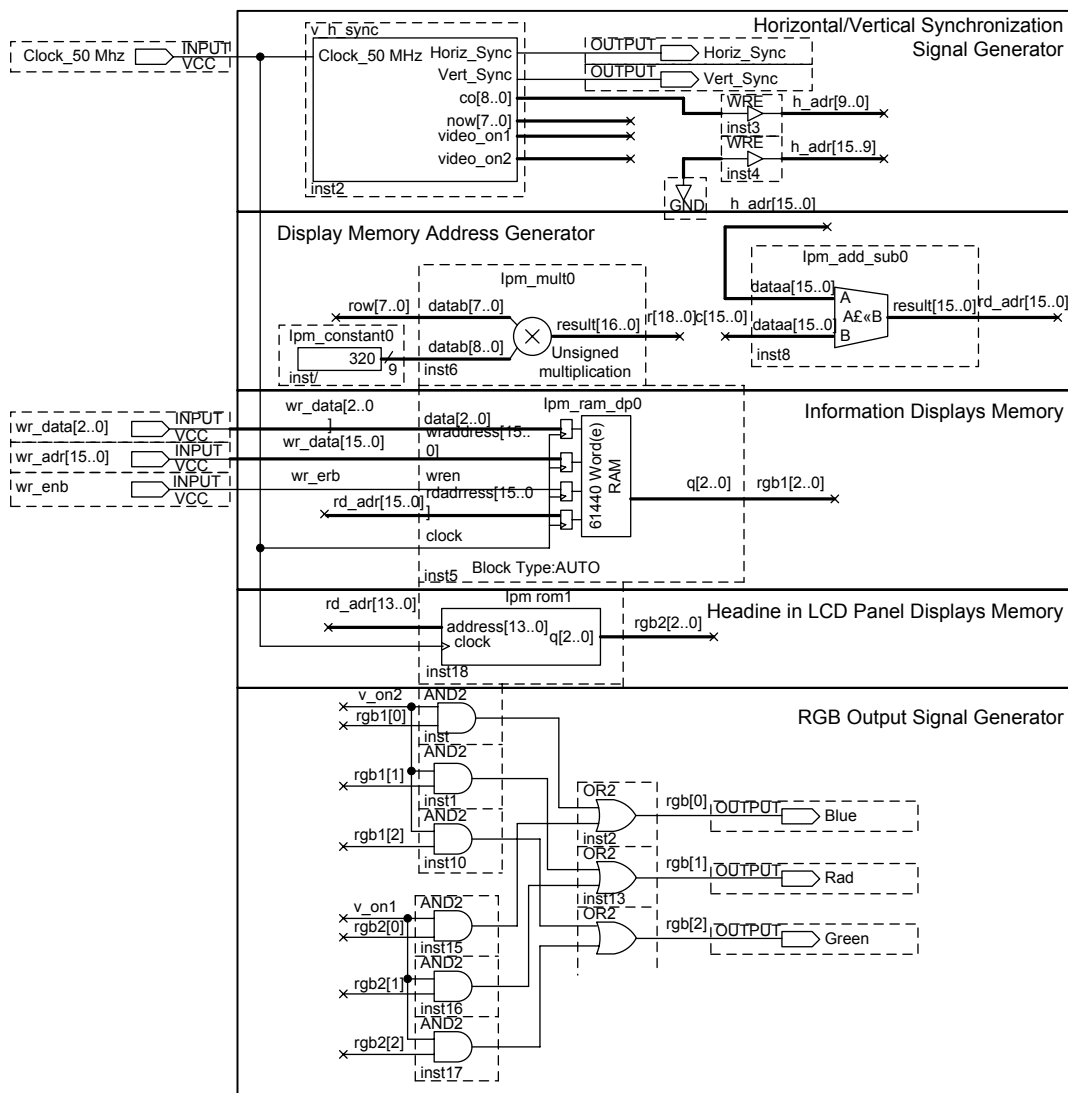
Figure 21 shows the VGA output interface circuit.

Figure 21. VGA Output Interface Circuit (from Altera UP3 Development Kit)
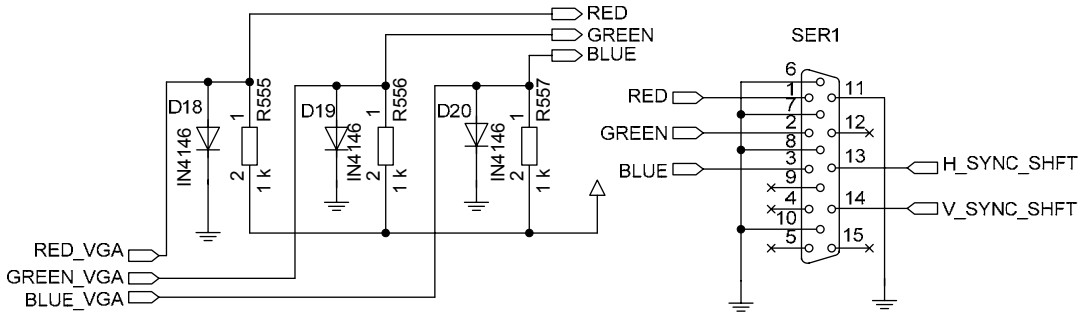
LCD Display Panel

A dot-matrix LCD controller is added to the MP3 receiver for displaying the related MP3 information (such as MP3 file name, length, bit rate, receiving status, and play menu) on the 640 x 480-pixel LCD panel. The panel information operation is controlled by the touch panel.

Touch Panel

Touch panels are becoming more and more popular these days. They are mainly used in environments where there are space constraints and it is not convenient to use a normal keyboard. These applications include operating table supervision systems, self-service meal ordering systems, PDAs, cell phones, logistics management, and inventory management. To make it easier for the user, a four-wire touch panel, rather than an ordinary keypad, is used in the system. Its principle is described as follows.

The basic structure of the four-wire resistive element is very simple: it is made up of two resistive films (see Figure 22). On the X axis of the top resistive film, there are X+ and X- poles connected with the resistive film; and another two Y+ and Y- poles connected with the bottom resistive film. A voltage is applied interactively onto one side of the pole of the two-layer resistive film. When the two-layer resistive film is touched, a voltage value resulting from the touch of the resistive films can be measured via the pole on other side, which is not electrically connected, and hence the X and Y coordinate of the touch point can be obtained. The touch panel outputs X, Y coordinates in serial mode after processing by the interface integrated circuit. Its data format is shown in the tables following Figure 22.

Figure 22. Touch Panel Diagram
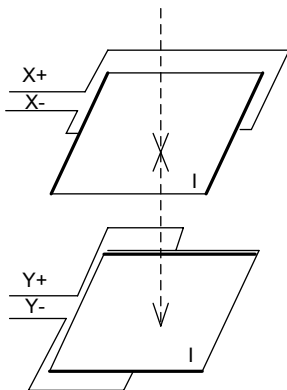
Description	Data Byte				
Pen Up	1	2	3	4	5
	BF	00000xxx	0xxxxxxx	00000yyy	0yyyyyyy

Description	Data Byte				
Pen Down	1	2	3	4	5
	FF	00000xxx	0xxxxxxx	00000yyy	0yyyyyyy

If the first serial data character is 0xFF, the following four characters are X, Y coordinates of the touching point of the touch panel; if it is 0xBF, the characters are the X, Y coordinates of the leaving point of the touch panel. We have mapped the system LCD screen into 40 horizontal characters and 30 vertical characters. Therefore, the program divides the read X, Y value with 40 and 30 to determine the touch display position of the character. Every time the panel is touched, the Nios II CPU issues a short beep to confirm the touch command.

Ethernet Chip

We used SMSC's LAN91C111 device as the Ethernet control chip in our MP3 receiver. The LAN91C111 device is a 128-pin TQFP, full-duplex network chip that can be connected with 8-, 16- or 32-bit microprocessors and can work in 10/100-Mbps mode. See the following table.

Item	Byte/Bit	7	6	5	4	3	2	1	0
Command code	1	C7	C6	C5	C4	C3	C2	C1	C0
X-High byte	2	0	0	0	0	0	X9	X8	X7
X-Low byte	3	0	X6	X5	X4	X3	X2	X1	X0
Y-High byte	4	0	0	0	0	0	Y9	Y8	Y7
Y-Low byte	5	0	Y6	Y5	Y4	Y3	Y2	Y1	Y0

Currently, the LAN91C111 chip used by our system network interface has two main function blocks: the MAC and PHY. The MAC is used mainly for digital data processing and the PHY for simulation data processing. Figure 23 shows the LAN91C111 simplified circuit diagram with the MAC and PHY function blocks. Figure 24 shows the block diagram of the LAN91C111 device's internal functions.

Figure 23. Basic Connection Block Diagram
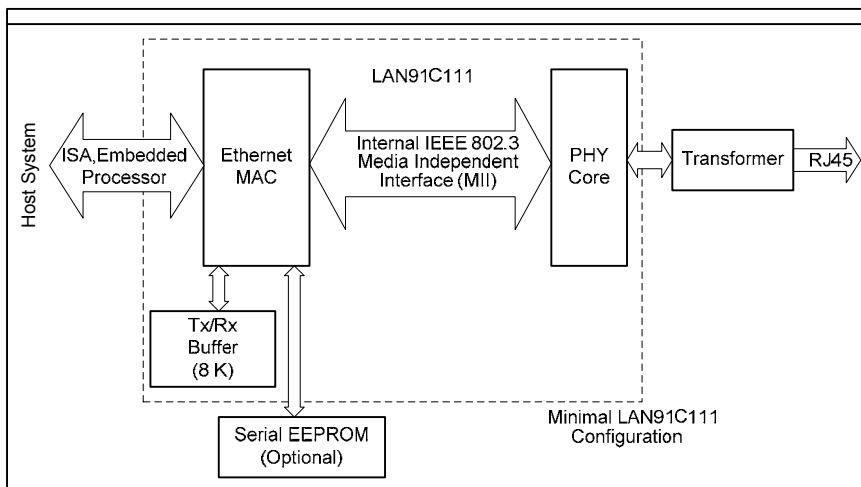
Figure 24. Basic Functional Block Diagram
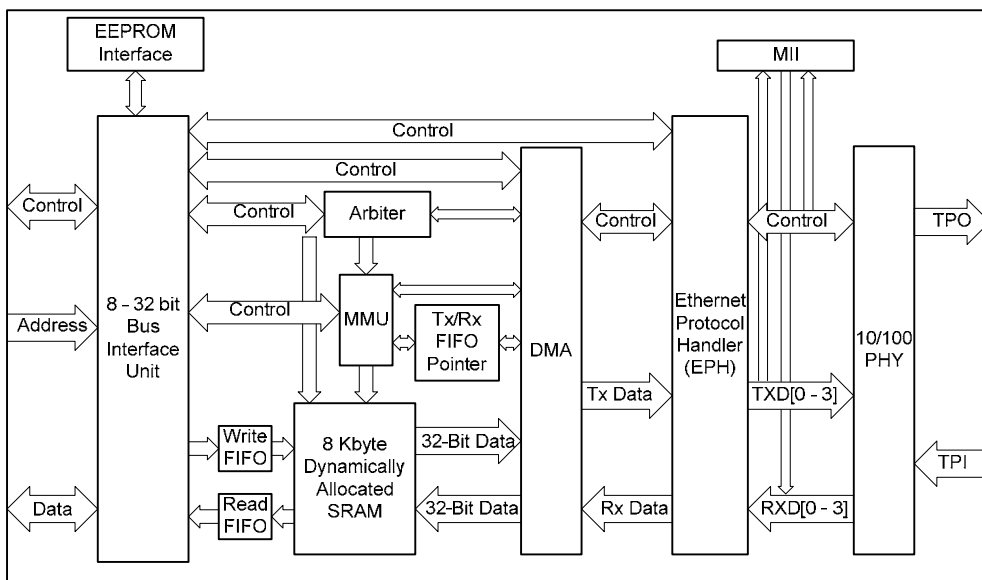
The LAN91C111 device features an 8-Kbyte FIFO buffer, which is used to store the transmitted and received data packets. The FIFO buffer can be accessed externally in DMA mode. The device uses 9,346 (64 x 16-bit EEPROM) to store resource configuration (e.g., I/O address, boot ROM base address, and interrupt request source) and ID parameters. The chip control circuit comprises four register banks. The first 16 registers are used for control and status, registers 16 through 23 are used for DMA data access, and register 24 through 31 are used for chip reset. The transmitted or received data packets range from 60 to 1,514 bytes:

Item	Description	Size
Destination address	MAC address of target node	6 bytes
Source address	MAC address of sending node	6 bytes
Length	Packet length	2 bytes
Data	Data	46 ~ 1,500 bytes

The transmit and receive flow of data packets are shown in Figures 25 and 26, respectively. The packet transmission flow includes memory configuration, packet buffer writing, packet array arrangement, and transmission.

Figure 25. Data Packing Transmission Flow
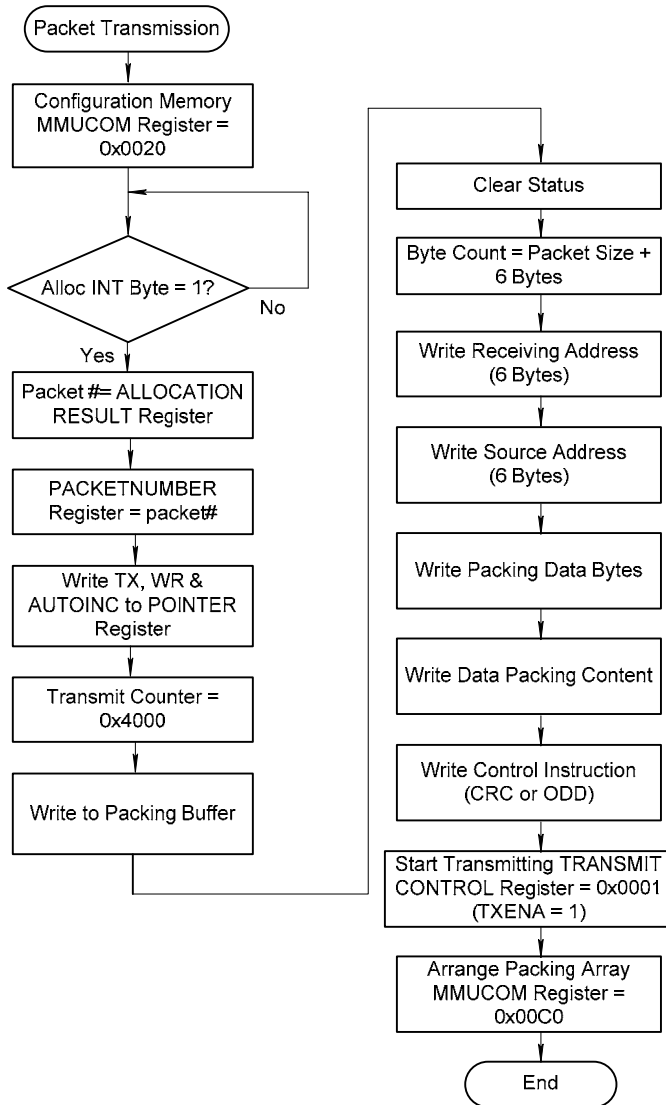
Figure 26. Data Packet Receiving Flow
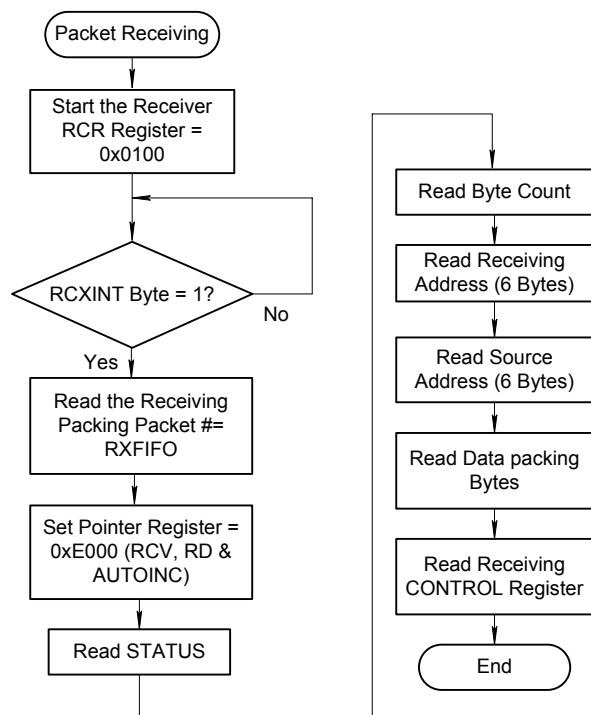
Operating System

We designed the system software based on the uClinux RTOS that can be implemented on the Nios II processor. The main functions of each task are briefly described as follows (see Figure 27):

■ *Transport stream (TS) processing program*—This program implements the TS input, output, and memory management functions. It is similar to the physical layer of the communications system, which implements transmission control of the hardware system. In general, this task stores the data read from the FIFO buffer in different segments of SRAM, and submits the index address. Meanwhile, the data stream to be output is read from SRAM and written to the output FIFO buffer according to the application requirements. In special cases, the input data volume is modified according to the input data stream rate and memory availability.

■ *MP3 system layer analyzer*—The MP3 information that is entered into the data stream is analyzed according to the MP3 system layer standard. Various acquired parameters and audio information are sorted and stored. The data is reassembled according to the simple network management protocol (SNMP) data structure and is updated when necessary. The results are then submitted to the trigger program of the transmission error transaction.

■ *Transmission error supervision program*—The program first completes the synchronization and analysis in succession according to different priorities, and then stores the analysis results data in a structured format to submit to the communications module. It implements a fault prompt and alarm via the pre-designed fault mode. Too many error alarms cause an information jam; therefore, it is helpful to be judicious in judging problems by combining it with the related errors into higher-level alarm information.

- ■ *Communications service program*—The program completes the design of Ethernet transmission control according to TCP/IP protocol and SNMP protocol. Data output transmits the statistical information database and the analytical database to the controller side, based on the standard SNMP protocol. Meanwhile, control command communications are made via TCP or UDP protocols. Semantic analysis can be made for statistical information data entered by SNMP by means of additional analysis software. Nevertheless, the program needs to transmit local hardware timing information as the reference, or display analysis data on the console directly. After adding the web server function to the communications service, the analysis results can be displayed directly via a browser.

Figure 27. Embedded Operational System Software Structure
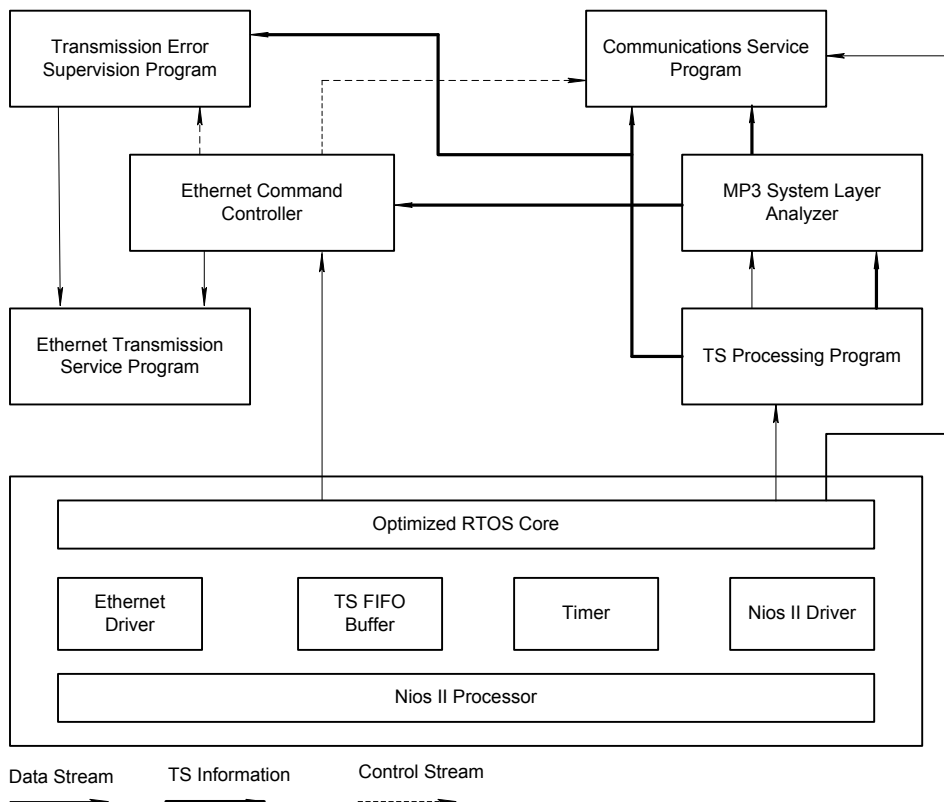
Design Methodology

The development of the system is divided into system planning, hardware circuit design, software program design, OS migration, and system integration test.

At the beginning of system planning, we decided to perform MP3 coding by combining software and hardware. During the testing phase, we realized that using only software programs will not work in real-time, as they consume large amounts of time. Also, if we used MP3 coding hardware circuits along with the FPGA, these circuits would occupy too many chip resources, and some circuits would operate poorly. Therefore, we decided to use an additional MP3 coding chip to save the process of implementing a simulation circuit.

The hardware circuit design includes the Nios II CPU, touch pad, compact flash (CF) card, memory, Ethernet interface, VGA display interface, and MP3 chip control interface. We developed the circuits in VHDL, and used the SOPC Builder tool for the CPU and the peripheral design. Finally, we used the Quartus II software to compile and compose our design.

The software program was written in the C language and compiled with the gcc compiler. For making easy modifications to the system during development, we performed the interface circuits tests of all using independent programs without involving the OS. This process saved us development time because we did not have to recompile the software/hardware combination of the design.

The uClinux RTOS was downloaded to the Nios II development board. The CF card access and Ethernet operated under the RTOS, so we did not have to write drivers.

After completing the actions and tests above, we took enough time to integrate and test the system, using the following steps:

1. Establish the kernel project. Plan and compile the Nios II CPU system that generates the PTF file for the establishment of the kernel (including selecting the development board whose kernel configuration is set to the Stratix® or Cyclone™ device).

2. Establish a file system project, including the basic instructions to be performed on the OS, as well as some application programs such as boa server, telnet, and ftp.

3. Download constructed kernel (vmlinux.bin) and file system, romfs.bin, to the flash ROM, and then download the SRAM Object File (.sof) or Programmer Object File (.pof) of the CPU. Once the software development kit (SDK) window changes to Nios II terminal mode, you can start the OS and check some messages, as well as control the OS after inputting the user account and password.

4. The development of the application program must establish the write program, build the makefile, and specify the option setting of the compilation in the integrated development environment (IDE). The program should copy the .exe file generated by compilation to the filesystem/bin subdirectory, rebuild the project to download to the flash ROM, and start the RTOS to test the program.

5. When testing a program, you must first start up the RTOS network connection and download the program execution file to the CF card via ftp or telnet.

When the tested program is in an endless loop, it can skip over the execution program if the user presses the Ctrl+C keys when testing with the SDK. However, if you press the Ctrl+C keys in the RTOS, the program only leaves terminal mode and does not end the program.

Design Features

This design project integrates the Nios II CPU, the MP3 encoder, Ethernet, the RTOS, and other software/hardware technologies on the SOPC, and completes the embedded network MP3 broadcast system. The system can download MP3 messages through Ethernet, which are then played directly on the MP3 receiver. This system is applicable for use in:

■   Public places, replacing conventional loudspeaker broadcasts and providing good quality audio and voice messages.

■   Music audio in shops, replacing CD audio players and providing the convenience of the latest MP3 technology.

The main function blocks in the system design include:

■   Nios II CPU and peripheral circuit plan

■   MP3 encoder interface circuit

■   VGA display interface circuit

■   Embedded Ethernet

■   Embedded OS uClinux

■   MP3 file server

■   SOPC software/hardware system integration

Considering the flexible design of the system software/hardware, we adopted the SOPC design methodology to complete the system. If the system were to be implemented only using software programs, it would be extremely difficult to process the MP3 data in real-time, or it would require a higher performance processor, which is too costly and consumes more power. Many application programs of the system cannot be realized in hardware circuitry, and therefore are not flexible. Therefore, an FPGA that contains the Nios II soft processor is the best choice for this design.

Conclusion

The purpose of our design, the Embedded Network MP3 Playing System, is to make general public announcements at selling booths and public places. These messages are played in real-time, and could include popular music or electronic information (in MP3 file format) that can be transmitted to MP3 players by the main control room to the Internet or LAN.

The system core uses a 32-bit RISC Nios II embedded soft processor, which was released by Altera in 2004. The system OS is uClinux. The development tools, such as the SOPC Builder and the Quartus II software version 5.0 IDE, helped us to partition the software/hardware module design, compile, combine, program, and test, as well as to integrate the program into the Altera Stratix FPGA development board.

The Nios II processer has three types of optimal CPU variants: one that has high system performance, one that uses the fewest logic resources, and one that provides a balance between system performance and logic resources.

Although Altera's development tools were good, we faced a few problems during development.

■ It was difficult to make an optimal choice because selecting the CPU, parameters of the peripheral components, and even circuit combinations were very complicated.

■ Whether the RTOS was in use or not, the OS name was not matched during hardware development, which we needed to change manually.

■ It was difficult to write this report because the sections were out of order and contained repeated information. Instead, we would like to suggest the following project report sections: Motivation and Purpose, System Architecture (including circuit diagram, program flow, and specifications for use), Design Principle, Design Description (detailed circuit diagram and program description), Test and Experiment Result, Conclusion, and Appendix.

The initial hardware design of the system was to select the Nios II /f CPU using SOPC Builder, and add a user logic interface, designed by us, to connect the VGA display. At the beginning of the design, we took more time to add a display with extra pixels, but considering the memory resources used by the system and the size restriction of some RAM and ROM, we used the Nios II /s CPU instead, compiled it in the Quartus II software, and implemented a full-screen display. Besides the LCD display, the system also supports the touch screen input by the UART interface, and allows further design modifications on the touch screen functions. The play of the MP3 decoder is controlled by two SPI interfaces. We spent a lot of time at the beginning of the project on this aspect, due to our unfamiliarity with the communication protocol and SPI timing, learning how the function base of the SPI is applied in the Nios II system, and verifying using an external MP3 decoder circuit. After completing a few modules, we still had some difficulties when integrating the uClinux RTOS, accessing the CF card, and performing Ethernet transmission. We overcame these problems in the end.

For this kind of project, no matter how the software and hardware cooperate, or what the setting and operation of the RTOS, each function that needs to be implemented successfully must be understood and developed with the correct debugging procedures. You cannot wade into this design blindly and use trial and error. That approach wastes a great deal of time and lowers your confidence. A special thanks to our instructors for providing such great help and important suggestions about the design and operation of this product. We thank all our college mates for their active participation and sincere devotion during this competition. Through this competition, we learned how to design an embedded system and perform system integration. We had a pleasant experience entering the competition, and we hope to do even better next time.

Second Prize

Implementation of the H.264/AVC Decoder Using the Nios II Processor

Institution: **Seoul National University**

Participants: **Im Yong Lee, Il-Hyun Park, and Dong-Wook Lee**

Instructor: **Ki-Young Choi**

Design Introduction

H.264/AVC is a video standard of the ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG). This standard has been developed in response to the growing need for higher video compression in applications such as videoconferencing, digital storage media, television broadcasting, internet streaming, and communication. The H.264/AVC standard has been designed to enable the coded video representation in a flexible manner for a wide variety of network environments.[1]

We started our design from Joint Model Reference code. Because the design requires a lot of computations with various sophisticated compression techniques, we needed a high-performance system for real-time video processing. We achieved the necessary performance for a reduced frame rate using the Nios® II Development Kit. Specifically, we used the versatile features of the Nios II configurable processor, such as configurability of the memory hierarchy and custom instruction extensions.

Function Description

Figure 1 shows the H.264/AVC decoder block diagram.

---

[1] ITU-T Rec. H.264(05/2003)

Figure 1. H.264/AVC Decoder Block Diagram

F^1_{n-1} (Reference)

(1 or 2 Previously Encoded Frames)

MC

Inter

P

Intra Prediction

Intra

F^1_{n-1} (Reference)

Filter

uF^1_n

D^1_n

T^{-1}

Q^{-1}

X

Reorder

Entropy Decode

NAL
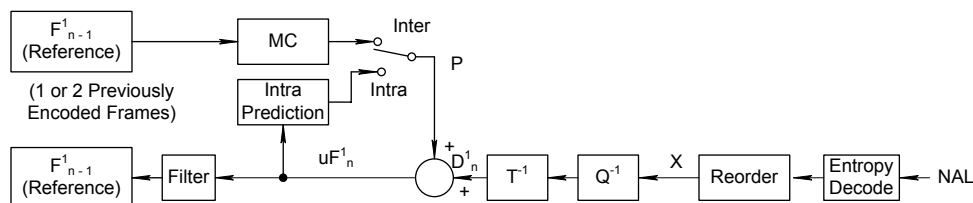
We implemented an H.264/AVC decoder, which can decode about 12 frames per second, with Nios II processor-based system-on-a-programmable-chip (SOPC) solution running at 90 MHz. The function blocks, MC, Intra Prediction, and Filter were implemented as software modules, and the context-based adaptive variable length coding (CAVLC) decoder was implemented using custom instructions. The Inverse Integer Transform and Inverse Quantization blocks were implemented as a single intellectual property (IP) module featuring an Avalon® slave interface. We also implemented the thin-film transistor (TFT) LCD controller and YUV-to-RGB color space converter to display decoded pictures. An expansion prototype connector links our TFT LCD panel to the Nios II development board.

Performance Parameters

Our performance target was to achieve quarter common intermediate format (QCIF) (176 x 144 pixel resolution and 30 frames per second (fps)) decoding capability based on a 200-MHz SOPC solution. With the FPGA implementation, we achieved 90 MHz maximum and we could decode about 12 fps. If we fabricated this solution using 0.18-micron technology, we could increase the clock frequency to 200 MHz, which can process about 27 fps. So, we would still need to increase the performance by more than 10% to meet our original performance target. However, 27 fps is good enough for today's mobile video streaming service.

Design Architecture

The Figure 2 block diagram details our design implementation. We used the Nios II/f (fast) processor's custom instructions for CAVLC decoding. The Inverse Quantization and Inverse Integer Transform blocks were combined into a single IP module with an Avalon slave interface. We used three dual-port RAM blocks for the YUV frame buffer. To transfer the frame data to the frame buffer, we designed an interface between the dual-port RAM and Avalon bus.

Nios II Configuration & Memory Hierarchy

We chose the Nios II/f processor with a hardware multiplier using DSP blocks and a hardware divider. This scheme gives an estimated performance of 102 MIPS (Dhrystones 2.1) at 90 MHz at most, based on a 32-Kbyte instruction cache and 32-Kbyte data cache. The line size of the data cache is 16 bytes. We found that the performance was saturated at this cache configuration and we could get a little improvement by further increasing the cache size. In addition, the system has a tightly coupled data memory of 24 Kbytes. Because the YUV frame buffer uses many M4K blocks, this configuration is almost the maximum amount of memory blocks that can be allocated to cache and then tightly coupled to memory.

Figure 2. Block Diagram of Implemented H.264/AVC Decoder
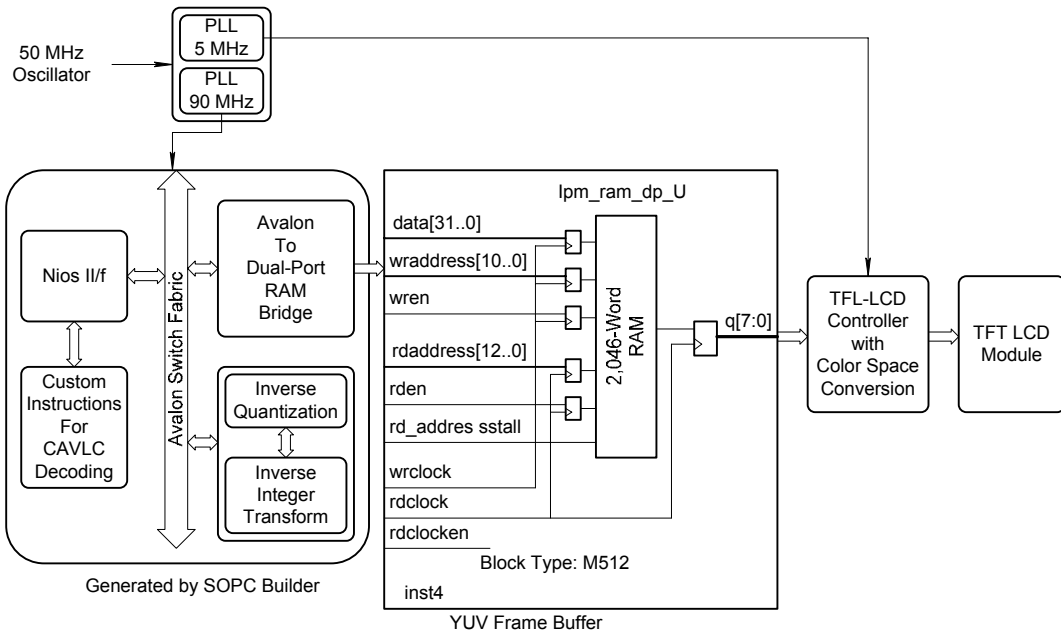
Tightly coupled data memory handles read-only data memory (.rodata), heap memory, and stack memory. Our design application uses about 16 Kbytes for read-only data, which stores frequently used coefficients. The remaining tightly coupled data memory is enough for the heap and stack. We managed to obtain about a 7% speed increase with this memory design modification.

Custom Instructions for CAVLC Decoding

The ReadCoeff4x4_CAVLC function reads an encoded bitstream using CAVLC and decodes coefficients of a 4 x 4 macro block. Figure 3 shows the process of CAVLC decoding. Each block in Figure 3 features 2 to 4 inputs and 1 to 2 outputs. Each of the inputs and outputs has a value ranging from 8 to 24 bits. Each block takes several execution cycles in the best case and several hundred cycles in the worst case. Each block is called more than several hundred times per frame. Because the result of each block is determined by the input data and multiple execution of the following block, it is very difficult to implement this function as a separate hardware IP module. Specifically, implementing each block of Figure 3 as an independent hardware block would cause high data communication overhead. By implementing these blocks as custom instructions, we can use the processor's register to lower overheads on data communications.

Figure 3. CAVLC Decoding Process
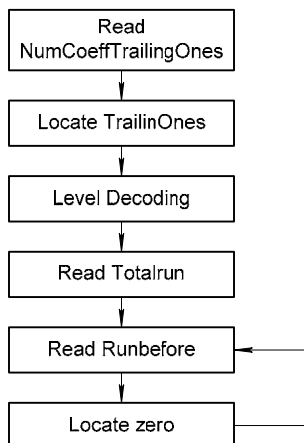
Among the six blocks in Figure 3, five blocks (except the Level Decoding block) have the same structure (see Figure 4). Each of the five blocks first looks up the length table to obtain information on the bits required to be read from the bit stream. Following this, the blocks read that many bits of data from the bit stream and compare the data with the code book. This sequence is repeated until the block finds an exact match. Although all five blocks have the same structure as described above, they have been implemented with different custom instructions because they have different lengths and code tables. Because the largest table size is 3 x 17, the iteration amounts to 3 x 17, worst case.

Figure 4. Flow Chart of Each Block CAVLC Decoding
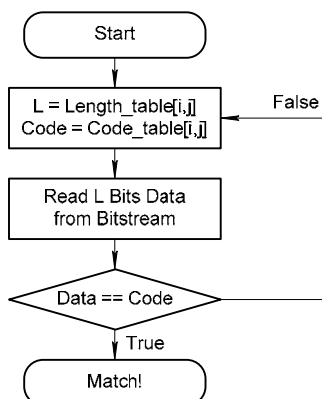
Figure 5 shows the implementation structure of each custom instruction for CAVLC decoding. By loop unrolling and parallel comparison we managed to maximize inherent parallelism and arrived at the exact match in 1 cycle. Using this custom instruction, we achieved about 13% increase in execution speed.

Figure 5. Custom Instruction Implementation of CAVLC Decoding
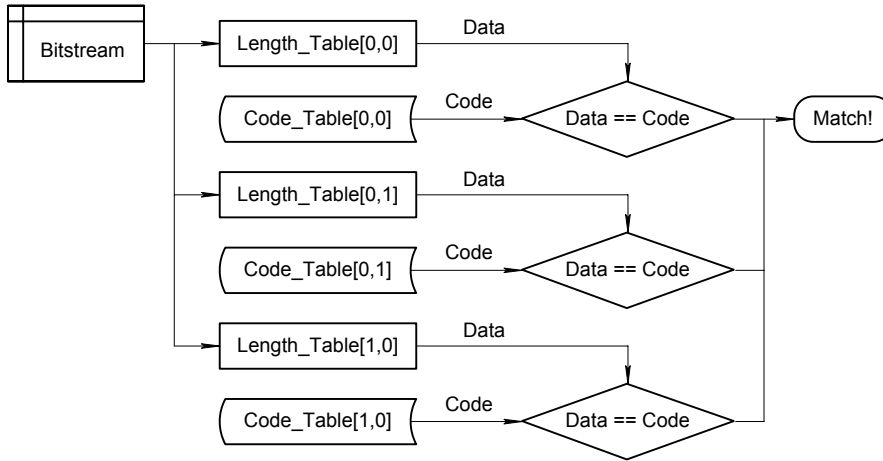
Inverse Quantization & Inverse Integer Transform[2]

We will skip the elaborate mathematical details and simply note that the inverse transform is given by

$C_i^T W C_i$, where W has elements $W_{ij}'s$, which are scaled coefficients computed by

$$W_{ij} = Z_{ij} \cdot V_{ij} \cdot 2^{\lfloor QP/6 \rfloor}.$$

The value of V_{ij} for $0 \le QP \le 5$ is defined in the standard as shown in Table 1.

Table 1. Rescaling Factor V

QP	Positions (0,0),(2,0),(2,2),(0,2)	Positions (1,1),(1,3),(3,1),(3,3)	Other Positions
0	10	16	13
1	11	18	14
2	13	20	16
3	14	23	18
4	16	25	20
5	18	29	23

Z_{ij} is the transformed coefficient which is the output of CAVLC decoding and QP is the quantization parameter which is given by the user when he encodes raw video stream.

We implemented the functionality described above as a single IP module. Inputs to the IP module are sixteen 8-bit data words and sixteen 16-bit data words and the outputs are sixteen 8-bit data words. Because it requires multiple input ports and multiple output ports, we found that it is more efficient to implement it as an IP module than as a custom instruction. The module takes in sixteen adders and

---

[2] H.264/MPEG-4 Part 10 Tutorials at www.vcodex.com

completes operation in five cycles. Using this IP module, we have achieved about a 20% increase in speed.

YUV Frame Buffers

Our TFT LCD controller uses a 5-MHz clock. This is the typical clock frequency to refresh TFT LCD 60 times per second. Because the TFT LCD controller runs on a clock domain different from that of the decoding system, the YUV frame buffers must be implemented as dual-port RAM. So, we used the parameterized dual-port RAM function altsyncram. Even though the dual-port RAM needs only 8 bits at the output port, we configured the input port to be 32-bit wide, because the inherent structure of altsyncram makes it efficient in terms of the data transfer rate.

Design Methodology

For design and implementation, we used various tools from Altera such as Quartus® II software, SOPC Builder, and Nios II integrated development environment (IDE), which are seamlessly integrated and easy to use. This complete toolset from Altera made it easy for us to develop the SOPC solution. In addition, support for third-party EDA tools such as the ModelSim® software was very helpful to verify the behavior of the SOPC design. Figure 6 details the overall design flow and tools we used.

Figure 6. Overall Design Flow & Tools Used
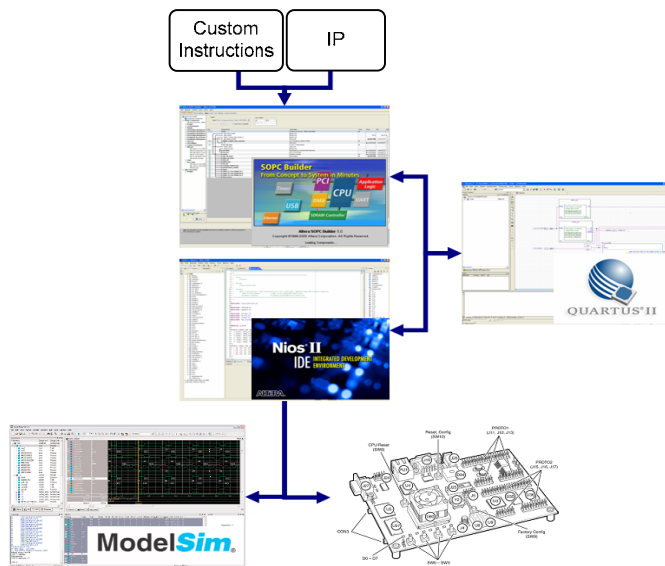
Design Features

The following are salient features of our H.264/AVC design.

■ Optimal configuration of memory design hierarchy and layout.

■ Deployment of custom instructions for CAVLC decoding.

■ Implementation of IP modules for Inverse Quantization and Inverse Integer Transform.

■ Design of TFT LCD controller with YUV-to-RGB color space converter.

■   Design of dual-port RAM for intra-communication between different clock domains.

Conclusion

The Altera Nios II design contest allowed us to design an H.264/AVC decoder targeted for Altera's FPGA, using Altera tools. In our opinion, we have extensively utilized the versatile features of Altera's Nios II configurable processor and SOPC Builder to make the video decoder process 12 QCIF frames per second with a 90-MHz clock frequency. Altera's Nios II development kit gave us a valuable opportunity to experience three alternative ways of design implementation (software, custom instruction, and hardware IP) and how to combine them in a harmonized way to optimize the design.

Third Prize

Spectral Estimation Using a MUSIC Algorithm

Institution: **Indian Institute of Technology, Kanpur**

Participants: **Jawed Qumar**

Instructor: **Baquer Mazhari**

Design Introduction

I have implemented a high resolution spectral estimation multiple signal classification (MUSIC) algorithm in an Altera® Stratix® FPGA. MUSIC detects signal frequencies by performing an eigen decomposition on the data vector covariance matrix from received signal samples. High-resolution spectral estimation is a major challenge of any advanced Doppler radar, cellular mobile base stations, etc. Eigen value decomposition (EVD) and MUSIC temporal spectra computations with a cyclic Jacobi processor based on a Coordinate Rotation Digital Computer (CORDIC), is the major signal processing being implemented using an Altera Stratix FPGA. All the digital signal processing (DSP) functions are based on fixed-point arithmetic and are well suited for the Stratix FPGA architecture. The feature-rich Sratix FPGA is armed with a Nios® II processor that has custom instruction and multi-mastering capabilities, as well as a powerful system development platform: SOPC Builder. The Nios II processor integrated development environment (IDE) has made the FPGA an attractive alternative to implement algebraic signal processing algorithms.

Function Description

The MUSIC algorithm is a kind of directional of arrival (DOA) estimation technique based on eigen value decomposition, which is also called the subspace-based method. Here, we consider a unitary MUSIC algorithm. With this, the eigen decomposition of correlation (covariance) matrix in the MUSIC algorithm can be solved with real numbers only. This system achieves high performance in EVD and MUSIC angular spectra computation with a cyclic Jacobi processor on a CORDIC and spatial DFT respectively. The unitary MUSIC computational flow involves the following steps:

1. Estimation of the correlation matrix, including unitary transform.

2. EVD of the correlation matrix.

3. Computation of the MUSIC spectrum.

4. Local Maximum detection.

I have implemented EVD via a CORDIC-based Jacobi processor. The EVD computation processor for MUSIC DOA uses a CORDIC-based Jacobi method. The cyclic Jacobi processor computes real symmetric eigenvalue problems by applying a sequence of orthonormal rotations to the left and right sides of the target matrix (unitary transformed K X K real symmetric correlation matrix Ryy) as:

$$E^T \cdot R_{yy} \cdot E = D,$$
$$\left(\because \begin{array}{l} E = J_1 \cdot J_2 \cdot J_3 \cdots, \\ J = W_{12} \cdot W_{13} \cdot W_{K-1,K} \end{array} \right)$$

Where Wpq is an orthonormal plane rotation over an angle θ in the (*p, q*) plane whose elements are Wpp = cos θ, Wpq = sinθ, Wqp = −sin θ, Wqq = cosθ (*p > q*). J is the multiple rotation of Wpq's in the cyclic-by-row manner of (*p, q*), which is called a Jacobi sweep, and the superscript T and subscript K denote transposition and array length, respectively. This processor employed the hardware friendly CORDIC algorithm for vector rotators and arctangent computers to solve the above equations, which were the basic processing unit. Because the fixed-point operation is applied, of course approximation errors exist. But when it was implemented with the above 16-bit precision, we could get reasonable performance. In the next section, implementation angular spectrum is computed after the EVD step. See Figure 1.

Figure 1. System Overview
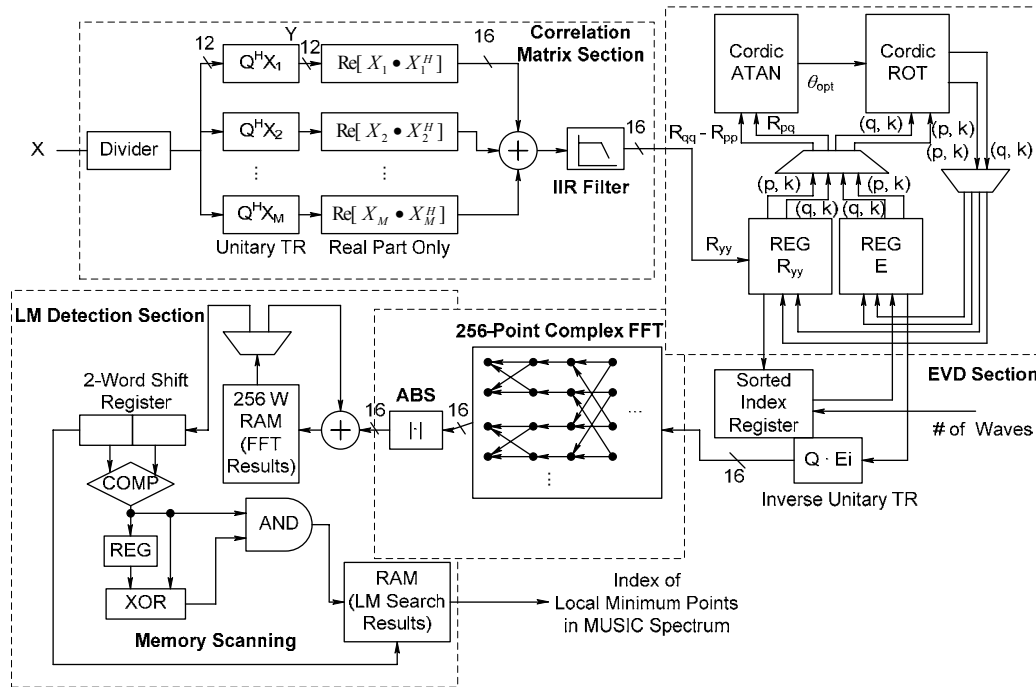
Performance Parameters

The estimated performance of the dominant core functions is the number of occupied logic blocks in the FPGA and f_{MAX} is the maximum clock frequency at which normal operation can be guaranteed. The minimum computation time, *tmin,* is calculated by required clks $* f_{MAX}$. I assumed that less than 2 coherent/incoherent waves arrived at only 4-element uniform linear array antenna. For spectrum generation, 256-point radix-4 complex fast Fourier transform (FFT) was employed and the FFT with 256 spatial data composed of N elements of the noise eigenvector and $(256-N)$ zeroes interpolates the spectrum fine and smoothly. All computations were performed by fixed-point arithmetic with 12-bit input data from ADCs. On the other hand, the estimation accuracy of the EVD system depends on so many factors that the proper assessment has some difficulties in detailed analysis. For example, the effect of finite bit-length and bit-truncation by scaling in the fixed-point operation, the estimation errors caused by non-uniform discrete wavefront, and so forth.

Design Architecture

The EVD of the input matrix X can be performed, as illustrated in Figure 2, using the well known systolic array architecture. The rows of matrix X are fed as inputs to the array from the top, along with the corresponding element of the vector y. The R and u values, held in each of the cells once all the inputs have been passed through the matrix, are the outputs from the EVD. These values are subsequently used to derive the coefficients using a back substitution technique.

Figure 2. EVD of the Input Matrix
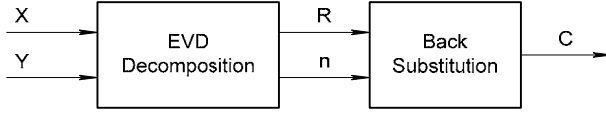
The CORDIC rotation-based algorithm is implemented in a very efficient pipelined manner using a triangular systolic array. The schematic is shown in Figure 3, for M = 4 antenna elements.

Figure 3. CORDIC Rotation-Based Algorithm Schematic

(a)   (b)
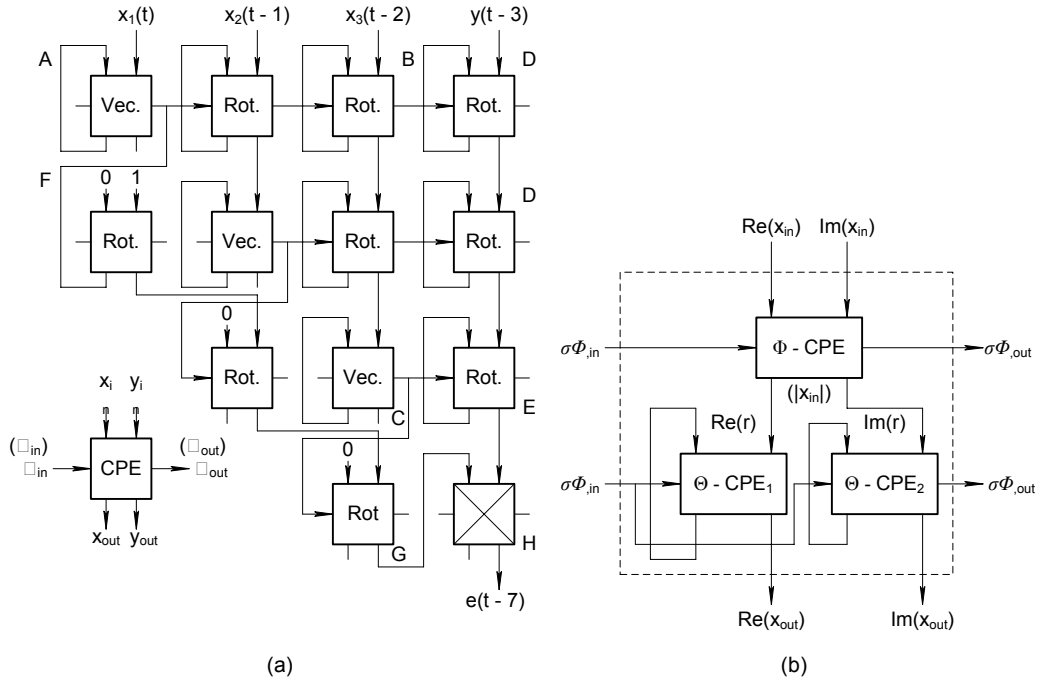
The cells in the triangular array (A-B-C) store the elements of the evolving triangular matrix R[i], and the ones in the right hand column (D-E) store the elements of the updated vector u[i]. The data flow is from top to bottom, while the rotation angles are propagated from left to the right of the array. In this implementation, the array entirely consists of CORDIC processor elements (CPEs), which work completely synchronously, driven by a single global master clock. Because all the CPEs need the same amount of time to perform their computations they never get flooded with data. Thereby, the CPEs designated with Vec are configured to the "vectoring" mode of operation, and those labeled with Rot operate in the "rotation" mode. Each row performs a given rotation, whereby the rotation angle is determined by the CPE in vectoring mode at the beginning of the row. The rotation angle is passed to the rotation CPEs to the right with one clock cycle delay, thus requiring the elements of the data vector to be applied to the array in a time-staggered fashion, as indicated by the indices in Figure 3. To handle the complex data, complex CORDIC is used. As shown above, it is comprised of three CPEs interconnected according to figure 4b. In vectoring mode ($Im(r_{m,m}) = 0$), the imaginary part of the complex value x_m is annihilated by the Φ-CPE and subsequently $|x_m|$ is zeroed in θ-CPE$_1$. The complex Givens rotation is then coded by the two sequences of rotation coefficients $\{\sigma_{\Phi_j}\}$ and $\{\sigma_{\theta_j}\}$. By applying these rotation coefficients to a supercell configured to operate in the rotation mode, the incoming vector $(Re(x_{in})\ Im(y_{in}))^T$ is rotated by Φ_m in the Φ-CPE and subsequently the real and imaginary parts of $r_{m,n}$ and $x_{m,n}$ are each rotated by θ_m in θ-CPE$_1$ and θ-CPE$_2$, respectively.

77

The heart of the design is the EVD decomposer block. The hardware implementation is carried out directly using systolic array. I worked this out first with a kind of direct mapping, where as many CORDIC blocks are required. The aim was first to get the R matrix and U matrix from the given input of X and Y matrix. The rest of the task is taken care of by the Nios II processor. The number of logic elements used was very high. Also, the EVD update can be done very fast: this is not required for so many practical applications, for example, a radar system where the interference environment changes in milliseconds or hundreds of microseconds. So excess hardware utilization and achieving high speed is of little interest. To address this problem, the array can be mapped to a reduced number of CPEs on a time-shared basis.

Complex CORDIC blocks are required, so as to implement the complex data. Each complex CORDIC block consists of three basic CORDIC blocks. I have implemented systolic array for four antenna elements. As mentioned, this approach gave satisfactory output, but the problem with the scheme is that it consumes too many logic elements, which is not practical. So, I have worked out another scheme which does the same thing with only two complex CORDIC blocks. This approach is called mixed mapping which consumes less logic elements. The benefit is achieved with the scheme, but latency also will be there. This latency is unavoidable. The scheme is practical, as resource utilization is well within the limits of the Stratix FPGA. This requires an additional state machine to control the operation. Out of the two CORDIC blocks, one is for vectoring mode and another for rotating mode of operation.

Figure 4 is the block diagram representation of the design.

Figure 4. Block Diagram
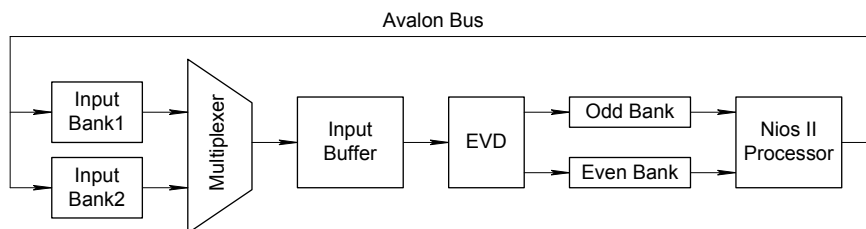
The data on which EVD is to be carried out is in bank1 and bank2. The multiplexer will select the bank alternately and will pass it to the input buffer. This input buffer is controlled, so when required, it is being read and given to the EVD. The EVD will write data alternately in the odd and even memory bank. This is because when Nios II is reading from one bank, the EVD will write data in the other memory bank. The Nios II processor communicates with the peripheral using the Avalon® bus.

Figure 5 is a schematic representation of the EVD. The input data is 32 bit and of complex nature. The EVD requires two types of operations, namely boundary-cell and internal-cell operation. As we have used a mixed mapping approach, the scheduling of the complex CORDIC block is a must here. This is achieved at the cost of speed. The load enable signal initiates the EVD decomposition task. A separate controller generates the load enable signal when required. The output of the EVD is intentionally stored once in the odd memory bank and once in the even memory bank because, for example, if the Nios II processor is reading from the odd bank, the EVD can write into the even memory bank and vice versa.
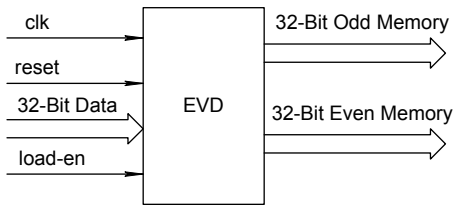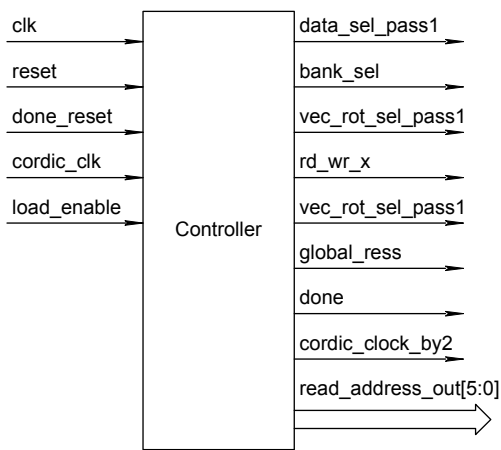
Figure 5. EVD Schematic Representation

Figure 6 is a simplified view of the controller responsible for generating control signals, as necessary.

Figure 6. Controller

The register transfer level (RTL) view of the controller is shown in Figure 7.

It is a finite state machine that uses a counter and mealy state machine. It generates the following control signals for different blocks. It is the central unit for the EVD processor. When the load_en signal comes, as long as high loading of the data takes place, as soon as load_en goes low, the controller acts. It generates:

1. bank select signal for switching the y memory bank data and address.

2. vec_rot_sel signal, which is used to multiplex between the vector and rotation modes of the complex CORDIC.

3. address signal for writing into the memory and reading from the memory.

4. Done signal, which goes high when the EVD operation is over.

Figure 7. RTL View of Controller

Figure 8 shows the complex CORDIC block and the equivalent RTL is shown in Figure 9.
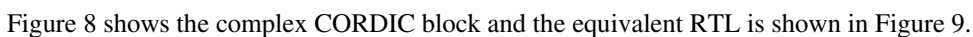
Figure 8. Complex CORDIC

Figure 9. RTL of Complex CORDIC
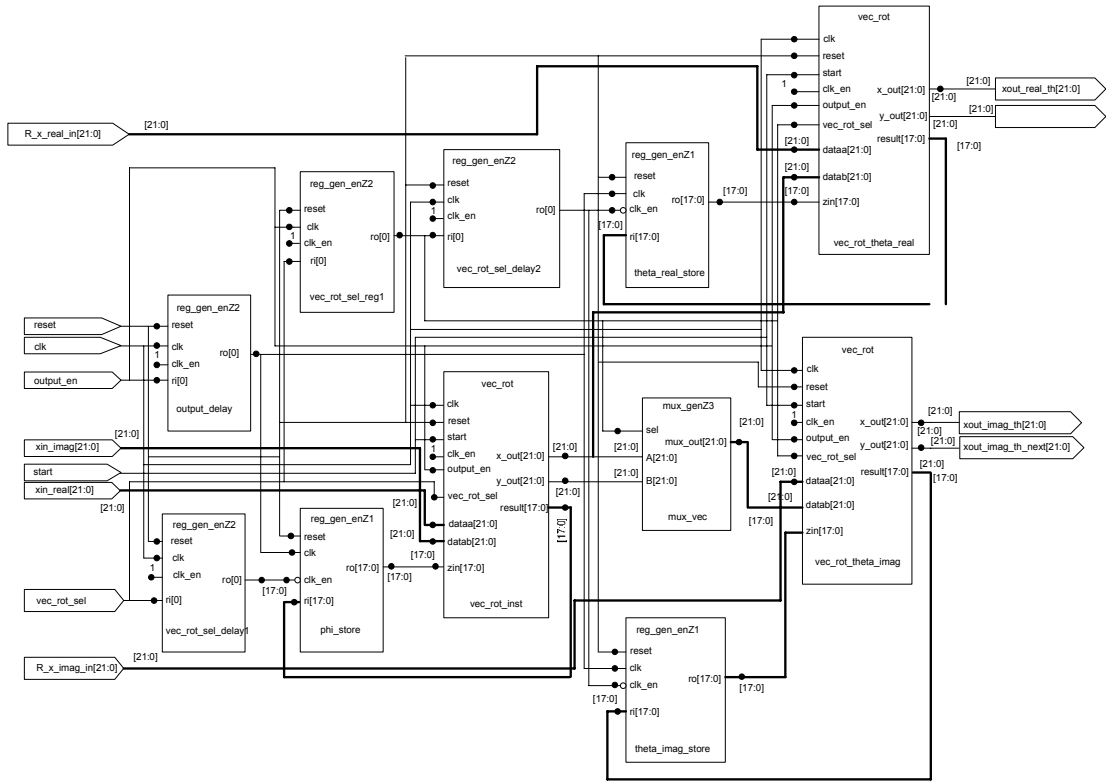
This complex CORDIC block is the key block for EVD. It comprises three CORDIC blocks and one phi-CORDIC block. These blocks are used for compensating the imaginary part of the complex input, the two theta-CORDIC ones are for the real part and the other is for the imaginary part. Because we are using a complex CORDIC in a time division multiplex manner, the angles phi and theta are stored in vector mode and these angles are used subsequently in rotation mode. The output block is important, as shown in Figure 10, for storing the final result and generating the control-signal-like interrupt when EVD is over. It also provides all necessary addresses and bus control signals for interfacing with the Nios II processor.

Figure 10. Output Block
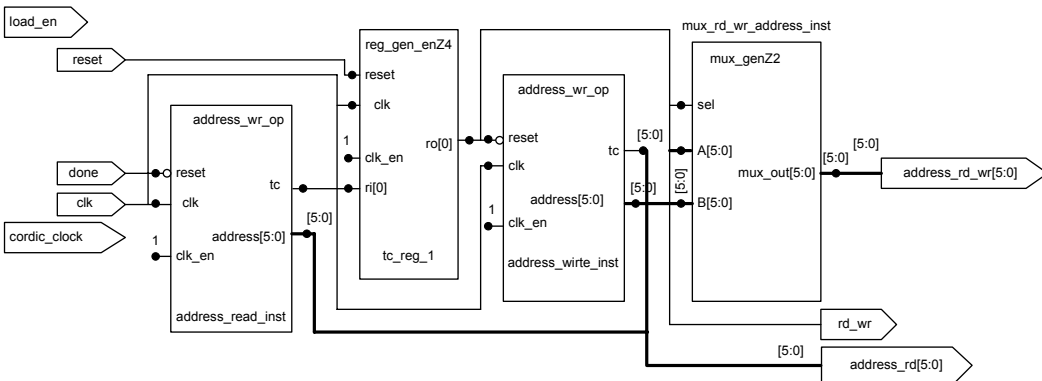
CORDIC Architecture

I have implemented CORDIC as an iterative architecture that is a direct translation from CORDIC equations.

The CORDIC rotator is normally operated in one of two modes. The first mode, called rotation mode, rotates the input vector specified angle. The second mode, called vectoring, rotates the input vector to the x-axis while recording the angle required to make that rotation.

Rotation Mode

$$x_{i+1} = x_i - y_i \cdot d_i \cdot 2^{-i}$$
$$y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i}$$
$$z_{i+1} = z_i - d_i \cdot \tan^{-1}(2^{-i})$$

$$x_n = A_n \left[x_0 \cos z_0 - y_0 \sin z_0 \right]$$
$$y_n = A_n \left[y_0 \cos z_0 + x_0 \sin z_0 \right]$$
$$z_n = 0$$
$$A_n = \prod_{i=0}^{n} \sqrt{1 + 2^{-2i}}$$
$$d_i = \begin{cases} -1, & z_i < 0 \\ +1, & \text{otherwise} \end{cases}$$

In rotation mode, the angle accumulator is initialized with the desired rotation angle. The rotation decision at each iteration is made to diminish the magnitude of the residual angle accumulator. The decision at each iteration is therefore based on the sign of the residual angle after each step.

Vectoring Mode

$$x_{i+1} = x_i - y_i \cdot d_i \cdot 2^{-i}$$
$$y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i}$$
$$z_{i+1} = z_i - d_i \cdot \tan^{-1}(2^{-i})$$

$$x_n = A_n \sqrt{x_0^2 + y_0^2}$$
$$y_n = 0$$
$$z_n = z_0 + \tan^{-1} \left(y_0 / x_0 \right)$$
$$A_n = \prod_{i=0}^{n} \sqrt{1 + 2^{-2i}}$$
$$d_i = \begin{cases} +1, & y_i < 0 \\ -1, & \text{otherwise} \end{cases}$$

In vectoring mode, the CORDIC rotator rotates the input vector through whatever angle is necessary to align the result vector with the x axis. The result of the vectoring operation is a rotation angle and the scaled magnitude of the original vector (x component of the result). The vectoring function works by seeking to minimize the y component of the residual vector at each rotation. The sign of the residual y component is used to determine which direction to rotate next.

An iterative CORDIC architecture can be obtained by duplicating each of the three difference equations in hardware as shown in Figure 11. The decision function, d_i, is driven by the sign of the **y or z re**gister, depending on whether it is operating in the rotation or vectoring mode. In operation, the initial values are loaded via multiplexers into the **x, y** and **z** registers. Then on each of the next **n** clock cycles, the values from the registers are passed through the shifters and adder-subtractors and the result is placed back in the registers. At each iteration, the shifters are modified to cause the desired shift for the operation. Likewise, at each iteration, the ROM address is incremented so that the appropriate elementary angle value is presented to the **z** adder-subtractor. On the last iteration, the results are read directly from the adder-subtractors.
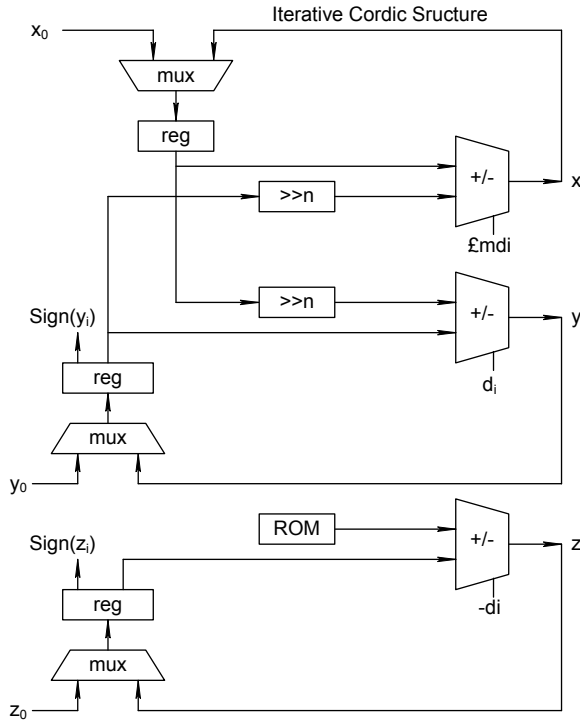
Figure 11. Equations in Hardware

Figure 12 shows a hardware-level simulation result. Hardware-level simulations were performed by the direct measurements with only the DSP part of real hardware, to efficiently evaluate the validity of the system. I used the input data made by an offline PC in advance, and obtained the results with real hardware operation. With these hardware-level simulations, we could verify the function of the digital signal processor. In this simulation, it was assumed that 2 coherent (or fully correlated) waves were impinging at 4 antennas from the DOAs of -15 and 20 degrees, respectively. And two waves were the same power and the input SNR was 15 dB. For the spectrum computation, the FFT of 256 points, including 3-spatial data of the noise eigenvector's elements (1 dimension was used for spatial smoothing) and 253 zeroes, was applied. The final result waveform output is shown in Figure 13, which shows CORDIC and EVD decomposed values.

Figure 12: Hardware Simulation Result of MUSIC (EVD) & Its Inverse (SNR 15 dB)
(4 Antenna with 2 Coherent Waves at -15 & 20 Degrees)
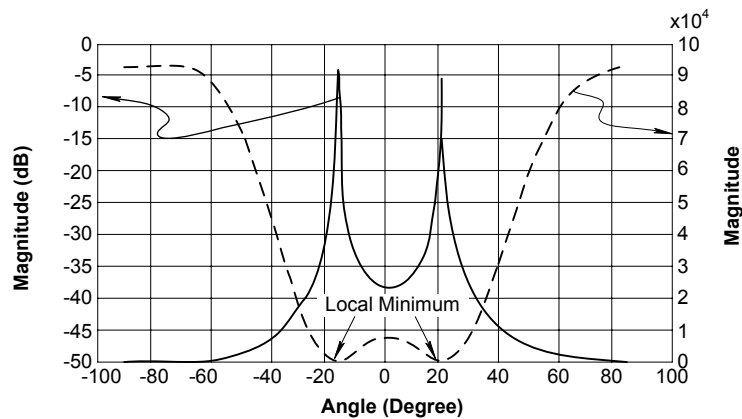
Figure 13. Final Result Waveform
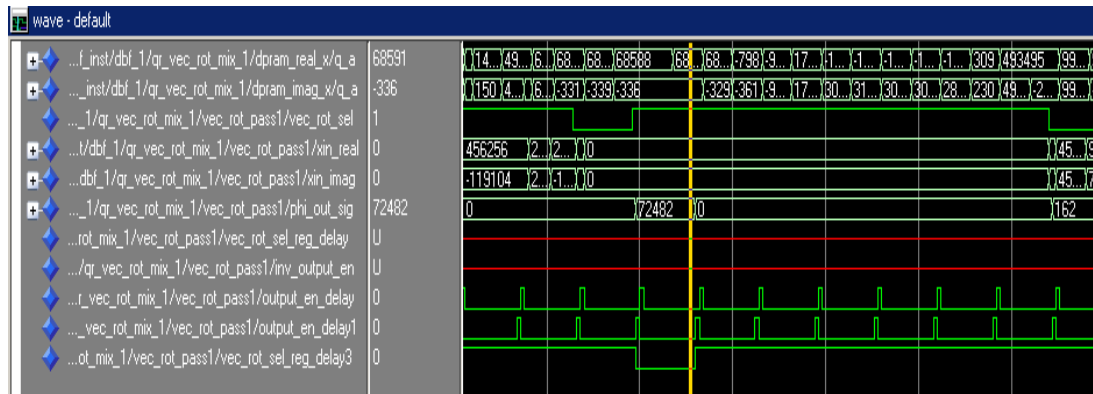
FPGA Implementation

As discussed earlier, I am going to develop the EVD, which is the IP for the system. It is the responsibility of the Nios II processor to read the values of the R and U matrix from the EVD. The Nios II processor is responsible for the two tasks namely: 1) reading the R and U matrix 2) back substitution. Back substitution involves calculating the weights and putting them back.

I developed the software for the above mentioned tasks. It takes approximately 57 μs to accomplish the specified task (4 antenna elements). This information is useful to calculate the throughput of the system. The software part also includes the interrupt service routine such that the Nios II processor will read the data and do the back substitution repetitively. The duration between each interrupt is also programmable and in synchronization with the system clock. For the above tasks I developed two peripherals, with one master and one slave each. The master reads data from memory and the Nios II processor does the necessary calculation for generating the new weights. The slave interface, which consists of a counter, is generating interrupt. The processor acknowledges the interrupt after 8 μs so that is to be taken care of while periodically generating the interrupt.

The hardware-software co-simulation in the ModelSim® tool helped me to resolve the problem, and to estimate the time taken by the processor to acknowledge the interrupt. The program developed for the back substitution is not fixed for four antenna elements, but it is a general program, applicable to any number of antenna elements.

The Avalon bus is a simple bus architecture designed for connecting on-chip processors and peripherals together into a system-on-a-programmable-chip (SOPC) solution. See Figure 14. It is an interface that specifies the port connections between master and slave components. Basic Avalon bus transactions transfer a single byte, half word, or word between a master and slave peripheral. After the completion of a transfer, the bus is available on the next clock cycle for any another transaction.

Figure 14. Avalon Bus
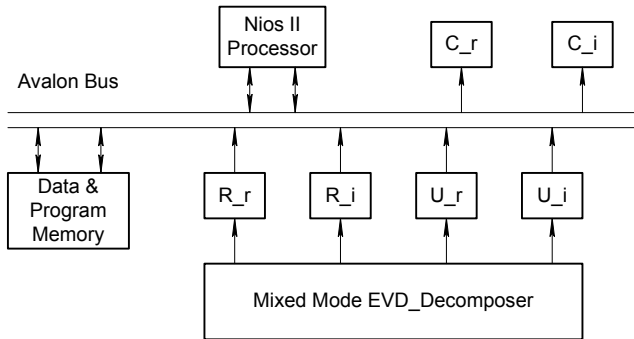
Some key features of the Avalon bus are:

- Memory and peripherals may be mapped anywhere within the 32- bit address space.

- All Avalon signals are synchronized to the Avalon bus clock, which simplifies the timing behavior of the Avalon bus and facilitates integration with high-speed peripherals.

- Separate, dedicated address and data paths provide the easy interface to on chip user logic. Peripherals do not need to decode data and address bus cycles.

- The Avalon bus automatically generates chip select signals for all peripherals, greatly simplifying the design of Avalon peripherals.

- Multiple master peripherals can reside on the Avalon bus. The Avalon bus generates the required arbitration logic.

- The Avalon bus also handles the details of transferring data between peripherals with mismatched data widths.

Device Utilization Summary

| Family	Stratix
Device	EP1S10F780C6ES
Total logic elements	8,236 / 10,570 (77 %)
Total pins	34 / 427 (31 %)
Total memory bits	61,856 / 920,448 (6 %)
DSP block 9-bit elements	8 / 48 (16 %)
Total phase-locked loops (PLLs)	1 / 6 (16 %)
Total DLLs	0 / 2 (0 %)

Test Results & Comparison

I have undergone a full design cycle of an SOPC implementation, i.e., hardware-software co-design, integration of peripherals with Avalon bus, etc. A hardware-based approach is accelerating the performance. The new hardware-based computing will solve the bottleneck of algorithmic signal processing. It is discovered that, if a CORDIC block is implemented in software only, it takes 8,600 clock cycles to complete the vectoring mode of operation as opposed to what I have achieved: 16 clock cycles to accomplish the same task in hardware. This result can motivate a CORDIC-based EVD. With respect to accuracy, if we compare the Arctan function implementation in software only, it requires approximately 20,000 clock cycles to achieve the same accuracy as the Arctan IP developed with a hardware approach. We achieved the desired functionality with the Nios II processor running at a clock speed of 50 MHz on a Stratix board. Our design of the EVD IP only takes 55 percent of the chip area on the Stratix FPGA.

Performance Comparison

Software Approach

Method	CORDIC (Cycles) ⬇	CORDIC EVD (Cycles) ⬇
Direct Equation	8,600 (172 us)	90,3000 (18 us)
Arctan Series Expansion	20,000 (400 us)	2,100,000 (42 ms)

Hardware Approach

CORDIC (Cycles) ⬇	CORDIC EVD (Cycles) ⬇
16	16 (EVD update latency will be 16 cycles) = 320 ns

Logic Elements Utilization for EVD Decomposer

Method	Logic Elements
Direct Mapping	34,055
Mapping Each Row	7,811
Mixed Mapping	4,946

Design Features

I tried different mapping architectures for optimum implementation. This section shows different mapping for seven antenna elements. Figure 15 shows direct mapping.
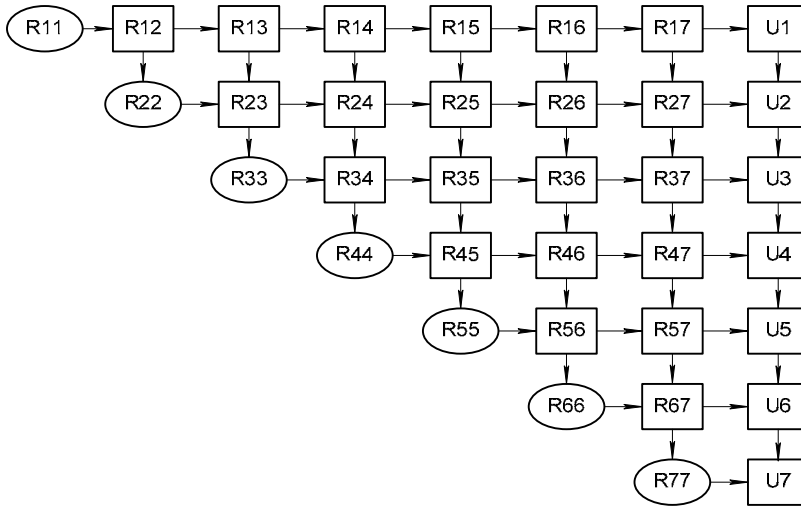
Figure 15. Direct Mapping

Figure 16 shows mix mapping and Figure 17 shows row mapping. Round blocks indicate the vectoring mode of operation. Square blocks indicate the rotating mode of operation.
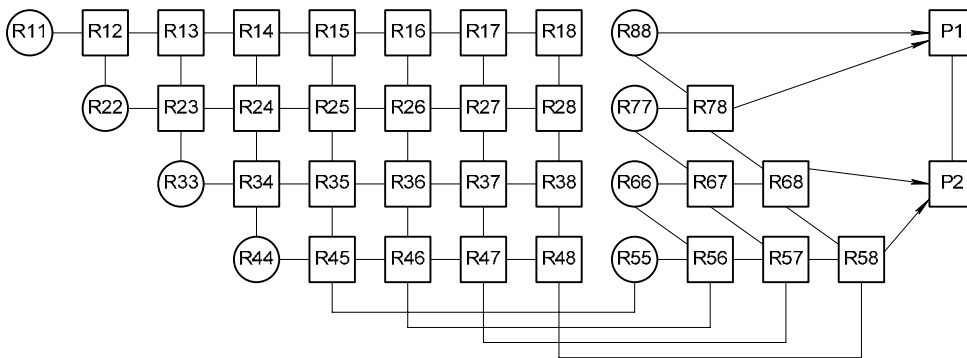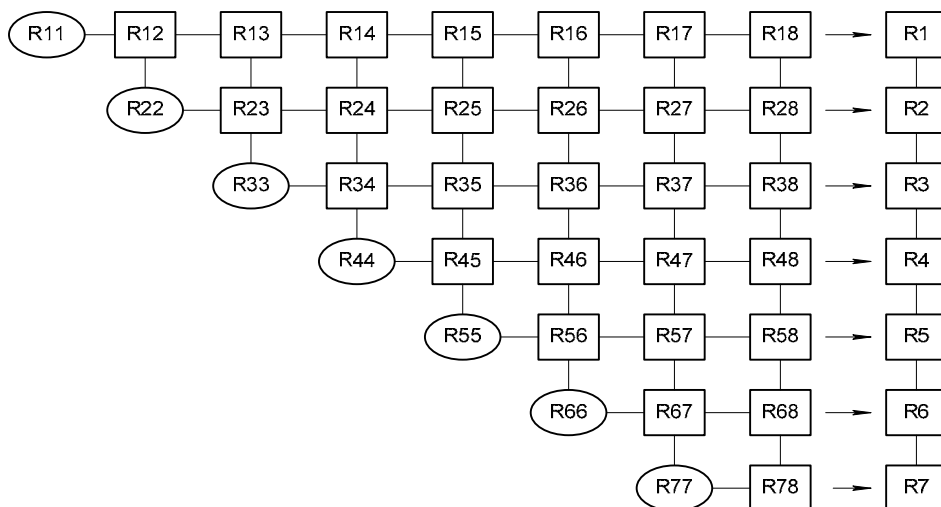
Figure 16. Mix Mapping

Figure 17. Row Mapping

Conclusion

From the above design, it is evident that for real-time implementation of computationally intensive algebraic signal processing algorithms, an FPGA-based SOPC solution is a promising, futuristic technology.

Third Prize

Nios II Soft Core-Based Full-Color LED Music Sight Light Control System

Institution: **Harbin University of Science & Technology**

Participants: **Zhong Qiubo, Gao Junfeng, and Liu Xiaoping**

Instructor: **Dong Huaiguo**

Design Introduction

Lighting sources have evolved beyond incandescent lamps. After the launch of the China Green Lights Program, new LED lighting products have attracted wide attention in the lighting and decoration industry with their energy savings, extended life, wide application, flexible control, brilliant color, and environmental efficiency. Our design is a musical landscape lamp control system with high-level simulation software, which transfers data and MP3 files to the control system through a compact flash (CF) card. Our design also includes a series of steps including lamp installation, layout, scenario data editing, simulation, preview, and so on. The product has applications in city beautification, lighting, and music integration in public places.

Controlled by a control panel or a dedicated computer, traditional landscape lamps feature only seven-color changes, in a simple way. The speed of the dedicated computer is limited: its hardware pulse width modulation (PWM) generally has only three to six paths, and can only be expanded with software modifications. Therefore, a traditional system cannot meet the requirements of high-speed data transfer. Additionally, traditional systems cannot display a smooth gradient and the jumps are noticeable with the naked eye. Therefore, we need a powerful processor to implement a soft gradient with jumps that cannot be seen by the naked eye. Considering the cost, we adopted one controller to handle several lamps. We also implemented a packet-control mechanism to handle a large number of lamps by communication between computers.

The embedded 32-bit Nios® II soft core processor helped us create a highly integrated landscape lamp control and MP3 playing system. The computing power of the Nios II processor enables simultaneous LED lamp operation based on different scenarios and music. Further, algorithms can be developed on PC using C and can be migrated to the Nios II processor, which shortens the development cycle of the whole system. Altera's SOPC Builder can help to create and deploy users' Nios II instructions, and add customized intellectual property (IP) cores to create a more powerful system-on-a-programmable-chip

(SOPC) system. Combined with the Cyclone™ FPGA, the designed product delivers a high price/performance and promises good market prospects.

Function Description

This section provides our design's functional description.

Major System Functions

Our LED musical landscape lamp control system controls 10 LED lamps (which can be increased in number, if needed). At least 256 color changes are realized through RGB color mixing and you can change five different parameters to achieve the desired effect: static, gradient, dim, bright, and flicker. You can change the duty cycle of the PWM to control a scene made up of changes to 10 lamps, form a scenario with several scenes, and then create an animation effect by playing these scenarios continuously. Simultaneously, you can play MP3 files, creating a dynamic scenario in which light changes with the rhythm of music.

System Components

The system includes a display unit, drive unit, control uni,t and data communications unit, which are controlled by the μC/OS real-time operating system. The control unit has three tasks:

■ Read lamp control data and MP3 data from the computer to the CF card memory.

■ Get and analyze data from memory, and send analyzed lamp control data to the LED lamp drive unit. Then, the multi-path PWM display unit implements the LED lamp scenarios.

■ Send MP3 data to the decoder for decoding and play via serial peripheral interface (SPI) SPI bus.

Figure 1 shows the system hardware design diagram (see the "Design Architecture" section). We used the FS embedded file system for data management, based on the real-time and multi-tasking features of μC/OS real-time operating system (RTOS). The drive unit is a self-customized, full color lamp-control intellectual property (IP) core, each controlling a lamp via PWM circuit. We use the lamp-control data to display scenario changes through the PWM port. The timer provides 10-ms interrupts, after which scenario-data analysis is carried out. The module drawn in dashed lines can be modified. Several Nios II control systems can be used for control, based on multi-computer communications, when you need to handle a large number of landscape lamps. Then, only one control module needs MP3 functionality, and the other modules may not need it. This functionality can be programmed in the SOPC Builder tool to save development costs.

Display Unit

The tricolor LED chip is the core component of the display unit. The LED is the most widely used lamp in electronic components, and tests have proven that three basic colors (red, green, and blue) can be mixed in different combinations to obtain other colors.

Control Unit

In the control unit, the μC/OS RTOS runs tasks by means of semaphore. The lamp control task software flow chart is shown in Figure 4 in the "Design Architecture" section.

Display Drive Principle

Generally, there are two ways to control LED brightness: by changing the current flowing through LED or by controlling the on/off period of the LED by the PWM. Controlling the LED working current allows for a wider range of LED brightness control. However, current control is difficult to realize in software; therefore, it is unsuitable for digital control. In contrast, the PWM method is widely adopted in digital circuits because it can be implemented easily in software. According to Talbot's law:

$$\overline{L} = \frac{1}{T} \int_0^T L(t)dt$$

in which, $\overline{L}$ is the visual brightness of cyclic change sensed by the eyes and T is the cycle. When brightness function $L(t)$ is a constant L, the visual brightness changes into $\overline{L} = \frac{t}{T}L$, when PWM actually controls the working time of the LED by changing the working time in a cycle periodically to change LED brightness.

Continuously changing the LED's working time in a cycle continuously changes the LED brightness and grey scale. Dividing cycle T by n equal periods results in n grey scales of LED. To ensure that the brightness transition is not perceived by human eyes when the LED grey scale changes (i.e., no flicker), the on-and-off frequency LED should be larger than critical frequency, and the cycle should not be longer than 0.1 - 0.2 s. Tests have shown that when the LED grey scale is 256, the mix of three basic colors will not create transitions, and human eyes can perceive the color gradient. The PWM cycle of this system is 2 ms and grey scale is 256. We can generate 256 colors controlling the three basic LED colors, and use fragment delay to control the duty cycle. The basic colors are mixed according to a certain brightness ratio, which is a certain grey scale. Different grey scales correspond with different duty cycles and different LED working time cycles.

Display Drive Unit

The display drive unit design for the 10 self-customized system peripherals is shown in Figure 2. The diagram features the design of a 30-path output port with 10 PWM controllers, respectively, for scenario changes of 10 LED lamps. The PWM circuit has two caches, back and front. The control arithmetic unit sends data that needs to be stored to the back cache of The PWM. The PWM checks whether the back cache has data to be updated, if not, it continues to read the PWM value from the front cache. We check the back cache each time playing finishes, and if data is updated, we move the data from the back cache to the front cache, and play the new data. If the PWM value is 255, the output waveform is at high logic level; if PWM value is 0, the output waveform is at low logic level. If the PWM is between 0 and 255, output is made according to the relevant duty cycle based on the fixed cycle. Figure 5 shows the software design flow.

Data Communications Unit

The data communications unit transfers lamp control data and MP3 music files from the computer to the control system via the CF card. If there are several control systems, multi-computer communications with an RS-485 serial port can be used.

Performance Parameters

The design's performance parameters are as follows:

■ The Nios II frequency required by the system is 85 Hz and the peripheral PWM's cycle is 2 ms, which is divided into to 256 parts. This scheme enables the lamp control data display using interrupt data processing that is performed every 10 ms.

■ The system relies on the µC/OS II RTOS to handle multiple tasks and makes it possible to simultaneously execute landscape lamp scenario display and MP3 music play operations.

■ The landscape lamps can support a 256-color display and five operation modes: static, gradual bright, dark, change, and flicker.

■ During a gradual change, jump phenomenon cannot be observed by the naked eye. Instead, multiple colors and gentle gradual change is displayed.

■ Fluent and clear MP3 play.

A combination of the self-defined IP core and the Nios II processor greatly accelerates operation and processing. Also, using the Nios II soft core, you can set the cycle of PWM at 2 ms and enable simultaneous operations of MP3 play and landscape lamp scenario display.

Design Architecture

Figures 1 and 2 show the hardware design. Figures 3 through 7 show the project software flows.

Figure 1. Hardware Design
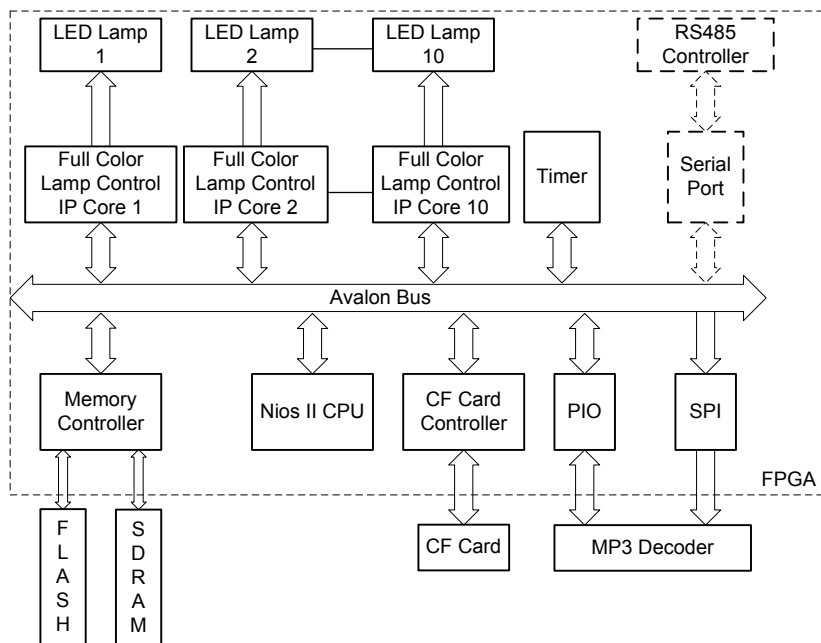
Figure 2. Full Color Lamp Control IP Core Hardware Design
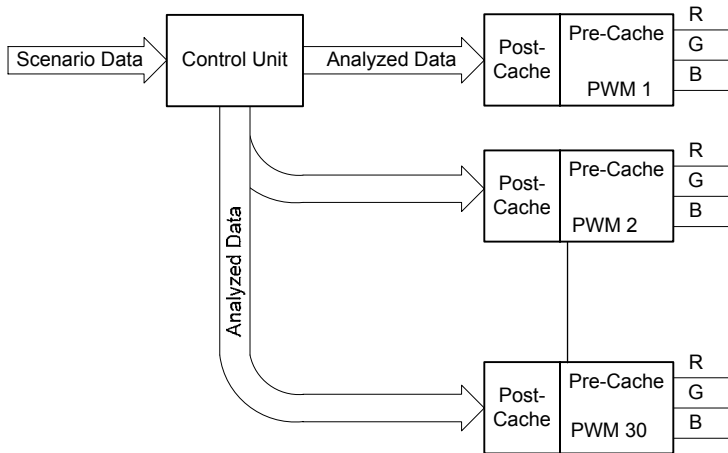
Figure 3. Timed Interruption Software Design Flow
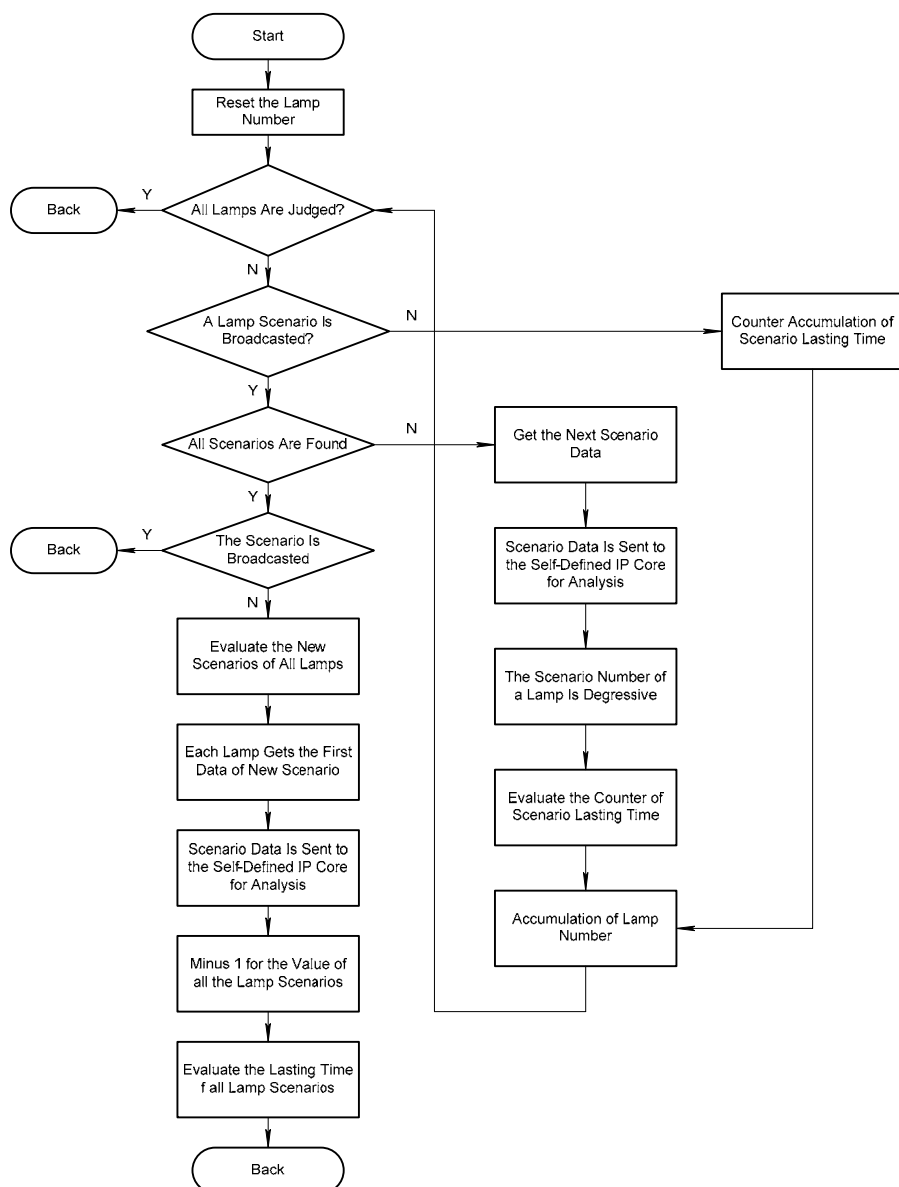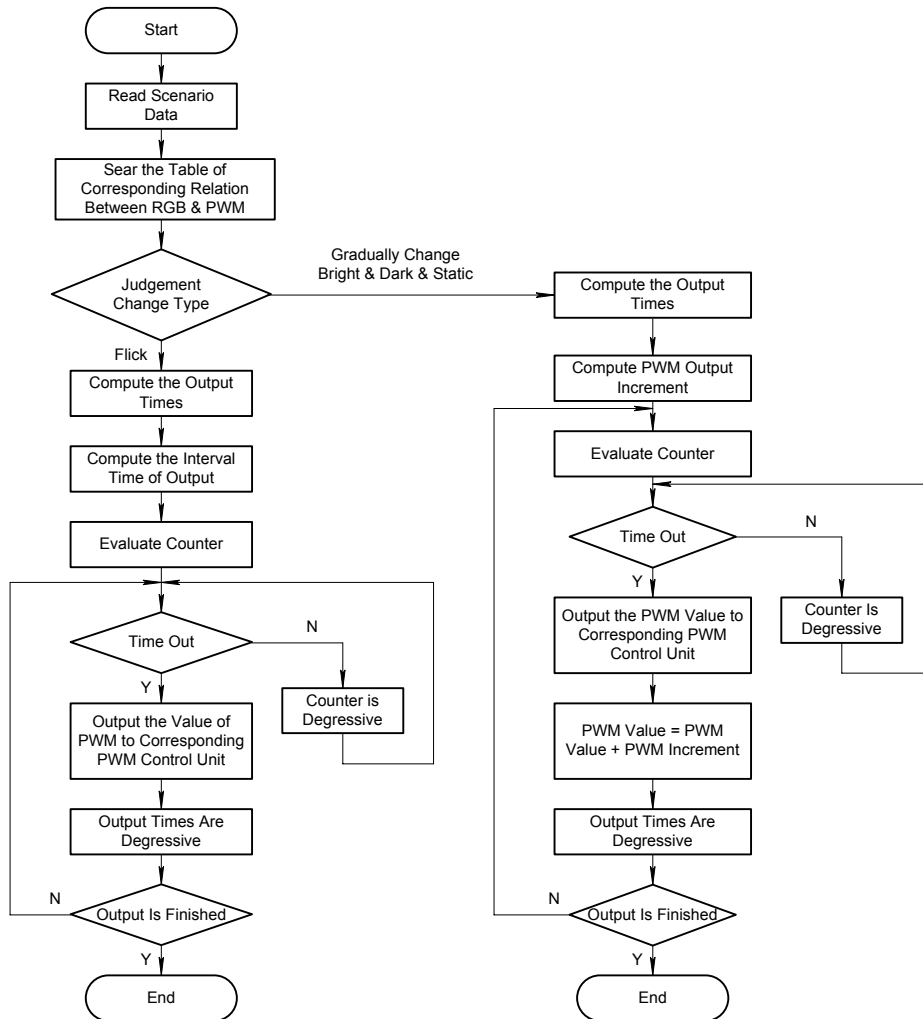
Figure 4. Software Design Flow Diagram

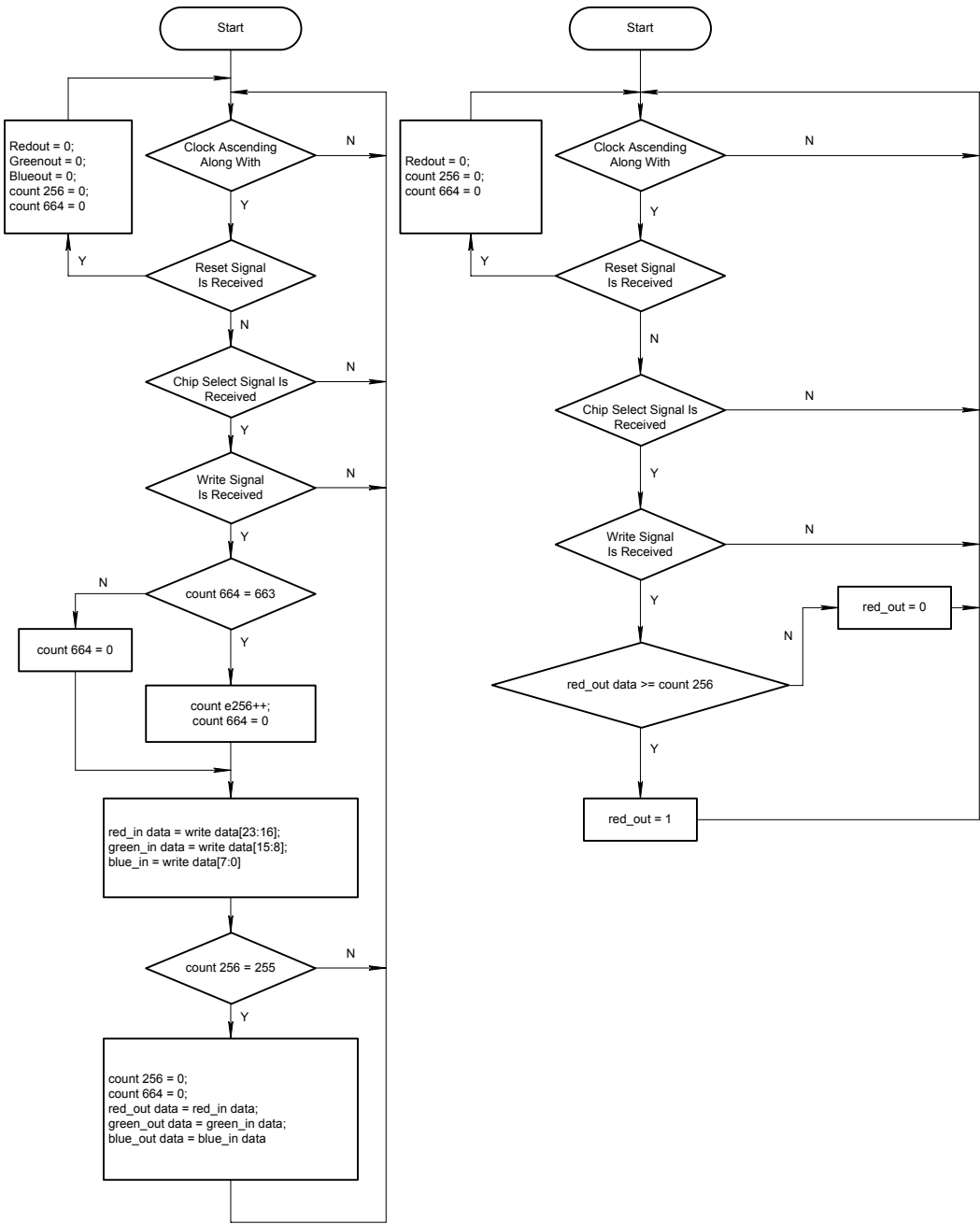*Figure 5. Peripheral PWM Software Design Flow Diagram*
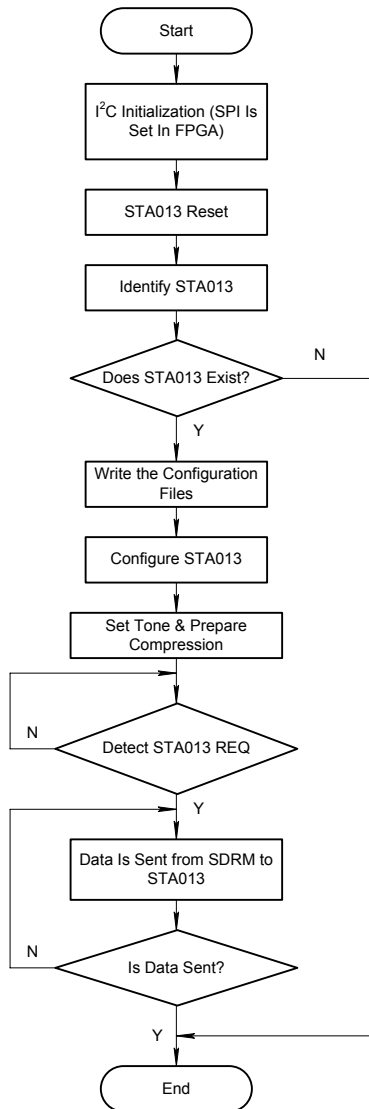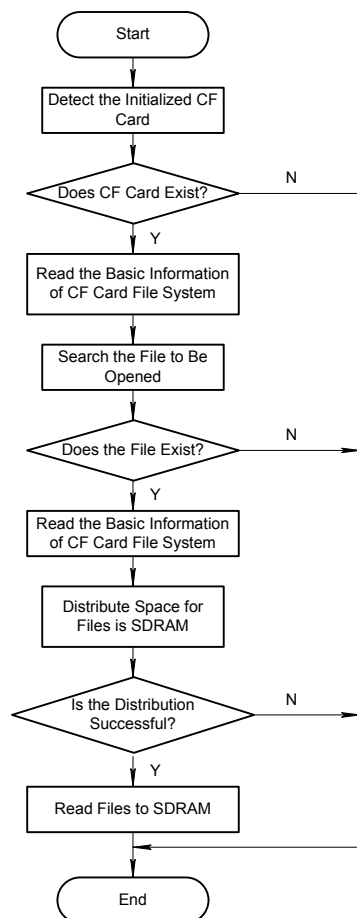
Figure 6. MP3 Design Flow Diagram

Figure 7. CF Card Read Software Design Diagram

Design Methodology

We adopted a design methodology that blends nicely with the self-defined peripherals option in the SOPC Builder tool. The system displays the landscape lamp scenario and plays MP3 files under the semaphore control mode of μC/OS-II RTOS in the Nios II integrated development environment (IDE) after download. Our design comprises two modules: hardware and software.

Hardware Design

We extended the system hardware by adding the STA013 MP3 decoder and D/A converter for playing MP3 files on the Altera® Cyclone II EP2C3. We implemented the lamp control on the UP3 development board to promote the application of Cyclone II FPGA, which is the most cost-effective device that offers the best price-performance ratio among competing devices. We implemented all system functions on the EP2C3 device. Additionally, we applied the ULN2803 power drive to control the voltages used in the LED lamp display (see the Appendix for the circuit schematic).

Full-Color Lamp Control IP Core Design

We used Verilog HDL to design the self-defined peripheral full-color lamp control IP core's control unit, which implements a 10-lamp, self-defined IP core controller with 30 PWM circuits. The software design flow is shown in Figure 5. The cycle of PWM is set to 2 ms, as shown in Figure 8.
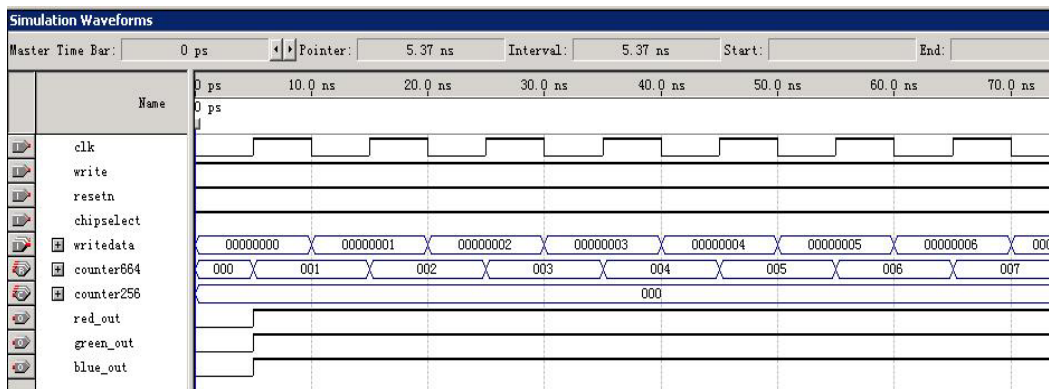
Figure 8. PWM Timing Diagram
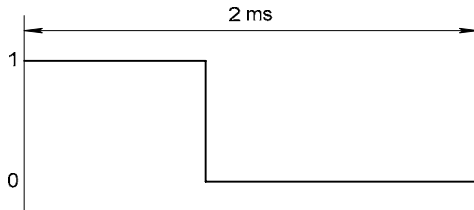
Function Simulation

After PWM design, we carried out functional simulation as shown in Figure 9.

Figure 9. PWM Functional Simulation

The simulation variables in the oscilloscope display are described as follows:

■ `clk`: clock signal.

■ `resetn`: PWM reset signal.

■ `chipselect`: PWM chip select signal.

■ `write`: PWM write signal.

■ `writedata`: data written to PWM.

■ `red_out`: red corresponding output signal in PWM.

■ `green_out`: green corresponding output signal in PWM.

■ `blue_out`: blue corresponding output signal in PWM.

■ `counter256`: one cycle is divided into 256 parts for computing the duty ratio of each PWM cycle.

■ `counter664`: each part comprises 664 clock cycles for computing whether the count is over or not.

MP3 Design

We used the I^{2}C bus to control the STA013 device. In this way, we were able to transfer MP3 data from SDRAM to STA013 through SPI, which is set using the SOPC Builder tool as shown in Figure 10. We added four PIO interfaces in SOPC Builder to connect with the SDA, SCL, DATA_REQ, and RESET pins. The PIO connected with SDA is set as a bidirectional port.
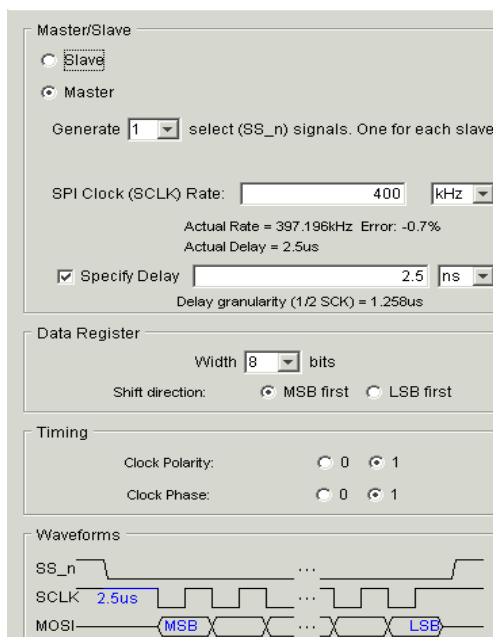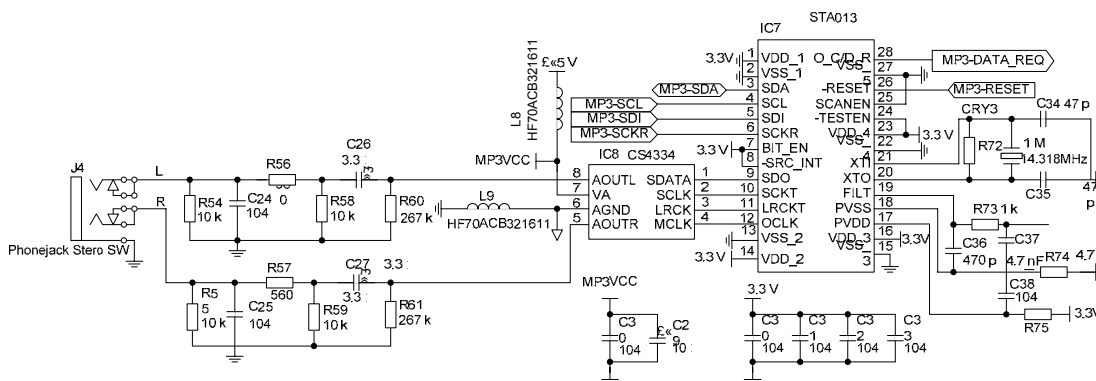
Figure 10. SPI Setting

Figure 11 shows the MP3 decoding circuit.

Figure 11. MP3 Decoding Circuit Schematic

SOPC Builder Configuration

After finishing the IP core design and simulating, we used the SOPC Builder tool to configure the whole system. The settings are shown in Figure 12.

Figure 12. SOPC Builder Settings

Compiler

After configuring all the required system parts, the SOPC Builder tool assigns pin definitions with the Quartus® II development tool and then compiles. See Figures 13 through 17, which show the Compiler output.

Figure 13. Compiler Analysis Report
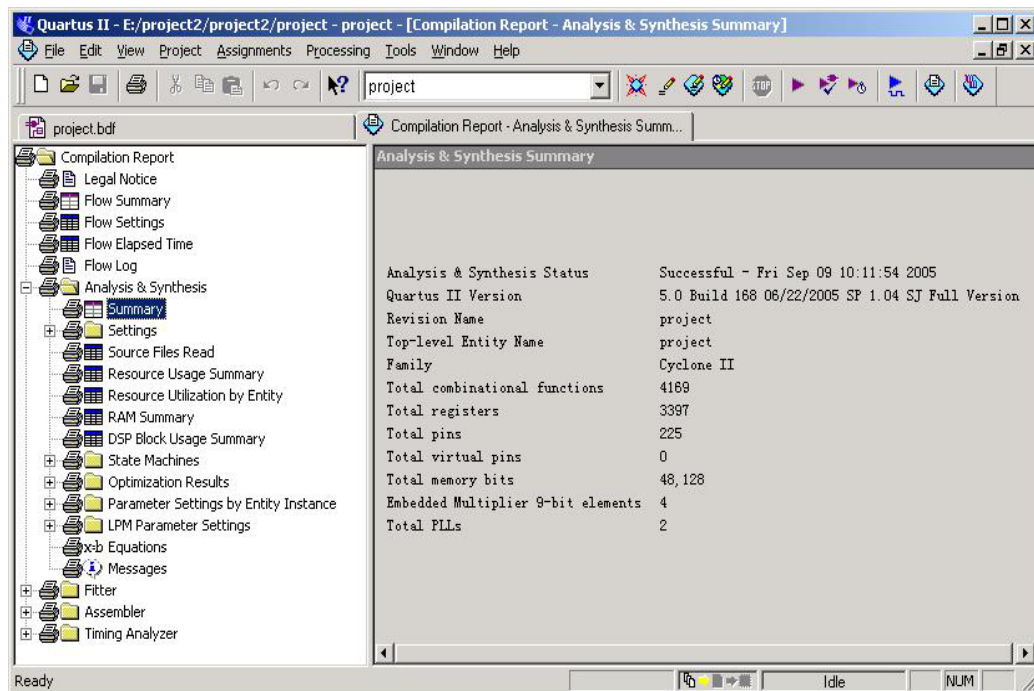
Figure 14. Assembler Report
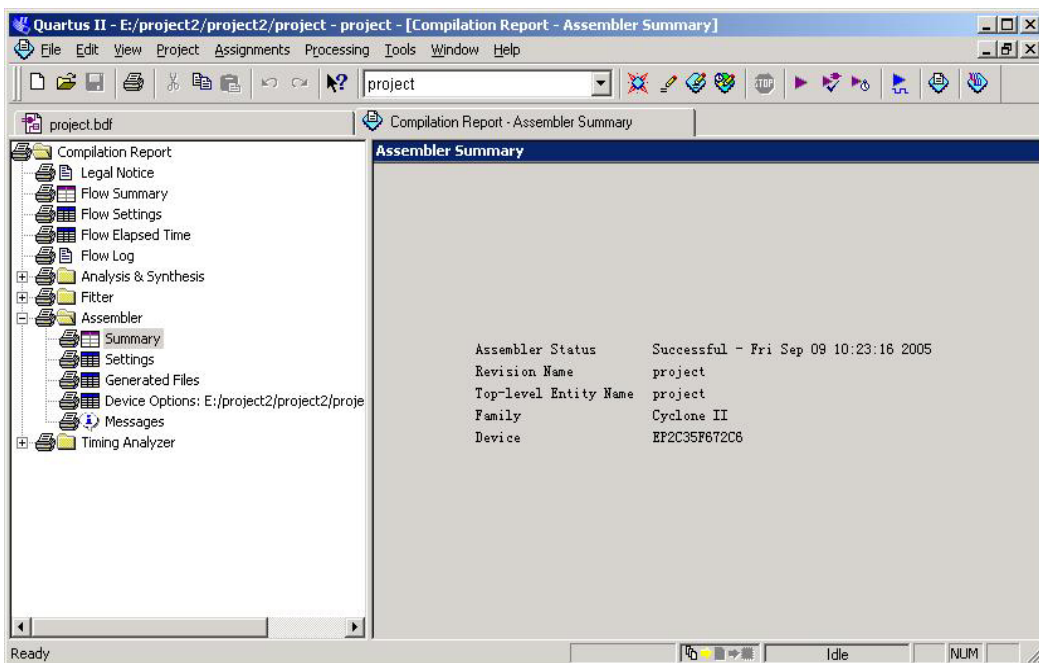
Figure 15. Fitter Report
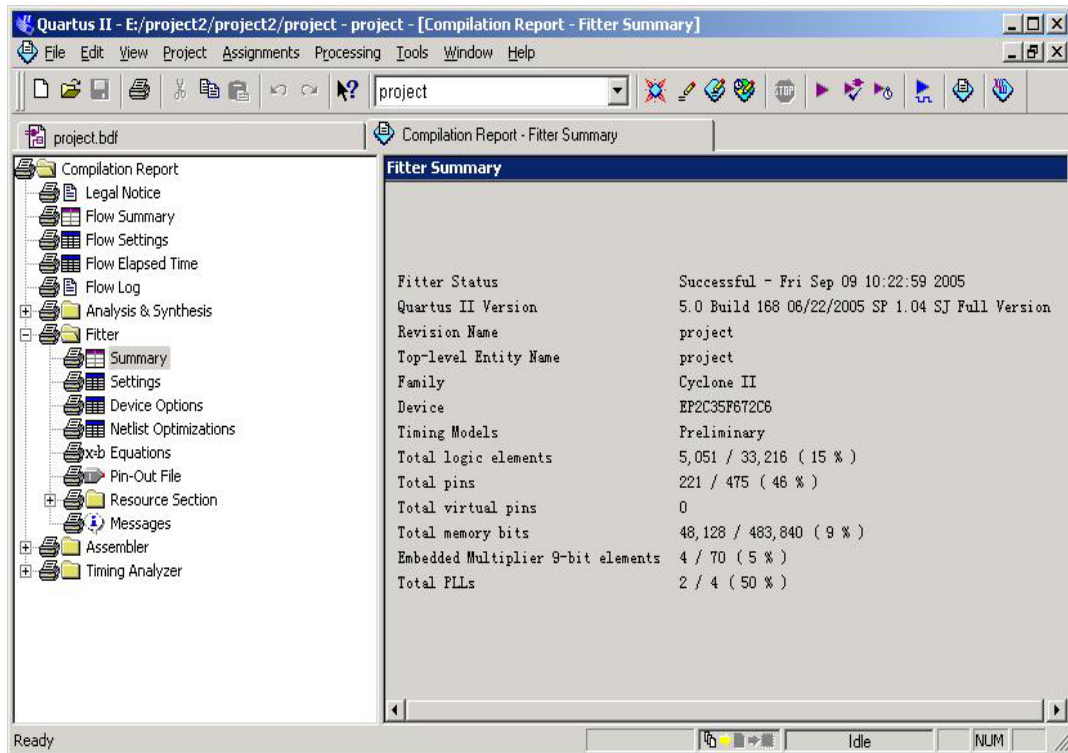
Figure 16. Flow Report
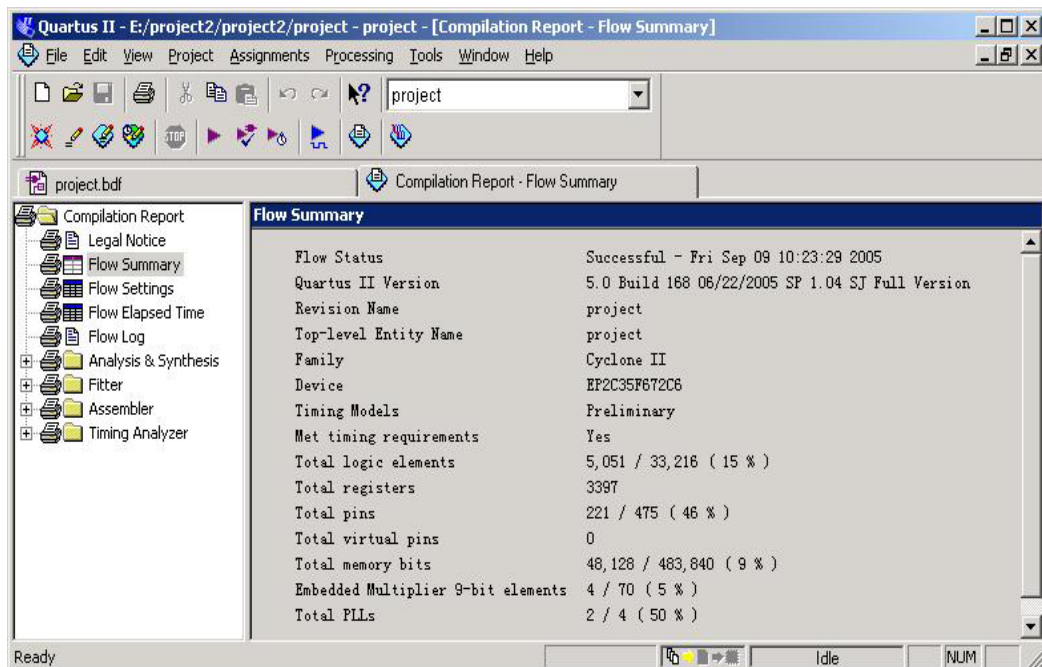
Figure 17. Timing Analyzer Report
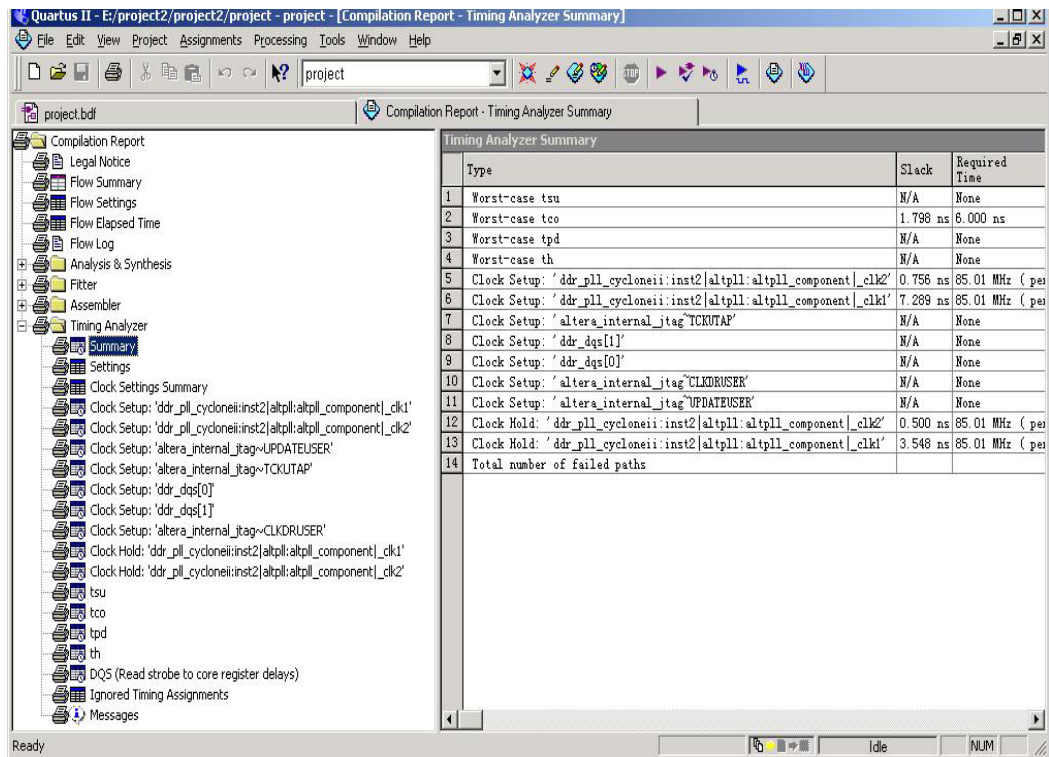
Software Design

The basic design of the software module is to send out the play and data read tasks accurately and in a timely manner using the µC/OSII RTOS.

The first task is to read the CF card and send its data and MP3 files to SDRAM for the next two tasks. In our design, we used the Quartus II version 5.0 compact flash core as the interface between the CF card and the Nios II processor. This routine uses two pointers, *MP3data and *pwmdat, to assign space for data and MP3 files on the SDRAM. We designed a small file allocation table (FAT) file system for CF card reading. This system:

- Does not support long file names.

- Does not support the FAT12 file format.

- Sets data and MP3 files in the root directory of the CF card.

- Does not support writing into the CF card (the data in CF card can be written from by PC with reader/writer).

We defined three data structures: BPB, file directory entries, and FAT. The specific definitions are as follows:

```
typedef struct
{
    unsigned char  Type;           // file format type
    unsigned char  StartLBA;       //BPB start sector
```

104

```
    unsigned char  SectorsPerCluster;    //sectors per cluster
    unsigned char  LShift;               //shift number of SectorsPerCluster
    unsigned short  SectorsBeforeFAT;    // reserved FAT number
    unsigned char      FATs;                 //FAT number
    unsigned short  FAT16RootEntries;    // number of root directory entries
    unsigned long  TotalSectors;         //total sector number
    unsigned short  SectorsPerFAT;       //sector number per FAT
    unsigned long  FAT32RootStartCluster; //start cluster of root directory when the
file format is FAT32

} FS_TBPB;
typedef struct
{
    unsigned long  StartLBA;   //start sector of file allocation table
    unsigned char  LShift;     // shift number of file format type
    unsigned long  DataStartLBA; //start sector of data area
} FS_TFAT;
typedef struct
{
    unsigned char  Attrib;   //file attributes
    unsigned long  StartCluster;   //file start cluster
    unsigned long  StartLBA;       //file start sector
    unsigned long  CurrentCluster;   //current cluster
    unsigned long  CurrentLBA;     //current LBA
    unsigned long  Offset;          //system reserved
    unsigned long  Length;         //file length
} FS_TFile;
```

Refer back to Figure 6 for a detailed software flow.

At system initialization, we invoke CF card initialization function IDE_initialize() to determine whether the CF card exists or not. If the CF card exists, we read the basic information of the FAT file system, such as the file format the CF card has adopted, start sector of root directory, and data area. We invoke the FS_SearchFile (char *FName, FS_TFile *R, unsigned char dir) function to search the file to be read and then assign a buffer for the file with a pointer. Because SDRAM has enough space, the file data can be totally read into SDRAM, which is the file size in SDRAM. One sector is read each time until all data is moved into SDRAM. The key to FAT file system design is to get data of the next cluster after reading the current one. In this design, we defined the function, FS_GetNextCluster(unsigned long Cluster). We read the whole cluster chain into an array when opening a file. Although this routine occupies some space on the SDRAM, the search of cluster in future will not read the FAT table. This is because the function slows down system speed.

The second task is to display the scenario file and to receive the scenario data of different lamps as well as search the PWM values R, G, B binary-coded according to Table 1. This task judges the changing modes, such as gradual change, bright, dark, and static, in the same control mode. The flicker mode is handled differently.

Formatting of scenario data comprises five bytes: the last byte indicates the address information and the first four bytes are shown as follows:

Front color	Back color	D15 D14 D13 D12 D11 D10 D9 D8	D7 D6 D5 D4 D3 D2 D1 D0

Front color and back color separately occupy one byte; D15 in the third byte is the marker bit of FLICK, following two situations that may occur in terms of D15's value:

■ D15=0, gradual change, bright and dark as well as static, D14……..D0 indicate the lamp on lasting time.

■ D15=1, flick mode, D14……..D9 indicate the flicker time, D8………D0 indicate the lamp flicker lasting time.

For gradual change, bright, and dark as well as static, the data increment PWM_D sent to PWM per cycle is computed using the following formula:

```
PWM_D =(PWM_BACK_COLOR – PWM_FRONT_COLOR)/(LASTING_TIME/10ms)
PWM_FRONT_COLOR  PWM value of front color
PWM_BACK_COLOR   PWM value of back color
LASTING_TIME        lasting time of scenario
```

10 ms is the cycle period time of the PWM.

Interrupt time scenario is TIMES = LASTING_TIME /10ms, in which the increment of static mode is 0. Accordingly, based on the principle that PWM_D sends TIMES to PWM per 10 ms, we can achieve the control of gradual change, bright and dark as well as static.

FLICK (flicker), the lasting time of such a scenario can be obtained in terms of the following formula LASTING_TIME:

```
LASTING_TIME=FLICK_TIMES*FLICK_TIME
FLICK_TIMES    flicker times
FLICK_TIME     flicker lasting time
```

Thus, this routine delivers PWM_FRONT_COLOR and PWM_BACK_COLOR by turns to PWM using FLICK_TIME as the interval, and after delivering FLICK_TIMES, ends the control of flicker function.

Table 1. Look-up Table of RGB Binary Value & Corresponding PWM Value

Value	R	PWM	G	PWM	B	PWM
	000	0	000	0	00	0
	001	36	001	36	01	85
	010	72	010	72	10	170
	011	108	011	108	11	255
	100	144	100	144		
	101	180	101	180		
	110	216	110	216		
	111	255	111	255		

When the 10-ms interrupt is received, the processed scenario data is delivered to the self-defined peripherals for display (the design flow is shown in Figure 3). All lamps are judged in the interrupt cycle to determine whether the system needs to play a scenario completely. If the present scenario is totally played out, data for the next scenario is collected and delivered to the control unit of the self-defined IP core for analysis and processing. If the scenario is still incomplete, the interrupt routine returns.

The third task is to play MP3 format music. To fulfill this task, tone quality has to be taken into account. It is interesting to observe how landscape lamps appear to change with anamorphic music, and therefore we adopted a secure hardware based decoding solution. We have used the STA013 decoding chip and the CS4334 D/A converter. Refer back to Figure 6 for the detailed design flow.

When the task is activated, it first initializes the I²C bus, and then invokes the sta_Init() function to initialize STA013. This initialization includes resetting STA013, verifying ST013, and writing the configuration files, which are loaded in STA013_UpdateData[] array. The following operation configures STA013 and set tone as well as prepares data for compression. We start by invoking the decode control function sta_SendToDecoder (unsigned short len, unsigned char MP3_data[]) for

decoding. When the DATA_REQ pin of STA013 is high, it indicates that STA013 needs new MP3 data to compress and play. By querying the sixth bit of the status register in SPI core we judge whether status register TXDATA requires new data (or whether previous data was delivered to STA013); if this bit is low, we write new MP3 data to TXDATA. The received data from STA013 is decoded and played.

The difficult problems in the above routines are in the decoding time sequence setting and phase-locked loop (PLL) configuration. The data input/output accords a certain standard of time sequence. For instance, here we set the SPI frequency clock to 400 kHz so that the music can be played smoothly. If this frequency is too high or too low, it will affect the tone quality and music rhythm. An improper setting can even cause cacophony. The PLL may impact the operating clock of the on-chip components. Therefore, we had to be careful with the PLL setting, because a wrong setting of PLL may generate sampling drift and consequently cause anamorphic music.

Design Features

The system uses the Nios II soft core combined with an FPGA to control LED lamps. At least 256 lamp colors can be displayed in our system with full dynamic effects based on five changing modes: static, gradual changing, bright, dark, and flicker. Simultaneously, we can change the lamps' colors along with MP3 music rhythm. Our system can be used in applications that integrate decoration lamps with music in public places. Because of the nearly 200 MIPS capacity of the Nios II soft core, no color leap appears in the gradually changing LED color. By deploying the user-defined peripherals, the system can quickly perform data analysis of lamp control, and allows for easy expansion of peripherals. Using the SOPC Builder tool, it was easy for us to delete and add the MP3 expansion circuitry and the user-defined peripherals. By taking full advantage of the FPGA, we were able to develop PWM IP core, expand multi-PWM circuits in peripherals based on the design requirement. After optimization of a design, the system' logic units are much reduced when compared with the purely traditional embedded, bus-based designs.

Conclusion

With more than two months of learning, we have been able to appreciate the Quartus II tool's powerful design functions and flexibility. The system provided us with many common IP cores in SOPC Builder, which helped in our design work and enabled us to add our self-defined IP cores and commands to meet the customer specific requirements. This approach made our design more flexible, especially the self-defined commands, which when added to existing 256 colors, are sufficient to meet most customer requirements. Additionally, the Quartus II software provided the functions from the start of the design to completion. These functions are easy to handle in the GUI. As for software development, the Quartus II software also integrates the Nios II IDE. We were able to finish the program design and download the final design using the Nios II IDE GUI.

Additionally, when we compared Quartus II version 4.2 and Quartus II version 5.0, we noticed that with Quartus II version 5.0 we can save system compilation time. Previously, even a small design modification needed the whole system to be recompiled. However, the Quartus II software version 5.0 provides optimized compilation, which only compiles the modified parts each time. As for the system design, we know about the advantages of FPGA and soft core design methods, especially during product development. With these methods, we can shorten the development cycle, reduce development risk, and get the early-to-market advantage. The Quartus II tool provided us with abundant materials for development, which are easy to understand, and each user reference emphasized a design principle by illustrating it with diagrams and code samples. By studying these materials, we were able to develop our own systems easily and wrote programs based on our requirements.

Appendix

Schematic Circuit of LED Lamp

Schematic Circuit of LED Drive

Third Prize

3-D Accelerator on Chip

Institution: **Donga & Pusan University**

Participants: **Young-Hee Won, Jin-Sung Park, Woo-Sung Moon**

Instructor: **Sam-Hak Jin**

Design Introduction

Recently, consumers are becoming interested in cellular phones and portable game devices that play 3-dimensional (3-D) games. It is difficult for mobile device processors to compute 3-D graphic operations because they require a lot of arithmetic operations and most mobile device processors cannot process them. To solve this problem the processor chip must incorporate 3-D accelerator units to reduce the computing time. In this project, we developed a 3-D graphic display system that quickly computes 3-D graphic operations by including a hardware 3-D accelerator in the chip, and creating applications in it.

Using the Nios® II processor to develop 3-D graphic displaying system makes it possible to integrate the main processor and 3-D accelerator in one chip. The system is smaller, faster, and more stable than when using a hard-core processor chip and a separate 3-D accelerator chip.

Our 3-D graphic display system is based on OpenGL ES version 1.1, which is a royalty-free, cross-platform application programming interface (API) for full-function 2-D and 3-D graphics on embedded systems, including handheld devices, appliances, and vehicles. It is a well-defined subset of desktop OpenGL, creating a flexible and powerful low-level interface between software and graphic acceleration. OpenGL ES Pipeline is based on OpenGL 1.3 Pipeline, which includes geometry processing, rasterization, fragment processing, and frame buffer operations. Programmers who have used the desktop OpenGL can easily develop application programs for OpenGL ES. Therefore, our system can also be used to develop 3-D graphics applications, such as 3-D games.

Function Description

Our system is based on OpenGL ES version 1.1; therefore, the system offers same interface. If the application program makes vertex data by using offered functions, it computes the 3D operations, such as rotation or transfer 3-D vertexes, lighting, clipping, and so on. Additionally, the software makes final vertex data that is viewed by the camera set to see the 3-D world. See Figure 1.

Figure 1. Pipeline Flow of OpenGL ES in the System
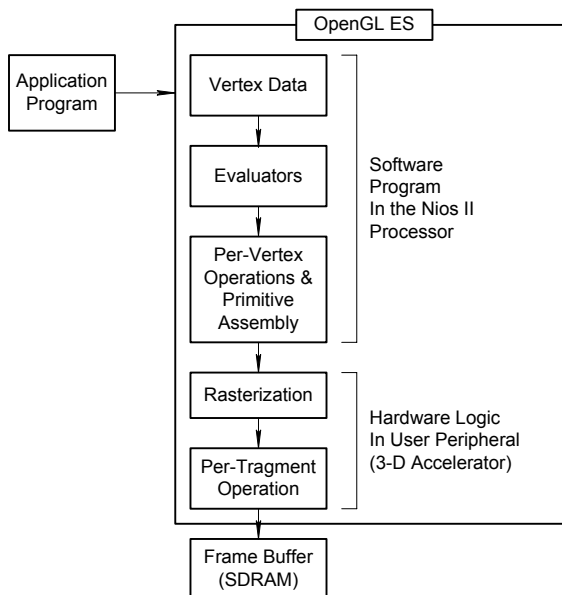
The software part of the 3-D graphic processing function delivers the positions of the three points of the triangles and the color of polygons to the 3-D accelerator, which is the hardware part of the 3-D graphic processing function. The 3-D accelerator makes the x, y, and z positions of the inside of the polygons to fill up it. It also writes the color data of points to the exact address of the frame buffer, which is placed in SDRAM. The thin-film transistor (TFT)LCD controller module independently reads the frame buffer and displays the 3-D graphics on the TFT LCD repeatedly. The application programmer programs the applications using the OpenGL ES library and the system displays the 3-D graphic output of application.

Performance Parameters

The most important performance parameter of a 3-D graphic display system is the speed of computing and displaying a frame of 3-D graphics. The general unit of 3-D graphic display speed is frames per second (fps). We tested the speed of our system displaying a cube spinning on an x, y diagonal axis, as shown in Figure 2.

Figure 2. Test Display
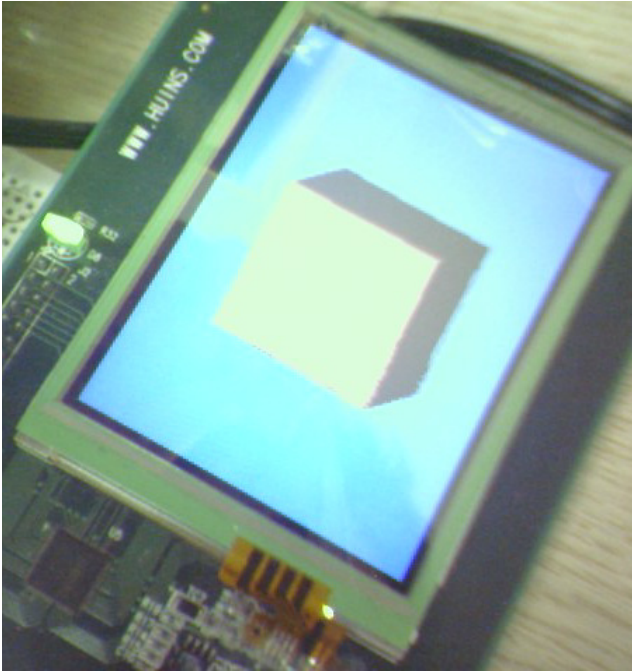
Design Architecture

Figure 3 shows the top-level block diagram. The Nios II processor connects to ext_ram_bus, 3-D accelerator, 7-segment controller, SDRAM controller, TFT-LCD controller, and so on. The ext_ram_bus module is a tristate Avalon® bus bridge that connects the Nios II processor to flash memory and SRAM, which are the instruction memory and data memory, respectively, used to run the Nios II processor.

Figure 3. Top-Level Block Diagram
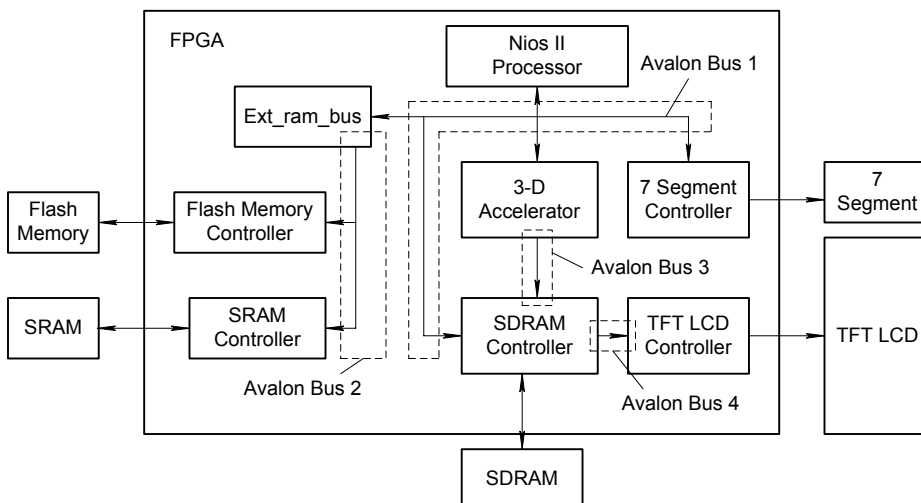
The 3-D accelerator (Renderer) is a slave of Avalon bus connected to the Nios II master (see Figure 4). The Renderer receives the polygon data and starts rasterization. After rasterization, the outline pixel position data of the triangle of polygon is restored in the span buffer, which uses the Altera® FPGA's internal memory. The structure of the span pixel data is x (8-bit integer) and z (32-bit fixed-point real number), and the address is y (9-bit integer).

Figure 4. Block Diagram of Rendering Module
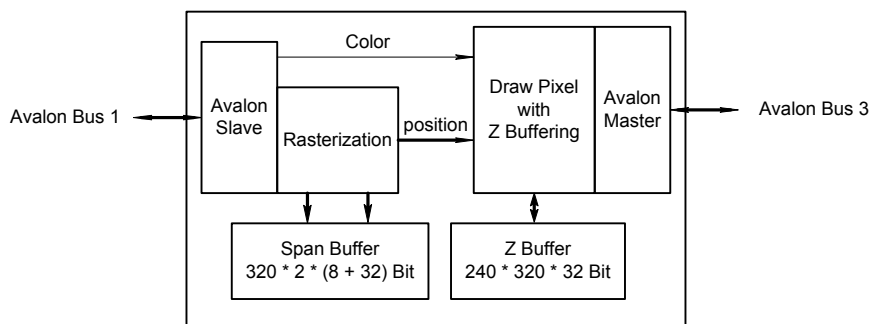
After rasterization, the position data of y, the right side of x, the left side of x, and the z position of each side is delivered to the draw pixel module. This module makes the x and z position data of the pixels between the received pixel of the right side and left side and loads the z position data of that (x, y) position to compare the depth that is drawn and to draw. If the pixel to draw is closer to the camera than the pixel drawn previously, the module writes color data of the pixel to the frame buffer in SDRAM through the Avalon bus connection and the z position data to the z buffer. If it is further than the currently drawn pixel, it is ignored. See Figure 5.

Figure 5. The Rasterization Process
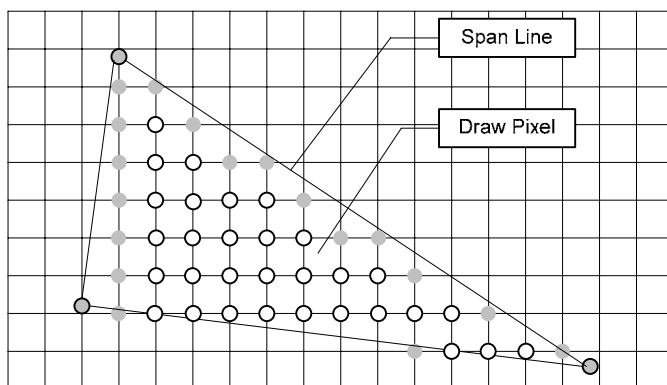
The LCD Control module displays the data of the frame buffer to the screen of the TFT LCD. It receives the base address of the frame buffer from the Nios II processor. The SDRAM Control module, a master of Avalon bus synchronized to the VGA Controller's sync signals, repeatedly reads data from the frame buffer in SDRAM. See Figure 6.

Figure 6. LCD Controller Block Diagram
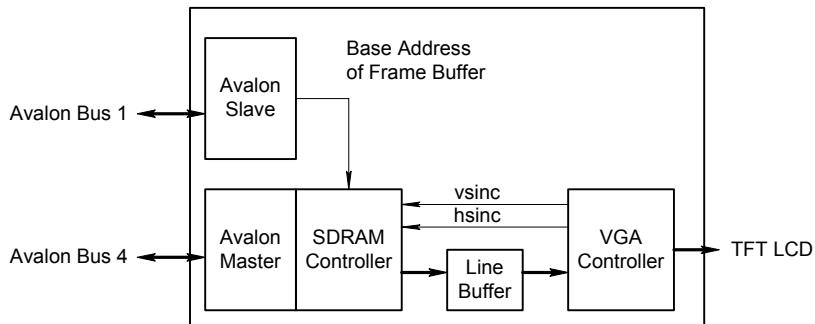
While the software operates, it uses a custom instruction to multiply or divide fixed-point real number type data.

The software is very complex and can be changed by the application program. We developed the OpenGL ES library by modifying the details of open-source OpenGL ES code to fit into our system. The functions have same interface with OpenGL ES that the well-defined subset of desktop OpenGL has. Therefore, an application program using our functions operating in the system processes the 3-D graphic operations as shown in Figure 1, in software and hardware.

Design Methodology

We developed the LCD Controller first and displayed a sample image to the TFT LCD with a simple software program. Next, we developed the OpenGL ES software library by modifying the open-source code. We reprogrammed the process of writing pixel data to the frame buffer because our system uses memory differently. We changed computations using floating-point numbers to fixed-point numbers, and some functions, including those concerning lighting, were changed to fit our system.

After we checked the 3-D graphic frame displaying of our test application without the 3-D accelerator, we developed the 3-D accelerator module. We changed the software functions to deliver data to hardware instead of computing it with the main processor.

We used a 50-MHz system clock to reduce compilation time. After we checked the system's operation, we set a phase locked loop (PLL) to generate a 100-MHz clock to run the final system.

The main CPU is the Nios II processor. The design uses the Nios II /f processor, flash memory, and RAM controllers, which are connected to the Nios II processor by a tristate Avalon bridge, ext_ram_bus. Additional modules, such as a timer, JTAG UART, etc., run and debug the Nios II processor. The 7-segment PIO Controller displays the system speed.

We wrote the 3-D accelerator module (Renderer) in VHDL with Avalon bus slave and master signals. We used the **New Component** menu option in SOPC Builder to add the VHDL code. The slave side of the module connects to the Nios II processor, and the master side connects to the SDRAM controller, as shown in Figure 7.

Figure 7. SOPC Builder System

We used the **New Component** menu option in SOPC Builder to include the LCD Controller module and specify its signals. The signals between the LCD Controller and Avalon bus became the pins of the Nios II module. The slave side of the module is connected to the Nios II processor and the master side is connected to the SDRAM controller. We developed the LCD Controller module in VHDL, created a Symbol File of it, and connected it to the Nios II module. See Figure 8.

Figure 8. Top-Level Circuit
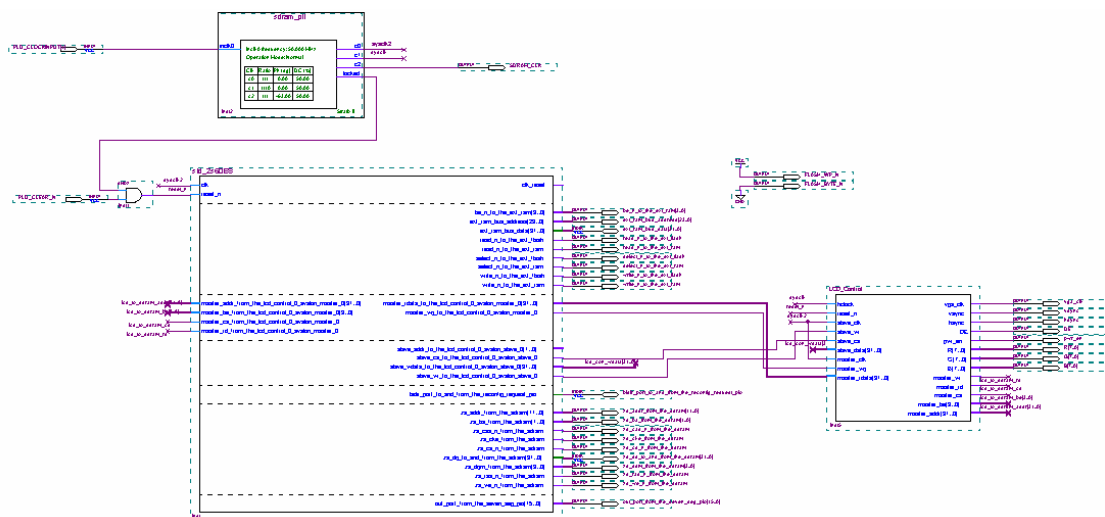
Figure 9 shows the LCD Controller block association in the Quartus® II software. We created each block in the LCD Controller module using VHDL, and used the Quartus II MegaWizard® Plug-In Manager to develop the line buffer, which uses the FPGA's internal memory.

Figure 9. LCD Controller Block Diagram

We also used the Quartus II MegaWizard Plug-In Manager to create the 3-D accelerator line span buffer, and the multiplication and division operations.

While the software operates, it performs many multiplication and division operations of fixed-point real numbers. These numbers are defined as:

```
#define __GL_X_MUL(a,b) ((__gl_x)((((__gl_ll)(a))*(b))>>__GL_X_FRAC_BITS))
#define __GL_X_DIV(a,b) ((__gl_x)((((__gl_ll)(a))<<__GL_X_FRAC_BITS)/(b)))
```

Where `__gl_x` is a 32-bit, fixed-point read number, `__gl_ll` stores 64-bit data, and `GL_X_FRAC_BITS` means the bits under point, defined as 16. We created custom instructions for these numbers as shown in Figures 10 and 11.

Figure 10. Fixed-Point Division Custom Instruction

Figure 11. Fixed-Point Multiplication Custom Instruction
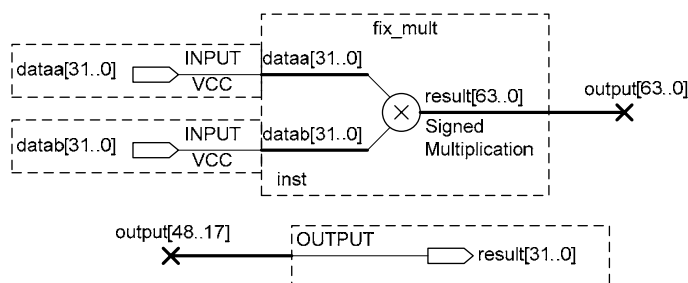
We used the Quartus II MegaWizard Plug-In Manager to create the multiplication and division processors and added these modules to the design as custom instruction with the definition:

```
#define __GL_X_MUL(a,b)   (__gl_x)__builtin_custom_inii( 0, a,b)
#define __GL_X_DIV(a,b)   (__gl_x)__builtin_custom_inii( 1, a,b)
```

Using custom instructions makes the system faster because the custom instruction only uses 1 clock cycle instead of more than 2 required without the custom instruction.

Design Features

We developed the 3-D graphic display system with the Nios II processor and our 3-D accelerator in one chip. This design makes it possible for small devices, such as cellular phones or portable game devices, to display 3-D graphics faster in a smaller device.

Because the 3-D accelerator is in the same chip as the main processor, the system size is smaller than the same functional system using a hard-core processor chip, separate 3-D accelerator chip, the signal connections between the two chips, power sources, memories, and so on.

The design is also faster than using variable chips, because the access to SDRAM is controlled by the Avalon bus without complex control signal protocols. Additionally, custom instructions make it more capable. The system has reduced wires and power source regulators by using a synchronized clock from the PLL blocks. Therefore, the system can run without noise or clock sync errors, making it much more stable than using variable chips.

Conclusion

We developed the 3-D graphic display system with the Nios II soft-core processor and our designed 3-D accelerator in one chip. This system allows small devices, such as cellular phones or portable game devices, to display 3-D graphics faster in a small size. Although the system is not currently fast enough for a consumer product, we could develop additional hardware modules for various operations and change the software processes to hardware processes to divide the processing loads in each section of pipeline, thereby increasing the computing speed.

The Nios II processor is very useful for embedded system engineers. With it, we were able to integrate a processor and our designed hardware in one chip, making the system smaller, faster, and more stable. In particular, using Nios II custom instructions makes the system much more efficient than using hard-core processors or only FPGAs.

The Avalon bus was very easy to create and use to connect blocks in the FPGA. For example, with just a few mouse clicks in SOPC Builder made it possible to connect blocks, even several master blocks accessing several slave blocks. In our system, the Nios II processor, 3-D accelerator, and LCD controller are all masters of the SDRAM controller, and using the Avalon bus makes the operation smooth, without error or crossing data.

The Quartus II software, which includes SOPC Builder and MegaWizard Plug-Ins, made it very easy to include and connect several current designs. It reduced our development period and made the design process less complex.

First Prize

Cryptographic Algorithm Using a Multi-Board FPGA Architecture

Institution: **Indian Institute of Technology, Chennai**

Participants: **G. Ananth and U.S. Karthikeyan**

Instructor: **Dr. V. Kamakoti**

Design Introduction

Information security has assumed a significant importance in today's world, especially because minor breaches can lead to major risks in the fields of national security and other e-commerce applications and transactions. This necessitates implementing cryptographic algorithms in hardware to achieve better security and faster response as opposed to any software implementation. A promising solution combining high flexibility with the speed and physical security of traditional hardware is the FPGA.

Implementing cryptographic algorithms requires the generation of random numbers that can be then used in any algorithm to derive the keys for carrying out a secure transmission. Keeping this in mind, a design was created implementing a multi-board architecture using two Altera® boards. One board constantly generates random numbers using a data encryption standard (DES) random bit generator and at the same time keeps polling its input port for requests by another program designed to receive random numbers. The second board contains a design that implements the RSA algorithm and incorporates the reception of random numbers on the fly by means of hardware interrupts. On receiving the random number, the second board sends an acknowledgement back to the first board to continue the process. The designs (implemented as peripherals) on each board make use of a Nios® embedded processor for communicating and exchanging data between the driver program and the peripheral.

The FPGA device family chosen for implementing the RSA algorithm is Altera's APEX™ 20KE device family. APEX devices are high-density FPGAs that allow complex designs to be implemented on a single device. The target device was an APEX 20K EP200EFC484-2X and the design files were written in Verilog HDL, while compilation, synthesis, fitting, placement, and routing was carried out using the Quartus® II software. The Nios development board provided a hardware platform to immediately start developing embedded systems based on Altera APEX devices. The Nios development board was preloaded with a 32-bit Nios embedded processor system reference design.

The highlight of this project is the efficacious use of interrupts for inter-board communication and the use of numerous custom peripherals for both random number generation and implementing the RSA algorithm and hardware acceleration.

Functional Description

The functional description of this project is depicted through the flow diagram below. It is essentially comprised of two flows. One flow is the generation of the random number using the DES-based random bit generator. See Figure 1.

Figure 1. DES-Based Random Bit Generator
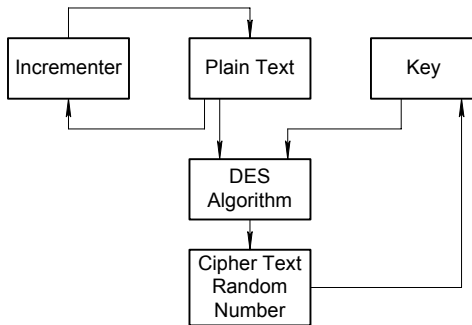
The flow diagram for the RSA implementation is as follows:

1. A request is sent from the RSA module to fetch a random byte.

2. On receipt of a request, a random byte is sent by the DES random bit generator that continuously polls a designated port for the request for random bytes.

3. It also signals READY after sending the random byte, and indicates readiness to accept the next request from the device (FPGA running RSA).

Figure 2. Handshaking Between RSA Module & DES Random Bit Generator
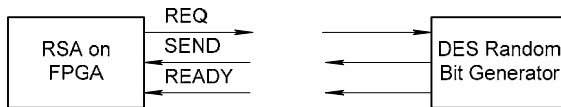
Performance Parameters

The performance parameters entail results obtained for the random number generator as well as for the RSA implementation.

Random Number Generation

A comparative overview of the results obtained for all nine designs implemented during the course of this project is tabulated below:

■ Resource Requirements

■ Session Yield

■ Bit Generation Speed/Throughput

■ Lines of Verilog HDL Code

Resource Requirements

Number	Item	Logic Elements (LEs)	Pins	Memory Bits	Phase-Locked Loops (PLLs)
1.	Maximum available resources on Altera APEX 20K EP200EFC484-2X	8,320	376	106,496	2
2.	Resources utilized by the standard Nios processor with essential peripherals	2,641	111	26,496	0

Resource Utilization per Design

Number	Design	LEs	Pins	Memory Bits	PLLs
1.	PLL-Based TRBG	4,737	113	26,496	2
2.	Ring Oscillator-Based TRBG	2,772	119	26,496	0
3.	Modified LILI-II PRBG	4,114	111	26,496	0
4.	Nonlinear Combiner Model-Based PRBG	7,419	111	26,496	0
5.	Nonlinear Combiner Model (Enhanced With Memory)	8,312	111	26,496	0
6.	Nonlinear State Filter Model-Based PRBG	8,312	111	26,496	0
7.	DES-Based PRBG	4,969	111	26,496	0
8.	DES-ALFG-Based PRBG	8,043	111	26,496	0
9.	BBS-Based PRBG	6,449	111	26,496	0

Session Yield

Number	Design	Yield per Session
1.	PLL-Based TRBG	362 bits
2.	Ring Oscillator-Based TRBG	> 246 Kbits
3.	Modified LILI-II PRBG	> 40 Kbits
4.	Nonlinear Combiner Model-Based PRBG	> 2 Mbits
5.	Nonlinear Combiner Model (enhanced with memory)-Based PRBG	> 2 Mbits
6.	Nonlinear State Filter Model-Based PRBG	> 40 Kbits
7.	DES-Based PRBG	> 526 Kbits
8.	DES-ALFG-Based PRBG	> 1.87 Mbits
9.	BBS-Based PRBG	> 40 Kbits

 Although the session yield of the true random bit generator is based on the PLL and implemented as the Nios peripheral appears to be low, it is conjectured that it will perform far better as a stand-alone device. Moreover, 362 bits per session may be considered adequate for session key initialization vector requirements.

Bit Generation Speed/Throughput

The bit generation speed/throughput shown below has been worked out in terms of the clock cycles taken to generate one random bit, based on the implemented algorithm. Its translation into throughput has been done for a clock speed of 33.3 MHz.

Number	Design	Clock Cycles per Random Bit	Throughput for a Clock at 33.3 MHz
1.	PLL-Based TRBG	< 3	> 11.1 Mbps
2.	Ring Oscillator-Based TRBG	-	0.59 Mbps
3.	Modified LILI-II PRBG	5	6.66 Mbps
4.	Nonlinear Combiner Model-Based PRBG	3	11.1 Mbps
5.	Nonlinear Combiner Model (Enhanced with Memory)-Based PRBG	3	11.1 Mbps
6.	Nonlinear State Filter Model-Based PRBG	4	8.325 Mbps
7.	DES-Based PRBG	140	7.611 Mbps
8.	DES-ALFG-Based PRBG	Initially 140 subsequently amortized to < 140	> 7.611 Mbps
9.	BBS-Based PRBG	520	64.038 Kbps

Lines of Verilog HDL Code

Number	Design	Lines of Verilog HDL Code
1.	PLL-Based TRBG	275
2.	Ring Oscillator-Based TRBG	275
3.	Modified LILI-II PRBG	177
4.	Nonlinear Combiner Model-Based PRBG	189
5.	Nonlinear Combiner Model (Enhanced with Memory)-Based PRBG	322
6.	Nonlinear State Filter Model-Based PRBG	324
7.	DES-Based PRBG	1,143
8.	DES-ALFG-Based PRBG	1,191
9.	BBS-Based PRBG	601

The RSA algorithm was implemented as separate peripherals performing the following operations:

■   Random number receiver

■   Multiplicative inverse calculator

■   Modular exponentiation calculator

After implementing these peripherals, all were combined to form a RSA integrated design working through a C driver program, which passed inputs and outputs between the various peripherals, in order. Due to the paucity of the space on the FPGA in terms of the number of LEs, only a 32-bit RSA integrated algorithm was implemented. Space on FPGA (number of LEs) permitting, this design can easily be scaled up.

Random Number Receiver

The random number receiver was implemented to receive one byte of random number through the external pins on the board. The peripheral consumed the following resources.

Family	APEX 20KE
Device	APEX 20K EP200EFC484-2X
Total LEs	2,783/8,320 (33%)
Total Pins	121/376 (32%)
Total Memory Bits	26,496/106,496 (24%)
Total PLLs	0/2 (0%)

The total time taken for compilation, synthesis, fitting, placement, and routing of this peripheral was 4 minutes and 42 seconds.

Multiplicative Inverse

This peripheral was implemented to compute the secret key using an extended Euclidean algorithm. Since the algorithm implemented required division operations to compute the remainder and quotient at every step, it consumed a lot of resources. In simulation, this algorithm was tried and tested up to 128 bits, but in hardware, it could be implemented only up to 48 bits. Total time taken for compilation and synthesis, fitting, placement, and routing was 12 minutes, 31 seconds. The compilation report for this peripheral was:

Family	APEX 20KE
Device	APEX 20K EP200EFC484-2X
Total LEs	6,524/8,320 (78%)
Total Pins	111/376 (29%)
Total Memory Bits	26,496/106,496 (24%)
Total PLLs	0/2 (0%)

As can be seen from the compilation report, a 48-bit implementation itself consumes 6,524 LEs. Hence, if used along with other peripherals such as the exponentiator and the random number receiver, no other peripheral would be able to fit on the FPGA. Therefore, only a 32-bit implementation was used in the RSA integrated implementation.

Exponentiator

This peripheral was implemented to carry out the following tasks:

■ Primality check using Fermat's Theorem

■ Encryption

■ Decryption

The algorithm implemented was the Montgomery exponentiation algorithm, which in turn uses the Montgomery multiplication algorithm for the intermediate steps. The modular multiplication was implemented using the systolic array architecture, which is quite resource efficient. In simulation, a 512-bit exponentiation was implemented, however, in hardware only a 128-bit exponentiation was possible. The total time taken for compilation, synthesis, fitting, placement, and routing was 11 minutes, 36 seconds. The compilation report for this peripheral was:

Family	APEX 20KE
Device	APEX 20K EP200EFC484-2X
Total LEs	6,971/8,320 (83%)
Total Pins	111/376 (29%)
Total Memory Bits	26,496/106,496 (24%)
Total PLLs	0/2 (0%)

The peripheral consumed 6,971 LEs, hence a higher implementation such as 256- or 512-bit exponentiation was not possible, despite a resource-efficient architecture. The 256-bit exponentiator

itself required 10,277 LEs, while a 512-bit exponentiator required 17,459 LEs. In the RSA integrated implementation, only a 32-bit exponentiator was included, since two other peripherals, the random number receiver and the multiplicative inverse, were also required to be fitted on the same chip.

RSA Integrated

The RSA integrated peripheral implements the complete RSA algorithm primitive, which includes the following operations:

■   Receiving random numbers.

■   Primality checking

■   Computation of multiplicative inverse.

■   Computation of modular exponentiation.

All of the above operations were implemented as separate peripherals and fitted on the same chip. A C driver program then interacts with all the peripherals and passes appropriate values between them. This requires that all the peripherals are instantiated correctly in the C program. The total time taken for compilation, synthesis, fitting, placement, and routing was 13 minutes, 8 seconds. The compilation report for this integrated design was:

Family	APEX 20KE
Device	APEX 20K EP200EFC484-2X
Total LEs	6,984/8,320 (84%)
Total Pins	121/376 (32%)
Total Memory Bits	26,496/106,496 (24%)
Total PLLs	0/2 (0%)

The RSA integrated implementation of 48 bits, excluding the random number receiver and the primality checker, consumed 8239 LEs, which is almost 99% of the total available LEs on the board. Hence the final implementation was scaled down to 32 bits to accommodate the random number and the primality check peripherals.

Execution Time & Throughput

The RSA algorithm has been implemented with a modulus of 32 bits, with a multi-board architecture also included to receive the random numbers on the fly. However, this makes the measurement of the execution time difficult since it involves an interrupt-driven mechanism. By simulation, the execution time and the throughput for only the encryption/decryption can be approximated for a clock speed of 33 MHz. In the case of RSA, the encryption and decryption is carried out by modular exponentiation, and for a modulus of 32 bits, it took 1,555 clock cycles, which gave a throughput of 0.68 Mbps.

Design Architecture

The system architecture entails two parts, namely:

■   Generation of random numbers using the DES random bit generator

■   Implementation of RSA using the random number generated by the above method

PRBG Based on Block Cipher Techniques

This section describes how the random numbers were generated.

DES Random Bit Generator

The data encryption standard (DES) was developed by an IBM team around 1974 and adopted as a U.S. national standard in 1977. Since that time, many cryptanalysts have attempted to find shortcuts for breaking the system. It is defined by the American standard FIPS 46-2. We wish to encipher a 64-bit plaintext message block under the 56-bit key, to produce a 64-bit ciphertext message block c=Ek(m). Decipherment, or recovering plaintext from ciphertext, is denoted m = Dk(c).The plaintext message block m is subjected to an initial permutation P, and the result is broken into two 32-bit message halves, m0 and m1. Intermediate message halves,m2,...m17 are then created in sixteen rounds according to the procedure described below. Finally, the 64-bit ciphertext c is generated by applying the inverse permutation IP^{-1} to the two message halves m17 and m16.)

The plain text and intermediate message halves m{0},m{1},m{1},...,m{17} are related as follows:

 m{i+1} = m{i-1} XOR f(k{i},m{i}) , i= 1, 2,...,16

Here k is the secret 56-bit key and i is the number of the round (from 1 through 16). Also, k{i} (round key) is a selection of 48 bits from the 56 bits of k. This selection, or key schedule, depends on the round number i. Now we describe the substitution function f. There are eight S-boxes, S{1},...,S{8} described in the standard. Each S-box is a table lookup, using six bits as input and providing four bits as output. For each S-box, say S{j}, six consecutive bits are selected from the 48 bits of k{i}, namely bits 6j - 5, 6j - 4, ...,6j. Also, six consecutive bits are selected from m{i}, namely bits (4j - 4, 4j - 3,..., 4j + 1) mod 32. The mod 32 is shorthand for the convention that for j = 1, the bits are 32, 1, 2, 3, 4, 5, and for j = 8 the bits are 28, 29, 30, 31, 32, 1. Two adjacent S-boxes share two message bits. For instance, S{1} uses message bits 32, 1, 2, 3, 4, 5, while S{2}, uses message bits 4, 5, 6, 7, 8, 9, and they share bits 4 and 5. (Key bits are not shared among S-boxes on one round.) S{8}, and $S{1}, are considered to be adjacent because they share message bits 32 and 1. The six key bits and the six message bits are XORed together bitwise, and the resulting six bits are used as input for a table lookup.

That is, the six inputs to S-box S{j} at round {i} are:

m{i}[4j -4] XOR k{(i)}[6j -5],

m{i}[4j -3] XOR k{(i)}[6j -4],

m{i}[4j +1] XOR k{(i)}[6j]

or, written another way,

[4j - 4, 4j - 3, 4j - 2, 4j - 1, 4j, 4j + 1]XOR k{i}[6j - 5, 6j - 4, 6j - 3, 6j - 2, 6j - 1, 6j].

Each of the eight S-boxes implements a different table, each with 26 entries of four bits each. These tables are described in the standard. The eight S-boxes together put out 8 x 4 = 32 bits. These bits are permuted according to a permutation P that is fixed for all rounds i. The resulting 32-bit quantity is the value of f(k(i)},m{i}).

In summary, the 64-bit message undergoes a permutation IP to produce two 32-bit message halves m{0} and m{1}. Then we compute the 32-bit quantity f(k{(1)},m{1}) and XOR that quantity with m{0}, to produce m{2}. We use this new quantity m{2} to compute f(k{(2)},m{2}) and XOR that quantity with m{1}, to produce m{3}. We continue in a like fashion until m{16} and m{17} have been computed. These two message halves are interchanged and then subjected to the permutation IP^{-1}, to produce the ciphertext c. Decryption is easily accomplished by a user in possession of the same key k. First, one applies the permutation IP to c, to produce the message halves m{17} and m{16}. Next, one computes f(k{(16)}, m{16}) and XORs that quantity with m{17} to recover m{15}. Recalling that m{17} = m{15} XOR f(k{(16)}, m16), we have m{17} XOR f(k{(16)}, m{16}) = [m{15} XOR f(k{(16)}, m16)] XOR f(k{(16)}, m16) = m{15}, because of the identity (A XOR B) XOR B = A. Similarly, one computes m{14} = m{16} XOR f(k{(15)}, m{15}), and continues in like fashion until one has computed m{1} and m{0}. Applying IP^{-1} to the pair (m{0}, m{1}), one recovers the plaintext message m.

DES RBG Design

DES was implemented in ECB mode for any arbitrarily selected IV, using a secret key. The ciphertext emerging after each round of encryption was thereafter used as the key for the next round of encryption, while simultaneously incrementing the plaintext once in counter mode. This fundamental operation is iteratively executed for the desired number of times, with an interrupt being raised after each execution. The random bits are read back by the driver as 32-bit words, and the next iteration by the hardware is triggered as per the interrupt service routine.

Implementation Details

The design entry was created in Verilog HDL. Quartus II software was used for the compilation, analysis, synthesis, fitting, assembling, and timing analysis. The random bit generator was designed as a peripheral device to the embedded Nios processor. A device driver written in C was used to control the peripheral device. Further, the DES RBG was adapted to serve as the random bit generator for the RSA implementation created by a colleague. The DES RBG continuously polls a designated port for the request for random bytes. On receipt of the same, it generates a 64-bit word of random bits and sends the lower order byte to the requesting device. It signals "ready" after doing so, indicating readiness to accept the next request. The implementation performed as a multi-board design and the checking for primality of the generated random number is done at the distant end.

The statistical performance of both generators fails to impress. The DES-ALFG generator is an absolute flop, while the DES generator is scarcely much better against this benchmark. At a pinch, the DES generator could be used—in spite of a little bias in its output, it exhibits no periodicity—but the ALFG as a primitive for cryptographic random number generation does not pass any statistical test other than block-frequency. In summary, the block cipher-based approach, for the primitives selected, has yielded disappointing results.

RSA Design

The design for RSA includes the design for random number generation, multiplicative inverse, and modular exponentiation.

Random Number Generation

The RSA algorithm requires that two random prime numbers of n/2 bits be generated, where n is the number of bits in the modulus. These random prime numbers are then tested for primality before they are used in the algorithm proper. Since there is a separate project on \emph{Random Number Generation} being implemented, no random numbers were generated as part of this project. However, a peripheral module to receive the random numbers generated by an external program on a different board was implemented and incorporated in the main RSA algorithm.

Architecture of Random Number Receiver

The project has been implemented on Altera's APEX 20K EP200EFC484-2X board, which has a space limitation as far as the number of LEs is concerned. Also, the board has been manufactured in such a way that it does not permit daisy-chaining architecture to overcome the above limitation. Hence, the only method available is to use the external pins on the board, connect those to another board, and exchange data between the two. This, however, has certain limitations, such as the numbers of bits that can be exchanged, the timing issues between the two independent programs, and the requirement of exchanging signals between the boards to facilitate communication as per specific requirements. A multi-board architecture was realized to exchange data between two boards connected through external pins. Due to the limitations mentioned above, a peripheral module for handling random numbers of 16 bits each was implemented. This design is completely scalable and, hardware permitting, can receive any number of bits from another board.

This peripheral module has the following components:

■ Random number receiver module

■ Driver program, which receives the random numbers from the random number receiver module

■ Primality check module, based on Fermat's Theorem and utilizing the exponentiator peripheral

Random Number Receiver Module

This is a module written in Verilog HDL and it resides on the hardware (FPGA). To receive the random numbers and to communicate with another board, 10 external pins have been mapped with this module. On eight of these external pins, the module receives the random numbers, one byte at a time. Of the other two pins, one is used to send a \emph{start} signal to the other board and the other to receive the \emph{done} signal from it. A common ground is necessary for this type of data exchange. On receiving the \emph{done} signal from the second board, this module transfers the byte received on the external pins, first to an internal register and thereafter to the driver program. After sending that byte to the driver program, it is ready to receive the next byte. The number of bytes to be received can be set at the beginning of the data exchange. On completion, it hands over control to the driver program for further processing of these random numbers received.

Random Number Receiver Block Diagram

The random number receiver has been implemented as a peripheral and shown in the figure given below. The block diagram also shows the random number generator peripheral implemented on a different board. Both these peripherals exchange data and signals through the external pins of the Altera board. As explained earlier, these pins have been mapped on to the inputs and outputs of the peripherals in the FPGA.

Driver Program

This program has been written in C and it interacts with the random number receiver module through the Nios processor. With each hardware interrupt, it activates its hardware handler subroutine and captures the byte sent into an array. It then combines two bytes at random and then sends it to the primality check module. If the primality check is positive, this driver program stores that 16-bit random prime number to be used subsequently in the RSA algorithm, else it discards that number. The same process is repeated until it gets at least two prime numbers of 16 bits each. These two prime numbers eventually make p and q for the RSA algorithm. After obtaining p and q, it also computes n = pq, which is the modulus, and phi = (p-1)(q-1)}, which is phi(n).

Primality Check Module

This module is based on the Fermat's Theorem, which states that for any integer a, and any prime number n, if n is prime then

$$a^{\{n\}} \bmod n = a$$

If $a^{\{n\}} \bmod n$,n eq a, then n is not prime. By testing sufficient number of a's, all composite a's can be excluded and all primes can be retained. Another variation of Fermat's Theorem that can also be utilized to carry out a primality check is Euler's Theorem. It states that, if a is any integer and p is prime, such that gcd(p,a) = 1, then

$$a^{\{p-1\}} \bmod p = 1$$

This is possible only if p is prime. The existing modular exponentiation architecture can be utilized to carry out the exponentiation required by Fermat's theorem or Euler's theorem to determine whether the number is prime or not. If the number is prime, then, the driver program retains that number to be further handed over to the main RSA driver routine.

Multiplicative Inverse

The multiplicative inverse of a number, over a modulus, is computed based on the Extended Euclidean algorithm. The algorithm needs to do integer division twice for that which the module calModulus makes use of. This is by far the most time consuming, as well as resource consuming, operation in RSA. The Altera APEX 20K EP200EFC484-2X board is able to accommodate the algorithm for computing the multiplicative inverse only up to 48 bits. The design incorporates two modules:

■ Extended Euclidean module

■ Modulus

Extended Euclidean Module

This is the top-level module, which takes as input the value of exponent e and the value of phi. Based on the value of e, it goes through the various steps of the Extended Euclidean algorithm. For each step, it sends the dividend and divisor values to the modulus for performing the integer division. The modulus returns the remainder and quotient after the division operation. Finally, the inverse value is returned after ascertaining that the last non-zero remainder is one, and the algorithm is executed for two steps beyond the Euclidean algorithm.

Modulus

This module is based on the non-restoring division method of calculating the modulo. It takes two inputs, the dividend and the divisor. After division, it returns the remainder and quotient back to the Extended Euclidean module. The multiplicative inverse computed by this peripheral is based on the value of phi generated, as well as the value of exponent e chosen. The value of e chosen is actually the public key and the multiplicative inverse computed is the secret key or d. This value of d is then used during the decryption phase for computing the original plaintext.

The module for computing the multiplicative inverse has been implemented as a peripheral on the FPGA. The driver program sends the exponent value and the phi value to this peripheral through the Nios processor. The peripheral computes the secret key or the inverse value of the exponent with respect to phi and returns it via the Nios processor to the driver program.

Modular Exponentiation

An architecture for modular exponentiation proposed by Thomas Blum and Christof Paar was chosen for implementation. It is based on the Montgomery exponentiation and Montgomery modular multiplication for radix 2. It is a resource-efficient architecture suitable for implementation in FPGAs. Its design is based on an exponentiator, which handles the exponentiation and feeds values to a systolic array that computes the modular multiplication. The architecture essentially consists of two basic units, the exponentiator and the systolic array.

Exponentiator

This is the top-level module and is based on the Montgomery exponentiation algorithm. It takes as input the following parameters:

■ Modulus m

■ Message x

■ Exponent e

■ Number of bits in exponent

■ Precomputation factor R^{2} mod m

The precomputation factor and A are fed as inputs so that all values in the intermediate stages of exponentiation are in Montgomery domain carrying a factor of 2^{n+2}, where n is the number of bits in the modulus. This module first feeds the values of x and R^{2} mod m to the systolic array for computation of $\widetilde{x}$. Thereafter, it first checks the exponent bit and then feeds appropriate values to the systolic array for multiplication. At the end it feeds the result and value 1 again to the systolic array to obtain the final result, thereby getting rid of the additional factor of 2^{n+2}. The final result so obtained is either the ciphertext or the plaintext depending upon whether it is encryption or decryption. In case of encryption, the exponent used is 65537, while in the case of decryption it is the secret key or d computed as the multiplicative inverse earlier.

Systolic Array

The systolic array computes the modular multiplication based on the Montgomery modular multiplication algorithm. A systolic system comprises a set of interconnected cells, each capable of performing a specified operation. The cells and operations performed by them are usually identical. The time taken for processing by each of the cells is identical. Individual cells are connected only to their nearest neighbors. The flow of data between the cells is rhythmic and regular. Except those at the boundary of the array, the cells do not communicate with the outside world. Systolic architectures are essentially suited for implementing computationally bound operations. The following arithmetic operation is required to be implemented.

$S_{i+1} = (S_i + q_iM)/2 + a_iB, \quad q_i, a_i \{0,1\}$

The above equation can be modified into

$S_{i+1} = (S_i + q_iM + 2a_iB)/2, \quad q_i, a_i \{0,1\}$

Instead of using two adders for computing the addition required in the above step, the sum 2B + M is precomputed and stored in a register. A single adder is sufficient to add 0, 2B, M or 2B + M to S_i, depending on the values of a_i and q_i. The same adder can also be used to precompute 2B + M. The systolic array has the following inputs and outputs:

Inputs:

■   `modulus`: Modulus value sent by exponentiator

■   `var1`: First variable to be multiplied

■   `var2`: Second variable to be multiplied

■   `clk`: Clock for synchronization

■   `reset`: Reset signal

Outputs:

■   `result`: Result of the modular multiplication

All the processing elements get instantiated in this module. The signals and the data exchanged between the processing elements are declared as wires in the systolic array module. During each clock cycle, each processing element computes u bits of $S_{i+1} = (S_i + q_iM + 2a_iB)/2$. Now, during the clock cycle I, the processing element1 computes u bits of S_i. During clock cycle i+1 processing element2 computes the next u bits of S_i. To compute u bits of S_{i+1} the processing element1 requires the LS bit of S_i, computed by processing element2. This is on account of the division by two required in step $S_{i+1} = (S_i + q_iM + 2a_iB)/2$. Thus, clock cycle i+1 is unproductive for processing element1. Therefore, each unit of the systolic array stays idle in every alternate clock cycle. To achieve parallelism each processing element computations during one cycle and remains idle in the next clock cycle. This alternate clock cycle computation ensures parallelism and a complete utilization of all its units.

To compute $S_{i+1} = (S_i + q_iM + 2a_iB)/2$ where $M = \sum_{i=1}^{n-1} {m_i2^i}$, $m_i {0,1\}$ and $B = \sum_{i=1}^{n} {b_i2^i}$, $b_i {0,1\}$, n/u + 1 units are needed. The unit n/u + 1 is used to process the most significant bit of part b and has no part mod inputs. There are two buses each for loading the M and B. The M even bus and B even bus are connected to units processing element1, processing element3, ...processing element(n/u + 1). The M odd bus and B odd bus are connected to units processing element2, processing element4}, processing element(n/u)

The s1 out output of processing element1 is connected back to its input. This is required for subsequent passes through the loop. The carry generated in each addition is also propagated to units in the left through the use of c_out and c_in pins. The s0 out is connected to the s0 in the pin to the right in order to send the LS bit of the left shifted S_i(division by 2). The result bits are pumped backwards to _emph{processing_element1} through the use of the res _out and res_in pins as only processing element1 is connected to external modules.

Operation:

1.   Initially, the values M and 2B are fed to all the units and saved in registers.

2.   The computation of $S_{i+1} = (S_i + q_iM + 2a_iB)/2$ begins by initialization followed by giving the a_i input at the a_i_In pin of processing_element1}.

3.   At the end of computation, the result is pumped across the units to processing_element1.

Processing Element

At the heart of the processing element is a u bit Adder. The result of the adder is latched into S_Reg (including the carry). An extra S_Reg_2 is required to introduce the one clock delay before the result is fed back as input for the next pass through the loop after it is left-shifted. The LS bit of the shifted input comes from the neighboring unit from S0_In pin. Registers B_Reg, M_Reg, and BMSum_Reg are used to save 2B, M, and B + M. Multiplexer Mux_B is used to selectively input B_Reg with the B_In or the S_Reg. Multiplexers Mux_1 and Mux_2 select the appropriate inputs for the adder. Mux_Res selectively outputs the result of the adder or Result_In from the neighbor. Control_Reg and qa_Reg are used to latch the values before passing it along to their neighbors. Each processing element of the systolic array computes u bits of modular multiplication. Each processing element has the following inputs and outputs.

Inputs:

■   `res_in`: Result bits from the (left) neighboring unit

■   `qa_in`: q_i,a_i bits

■   `c_in`: Carry bit from (right) neighbor

■   `s0_in`: LS bit of S_I from (left) neighbor required on account of division by 2 of S_i

■   `part_b`: u bits of 2B fed externally at the start for saving and precomputation of 2B + M

■   `part_mod`: u bits of M fed externally at the start for saving and precomputation of 2B + M

■   `clk`: Clock signal for synchronization

■   `reset`: Reset signal

■   `start`: Start signal from exponentiator to begin computation

■   `flush`: Flush signal from the exponentiator to flush all the registers before the next multiplication

Outputs:

■   `res_out`: Result bits computed by the unit

■   `s0_out`: LS bit of result for the (right) neighbor

■   `qa_out`: q_i,a_i bits for the (left) neighbor

■   `c_out`: Carry bit generated by the addition

■   `s1_out`: 2nd LSB required as q_i input for unit 1, can be taken from the LSB of res_out, also

■   `next`: Next signal for the neighbor to start computation

Modular Exponentiator Block Diagram

The modular exponentiation has been implemented as a peripheral comprising all of the modules mentioned above. The top-level module in this peripheral receives the exponent value, modulus value,

correction factor, message, and the number of bits in the exponent from the driver program via the Nios processor in 32 bits each. The peripheral then computes the value of the exponentiation and returns it back to the driver program.

Design Methodology

RSA Implementation

Altera's APEX 20KE FPGA family was chosen for implementing the RSA algorithm. APEX devices are high-density FPGAs that allow complex designs to be implemented on a single device. The target device was an EP20K200EFC484-2X. The design files were written in Verilog HDL, while compilation, synthesis, fitting, placement, and routing were carried out using Quartus II software.

Design Flow

The complete implementation of the RSA in FPGA was performed in the following stages:

4. Design entry

5. Compilation and synthesis

6. Fitting, placement, and routing

7. Interaction with the C Driver program

Design Entry

The designs for the project were specified by using the Verilog HDL. The Verilog HDL files are essentially the source files, giving the structural description of each of the sub-units.

Random Receiver

This contains the design file random_receiver.v, which receives the random numbers on the output pins, generated on the other board. A total of 10 external pins were used to collect the random numbers one byte at a time. The balance of the two pins was used for synchronization purposes. This Verilog file contains the mechanism of raising hardware interrupts and throwing out the byte received to the driver program for further processing.

Multiplicative Inverse

This contains the following design files:

■ *calModulus.v*—This module performs the division operation, given the dividend and divisor, and returns the remainder and quotient after the division operation. The size of the inputs and outputs of this module are parameterized to facilitate easy scalability.

■ *topInverse.v*—This module implements the extended Euclidean algorithm for calculating the multiplicative inverse. It instantiates the calModulus.v module for performing the division operation. The inputs and outputs of this module are also parameterized.

Modular Exponentiation

This contains the following design files:

- *processing element.v*—This gives the structural description of the processing element of the systolic array. The word size of the processing element is parameterized and can be altered. Each processing element computes the sum as per the algorithm.

- *systolic_array.v*—This module instantiates a series of processing elements and specifies the interconnections between them in terms of inputs and outputs. It returns the result of a multiplication to the exponentiator module, based on the Montgomery modular multiplication algorithm.

- *monty_expo.v*—This is the top-level module that implements the Montgomery exponentiation algorithm as a series of modular multiplications with the help of the underlying systolic array module.

Compilation & Synthesis

The design files form the input to the compilation and synthesis tool (i.e., Quartus II development software). The design files are first included in the project **standard_32** directory within Quartus II software. Thereafter, a new peripheral is created for each top-level module with the help of the SOPC builder. The SOPC builder is then generated to build the user-defined peripherals along with the design files of the **standard_32** directory. The operating frequency and the target devices are selected at the time of opening a new project. Finally, the whole project is compiled and synthesized.

Fitting, Placement & Routing

Quartus II development software is also used for this purpose. The netlist file generated during the compilation and synthesis forms the input to it. The fitter in Quartus II software assigns each logic function to the best logic cell location for routing and timing. It also selects appropriate interconnection paths and pin assignments. The final output is the **standard.sof** file, which contains the complete routed application.

Interaction with C Driver Program

The design files implemented in the hardware are actually peripherals to the Nios processor and work through the Avalon® bus signals. To write/read data to/from the peripheral, a C driver program is used. This C program is loaded in \cpu_sdk\src project subdirectory within standard_32. The nios_build and nios_run utilities are then used to compile the C program and run it on the design files already downloaded to the FPGA. The C program includes the nios.h, which in turn includes all the header files required for compilation. Also, the peripheral created in the SOPC builder is instantiated in the C program along with its IRQ number. The handler function in the C program then performs the functions mentioned inside the handler in the event of the peripheral raising an interrupt. The data is written to the peripheral through the writedata Avalon signal while the reading of data from the peripheral is done through readdata. Both writedata and readdata work for specific addresses that need to be mentioned in the C program.

Implementation Issues

This section describes the implementation issues for this project.

Use of External Pins

For peripherals involving use of external pins, the additional pins used are marked as export, before generation in the SOPC Builder. After generation, physical assignment of each and every pin is carried

out using the assignment editor within Quartus II software. The external pins to be assigned are selected through the Nios development manual. The balance of the operations is similar to that described in earlier sections. This configuration and implementation was carried out for the random_receiver module and peripheral.

16-Bit Implementation

Handling large numbers (1024 bit) makes debugging and functional verification very difficult. Also, the time taken by the software tools, especially the placement and routing (fitter) and simulator, is extremely high. Therefore, a 16-bit exponentiate was built and tested thoroughly as a first step. The exponentiate was then scaled up from 16 bits to 512 bits.

Modular Design

The design of the exponentiate is modular with the processing element and the systolic array being independently implemented and tested. Finally the modules were then integrated together and tested. Same is the case for the other peripherals like multiplicative inverse and the random number receiver. The peripherals have been designed in such a manner that all inputs and outputs are parameterized and can be changed easily without affecting any other part of the module.

Design Scalability

The exponentiator and the multiplicative inverse peripherals scale linearly and therefore require little effort.

Testing & Verification

The test cases for testing were given using the C driver program to the Verilog HDL design file and then reading back the results in the driver program. Initial simulation and testing was carried out using iverilog, being faster. The testing and verification in the hardware takes time owing to time taken for compilation, synthesis, fitting, placement, and routing by the Quartus II software.

Processing Time

An important issue associated with the implementation is the processing time associated with Quartus II software. For the exponentiator, multiplicative inverse, and the random number receiver, the time taken for compilation, synthesis, and fitting is about 12 to 15 minutes.

Software Implementation

A software implementation of modular exponentiation algorithm, multiplicative inverse, modular multiplication, generation of random numbers, and multiplication of large integers was implemented in C and Java to verify the correctness of the results obtained. The Montgomery multiplication algorithm was also implemented to verify the correctness of the intermediate results during exponentiation. This was necessary since the intermediate results carry the additional factor of 2^{n+2} at each stage.

Design Features

The highlights of our design features that we implemented were:

■ Interboard communication between two Nios processors using interrupts. This entailed interrupt handling.

■ Use of peripherals around the Nios core. This facilitated quick prototyping at the design and trial stage.

■ The most significant advantage that is accrued by modeling the design as an Avalon bus peripheral of the Nios processor is that Altera allows control of the input to and the output from the peripherals through a device driver that can be written in C language. This fact allows verification of the cryptographic algorithm once burnt into the hardware, even after simulation is complete.

Conclusion

The entire course of this design was a period of cumulative learning and enrichment of our knowledge regarding the Nios processor and the FPGA. The most satisfying part of this project was the multi-board architecture implemented to make use of two boards simultaneously and realizing an asynchronous system. The use of minimal pins and LEs (as discussed in Part III) to achieve this cryptographic algorithm was one of the achievement of this project. Coupled to this was the fact that a hardware acceleration was achievable, as was hardware reusability.

Second Prize

SOPC-Based Word Recognition System

Institution: **National Institute Of Technology, Trichy**

Participants: **S. Venugopal, B. Murugan, S.V. Mohanasundaram**

Instructor: **Dr. B. Venkataramani**

Design Introduction

Real-world processes generally produce observable outputs, which can be characterized as signals. The signals can be discrete in nature (e.g., characters from a finite alphabet, quantized vectors from a codebook), or continuous in nature (e.g., speech samples, temperature measurements, music). The signal source can be stationary (i.e., its statistical properties do not vary with time), or non-stationary (i.e., the signal properties vary over time). The signals can be pure (i.e., coming strictly from a single source), or can be corrupted from other signal sources (e.g., noise) or by transmission distortions, reverberation, etc.

A problem of fundamental interest is characterizing such real-world signals in terms of signal models. There are several reasons to be interested in applying signal models. First, a signal model can provide the basis for a theoretical description of a signal processing system, which processes the signal to provide a desired output.

A second reason why signal models are important is that they are potentially capable of letting us learn a great deal about the signal source (that is, the real-world process that produced the signal) without having the source available. This property is especially important when the cost of getting signals from the actual source is high. In this case, with a good signal model, we can simulate the source and learn as much as possible via simulations.

Finally, signal models are important because they often work extremely well in practice, and enable us to realize important practical systems (such as prediction systems, recognition systems, identification systems, etc.) in a very efficient manner. There are several possible choices for the type of signal model used for characterizing the properties of a given signal. Broadly, one can dichotomize the types of signal models into the class of deterministic models and the class of statistical models.

Deterministic models generally exploit some known specific properties of the signal (e.g., that the signal is a sine wave or a sum of exponentials). In these cases, specification of the signal model is generally

straightforward; all that is required is to determine (estimate) the values of the signal model parameters (e.g., amplitude, frequency, phase of a sine wave, amplitudes and rates of exponentials, etc.).

The second broad class of signal models is the set of statistical models in which one tries to characterize only the statistical properties of the signal. The underlying assumption of the statistical model is that the signal is characterized as a parametric random process, and that the parameters of the stochastic process can be determined (estimated) in a precise, well-defined manner. The Hidden Markov Model (HMM) falls in this second category.

In simple terms, a HMM is a model used to create another model about which we know nothing except its output sequence. The HMM is trained to produce an output that closely matches the available output sequence, and can be assumed to model the unknown model with sufficient accuracy.

Speech recognition systems have been developed for many real-world applications, often using low-cost speech recognition software. However high-performance and robust isolated word recognition, particularly for digits, is still useful for many applications, such as recognizing telephone numbers for use by physically challenged persons. This formed the motivation for taking up this project.

Efficient implementation of a complete system on a programmable chip (SOPC) got an impetus with the advent of high-density FPGAs integrated with high-capacity RAMs and the availability of implementation support for soft-core processors such as the Nios® II processor. FPGAs enable the best of both worlds to be used gainfully for an application—the microcontroller or RISC processor is efficient for performing control and decision-making operations, while the FPGA efficiently performs digital signal processing (DSP) operations and other computation intensive tasks.

We aim to produce an efficient hardware speech recognition system with an FPGA acting as a coprocessor that is capable of performing recognition at a much faster rate than software. Implementation of systems using an Altera®-based system on a programmable chip enables time-critical functions to be implemented on hardware synthesized with VHDL/Verilog HDL code. The soft-core Nios II processor that is part of the FPGA can execute the programs, written in a high-level language. Custom instructions enable the feasibility of implementing the whole system on an FPGA with better partitioning of the software and hardware implementation of the speech recognition system.

Our project aims at developing a HMM-based speech recognition system with a vocabulary of 10 digits (digits zero to nine). We trained the system for three users for all the mentioned digits with a recognition accuracy of nearly 100%. Energy and zero crossings-based voice activity detection (VAD) was used for segmentation of the input samples and removing background noise. We used linear predictive coding (LPC-10) analysis, followed by cepstral analysis for feature vector extraction from speech frames. HMM was used for training the speech models and Viterbi decoding for recognition. Vector quantization (VQ) was used for reducing the memory requirement. We used direct parameter averaging of the HMM parameters during training, which has several advantages over Rabiner's approach, such as a lower data requirement, higher detection accuracy, and less computation complexity.

We implemented the feature extraction, training, and other preprocessing stages of HMM in software (C++/MATLAB) in the offline mode and the recognition process, including floating-point multiplication operation of the Viterbi decoding process, as custom hardware in hardware implementation and as an online process.

Functional Description

The aim is to design a system that will recognize an uttered digit from the recorded speech samples (recorded as .wav files and converted to text files using MATLAB). The digits have to be from a

predetermined vocabulary set for which the system is trained. The design can be split into software and hardware partitioning to exploit the facilities present in the Nios II processor. The overall design can be divided into two functional parts: training and recognition.

Training involves iteratively fine-tuning the parameters of the HMMs with digits from the given vocabulary set until it converges. The sequence of steps in training are:

■ Preprocessing

■ Codebook generation

■ Generating models for individual digits using a combination of the Forward algorithm, the Backward algorithm, and the Baum-Welch algorithm.

Recognition involves testing the given digit with each of the available digit HMMs, finding the model that gives the maximum probability, and concluding that the result corresponds with the digit uttered. The sequence of steps in recognition are:

■ Preprocessing

■ Finding the best fitting model using maximum likelihood algorithm, the Viterbi decoding algorithm

Preprocessing

Preprocessing involves the following steps:

1. *Recording*—Record the speech at a sampling frequency of 8 KHz with 16-bit quantization.

2. *VAD*—Using the endpoint detection algorithm, the starting and ending points are found. The speech is sliced into frames 450 samples in length. The energy and number of zero crossings of each frame is found. A threshold energy and zero crossing value is determined based on the computed values, and only frames crossing the threshold are considered. This removes most of the unnecessary background noise. A small vestige of frames beyond the starting and ending frames are included so as not to leave the starting and ending parts of the speech that do not cross the threshold but that may prove important in recognition.

3. *Pre-emphasis*—The digitized speech signal s(n) is put through a low-order LPF to spectrally flatten the signal and to make it less susceptible to finite precision effects later in the signal processing. The filter is represented by $H(z) = 1-az^{-1}$ where we have chosen the value of "a" as 0.9375.

4. *Frame blocking*—Speech frames are formed with durations of 56.25 ms (N = 450 sample length) and with an overlap of 18.75 ms (M=150 sample length) between adjacent frames. The overlapping is performed so that the resulting LPC spectral estimates correlate from frame to frame and are quite smooth.

   $x_q(n)=s(Mq+n)$ n=0 to N-1; q=0 to L-1 where L is the number of frames.

5. *Windowing*—Each frame with a Hamming window is windowed to minimize signal discontinuities at the beginning and end of the frames.

   $x'_q(n)=x_q(n) \cdot w(n)$

Where

$w(n)=0.54=0.46 \cos (2 \pi n/ N-1)$

6. *Autocorrelation analysis*—Perform autocorrelation analysis for each frame and find P+1 (P=10) autocorrelation coefficients.

7. $r_q(m)= \sum\limits_{n=0}^{N-1-m} x'_q(n)x'_q(n+m)$ $m=0,1,.....P$

8. The zeroth autocorrelation coefficient is the frame's energy, which was previously used for VAD.

9. Perform LPC analysis by employing Levinson and Durbin's algorithm to convert the autocorrelation coefficients to LPC parameter set.

$E^{(0)} =r_q(0)$

$K_i=\{ r_q(i) – \{ \sum\limits_{j=1}^{L-1} \alpha_j^{(i-1)}. r_q(|i=j|) \} \}/E^{(i-1)}$ $1 \leq i \leq P$

$\alpha_i^{(i)} = k_i$

$\alpha_j^{(i)} = \alpha_j^{(i-1)} - k_i. \alpha_{(i-j)}^{(i-1)}$

$E^{(i)} = (1- k_i^2) E^{(i-1)}$

Where α_m^{10} $1 \leq m \leq P$ are the LPC coefficients.

10. LPC parameters to cepstral coefficient conversion: The cepstrum coefficients are a more robust and reliable feature set than the LPC coefficients.

$c_0=\ln \sigma^2$ where σ is the gain of the LPC model.

$c_m= \alpha_m + \sum\limits_{k=1}^{m-1} (k/m) . c_k. \alpha_{m-k}$ $1 \leq m \leq P$

$c_m= \sum\limits_{k=1}^{m-1} (k/m) . c_k. \alpha_{m-k}$ $P<m \leq Q$

11. *Parameter weighing*—Sensitivity of the lower order cepstral coefficients to overall slope and the higher order coefficients to noise has necessitated weighing of the cepstral coefficients by a tapered window to minimize these sensitivities. We have used weighing by a band pass filter of the form

$w_m = [1 + (Q/2). \sin(\ m/Q)]$ $1 \leq m \leq Q$

12. *Temporal cepstral derivative*—Temporal cepstral derivatives are an improved feature vector for forming the speech frames. They can be used with the cepstral derivative in case the cepstral coefficients do not give acceptable recognition accuracy.

Vector Quantization

A codebook of size 128 is obtained by the VQ of the weighted cepstral coefficients of all reference digits, by all users. The advantages of VQ are:

■ Reduced storage for spectral analysis information

■ Reduced computation for determining similarity of spectral analysis vectors.

■ Discrete representation of speech sounds. By associating phonetic label(s) with each codebook vector, the process of choosing a best codebook vector to represent a given spectral vector becomes equivalent to assigning a phonetic label to each spectral frame of speech. This makes the recognition process more efficient.

One obvious disadvantage of VQ is the reduced resolution in recognition. Assigning a codebook index to an input speech vector amounts to quantizing it. This results in quantization error, which increases with decrease in codebook size.

There are two commonly used algorithms, the K-Means algorithm and the Binary Split algorithm. The K-Means algorithm describes the way in which a set of L training vectors can be clustered into M (<L) codebook vectors with:

1. *Initialization*—Arbitrarily choose M vectors as the initial set of code words in the codebook.

2. *Nearest neighbor search*—For each training vector, find the code word in the current codebook that is closest and assign that vector to the corresponding cell.

3. *Centroid update*—Update the codeword in each cell using the centroid of the training vectors assigned to that cell.

4. *Iteration*—Repeat the above two steps until the average distance falls below a preset threshold.

The Binary Split algorithm is a more efficient method than the K-Means algorithm because it builds the codebook in stages. First, it designs a 1-vector codebook, then uses a splitting technique on the code words to initialize the search for a 2-vector codebook, and then continues the splitting process until the desired M-vector codebook is received.

1. Design a 1-vector codebook, which is the centroid of the entire training set and hence needs no iteration.

2. Double the size of the codebook by splitting each current codebook yn according to the rule

$y_n^+ = y_n(1+e)$

$$y_n^{\cdot} = y_n(1\text{-}e)$$

where n varies from 1 to the codebook size and e is the splitting parameter.

3. Use the K-Means iterative algorithm to obtain the best set of centroids for the split codebook.

4. Iterate the above two steps until the required codebook size is received.

Hidden Markov Model

Recognition is achieved by maximizing the probability of the linguistic string, W, given the acoustic evidence, A. For example, choose the linguistic sequence W such that

$$P(W' \mid A) = \max P(W \mid A)$$

W

An HMM is characterized by the following:

- N, the number of states in the model. Although the states are hidden, for many practical applications there is often some physical significance attached to the states or to sets of states of the model. The states are interconnected in such a way that any state can be reached from any other state (e.g., an ergodic model); however other possible interconnections of states are often used by restricting the transitions. We denote the individual states as $S = \{S_1, S_2, \ldots S_N\}$, and the state at time t as q_t.

- M, the number of distinct observation symbols per state, i.e., the discrete alphabet size. The observation symbols correspond to the physical output of the system being modeled. We denote the individual symbols as $V = \{V_1, V_2, \ldots V_M\}$

- The state transition probability distribution $A = \{a_{ij}\}$ where $a_{ij} = P[q_{t+1} = S_j \mid q_t = S_i]$ $1 \le i, j \le N$

- The observation symbol probability distribution in state j, B = {bj(k)},

    where $b_j(k) = P[v_k$ at $t \mid q_t = S_j]$ $1 \le j \le N$, $1 \le k \le M$

- The initial state distribution $= \{_i\}$ where $_i = P[q_1 = S_i]$ $1 \le i \le N$

Given appropriate values of N, M, A, B, and   , the HMM can be used as a generator to give an observation sequence $O = O_1 O_2 O_3 \ldots O_T$ (where each observation O_t, is one of the symbols from V, and T is the number of observations in the sequence).

Three Basic Problems for HMM

Given the form of HMM, there are three basic problems of interest that must be solved for the model to be useful in real-world applications. These problems are as follows:

- *Problem 1, the scoring problem*—Given the observation sequence $O = \{O_1, O_2, \ldots O_T\}$ and a model $\lambda = (A, B, \pi)$, how to efficiently compute P(O/A), the probability of the observation sequence, given the model? The algorithm normally used to solve this is the Forward-Backward algorithm

- *Problem 2, the matching problem*—Given the observation sequence $O = \{O_1, O_2, \ldots O_T\}$, and the model λ, how to choose a corresponding state sequence $Q = q_1 q_2 \ldots q_T$ which is optimal in

some meaningful sense (i.e., best "explains" the observations)? The algorithm normally used to solve this is the Viterbi algorithm

■ *Problem 3, the training problem*—How to adjust the model parameters $\lambda = (A, B, \pi)$ to maximize P(O/A)? The algorithm normally used to solve this is the Baum-Welch Re-Estimation Procedures.

Solution to Problem 1

One of the most straightforward and highly inefficient methods to solve this problem is to enumerate all the possible state sequences of length **T** and finding the sum over all such state sequences to find the required probability. The forward-backward is a more efficient algorithm, which can be explained as follows:

Consider the forward variable $\alpha_t(i)$ defined as

$$\alpha_t(i)=P[O_1, O_2, O_3, \ O_t, q_t=S_i \,|\, \lambda\,]$$

That is, the probability of the partial observation sequence, $O_1, O_2. . . O_t$ and state S_i at time t, given the model λ. We can solve for $\alpha_t(i)$ inductively, as follows:

1. Initialization:

    $$\alpha_1(i)= \pi_i. \ b_i(O_1) \ 1\leq i\leq N$$

2. Induction:

    $$\alpha_{t+1}(j)=[\ \sum_{i=1}^{N} \alpha_t(i).a_{ij} \ ] \ b_j(O_{t+1}) \ \ 1\leq t\leq T-1; \ 1\leq j\leq N$$

3. Termination:

    $$P(O|\lambda)= \ \sum_{i=1}^{N} \alpha_T(i)$$

In a similar manner we can solve for the backward variable $\beta_t(i)$ iteratively as follows:

1. Initialization:

    $$\beta_T(i)= 1 \ \ 1\leq i\leq N$$

2.   Induction:

$$\beta_t(i) = \sum_{j=1}^{N} \beta_{t+1}(j) \cdot a_{ij} \cdot b_j(O_{t+1}) \quad T-1 \geq t \geq 1; \ 1 \leq i \leq N$$

Solution to Problem 2

Unlike Problem 1, for which an exact solution can be given, there are several possible ways of solving Problem 2, namely finding the "optimal" state sequence associated with the given observation sequence. The difficulty lies with the definition of the optimal state sequence; i.e., there are several possible optimality criteria. To implement this solution to Problem 2, we define the variable

$$\gamma_t(i) = P(q_t = S_i \mid O, \lambda)$$

$$\gamma_t(i) = \alpha_t(i) \cdot \beta_t(i) \ / \ P(O \mid \lambda)$$

$$= \ \alpha_t(i) \cdot \beta_t(i) \ / \ \sum_{i=1}^{N} \alpha_t(i) \cdot \beta_t(i)$$

Because $\alpha_t(i)$ accounts for the partial observation sequence $O_1, O_2, \dots O_t$, and state S_i at t , while $\beta_t(i)$ accounts for the remainder of the observation sequence $O_{t+1}, O_{t+2}, \dots O_T$ given state S_i at t.

Viterbi Algorithm

To find the single best state sequence, $Q = \{q_1, q_2, \dots q_T\}$, for the given observation sequence $O = (O_1, O_2, \dots O_T\}$, we need to define the quantity

$$\delta_t(i) = \max_{q_1, q_2, \dots q_t} P[q_1, q_2, \dots q_t = i, O_1, O_2 \dots O_t / \lambda]$$

That is, $\delta_t(i)$ is the best score (highest probability) along a single path, at time t, which accounts for the first t observations and ends in state S_i. By induction we have

$$\delta_{t+1}(j) = [\max \delta_t(i) \, a_{ij}] \cdot b_j(O_{t+1}).$$

To retrieve the state sequence, we need to keep track of the argument that maximized, for each t and j. We do this via the array $\psi_t(j)$. The complete procedure for finding the best state sequence can now be stated as follows:

1.   Initialization:

$$\delta_1(i) = \pi_i b_i(O_1), \ 1 \leq i \leq N$$

$$\psi_1(i) = 0$$

2. Recursion:

$$\delta_t(j) = \max\ [\delta_{t-1}(i)a_{ij}\ ]\ b_j(O_t)\ ,\quad 2 \le t \le T\ ;\qquad 1 \le j \le N$$

$$\psi_t(j) = \text{argmax}\ \ [\delta_{t-1}(i)a_{ij}],\ \ 2 \le t \le T\ ;\ 1 \le i \le N\ ;\quad 1 \le j \le N$$

3. Termination:

$$P^* = \max\quad [\delta_t(i)]$$

$$1 \le i \le N$$

$$q_T^* = \ \text{argmax}\ [\delta_t(i)]$$
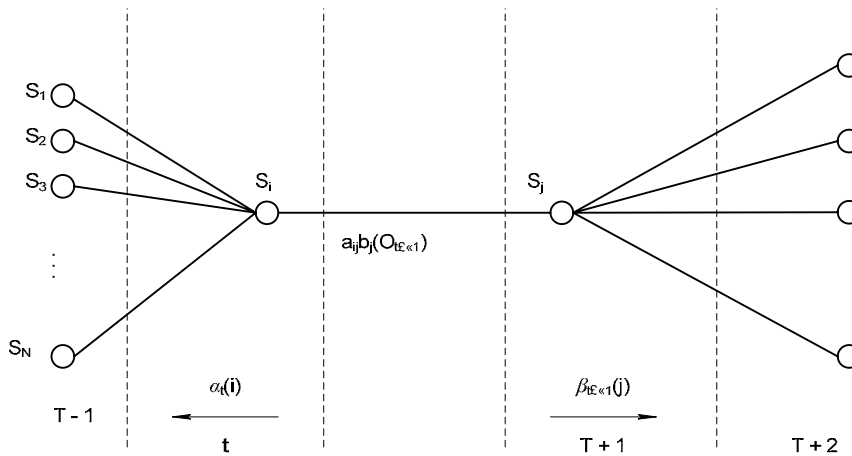
$$1 \le i \le N$$

4. Path (state sequence) backtracking:

$$q_t^* = \psi_{t+1}(q_{t+1}^*),\ t = T\ \text{-}\ 1,\ T\ \text{-}\ 2,\ ..\ 1.$$

The lattice (or trellis) structure given in Figure 1 efficiently implements the computation of the Viterbi procedure.

Figure 1. Lattice Structure

Solution to Problem 3

The third, and by far the most difficult, problem of HMMs is to determine a method to adjust the model parameters (A, B, π) to maximize the probability of the observation sequence given the model. There is no known way to analytically solve for the model, which maximizes the probability of the observation sequence. In fact, given any finite observation sequence as training data, there is no optimal way of estimating the model parameters. We can, however, choose $\lambda = (A, B, \pi)$ such that $P(O \mid \lambda)$ is locally maximized using an iterative procedure such as the Baum-Welch method.

To describe the procedure for re-estimation (iterative update and improvement) of HMM parameters, we first define $\xi_t(i,j)$, the probability of being in state S_i at time t, and state S_j, at time t+1, given the model and the observation sequence,

144

$\xi_t(i,j) = P(q_t=S_i, q_{t+1}=Sj \mid O, \lambda)$.

We can write $\xi_t(i,j)$ in the form

$\xi_t(i,j) \quad = P(q_t=S_i, q_{t+1}=S_j, O \mid \lambda) / P(O \mid \lambda)$

$\quad\quad = \alpha_t(i) \, a_{ij} \, b_j(O_{t+1}) \, \beta_{t+1}(j) / P(O \mid \lambda)$

$$= \alpha_t(i) \, a_{ij} \, b_j(O_{t+1}) \, \beta_{t+1}(j) / \left[\sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_t(i) \, a_{ij} \, b_j(O_{t+1}) \, \beta_{t+1}(j) \right]$$

We have previously defined $\gamma_t(i)$ as the probability of being in state S_i at time t, given the observation sequence and the model; hence we can relate $\gamma_t(i)$ to $\xi_t(i,j)$ by summing over j, giving

$$\gamma_t(i) = \sum_{j=1}^{N} \xi_t(i,j)$$

If we sum $\gamma_t(i)$ over the time index t, we get a quantity that can be interpreted as the expected (over time) number of times that state S_i is visited, or equivalently, the expected number of transitions made from state S_i (if we exclude the time slot t = T from the summation). Similarly, summation of $\xi_t(i,j)$ over t (from t = 1 to t = T - 1) can be interpreted as the expected number of transitions from state S_i to state S_j. That is,

$$\sum_{t=1}^{T-1} \gamma_t(i) = \text{expected number of transitions from } S_i$$

$$\sum_{t=1}^{T-1} \xi_t(i,j) = \text{expected number of transitions from } S_i \text{ to } S_j \text{ in O.}$$

Using the above formulas (and the concept of counting event occurrences), the method for re-estimation of the parameters of an HMM is as follows:

Π'_j = expected frequency (number of times) in state S_i at time (t = 1) = $\gamma_1(i)$

a'_{ij} = expected number of transitions from state S_i to state S_j / expected number of transitions from state Si

145

$$= \quad \sum_{t=1}^{T-1} \xi_t(i,j) \ / \ \sum_{t=1}^{T-1} \gamma_t(i)$$

b'$_j$(k)= expected number of times in state S$_j$ and observing symbol v$_k$ / expected number of times in state S$_j$

$$= \sum_{t=1}^{T} \gamma_t(i) \quad / \ \sum_{t=1}^{T} \gamma_t(i)$$

such that Ot=vk

Based on the above procedure, we iteratively use λ' in place of λ and repeat the re-estimation calculation, to improve the probability of O being observed from the model until some limiting point is reached. The result of this re-estimation procedure is a maximum likelihood estimate of the HMM. We use a terminating condition of λ' not varying by more than a certain fraction (say 10%) from λ.

Training

Training the HMM for each digit in the vocabulary is done using the Baum-Welch algorithm. The codebook index will be the observation vector for the HMM.

Recognition

Recognition of the uttered digit is found by employing the maximum likelihood estimate such as Viterbi decoding algorithm. This implies the model with the maximum probability will be the uttered digit.

Performance Parameters

This section describes the performance parameters for the project.

Recognition Accuracy

100% recognition accuracy for a three–user dependent system implemented with input from the trained vocabulary alone.

Design Implementation Times

Total design run time for recognition:

■ 96.53 s (full software implementation)

■ 93.936 s (with floating-point multiplication operation of the Viterbi decoding process block implemented as custom block and the rest as software)

Implementation time for the recognition process (excluding preprocessing step) and finding the probability for the input digit for all models using the Viterbi block in software: 5.48 s.

Implementation time for a floating-point multiplication operation of the Viterbi decoding process block implemented as a custom block: 0.5 ms (Refer to Snapshot 4 in the Appendix). The Viterbi block uses 4810 such multiplications, so the estimated run time of the whole Viterbi process using the custom block is 2.405 sec.

Implementation time for a floating-point multiplication operation of the Viterbi decoding process block implemented as software: 5.7 ms (Refer to Snapshot 5 in the Appendix).

Speed-up factor achieved by using custom block: 11.4 times.

Design Metrics

Memory requirement—M4Ks (Refer to Snapshot 1 in the Appendix):

■   Whole design: 9 out of 20

■   Custom block alone: 2 out of 9

Logic area used: 4,233/5,980 logic elements (LEs).

Design Architecture

Figure 2 shows the software block diagram.

Figure 2. Software Design Block Diagram of Front-End Feature Analysis for HMM
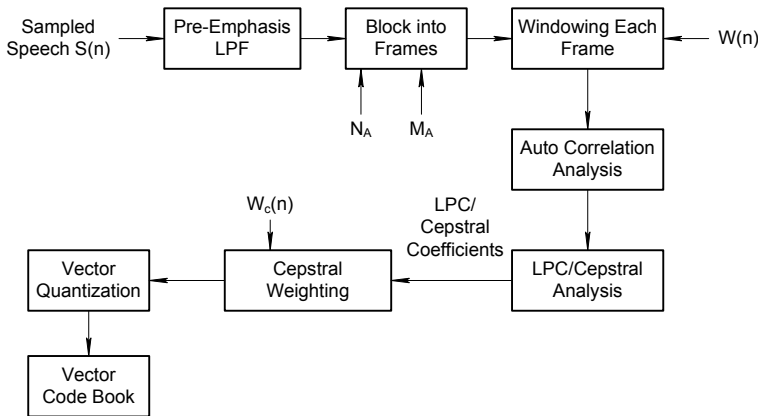
LPC Feature Analysis

Observation vectors are obtained from speech samples by performing VQ of LPC coefficients. The overall system is a block processing model in which a frame of N_A samples is processed and a vector for each frame is computed.

The steps in the processing are as follows:

1. *Pre-Emphasis*—A first-order digital network processes the digitized speech signal to spectrally flatten the signal.

2. *Blocking into Frames*—Sections of N_A consecutive speech samples are used as a single frame with the overlap between adjacent frames being M_A.

3. *Frame Windowing*—Each frame is multiplied by an N_A sample hamming window $w(n)$ to minimize the adverse effects of chopping an N_A-sample section out of the running speech signal.

4. *Autocorrelation Analysis*—Each windowed frame of speech samples is autocorrelated with a lag of 10.

5. *LPC/Cepstral Analysis*—The LPC coefficients (Q) of order 10 are computed from the autocorrelation coefficients using Durbin's recursion method, and LPC derived cepstral vectors are computed.

6. *Cepstral Weighting*—The Q-coefficient cepstral vector $C_l(m)$ at time frame 'l' is weighted by a window $W_c(m)$ of the form.

   $W_c(m) = 1 + (Q/2) \sin(\Pi m/Q), \quad 1 \leq m \leq Q$

   to yield the weighted cepstral coefficient as

   $C'_l(m) = C_l(m) \cdot W_c(m)$

7. *VQ*—For a HMM with discrete observation symbol density, VQ is required to map each continuous observation vector into a discrete codebook index. Once the codebook of vectors has been obtained, the mapping between continuous vectors and codebook indices becomes a simple nearest neighbor computation, that is, the continuous vector is assigned the index of the nearest (in a spectral distance sense) codebook vector. Thus, the major issue in VQ is the design of an appropriate codebook for quantization. We have taken codebook size as 128.

Figure 3 shows the hardware design block diagram.

Figure 3. Hardware Design Block Diagram
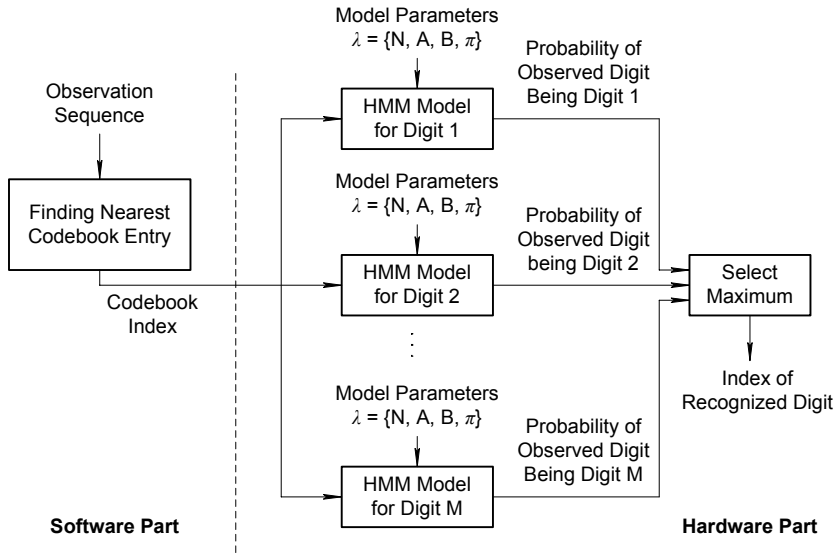
The Hidden Markov Model (HMM)

Recognition is achieved by maximizing the probability of the digit W, given the acoustic evidence, A, that is, choose the digit W such that

$$P(\hat{W}|A) = \max_{W} P(W|A)$$

Elements of a Discrete Hidden Markov Model

N is the number of states in the model

M is the number of distinct observation symbols per state. We denote the individual symbols as

$$V = \{v_1, v_2, \cdots, v_M\}.$$

$A = \{a_{ij}\}$, the state transition probability distribution where

$$a_{ij} = P[q_{t+1} = S_j | q_t = S_i], \qquad 1 \le i, j \le N.$$

q_t is the state of the HMM at time t

B is the observation symbol probability distribution in state j, $B = \{b_j(k)\}$, where

$$b_j(k) = P[v_k \text{ at } t | q_t = S_j], \qquad 1 \le j \le N \qquad 1 \le k \le M.$$

π is the initial state distribution $\pi = \{ \pi_i \}$ where

$$\pi_i = P[q_1 = S_i], \quad 1 \le i \le N.$$

Given appropriate values of N, M, A, B, and π, the HMM can be used as a generator to give an observation sequence

$$O = O_1 O_2 \cdots O_T$$

where each observation O_t is one of the symbols from V, and T is the number of observations in the sequence. A complete specification of an HMM requires specification of two model parameters (N and M), specification of observation symbols, and the specification of the three probability measures A, B, and π.

Design Methodology

We used an Altera UP3 board using a Cyclone™ EP1C6Q240C8 FPGA with a Nios II soft-core CPU. The board has 128 Mbytes of SRAM and 8 Mbytes of SDRAM, and we used both memories for our design. The SDRAM is particularly important because the SRAM alone cannot handle our data and program (Refer to Snapshot 6 in the Appendix). We used a PLL for clock input to the SDRAM for clock skew minimization. The PLL was generated using the Altera MegaWizard® Plug-In Manager.

Design Flow in Training

First, the input speech samples were preprocessed to extract the feature vectors. Then the codebook was built, serving as the reference code space with which we compare the input feature vectors. We have worked with both K-Means and Binary split algorithms and we decided to use the Binary split algorithm in our final design since it is more efficient. Then for training the HMMs, the same weighted cepstrum matrices for various users and digits are compared with the codebook, and their corresponding nearest codebook vector indices are sent to the Baum-Welch algorithm for training a input index sequence model. The Baum-Welch model is an iterative procedure and we have kept the iteration limit as 20. We now have three models for each digit corresponding to the three users in our vocabulary set, and we average the A, B, and π matrices over the users to generalize it. For the design to recognize the same digit uttered by a user for whom the design has not been trained, the zero probabilities in the B matrix have been replaced by a low value so that on recognition it gives a non-zero value. This overcomes the problem of less training data to some extent. Training is done in software and we have included the speech samples required for the software design as arrays. See Figure 4.

Figure 4. Training
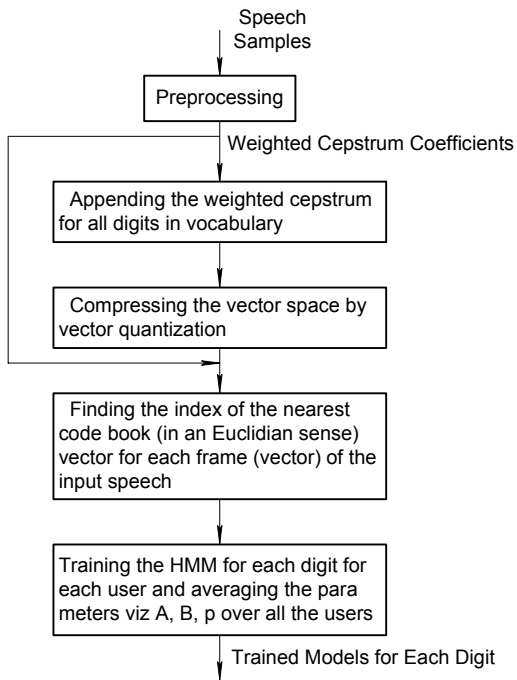
Design Flow for Recognition

The input speech sample is preprocessed to get extract the feature vector. Then the index of the nearest codebook vector for each frame is sent to all the digit models out of which the model giving the maximum probability is chosen. Viterbi is computation intensive, so the processing steps in it have been ported to the FPGA for better speed of execution. After the soft-core processor has completed all the preprocessing steps, the required data is passed to the hardware to do the rest of the processing. Data is through the dataa and datab ports and the prefix port used for the control operations. (Refer Snapshots 2a and b in the Appendix.) See Figure 5.

Figure 5. Data Processing
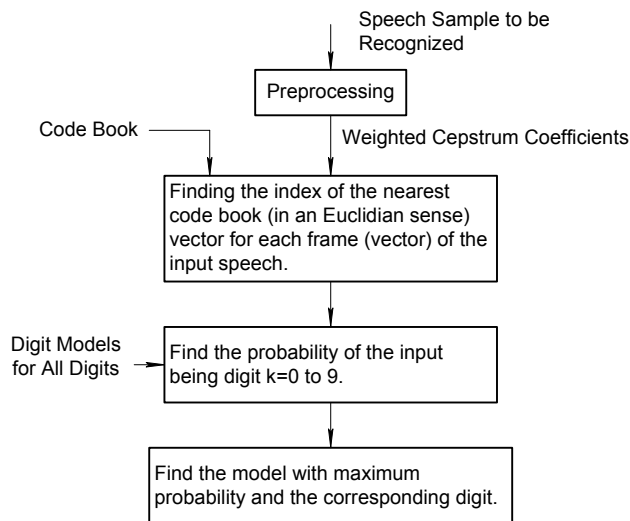
Implementation Summary

Number of states of the HMMs, N: 10

Codebook size, M: 128

Order of LPC, P: 10

Number of weighted cepstrum coefficients per frame vector, Q: 11

Number of digits: 0 – 9

Speech sampling rate: 8 KHz, PCM encoded

End point detection (VAD): short-time energy-based thresholding

Feature analysis: LPC analysis

HMM training for multiple observation sequence: ensemble training (direct parameter averaging)

Design Features

Performance comparison of hardware and full software implementation could be done with the help of the facility to measure the running time of the code (Refer to comparison Snapshots 4 and 5 in the Appendix).

The HMM is rich with mathematical structure as the training of the model uses the Baum-Welch algorithm and the recognition decoding employs the Viterbi algorithm. Hence, these algorithms can be efficiently implemented using FPGAs.

The Nios II processor enables the optimum sharing of hardware and software implementations by executing more computation intensive tasks in hardware and the remaining algorithm blocks in software.

Implementation issues include:

■ The Viterbi design was too big to be implemented as such in the Cyclone device. So we decided to implement only the main processing part (i.e., the floating point multiplication) in hardware. We included the synthesis report of the whole Viterbi block alone using the LeonardoSpectrum™ tool using another APEX™ FPGA (Refer to Snapshot 3 in the Appendix).

■ Scaling. As T becomes sufficiently large, the range of the multiplication factors starts to reach exponentially to zero. The basic scaling procedure is used is to multiply these factors by a scaling coefficient so that they do not exceed the precision range of the machine. Over and above this, since all the values in the Baum-Welch and Viterbi algorithms are probabilities, maintaining is of utmost importance for proper results. Therefore, we have used a floating-point data format in the C program with a structure (mantissa and exponent) and used a normalization function, which removes the leading zeros and accordingly adjusts the exponent whenever it is called.

■ Initial Estimates of HMM Parameters. There is no simple or straightforward answer to the above question. Either random (subject to the stochastic and the nonzero value constraints) or uniform initial estimates of the π and A parameters have to be used. However, for the B parameters, good initial estimates are helpful in the discrete symbol case. We have used uniform initial values for A and B and random values for π.

■ Insufficient Training Data. The observation sequence used for training is finite. Thus there is often an insufficient number of occurrences of different model events (e.g., symbol occurrences within states) to give good estimates of the model parameters. One solution to this problem is to increase the size of the training observation set, which is often impractical. A second possible solution is to reduce the size of the model (e.g., number of states, number of symbols per state, etc) at the expense of the recognition rate. For the design to recognize a digit in the vocabulary uttered by a user for whom the design has not been trained, the zero probabilities in B matrix have been replaced by a low value so that on recognition it gives a non-zero value. This implementation overcomes the problem of less training data to some extent.

Conclusion

We learned that the Nios II processor is powerful enough to implement a full speech recognition process. Its memory and logic capabilities enabled us to implement the design. We implemented the whole design in C++ in a PC environment to check the functionality. We then ported it to the FPGA environment. That gave us a good knowledge of the issues involved in porting a software code to the SoC environment. Finally, we learnt how to include custom block in a design, how to communicate with it from the soft-core processor, what the problems are that may crop up in such a process, and how to solve them.

Probable Future Improvements

■ Implementing floating point vector dot product as custom block in the short term.

■ Optimizing the Viterbi part so that it can be fitted into the Cyclone device.

■ Implementing other computation intensive tasks in the design like LPC processing in hardware to improve recognition time.

■ Using Bakis model, which is claimed to model speech signals better.

■ Using flash memory for using file read/write functions in the soft-core processor itself.

■ Accessing SDRAM from software and hardware to pass large data, which is our requirement.
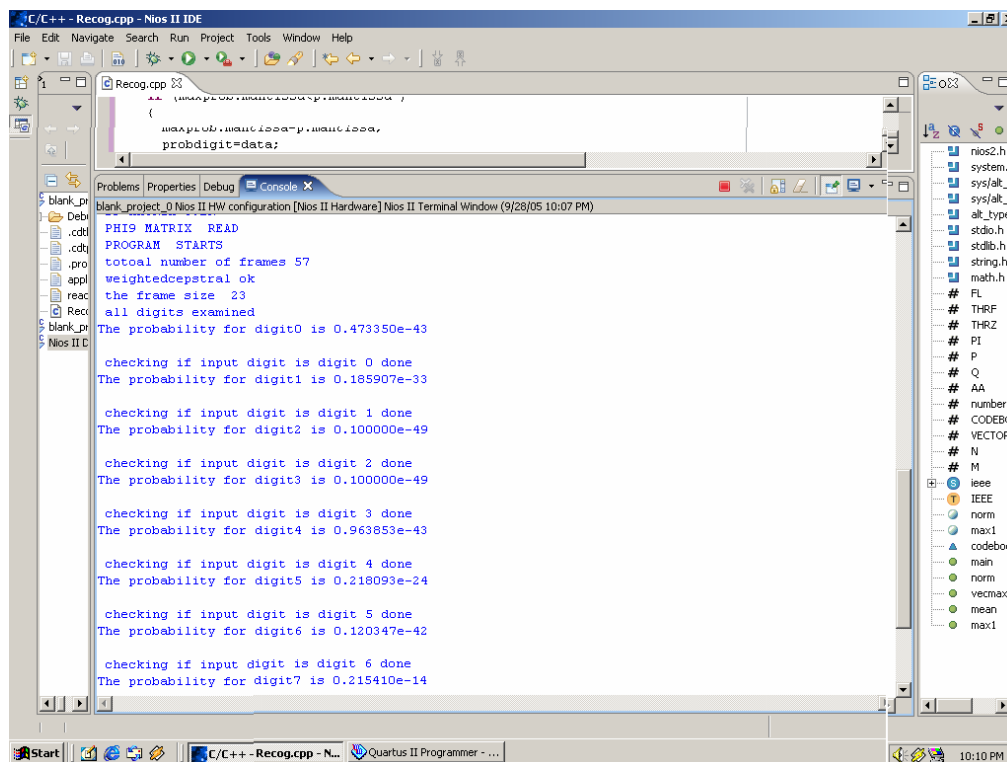
Results

The hardware and software results were verified and were found to match. The process of software/hardware co-design using SOPC is much different than conventional methods such as using microprocessor-based software routines alone or ASIC/FPGA-based full hardware implementation alone. It helped us to achieve the best of both worlds. The ability to include our hardware as a custom design in the FPGA and calling it from the software using custom instructions provided added flexibility to our design. We identified the computation intensive blocks in the design and were able to port it to the hardware for better speed. The soft IP cores helped us speed up our design time. We learned to pass data from the soft core to the hard core processor and tackling the issues in the process helped us to gain better understanding of the Nios II processor.

Appendix: Implementation Snapshots

Snapshot 1. Fitter Summary for Custom Block

MAX+PLUS II	File	Edit	View	Project	Assignments	Processing	Tools	Window	Help

For_UP3

For_UP3.bdf — Compilation Report - Fitter Resource

Fitter Resource Usage Summary

	Resource ▽	Usage
1	Virtual pins	0
2	User inserted logic elements	0
3	Total memory bits	20,160 / 92,160 (21 %)
4	Total logic elements	4,233 / 5,980 (70 %)
5	Total fan-out	19341
6	Total RAM block bits	41,472 / 92,160 (45 %)
7	Total LABs	518 / 598 (86 %)
8	M4Ks	9 / 20 (45 %)
9	Maximum fan-out node	SYSTEM_CLK
10	Maximum fan-out	1610
11	Logic elements in carry chains	786
12	Logic elements by mode	
13	Logic element usage by number of LUT inputs	
14	I/O pins	48 / 185 (25 %)
15	Global signals	8
16	Global clocks	8 / 8 (100 %)
17	Average fan-out	4.50
18	-- 4 input functions	1913
19	-- 3 input functions	1571
20	-- 2 input functions	468
21	-- 1 input functions	179
22	-- 0 input functions	102
23	-- synchronous clear/load mode	685
24	-- register cascade mode	0
25	-- qfbk mode	353
26	-- normal mode	3493
27	-- asynchronous clear/load mode	1501
28	-- arithmetic mode	740
29	-- Register only	198
30	-- Combinational with no register	2464
31	-- Combinational with a register	1571
32	-- Clock pins	1 / 2 (50 %)
33		
34		

Snapshot 2a. Recognition Process for Digit 7 Uttered by User 1 as Input

Snapshot 2b. Recognition Process for Digit 7 Uttered by User 1 as Input

Snapshot 3. Synthesis Report of Whole Viterbi Block Implemented in EP20K1000EFC896 Device

```
Info: Attempting to checkout a license to run as LeonardoSpectrum Level 1 Altera
Info: License passed
Session history will be logged to file 'C:/Exemplar/LeoSpec/v19991j/bin/win32/exemplar.his'
Info, Working Directory is now 'C:\Exemplar\LeoSpec\v19991j\bin\win32'
->set _xmp_enable_renoir FALSE
FALSE
Info: system variable EXEMPLAR set to "C:\Exemplar\LeoSpec\v19991j"
Info: Loading Exemplar Blocks file: C:\Exemplar\LeoSpec\v19991j\data\xmplrblks.ini
Messages will be logged to file 'C:/Exemplar/LeoSpec/v19991j/bin/win32/exemplar.log'...
LeonardoSpectrum Level 1 Altera - v1999.1j (build 6.108, compiled Apr  6 2000 at 12:57:38)
Copyright 1990-1999 Exemplar Logic, Inc.  All rights reserved.

--
-- Welcome to LeonardoSpectrum Level 1 Altera
-- Run By ecad@VLSI-33
-- Run Started On Fri Sep 23 12:13:10 India Standard Time 2005
--
No constraint for register2register
No constraint for input2register
No constraint for input2output
No constraint for register2output
->set register2register 1073741824.0
1073741824.0
->set input2register 1073741824.0
1073741824.0
->set register2output 1073741824.0
1073741824.0
->set input2output 1073741824.0
1073741824.0
->_gc_read_init
->_gc_run_init
->set input_file_list { "D:/nios_modelsim/viterbi_ver4.v" }
 "D:/nios_modelsim/viterbi_ver4.v"
->set part EP20K1000EFC896
EP20K1000EFC896
->set process 1
1
->set flex_use_cascades TRUE
TRUE
->set chip TRUE
```

```
->set macro FALSE
FALSE
->set area TRUE
->set delay FALSE
FALSE
->set report brief
brief
->set hierarchy_auto TRUE
TRUE
->set output_file "C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.edf"
C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.edf
->set novendor_constraint_file FALSE
FALSE
->set target apex20e
apex20e
->_gc_read
-- Reading target technology apex20e
Reading library file `C:\Exemplar\LeoSpec\v19991j\lib\apex20e.syn`...
Library version = 1.6
Delays assume: Process=1
-- read -tech apex20e { "D:/nios_modelsim/viterbi_ver4.v" }
-- Reading file 'D:/nios_modelsim/viterbi_ver4.v'...
-- Loading module viterbi_ver4
-- Compiling root module 'viterbi_ver4'
"D:/nios_modelsim/viterbi_ver4.v",line 92: Info, Enumerated type _STATE_NAME_ with 28 elements encoded as onehot.
Encodings for _STATE_NAME_ values
        value    _STATE_NAME_[27-0]
==================================
      _STATE_0  --------------------------1
      _STATE_1  -------------------------1-
      _STATE_2  ------------------------1--
      _STATE_3  -----------------------1---
      _STATE_4  ----------------------1----
      _STATE_5  ---------------------1-----
      _STATE_6  --------------------1------
      _STATE_7  -------------------1-------
      _STATE_8  ------------------1--------
      _STATE_9  -----------------1---------
      _STATE_10 ----------------1----------
      _STATE_11 ---------------1-----------
      _STATE_12 --------------1------------
      _STATE_13 -------------1-------------
      _STATE_14 ------------1--------------
      _STATE_15 -----------1---------------
      _STATE_16 ----------1----------------
      _STATE_17 ---------1-----------------
      _STATE_18 --------1------------------
      _STATE_19 -------1-------------------
      _STATE_20 ------1--------------------
      _STATE_21 -----1---------------------
      _STATE_22 ----1----------------------
      _STATE_23 ---1-----------------------
      _STATE_24 --1------------------------
      _STATE_25 -1-------------------------
      _STATE_26 1--------------------------
      _STATE_27 1--------------------------

"D:/nios_modelsim/viterbi_ver4.v",line 164: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 215: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 248: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 307: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 285: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 307: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 343: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 378: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 440: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 417: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 440: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 417: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 440: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 487: Error, static loops are not allowed in implicit state machines.
No constraint for register2register
No constraint for input2register
No constraint for input2output
No constraint for register2output
->_gc_read_init
->_gc_run_init
->set input_file_list { "D:/nios_modelsim/viterbi_ver4.v" }
 "D:/nios_modelsim/viterbi_ver4.v"
->set chip TRUE
->set macro FALSE
FALSE
->set delay FALSE
FALSE
->set hierarchy_auto TRUE
TRUE
->set output_file "C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.edf"
C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.edf
->set sdf_write_flat_netlist TRUE
TRUE
->set target apex20e
apex20e
->_gc_read
-- Reading target technology apex20e
Reading library file `C:\Exemplar\LeoSpec\v19991j\lib\apex20e.syn`...
Library version = 1.6
```

```
Delays assume: Process=1
-- read -tech apex20e { "D:/nios_modelsim/viterbi_ver4.v" }
-- Reading file 'D:/nios_modelsim/viterbi_ver4.v'...
-- Loading module viterbi_ver4
-- Compiling root module 'viterbi_ver4'
"D:/nios_modelsim/viterbi_ver4.v",line 92: Info, Enumerated type _STATE_NAME_ with 28 elements encoded as onehot.
Encodings for _STATE_NAME_ values
        value    _STATE_NAME_[27-0]
================================
        _STATE_0  --------------------------1
        _STATE_1  -------------------------1-
        _STATE_2  ------------------------1--
        _STATE_3  -----------------------1---
        _STATE_4  ----------------------1----
        _STATE_5  ---------------------1-----
        _STATE_6  --------------------1------
        _STATE_7  -------------------1-------
        _STATE_8  ------------------1--------
        _STATE_9  -----------------1---------
        _STATE_10 ----------------1----------
        _STATE_11 ---------------1-----------
        _STATE_12 --------------1------------
        _STATE_13 -------------1-------------
        _STATE_14 ------------1--------------
        _STATE_15 -----------1---------------
        _STATE_16 ----------1----------------
        _STATE_17 ---------1-----------------
        _STATE_18 --------1------------------
        _STATE_19 -------1-------------------
        _STATE_20 ------1--------------------
        _STATE_21 -----1---------------------
        _STATE_22 ----1----------------------
        _STATE_23 ---1-----------------------
        _STATE_24 --1------------------------
        _STATE_25 -1-------------------------
        _STATE_26 -1-------------------------
        _STATE_27 1--------------------------

"D:/nios_modelsim/viterbi_ver4.v",line 216: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 249: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 308: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 286: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 308: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 344: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 379: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 441: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 418: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 441: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 418: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 441: Error, static loops are not allowed in implicit state machines.
"D:/nios_modelsim/viterbi_ver4.v",line 488: Error, static loops are not allowed in implicit state machines.
No constraint for register2register
No constraint for input2register
No constraint for input2output
No constraint for register2output
->_gc_read_init
->_gc_run_init
->set input_file_list { "D:/nios_modelsim/viterbi_ver4.v" }
 "D:/nios_modelsim/viterbi_ver4.v"
->set chip TRUE
->set macro FALSE
FALSE
->set delay FALSE
FALSE
->set hierarchy_auto TRUE
TRUE
->set output_file "C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.edf"
C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.edf
->set target apex20e
apex20e
->_gc_read
-- Reading target technology apex20e
Reading library file `C:\Exemplar\LeoSpec\v19991j\lib\apex20e.syn`...
Library version = 1.6
Delays assume: Process=1
-- read -tech apex20e { "D:/nios_modelsim/viterbi_ver4.v" }
-- Reading file 'D:/nios_modelsim/viterbi_ver4.v'...
"D:/nios_modelsim/viterbi_ver4.v", line 526: Warning, system task enable ignored for synthesis
-- Loading module viterbi_ver4
-- Compiling root module 'viterbi_ver4'
"D:/nios_modelsim/viterbi_ver4.v",line 93: Info, Enumerated type _STATE_NAME_ with 28 elements encoded as onehot.
Encodings for _STATE_NAME_ values
        value    _STATE_NAME_[27-0]
================================
        _STATE_0  --------------------------1
        _STATE_1  -------------------------1-
        _STATE_2  ------------------------1--
        _STATE_3  -----------------------1---
        _STATE_4  ----------------------1----
        _STATE_5  ---------------------1-----
        _STATE_6  --------------------1------
        _STATE_7  -------------------1-------
        _STATE_8  ------------------1--------
        _STATE_9  -----------------1---------
        _STATE_10 ----------------1----------
        _STATE_11 ---------------1-----------
```

159

```
      _STATE_12  ---------------1-----------
      _STATE_13  --------------1------------
      _STATE_14  -------------1-------------
      _STATE_15  ------------1--------------
      _STATE_16  ----------1----------------
      _STATE_17  ---------1-----------------
      _STATE_18  --------1------------------
      _STATE_19  -------1-------------------
      _STATE_20  ------1--------------------
      _STATE_21  ------1--------------------
      _STATE_22  -----1---------------------
      _STATE_23  ----1----------------------
      _STATE_24  ---1-----------------------
      _STATE_25  --1------------------------
      _STATE_26  -1-------------------------
      _STATE_27  1--------------------------
```

```
"D:/nios_modelsim/viterbi_ver4.v", line 39:Info, D-Flipflop reg_si(237)(8) is unused, optimizing...
"D:/nios_modelsim/viterbi_ver4.v", line 39:Info, D-Flipflop reg_si(238)(8) is unused, optimizing...
"D:/nios_modelsim/viterbi_ver4.v", line 39:Info, D-Flipflop reg_si(239)(8) is unused, optimizing...
Info: Finished reading design
->_gc_run
-- Run Started On Fri Sep 23 13:24:10 India Standard Time 2005
--
-- optimize -target apex20e -effort quick -chip -area -hierarchy=auto
Using default wire table: apex20e_default
Warning, View .work.viterbi_ver4.INTERFACE needs partitioning, you may want to optimize this block in "auto
dissolve" hierarchy mode...
Warning, View .work.viterbi_ver4.INTERFACE needs partitioning, you may want to optimize this block in "auto
dissolve" hierarchy mode...
Warning, View .work.cx0.partition_vx0 needs partitioning, you may want to optimize this block in "auto dissolve"
hierarchy mode...
-- Start optimization for design .work.cx2.partition_vx0
Using default wire table: apex20e_default
            est est
      Pass   LCs Delay DFFs TRIs  PIs POs    --CPU--
                                              min:sec
       1     6961  106 1809    0  810 688     07:07
-- Start optimization for design .work.cx0.partition_vx0
Using default wire table: apex20e_default
            est est
      Pass   LCs Delay DFFs TRIs  PIs POs    --CPU--
                                              min:sec
       1     8351  116 2411    0  467 545     07:43
-- Start optimization for design .work.cx1.partition_vx0
Using default wire table: apex20e_default
            est est
      Pass   LCs Delay DFFs TRIs  PIs POs    --CPU--
                                              min:sec
       1     6087   73 1472    0  606 331     01:56
-- Start optimization for design .work.viterbi_ver4.INTERFACE
Using default wire table: apex20e_default
            est est
      Pass   LCs Delay DFFs TRIs  PIs POs    --CPU--
                                              min:sec
       1     7435   58 1767    0    2  18     03:24
Using default wire table: apex20e_default
-- Start timing optimization for design .work.viterbi_ver4.INTERFACE
No critical paths to optimize at this level

********************************************************

Cell: viterbi_ver4   View: INTERFACE   Library: work

********************************************************

 Number of ports :                    20
 Number of nets :                  31206
 Number of instances :             29214
 Number of references to this view:     0

Total accumulated area:
 Number of GND:                        4
 Number of I/Os:                      20
 Number of LCs:                    28938
 Number of Memory Bits:             7680
 Number of VCC:                        1

**********************************************
Device Utilization for EP20K1000EFC896
**********************************************
Resource           Used    Avail   Utilization
----------------------------------------------
I/Os                  20     896       2.23%
LCs                28938   38400      75.36%
Memory Bits         7680  540672       1.42%

----------------------------------------------
                    Clock Frequency Report

    Clock            : Frequency
    ----------------------------------

    clk              : 12.1 MHz

                    Critical Path Report
```

```
Critical path #1, (unconstrained path)
NAME                                        GATE              ARRIVAL         LOAD
-------------------------------------------------------------------------------------------
clock information not specified
delay thru clock network                                     0.00 (ideal)


reg_i5(7)/regout                            apex20_lcell_normal  0.00  2.30 up          1.45
modgen_add_6752_ix86/combout                apex20_lcell_arithmetic  4.96  7.25 up          2.05
modgen_mux_6996_ix304/cascout               apex20_lcell_normal  2.01  9.26 up     1.22
modgen_mux_6996_ix306/combout               apex20_lcell_normal  0.53  9.79 up     1.22
modgen_mux_6996_ix310/cascout               apex20_lcell_normal  2.04  11.83 up    1.22
modgen_mux_6996_ix312/combout               apex20_lcell_normal  0.53  12.36 up    1.22
ix1500353/cascout                           apex20_lcell_normal  2.01  14.36 up    1.22
ix1500226/combout                           apex20_lcell_normal  0.53  14.89 up    1.22
ix1415575/Y                                 NOT        1.49  16.38 up          1.22
modgen_mux_6996_ix730/combout               apex20_lcell_normal  3.10  19.48 up         1.55
modgen_gt_7005_ix39/combout                 apex20_lcell_arithmetic  4.96  24.43 up         2.05
ix1507344/combout                           apex20_lcell_normal  3.39  27.82 up         1.64
ix6362/Y                                    NOT        4.20  32.02 up          2.05
ix1504380/combout                           apex20_lcell_normal  2.18  34.21 up    1.22
modgen_mux_7532_ix316/cascout               apex20_lcell_normal  1.98  36.19 up    1.22
modgen_mux_7532_ix318/combout               apex20_lcell_normal  0.53  36.71 up    1.22
modgen_mux_7532_ix342/combout               apex20_lcell_normal  2.18  38.90 up    1.22
ix1504445/cascout                           apex20_lcell_normal  1.98  40.88 up    1.22
ix1503646/combout                           apex20_lcell_normal  0.53  41.41 up    1.22
ix15950/Y                                   NOT        1.49  42.90 up          1.22
modgen_mux_7532_ix730/combout               apex20_lcell_normal  3.10  45.99 up         1.55
modgen_gt_7522_ix23/cout                    apex20_lcell_arithmetic  2.01  48.00 up          1.22
modgen_gt_7522_ix25/cout                    apex20_lcell_arithmetic  0.09  48.09 up          1.22
modgen_gt_7522_ix27/cout                    apex20_lcell_arithmetic  0.09  48.18 up          1.22
modgen_gt_7522_ix29/cout                    apex20_lcell_arithmetic  0.09  48.27 up          1.22
modgen_gt_7522_ix31/cout                    apex20_lcell_arithmetic  0.09  48.35 up          1.22
modgen_gt_7522_ix33/cout                    apex20_lcell_arithmetic  0.09  48.44 up          1.22
modgen_gt_7522_ix35/cout                    apex20_lcell_arithmetic  0.09  48.53 up          1.22
modgen_gt_7522_ix37/cout                    apex20_lcell_arithmetic  0.09  48.62 up          1.22
modgen_gt_7522_ix39/combout                 apex20_lcell_arithmetic  0.53  49.15 up          1.90
ix1507591/combout                           apex20_lcell_normal  4.96  54.11 up    2.05
ix1507333/combout                           apex20_lcell_normal  4.90  59.00 up    2.05
ix1501713/combout                           apex20_lcell_normal  3.70  62.70 up    1.73
ix1502628/combout                           apex20_lcell_normal  2.21  64.91 up    1.22
ix1503203/combout                           apex20_lcell_normal  2.16  67.07 up    1.22
ix1507603/combout                           apex20_lcell_normal  2.47  69.53 up    1.34
ix1507602/combout                           apex20_lcell_normal  3.10  72.63 up    1.55
ix1502721/combout                           apex20_lcell_normal  2.21  74.84 up    1.22
ix1503153/combout                           apex20_lcell_normal  2.21  77.05 up    1.22
ix1497841/combout                           apex20_lcell_normal  4.87  81.92 up    2.05
reg_e_delta(5)(7)/ena                       apex20_lcell_normal  0.00  81.92 up    0.00
data arrival time                                            81.92


data required time  (default specified - setup time)         not specified
-------------------------------------------------------------------------------------------
data required time                                           not specified
data arrival time                                            81.92
                                                             ----------
                                                             unconstrained path
-------------------------------------------------------------------------------------------



-- Design summary in file 'C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.sum'
-- Writing file C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.edf
Info, Writing xrf file 'C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.xrf'
-- Writing file C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.xrf
Info, Writing batch file 'C:/Exemplar/LeoSpec/v19991j/bin/win32/viterbi_ver4.tcl'
-- CPU time taken for this run was  3017.14 sec
-- Run Successfully Ended On  Fri Sep 23 14:14:27 India Standard Time 2005
0
Info: Finished Synthesis run
```
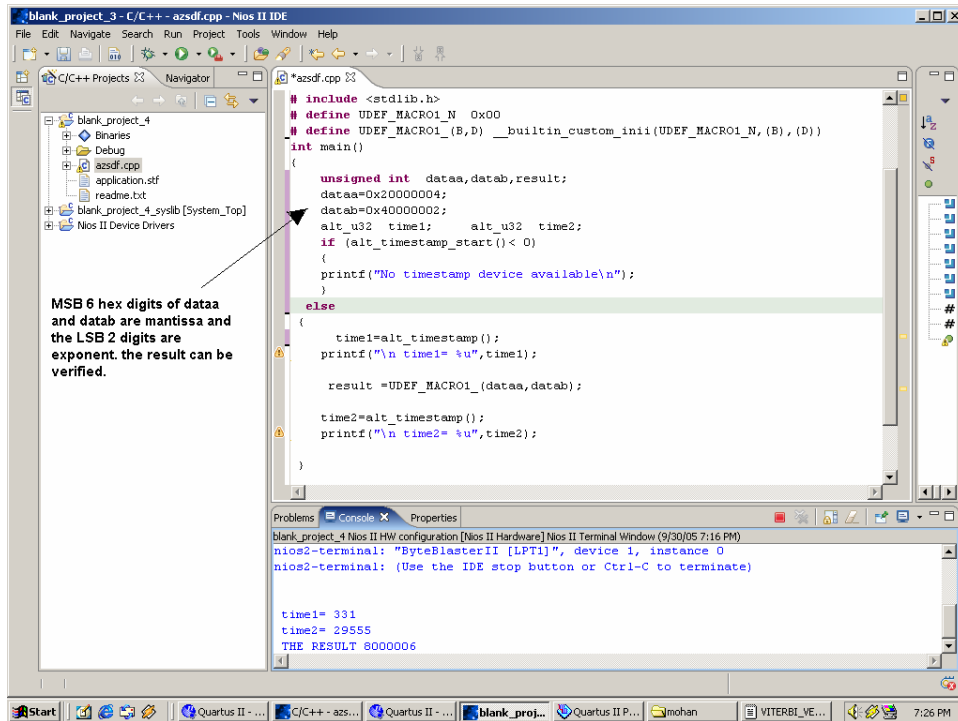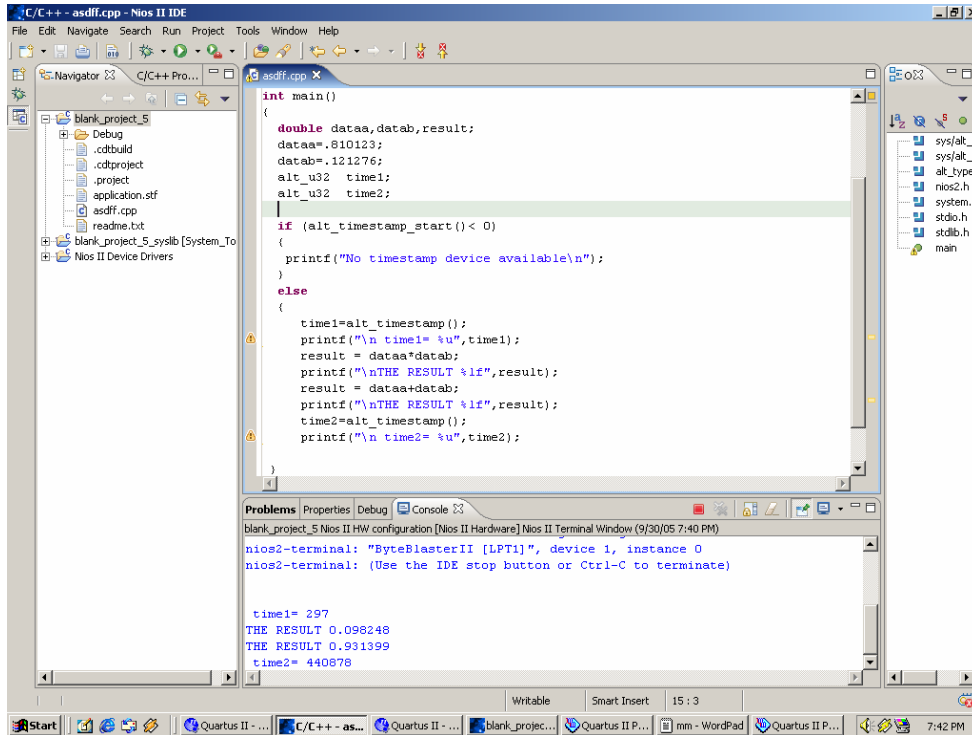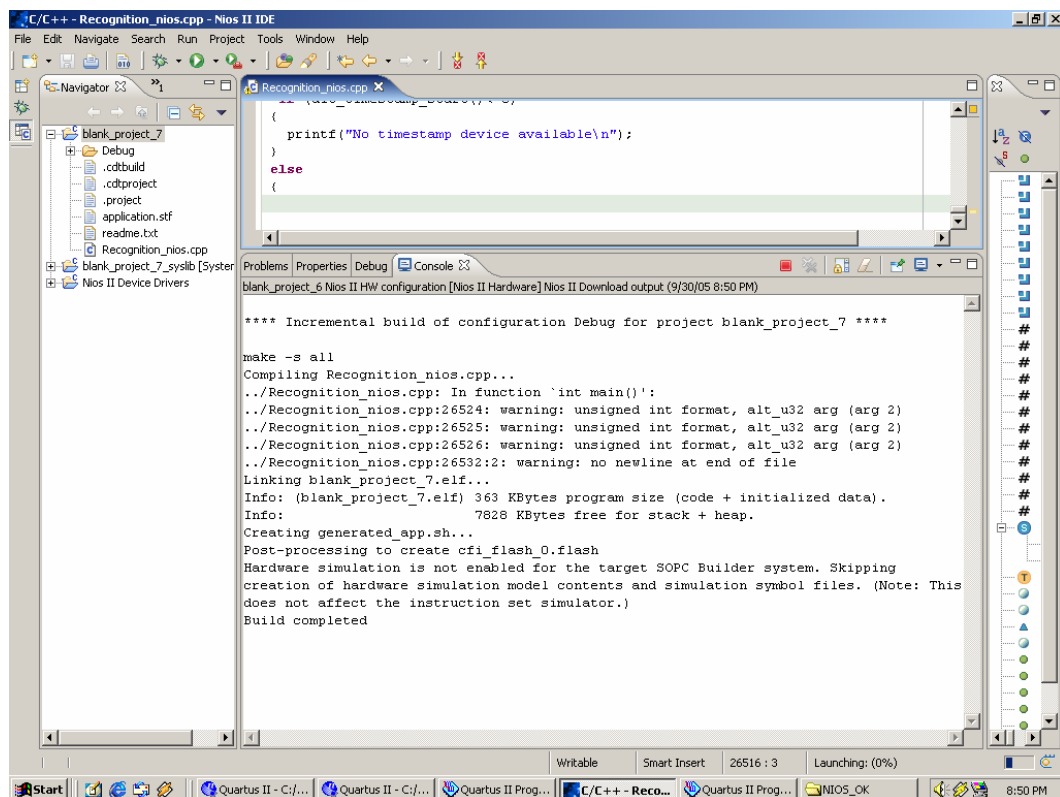
Snapshot 4. Hardware Floating Multiplier Custom Block & its Run Time with System Frequency of 48 MHz (Run Time Mentioned as Number of Clock Ticks)

Snapshot 5. Software Floating Multiplier & its Run Time With System Frequency of 48 MHz (Run Time Mentioned as Number of Clock Ticks)

Snapshot 6. Memory Usage Summary in Nios II IDE

Second Prize

Intelligent Card Technology-Based Biometrics Identification System

Institution: **Institute of Information Science, School of Computer, Beijing JiaoTong University**

Participants: **Tang Hui, Liu Lulu, and Qin Lunming**

Instructor: **Ding Xiaoming**

Design Introduction

The design of a secure-identity authentication system involves a complicated decision-making process that verifies security certificates on-the-fly. When creating this design, we had to make a choice between design complexity and security considerations to arrive at an optimized solution. Combining biometrics recognition with smart card technology helped us design a good security/authentication system. Figure 1, an excerpt from a thesis of the Smart Card Alliance, 2004, illustrates the advantages of this approach.

Figure 1. Relationship between the Smart Card with Biologic Features & Security Level (Palm Print Recognition)
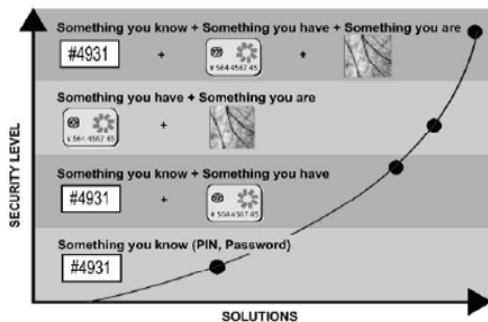
Advances in biometrics recognition and smart card technologies have enabled designers to use them in many applications. We carefully reviewed the status of these two technologies to use the best principles in our design, and then we combined our design knowledge with the Altera® system-on-a-programmable-chip (SOPC) concept. We used the cutting-edge features of the Nios® II soft core processor to develop a Smart Card Biometrics System. For the biometric feature, we used palm print identification because of its usage worldwide and because we have a strong theoretical foundation on the biometrics of palm printing. (We are also familiar with voice print biometrics.) Given this backdrop, we can design richly featured, highly secure identification systems based on either technology. Alternatively, we can combine both palm print and voice print to arrive at a multimode biometric feature smart card.

Application Scope

Combining biometrics recognition with smart card technologies allows high-performance deployment in many areas. For example, biometrics recognition technology can enable an enhanced security level feature for smart card applications. Additionally, using smart card technology saves time in storing and retrieving data from a local database, thus accelerating the matching process. The application scope and solutions that are available today using these two technologies are described as follows:

- *School e-Card*—Today, most schools are still using magnetic cards, which have severe limitations such as short lifespan, limited data volume, and poor security performance. Instead, using a combination of biometrics and smart card technologies, schools can enhance security, data volume, card lifetime, and application scope.

- *Talent Card*—At present, the Beijing Talent Service Center is developing relevant standards for the talent industry, and is going to distribute cards, using smart card technology, for all talents to facilitate planning and management. Security is a big factor because of the importance of the talent cards. Incorporating biometrics into the smart card will undoubtedly ensure high security for the whole system.

- *High-Class Club Member Card*—Member cards of a high-class club are very valuable. Losing a card results in a loss to both the club and the card owner. Combining biometrics recognition with smart card technologies can make member cards more secure.

- *Door Lock Management System*—The biometrics recognition system will be a great advantage in the performance of a door lock secure management system. These systems are traditional security applications in which one can deploy biometrics and smart card technologies to a great advantage.

The users for these applications is decided by the application scope. Our smart card biometrics recognition system has a variety of users ranging from high-end users, such as senior club members, to ordinary users such as school students and residents in a housing colony.

Advantages of Using the Nios II Processor

There are two main advantages of using the Nios II processor: it can be broken into modules and configured. Each subsystem is composed of function modules, which coordinate with each other to form a complete system. The SOPC design approach minimizes the system's reliance on submodules during the design process and increases the cohesion between modules. Therefore, our system design tasks are divided into small modules. Taking this design approach we can rapidly build a complete system with greater precision that meets the design requirements. This design method minimizes the influence of a single module on the whole system.

The Nios II processor features configurable, built-in hardware and software system modules. We can configure the system dynamically for different system requirements during the development stage to optimize system performance under different conditions.

Function Description

In applications involving biometric recognition, the recognition function needs to be executed with precision at high speed for large-scale deployment of the system. The essence of biometric system design is to speed up the processing in each segment while ensuring precise recognition. Based on this concept, our system has distinct features in many aspects. Our system implements the features described in the following sections.

Collecting Biometric Data & Transmission in DMA Mode

A simulated collector collects the biometric data, which saved development time. We also reduced the time required for image data transmission after collection. Because the data is stored in blocks, it can be transmitted in DMA mode. The Nios II processor provides good support for data transmission in DMA mode, which makes it possible for designers to select appropriate transmission mode to meet the design requirements.

Biometric Data Extraction & Compression, DSP Builder & Customized User Instructions

The extraction and compression of biometric data is complicated because it involves digital signal processing (DSP). We had some problems determining which algorithm was more effective when deployed under an embedded system environment. To solve the problem, we tested existing algorithms in a PC environment, and chose the most appropriate algorithm that met the practical requirements for use in an embedded system.

It was quite difficult to achieve real-time processing using software mode while executing the algorithm. Therefore, we used a hardware mode to accelerate processing in key segments of the algorithm to satisfy the system design requirements. Specifically, we used the following approaches:

■   The MATLAB software is easy to use and allows for quick computation of arithmetic operations, so we used it to perform algorithm-level emulation. Next, we generated hardware modules using Altera's DSP Builder tool, and integrated it into our system as the system's DSP module. This approach enhanced system performance.

■   We implemented frequently called basic functions and statements used during processing with the Nios II processor's customizable user instructions. These custom instructions improved the processing speed.

Card Reader/Writer Integration & User-Defined Peripherals

To integrate the smart card reader/writer module, we removed the redundant serial port components and assigned the Rx/Tx pins of the serial port to the extended I/O using SOPC Builder. This scheme made it possible to operate the card reader/writer module and collection module in one daughter board. Then, we integrated the control unit of the card reader/writer to the system bus using customized peripheral feature of the system. In this way, we simplified the control design and ensured faster data processing. Because the biometric feature data in the smart card is stored in blocks, we used DMA transmission mode to reduce processing time during transmission.

Control of a Complex System Using RTOS vs Multi-Core Technology

We tried to avoid a complex control mode scheme using the modularization concept during the control design of the whole system. However, we anticipated that the system complexity might exceed our expectations at the early stages of the design. Fortunately, the Nios II processor solved the problem by making it easy to load the real-time operating system (RTOS) to the system. The management of real time tasks, memory, and peripheral devices using the real time operating system allowed us to take advantage of the capabilities of Nios II processor fully. By doing so, we enhanced the utilization of system resources and processing capability, reduced processing time, and speeded up the system response. In our opinion, you can also consider the multi-core operation in your design.

Performance Parameters

Effective recognition and speed are the most important performance parameters for the biometric identification system. Our goal was to enhance the recognition rate and processing speed. For faster processing, we took advantage of Altera's design platform and reduced recognition speed to an acceptable figure. This figure included the time for a single recognition process (including complex pre-processing, feature extraction, compression, and smart card reading) to within 4 seconds, without any design modification.

To effectively verify the recognition rate, we exercised the system 200 times, each time verifying the palm print samples against a database of palm print samples library. The experimental data indicated that through the selection of an appropriate threshold value, the system can resist all of the 200 attacks.

The excellent performance of the Nios II processor made it easy for us to realize our design goals. We chose the Nios II/f core because we had a strict requirement with regard to the processing speed. Combining the Nios II processor and FPGA design, we developed an ideal mix of processor, peripherals, memory, and I/O interface. This design approach enabled us to meet the design requirements with a high resource utilization: 92% of logic element (LE) usage and 84% of on-chip memory usage.

Using the Nios II processor helped us enhance system performance by modifying function blocks, even amidst design stages. For example, we used a 50-MHz MCLK and the Nios II/s core, but this design required 12 seconds to extract voice prints, which was unacceptable. Next, we used a 75-MHz MCLK and the NiosII/f processor, and added ICACHE and DCACHE functions, which took about 3 seconds to extract the biometrics data and met our design requirements.

We were able to enhance system modules' performance further by using RTOS, DMA, and user-defined instructions from the Nios II processor.

Design Architecture

Taking palm-print recognition as an example, this section describes the design flow and system modules. For other biometric systems, the difference lies in the collection and feature extraction modules. The rest of the modules remain the same, which is a major advantage of using SOPC in a modular design.

The system has two major functions: registration and authentication. In the registration function, the system has to complete at least the extraction and comparison of biometric data and card reader/writer control. During registration, the palm print feature data is extracted through a palm print collection terminal and is stored in the smart card using the card reader/writer module. Figure 2 outlines the registration and authentication system tasks.

Figure 2. Registration Process Diagram of Palm Feature
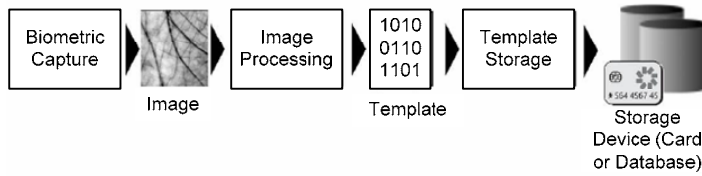
During authentication, both the smart card and related palm print data are displayed. The collected palm print data is compared with the palm print data stored in the smart card in real time through processing of data in collection terminal. The authentication is a success when the two data values are consistent with each other, or else the authentication fails. See Figures 3 through 7.

Figure 3. Authentication Process Diagram of Palm Print Data
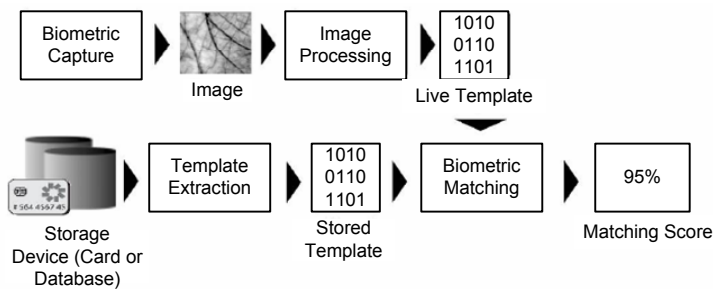
Figure 4. Simple Block Diagram of Biometric System
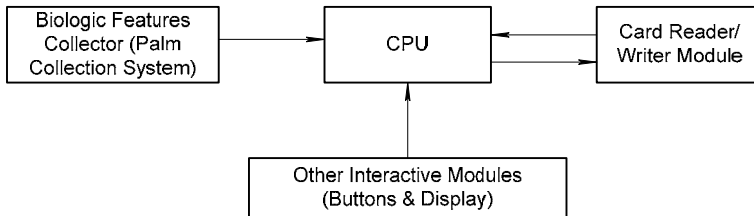
Figure 5. Design Diagram of System Hardware
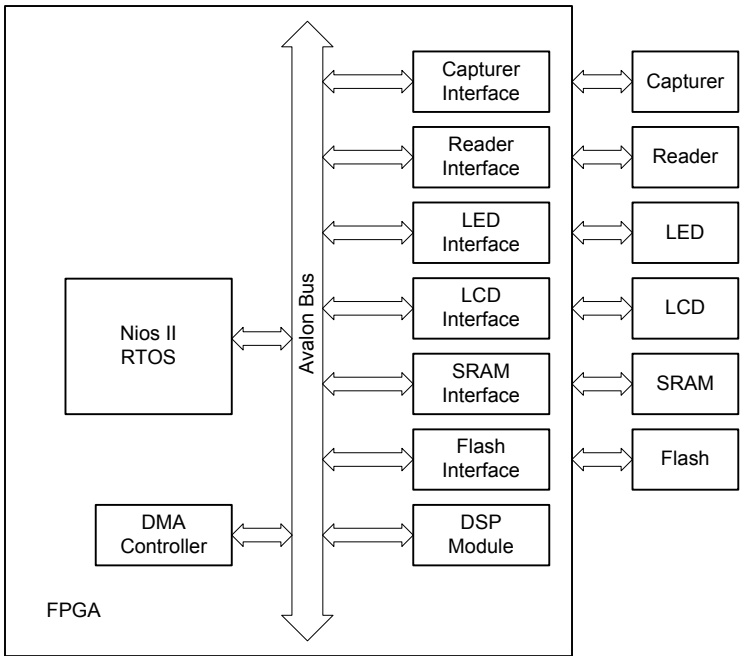
Figure 6. Operating System & Task Distribution Diagram
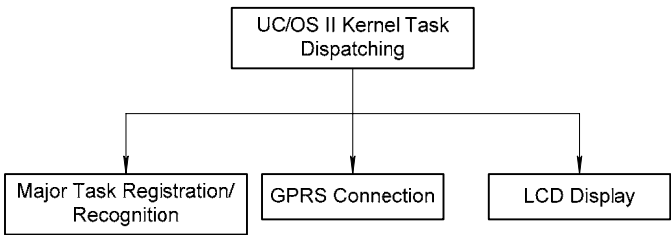
Figure 7. Main Task Workflow Diagram of the System
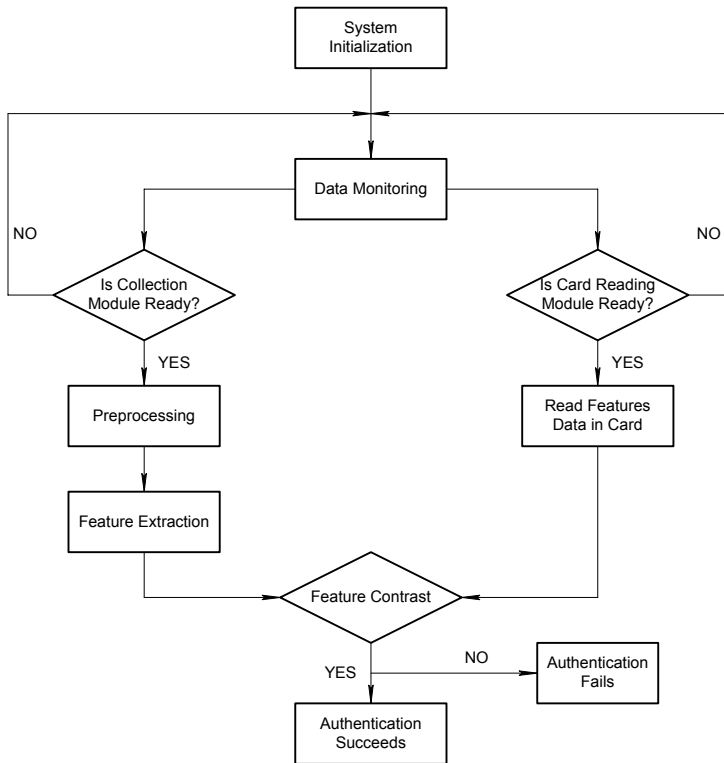
Design Methodology

This section describes the design methodology for the system.

Design Environment

For the purposes of this contest, we wanted to use a common design environment based on UP3 education suites. Based on the UP3 board, we developed and realized many basic modules, such as palm print (voiceprint) collection, preprocessing, feature extraction, and the read/write tasks of the smart card. Due to a lack of sufficient resources on the UP3 board, it was hard to meet the design requirements for system speed, design integration, and hardware processing speed. Therefore, we decided to switch to Altera's development board during the middle phase of the design.

Design of System Hardware

Figure 8 shows the SOPC Builder tool settings.

Figure 8. System Design Configuration

Use	Module Name	Description	Clock	Base	End
☑	⊞ **cpu**	Nios II Processor - Altera Corporation	clk	🔒 0x00800000	0x0
☑	⊞ **jtag_uart**	JTAG UART	clk	**0x008008E0**	0x0
☑	⊞ **ext_flash**	Flash Memory (Common Flash Interface)		🔒 0x00000000	0x0
☑	⊞ **ext_ram**	IDT71V416 SRAM		🔒 0x02000000	0x0
☑	⊞ **ext_ram_bus**	Avalon Tri-State Bridge	clk		
☑	⊞ **sys_clk_timer**	Interval timer	clk	**0x00800800**	0x0
☑	⊞ **high_res_timer**	Interval timer	clk	**0x00800820**	0x0
☑	⊞ **sdram**	SDRAM Controller	clk	🔒 0x01000000	0x0
☑	⊞ **pio_remain**	PIO (Parallel I/O)	clk	**0x00800860**	0x0
☑	⊞ **reset_getimage**	PIO (Parallel I/O)	clk	**0x00800870**	0x0
☑	⊞ **seven_seg_pio**	PIO (Parallel I/O)	clk	**0x00800880**	0x0
☑	⊞ **rdy**	PIO (Parallel I/O)	clk	**0x00800890**	0x0
☑	⊞ **i2c**	OpenCores I2C Master	clk	**0x00800940**	0x0
☑	⊞ **button_pio**	PIO (Parallel I/O)	clk	**0x008008A0**	0x0
☑	⊞ **reconfig_request_pio**	PIO (Parallel I/O)	clk	**0x008008D0**	0x0
☑	⊞ **uart1**	UART (RS-232 serial port)	clk	**0x00800840**	0x0
☑	⊞ **uart2**	UART (RS-232 serial port)	clk	**0x00800900**	0x0
☑	⊞ **cam_buf_bus**	Avalon Tri-State Bridge	clk		
☑	⊞ **cam_buf**	IDT71V424 SRAM		🔒 0x00880000	0x0
☑	⊞ **dma**	DMA	clk	**0x00800920**	0x0
☑	⊞ **onchip_ram_64k**	On-Ch (avalon_tristate) or ROM)	clk	🔒 0x02100000	0x0

Design Implementation

We broke down the system tasks into five parts:

■ Biometric Feature (Data) Collection Module

■ Preprocessing Module

■ Voice Information Preprocessing

■ Feature Data Extraction & Compression

■ Smart Card Read/Write Module

Biometric Feature (Data) Collection Module

The biometric feature collection module is the indispensable front-end, whose quality of data collection directly impacts the recognition effectiveness of the whole system. This module also affects the collection speed, which can undermine system performance if not addressed properly.

In the voiceprint data collection system, data is captured through PC microphone and audio card. For collecting palm print data, we simulated camera collection action and processed this data using the SAA7113 chip. The SAA7113 chip in turn is controlled by I²C bus, whose control module is realized using the customized peripheral function feature available with the Altera SOPC Builder design tool. Also, thanks to the availability of many intellectual property (IP) cores and customized peripherals, you can easily integrate them into your system using SOPC Builder. This is one of the major advantages of SOPC design.

We designed the IP core that controls camera and external storage devices and integrated it into the system; this IP core can complete most of the control operations of palm print image collection. We wrote this IP core in VHDL and its function module is shown in Figure 9.

Figure 9. Front-End Palm Print Collection Control Module
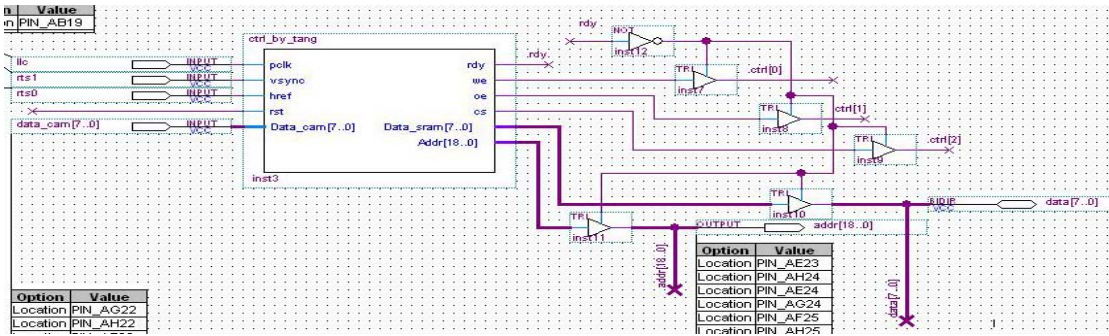
Preprocessing Module

The collected palm print image data contains many kinds of noise and features different palm print sizes, whose image locations may vary based on the angle of exposure. So it is necessary to preprocess the collected image that forms the basis of the palm print feature extraction so it can be compared with stored image data. Therefore, the preprocessing module plays a key role in the system's palm print recognition effectiveness. The preprocessing module we used has a three-step approach:

■ *Area of Interest Positioning*—The positioning algorithm first makes a binary processing of the source image, and then obtains data of the interested area using a geometry morphology algorithm. The existing positioning algorithm is quite complicated, so it is realized mainly with software. Due to the large size of the image, we used hardware to perform binary processing, and subsequently used DMA to transmit the image data to the main storage.

■ *Image Balance*—Because the captured image data is easily affected by light, we performed light-balance processing to set the image data on a grayscale standard. To complete the image balance, we analyzed the image grey histogram, and then processed this data by hardware.

■ *Median Filter*—Median filtering is a common method to remove noise effects present in the signal. We developed a parameterized median filter hardware module that can handle a 2-D median filter on an N[x]N window. In our system, we used a 3[x]3 window to perform median filtering on the interested area to remove noise.

Voice Information Preprocessing

Voice preprocessing includes quantifying, split frame, pre-weighting, breakpoint checking, and removing noise from voice data.

■ We digitized the input simulation voice signal by quantifying and sampling the audio signal. The digitized signal is then downloaded onto the development board flash memory.

■ The voice signal remains stable from 10 ms up to 30 m, which means all algorithms have to process data within this short time period. We chose a 30-ms voice sample as a data frame in our design.

■ Because the voice signal is affected by the glottis pulse shape and lip radiation, the voice-signal spectrum brings down the high-frequency component, which equals to a 6-db drop per octave. To remove this effect, we enhanced the high-frequency component using a pre-weighted, simple, first-order FIR filter, in the form of $H(z)=1-a*z-1$. The pre-weight filter helps to compress the dynamic range of the input voice effectively, and makes the linearity forecast analysis more stable. Moreover, this high-pass filter can also filter the DC component in the input signal.

It is important to determine the voice signal start and end points correctly. Some popular voice parameter data that help to judge voice signal start and end points include short-time energy, short-time average power, and short-time zero-cross rate.

Feature Data Extraction & Compression

We used the I2DPCA algorithm to perform feature extraction on the preprocessed palm print image. The I2DPCA is a proven subspace analysis algorithm whose validity has been verified in laboratory analysis. A major advantage of the I2DPCA algorithm lies in the fact that it helps you to reduce feature dimensions while ensuring a high-recognition rate. Because there is limited storage space on a smart card, the I2DPCA algorithm was a great help in storing the palm and voice feature data. But deploying the I2DPCA algorithm was not easy; it took complex arithmetic operations that included many iterative floating-point operations. To ensure real-time operation of the system, we added custom instructions for addition, subtraction, multiplication, and division operations on floating-point data.

The extraction of voice print data features involves keeping the language content while preserving individual voice characteristics. There are two kinds of voice characteristics: the difference of inborn vocal organs, such as the acoustic duct length and vocal cords and acquired speech characteristics, such as the dialect and tones. Because it is hard to extract these characteristics separately, we stored both.

Presently, the Mel-Frequency Cepstral Coefficients (MFCC) method is extensively used to differentiate speakers' voice feature parameters. The MFCC method involves computing real voice signals, applying a FFT on them, and then convoluting the resulting logarithm energy spectrum with Mel-Scale Triangular Filter. Finally, we carried out a discrete cosine transform (DCT) on the vector composed by filter outputs.

Smart Card Read/Write Module

This module uses ZLG500B from ZLG Corporation. The following table describes the system interface definition.

Pin	Symbol	Type	Description
J2-1	CTRL	Output	Control wire output
J2-2	BZ	Output	Buzzer signal output, high usually, output square-wave or low-level enabled
J2-3	CON485	Output	RS485 control, low usually, high during TXD sending
J2-4	VCC	PWR	Power plus end
J2-5	RST	Reset	MCU reset, high-level enabled
J2-6	GND	PWR	Power minus end
J2-7	RXD	Input	UART receiving end
J2-8	TXD	Output	UART sending end

Adding two serial ports and three PIOs into SOPC Builder completed the hardware design of this module. We had to assign the pins of the serial port to receive/send and set up the baud rate accordingly.

We defined the following control characters in our software design:

```
#define STX          0x20   //Start of Text
#define ACK          0x06   //Affirmative Acknowledgment
#define NAK          0x15   //Not Affirmative Acknowledgment
#define ETX          0x03   //End of Text
```

For communication purposes, the host sends data to the ZLG500B device, and after it executes the command, it returns the state of command execution and relative data to the Nios II processor. The receiving/sending part must be initialized before communication starts. First, the Nios II processor

sends out an STX, and waits for the ACK response from the ZLG500B device. If the Nios II processor does not receive a response within 10 ms or receive a NAK, it sends another STX. It repeats this action three times. Then, the Nios II processor quits the transmission mode and returns an error code to the main program that processes the error. On the other hand, if the Nios II processor receives the ACK response from the ZLG500B device, it sends out a block of data, and transmits an ETX to signal the end of transmission. The following table shows the format of Nios II transmission:

Nios II Processor	Data Transmission Direction	ZLG500B	Description
STX	——>		If the Nios II processor is unable to receive an ACK or NAK within 10 ms, it resends STX at least once; after receiving an ACK, the Nios II processor must send data within 50 ms, and the time interval between 2 bytes sent must be less than 10 ms.
	<——	ACK	
DATA+ETX	——>		

After this, the Nios II processor waits for status data and response from the ZLG500B device. If it does not receive a response within 300 ms, the Nios II processor quits transmission mode and reports an error code to main program. The format of ZLG500B transmission is shown in the following table:

ZLG500B	Data Transmission Direction	Nios II Processor	Description
STX	——>		ZLG500B won't resend STX if it hasn't received ACK within 50ms.
	<——	ACK	
DATA+ETX	——>		

The biometric feature information read from a user's smart card is matched with collected data using the above processes. During matching of biometric features, the algorithm adopts Euclidian distance to represent the distance between two palm print features. First, the palm print feature data in the user's smart card are read into system and matched with palm print features obtained using the described processing tasks. By converting the obtained biometric feature data from a floating-point to fixed-point format, the smart card can process data faster and can conserve storage space.

The SOPC concepts utilized in our design are described as follows:

■ *Modular System Design*—In the early stages of design, we spent extra time partitioning the system design. Design partitioning is useful because it helps to define every task of the system. It also helps to simplify system design, thereby improving the designers' confidence. Based on the partitioned modules, the applications' scope and expandability can be correctly evaluated at the beginning of the design cycle. Therefore, it is possible to initiate marketing tasks during product design while simultaneously working on R&D, which is a great benefit to enterprises because it reduces the product's launch period.

■ *System Integration*—Embedded systems need to have a balanced design that encompasses aspects of product volume, power, and design integration. Therefore, it is important that designers carefully think over integration and balance between integration and cost during the realization of the system. In our design, besides the front-end collection module, all other system functions were completed using a development board, which made it possible to achieve a highly integrated design under the same design goals. It would be very difficult to realize the system without using a soft core processor based on an FPGA design approach.

■ *Diverse Implementation of Different Modules*—The implementation methods are diverse for different modules. For example, the front-end collection module used an IP core for its realization while the preprocessing module relied on software and hardware (processing arithmetic functions based on customized instructions). The palm feature extraction module adopted a combination of software and hardware and the smart card module used built-in peripherals. Thanks to the SOPC methodology, it was quite easy to implement these modules using Altera's development tools.

■ *Diversification of Module Combination*—After attaining the basic design goals of the design, we could easily redesign using the existing modules, if needed. For example, using the µC/OS-II operating system, we added a general packet radio transmission (GPRS) module, and at the same time managed to run the registration recognition program. Thus, we were able to display two subtasks with GPRS on LCD, greatly enhancing the application scope of the system.

Further, the choice of biometric feature can be selected using palm print data collection module or voiceprint module. Or, we can also choose a multi-module biometric feature module, which can handle both voice and palm print data with enhanced security features.

■ *System Upgrade*—Using the SOPC approach, it is fully possible to reconfigure the system both at the beginning and middle of design cycle. If need be, you can also reconfigure the system after completing the design.

Design Features

The following table shows a comparison of system demands versus SOPC design platform features:

	System Demand	**Platform Features**
Hardware	High-speed computing capacity is required in the extraction and compression of biometric features and feature matching. When integrating a smart card reader module with the system, the traditional method of serial port communication will not help in achieving a highly integrated system.	When designing system hardware, the designer can use SOPC Builder development platform to choose between a hardware acceleration module or a DSP module according to system demands so as to meet the special requirements of hardware acceleration. SOPC design is convenient for management and operation of peripherals. For our system, the serial port can be located at the expanded I/O interface easily to realize interconnection between boards. By doing so, you can avoid the serial port's unreliability of long wire connection. Furthermore, the smart card reader module can be integrated into the system as a user-defined peripheral to improve system integration.
Software	Currently, the most efficient extraction and compression algorithms as well as matching of biometric features algorithms are all available on PC and are simulated using simulation tools like MATLAB. The embedded implementation will not be as efficient because of above reasons. The effective and coordinated operation of modules requires the best use of RTOS.	The Nios II processor can be adapted to enhance and reduce system performance according to specifications. For example, we can take advantage of the Nios II processor's flexibility and clipping of data values when comparing the advantages of arithmetic with embedded systems to choose the best option for implementation. Additionally, the FPGA provides advantages in time sequence and logical processing. The custom instructions are crucial for high-speed processing in some applications. In a Nios II system, the RTOS is very easy to deploy, which makes it easy for a designer to handle a complex system. In addition, the multi-processor kernel technology and DMA functions ensure excellent system performance.
Feature upgrades and cost	System developers expect quick launch of products and a longer life-cycle to sharpen the competitive edge and beat rivals. At the same time, users want the latest features in their products. Cost is a crucial element in the development of embedded systems. Therefore, lowering the system cost through design is a key problem for designers at the beginning of the project.	Because the FPGA is a programmable device, the time to market is relatively short. The Nios II processor's flexibility coupled with Altera's integrated development kit, abundant reference designs, powerful hardware development tools (SOPC Builder), and software development tools (Nios II IDE) make it easy to achieve all the expected design goals. The Nios II processor allows easy upgrade of both hardware and software in real time. An FPGA-based system design integrates processor, peripheral, memory, and I/O interface to lower cost, complexity, and power consumption.

Conclusion

Referring to the design requirement documents provided in the contest, we summarized our system design in the following table:

Design Phase	Category	Score	Examples
Design Concept	Complexity	5	The design adopts two tasks of RTOS: high PRI of major task is the processing of biometric features the subtask is the LCD display.
		4	Design uses a DSP algorithm. The palm print recognition uses AOI-orientated algorithm of the lab and I2DPCA feature-based recognition algorithm. Voice print recognition adopts FFT transforms.
		3	Design uses greater than 70% LE/memory utilization (refer to compilation report LE/memory utilization which are 92% and 84%, respectively).
		2	Design uses more than two masters on Avalon® bus (DMA utilization).
	Functionality	5	Design realized customized peripherals and customized instruction on the same chip. These included customized peripherals idt71lv424, I^{2}C control and customized peripherals floating point instructions.
		2	Design connects more than eight peripherals on SOPC bus.
Design Implementation	Completeness	5	Design fulfills complete software implementation and hardware demonstration.
Documentation	Completeness	5	We submitted an integrated design document with diagrams and complete description of the final design.

Throughout this paper, we have described the convenience of using the Nios II processor and an SOPC design approach for embedded designs. Using tools like the Quartus II software and SOPC Builder, as well as embedded debugging software, gave us great confidence while accomplishing our design.

Altera provided us with an excellent design approach—SOPC design based on the Nios II processor and an FPGA as well as popular design tools—Quartus II software and SOPC Builder. At the same time, Altera and many third-party providers supplied plenty of debugging software. We made good use of these tools in our design, which shows our design strength.

Early on in the design cycle we needed to be careful in formulating our design so that we could solve problems with a clear mind. A strong careful beginning also helped avoid problems and is a good design technique. It is natural that we would still encounter some problems, but a solution to these problems enables a 'great leap' in reaching the design goals. Of course, this maturity towards problem solving comes from being exposed to many design approaches and experience.

There are many ways of implementing the design and a specific implementation depends on the design target and ease of implementation. Do abstain from trying to realize all functions in software owing to your familiarity with the C programming language. Also, your familiarity with VHDL may make you overlook publicly available IP resources. Make sure that the SOPC design based on Altera FPGAs calls for a simplified design and emphasizes the idea of system analysis design in an embedded system. This approach is not to be confused with module design, which is a necessary requirement. This is the trend of embedded design today.

The quality of a design team is very important. Good coordination between team members is important to accomplish increasingly complex embedded design systems.

Second Prize

Real-Time Driver Drowsiness Tracking System

Institution: **School of Electronic and Information, South China University of Technology**

Participants: **Wang Fei, Cheng Huiyao, Guan Xueming**

Instructor: **Qin Huabiao**

Design Introduction

Our real-time drowsiness tracking system for drivers monitors the alertness status of drivers during driving. The system judges whether the driver is distracted or dozing, and monitors the driver's continuous working hours, vehicle speed, and other information. When the system finds extreme fatigue or other abnormal conditions with the driver's behavior it alerts with a voice warning.

Drowsy driving is an invisible killer for drivers especially while driving on highways. An amazing fact derived from a large number of traffic accidents is that about 20% of traffic accidents are due to drivers' drowsy driving. In addition, drowsy driving is the reason for 22%-30% of severe traffic accidents resulting in death, ranking it as the top of the cause list. So drowsy driving is undoubtedly very dangerous for traffic security. Chinese and international vehicle manufactures are all busy in designing drowsy-driving monitor devices. Most of them have deployed advanced test technology combining embedded systems and intelligent control technology to design their drowsy driving monitor systems, so as to reduce traffic accidents caused by drowsy driving. Our system was designed for the same purpose.

This design adopts the Altera® Nios® II soft core embedded processor and combines image processing and mode identification functions to complete the design of the whole system. We have also used Altera's Cyclone® FPGA to build the necessary peripheral circuits. Taking advantage of the unique self-defined instruction mode of the Nios II processor, we have deployed pure hardware mode to realize image processing algorithms, which greatly improve the system's real-time performance, accuracy and reliability.

The system uses a camera to capture the driver's image, and makes a collaborated analysis to the image information using the processor and an image-processing module before dispatching data to the Nios II processor. The system judges whether the driver is distracted or dozing, and takes appropriate action. For example, it issues a voice warning immediately when the driver is about to doze off. In addition, the system can monitor the driver's continuous driving hours and driving speed. In this case, if the driver has been driving continuously, for example, for five hours or more and/or the driver is over the speed limit set by road management, the system will send appropriate warning signals. Further, the system can also record the driver's status, driving hours, speed and vehicle status and store this information as evidence in case of a traffic accident. In this way, the system can monitor the driver's status in real-time, restrict driver's mistakes while driving, and effectively prevent serious traffic accidents caused by drowsy driving.

The system, which is based on the Altera system-on-a-programmable-chip (SOPC) concept, utilizes the rich logic resources of the Cyclone FPGA to implement complex system-level functions while integrating the Nios II soft core processor. When compared to traditional DSP designs, our SOPC design approach simplifies design complexity, simplifies debugging, and derives a scalable system. The SOPC approach also makes system maintenance and upgrades easy when required.

The system can be used on long distance coaches, trucks, and cars.

Function Description

The following functions comprise the real-time drowsiness tracking system for drivers:

■ Judge driver alertness based on the inputs received during day and night driving conditions and deliver the information to the Nios II processor for further processing

■ Capture the driver's image in real time, then process and analyze it to judge whether the driver's eyes are open or closed or if he is dozing. If the driver is found dozing while driving, a warning voice signal will be sent.

Due to the strict real-time performance requirement of the system, we implemented the image-processing algorithms in hardware. The implementation difficulty and robustness of the algorithm will have to be comprehensively studied when choosing the algorithm. The basic image processing algorithm can be realized in the following three steps (for a detailed algorithm flow, see Figure 1):

1. *Face-Zone Positioning*—Pre-process the original image of the driver to determine the zone of face and capture it. In this way, the image zone, which needs further processing, can be shrunk to speed up the processing. In addition, this technique also lightens the influence of a complex background for face feature judgment.

2. *Eye-Zone Positioning*—Determine the position of eyes in face zone, and capture driver's image eyes zone for further processing.

3. *Eyes Open/Close Judgment*—Analyze the open/close status and winking frequency of the driver's eyes to determine whether the driver is distracted or dozing.

Figure 1 shows the image processing algorithm flow.

Figure 1. Image Processing Algorithm Flow of Real-time Drowsiness Tracking System for Drivers
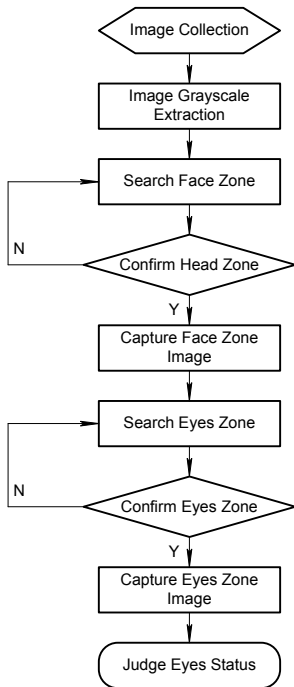
Performance Parameters

The following table shows the correct judgment rate testing.

Test Scenario	Sample Number	Correct Judgment	Threshold Value	Correct Judgment Rate	Notes
Day	80	54	1000	67.5%	
Night	60	43	1100	71.7%	
Uniform Background	80	54	1000	67.5%	Day
Complex Background	80	50	1000	62.5%	Day

The following table shows the program response time.

Transmitting Time of One-Frame Image	Head Check Module Processing Time	Head Check Module Processing Time	Head Check Module Processing Time	Total Response Time
33.3 ms	2.56 ms	2.56 ms	0.27 ms	80 ms

The image collection module includes camera and external SRAM. We have used the digital, color CMOS image sensor OV7620 from US-based OmniVision as the image collection device. The OV7620

is a 300k pixel image sensor that is compact, affordable, powerful, and easy to use. The sensor provides high quality digital images with a 10-bit dual-channel A/D converter, which meets the system's resolution requirements. In addition to being a CMOS image sensor, the OV7620 also features the following items:

■ 326,688 pixel, 1/3-inch sensitive area, VGA/QVGA format.

■ Supplies color and black/white image data in interlace and line scan mode.

■ Supports output digital format: YCrCb4: 2: 2, RGB4: 2: 2 and RGB original image data.

■ Outputs 8- or 16-bit image data in CCIR601, CCIR656, and ZV interface mode.

■ Configurable through internal registers to meet the application image requirements.

■ Output window can be adjusted from 4 x 4 to 664 x 492.

■ Automatic gain and white balance control, 1 - 500 times automatic exposure range.

■ Multiple adjusting functions such as brightness, contrast, gamma correction, and sharpness.

■ Operates from a single 5-V supply, power rating of < 120 mW, standby power loss < 10 uW.

Because the OV7620 outputs 8-bit wide data, the external SRAM is based on the IDT71V424 device that has the same data width as the OV7620. The IDT71V424 buffers the driver's image. The IDT71V424 is a 512-Kbit SRAM device with a read/write period of 10 ns, which meets the system requirement.

Design Architecture

Figure 2 shows the system hardware design.

Figure 2. System Architecture of Real-Time Drowsiness Tracking System for Drivers
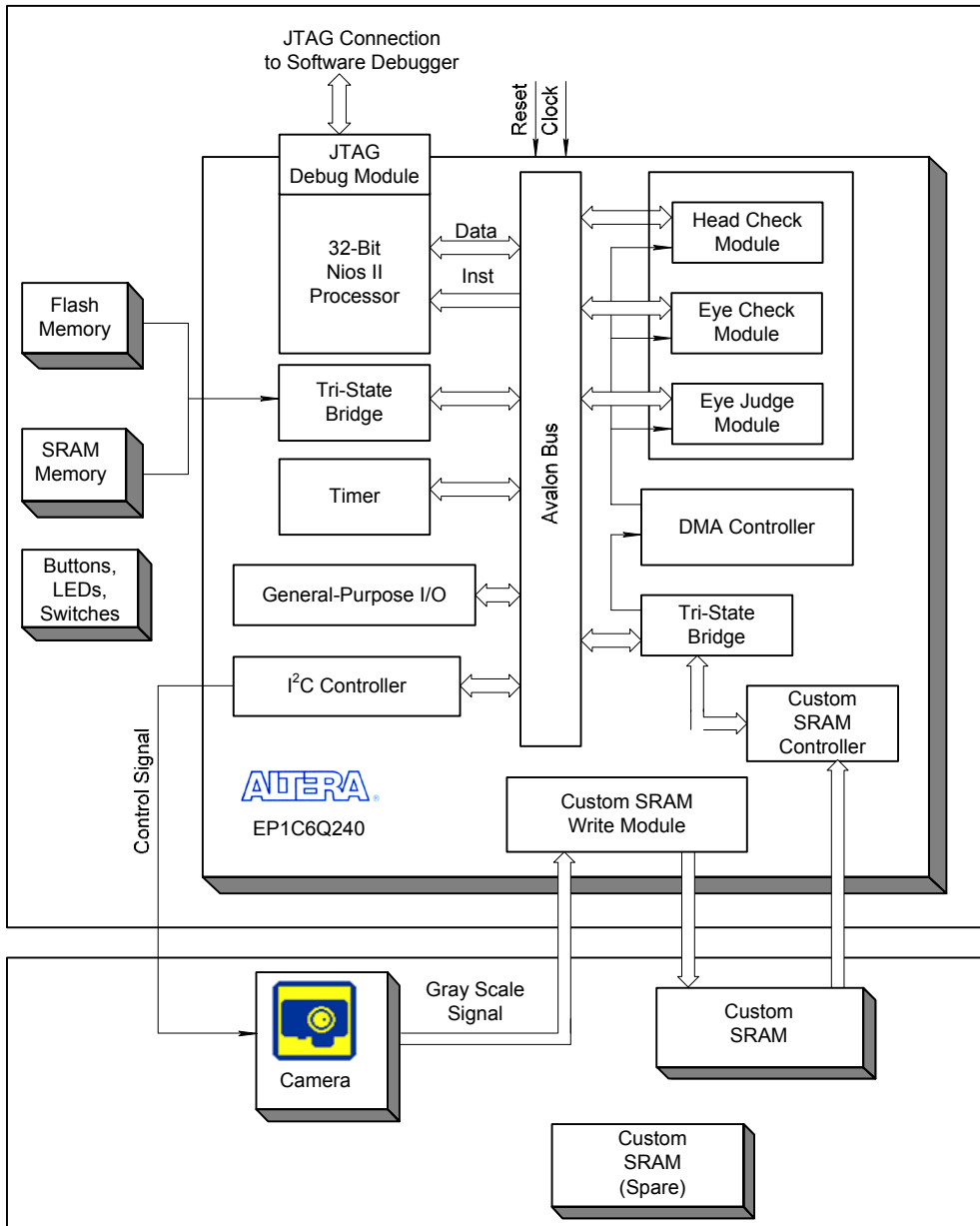
Figure 3 shows the software flow.

Figure 3. Software Flow of Real-Time Drowsiness Tracking System for Drivers
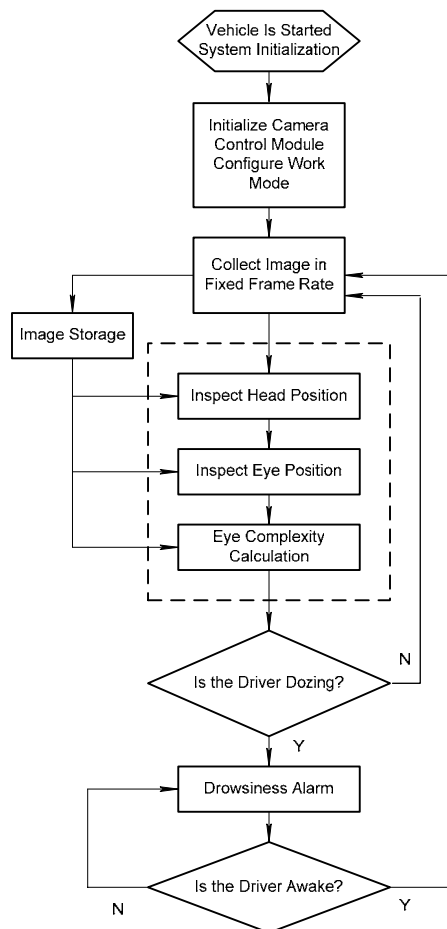
Design Methodology

This section describes the hardware and software design.

Hardware Design of the System

The hardware design consists of the OV7620 control module, the read/write control module of the external SRAM, and the hardware realization of the relevant image processing algorithm (core part).

OV7620 Control Module

We configured the OV7620's internal register with the serial camera control bus (SSCB), to change the camera-operation mode. The SCCB bus, which complies with I^2C Bus specifications, is a popular bus mode used by image sensors, currently. The SCCB bus includes two signal lines, SCL and SDA, which represent the clock and serial data I/O, transmit start, address, stop and read/write data commands.

The OV7620 outputs a 27-MHz synchronization clock with a 320 x 240-resolution image data at 30 frame/second speed. Therefore, it is quite difficult to implement a system at such high data speeds when extracting images by the Nios II processor directly. To solve this problem, we developed an independent OV7620 control module with I^2C Bus control using the C programming language to

configure the camera and read/write the OV7620's control registers. In this way, we can avoid using the SCCB bus. As far as the Nios II processor is concerned, image collection can be realized just by sending start and configuration commands, waiting for the completed signal return from control module, and storing data in external SRAM.

First, the OV7620 is initialized and configured by camera control module via the SCCB bus to operate as a sensor with QVGA format (320 x 240), 30 frame/second, 16-bit, YCrCb4: 2: 2 digital video signal output, default image saturation, brightness and contrast, AGC, and white-balance control. Then the frame synchronization signal VREF, line synchronization signal, and pixel clock signal PCLK, generated by OV7620 are received and are referred to when writing data to external SRAM.

External SRAM Read/Write Control Module

Saving the image data collected by the camera to the external SRAM is a major system design challenge. The image-processing algorithm used by the system has a strict specification with regard to image quality. Further, when we save images in external SRAM, due to cable crosstalk and noise from other components, many transmission errors could occur during SRAM read/write operations.

To solve these problems and enhance system precision, we reduced the system clock from 48 MHz to 30 MHz. Our tests have shown that the system makes almost no mistakes when reading and writing to SRAM at lower clock rates.

As referred to earlier, the pixel clock output by the camera is at 27 MHz. Since the PCB is compact and mounted on a vehicle, even pixel data observed with a 500M-sample logic analyzer appears to be unstable under such a high frequency, let alone the pixel data extracted using the FPGA. Therefore, we decide to configure the camera as a QVGA sensor, 16-bit output, ignoring color difference signal and extracted brightness signal only. In this way, the actual pixel output frequency of the camera gets reduced to 6.75 MHz, and then we can extract stable pixel data using the FPGA.

The SRAM control module, written in Verilog HDL, judges when the data signal is valid according to field synchronization, line synchronization, and pixel clock signal output by the camera, generates the CS, WE, and address signals needed in SRAM writing, and writes pixel data to the appointed SRAM address. Because the Nios II processor and our control module control SRAM, we wrote a multi-path noise suppressor program to assign control signals to SRAM.

Hardware Realization of the Relevant Image Processing Algorithm (Core Part)

The image-processing algorithm of the design was realized in hardware. Using Verilog HDL, all hardware modules were developed and attached to the Avalon® bus as the Nios II processor's peripherals. Each module has its own register and task logic sub-module. The Nios II processor calls the DMA controller to transmit different image data to the three SRAM-based algorithm modules using the following process:

1. The Nios II processor starts the DMA controller, transmits the whole image to the first module (the face check module), responds to the interruption signal from the module after it starts running, and writes operating result from the module.

2. The processor enters the result of the face check module into the second module (the eye check module), starts the DMA controller again, inputs face data, writes the operating module data, finds line position of the smallest point and judges the position of eyes.

3. The processor enters eye data to the third module (the eye-complexity calculation module) via DMA controller. This module outputs the ultimate eye-complexity calculation results to the Nios II processor for judgment.

In the above operation, the threshold values of several operating data in algorithm hardware module are entered by the program in the Nios II processor. These threshold dates were obtained through actual tests while running the system.

System Software Design

The software design comprises camera control and invoking of algorithm modules. The code flow is shown in Figure 3. The flow is as follows:

■ The system software first invokes init_button_pio() function to set up a user interrupt service to start the camera control program after a button-interrupt response.

■ The function init_camera() initializes the camera and sets its working mode by writing data to camera register.

■ The function camera_sent() sends control signals to the camera to send image data.

■ The program writes camera data to SRAM in SRAM control module, and then enters the algorithm module.

■ The function DayCheck() judges whether it is day or night according to the external input and modifies the hardware input parameter with this value.

■ The function HeadCheck(), EyeCheck() and JudgeEye()are three key hardware modules that invoke the following subfunctions, respectively.

■ PhEnable(address) is the write reset signal for entered module address parameter.

■ Sram2Dma_io(sram_start,data_length,ph_address) invokes the DMA controller, and enters data_length long data to algorithm module address of ph_address from the SRAM address of sram_start.

■ Values exchanged between each module are selected according to the output result of DayCheck() function. When set to night conditions, the output MAX_DATA of head check module is selected as the driver's head starting position, and MIN_DATA as the driver's head ending position. This is reversed in a daytime situation. The EyeCheck() function selects the smallest point as eye position based on the day/night entry value, and it is set to the second point at night, and third point at daytime. In addition, the complexity threshold in JudgeEye() is also modified according to the day/night entry.

■ The function SendWarning() controls the lamp according to the judging value. The lamp is on and sends warning if the driver's eyes are closed, and it will be off when the driver's eyes are open.

Our design debugging process proceeded as described below:

Early-term—System programming, algorithm research. We made the plan, specified the functions realized in this plan, and partitioned the functions separately as to be realized in hardware or software modules. We collected the camera materials, researched image identification algorithm, and adopted the most appropriate scheme for the design.

Mid-term—Write program, implement tests. This area was divided into three parts:

1. *Camera control*—Read and writer registers via I²C Bus using the Nios II processor and enter the camera output image to PC for observation and analysis. Adjust the focus, analyze the image quality and brightness, and make preliminary plan on algorithm parameters and threshold figures.

2. *SRAM read/write*—By means of image comparison by the camera and recovered image from SRAM, verify the read/write logic and final clock frequency.

3. *Algorithm modules*—The algorithm plan adopted in this design comprised three modules. A certain number of images were selected, and were divided into two parts: eye open and eye close. MATLAB simulations were made on each modules to confirm the operation results and generate ModelSim-simulated **.txt** document and hardware processed **.bin** document. Compile HDL files and make time sequence simulation on documents generated by MATLAB in a ModelSim environment. Then download **.bin** document to flash storage, read image data (content of **.bin** document) to SRAM by invoking the HAL function, simulate the output data of the camera, invoke DMA to transmit data to run in hardware, and use SignalTap® II to observe the resulting data.

Then, we compared the two simulation results with hardware data output, debugged, and selected the appropriate thresholds.

Late-term—Combined adjusting. Because each part provides image data to the next module in mid-term, each module ran under normal mode under the condition that the image data has met the requirements of the lower-level module. We debugged the system hardware under bright and dark light, day and night conditions in a complex environment to determine the threshold parameters.

Design Features

The design features are as follows:

■ System image processing adopts a pure hardware approach that relies on parallel processing and pipelined technology. Compared to the DSP system that completes corresponding processing algorithms based on serial instructions, our system greatly reduces the speed of image data collection and processing, which meets the real-time requirements of the system.

■ During the processing and judging for the image information, multiple modes of identification algorithm were adopted comprehensively, which made image processing flexible. This approach enhanced system accuracy and reliability.

■ SOPC Builder not only provides abundant intellectual property (IP) resources and parameter selection, but also supports user-defined IP. This design approach helped us to realize the algorithm using the hardware description language method and enabled us to define the IP of the algorithm as special instruction via self-defined instruction mode. We used this approach to invoke software modules when implementing repeated operations during image processing and thus reduced system processing speed.

■ The Nios II system based on an FPGA offers extra design capability and extensibility, which can be used to adjust the algorithm complexity and balance the robustness and real time performance according to application requirements. We can also use this flexibility to load or remove input and output modules and other peripheral devices according to application requirements, such as an on-board entertainment module, anti-theft module, and GPS positioning module.

■ This design adopts DMA as transmission mode, thus ensuring data input data of one pixel in a clock cycle, which remarkably speeds up data transmission. DMA transfers save almost 75% of transmission time when compared with using the Nios II processor for transmission. In addition, DMA does not utilize any bus resource. Therefore, we could use the Nios II processor to implement other functions at the time of DMA transmission.

Conclusion

The Nios II soft core processor represents a brand new concept in embedded design and overturns many of the traditional embedded system design concepts. The three-month contest provided us with a better understanding of this design approach.

Based on the Nios II processor as its core element, the embedded system design platform features remarkable flexibility. SOPC Builder provided us with abundant peripheral resources. The fact that we could write our own peripheral modules freely by adding self-defined logic was of crucial importance to us. For instance, in our real-time drowsiness tracking system for drivers, the I^2C bus module that controls the camera and the module implementing the image processing algorithm were all integrated in the Nios II processor and managed by the Nios II processor to be the peripheral of Avalon bus. Using self-defined logic, instead of writing logic modules that are independent of the Nios II system, could speed up the design flow and improve system compatibility.

The advanced debugging tools embedded in the Altera system impressed us a lot as well for it give us a big help in system design. Compared with the general logic analyzer instrument, the SignalTap® II embedded logic analyzer is more flexible and stable. The key difference was in that we could observe the floating situation of internal signal in the FPGA, and debug the logic compiled by ourselves in an intuitive mode. Besides, we often found a certain logic module worked normally in a single synthesis but failure occurs when it was synthesized with other logic modules. LogicLock™ technology solves this problem well. The powerful design and debug tool made the designing easier and more convenient. For us, the biggest improvement in the Nios II processor is the introduction of the integrated development environment (IDE) which provides a unified visualized interface for program, debug, and download operation. Thanks to this IDE, there was no need to memorize the complicated compile, link, and download instructions.

It is a big challenge for us to write the image-processing algorithm operated on the PC as a hardware module and integrate it into the Nios II system. Although the Real-time Drowsiness Tracking System for Drivers in this text is only a rudimentary application that needs further enhancements, it is a basic design that inspects driver's alertness in driving. In addition, the excellent upgradeability of the Nios II processor will make it easy for us to improve the algorithm and optimize the system in the future.

Third Prize

High Aberrance AES System Using a Reconstructable Function Core Generator

Institution: **I-Shou University, Department of Computer Science and Information Engineering**

Participants: **Chen JianHong, Liu Yu, and Chia-Hau Shiu**

Instructor: **Ming-Haw Jing**

Design Introduction

Cryptography is an essential part of communication or information security. The Advanced Encryption System (AES) was launched as a symmetrical cryptography standard algorithm by the National Institute of Standard and Technology (NIST) in October 2000. Rijndael provides the AES algorithm's architecture, and the algorithm's operation modules are based on finite field mathematics. Coding contains the SubBytes, ShiftRows, MixColumns, AddRoundKey, KeyExpansion modules, and the corresponding transcoding module. We use a look-up table (LUT) for the SubBytes and KeyExpansion modules. This LUT, which is referred to as an S-box, takes up 256[x]8-bit memory.

In addition to containing the original AES specification, a flexible architecture is needed to produce additional inputs that can change to irreducible polynomials, Affine transform matrix, and round number parameters. This algorithm design makes AES decryption impossible even with the golden key, and its variability can be expected to increase by more than 10 million times. This design needs the software and hardware to cooperate, and takes advantage of the FPGA architecture to realize a highly variable AES quickly.

Using the SOPC Builder tool we were quickly able to set up parameters to generate the Nios® II control modules required for development. The Nios II microprocessor uses a RISC core, and can be combined with a variety of peripherals, custom instructions, and custom hardware accelerators, including algorithm logic operation, bit (group) operation, data transfer, flow control, condition instruction, and so on. You can program these hardware accelerators as function calls in the C or C++ languages. Our system adds fundamental components based on finite field mathematics and implements a high-speed

calculator and functional modules in software. We also performed special functions with custom instructions and used the GNU C/C++ compiler and Eclipse IDE.

Design Concept

We used hardware-software co-design to complete the test platform for the AES software and hardware data. Using the test platform, we were able to properly assess AES hardware and the control program module operation as shown in Figure 1.

Figure 1. Co-Design of Software & Hardware
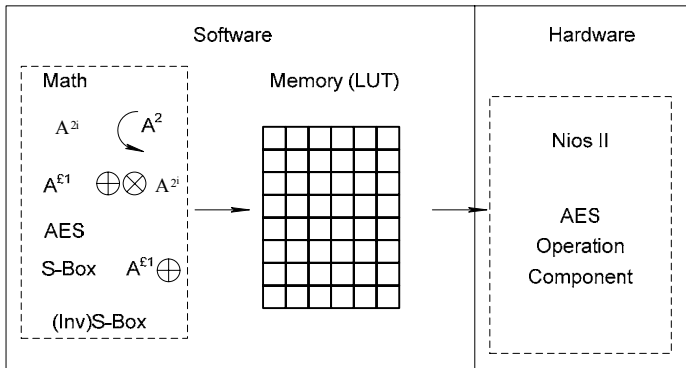
Today, many designers use a fixed irreducible polynomial for higher efficiency and a smaller footprint of the AES intellectual property (IP). For long-term use, however, the fixed irreducible polynomial has been proven to make the system's golden key obvious, thus increasing the decryption rate of confidential files. The decryption methods include side channel, time channel, and power side channel attacks. Some systems can even be decrypted by an inside job. To overcome these deficiencies, we designed an AES with high variability that can generate a LUT in real time through parameter input to provide a dynamic AES core. See Figure 2. The input variables of this system are different from that of the traditional AES, which cannot decrypt the encrypted document. See Figure 3.

Figure 2. AES with High Variability
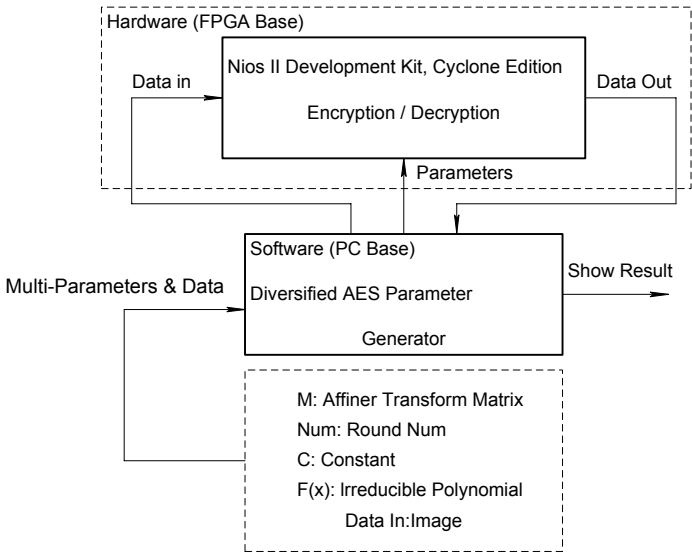
Figure 3. Traditional AES Cannot Decrypt the Encrypted Document

(1) Irreducible Polynomial
(2) Affine Transform Matrix
(3) Round Number
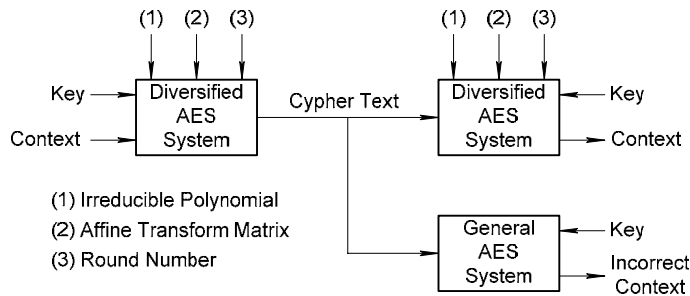
Based on Rijindael's AES theory, we divided the functions into encoding and transcoding. The operation module for both parts is shown in Table 1 (each module is described in later sections). The generation of the S-Box form (see "Implementation Method" for more information) is the key to using AES theory. However, this generation must largely use finite field mathematical operations, such as multipliers and squarers. These operations can be realized in software, so we can generate the required S-Box and (Inv) S-Box. See Figure 4. This group integrates the operation modules of the AES encoding/transcoding functions and requires the inclusion of four main components: (Inv)SubBytes, (Inv)ShiftRows, (Inv)MixColumns and (Inv)AddRoundKey. You can implement the functions using the Nios II software or by using hardware to accelerate the complete flow of encoding/transcoding. For instance, the (Inv)ShiftRows and (Inv)MixColumns components are created in hardware.

Table 1. Encoding/Transcoding Algorithm in Rijndael's AES Theory

Encryption	Decryption	Our Implementation
AddRoundKey for Round=1 to N-1 *SubBytes* **ShiftRows** **MixColumns** *AddRoundKey* end for *SubBytes* **ShiftRows** *AddRoundKey*	*InvAddRoundKey* for Round=1 to N-1 **InvShiftRows** *InvSubBytes* *InvAddRoundKey* **InvMixColumns** end for **InvShiftRows** *InvShbBytes* *InvAddRoundKey*	Bold : Software or Hardware Italic : LUT

The architecture of the AES operation core can be divided into three types (see Figure 4):

■ *Hardware component*—Operation efficiency for accelerating the AES.

   Operation component of the AES theory: (Inv)ShiftRows, (Inv)MixColumns.

   Selection of the demultiplexer for encryption/decryption.

■ *Operation of the dynamic table*—Generation of (Inv)S-Box and (Inv)Key Expansion.

■ *Software operation*—Establishment of the dynamic table, system combination, core component control and operation control, data flow control, and interface control.

Figure 4. Architecture of the AES Operation Core
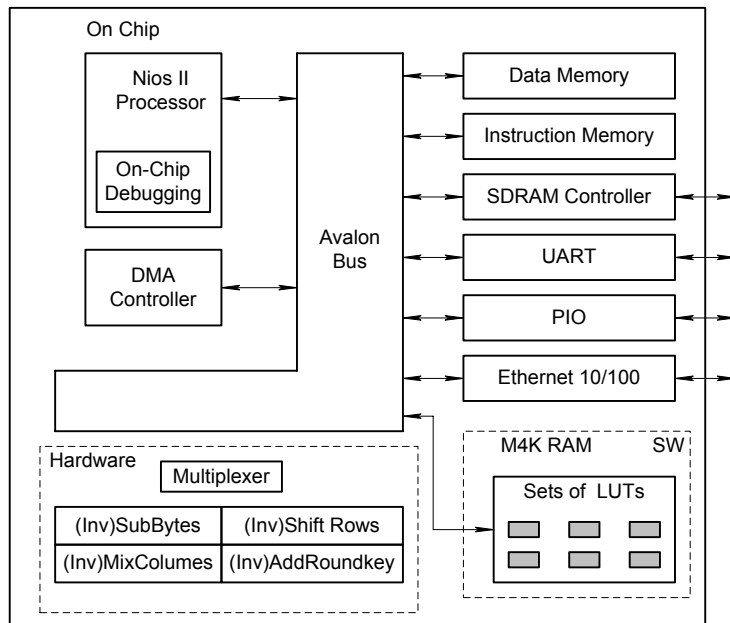
Diversified AES Application Scope

The application scope includes:

■ Secure wireless communications.

■ Protect network routers.

■ Secure electronic financial transactions.

■ Secure video surveillance systems.

■ Encrypted data storage.

■ Secure network storage systems.

Target Users

The target users include:

■ Manufacturers of wireless network bridges and wireless network adapters that support the AES security mechanism.

■ Manufacturers of encrypted VPN products or firewalls.

■ Manufacturers of encrypting chips for mobile phones.

■ Manufacturers of private network hardware or high-capacity hardware array.

■   Manufacturers of ATM secure-exchange devices.

■   Manufacturers of portable communications or storage systems.

■   Manufacturers of private sensor network devices.

Nios II Development Kit

We used the Nios II Development Kit, Cyclone™ Edition, which contains the Cyclone EP1C20FC400 FPGA, to implement our design. The board features 36-Kbyte RAM, 1-Mbyte SRAM, 16-Mbyte SDRAM, 8-Mbyte flash, 10/100 Ethernet PHY/MAC, two serial ports (RS-232 DB9 port), and so on. See Figure 5.

Figure 5. Nios II Development Kit, Cyclone Edition

Function Description

This section describes the functionality of the system.

Expected Functionality

To implement this design we:

1. Used the Quartus® II software version 5.0 to implement the various APUs in VHDL for a high-variability AES system.

2. Designed LUT generator and co-processors.

3. Built the entire AES system using Altera's system-on-a-programmable-chip (SOPC) design methodology.

4. Completed real-time transmission of plain text and cryptograph using a 115.2 Kbps UART interface.

5. Completed 128-bit AES encoding/trancoding with SOPC Builder's C++ compiler.

6. Supported a multi-variable input interface to generate different AES encoding/transcoding processes.

Implementation Method

We used the following implementation method:

1. Completed various APUs designed by VHDL for a high variability AES system.

    a) According to the AES theory, three input methods can generate a high-variability AES system: the irreducible polynomial, the Affine transform matrix, and round numbers.

    b) The APUs were coordinated according to the input requirements of the multiplier, squarer, S-Box, KeyExpansion, (Inv)SubBytes, (Inv)ShiftRows, (Inv)MixColumns, and (Inv)AddRoundKey.

    c) Compiled VHDL code in the Quartus II software version 5.0, and completed functional validation.

    d) According to the specification of Federal Information Processing Standard Publication 197, completed simulation of the software with BCB version 6.0, and validated it.

2. Designed the LUT generator and co-processor.

    a) Analyzed the operation structure of SubBytes and InvSubBytes according to input parameters, and generated the key required by S-Box and (Inv)S-Box form in the software.

    b) Downloaded and stored the generated S-Box, (Inv)S-Box, and Key to the development board.

3. Built the system using the Altera® SOPC Builder tool.

    a) Initiated data sampling using the Cyclone FPGA standard functions.

    b) Added to the user's customized PIO. See Figure 6. The setting of each PIO is shown in Table 2.

Figure 6. Add User's Customized PIO

Table 2. Customized PIO Specification

Name	Size	Direction	Purpose
Aes_data0~15	8 bits	Bidirectional	Transmit encrypted data
aes_ctl_out	32 bits	Export	Control external AES components

4.   Defined UART baud rate: set as 115.2 Kbps, no parity, data bit=8, stop bit=1; as shown in Figure 7.

Figure 7. Communication Setting of UART Component
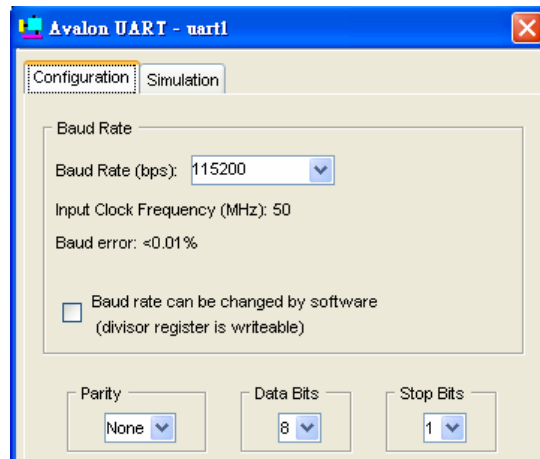
5. Completed 128-bit AES encoding/transcoding process with the SOPC Builder C++ compiler.

    a) Compiled GUI interface program with the SOPC Builder C++ compiler.

Figure 8. Provide User's Input Parameter Interface

    b) Based on the Federal Information Processing Standard Publication 197 specification, we completed the software test platform, which validated the whole system, as shown in Figure 9.

Figure 9. Comparison of Specification (Up) & Test Platform (Down)

    c) According to the integration of the test data, testing system, and the test pattern provided by the specification of Federal Information Processing Standard Publication 197, in Figure 10, the numbers marked in red are the result of the encoding/transcoding process.

Figure 10. Encoding/Transcoding Results by Test Pattern Input

6. Supported a multi-variable input interface to generate different AES encoding/transcoding processes, as shown in Figure 11.

Figure 11. Multi-Variable Input Interface
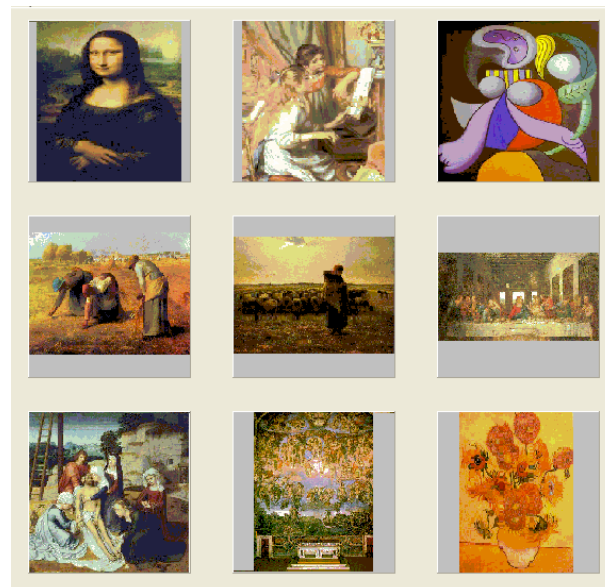
Performance Parameters

The key function of the system while operating on this group is to perform the graphics for encoding/transcoding. Because AES is a symmetric-password system, the method for encoding and transcoding exception sequences is almost the same and the analysis of the following performance parameters lists only the encoding process. The graphics encoding in this group is a 256[×]256-pixel, 8-bit bitmap. Because the system can process 128-bit data each time, it needs 256[×]256[×]8 ÷ 128 = 4096 times encoding in all.

Four encoding functions—including SubBytes, ShiftRows, MixColumns, and AddRoundKey—are used during every encoding process, with each function needing a great number of memory read/write actions (in this group, data memory is set as external SDRAM). The performance analysis is shown in Table 3.

Table 3. Performance Analysis During Encoding

Function Description	Memory		Expected Number of Cycle (Times)
	Read (Times)	Write (Times)	
Load this encoding data from memory (128 bit)	16	16	400
SubBytes	48	16	3200
ShiftRows	16	16	5600
MixColumns	16	16	5600
AddRoundKey	32	16	2400
Write Encoded Data to Memory (128 bit)	16	16	400
Flow and Peripheral Control	0	0	2000

Experimenting with minimum and maximum round, the minimum round is 3 and the maximum round is 11.The numbers shown in bold in Table 3 must be operated repeatedly in accordance with different rounds, and the time of the repeated operation is Round-1. We have arranged the expected time and the actual (experienced) time in Table 4, and added the completed PC software simulation time for comparison. It is obvious that the Nios II/s, the standard processor, outperforms the PC software port.

Table 4. Time Analysis

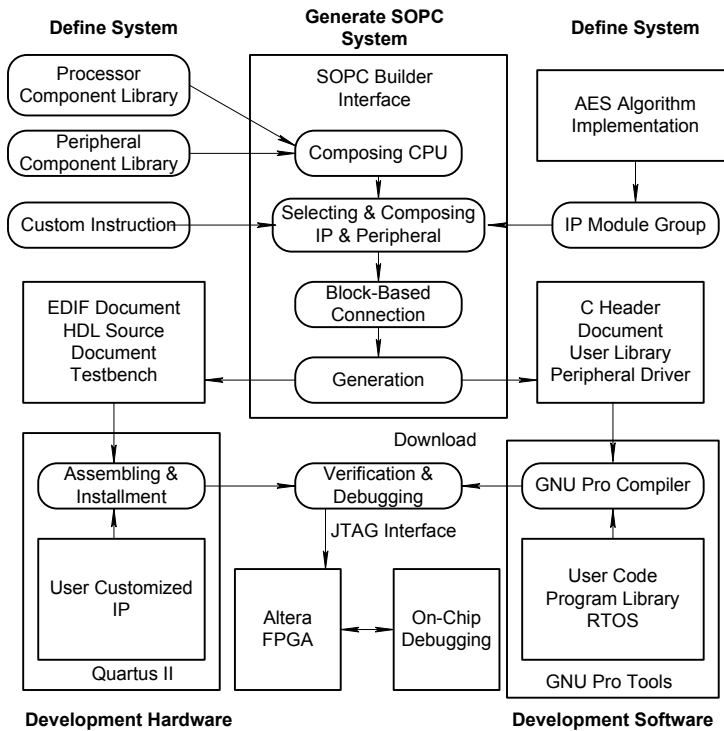| Round Number	Expected Time (s)	Use Time (s)	PC Software Simulation (s)
3	[400+(3200+5600+5600+2400)×2+400+2000]×20ns × 4096 2.98	3	4
11	[400+(3200+5600+5600+2400)×10+400+2000]×20ns × 4096 13.99	13	27

Design Architecture

This section describes the design architecture.

System Design

Figure 12 shows the system design diagram.

Figure 12. Diagram of System Design

We created the user link library of the system. See Table 5 for each program.

Table 5. System Program Description

Function Name	Function Description
Load Parameters	Load Parameters from PC Port
Receive Image From PC	Load Pictures from PC Port
Encryption	Encrypting
Decryption	Decrypting
Send Cipher to PC	Return Completely Encoding Buffer to PC
Send PlainText to PC	Return Completely Decoding Buffer to PC
Text Device	Test Device
Change Mode	Switch Automatic Mode and Debug mode
Print Source Buffer	Return Gradually Source Buffer to PC
Print Cipher Buffer	Return Gradually Encoding Buffer to PC
Print Plaintext Buffer	Return Gradually Decoding Buffer to PC

Figure 13 shows the system diagram.

Figure 13. System Diagram
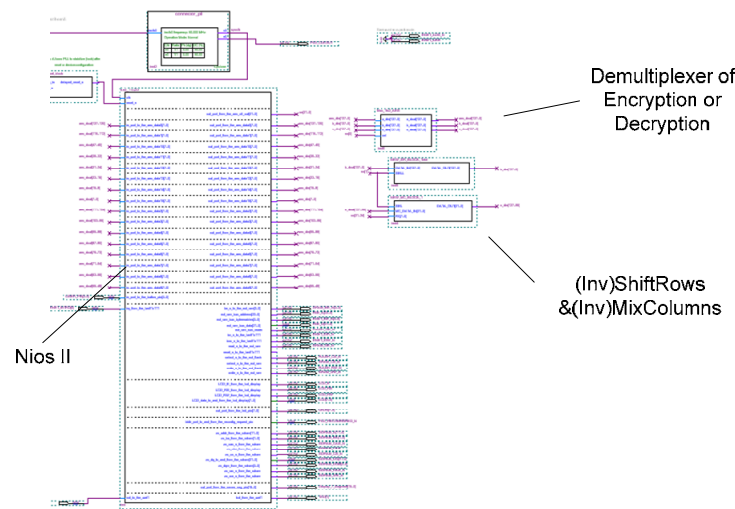
Hardware Design

This section describes the hardware design. We first created the multiplier, including the product and modulation. See Figure 14.
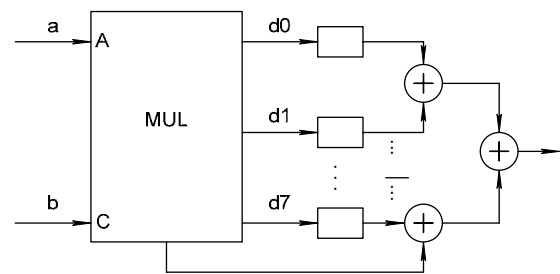
Figure 14. Multiplier Design Diagram

For the product: Computing the result (c with 15 bits) of two 8-bit operations that are multiplied by b. See Figure 15 for the theory, and Figure 16 for the waveform diagram.

Figure 15. Product Design Module
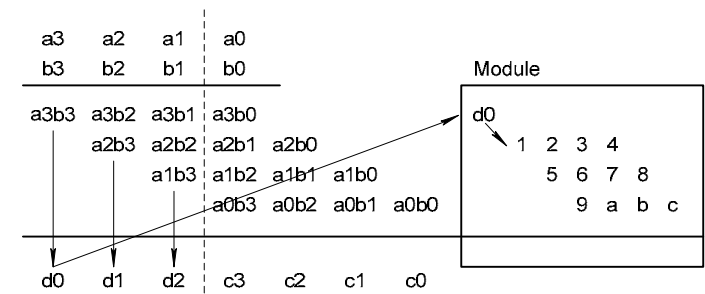
Figure 16. Product Simulation Waveform Diagram
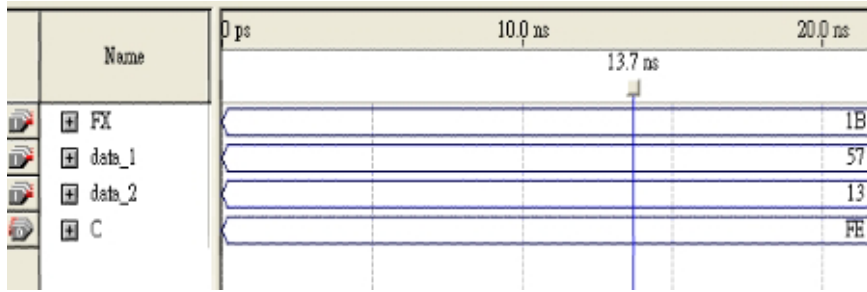
For modulation: Input into C (15 bits), rank-reduced according to different f (x) (where f (x) must be an irreducible polynomial), the result is M (8 bits). The hardware circuit is shown in Figure 17, where f_7 and f_6 denote the top bit and secondary top bit of f (x). These values are input at one port of this layer's AND gate.

Figure 17. Modulation Hardware Circuit Diagram
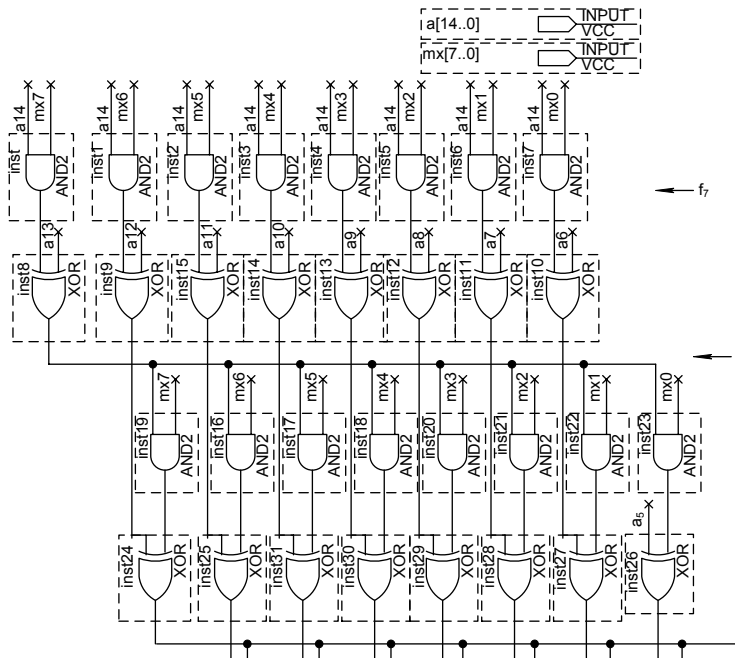
For (Inv)ShiftRow: Combine ShiftRow and (Inv)ShiftRow components and send required parts by multiplexer. Sel=0 is required one for encoding and sel=1 is required for decoding. See Figures 18 and 19.

Figure 18. (Inv)ShiftRow Design Diagram
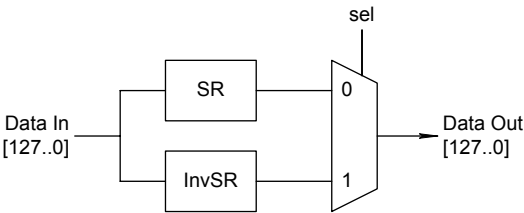
Figure 19. (Inv)ShiftRow Simulation
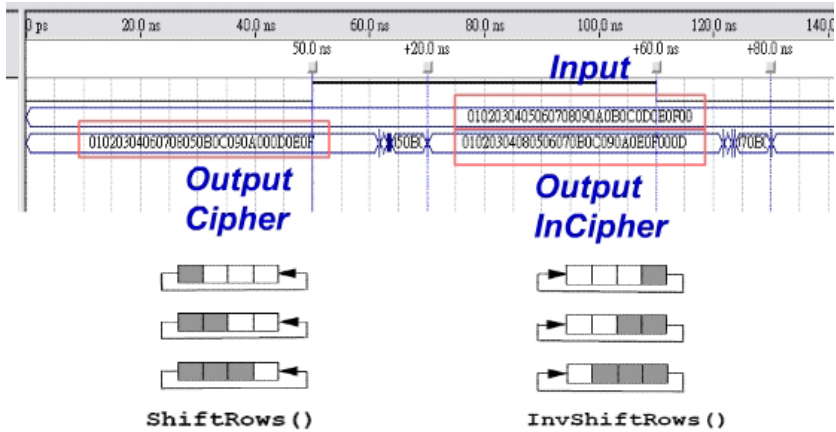
For (Inv)MixColumns: combine and design MixColumns and InvMixColumns components, generate different Word_MixCs in accordance with input different MixColumns Polynomials; the Word_MixC exports encoding/transcoding data at the same time it is encoding for sel = 0, transcoding for sel=1. See Figures 20 and 21.

Figure 20. (Inv)MixColumns Design Diagram
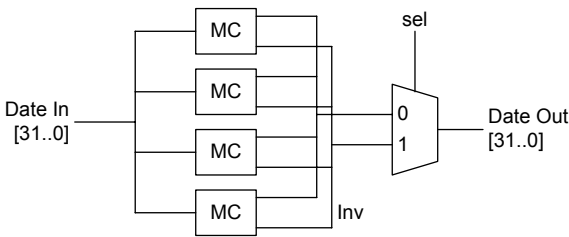
Figure 21. (Inv)MixColumn Simulation
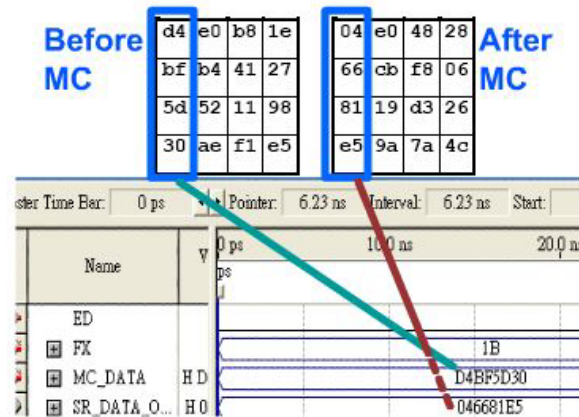
Software Design Flow

The function flow is as follows:

1. *Automatic software simulation*—Create an S-Box and complete encryption/decryption simulation and image encryption/decryption process. See Figure 22.

Figure 22. Automatic Software Simulation Flow
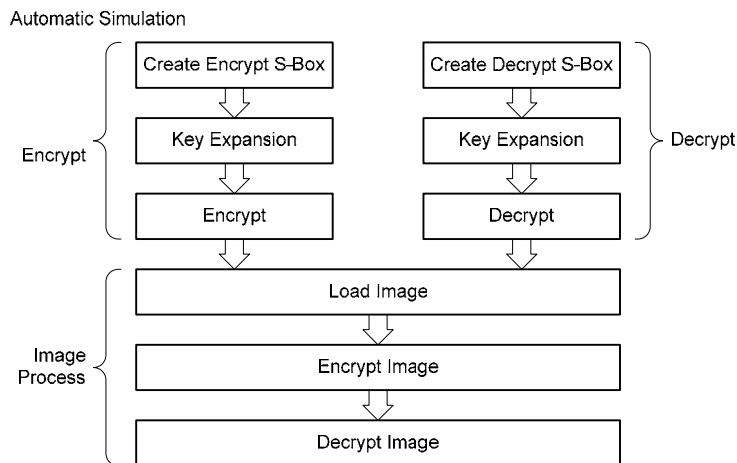
2. *Hardware encryption/decryption*—The RS-232 interface is used to pass parameters and data and to receive the data after verification. See Figure 23.

Figure 23. Hardware Encryption/Decryption Flow

Hardware Encrypt & Decrypt
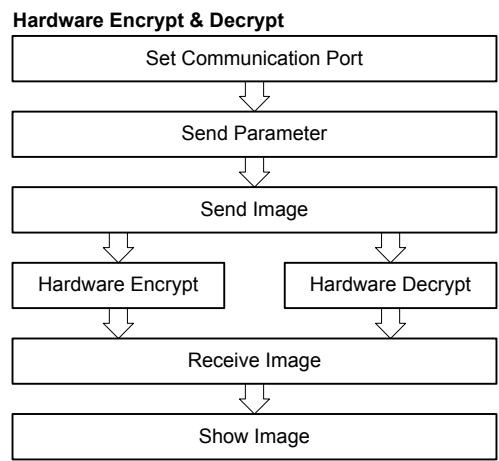
See Table 6 for the description of the software pseudo-code and function.

Table 6. Software Pseudo-Code

DAES Encryption	DAES Decryption
Load_Parameter	**Load Parameter**
Generate_Encrypt_SBox	**Generate_Decrypt_SBox**
Key_Schedule	**Key_Schedule**
for Image_rowcount=1 to row	for Image_rowcount=1 to row
for Image_colcount=1 to column	for Image_colcount=1 to column
AddRoundKey	**InvAddRoundKey**
for round=1 to N-1	for round=1 to N-1
SubBytes	**InvShiftRows**
ShiftRows	**InvSubBytes**
MixColumns	**InvAddRoundKey**
AddRoundKey	**InvMixColumns**
end for	end for
SubBytes	**InvShiftRows**
ShiftRows	**InvShbBytes**
AddRoundKey	**InvAddRoundKey**
end for	end for
end for	end for

See Table 7 for the software function description.

Table 7. Software Functions Description

Coordinate All Required Functions	
Basic function:	
Multiplication	Multiplication that operates on finite field
Inverse	Inverse element that operates on finite field
Matrix Inverse	Inverse Matrix that computes Affine Transform Matrix
Image Process	Sub-program of image process
Debug RS232 communication	Sub-program sent by UART interface
AES function:	
Load Parameter	Load parameter, user can change source parameter
Generate_Encrypt_SBox	Generate_Encrypt_SBox: Required S-Box for generation of encryption
Key Schedule	Key is expanded for the use of AddRoundkey
Create Encrypt S-Box	Table that creates S-Box
Create Decrypt S-Box	Table that creates (Inv)S-Box
Key Expansion	Key that creates each Round
Encrypt	Encrypted sub-program
Decrypt	Decrypted sub-program

The key functions are described as follows:

1. *Parameter input*—The user can input parameters and increase variability function. See Figure 24.

Figure 24. Parameter Input Interface

2. *Data verification*—Provide complete encryption/decryption process. See Figure 25.

Figure 25. Data Verification Interface
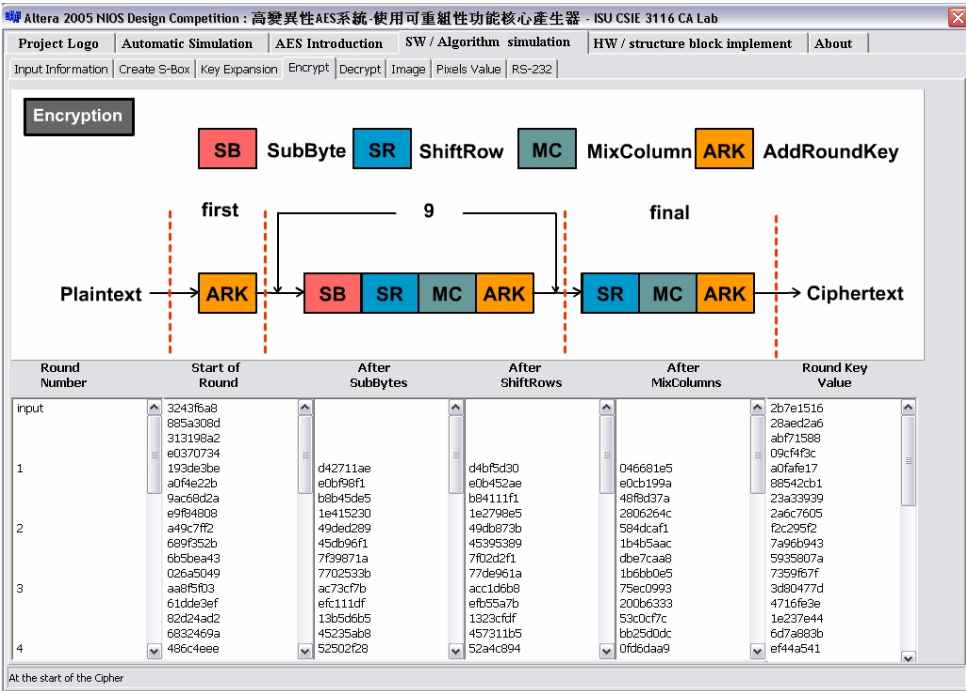
3. *Image process*—Load a 256[x]256, 8-bit image and perform encryption/decryption process. See Figure 26.

Figure 26. Image Process Interface
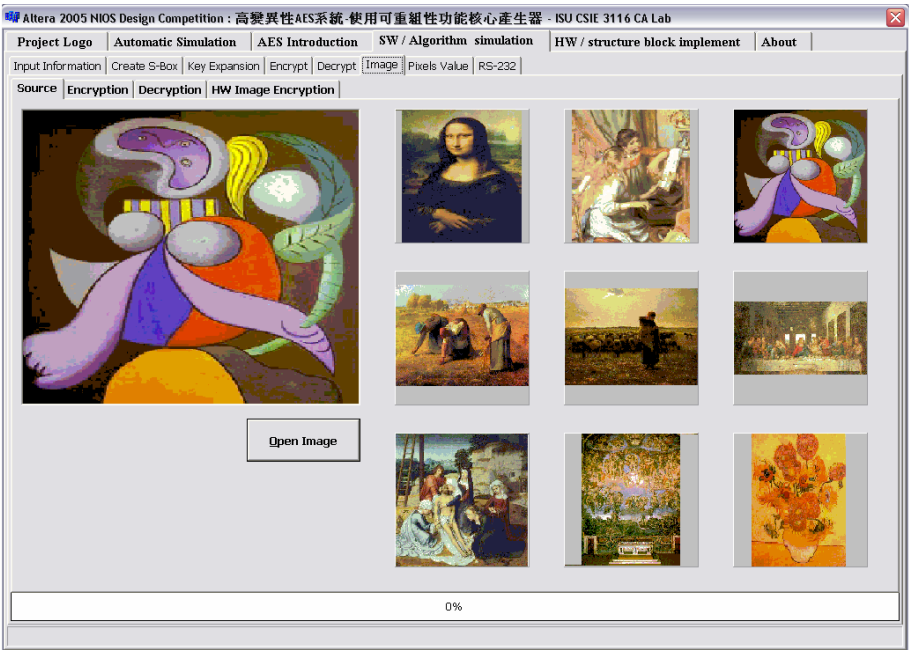
4. *Debug communication*—Based on RS-232 communication handle, the hardware returns the data to make debugging easy. See Figure 27.

Figure 27. Communication Debug Interface
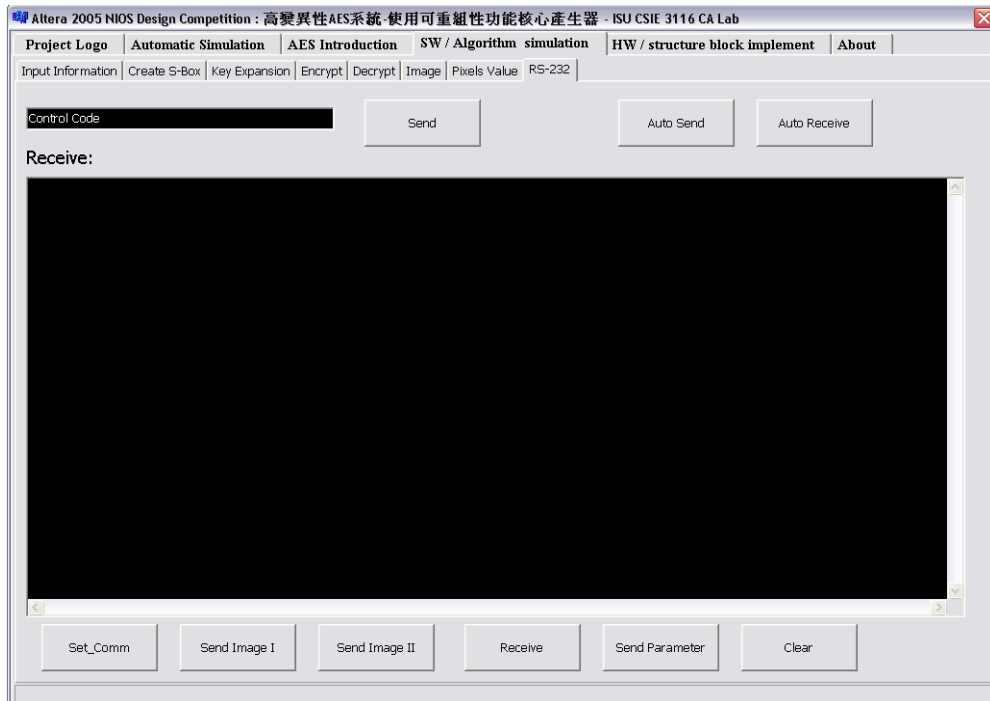
Design Methodology

This section describes the design methodology.

Realization Method

The realization method includes the following steps:

1. *Definition of the AES system*—Including the processor, memory, and peripheral components.

2. *Generation of the system*—Generate the document using the SOPC Builder tool.

3. *Design the hardware*—Build the required components with VHDL code, and incorporate, compile, and simulate the circuits.

4. *Design the software*—Use the Nios II integrated development environment (IDE) to generate the related headers and drivers, write the application program, and compile it as an .elf executable file.

5. *Simulation*—Simulate with the ModelSim software tool. If there is a problem, return to step 2 to modify the system design software/hardware.

6. *Verification*—Perform verification by downloading the software/hardware with the JTAG port onto the RAM of the Cyclone development board.

7. *Test*—Produce test results by combining the user interface software of the PC port development board and by delivering huge volumes of data for measurement.

Design Process

The SOPC design approach provides an integrated software/hardware with an IDE including logic part (IP design), storage part (RAM), and computation core (CPU or DSP). The process of our system design is as follows:

■ *Selection of algorithm and core*—We were able to handle the entire AES operation process with the Nios II processor by adopting Rijndael's AES algorithm. We added input variables to generate the dynamic form that is stored on the on-chip RAM.

■ *Selection of the IP and design of custom IP components*—Our design uses general-purpose IP components, whose detailed explanation files can be downloaded from Altera's website. Additionally, we developed custom IP components according to the requirements of the system design and connected it onto Avalon® bus.

■ *Design of the software/hardware system*—Software and hardware modules are used together in this design. This method creates challenges because the development of the software involves the plan and assignment of hardware resources, and is dependant upon system performance. However, the SOPC Builder and the Nios II IDE tools provide us with an integrated software/hardware design development system, making it easy to accelerate the design process.

Design Features

■ *Dynamic Form Generation*—Based on the three input variables, the dynamic form of the S-Box and MixColumns Matriqs are generated and stored in RAM. Thanks to the FPGA architecture, we can use the Nios II processor to control the operation component of each AES, and perform data move, access, and operation with the Avalon bus. In this way, we have successfully implemented the high variability AES password system on the FPGA.

■ *100% Realization of Software/Hardware*—The software/hardware platform of the Diversity AES project was successfully designed in this group, greatly improving the security of the AES.

■ *Personalized Demo Program*—In this group, the whole Diversity AES process is shown at the software port by AutoRun, which enables the user to understand quickly the design operation.

■ *Connecting Three Customized IP Functions to the Nios II Core*—Because the Nios II processor is flexible, we were able to design the PIO of external communication according to the design requirements. This means we were able to combine (Inv)ShiftRow, (Inv)MixColumn, and demultiplexer to accelerate the efficiency of the AES encoding/decoding operations in this group.

■ *Solution to UART-Related Envelope-Packet Transmissions*—Because the UART IC on the development board connects with a 25-MHz quartz oscillator, frequency errors may occur. As a result, a few envelope packets may not be delivered during a large volume data transmission. Therefore, we reduced the envelope packets in this group, making them suitable for transmission and successfully solving the problem.

Conclusion

During Altera's 2005 Nios II processor competition, our design group divided the design tasks into system integration, hardware development, and software design as follows.

■ *System*—The convenience of the Nios II IDE and the SOPC Builder tools gave us the flexibility to realize the design quickly on a prototype machine, which accelerated the development process. Through this competition, we have learned the process of consumer-electronics product development. The SOPC design approach reduces the cost of manpower and material resources during development. Because of this, we believe that the design approach will become popular in the future. While we did not add many components, this competition made us appreciate the potential of more system integration capabilities. Additionally, we hope that Altera can provide a variety of demo board demonstration programs that will enable interested students to quickly grasp the development process of FPGA designs.

■ *Hardware*—In this competition, we used a top-down design approach and planned the complete design of the hardware in the beginning. This meant that a set of data stream rules needed to be established at the start of the planning stages. These rules eliminated problems during the design stage, allowing the project to be completed on time. Teamwork became an integral part of this contest. Although the Quartus II tool was easy and flexible to use, there were design issues that required experience; for example, using different frequencies while accessing the RAM. In conclusion, this competition provided us with an important opportunity to learn about teamwork and problem-solving, understand system development, and resolve challenging design questions.

■ *Software*—We developed the necessary software interface, stressing communication and message exchange with the Nios II processor. We completed this task with the RS-232 interface, and learned a lot about message transmission. We adopted the SOPC Builder C++ tool to create the software design for the Window's interface. We used it as a verification tool and managed to perform Nios II communication debugging for the phase test. We hope to learn more about SOPC design in the future!

Third Prize

Wireless Multifunction Digital Storage Center

Institution:	**Beijing University of Industry**
Participants:	**Chen Zhuo, Dai Nan, and Fang Dongyu**
Instructor:	**Xu Xiangdong**

Design Introduction

With the ever increasing use of data-transmission networks, data is increasing in geometric progression, and the traditional storage network architecture cannot cope. To deal with the massive amounts of data, data storage architecture is evolving to centralize data and features expandability, adaptability and reliability. This evolution now serves large enterprises and system management enterprises (SMEs). Interestingly, studies now show that many customers set aside a significant amount of their IT budget to handle storage. A wireless multifunctional digital storage center can efficiently provide mass data storage.

A wireless multifunctional digital storage center has centralized data and features a separate OS for data management and network performance enhancement. This cost-effective design meets the requirements of SME data transmission and storage. The system provides wireless network access applicable to wireless office environment and enables network access through an RJ-45 network interface. This setup makes it convenient for users accessing the WAN and LAN ports to set task allocation for the device. The shared document database on user management ensures the safety of stored data; the operation and management tasks controlled by web interface, LCD, and keypad make it easy to use the device. For experienced users who are familiar with Linux commands, the device provides a Java command-line mode for remote control.

Our design is based on an Altera® FPGA and the Nios® II processor, as well as Altera development tools. We applied system-on-a-programmable-chip (SOPC) design principles and used the Microtronix tool to migrate the uCLinux OS and integrated wireless data transmission storage, user management, FTP server, web server, Windows document share directory service, remote upload/download, and remote control functions. The uCLinux OS handles dispatching and management tasks of the system and ensures expandability of the product design. For enhanced performance, users can upload software to update the device in the software layer.

210

Our design is based on embedded SOPC technology that integrates Ethernet, compact flash card, SRAM, and PIO interfaces with the system to facilitate the functional modules integration with the whole system. The design uses a TCP/IP connection to connect to the Internet directly to perform data storage/transmission and system control based on the user's requirement, simplifies data upload/download, and facilitates document management. See Figure 1.

Figure 1. System Flow Chart
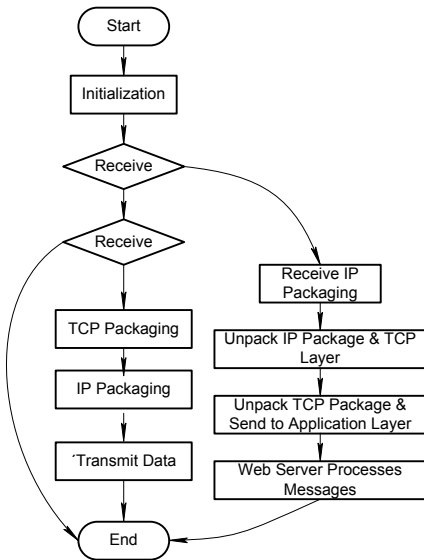
Function Description

The system contains the following parts:

■ *Multi-User Storage Management System*—The operation rights of managers and common users are kept separate, and are validated with passwords. This mechanism has enhanced device data security and improved resource utilization.

■ *Management & Control Base on Web Interface*—To establish a web service function, the uCLinux OS can read the device state information any time. At the same time, users can control the storage center with a web browser using the common gateway interface (CGI) protocol. Common users can only access functions such as examining system status and remote download, whereas managers can perform additional functions, such as user addition/deletion, open/close of Samba and FTP services, FTP and Samba directory setting, restart of system, and disconnection of FTP.

■ *FTP Function*—This function establishes FTP service on uCLinux OS, which is the same principle as a Web server function.

■ *Windows Shared Directory*—This function helps the user to migrate Samba software onto the uCLinuxoperating system and realizes file sharing using the SMB communication protocol. It also helps the user view the shared directory of the product on the network.

■ *Local LCD and Keypad Control*—By interfacing the LCD and keyboard, users can read and control the local device status using the PIO signal lines of development panel. Our system can automatically receive the DHCP IP address distribution, and the IP address can be set manually by

the user if no DHCP server is configured. You can also read the IP address that has established the FTP connection in real time, as well as the download task name and status.

■ *Wireless Network Access*—The product can be used in a wireless office environment with a access point (AP) supplied by Huawei Corporation.

■ *Java Telnet Application (JTA)*—This application should not be confused with the Java Transaction application programming interface (API). This application is a client tool that integrates telnet and ssh, and gives advanced users a command-line mode for the device based on the Linux OS. Furthermore, mobile devices that support Java can carry out access and control, thereby improving the expandability of the device.

Performance Parameters

CPU: Nios II/f Core
JTAG DEBUG LEVEL: LEVEL 1
Small LCD refresh time: 0.1 μs
Small LCD response time: 1000 ns
LCD Size: 128[x]64 lattice
Time interval for reading information on LCD system: 5 seconds (on a time-sharing basis)
Average memory utilization: 75%
Capacity of CF card: 16 MByte (extendable)
Size of uCLinuxoperating system: 2012 KByte (kernel)
File system size: 3,612 KByte
Nios II kernel frequency: 50 MHz
Nios II kernel Mps: 23 MIPS (Dhrystones 1.1)
Number of consumed logic unit: 6,409 / 10,570 (60 %)
Pin utilization: 229 / 427 (53 %)

Design Architecture

Figures 2 and 3 show the design architecture.

Figure 2. Design Architecture
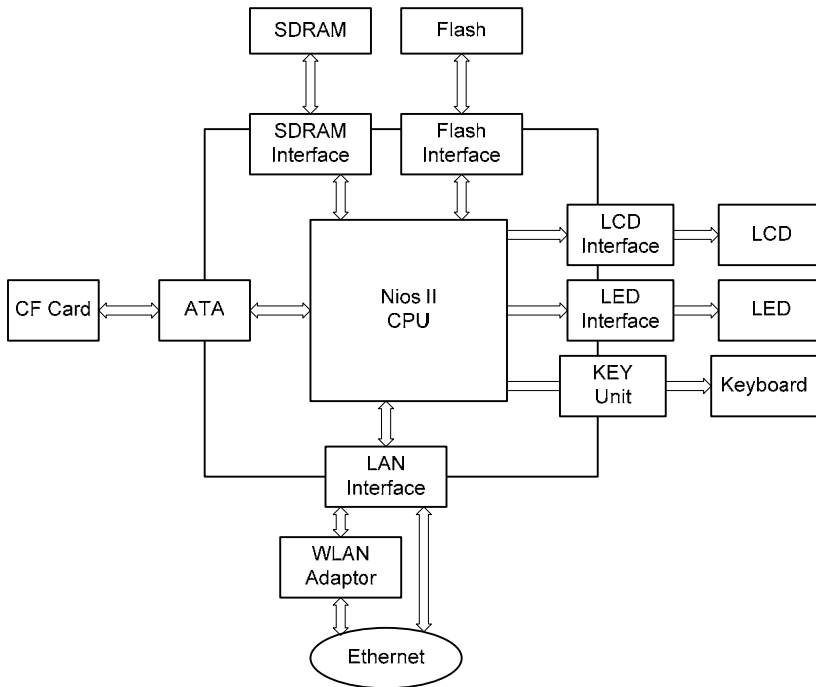
Figure 3. System Operation
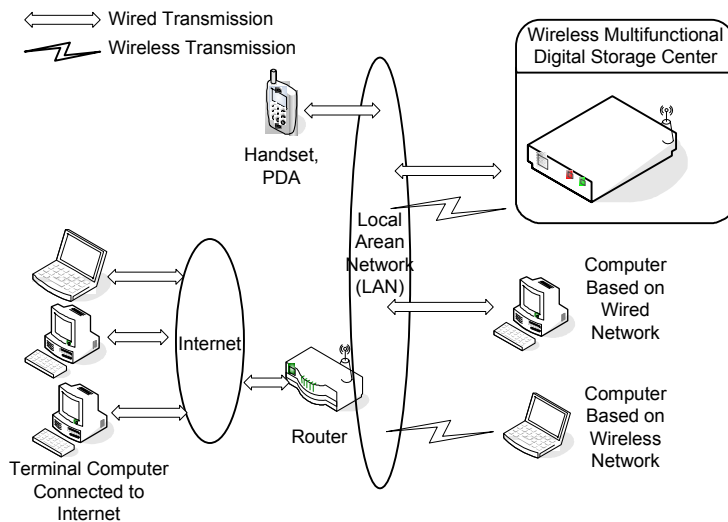
Design Methodology

Altera's SOPC Builder improves the working efficiency of the design. Using SOPC Builder, we easily integrated the flash memory, Ethernet, compact flash card, and SDRAM modules together with the IP kernel provided by Altera. In doing so, we avoided having to design peripheral circuitry, which greatly reduced the design time and cost.

Our product design is based on the uCLinuxoperating system that handles Samba (network sharing software of Linux and NT system), ftpd (software supporting FTP service), httpd (the simplest web server software), Boa (web sever software that can support authentication and CGI), uncgi (analysis software of webpage form), and wget (download software based on HTTP and FTP protocols running on Linux's operating system).

We created the user management and system setting functions using CGI forms, shell scripts, standard expressions, and the file system.

We can control the LCD by a set of keyboard commands. Users can read the main menus of the data analysis module and all subfunction menus using up/down keys. The keypad and LCD provide the interaction between the user and designed module. See Figure 4.

Figure 4. LCD & Digital Keyboard
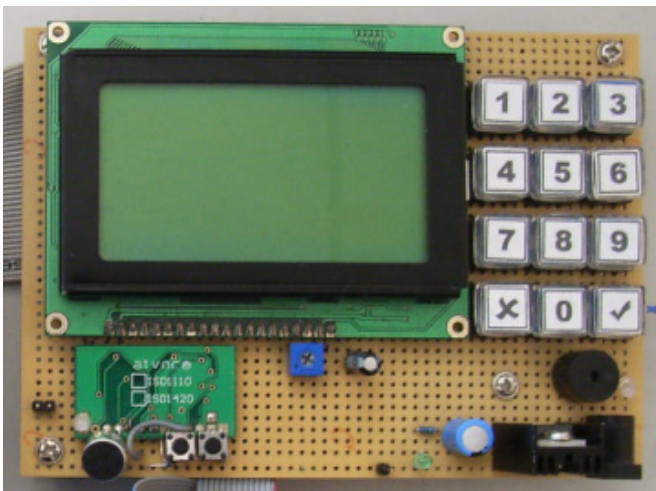
The LCD control program and the keypad was compiled with the Nios II integrated development environment (IDE) and uCLinuxdevelopment plug-in provided by Microtroni. We used the multithreading technique in the uCLinuxoperating system to implement keystroke processing operations, display screen refresh, and read different system state operations under different rates. In doing so, we took full advantage of the dispatching function of the operating system and omitted the more complicated information exchange among function modules to save CPU resources.

With a view to enhance functional features of the system at a later date, we have not fully addressed features such as the security of CGI and shell script programs, the remote control of handset WAB, the handling of independent memory space for multi-user operation, as well as personal ID authentication of the IC card. Moreover, concerns on the stability of migrated software will require a validation process that will take a long time to process.

Design Features

With an SOPC design methodology, our design integrates SDRAM, Ethernet, compact flash, and flash control interface modules. This methodology helped us to simplify circuit design and reduce costs. Compared to the available memory devices in the market, our product offers a variety of advantages such as wireless network access, hierarchical user-management mode, reliable security, and multi-control mode.

■ *Wireless Network Access*—Adapting the device for wireless AP mode, users can easily access an office network or wireless network, and conveniently move from one network to another.

■ uCLinux*Operating System*—Because of the open source nature of uCLinuxoperating system, you can easily adopt many software modules into the design. In doing so, you can add the relevant software modules to expand the device functions and update system software.

■ *Easy & Maneuverable Control Mode*—We can easily manage data and users by deploying the web browser interface without the need for storage drives. You can do this even with the PDA and smart phones that can access network to carry out remote setting of the device. The local LCD and keypad can monitor the device in real time, directly manage the system using with control panel on storage center even when the Internet is unavailable.

■ *Hierarchical User Management & Virtualized Storage*—By adopting a hierarchical user-management system, in which the administrator and normal users are provided with separate rights and different control capabilities for the system, you can enhance system security. The process works as follows: users log into the storage center via a password authentication system, which is adopted by different levels of administrators to prevent unauthorized access.

■ *Remote Download Based on FTP & HTTP*—The user can access the LAN or WAN where the device is located, and can download data remotely and store into the device.

■ *JTA (Java Telnet Application)*—Any mobile equipment supporting Java can be used to control the equipment, which makes for easy expansion of storage applications.

Conclusion

In this design competition, we gained considerable knowledge of the Nios II processor and Altera's development tools, and realized the benefits brought by the SOPC design concept and understood the development features of Nios II processor.

The building block system of SOPC design enables flexible custom peripherals to suit different needs; Altera has provided drivers for most of the peripherals, which reduces the difficulty of hardware design, shortens development cycles, increases reliability of design, and helps those unfamiliar with IP core development to integrate system modules. Furthermore, we have access to many resources via the network, which greatly expanded our scope and depth of development, reflected the flexible and comprehensive features of embedded development, and significantly reduced development cost.

With the Nios II embedded processor and uCLinuxoperating system, many Linux resources can be adapted into applications, which eliminates repeat development of application software, thereby simplifying software development and reducing cost.

In the development process, however, we also found some disadvantages of the Nios II processor application and SOPC development. For instance, the development board lacks expandability; we hope

that in the future the development board will allow the user to control more pins, and that it will integrate more interfaces.

Thanks to Altera and Cytech for providing this competition as a learning opportunity for us.

Third Prize

Nios II Soft Core-Based Double-Layer Digital Watermark Technology Implementation System

Institution: **China University of Science and Technology**

Participants: **Lian Jiezhen and Ye Qingfeng**

Instructor: **Liang Xiaowen and Li Yuhu**

Design Introduction

With increasingly powerful functions available from digital imaging software, the reliability and authentication features of digital images are gradually decreasing. Therefore, reliability and authentication are important for a wide range of digital image applications. Digital watermarking provides digital copyright protection, ensures integrity, and assists during digital transmission. The technology guarantees the reliability and authentication of digital images. We can expect to see more usage of digital watermarking in the future.

Today, digital watermarking algorithms are generally realized in software. However, such algorithms have low execution efficiency and suffer from not being robust enough to withstand hacking. Watermarking technology implemented in hardware can overcome these weaknesses. Additionally, hardware-based watermarking offers high reliability, imperviousness to hacking, and high-execution speed. Thus, hardware-based watermarking can be deployed safely in applications that need high reliability and speed. Given today's advances in FPGA technology, we can implement hardware-based watermarking technology quite efficiently. In particular, with its outstanding performance and numerous design advantages, the Altera® Nios® II embedded soft-core processor is the first choice to implement a hardware-based watermarking system.

Application Scope

Our project is designed to monitor devices in public places, aid in forensics, provide traffic-offense monitoring identification of medical images, and important news pictures.

Target Users

Our design targets users with special requirements regarding the credibility of digital images, such as government departments, traffic offense governing agencies, courts, hospitals, and the media.

Why We Chose the Nios II Processor for Our Design

We chose the Nios II processor for the following reasons:

■ Being completely customizable, the Altera Nios II processor has high performance, and supports flexible product development designs at low cost.

■ The Nios II processor's customizable instruction set can accommodate complicated arithmetic operations and can accelerate algorithm processing, which provides faster execution than implementing these operations in software.

■ A configurable design enhances system performance. Implementation of a watermarking algorithm requires a custom instruction set, DCT/IDCT hardware accelerator, and so on. The configurable design approach enabled us to achieve our performance goals in a short time.

■ The MicroC/OS-II and real-time operating system (RTOS) in the Nios II integrated development environment (IDE) are very user friendly.

■ Based on the Nios II processor, we can enhance system performance by adjusting the Avalon® switch fabric. This technology supports multiple parallel data channels to achieve high throughput in watermarking applications.

■ Implementing the processor, peripherals, memory, and I/O interface in a single FPGA can reduce the total system cost.

■ Rapid system implementation. We could go from the original concept design to system realization in a short time because of using the Nios II processor. Additionally, we could easily upgrade the hardware and software on site. The flexibility allows us to design products in line with the latest specifications and equipped with new features.

■ Unmatched levels of flexibility. The Nios II processor features complete customizability and reconfiguration. For example, it supports three processor cores, peripherals, the Avalon switch fabric, custom instructions, and hardware acceleration. All of these functions can be implemented using commonly available Altera FPGAs.

■ Powerful combination of Altera intellectual property (IP) functions and FPGAs. Using IP optimized for the FPGA architecture, we can redesign standard functions easily, rapidly customize hardware peripherals, focus on design partitioning, and improve our design knowledge.

■ Integrated development kits. The Quartus® II software, SOPC Builder, ModelSim®-Altera software, and SignalTap® II embedded logic analyzer provide a complete set of test and debug tools for hardware design. With the Nios II IDE, it is possible to simplify software design and all software development tasks, such as program editing and debugging.

Function Description

While designing the system for this project, we gave consideration to whether the implementation was feasible and practical. The two-layer digital watermarking algorithm features IPR protection, authentication, creation identification, and other functions.

Function 1

This design implements both IPR protection and creation functions, and delivers the following advantages:

■ Adds watermarking to the image without affecting the nature of the original image.

■ Provides unique watermarking that is difficult to counterfeit.

■ Allows you to generate a large number of watermarks.

■ Supports authentication and watermarking identification without the original image.

■ The watermarking can resist the JPEG compression algorithm with a high compression ratio.

■ The watermarking offers strong resistance to ordinary geometrical transforms, such as low-pass filtering, sharpening, and scaling.

■ Several watermark effects can be applied simultaneously, thereby protecting the intellectual property rights (IPRs) of the owner and purchaser.

■ The algorithm is secure from strong attacks, such as collusion attacks and extra watermarking, to authenticate images effectively. Where images have been modified, we can show the part that was modified, which helps to distinguish common operations, such as JPEG compression, from deliberate image modification.

The image processing algorithm should be the first design objective when creating digital image watermarking technology in hardware. We designed a two-layer digital watermarking algorithm for this design contest. In our algorithm, we selected the secure Gaussian random sequence as the watermarking signal on the first layer. Additionally, to make the watermarking more secure, the watermark data is generated by a unique key that generates a unique watermark. The second layer watermark is based on a look-up table algorithm for the authentication of the image's integrity and reliability. Finally, we simulated the algorithm with the MATLAB software to verify whether the algorithm correctly realized the functions required for two-layer watermarking.

Function 2

We used the Nios II development board to implement the watermarking algorithm. A PC performs real-time tasks for proper operation of the algorithm. In practical applications, the detection algorithm cannot be implemented on a development board. Instead, it is more likely to be implemented on a PC. Therefore, we needed to design the host system and a user-friendly graphical user interface that features all control and detection functions. We designed the GUI to be very user friendly and to enable users to intuitively see and operate functions of the two-layer digital watermarking system.

To implement the watermarking algorithm, we needed to create the communication system between the Nios II development board and the host system. Because the USB interface did not work well for us in practice, we used a JTAG UART serial port instead. Communication was based on the host-based file system module in Nios II version 5.0 and operational instructions for files from the C-language library.

Because we needed to simulate the algorithm in early stages in the MATLAB environment, and because the detection algorithm involves complex mathematical computations, we decided to use the MATLAB software to design the host control and detection system as well as the GUI.

In our design, you can display the original image, modified watermarking image, added watermarking, and detected watermarking simultaneously, to facilitate easy comparison on the same display. At the same time, using multiple buttons, users can easily perform operations such as AuthWatDect, OriImgDect, Start, Add, and so on. We designed these buttons to have corresponding functions on the host system. For example, pressing the Add button to add images first invokes the corresponding function on the host system to generate a 128-bit key. Pressing the Start button notifies the Nios II system to start the watermark computation.

The host system mainly detects authentic watermarking and its integrity on a PC. Pressing the AuthWatDect button starts detection of authentic watermarking. The system obtains threshold values, after calculating the relevant peaks, to judge whether a watermark has been added and the type of watermark. Pressing the OriImgDect button starts the integrity detection based on data values from a look-up table. If the image was not modified, the detection image in the look-up table is white. If the image has been modified, the modified part in the detection image is black.

Function 3

A core function of the system is to create the two-layer digital watermarking algorithm with the Nios II soft core processor. The system comprises two functions, which are implemented as follows:

■   The digital cameras supporting external monitors are connected with the right interface to allow high-speed data transfer. This process ensures a reliable, practical watermarking implementation. We used the JTAG UART for data downloading and appropriate switches to ensure communication with mainstream digital cameras and hosts at high speed.

■   Extend the functions of a digital watermarking system with peripheral components. During the insertion of a watermark, the text "watermarking…" is displayed on the LCD. When a watermark is successfully added, the text "it is great" appears, making the system more humanistic.

Performance Parameters

This section describes the system's performance parameters.

Fixed-Point Arithmetic

Because of the real-time requirement of the application, we relied on Nios II fixed-point processing, and implemented the arithmetic to handle fixed-point processing to determine the suitable value arrangement, ensuring computing efficiency and accuracy. In the design's arithmetic calculations, the RGB signals in the original image pixel are set to 8-bit data. To cater to negative value components that might be produced during the YCbCr conversion to color space, we set the Y, Cb, and Cr as 9-bit (-256 to +255) data. Similarly, the two-dimensional DCT conversion results fall in the range of -2048 to 2047, and the quantified value by the quantizer falls within the range of -128 to 127. The LUT design is set to 256 digits.

Optimization of Division

Because the Nios II processor has no special arithmetic instructions, division operations are performed by successive subtraction, which consumes a lot of computing resources. Although we encounter fewer division operations in arithmetic computation, e.g., in normalized operation of partial weight factors, the division operation still has a certain influence on performance. Therefore, we tried to compute these

operations in an optimized way to approximate values through binary shifting and module arithmetic. Because, shift arithmetic can be performed with other operations, we can improve the computing speed while ensuring accuracy. This method also helps conserve system resources.

Using Parallel Processing with the FPGA & Nios II Processor

A lot of multiply-accumulate (MAC) operations are applied during discrete cosine transform (DCT) conversion and during the addition of Gauss serial watermarking with copyright protection. Using the Nios II processor, we can adjust system performance using the Avalon switch fabric, which supports several parallel data channels and helps to implement image watermark processing.

Memory Management

The Nios II processor offers unprecedented flexibility through its customizable and reconfigurable architecture. It features a powerful combination of optimized IP for the FPGA system architecture, which allows us to add SDRAM, SRAM, and flash using the SOPC Builder tool. We use these tools to implement a large scale, real time system with abundant memory resources.

Communication Between the Development Board & Host PC

The JTAG UART interface must only be used for communication because the UP3 board needs to use a special USB-Blaster™ data cable for downloading. Because the UART interface has a slow communication speed, we adopted several measures to avoid a time-penalty for using the Nios II processor for image processing, and to utilize CPU resources more efficiently. For example, the image that needs to be processed is read in eight lines of data each time and sent back to the host PC after processing. Then, we read the next eight lines of data, and repeat the previous steps. The host PC and Nios II processor are linked via the JTAG UART interface, which greatly influences the interactive speed between them, and limits the whole image processing speed.

A new function provided in the Nios II processor version 5.0 also facilitates file output. Selecting the software component in the **syslib** engineering attribute, and then choosing **Add this software component** in the Altera host-based file system, you can read/write host PC files in debug mode.

To implement the watermarking algorithm arithmetic, we integrated the first and second layer watermarking. Figure 1 shows the watermark insertion arithmetic block diagram. Figure 2 explains the watermark extraction arithmetic block diagram. Our performance tests showed that combining two-layer watermarking has no influence on an individual layer's performance. However, this combination allows you to determine the sequence of multiple watermarking without any interference, in case they were added to an image.

Figure 1. Arithmetic Insertion of Double-Layer Digital Watermarking
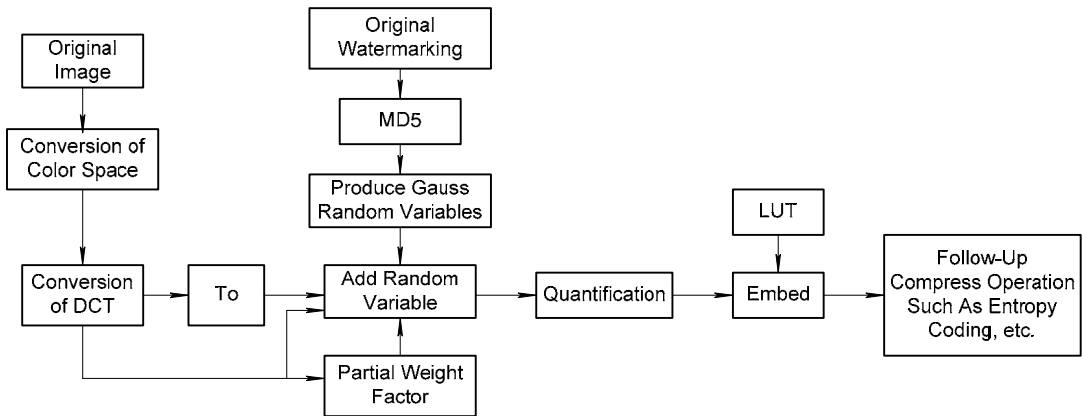
Figure 2. Arithmetic Extraction of Double-Layer Digital Watermarking
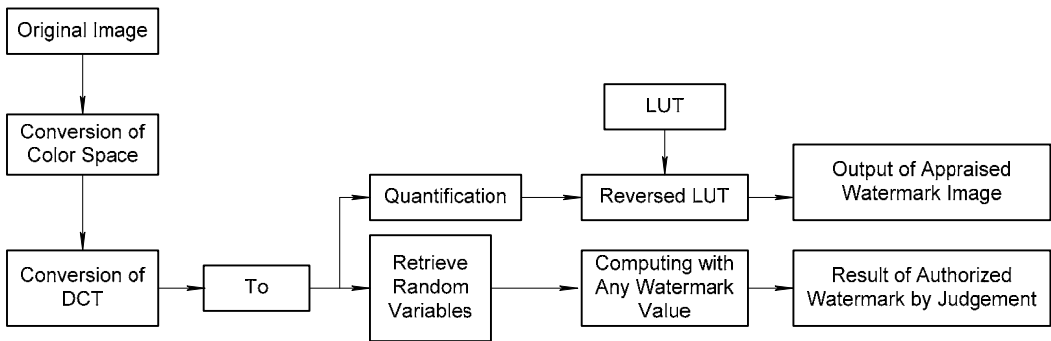
In Figure 3, the host system sends images that need to be watermarked to the Nios II processor via a USB-Blaster communication interface. After applying double-layer digital watermarking, images are sent back to the host. Normal operations are handled by the host PC in real time. Arithmetic testing cannot be performed in the Nios II system, but can be done in the host PC. Therefore, we designed an easy-to-use host system that features all control and testing functions.

Figure 3. Digital Watermarking Flow Chart
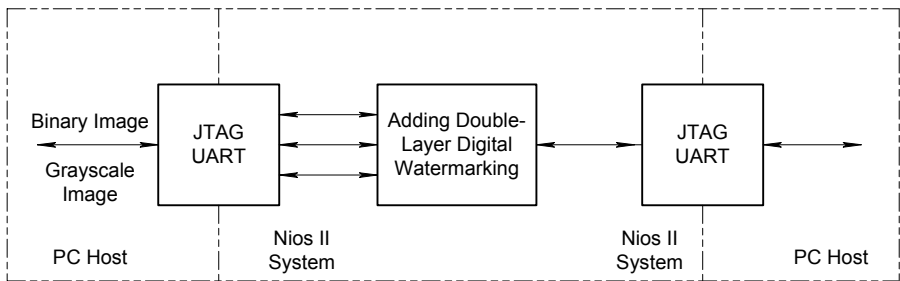
In Figure 4, external SDRAM has more memory space in the Nios II hardware system, and the SRAM can be used for faster storage and access. Therefore, we employed multi-master bus technology to

install SRAM to handle faster data requiring rapid storage, process some commands, and variables of operating system. On the same bus, we can use the peripheral 8-Mbit SDRAM as data cache.

Figure 4. Nios II Hardware System
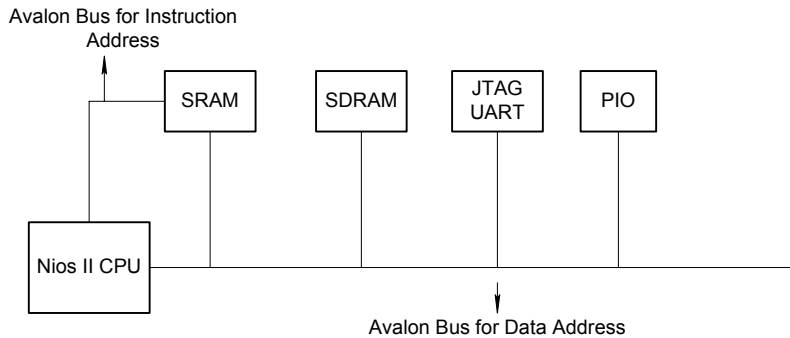
Design Methodology

Our design methodology involved the following steps:

1. Design the arithmetic algorithm and improve simulation to enable the Nios II processor to execute parallel processing fully.

2. Propose a system hardware block diagram and simplify the design flow by deploying powerful IP cores for easy design configuration.

   a) This task involved communication between modules that have different interfaces. Using Altera's system-on-a-programmable-chip (SOPC) design technology, it was easy to create the control interface for these modules, avoid extra peripheral circuits, and reduce overall system design costs.

   b) We could add the IP cores of SRAM, SDRAM, and flash memories directly. However, this method can lead to bus collusion. Therefore, we designed a bus-multiplexing module in HDL using the Quartus II software. Based on this design, three signals ensured a non-collision operation, which highlights the powerful flexibility and control made possible through the user-defined configuration of the Nios II processor. It also illustrates the tremendous advantage of the Nios II soft core, and highlights the fact that the combination of the Nios II processor and an FPGA meets the future market demands perfectly.

   c) We chose the CPU based on available hardware resources. Considering the actual situation of the EP1C6, we used it as the standard CPU for the design, which was sufficient to meet our needs. This selection illustrates the advantage of using an embedded CPU like the Nios II processor. When you choose a CPU in a hard-core development board like the ARM MCU, it cannot be changed. Therefore, this design approach does not provide design configuration flexibility, leads to low design re-use, and has limited application usage when compared with the Nios II processor.

   d) Because we could not use the USB interface for high-speed data transmission, we chose JTAG UART serial communication IP core to communicate with the host PC. We selected the LCD IP core to display prompt information on the LCD. Using the IP cores can save significant time and cost when implementing peripheral circuits and interface protocols. Additionally, components can be easily added or removed, and component parameters can

be changed. By using the Nios II processor, we can fully accomplish the configuration of an embedded system from CPU to peripheral circuits, as planned.

e) Because the current SOPC Builder library does not have a built-in way to target the flash parameters of the UP3 board, we needed to use a separate Flash Programmer User Guide, which was provided in the installation directory of the Nios II processor, to program the flash memory. After debugging and compilation, we successfully programmed the flash memory to perform the whole FPGA data configuration.

3. Employ the Nios II IDE for software development.

a) On a stand-alone PC, we used a C program to compile and debug software.

b) We built new modules in the IDE, finished control of the LCD display by programming a module available in the LCD template. We also needed to debug the main program in IDE mode. First, we needed to burn the hardware into the FPGA, then we programmed the peripherals. During debugging, we thought that utilization of the Nios II software was not sufficient. "Run as Hardware" needed to be recompiled before being written in hardware language for each iteration, so the debugging process was very slow.

c) For the read/write of the host-PC file system, we used the Host Based File System in the Nios II processor version 5.0 in the Debug As Hardware mode. When we tried the Nios software version 1.1, it failed. We think it would be useful if the Host Based File System could be performed in a "Run As" mode condition.

d) The program can be run under SDRAM and SRAM, separately. To conserve data in case of a power failure, we can burn the program into flash memory by using a flash programmer. Then, we reset the address of the CPU to flash memory, and we can automatically configure the FPGA after power up. Then, the program and files can be read into SRAM and SDRAM easily.

4. We designed a host system with test functions and a user friendly GUI, which intuitively shows the effect after Nios II processing.

In this design, the algorithm arithmetic of adding watermarking is performed on the Nios II development board. Normally we do this addition using a host PC in real time. In a practical application, the test arithmetic cannot be performed on the development board, but we can do it on a PC. Therefore, we needed to design a host system with total control and testing functions, and a user-friendly graphical user interface (GUI). Using this GUI, users can intuitively operate and implement the double-layer digital watermarking system.

Because the pre-simulation of arithmetic calculations was performed in the MATLAB software, and the testing arithmetic involves many complicated mathematical operations, we designed the host control, testing, and GUI using the MATLAB software.

The host system is mainly designed for testing watermark copyrighting and integrity. If you need to test watermark copyright, click **AuthWatDect**. Following this command, the threshold is obtained after computation of relevant peaks and judging whether watermarking has been added; if so, the program determines which watermarking was added. If you need to test the watermark integrity, click **OriImgDect** to test the LUT; if the image is not modified, the testing diagram of the LUT is white, otherwise, black spots appear on the modified parts.

Design Features

To perform two-layer digital watermarking, we designed the system using a local restructuring concept to come up with a strong anti-interference capability for the watermarking function. This process also made it possible to inspect the watermark effectively without the original document. Additionally, the embedded basis of two-layer watermarking algorithm allows us to distinguish, reliably, if images have been modified, and when modified, those areas are marked. We can deploy the watermarking algorithm in a digital image application and have it processed in the frequency domain. The algorithm uses a visual masking model for extra robustness. The two-layer watermarking algorithm also features the IP protection and production authentication functions.

The Nios II processor aids in algorithm implementation in the following aspects:

■ To evaluate the execution of algorithm, we need to look at the DCT transformation, random number generation, and local weighting factor calculations in the visual model. Thanks to the powerful processing capability of the Nios II CPU, we can add user-defined instructions and a hardware accelerator (such as a DCT/IDCT hardware accelerator). Additionally, the Nios II processor's broadband converting architecture supports multiple parallel data channels to speed up overall processing.

■ A programmable system is necessary to generate the algorithm according to a hardware-based digital watermarking system. Therefore, we naturally thought of using an FPGA in our design. During the design stage, we faced the problems of development costs and technical difficulties. We were able to overcome these problems because the Nios II embedded soft core delivers better price/performance and lowers technical hurdles in development. Using the Nios processor we were able to implement the processor, peripherals, memory, and I/O interface on a single FPGA, which helped us to reduce the total system cost. Additionally, Altera's comprehensive developing tools for the Nios II processor and GUI provided many optimized IP cores, which reduced the software development cost. Therefore, we were able to pay more attention to the design details and completed the digital watermarking system.

■ The digital watermarking technology we chose to design is a hot IT technology at present, and will have a wide application in the future. Altera's SOPC design approach allowed us to keep the technology lead in digital watermarking, but also can ensure an early entry into the market with hardware-based products. In this way, we can reap benefits quickly.

The Nios II soft core processor offers flexibility, which helps make choices between multiple system setup combinations to achieve optimum performance, features, and cost goals. Using the Nios II processor in a design allows us to put our products into market faster and prolong the product lifecycle. According to the varying requirements of our users, the Nios II processor can be implemented using on-site hardware and we can add software upgrades easily. In this way, we can make the product comply with the latest specifications, have the latest features, avoid processor obsolescence, and hold off competition.

■ The two-layer digital watermarking technology software should be created on different Nios II hardware systems to provide a strong migration capability. The design contest required us to create the design on the UP3 board. However, we believe the software part of the two-layer watermarking algorithm can be migrated easily to other FPGA development boards.

Altera has embedded MicroC/OS-II in the Nios IDE, which let us set up RTOS for Nios II processor applications quickly.

The MicroC/OS-II RTOS provides portable, modifiable, reducible, real-time multi-task functions. Therefore, a MicroC/OS-II based program can be migrated to other Nios II hardware system easily without worrying about the low-level hardware. Additionally, the MicroC/OS-II program supports all HAL services, and you can use it to invoke the API function in HAL. The RTOS can run on different processors to provide the same API interface for users.

Because our algorithm is quite complicated, the multi-tasking operation of MicroC/OS-II RTOS fully exerts CPU utilization and modularizes the application. When designing complicated applications, designers often adopt a hierarchical model to make it easy to design and maintain the software modules. Additionally, the MicroC/OS-II RTOS allows you to split the application into several tasks, further simplifying the design of the system.

■ We designed the Nios II flow diagram after the overall design and validation of the algorithm. The key problem was how to implement the design with the Nios II processor better and faster. The technical support and detailed help files of the Nios II processor enabled us to have a quick start and rapidly master the programmable design skills. We believe that our two-layer watermarking algorithm is an optimized implementation.

Conclusion

This contest helped us better understand the Nios II processor. We think the Nios II processor will be widely used in the future because of its new design methods and modes, especially the SOPC concept, which is described as follows:

■ The Nios II processor complies with the developing trend of industrial technology—*software-like hardware* design. Using the Nios II processor can reduce the developer and material costs, thereby improving competitiveness. Additionally, the software-like hardware design makes simulation to hardware easy, reducing hardware design errors.

■ The Nios II processor enhances system robustness, which is an advantage of a single chip solution. The Nios II soft core and its development platform help the developer to build most of the modules flexibly. Corresponding drivers can be developed for most of the peripherals used, minimizing design errors.

■ The Nios II processor helps users to protect their intellectual property. Primarily, we can prevent reverse engineering when we use the Nios II processor in our designs.

■ The Nios II processor is significant, because it exploits a new, development space for FPGAs implementing SOPC-based designs. The UP3 board we used uses a Cyclone™ FPGA, and provides excellent support with a great number of logic units. Additionally, the Cyclone devices's low cost and its configurability will push DSP users to use the combination of FPGA and the Nios II processor for system designs to achieve better performance.

The success of this Nios II design development was possible due to:

■ The Nios II processor's excellent hardware scheme, which made it possible to complete the design with merely a few external hardware parts and allowed us to utilize the development board's resources fully.

■ Judicious usage of the Nios II processor's flexibility helped us to implement the synchronization of various hardware modules and design software, which sped up the design significantly.

The experience we had from the design is described as follows:

■ When designing with conventional (hard core) processors, the designer needs to pre-plan how to generate address decode and control signals. Once errors occur during validation, it is very difficult and complicated to make the necessary modifications. The Nios II processor eases this situation. You just need to connect the corresponding function module with the FPGA, paying attention to the interface and timing. Additionally, SOPC Builder provides a lot of parameterized IP and hides a great deal of initialization details from the designer, such as the UART baud rate, flash timing, and address line, making it easy to develop the system. The tool also provides complete C language header files and hides hardware details, which simplifies software development.

■ Abundant peripherals and easy integration. SOPC Builder helps designers design with SOC and IP. Besides providing many IP cores based on Avalon bus, the tool also supports an open IP integration environment. The users can easily integrate their IP with SOPC Builder, which protects their IP and promotes design reuse.

■ The Nios II soft core processor still needs further improvement, however. It is affected by system design complexity, the FPGA, and some other factors when the CPU wants to run with a stable frequency. Sometimes the invoked phase-locked loop (PLL) module is not stable enough in the actual debugging process. The hardware prompt "leaving processor paused" occurs occasionally, which may be due to variance of the actual frequency from the PLL and that of the marked frequency. In this case, you could power off to restart the board, or reprogram the hardware a few times.

■ The tool version and license need to be handled carefully when using the Quartus II software and Nios II IDE. You need to upgrade if there is a new version of the tool. There is no problem with hardware settings and hardware during earlier debugging efforts. However, an error always occurs in the SOPC Builder tool when reading and writing host system files. We wanted to realize it by means of a JTAG UART. We used version 1.1 of the tool previously, which does not contain the software component of host based file system. We often failed at the very beginning when we wanted to operate the host file with the standard C library function, using the example of character device operation using the string character. Then we upgraded to Nios II version 5.0, which adds the "filling the file on the pc" module. Its drawback is that it works only in debug mode. We hope that Altera can address this problem in future versions. In addition, it is better to upgrade the Quartus II tool to the latest version, 5.0, to avoid licensing problems.

■ When Altera introduced the Nios II soft core processor, abundant documentation was provided on the Altera web site, ranging from hardware operation and software guides to a many software routines. Altera also made timely updates. Although there is a big difference between the Nios II and Nios processors, we solved the general problems by referring to the detailed technical documents. However, in our opinion, the documentation is not completely reliable sometimes. For instance, we could have failed if we had followed the guide when we made a flash programmer with target board. Finally, we implemented it by adding asmi and some other modules. If more Nios II guidebooks are available in the Chinese language, using the Nios II processor would be much easier.

Third Prize

Portable Vibration Spectrum Analyzer

Institution: **Institute of PLA Armored Force Engineering**

Participants: **Zhang Xinxi, Song Zhuzhen, and Yao Zongzhong**

Instructor: **Xu Jun and Wang Xinzhong**

Design Introduction

We designed a portable vibration spectrum analyzer based on the Altera® Nios® II soft core processor and FPGA. The instrument is used in fault monitoring and diagnosis of rotary machines, which are used in battle tanks, armored cars, and vehicle engines. The operation of these vehicles may be affected by abnormal vibrations for different reasons, including serious accidents that may lower fighting strength and productivity. To solve the vibration diagnostics problem, we planned to design a portable vibration spectrum analyzer. Our instrument can analyze the vibration spectrum of rotary machines such as engine and gear case in real time.

First, vibration signals are collected by vibration sensor and sent to the FPGA after being processed by a high-speed A/D converter. Next, we perform digital filtering of these signals using the FPGA and send the data to the Nios II processor for fast Fourier (FFT) transformation using hardware acceleration, Finally, the Nios II processor analyzes the spectrum of transfered results and displays the relevant time domain waveforms and spectrum curves as well as a few major parameters such as major peak value and major lobe frequency on a color LCD.

Our system can display both time domain waveforms and spectrum curves in real time and store the required waveform and frequency into flash memory through a key-press action. Playback of waveforms is also available. By observing the spectrum curve, a technician can zero in on some abnormal vibration frequency and troubleshoot the faulty condition. In this way, imbalance, misallignment, and bush fragmentation may be expediently detected. Using the instrument, mechanics can quickly handle problems and avoid accidents and potential damage to vehicle engines. The systems' reliable multi-task real-time operating system (RTOS) µC/OS handles management tasks. The 256-color, 320 x 240 LCD display helps to migrate µC/GUI to the system; the systems' graphical user interface (GUI) enables convenient and user-friendly operation. The instrument can be used to monitor and analyze rotary machine vibrations and thus offer considerable military and economic benefits to users.

We chose Altera system-on-a-programmable-chip (SOPC) solution including the Nios II processor for the following reasons:

■ The Nios II soft core processor is implemented in an FPGA, changing the traditional microcontroller unit (MCU) plus FPGA system. The device combines control and digital signal process functions into the FPGA and enables a system on a chip that results in compact designs with reduced power consumption.

■ The Nios II based system has headroom for system upgrades. Because Nios II is a soft core processor, you can upgrade the CPU if you do not need to alter the peripheral hardware; in this way, designers can enhance system performance and prolong product life cycles.

■ The system uses several digital signal processing functions such as FFT and finite impulse response (FIR). With the matching development environment, we can customize the peripheral intellectual property (IP) based on Avalon® bus using customized instructions. By doing so, we have greatly improved the digital signal processing capacity of the system and realized associated logic circuitry using the FPGA.

In a high-speed digital system, faster signals make a transmission line of a PCB connection, and therefore signal integrity is impacted because of crosstalk, connection topology of chips, pin distribution and package, geometric and electrical property of the PCB, and voltage reference panel. Integrating these high-speed signals into an FPGA can solve most of these problems. In addition, this approach makes the best use of FPGA resources.

Function Description

The system enters into the wait master display after power up; the display shows design name and function overview; a prompt to press any key to continue appears at the bottom of the display; on pressing a key, the system enters master mode.

The structure of system master display is shown in Figure 1.
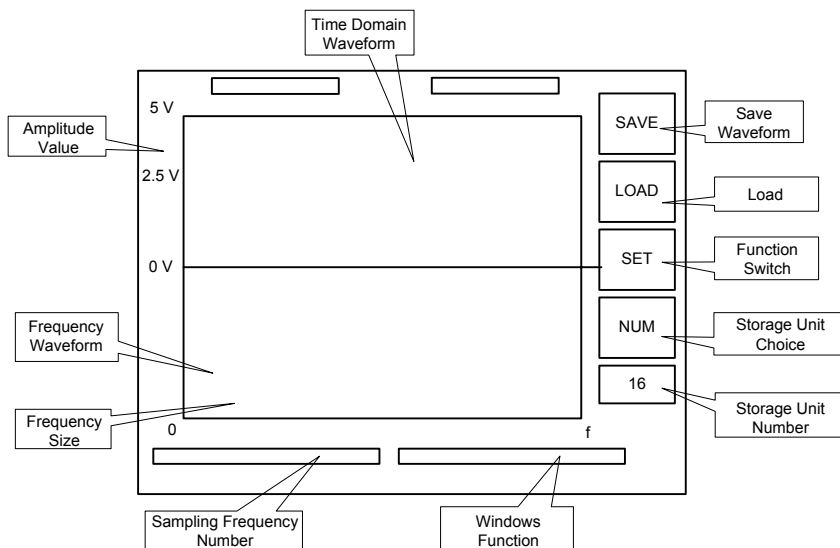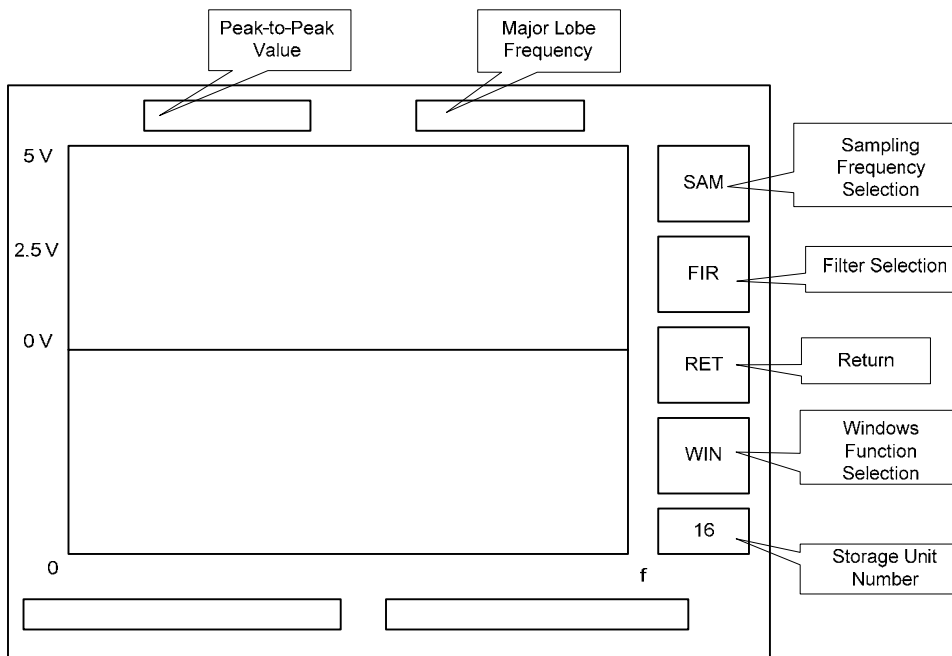
Figure 1. System Master Display

Figure 2 shows the second level menu options.

Figure 2. Second Level Menu Options

The systems' major functions and their implementation are as follows:

■ *Real-time display of spectrum and real-time measurement of main lobe frequency*—The real-time display and measurement of vibration signal spectrum are major functions of the system. Observation of vibration spectrum will help to detect vibrations that may cause fault or danger, making it possible to troubleshoot engines. After entering the main menu, the system will automatically perform FFT for the collected time domain digital signals. Next, these time domain signals are transformed into frequency domain signals, and spectrum analysis is performed and a spectrum curve is displayed on LCD. Because we have used a customized hardware floating-point multiplier to accelerate key algorithms in the FFT, it takes less than 0.1 s for one 512-point FFT. Because a naked eye can only distinguish 10 frames/s flushing speed, the software FFT algorithm accelerated by application hardware meets the required system performance for a real-time display of signal spectrum. To help the technicians, we have designed a real-time display function for easy display of main lobe frequency.

■ *Real-time display and measurement of time domain waveform*—The time domain signal waveform is an important reference for engine diagnostics. To facilitate comparison with frequency and relevant analysis, the time domain waveform is displayed in real time on the waveform area. The time domain waveform features amplitude coordinates, from which the amplitude information can be measured. On the time domain waveform, the peak-to-peak value of time domain waveform signal is also displayed. Any change in the amplitude of vibration signal can be detected through observing the peak-to-peak value, helping in easy diagnostics.

■ *Storage and playback of time domain and frequency domain waveforms*—To facilitate diagnostics using analysis and review of time domain and frequency domain waveforms, we have designed storage and playback functions of these waveforms. When any time domain waveform is deemed useful by technicians, you can press the **Save** button to store it. While storing, pressing **Num** button

230

changes the storage position. The storage action saves the current waveform and also the peak-to-peak value and main lobe of spectrum. The system can save 64 frames of waveforms and spectrum data on flash memory.

■ *9-level adjustable sampling frequency*—To improve the frequency resolution of the sampling signal, you can press the **SAM** button to set the sampling frequency of A/D controller and set the 9-level sampling frequency using a 4-bit control word. The sampling frequency is implemented through a controlling hardware-frequency divider.

■ *Hardware-only digital filter accelerates digital signal processing*—This system performs hardware-only digital filtering on collected signals. Three states have been set including, high pass, low pass, and no filter, selectable through the **FIR** button. The maximum and minimum frequencies of high-pass filter and low-pass filter are respectively, multiple values of 0.05x and 0.45y of sampling frequency. Because we have used a hardware filter, the system delivers excellent real-time performance. Because the FIR filter time cycle is shorter than the A/D transform cycle, there is no signal loss or delay. The digital filter uses an FIR algorithm to effectively filter out the interference noise of the device. The hardware filter uses a 16-tap direct FIR design and the filter parameters are fixed.

■ *Windows settings of waveforms*—For the system, we have set two window modes: a rectangular window, and a Hanning window. Because the system samples 512 points and processes it with FFT, it is equivalent to rectangular window in the sampling process; for a Hanning window, the system can restrain side lobe effectively. When time domain signal is obtained, it can weight the window function. The weighting function of Hanning window is:

$$w_H(n) = 0.5 - 0.5\cos(\frac{2\pi n}{N}) \ , \quad n = 0, 1, \ldots, N\text{-}1 \quad (2.1)$$

The button **WIN** adds a window and the effect is displayed in real time.

■ *Customized pulse-width modulation (PWM) peripherals generate standard waveforms for self-check*—By setting the peripheral PWM controller, a square wave signal with set frequency and pulse width can be generated. Before testing the vibration signal, the square wave signal can be tested initially so that the system can make accurate detection. The PWM controller based on the Avalon interface is easily customized and the waveform can be output through a program controlling the PWM.

■ *Migrate μC/GUI to the system for easy operation*—To make the display easy to operate, we transferred the graphic user interface μC/GUI to the system. With the display interface, diagnostics can be easily made by observing the frequency curve and functions set through buttons.

■ *Management with μC/OS Multi-tasking Real-time Operating System*—Because the system has multiple tasks requiring real-time operation, we used an RTOS to manage the tasks. The Nios II integrated development environment (IDE) provides the μC/OS whch has been used in many MPU applications. In our system, we have assigned five major system tasks, such as the button scan, LCD display/refresh, A/D collection, FIR control and flash timing storage.

Performance Parameters

The most important technical issues for the dynamic signal analyzer are frequency range, accuracy, and dynamic range. The frequency range is the range of frequencies an analyzer can detect. This depends on

the A/D converter and sampling speed. In addition, this function is also related to the bandwidth of the modulation-adjustment amplifier filter.

Amplitude-value accuracy refers to the full range accuracy of a corresponding frequency. This parameter depends on absolute accuracy window flatness and electronic hash level; a typical single channel's absolute accuracy is ±0.15~±0.3dB and the matching accuracy between channels is 0.1~0.2dB. The phase error between channels is 0.5~2 [deg].

Dynamic range depends on the word value (digits) of the A/D converter; furthermore, this also relates to the stop-band attenuation and FFT arithmetic error of the anti-alias filter as well as the background noise of electronic instruments.

Other major technical issues are described in the following sections.

Input section

Input impedance: Impedance of test instruments without being powered up. Generally, it is about 1-M Ω, which does not impact testing when coupled with external impedance.

Input-coupled mode: DC and AC.

Input range: the allowable voltage range of input.

Amplitude value error: ±0.1~±0.3dB.

Phase error: ±0.5~±1.0 deg.

Triggering mode: Free running, input signal triggering, signal source triggering, and external triggering.

Triggering level: Enables the operation of instruments.

Section of Analysis

Frequency range: Signal range available for detection.

Sampling frequency: Generally, it is 2.56 x analyzed frequency range.

Sampling points: Number of data points used in FFT operations.

Window function: Weighting modes of window functions.

Average mode: Provides average values of linearism, exponent, and peak value.

Parameters of the System Design

We surveyed the available test instruments in the market, and fixed the major parameters in our system to be:

■   Frequency measurement range: 0~100 kHz

■   Dynamic range: 60 dB

■   Amplitude value accuracy: ±0.3 dB

- Input range: 0~5 v

- Amplitude value error: ±0.3 dB

- Phase error: ±0.5

- Sampling frequency: 788 Hz~200 kHz. This is separated into nine segments: 200 kHz, 100 kHz, 50 kHz, 25 kHz,13 kHz, 6,300 Hz, 315 Hz, 1575 Hz, and 788 Hz.

- Spectrum resolution: With different sampling frequencies, the resolutions are set at 393.8 Hz, 196.9 Hz, 98.4 Hz, 49.2 Hz, 24.6 Hz, 12.3 Hz, 6.2 Hz, 3.1 Hz, and 1.5 Hz.

- Time domain waveform range : 0~5000 mV.

- Accuracy of spectrum major lobe frequency: ±0.5%

- Sampling points: 512.

- Windows: RSectangular and Hanning

- Measurement accuracy of time domain peak-to-peak value: ±3%

- Time-domain-amplitude value error: ±3%

- Storage of time domain and frequency domain waveforms: 64 frames

Testing the Measurement Setup

To check our design and validate the test parameters, we carried out measurements for a group of sine waves, using a signal generator. Next, we changed the waveform and amplitude values to get different data and analyzed the results. Here is the list of several major parameters.

Experiment instruments:
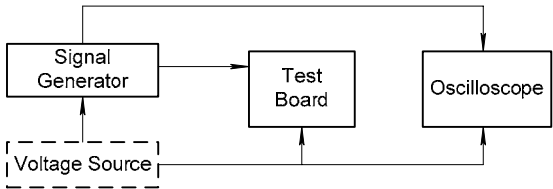
- JW—2B DC stabilization voltage supply   one

- GFG—8255A signal generator   one

- XJ4453A digital oscilloscope   one

- Test board   one

System Test Solution

The system test solution is shown in Figure 3.

Figure 3. System Test Solution

Experiment environment: room temperature: 24 degrees

Frequency Measurement Range

Measurement of lowest frequency:

Item Measurement Times	Set Value (Hz)	Measured Value (Hz)	Absolute Error (Hz)	Relative Error (%)
1	0.03	0	-0.03	0
2	0.7	0	-0.7	0
3	1.23	0	-1.23	0
4	1.65	1	0.65	
5	2	1	1	

When the lowest sampling frequency is 788 Hz, the lowest resolution of system is 1.5 Hz, so the error is comparatively large when measuring low frequency. In our system, frequencies below 1.5 Hz will be considered as 0 Hz. The error in low frequency collection is due to the deficiency of system resolution. Therefore, we need to improve the system resolution at a later stage.

When the sampling frequency is 200 Hz, the system can detect waveforms within a frequency range 100±0.4 kHz; considering the resolution, the waveform is deemed as 100 kHz.

Accordingly, based on this calculation, we think the frequency range of the system meets the design objectives. Further, in real applications, mechanical shocks contain low wave frequencies, making it possible to use our instrument without impacting measurements.

Spectrum Major Lobe Accuracy

Considering the massive data we need to process, for brevity's sake we show only data for the highest, lowest, and middle frequencies.

Method: Input the sine signals generated by the signal generator into the detection system.

Sampling frequency: 778 Hz   Resolution: 1.5 Hz

Item Measurement Times	Set Value (Hz)	Measured Value (Hz)	Absolute Error (Hz)	Relative Error (%)
1	36.8	36	0	0
2	24.1	24	0	0
3	11.7	12	0	0
4	14.5	15	0	0
5	8.8	9	0	0
Average			0	0

Sampling frequency: 200 kHz   Resolution: 393.8 Hz

Item Measurement Times	Set Value (Hz)	Measured Value (Hz)	Absolute Error (Hz)	Relative Error (%)
1	39.6 K	40.165 K	171	0.43
2	51.4 K	51.978K	184	0.36
3	21.0K	21.263K	0	0
4	5.03K	5.119K	0	0
5	60.38K	61.428K	652	1.08
Average				0.332

Sampling frequency: 25 kHz   Resolution: 49.2 Hz

Item Measurement Times	Set Value (Hz)	Measured Value (Hz)	Absolute Error (Hz)	Relative Error (%)
1	2.08K	2.116K	0	0
2	0.725K	0.738K	0	0
3	4.24K	4.331	42	0.99
4	9.98K	10.139	109	1.10
5	1.60K	1.624K	0	0
Average				0.402

Conclusion: Through analysis of the three different sampling frequency experiments, we feel the system accuracy of ±0.5% was achieved and it meets the design requirements.

Time Domain Peak to Peak Value Accuracy

Method: input the sine signal generated by signal generator into the digital oscilloscope and system.

Item Measurement Times	Set Value (Hz)	Measured Value (Hz)	Absolute Error (Hz)	Relative Error (%)
1	3.24	3.15	-0.09	2.77
2	2.61	2.53	-0.08	3.06
3	1.50	1.45	-0.05	3.33
4	3.82	3.76	-0.06	1.57
5	1.30	1.26	-0.04	3.07
Average				2.76

Analysis: The peak-to-peak value of waveform generated by signal generator is unstable in measurement. Therefore, the medium value is comparatively stable when taking the reading. Because the measured value of system is the mean value processed by the system, it is an accurate value.

Time Domain Waveform Amplitude Accuracy

| Item Measurement Times	Set Value (Hz)	Measured Value (Hz)	Absolute Error (Hz)	Relative Error (%)
1	2.40	2.36.	-0.04	1.67
2	1.83	1.89	0.06	3.28
3	1.06	1.10	0.04	3.37
4	0.84	0.81	-0.03	3.57
5	1.32	1.35	0.03	2.27
Average				2.832

Conclusion: With the experiment, the repeat (copy) accuracy of time domain waveform meets the requirements of our design.

Analysis and conclusion of system measurement: Through the analysis of experiment data, we were able to show that the basic performance of the system met our design objectives. The test indexes that were not perfect enough will be further improved.

Design Architecture

The hardware design block diagram is shown in Figure 4. The bold line highlights block diagram of the FPGA internal hardware circuit. The FPGA external circuit modules include A/D and signal-conditioning circuitry, keyboard, LCD, SDRAM, SRAM and flash memories.

Figure 4. Hardware Design Block Diagram
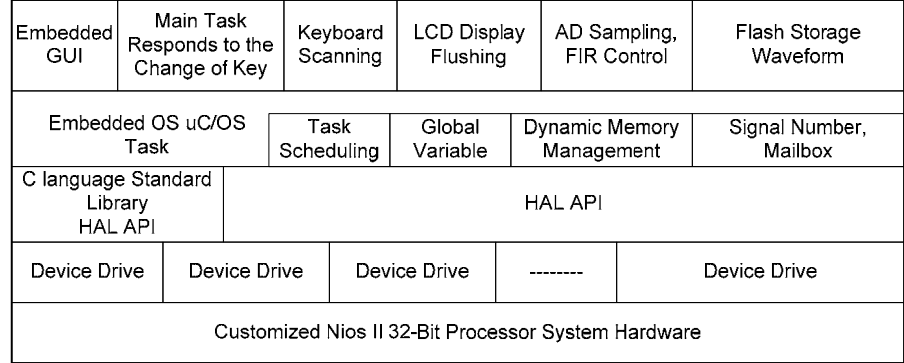

Stratix FPGA
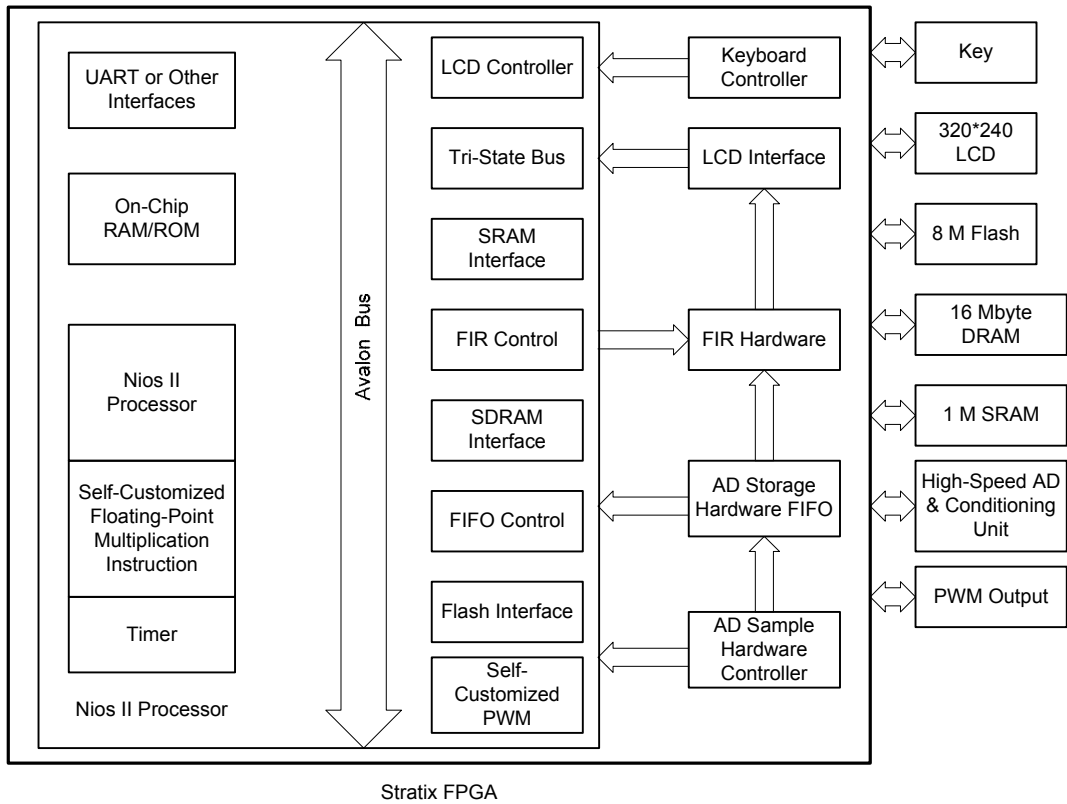
The software is implemented based on a Hardware Abstraction Layer (HAL) provided by the Nios II IDE. The software tasks are handled by a multitasking real-time operating system, the μC/OS II, which improves program readability and simplifies program development. We added a graphical user interface, μC/GUI for the LCD display to make it more user-friendly.

Six tasks comprise the software structure of the whole system, including the system main task, keyboard scanning, LCD display, A/D sampling, FIR control and flash timing storage. Where necessary, we can add other tasks.

The overall software structure is shown in Figure 5.

Figure 5. Software Block Diagram

| Embedded GUI	Main Task Responds to the Change of Key	Keyboard Scanning	LCD Display Flushing	AD Sampling, FIR Control	Flash Storage Waveform
Embedded OS uC/OS Task		Task Scheduling	Global Variable	Dynamic Memory Management	Signal Number, Mailbox
C language Standard Library HAL API	HAL API				
Device Drive	Device Drive	Device Drive	--------	Device Drive	
Customized Nios II 32-Bit Processor System Hardware					

Design Methodology

Our system design is based on Altera's SOPC solution. During the design process, we fully utilized the technical advantages of SOPC design for the system software/hardware synergy. By doing so, we were able to realize system functions within a very short time. A detailed description of hardware and software design follows.

System Hardware Design

Figure 6 shows the Block Design File (.bdf) diagram of the overall hardware design of system peripherals, including A/D controller, FIR filter, and A/D FIFO.

Figure 6. Hardware Design BDF
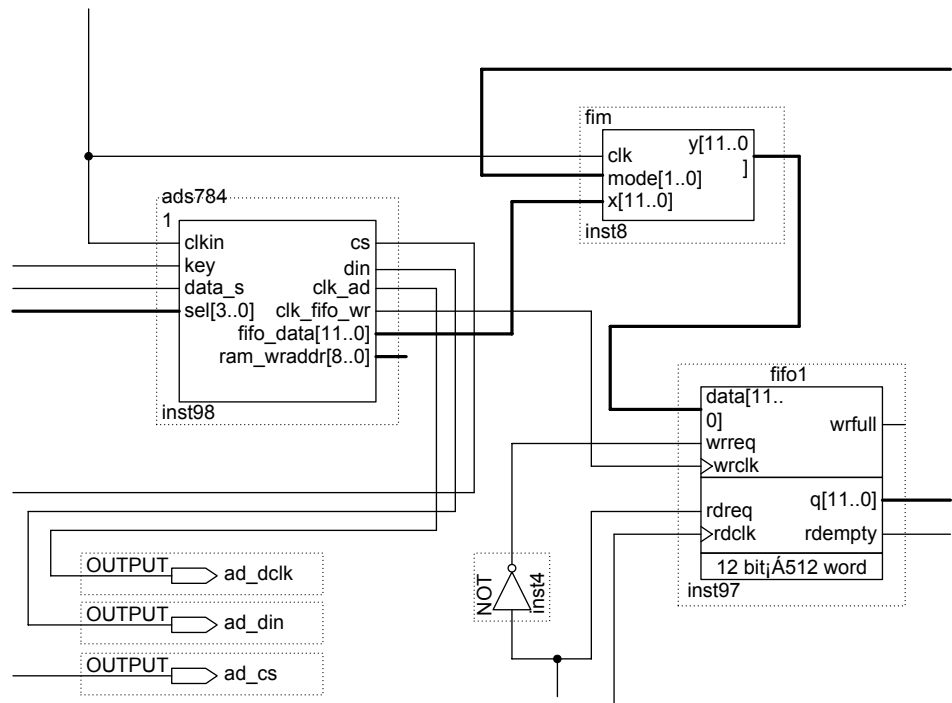
The symbol diagram of Nios II processor is shown in Figure 7. It shows the peripherals that Nios II integrates with the processor unit.
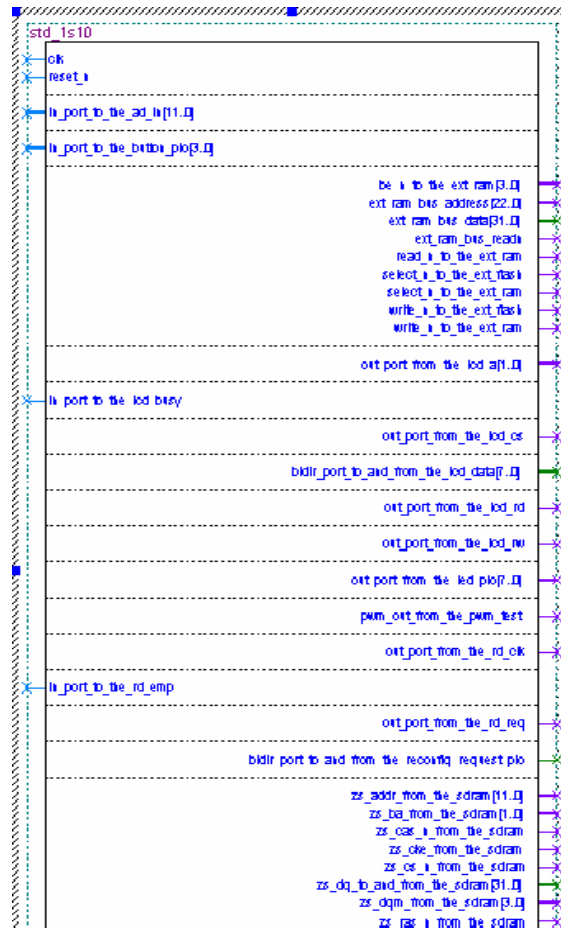
Figure 7. Nios II Symbol

Figure 8 shows the integrated IP modules of SOPC Builder.

Figure 8. Integrated IP Modules

In this system, the Nios II CPU uses the Standard type configuration.

Lcd_data, lcd_a, lcd_rd, lcd_rw, lcd_busy and lcd_cs are all peripherals of PIO type. They are used to simulate the control timing of an external LCD controller to control the LCD.

Ad_in: Data to send the sample data in FIFO to Nios II processor for processing.

Rd_clk,rd_req,rd_emp: Control line of A/D FIFO.

Selfir: Control line used for setting filter type.

Selfir: Control line used for controlling A/D sample frequency.

Pwm_test: Output port of self-customized PWM peripheral used for system self-test.

Now, we will describe the design process of each module in detail from the following aspects.

Design of PWM Peripheral Logic Based on Avalon Bus Interface

The Avalon bus structure promoted by Altera is used to connect the processor with its peripherals to build an SOPC system. Besides defining connection port between master device and slave device, the Avalon bus also defines the connection timing between the master device and slave device.

The Avalon data bus supports three widths: byte, word, and double word. When a transfer is finished, Avalon bus will perform a new transfer on the next clock cycle between either previous master and slave devices or new master and slave devices.

As a bus structure dedicated to SOPC design, the Avalon bus differs greatly from traditional ones. For better understanding of its architecture, we have to give detailed explanations on some words.

■ *Bus cycle*—A cycle of Avalon bus starts when the master clock rises and ends when it goes down. The bus cycle is used as reference for the timing of bus control signals.

■ *Bus transfer*—Avalon bus transfer is the reading and writing of data. It can take one cycle or multiple cycles according to the master and slave devices being used.

■ *Master port*—A set of ports on the master device. Directly connected to the Avalon bus, these ports initiate data transfer on the bus. One device may have several master ports.

■ *Slave port*—**A** set of ports on the slave device. Directly connected with the Avalon bus, these ports generate data interaction with the master port on Avalon bus. A master device may have slave port.

■ *Master/slave device group*—A group consisting of a master device and a slave device that both require data interaction. They transfer data through the master port and slave port and connect with the Avalon bus.

An Avalon bus comprises multiselector and arbiter. A system can have several Avalon bus modules. An Avalon bus features:

■ A maximum address space of 4-G bytes.

■ All signals are synchronized with its clock.

■ Offers independent address line, data line and control line for each peripheral, which simplifies peripheral interface.

■ The multiselector can automatically establish dedicated data channel for the transfer of data.

■ Can automatically generate chip select signals for the peripherals.

■ Its parallel multiple master device structure allows simultaneous data transfer of multiple master devices.

■ It has an interrupt processing function. Each peripheral has an independent signal line for interrupt request connected to the Avalon bus. The Avalon bus can generate the corresponding interrupt signal and then transfer it to Nios II.

Because of these advantages, Altera has added user-customizable logic to the SOPC system interface. As long as the interface and logic are designed and defined in accordance with specifications for the Avalon bus interface, the user-defined peripherals can be added to the system using development tools.

PWM Peripheral Function Design

We have designed the PWM peripherals to be Avalon bus slave peripherals. The bus controls the PWM by modifying its registers. The registers' addresses can be automatically mapped into the system, which can be modified in software.

The PWM is required to have functions as follows:

- *Set cycle*—Sets the number of cycle (using a 32-bit register) through clock_divide port, and sets the output cycle to be clock_divide times of clk, maximum $2^{32}=4294967296$ times of clk.

- *Set duty cycle*—Sets the ratio of low to high level (using a 32-bit register) duty_cycle, the value of which shall be less than that of clock_divide.

- *Control PWM output*—Decides whether PWM outputs or not with the control function.

Verilog HDL Design of PWM Logical Function

The core of PWM peripheral is a counter. It controls counter cycle with clock_divide, and the output is the result of the contrast between duty_cycle and counter.

Here is the Verilog HDL program source code for programming control cycle and pulse width.

```
always @(posedge clk or negedge resetn)            //PWM Counter Process
begin
    if (~resetn)begin
        counter <= 0;
    end
    else if(pwm_enable)begin
        if (counter >= clock_divide)begin
            counter <= 0;
        end
        else begin
            counter <= counter + 1;
        end
    end
    else begin
        counter <= counter;
    end
end

always @(posedge clk or negedge resetn)        //PWM Comparitor
begin
    if (~resetn)begin
        pwm_out <= 0;
    end
    else if(pwm_enable)begin
        if (counter >= duty_cycle)begin
            pwm_out <= 1'b1;
        end
        else begin
            if (counter == 0)
                pwm_out <= 0;
            else
                pwm_out <= pwm_out;
        end
    end
    else begin
        pwm_out <= 1'b0;
    end
end
```

The timing simulation of PWM peripheral is shown in Figure 9.

Figure 9. PWM Simulation
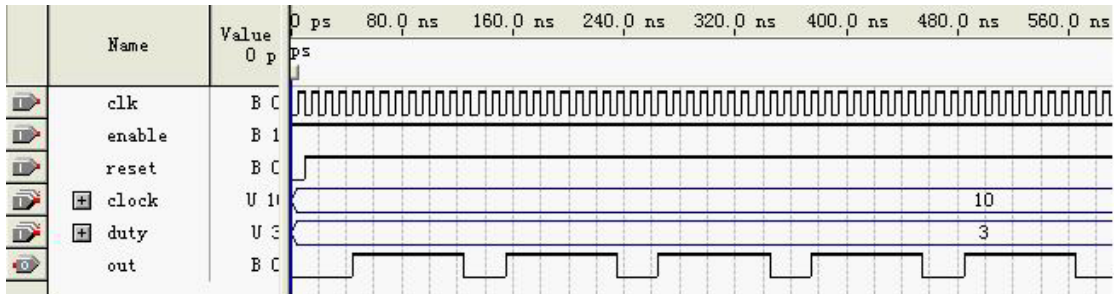
Design & System Integration of Avalon Interface Files

After finishing the design of PWM functions, we moved onto design the interface timing between function modules and Avalon bus. The interface timing is mainly responsible for transferring the bus signals to the register on control-function module to handle the communication between bus and control registers. The bus signals that are related to slave interface peripherals include clk, resetn, chip_select, address, write, write_data, read, and read_data. The address is mainly used to transfer the bus address and copy the address to register. Then, after selecting the appropriate control registers, the signals including write, write_data, read and read_data will perform reading and writing operations on these registers.

Customization & Integration of Hardware Floating-Point Multiplication, Addition & Subtraction Instructions

Floating-point multiplication is generally used in digital signal processing (DSP) algorithms. Because the Nios II processor does not have a floating-point multiplication instruction, defining a hardware floating-point multiplication instruction will remarkably speed up related DSP algorithms. This was implemented by taking advantage of the Nios II system's well-defined interface that allows user-defined hardware instructions. We simply need to call these user-defined instructions in the program to complete the execution of algorithms.

IEEE Standard Single-Precision Floating Point Number Standard

IEEE754 criteria defines binary floating-point number standard as 32-bit (single precision) and 64-bit (double precision) numbers. Because the system uses a 32-bit floating-point number, detailed instructions are defined for single-precision floating-point number. For standard of double precision floating-point number, please refer to related material.

It is indicated as:

$$N=(-1)s*M*2E\text{-'z'} \qquad (5.1)$$

N in (5.1) indicates floating-point number; S is sign bit value (positive number when the 31st bit is 0, and negative number when the 31st bit is 1). E represents 8 binary index from the 23rd to 30th bit, whose value is that from 0 to 255, and hence can indicate value between 2-m to 2}z'. M refers to binary decimal shown by mantissa. It is indicated as:

$$M=1+m2\}2\text{-'}+mz\}2\text{-}z+m\}o2\text{-' } \} - mot \qquad (5.2)$$

m in the formula indicates the ith number in the corresponding mantissa, which is 0 or 1.

It can be displayed as 1.0101 * 23, its sign bit in corresponding floating-point is OB, index part is 10000010B, mantissa is olo_ loo0_ 0000_ 0000_ 0000_ ooooB, and the whole 32 bit figure shows the value is 0100 0001 0010 1000 0000 0000 0000 0000B.

In practice, there are 5 cases for M and E, which should be processed differently:

1. (1) E=255,M-1}0   N does not represent a number

2. (2) E=255,M   1=0   N = (-1)s*

3. (3) 0<E<255   N = (-1)s * M * 2E-'z'

4. (4) E=O,M   1 } 0   N = (-1)s * (M   1) * 2}-}z}

5. (5) E=O,M-1= 0   N = (-1)s * 0

During floating-point operations, it is overflow as in cases 1, 2 and E > 255, and zero as in case 5.

## Design of Single-Precision Floating-Point Number Multiplication

According to IEEE754 standard, the algorithm of single precision floating-point number multiplication can be de divided into sign bit calculation, index bit calculation, and remainder multiplication calculation. The first two are easy to realize. The sign bit is two multiplicands sign bit; the index calculation is done by adding two 8-bit binary integers without symbol and subtracting 127 from the result; and then estimate whether the overflow summation is 0. The remainder multiplication calculation is the most difficult to implement because of floating-point number multipliers. The remainder multiplication calculation can be transformed into multiplication of two 24-bit integers without symbol. Therefore, the key to the designing of floating-point multiplier lies in the realization of a high-performance multiplier hardware of 24-bit integers without symbol.

The basic design concept of the current hardware multiplier is consistent with manual multiplication operation. First, obtain partial products, and then add partial products to get the result. The calculation is clear and easy and needs less hardware resources. However, it suffers from time delay disadvantage, with increasing multiplier digits. To reduce the computing time, you can consider using many improved algorithms, such as Booth, improved Booth, Wallace Tree, and Dadda.

Integration of Customization Instruction

The guide can be used to integrate custom instruction into the system after the designing of hardware multiplier and the corresponding interface unit (see Figure 10). The zxx_fp_mu instruction planted in the left library can be seen after the integration. In addition, the added customization multiplication instruction can be seen in right hand after clicking Add.

Figure 10. Integrating Custom Instructions
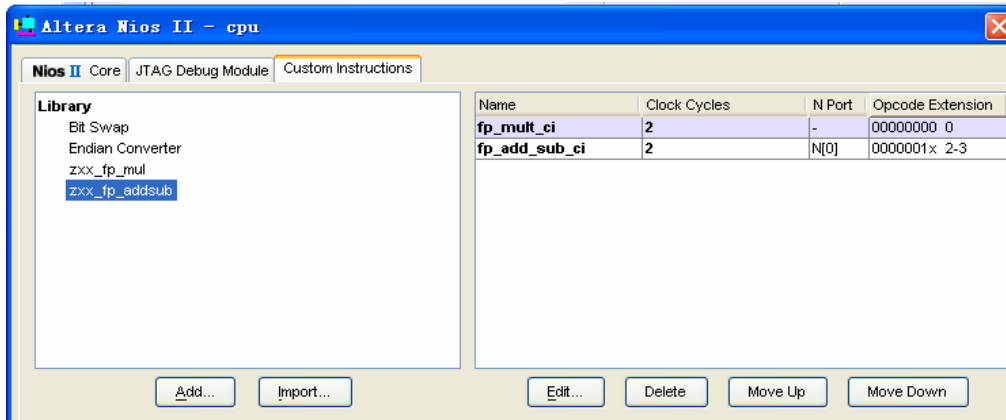
The following macro definition of floating-point multiplication and addition and subtraction can be seen in system.h after the compiling of the project document we have established. According to the macro definition, we could use ALT_CI_FP_MULT_CI(A,B) to operate hardware floating-point multiplication between A and B.

System macro definition document:

```
#define ALT_CI_FP_MULT_CI_N 0x00000000
#define ALT_CI_FP_MULT_CI(A,B) \ __builtin_custom_inii(ALT_CI_FP_MULT_CI_N,(A),(B))
#define ALT_CI_FP_ADD_SUB_CI_N 0x00000002
#define ALT_CI_FP_ADD_SUB_CI_N_MASK ((1<<1)-1)
#define ALT_CI_FP_ADD_SUB_CI(n,A,B)\
__builtin_custom_inii(ALT_CI_FP_ADD_SUB_CI_N+(n&ALT_CI_FP_ADD_SUB
_CI_N_MASK),(A),(B))
```

Designing a Pure Hardware FIR Digital Filter

There are always high frequency noises in vibration signal, which will affect the result of the spectrum analysis. In addition, unnecessary high frequency or low frequency signals are expected to be filtered in case of a specific pertinence; therefore, FIR digital filter is designed to filter unnecessary frequency waveforms. Concerning of real time requirements, FPGA logic resource is adopted to design pure hardware digital filter, to meet system requirements.

Operating Principle of FIR Digital Filters

Digital filter is a time-invariant discrete-time system used to complete signal filter processing with finite precision algorithm. Its input is a group of digital quantity and output is another group of digital quantity after transformation. The digital filter features in high stability, high precision, and high flexibility. As the development of digital technology, designing filter by digital technology is receiving more and more attention and application.

The system function of a digital filter can be indicated as constant coefficient linearity difference equation that shows input and output relations directly from H(z), i.e.

$$y[n] = x[n] \times f[n] = \sum_{k=0}^{L-1} x[k]f[n-k]$$

It can be seen that digital filter uses a certain operation to transform input serial to output serial. Most of ordinary digital filters are liner time-invariant (LTI) filters.

The digital filter can be divided into an infinite impulse response (IIR) filter and finite impulse response (FIR) filter according to the time characteristic of unit impulse response h(n). Concerning of the discrete time domain, it is called an IIR series if the system unit sample should be extended to infinite length; and a FIR series in case of finite length serial.

Compared with an IIR filter, a FIR filter has many unique advantages that can satisfy the requirements of amplitude frequency response when getting strict linearity phase characteristic to keep its stability. An IIR filter can be used for a non-linearity phase of a FIR filter. A FIR filter can be applied in wide ranging applications since the signal is required to have no clear phase distortion during data communications, voice-signal processing, image processing, and adaptive processing, whereas an IIR filter has a problem with frequency dispersion. For this reason, we have used FIR digital filter in this system.

Basic Structure of a FIR Digital Filter

A FIR filter includes three basic structures: direct form, cascade form, and frequency sample form. Direct form is the most popular structure and is adopted in our design. Therefore, we discuss only the direct form FIR filter here.

The direct form FIR filter is also referred to as tapped delay line structure or transversal filter structure. As you can see from the above table, each tapped signal is weighted by the appropriate coefficient (impulse response) along this chain, and you get the output Y(n)} via the addition of products.

Hardware Realization of FIR Digital Filter

The digital filter is based on the FIR algorithm, because it is more mature in filtering out random jamming. The filter hardware design is based on a 16-tap direct form FIR filter. The filter has a fixed coefficient. When the normalized frequency parameter is determined, the coefficient of the filter is first calculated with a math tool and then it is fixed in VHDL code.

The VHDL program of direct form FIR design is shown below:

```
   if clk'event and clk='1' then
     case modem is
     when "01" =>  -- high-pass
y<=(-2*(tap(0)+tap(15))-(tap(0)+tap(15))-
   (tap(0)+tap(15))/2+64*(tap(1)+tap(14))+16*(tap(1)+tap(14))
   -52*(tap(2)+tap(13))+41*(tap(3)+tap(12))-172*(tap(4)+tap(11))
   +2*(tap(5)+tap(10))+(tap(5)+tap(10))/2-385*(tap(6)+tap(9))
    +462*(tap(7)+tap(8)))/1024;
     for i in 15 down to 1 loop
      tap(i)<=tap(i-1);  --tapped delay line: shift one
     end loop;
     tap(0)<=x;
      when "00"    -----------------------low-pass-----
=>...........................
```

The advantage of direct form FIR filter design lies in the fact that you can get the result in a single period since parallel operations are made by multiple hardware multipliers and adders. However, this method uses up many logic resources due to parallel operations.

We can use a two-bit control word to select "High Pass," "Low Pass," or "None" status. The Nios II processor issues the control word, which can be set up using the SOPC Builder development tool.

Figure 11 shows the timing simulation.
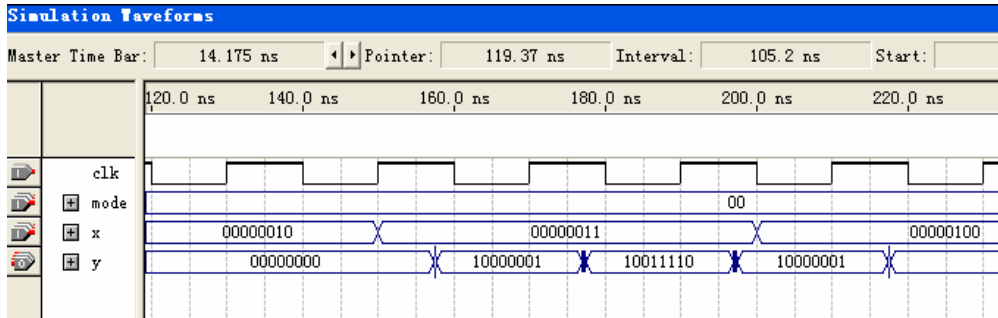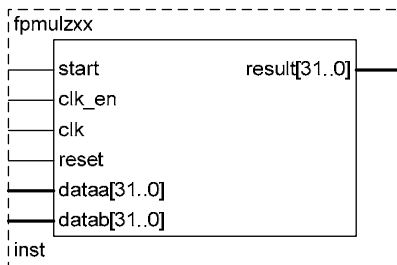
Figure 11. Timing Simulation

Figure 12 shows the symbol generated by the system.

Figure 12. Symbol

Design of A/D Sample Controller

To translate vibration sensor's simulation signal output into a digital signal, we deployed a serial 12-bit A/D ADS7841 to collect the sensor output. Keeping the main CPU focused on other system tasks, we designed an A/D hardware controller in an FPGA to control A/D samples and send sample data to A/D FIFO. Based on the control word output by the Nios II processor, we can also change sample frequency. Generally, frequency analysis error is due to the spectral leakage resulting from imprecise synchronization of sample window and actual waveform. Common methods to eliminate spectral leakage errors are based on hardware synchronization and windows processing techniques. Synchronization using phase-locked loop (PLL) circuitry is commonly employed in hardware synchronization. Therefore, a precise sample clock generated by an FPGA-based PLL circuit is used to implement strict sample synchronization to prevent overlapping and interval between windows, while synchronizing with the measured signal.

The ADS7841 device is a 4-channel, 12-bit sample simulation/digital converter with 8-, or 12-digit programmable output data under -40 to ~85 degree working temperature. The devices' typical power loss is 2 mW for a 200-kHz conversion clock and 5-V power input with a reference voltage from 0.1 V ~5 V. The ADS7841 features a power-down mode with 15 μW as the lowest power loss.

The basic circuit connection of ADS7841 is shown in Figure 13.

Figure 13. Circuit Connection
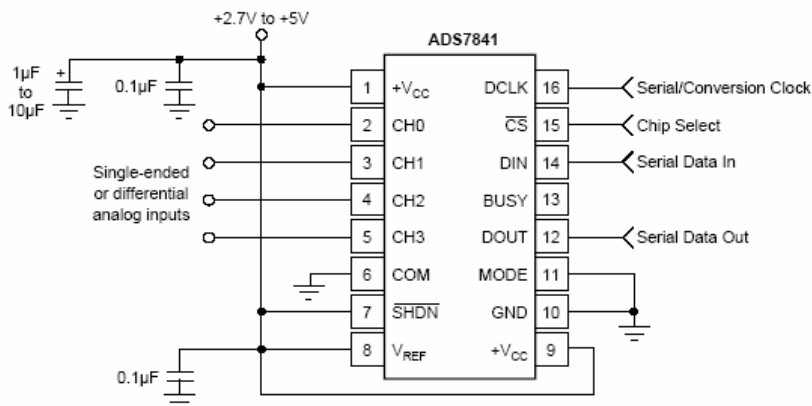
The device has an external reference and external clock pins with 2.7 V ~ 5.25 V as the working voltage range. The external reference voltage changes from 100 mV to +Vcc. The reference voltage has a direct influence on the range of input simulation signal. The input simulation signal circuit is determined by the ADS7841 conversion clock. The input of simulation signal is connected to one of the four input channels. The ADS7841 chooses the appropriate channel based on the control data input from DIN pin out (A0, A1 and A2) and SGL / $\overline{DIF}$. Relation between A0, A1, A2 and SGL / $\overline{DIF}$ and 4 channels and COM end is shown in Tables 1 and 2. This design is only for one channel input signal.

Table 1. Single-ended channel mode (SGL / $\overline{DIF}$ HIGH)

A2	A1	A0	CH0	CH1	CH2	CH3	COM
0	0	1	+IN				−IN
1	0	1		+IN			−IN
0	1	0			+IN		−IN
1	1	0				+IN	−IN

Table 2. Multiple-ended channel mode (SGL / $\overline{DIF}$ LOW)

A2	A1	A0	CH0	CH1	CH2	CH3	COM
0	0	1	+IN	−IN			
1	0	1	−IN	+IN			
0	1	0			+IN	−IN	
1	1	0			−IN	+IN	

The CHO channel can be selected for sample, therefore, A2=0, A1=0, A0=1 and SGL/ $\overline{DIF}$ =1 should be set in control word input from DIN. In order to output 12-bit data after conversion, the MODE pin out should be made low. In order to ensure normal operation of the ADS7841 device during data conversion, we should prevent the ADC from entering power-down or low power loss modes, by setting PD1 and PD0 to 1.

The timing diagram of ADS7841 is shown in Figure 13. The device needs 24 DCLK inputs to complete the conversion process. You need to program the ADC with the control word during the first, eight clocks. The conversion process enters sample mode when ADS7841 gets the control word denoting a specific channel for conversion. The ADC enters hold mode after the input of three DCLKs control word and then performs 12-bit data conversion after the lapse of 12 DCLKs.

From Figure 14, it is clear that the conversion clock frequency of ADS7841 F CLK=24 F DCLK. Due to hardware restrictions, the conversion frequency could utmost reach 200 kHz, in case of a 5-V power input. Because this design caters for only 3.3V levels of power and reference voltages, conversion frequency cannot reach 200 kHz. In addition, the conversion frequency cannot be too low as it relates to the discharge time of the ADC, confining the input sinusoidal signal to a certain frequency band.

The A/D conversion module accepts the simulation signal, converts it, and stores it in RAM. When all data has been converted and stored in RAM, the ADC begins to read data from RAM.

Figure 14. ADS7841 Timing Diagram
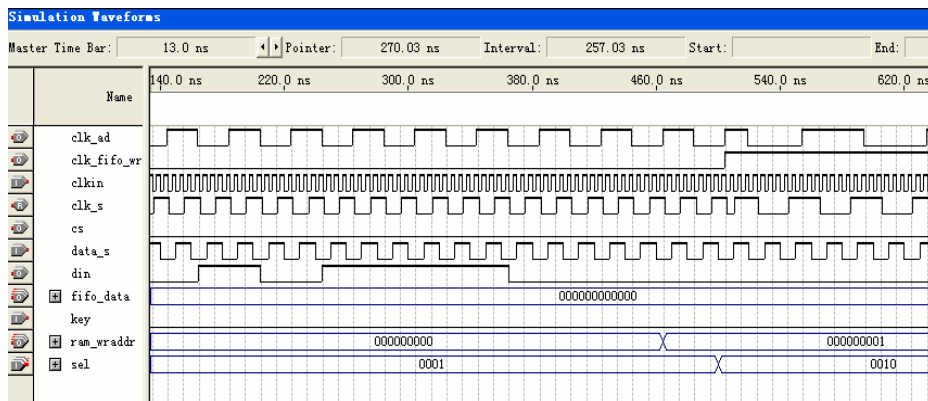
We need to program the A/D converter with control signals such as CS LD and control word DIN. Further, the output serial data from the ADC needs to be converted into a parallel format and fed directly to RAM. In addition, through programming, we need to realize RAM read/write, control clock, and address signals.

To effect changes in sample frequency, we designed a 4-bit control port that receives the control word sent from Nios II soft-core processor. Then, based on the received control word, the A/D controller changes sample frequency.

The VHDL source program is shown below:

```
    if clkin'event and clkin='1' then
    case sel is
    when "0001" => clk_s<=div(1);
    when "0010" => clk_s<=div(2);
 when "0011" => clk_s<=div(3);
    when "0100" => clk_s<=div(4);
    when "0101" => clk_s<=div(5);
    when "0110" => clk_s<=div(6);
    when "0111" => clk_s<=div(7);
 when "1000" => clk_s<=div(8);
    when "1001" => clk_s<=div(9);
    when others=>  clk_s<=div(1);
    end case;
    end if;
```

249

Figure 15 shows the A/D controller design.

Figure 15. A/D Controller Design
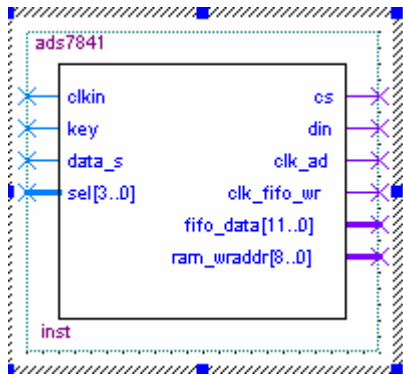
sel[3..0] is the sample-frequency control port.

Design of A/D Sample FIFO

The A/D sample data cannot be sent to the Nios II processor for immediate processing because the CPU has other scheduled tasks to perform. Therefore, some sample data needs to be buffered for processing by the ADC to save CPU time. Hence, we have designed a FIFO memory module to buffer A/D sample data. When data in the FIFO memory is full, the CPU starts processing it. We designed a dual-port, 512-word deep, 12-bit FIFO keeping in mind our 512-point FFT. We customized the FIFO design using components from Altera Library of Parameterized Modules (LPM). Figure 16 shows the customized FIFO symbol diagram.

Figure 16. FIFO Symbol

Figure 17 shows the FIFO time sequence.

Figure 17. FIFO Time Sequence
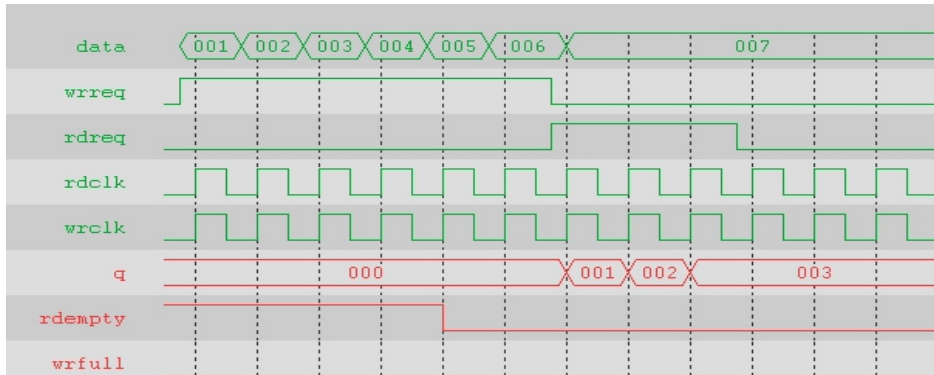
System Software Design

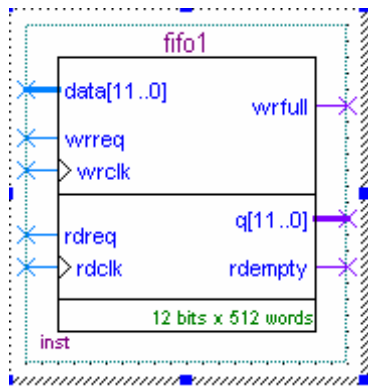The software design depends on the HAL API provided by Nios II IDE with the multi-tasking RTOS µC/OS II, managing all tasks. This approach greatly improved the readability of the program structure and made it easy for us to develop program modules. We wrote the LCD driver program and migrated the embedded GUI function packages (µC/GUI) onto the Nios II-based system. Utilizing the graphics functions provided by µC/GUI, we developed the waveform curve drawing function, window function and button operation function for a user-friendly display operation. Once again, programming with the µC/GUI greatly reduced our programming effort and made it possible for us to develop a user-friendly GUI. Figure 18 shows the software structure.

Figure 18. Software Structure

The system software comprises five tasks: system main program, keyboard scan, LCD display, A/D sampling, FIR control, and flash memory timing tasks.

For task intercommunication, we have used global variables instead of mechanisms, such as traffic handling, email, and message queue. When designing function tasks, we need to avoid transferring the same function to different tasks for function reuse. A detailed description of task designs will follow in the next section. The software design flow is described next.

Design of 320 x 240, 256-Color LCD Driver

Thanks to the simple interface of LCD drive panel, we could use the IO interface to simulate the control timing of drive panel. This was done by writing a simple driver program that was able to read/write data onto LCD. During the operation, the program could specify any one of 256 colors of the LCD.

The description of drive panel interface is as follows:

```
Ports:
CS  WR  RD  A1   A0   D[7..0]
H   X   X   X    X     HIZ
L   L   H   0    0     write data to controller
L   L   H   0    1     write X to controller
L   L   H   1    0     write Y to controller
L   L   H   1    1     write X to controller(for read data)
L   H   L   0    0     read data from controller
L   H   L   0    1     lock data written to X as parameter
```

According to the interface description, the driver program first writes in the coordinates x,y, then color data, when doing a read/write operation for a point on LCD. The subroutines of writing x, y coordinates, and color data are defined as follows:

Color data subroutine:

```c
void set_lcdwr_d_c(int x)
{
 IOWR_ALTERA_AVALON_PIO_DATA(LCD_A_BASE, 0);
 IOWR_ALTERA_AVALON_PIO_DIRECTION(LCD_DATA_BASE, 0xff);
 IOWR_ALTERA_AVALON_PIO_DATA(LCD_DATA_BASE, x);
 IOWR_ALTERA_AVALON_PIO_DATA(LCD_RW_BASE, 0x0);
  IOWR_ALTERA_AVALON_PIO_DATA(LCD_RW_BASE, 0x1);
}
```

X coordinate writing subroutine:

```c
void set_lcdwr_x_c(int x)
{ IOWR_ALTERA_AVALON_PIO_DATA(LCD_A_BASE, 0x01);
 IOWR_ALTERA_AVALON_PIO_DIRECTION(LCD_DATA_BASE, 0xff);
 IOWR_ALTERA_AVALON_PIO_DATA(LCD_DATA_BASE, x);
 IOWR_ALTERA_AVALON_PIO_DATA(LCD_RW_BASE, 0x0);
 IOWR_ALTERA_AVALON_PIO_DATA(LCD_RW_BASE, 0x1);
}
```

Y coordinate writing subroutine:

```c
void set_lcdwr_y_c(int x)
{
 IOWR_ALTERA_AVALON_PIO_DATA(LCD_A_BASE, 0x2);
 IOWR_ALTERA_AVALON_PIO_DIRECTION(LCD_DATA_BASE, 0xff);
 IOWR_ALTERA_AVALON_PIO_DATA(LCD_DATA_BASE, x);
 IOWR_ALTERA_AVALON_PIO_DATA(LCD_RW_BASE, 0x0);
 IOWR_ALTERA_AVALON_PIO_DATA(LCD_RW_BASE, 0x1);}
```

Subroutine for writing the x coordinate when reading a color value:

```c
void set_lcdwr_x_c_rd(int x)
{
 IOWR_ALTERA_AVALON_PIO_DATA(LCD_A_BASE, 0x3);
 IOWR_ALTERA_AVALON_PIO_DIRECTION(LCD_DATA_BASE, 0xff);
 IOWR_ALTERA_AVALON_PIO_DATA(LCD_DATA_BASE, x);
 IOWR_ALTERA_AVALON_PIO_DATA(LCD_RW_BASE, 0x0);
 IOWR_ALTERA_AVALON_PIO_DATA(LCD_RW_BASE, 0x1);}
```

Subroutine for reading color data:

```
unsigned int set_lcdrd_d_c(void)
{unsigned int m;
  IOWR_ALTERA_AVALON_PIO_DATA(LCD_A_BASE, 0x0);
  IOWR_ALTERA_AVALON_PIO_DIRECTION(LCD_DATA_BASE, 0x00);
 IOWR_ALTERA_AVALON_PIO_DATA(LCD_RD_BASE, 0x0);
 m = IORD_ALTERA_AVALON_PIO_DATA(LCD_DATA_BASE);
 IOWR_ALTERA_AVALON_PIO_DATA(LCD_RD_BASE, 0x1);
 return m;}
```

Subroutine for parameter look-up table:

```
void set_lcdrd_d_c_l(void)
{
 IOWR_ALTERA_AVALON_PIO_DATA(LCD_A_BASE, 0x01);
 IOWR_ALTERA_AVALON_PIO_DIRECTION(LCD_DATA_BASE, 0x00);
 IOWR_ALTERA_AVALON_PIO_DATA(LCD_RD_BASE, 0x0);
 IORD_ALTERA_AVALON_PIO_DATA(LCD_DATA_BASE);
 IOWR_ALTERA_AVALON_PIO_DATA(LCD_RD_BASE, 0x1);}
```

Subroutine for writing parameters:

```
void lcd_write_para(int para)
{ set_lcdwr_x_c(para);
  set_lcdrd_d_c_l();
}
```

Based on the data writing subroutines as above, the functions of read-dot and write-dot have been developed. The subroutine for write-dot is defined as follows:

```
void lcd_write_dot(int x,int y,int d)
{
     IOWR_ALTERA_AVALON_PIO_DATA(LCD_RW_BASE, 0x1);
    set_lcdwr_y_c(y);
  if (x>=256)
   {set_lcdwr_x_c(1);
     set_lcdwr_x_c(x%256);
     set_lcdwr_d_c(d); }
  else
    {set_lcdwr_x_c(0);
     set_lcdwr_x_c(x);
        set_lcdwr_d_c(d);
    }

    }
```

Subroutine for read-dot is defined as follows:

```
unsigned int lcd_read_dot(int x,int y)
{  unsigned int m;
    IOWR_ALTERA_AVALON_PIO_DATA(LCD_RD_BASE, 0x1);
    set_lcdwr_y_c(y);
    if (x>=256)
   {
   set_lcdwr_x_c(1);
   set_lcdwr_x_c_rd(x%256);
   m=set_lcdrd_d_c();
   return m;
    }
    else
   {set_lcdwr_x_c(0);
   set_lcdwr_x_c_rd(x);
   m=set_lcdrd_d_c();
   return m;
    }
}
```

Besides these basic LCD driver functions, we wrote other functions as follows:

void lcd_init_controlernios (**void**)

void showimage(**unsigned char** imageadd[],**int** imagesize) and other basic drive functions.

Migration of µC/GUI onto Nios II System

µC/GUI is a good graphics software for embedded systems developed by US-based Micrium Corporation. The software is open-source, portable, reducible, stable, and highly reliable. Using the µC/GUI, you easily display text, curves, graphics, and window objects (button, edit box, and bar-slide) on the LCD as you would on Windows OS. In addition, µC/GUI software provides a simulation library based on Visual C to help developers to simulate various effects of µC/GUI based on Windows OS.

Although µC/GUI can greatly reduce the difficulty of LCD display tasks in embedded systems, we need to develop separate driver programs to handle LCDs with screens of different resolution.

µC/GUI Structure

Software architecture of µC/GUI is shown as Figure 19. The µC/GUI function library provides user programs with a GUI interface including text, 2-D graphics, input device buttons, and window objects. The input devices could include keyboard, mouse or touch screen; 2-D graphics elements could include picture, beeline, polygon, circle, ellipse, and circular arc; window objects include buttons, edit box, progress bar, and checkbox. Using GUIConf.h header file, you can configure memory, window manager, support for OS and touch screen, as well dynamic configuration of memory size.

Further, you can define LCD hardware attributes such as LCD size, color, and interface functions in the LCDConf.h file.

Figure 19. Software Architecture
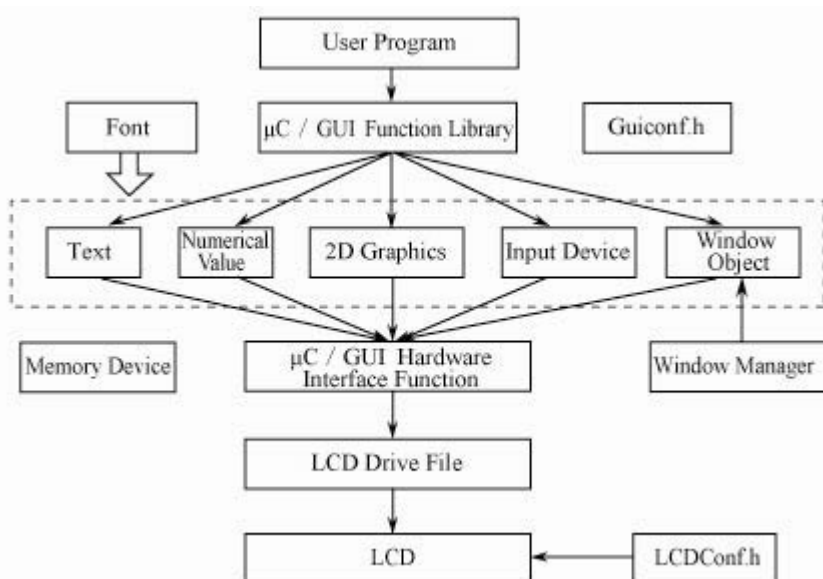
Migration Process

Modifying LCDConf.h Header file

The LCDConf.h file defines the size and color of LCD, and is modified to handle LCD parameters.

```
#define LCD_BITSPERPIXEL  8           //Bits Per Pixel
#define LCD_SWAP_RB        1            //if picture element DB is swapped

//Size of screen L and W pixel

#define SCR_XSIZE  (320)
#define SCR_YSIZE  (240)
#define LCD_XSIZE  (320)
#define LCD_YSIZE  (240)
```

The LCD read/write function is associated with hardware in which zxxniosdriver.c is customized, and the standard read/write functions are replaced with previously defined read/write functions.

```
static void SetPixel(int x, int y, LCD_PIXELINDEX c)
{ lcd_write_dot(x, y,c);}
unsigned int GetPixelIndex(int x, int y)
{ lcd_read_dot(x, y);}
```

Support options for the GUI can be changed by modifying the GUI.h file; when no LCD and memory devices are used, the values of the two devices are set to 0;

```
#define GUI_OS                  (1)  /* Compile with multitasking support
#define GUI_WINSUPPORT          (1)  /* Use window manager if true (1)
#define GUI_SUPPORT_MEMDEV      (0)  /* Support memory devices */
#define GUI_SUPPORT_TOµCH       (0)  /* Support a touch screen (req.
#define GUI_SUPPORT_UNICODE     (1)
```

In addition, several important files as above need to be modified, such as GUI_X.c and GUI_waitkey.c, but we will not discuss them here. System design can be performed directly by functions provided by the GUI when µC/GUI is migrated to the Nios II processor.

Software Optimization of FFT Algorithm Design

FFT Fundamentals

The fast Fourier transform (FFT) is an improvement on the discrete Fourier transform DFT algorithm.

The formula of a traditional DFT is as follows:

$$X(k) = DFT[x(n)] = \sum_{n=0}^{N-1} x(n) W_N^{nK} \quad , \quad n \leq k \leq N-1 \tag{5.3}$$

$$x(k) = IDFT[X(n)] = \frac{1}{N} \sum_{n=0}^{N-1} X(n) W_N^{-nK} \quad , n \leq k \leq N-1 \tag{5.4}$$

In which, $W_N^{nk} = e^{j\frac{2\pi}{N}nk}$

According to the formula, the result of $X(k)$ is obtained when every $x(n)$ term is multiplied by relative W_N^{nk} and then adding them. That is, N times of complex multiplication and N-1 times of complex addition. Computing $X(k)$ $(n \leq k \leq N-1)$ needs N2 times of complex multiplication and $N(N-1)$ times of complex addition. A complex multiplication needs four operations of real number multiplication and two operations of real number addition, computing $X(k)$ $(n \leq k \leq N-1)$ needs $4N^2$ times of real number multiplication and $2N(N-1)$ times of real number addition. When the number value of N is larger, for example, if it is 1024, you would need four million multiplications, which means real-time signal processing requires a high-speed processor.

But research has shown that the character of W_N^{nk} can be exploited to improve the operation efficiency of DFT. These include:

■ Periodicity of W_N^{nk} : $W_N^{nk} = W_N^{(n+N)k}$ (5.5)

■ Conjugate symmetry of W_N^{nk} : $W_N^{-nk} = (W_N^{nk}) = W_N^{n(N-k)}$ (5.6)

■ Condensability and expandability of W_N^{nk} : $W_N = W_{N/n}$ $W_N = W_{Nn}$ (5.7)

By taking advantage of above W_N^{nk} , properties and rearranging the order of $x(n)$ or (and) $X(k)$ and disassembling the sequence of $x(n)$ and (or) $X(k)$ into some segments, we can reduce the number of complex multiplications and enhance the operation speed of DFT, leading to the origin of FFT..

Radix-2 FFT Algorithm

If the length of sequence $x(n)$ equals $N = 2^M$, in which M is an integer (if M is not an integer, 0 is added to meet this requirement), by disassembling, the least DFT operation unit is 2-point. The least DFT operation unit in FFT operation is usually called radix, and hence, this algorithm is called radix-2 FFT algorithm of DFT.

$x(n)$ is first divided into two sub-sequences by N's odd number and even number:

$$\begin{cases} e(r) = x(2r) \\ f(r) = x(2r+1) \end{cases} \quad 0 \leq r \leq N-1 \tag{5.8}$$

DFT of N point is written as:

$$X(k) = \sum_{r=0}^{N/2-1} x(2r)W_N^{2rk} + \sum_{r=0}^{N/2-1} x(2r+1)W_N^{(2r+1)k}$$

$$= \sum_{r=0}^{N/2-1} x(2r)W_{N/2}^{rk} + W_N^k \sum_{r=0}^{N/2-1} x(2r+1)W_{N/2}^{rk}, \quad n \leq k \leq N-1 \tag{5.9}$$

According to the condensability and expandability of W_N^{nk} and $W_N^{2rk} = W_{N/2}^{rk}$, the formula is:

$$X(k) = \sum_{r=0}^{N/2-1} e(r)W_N^{2rk} + \sum_{r=0}^{N/2-1} f(r)W_N^{(2r+1)k}$$

$$= E(k) + W_N^k F(k) \tag{5.10}$$

in which,

$$\begin{cases} E(k) = \sum_{r=0}^{N/2-1} e(r)W_{N/2}^{rk} \\ F(k) = \sum_{r=0}^{N/2-1} f(r)W_{N/2}^{rk} \end{cases} \quad 0 \leq k \leq N/2-1 \tag{5.11}$$

$E(k)$ and $F(k)$ are the result of DFT of N/2, it is known from the character of DFT:

$$\begin{cases} E(k) = E(k + \dfrac{N}{2}) \\ F(k) = F(k + \dfrac{N}{2}) \end{cases}$$

Thus, this formula is written as follows:

$$\begin{cases} X(k) = E(k) + W_N^k F(k), \\ X(k + \dfrac{N}{2}) = E(k + \dfrac{N}{2}) + W_N^{(k + \frac{N}{2})} F(k + \dfrac{N}{2}) \quad 0 \leq k \leq N/2-1 \\ \qquad\qquad = E(k) - W_N^k F(k) \end{cases} \tag{5.12}$$

According to the formula above, as long as DFT $E(k)$ and $F(k)$ of two N/2 points is computed, the X (k) of all N points could be done by the linear combination of the formula (5.12). Due to $N = 2^M$ and $N/2 = 2^{M-1}$ being even numbers, the analysis may be continued until the last cell needing only two DFT points. As shown in Figure 20, the operation of the formula (5.12) is represented by signal streaming; the formula is called butterfly operation structure (butterfly operation) because the flow figure appears as a butterfly, also called twiddle factor.

Figure 20. Radix-2 Butterfly Cell
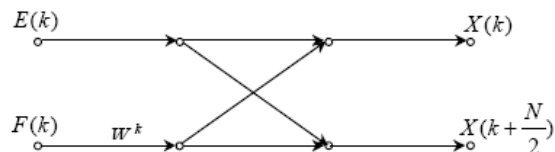
Some basic properties of radix-2 DIT FFT are derived according to the algorithm theory and twiddle factor above:

Resolve Series

From the analysis above, $N = 2^M$ is divided into M levels, of which every level contains N/2 butterfly operation, so the number of total butterfly operations is N/2 x M.

Operand Estimation

According to Figure 21, every butterfly operation needs one complex multiplication and two complex additions (subtractions), FFT of $N = 2^M$ point altogether needs $N/2 \times M$ of complex multiplications and $N \times M$ of complex additions (subtractions).

Radix-2 algorithm can reduce the arithmetic operation of DFT by half, which greatly increases computing speed.

Figure 21. 8-Point Radix-2 Algorithm Topology
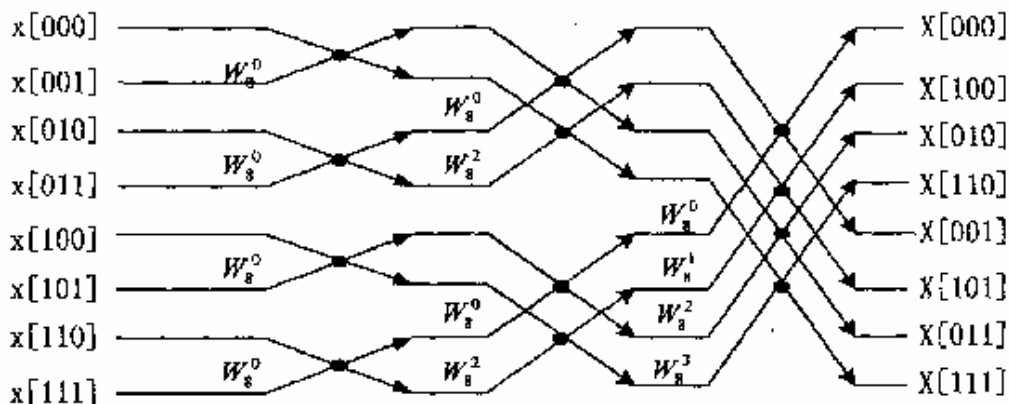
Radix-4 FFT Algorithm

Make $N = 4^M$ then:

$$X(k) = \sum_{l=0}^{3} W_N^{lk} \sum_{n=0}^{N/4-1} x(4n+l)W_{N/4}^{nk}$$

(5.13)

make respectively $k = 4r, k = 4r+2, k = 4r+1, k = 4r+3$,and r=0,1,…,4/N-1,According to the formula (5.13), as follows:

$$X(4r) = \sum_{n=0}^{\frac{N}{4}-1} [(x(n) + x(n+\frac{N}{2}) + (x(n+\frac{N}{4}) + x(n+3\frac{N}{4}))]W_{N/4}^{nr}$$

$$X(4r+2) = \sum_{n=0}^{\frac{N}{4}-1} [(x(n) + x(n+\frac{N}{2}) + (x(n+\frac{N}{4}) + x(n+3\frac{N}{4}))]W_N^{2n}W_{N/4}^{nr}$$

$$X(4r+1) = \sum_{n=0}^{\frac{N}{4}-1} [(x(n) + x(n+\frac{N}{2}) - j(x(n+\frac{N}{4}) - x(n+3\frac{N}{4}))]W_N^{n}W_{N/4}^{nr}$$

$$X(4r+3) = \sum_{n=0}^{\frac{N}{4}-1} [(x(n) + x(n+\frac{N}{2}) + j(x(n+\frac{N}{4}) - x(n+3\frac{N}{4}))]W_N^{3n}W_{N/4}^{nr}$$

(5.14)

Complex multiplications except for one imaginary number (j), may be not used in the basic cell of radix-4 algorithm. The series of FFT operations are decreased by half because of the algorithm of radix-4, so the number of multiplication required can also be relatively reduced.

Figure 22. Basic Cell of Radix-4 Algorithm
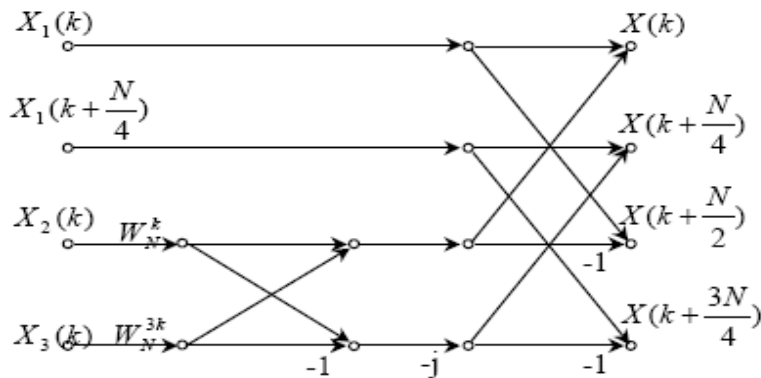
Splitting Algorithm

The basic principles of a splitting algorithm are to use radix-2 algorithm for an even sequence number output, radix-4 algorithm for an odd sequence number output. The Fourier transform algorithm is an

FFT algorithm, that has the least multiplication and addition times for all algorithms of $N = 2^M$ known.

The formula of splitting algorithm is as follows:

$$X(2r) = \sum_{n=0}^{\frac{N}{2}-1} [(x(n) + x(n + \frac{N}{2}))] W_{N/2}^{nr}, r = 0,1,\ldots,N/2-1$$

$$X(4r+1) = \sum_{n=0}^{\frac{N}{4}-1} [(x(n) + x(n + \frac{N}{2}) - j(x(n + \frac{N}{4}) - x(n + 3\frac{N}{4}))] W_N^n W_{N/4}^{nr}, r = 0,1,\ldots,N/4-1$$

$$X(4r+3) = \sum_{n=0}^{\frac{N}{4}-1} [(x(n) + x(n + \frac{N}{2}) + j(x(n + \frac{N}{4}) - x(n + 3\frac{N}{4}))] W_N^{3n} W_{N/4}^{nr}, r = 0,1,\ldots,N/4-1$$

(5.15)

Considering that the 512-point sampled data takes less system resources and the Nios II processor contains hardware multiplier, a simpler radix-2 algorithm is adapted in the system.]

FFT Cell Design

There are two schemes for the realization of FFT: hardware and software.

FFT Hardware Design

The operation cells of FFT are filled into sample-data RAM, Addwindow cell, dual-port DRAM of operation data, selector switch of multi-channel data, address generator, butterfly operation cell, twiddle factor ROM, result data FIFO, and control cell.

Figure 23. FFT Cell Architecture
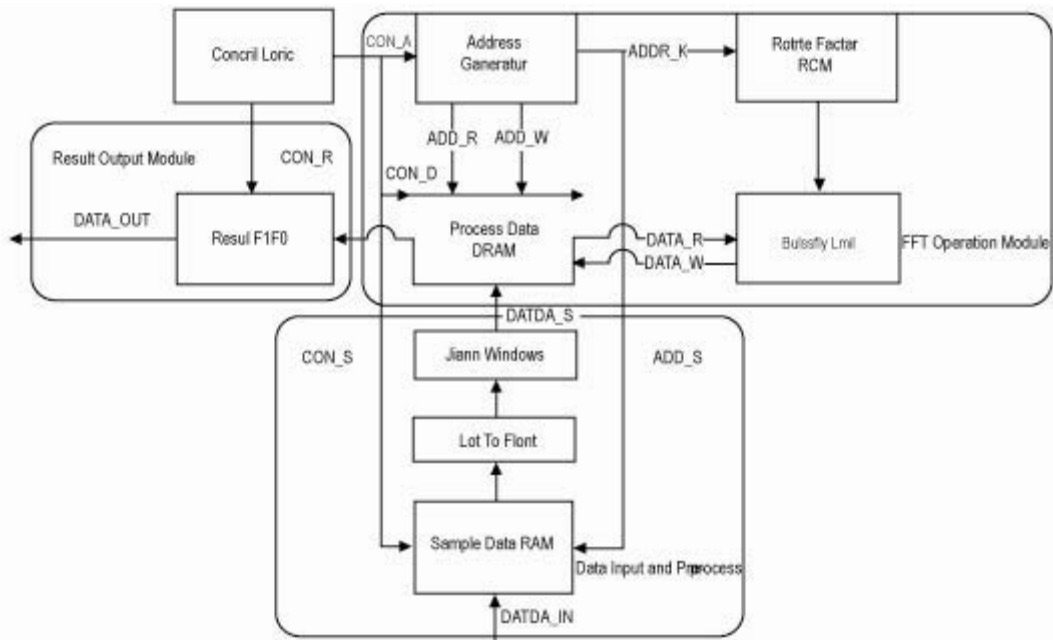
Butterfly Operation Cell Design

The butterfly operation cell is an important part of FFT operation cell, as it takes charge of performing radix-2 operation on input data, and then delivers data results. The structure is shown in Figure 24.

Figure 24. Architecture of Butterfly Operation Cell
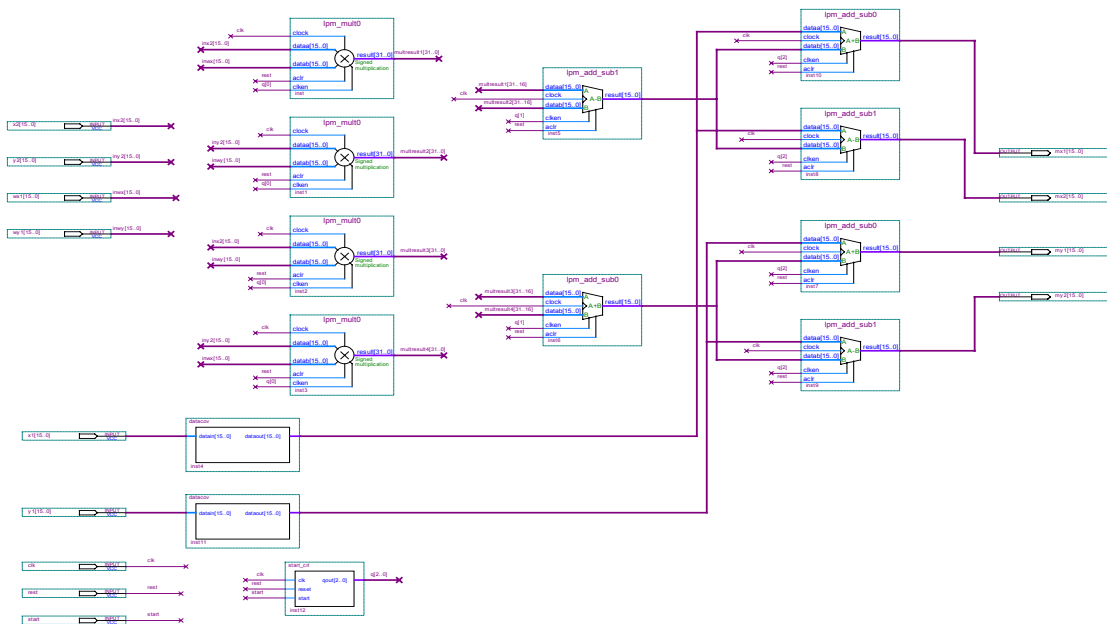
The design of butterfly operation cell completely depends on radix-2 operation, and butterfly operation cell is composed of four multipliers, three adders, and three subtractors.

Design of Other Cells

Sampled data RAM: Store collected data.

Window cell: Select different types of window functions.

Dual-port DRAM of operation data: Store the data during butterfly cell operation. Dual-port DRAM is used for handling complex data, and is matched with the real and imaginary parts of complex data, respectively.

Address generator: Generate sequential addresses, control data output of dual-port DRAM.

Twiddle factor ROM: Store twiddle-factor data which is the value of W_N^{nk}

Selector switch of multi-channel data: Due to butterfly operation, the data cell usually imports or exports two data values each time; where as RAM only reads and writes a data value once. The multi-channel transform switch can make data streaming more stable.

Result data FIFO: Stores data sent in by next cell.

Data Type & Length Selection

Data type directly affects the speed of operation, so it is necessary to adopt a compliant-data type.

We used 16-bit integer data type in the design, for the following reasons:

1. Our system's A/D converter handles 12-bit data, and the maximum value (A) it can export is not greater than 4096, and data (B) stored in twiddle factor ROM are the data magnified by 2^{16} times, the result of A multiplied by B is less than 2^{28} and greater than 2^{32}.

2. The data multiplied is reduces b 2^{16} at once to ensure that the operation of addition cell is correct. According to the principle of radix-2 algorithm, the last result should not be greater than $4069 \times 9 < 2^{16}$, and no overflow occurs.

Operation Flow

The A/D's sampling memory cell data are stored in dual-port DRAM. Computing starts when data storage is complete and a signal is sent. At first, address generator generates a set of addresses, and reads the data from DRAM and computes the radix-2 of FFT. A drive signal is generated while a compute cycle is finished, which enables the address generator to generate new address. When this FFT is completed, it signals the controller to read the data in DRAM to FIFO.

Operation Time

FFT is computed by a serial method in this system.

Giving due consideration to the stability of data streaming, the serial algorithm in the system needs seven periods for every butterfly operation, and the number of sampled data in the system is 512, resulting in a total period of 9×256×7 16128. To this you need to add the time for collection of data

cell writing to dual-port DRAM and the delivery of dual-port DRAM data to FIFO, which results in the system requiring a total of 16128+512×2=17152 periods.

Some Issues with Hardware

Designing an FFT using hardware would have put a strain on the systems' hardware resources because of the FPGA's limited capacity. In addition, it is quite possible that there would be hardware delays during the execution of FFT algorithm.

Method of Software Implementation

There are proven methods and different software tools available for the realization of FFT. Considering the independence we enjoyed during system design, we decided to showcase our creativity by writing the FFT algorithm of radix-2, instead of using the popular splitting algorithm.

Figure 25. FFT Software Programming Flow
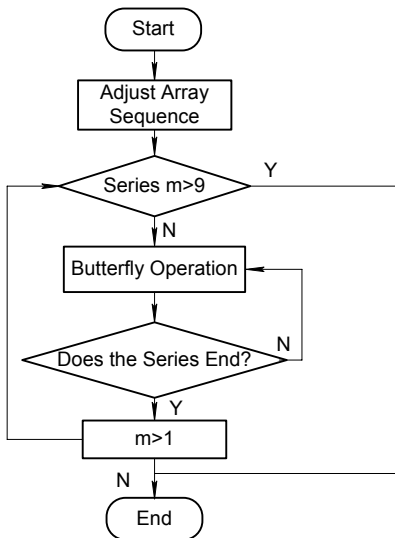
Primary program of butterfly cell on radix-2 algorithm:

```
for(m=0; m<M; m++)
{
    is=0; ie=id;
    do
    {
      n2=id;
    for (i=is; i<ie; i++)
        {
            k=(i-is)*t;
                xtr=xr[i+n2];  xti=xi[i+n2];
                xr[i+n2]=(xtr*wr[k]-xti*wi[k]);
             xi[i+n2]=(xtr*wi[k]+xti*wr[k]);

                xtr=xr[i];  xti=xi[i];
                xr[i]=xtr+xr[i+n2];
            xi[i]=xti+xi[i+n2];
                xr[i+n2]=xtr-xr[i+n2];
            xi[i+n2]=xti-xi[i+n2];
        }
        is=is+id*2;
        ie=is+n2;
```

263

```
        } while(is<N);
        id=id*2;
        t=t/2;
    }
```

By adapting hardware design to match the software design flow we were able to improve system performance. For example, by choosing parts that would execute slowly in software, we used hardware to speed up these areas, ensuring that the software adoption did not slow down the system performance. Our FFT cell designed using software is very stable and easy to manage and modify, and offers high controllability. So our decision to use software was vindicated in the design of the system. Although we could still use a series of optimizing algorithms involving hardware to speed up the FFT where we could process a 512-point FFT in less than 100 ms, it is a good enough performance because human eyes only distinguish a 10-frames per second images. Therefore, we have met the FFT processing algorithm's needs in real time application. We have optimized software design based on the aspects described in the following sections.

Custom Hardware Floating-Point Instruction Accelerating Key Algorithm

Although large numbers of multiplication operations are used in FFT operation, the system only uses 512 points as FFT. Therefore all parameters during FFT operation are defined as long integers without overflow, and this definition is controlled within acceptable number range. However, when we design more points of FFT or need to have higher definition for FFT, a float-point number range is necessary for FFT operation. Then, custom float-point multiplication instructions will greatly speed up FFT operations. After FFT is complete, the pattern value of every complex number is required to render a spectrum curve. In this case, floating-point number multiplication must be adopted for handling easy overflow of long integer variables because of extra index operations. The operation unit that computes one FFT with 512-point needs 1024 power operations, and hence a hardware floating-point multiplication instruction will greatly quicken the computing speed; tests show a hardware floating-point multiplication instruction can enhance 20% speed for the pattern value of FFT computing result.

The program for computing pattern value is as follows:

```
for(i=0;i<n/2;i++)
 {
   xr[i]=sqrt((float)xr[i]*(float)xr[i]+(float)xi[i]*(float)xi[i]);
 }
```

After using hardware floating-point multiplication instruction, it is as follows:

```
for(i=0;i<n/2;i++)
 {
   xr[i]=sqrt(ALT_CI_FP_MULT_CI(xr[i],xr[i])+ ALT_CI_FP_MULT_CI(xi[i],xi[i]));
 }
```

We decided to select 32-bit signed integer when selecting data types. The speed of software operation is obviously quicker during integer operations as against that of floating-point data. Using 32-bit integer data fully meets our requirement without overflow. Computing method matches that of hardware. Some tweaks have been added to the design because of the errors that crept in after adopting 32-bit integer data format: (1) Twiddle factor is magnified 1000 times while it is stored, and then it is reduced after computation to ensure the accuracy of next-level operation; (2) software is used for rounding while accepting or rejecting data.

Use system's on-chip hardware multiplier.

Optimize twiddle factor cell. Traditional FFT software algorithm always computes temporarily when it

$$W_N^{nk} = e^{-j\frac{2\pi}{N}nk} = \cos(\frac{2\pi}{N}nk) - j\sin(\frac{2\pi}{N}nk)$$

is used, but is comprised of sine (cosine) function,

and so software takes a lot of time. Thus, the importance of conserving memory in hardware design is applied to the system, that is, twiddle factor is computed early during initialization phase, and is stored in memory. When, it is used, it can be directly transferred according to memory address.
5) When the pattern value of the last result is computed, the 32-bit integer data is forced into floating-point data, to avoid overflow and ensure accuracy of next-level operation.

Design & Realization of AddWindow Processing Algorithm

Principle of AddWindow

Spectrum analysis is key to modern dynamic signal analysis including FFT and mean square spectrum analysis – power spectrum density (PSD). Following FFT, spectrum density is computed directly with signal FFT.

PSD on the basis of FFT can be computed according to the formula:

$$\hat{G}_{xx}(f_k) = \frac{2}{n_a T} \sum_{i=1}^{n_a} X_i(f_k) X_i^*(f_k)$$

$$\hat{G}_{xy}(f_k) = \frac{2}{n_a T} \sum_{i=1}^{n_a} Y_i(f_k) X_i^*(f_k)$$

$$(5.16)$$

in which n_a is the number of sample (average time), T is sampling period.

Because data processed by computer is discrete, the collected sampling signal is also discrete. Besides N, the values of other points are regarded as 0. In this case, leak occurs in the transformation process, i.e., the frequency component of one point is leaked to other frequencies. Window function can fix leakage of signal problems. Therefore, it is very significant to select the appropriate window function.

Leak indicates that the power of one narrowband in $\hat{S}(\omega)$ is expanded to adjacent frequency band, which makes $\hat{S}(\omega)$ lose strength. Leak is generated for the result of main lobe convolution of $S(\omega)$ and Window spectrum $W(\omega)$.

The main factor determining resolution is the length of used data or the length of data window.

$N > 2\pi k / BW$, of which BW is the distance of two spectrum-peaks in $S(\omega)$.

Common Window functions include rectangle window, trigonometric function, Hanning window, Hamming window, Kaiser window, Blackman window and flat top window. The common windows adopted by our dynamic signal analyzer system involves rectangle window, Hanning window, and flat top window.

1. Rectangle Window

$$w_R(n) = \begin{cases} 1, & n = 0,\, 1, \ldots,\, N-1 \quad \text{》ò } n = -\dfrac{N}{2}, \ldots -1,\, 0,\, 1, \ldots,\, \dfrac{N}{2} \\ 0, & \text{Æä Ëü} \end{cases}$$

(5.17)

2. Hanning Window

$$w_H(n) = 0.5 - 0.5\cos(\frac{2\pi n}{N}), \quad n = 0,\, 1,\, \ldots,\, N\text{ - }1$$

(5.18)

3. Flat Top Window

$$w_F(n) = w_R(n)[a_0 + 2\sum_{k=1}^{3} a_k \cos 2\pi k n], \quad n = 0, 1, \cdots, N\text{ - }1$$

(5.19)

Of which $a_0 = 0.99948, \quad a_1 = 0.95573, \quad a_2 = 0.53929 \quad a_3 = 0.091581$

Realization of AddWindow

In the system, two types of AddWindows are set up including rectangle window and Hanning window. For rectangle window, the 512 point FFT processing is done every time when it is sampled; for Hanning window, it can drive down sidelobe effectively, and directly perform window function weighing after time-domain signal is completed. In the program, the time-domain signal samples are made using AddWindow, which functions in the system as follows:

```
void winhanning(long int win[] )
{int i;
 for(i=0;i<512;i++)
 {
 win[i]=(long int)(win[i]*hcos[i]);
 }
}
```

of which hcos[] is coefficient array of AddWindow, and has completed computing when initialization, the program is as follows:

```
for (i=0; i<512; i++)
  {
  hcos[i]=0.5-0.5*cos(0.01227185*i);
}
```

The program structure does not compute cosine and multiplication every time when windowing which saves system time, and occupies less memory.

Design of Waveform Memory & Playback Program

For effective analysis and review of time-domain and frequency domain waveforms, we have designed the memory and playback function. Waveform and parameter data are stored in external flash, and we can realize store and read of data by hardware abstraction layer flash read/write interface function provided by HAL.

The main flash read/write functions are as follows:

```
        fd=alt_flash_open_dev(EXT_FLASH_NAME);
alt_read_flash(fd, offset, rdfft, length);
      alt_erase_flash_block(fd,offset,65536);
      alt_write_flash(fd, offset, wrfft, length);
```

The first function is used to make initialization operations before flash is read/written; the second function is to read data containing some length bytes, and place the read data to array rdfft[]; the third function is to erase flash block, where the relative blocks must be erased before it writes data to memory.

The following two arrays are defined to store time-domain waveform, frequency-spectrum curve and relative parameters:

```
  unsigned char fftm[64][512]
  long int wfv[128];
```

in which, first 64 bits of wfv[128] is used to store peak-value of time-domain waveform, later 64-bit data is used to store mainlobe frequency of frequency spectrum.

The program of flash storage operation is as follows:

```
 load_line_data(0x300010,fftm,32768);
 load_line_data(0x310000,wfv,512);
 for(i=0;i<256;i++)
 {
  fftm[pnum-1][i]=wave[i];
  fftm[pnum-1][i+256]=ffti[i];
  wfv[pnum-1]=everyw;
  wfv[pnum-1+256]=maxf;
 }                                      // read data from flash;
 write_line_data(0x300010,fftm,32768);
 write_line_data(0x310000,wfv,512);
```

The program of flash reading data operation is as follows:

```
  load_line_data(0x300010,fftm,32768);
   load_line_data(0x310000,wfv,512);
  for(i=0;i<256;i++)
   {
    wave[i]=fftm[pnum-1][i];
    ffti[i]=fftm[pnum-1][i+256];
    everyw=wfv[pnum-1];
    maxf=wfv[pnum-1+256];
    }
```

Partition of µC/OS Tasks & Their Design

Our system has five tasks, and the design of each of these tasks follows:

System Main

This task responds to keystroke commands and it is the most important part of system operation. With different key input, the program processes different states; when no keystroke is sensed, the task commands the A/D to sample and buffer the data in FIFO and compute FFT, and display the computed frequency spectrum on LCD. Data collection and spectrum display on LCD are the two important functions of the main task.

```
void maintask(void* pdata)
{
  while (1)
  {
    OS_ENTER_CRITICAL();//Close up all interruptions
    waitbuttonpress(edge_capture);
    OS_EXIT_CRITICAL();Open up all interruptions
    OSTimeDlyHMSM(0, 0, 3, 0);
  }
}
waitbuttonpress(edge_capture); this function responds to keystroke input.
```

Keyboard Scan Task

Scans keys, and when a key is pressed, the program assigns the captured key value to the global variable Edge_capture, which is used by the system main-task program. When we designed the Button_pio, we initialized the port to capture keystroke values as a falling-edge transition, with no interrupt option.

```
void keyscan(void* pdata)
{
  while (1)
  {
    edge_capture = IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE);
        OSTimeDlyHMSM(0, 0, 3, 0);
  }
}
```

Displaying Updated Tasks on LCD

During system initialization, a window is displayed on LCD and waveforms are displayed using the main task function. However, parameters of some process states (represented as bars) change continually following keystroke action during system operation. To handle this, we designed a display update task, which takes charge of updating these parameters of different states on the LCD bar; for the parameters of every state, we have defined the related global variables.

Here is a partial listing of update task program that is responsible for display update of peak-to-peak value of time-domain signal and main-lobe frequency of spectrum.

```
void lcdrefresh(void* pdata)//refresh the lcd;
{
  while (1)
  {
  GUI_SetBkColor(GUI_BLACK);
  GUI_ClearRect(78, 3, 225, 13);
  GUI_SetColor(GUI_WHITE);
  GUI_SetFont(&GUI_Font10_1);
  GUI_DispStringAt("vol:          mv", 58, 3);
  GUI_SetColor(GUI_RED);
  GUI_DispDecAt(everyw,80,3,4);
  GUI_SetColor(GUI_WHITE);
  GUI_DispStringAt("freq:          hz", 140, 3);
  GUI_SetColor(GUI_RED);
  GUI_DispDecAt(maxf,165,3,6);
    Omitting.......................................................
  OSTimeDlyHMSM(0, 0, 3, 0);
  }
}
```

A/D Collection & FIR Control Task

Controls sampling frequency of A/D converter and filter type of FIR according to key input. The routine changes sampling frequency and filter type by the received user parameters.

268

```
void ad_fir(void* pdata)
{
  while (1)
  {
  //Sample value delivered according to main task, changes the sampling frequency of
ADC.
    IOWR_ALTERA_AVALON_PIO_DATA(SELFRE_BASE, sam);
  //From value delivered according to main task, changes the sampling frequency of ADC.
    IOWR_ALTERA_AVALON_PIO_DATA(SELFIR_BASE, frem);
     OSTimeDlyHMSM(0, 0, 3, 0);
  }
}
```

Flash Memory Timing Task

During system initialization, all waveforms and parameters stored in flash memory are read into RAM memory. Thus, the system can save into flash memory each time while viewing waveforms during testing. You need to store data to middle array or you can directly read from middle array, and therefore the display of waveform is continuous. Nevertheless, the disadvantage with this method is that all stored data will be lost in case of system power-down. To avoid loss of data, the flash memory task continuously stores data to middle of array. The program is as follows:

```
  void saveflash(void* pdata)//refresh the lcd;
{
  while (1)
  {
  for(i=0;i<256;i++)
  {
   fftm[pnum-1][i]=wave[i];
   fftm[pnum-1][i+256]=ffti[i];
   wfv[pnum-1]=everyw;
   wfv[pnum-1+256]=maxf;
  }                                              // read data from flash;
  write_line_data(0x300010,fftm,32768);
  write_line_data(0x310000,wfv,512);
   Omitting.........................................................
  OSTimeDlyHMSM(0, 0, 3, 0);
  }
}
```

Design Features

This section describes the design features.

Implemented System-On-a-Chip with High Integration & Reliability

We were able to realize functions of the whole system (control and signal processing) on an FPGA, a result that is unparalleled when compared to traditional designs. As a 32-bit soft-core microprocessor with high performance, Nios II can be configured in an FPGA. Therefore, we can use it to implement a programmable system-on-a-chip function.

Custom Instruction Speeds Up Design Implementation

Because a great many floating-point multiplication operations are needed during the execution of FFT software algorithm and there is no hardware floating-point multiplication instruction in Nios II processor, we decided on a customized instruction. An excellent feature of the Nios II lies in the fact that you can design customized instructions. Our hardware floating-point multiplication instruction was designed with a general LE and added onto the instruction system. In addition, we defined a few other digital signal processing instructions. Using this design approach, we were able to significantly speed up the operation of digital signal-processing algorithms.

The digital filter was realized in hardware, which significantly speeded up digital signal processing.

Using the rich logic resources in the FPGA and based on a powerful development environment, we designed a digital hardware FIR filter with selectable high pass and low pass options. This filter speeded up digital signal processing.

Customization of Avalon Bus Interface IP LCD PWM Controller

An easy guide is provided in the SOPC Builder tool that helps engineers design IP cores based on the Avalon bus interface. Because the tool is integrated in software, we could easily design the interface driver program and added it onto the hardware abstraction layer, which makes system design easy. For instance, using the SOPC Builder tool, we could complete the design quickly even while adding several PWM controllers according to design requirements. This is one of the major benefits of an open bus interface.

Use of µC/OS II & µC/GUI

The powerful functionality and processing speed of Nios II processor, coupled with C-language support, made it convenient to migrate the µC/OS RTOS to the processor. Thanks to the Nios II IDE, we were able to develop applications easily and quickly. Based on the LCD control interface, we could migrate the µC/GUI to the system. Then, we made changes to software based on the GUI which resulted in a user-friendly system.

Soft Cores Made Interface Design Simple

Because Nios II is a configurable soft core processor, we could freely add the I/O interface according to design requirements. For example, we added several I/O interfaces for internal and external connection to/from the FPGA. Also, we adopted many peripherals in our design, such as an LCD controller interface, A/D controller, and FIR filter, which needed many I/O interfaces to communicate with the Nios II processor. Taking advantage of Nios II soft core, we could complete the design easily.

Conclusion

The design contest helped us to understand the following:

- A synergy of hardware/software in design is possible taking the Nios II design approach. For instance, we learned that customization instructions are a better method to accelerate key algorithms when realizing FFT with hardware or software design approach. Also, the algorithm flow could be easily controlled by software while resorting to hardware optimization where necessary. Traditionally, in system design you would design software first based on the hardware. In this design contest, for the first time, we could design hardware according to the software. For instance, we designed a customized hardware floating-point multiplier instruction according to the existing FFT algorithm. This is the first time we experienced the most interesting hardware/software synergy.

- Because some interfaces need a lot of customization, we needed to have a deep understanding of bus interface protocols, transport protocols, and peripheral interface. Previously we had worked on designs whose hardware was fixed. This contest deepened our understanding of the hardware layer.

- The differences between hardware and software design lie in SOPC design. We always need to design logic with HDLs and design software with C language. From the contest, we know more about the differences between hardware and software design.

■   We need more communication with other designers since SOPC technology is a very new and emerging technology. We have made many friends through the Nios II design contest, and in turn learned many things from them. In addition, the Nios II forum www.niosforum.com is always available for us to discuss problems with designers all over the world.

Appendix

Flow Summary

```
Flow Status              Successful - Wed Sep 14 15:00:39 2005
Quartus II Version       4.2 Build 157 12/07/2004 SJ Full Version
Revision Name            standard
Top-level Entity Name     standard
Family                   Stratix
Device                   EP1S10F780C6
Timing Models            Final
Met timing requirements  No
Total logic elements     5,208 / 10,570 ( 49 % )
Total pins               173 / 427 ( 40 % )
Total virtual pins       0
Total memory bits        577,280 / 920,448 ( 62 % )
DSP block 9-bit elements  8 / 48 ( 16 % )
Total PLLs               1 / 6 ( 16 % )
Total DLLs               0 / 2 ( 0 % )
```

Fitter Resource Usage Summary

	Resource	Usage
	Fitter Resource Usage Summary	
	Resource	Usage
1	Total logic elements	5,208 / 10,570 (49 %)
2	-- Combinational with no register	2822
3	-- Register only	598
4	-- Combinational with a register	1788
5		
6	Logic element usage by number of inputs	
7	-- 4 input functions	1917
8	-- 3 input functions	1705
9	-- 2 input functions	858
10	-- 1 input functions	382
11	-- 0 input functions	11
12		
13	Logic elements by mode	
14	-- arithmetic mode	1364
15	-- qfbk mode	456
16	-- register cascade mode	0
17	-- synchronous clear/load mode	1090
18	-- asynchronous clear/load mode	1683
19		
20	Total LABs	617 / 1,057 (58 %)
21	Logic elements in carry chains	1464
22	User inserted logic elements	0
23	Virtual pins	0
24	I/O pins	173 / 427 (40 %)
25	-- Clock pins	1 / 16 (6 %)
26	Global signals	14
27	M512s	2 / 94 (2 %)
28	M4Ks	15 / 60 (25 %)
29	M-RAMs	1 / 1 (100 %)
30	Total memory bits	577,280 / 920,448 (62 %)
31	Total RAM block bits	660,096 / 920,448 (71 %)
32	DSP block 9-bit elements	8 / 48 (16 %)
33	Global clocks	14 / 16 (87 %)
34	Regional clocks	0 / 16 (0 %)
35	Fast regional clocks	0 / 8 (0 %)

Third Prize

SOPC-Based Servo Control System for the XYZ Table

Institution: **Southern Taiwan University of Technology/Motor Engineering Research Institute**

Participants: **Dai Fuyu, Cai Xing'an, and Chen Jiasheng**

Instructor: **Ying-Shieh Kung**

Design Introduction

Electric power is mainly derived from a combination of an engine and a motor. The motor converts electrical power into mechanical energy, which is widely used in home appliances and industrial mechanical tools. When addressing specific applications, motor-driven tools generally need to make speed changes or have accurate positioning. In these applications, it is necessary to have a highly efficient servo-motor control to position the tools' movements accurately.

There are direct current (DC) and alternating current (AC) motors. DC motors used to be popular in the industry because of their simple controls—you only needed to control the armature voltage to vary the motor's speed. Because the motor's carbon brush and convector were mechanical components, they would produce sparks and cause damage when the motor was running, which was one major shortcoming of the DC motor. In addition, the DC motor posed a threat to the environment, had a short lifecycle, and was expensive to maintain.

AC motors can be classified into three different types: the synchronous motor, the induction motor, and the reluctance motor. The stator and rotor of an AC motor are the only contact bearing components. The spinning of the rotor is caused by the stator's magnetic field, and needs more complicated control technology (such as the magnetic field guide control) to implement different movements. With the development of semiconductor control devices, the computation required for AC motor control is easily met. Because of this advantage, AC motors are very popular today.

This project was created to study and design an integrated chip of multi-axis AC permanent magnet synchronous servo motor control system using system-on-a-programmable chip (SOPC) concepts using

an Altera® FPGA and the Nios® II embedded processor. We used the device to implement a three-axis XYZ table servo-control system, as shown in Figure 1.

The SOPC-based servo control system for XYZ table design mainly includes two modules: software and hardware. The software module is implemented using the Nios II processor, with programs handling communication between the control chip and PC, process control of three-axis XYZ table servo movement, and computation of movement tracking. Altera's FPGA implements the hardware module, which includes the functions to control the position of three motors on the three-axis XYZ table, a six-group PI controller algorithm computation, a three-group optical encoder signal-detection circuit, a three-group current-estimation circuit, a three-group vector-control coordinate conversion circuit, and a three-group space vector pulse width modulator (SVPWM) signal output.

Hardware digital circuits implement all position controllers of the three motors. We implemented the process control module in software because the computation is complicated and needs to be flexible. Further, the sampling frequency is not high. For instance, the moving track control is about 100 Hz. The position servo control of the three AC motors on the three-axis XYZ table must be implemented in hardware because the algorithm requires a faster execution speed (SVPWM frequency is 12 kHz, counter frequency is 3 - 4 MHz). Both the software and the hardware modules can run concurrently. In this way, you can improve the control feature of a three-axis XYZ table servo system. The three-axis XYZ table servo-control chip completed in this design offers digital control, improved system performance, and stability. The device also helps reduce the controller's size and cost. Figure 1 shows a block diagram.

Figure 1. Block Diagram of XYZ Table Servo Control System
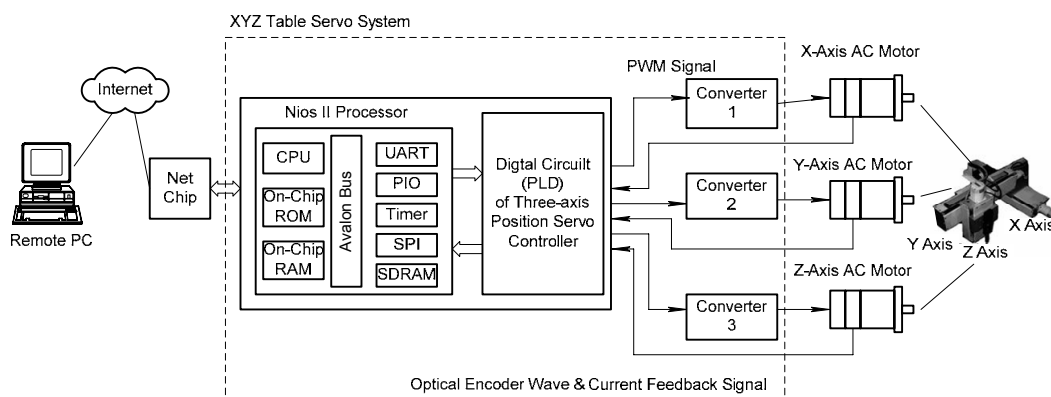
Our design can be applied in a variety of applications including CNC computer lathe processes, electrical-discharge machines, engraving machines, professional-drafting machines, mold and metal surface treatment, and high-tech semiconductor surface technology treatment.

We previously implemented the integrated design of AC servo system by using Texas Instruments' 243DSK development board and Altera's FPGA, which combined a microprocessor and FPGA architecture. As this design used a double-chip architecture, it needed an extra network control chip and linkage socket in case of additional network control. Unfortunately, this design approach resulted in increased system module size and costs. We combated these issues by using the Nios II development board. It is compact, economical, has a more stable control system design, and features a modular design. In addition, the powerful hardware circuits of Nios II development board offer functions such as a network control chip and extended memory. Therefore, we designed SOPC hardware with the Nios II development board and managed to complete the design much faster than the previous effort.

The SOPC Builder environment makes it easy to use the Nios II hardware development kit to create the required Nios II processor functionality, develop the complicated logic circuits, and easily modify the computing parameters. In addition, by using design partition techniques, we judiciously implemented a few fixed mode and high-speed logic operations in hardware. Then, using the Quartus® II integrated development environment (IDE), we quickly implemented and verified our hardware circuit design. Taking full advantage of the high performance Nios II embedded processor and the abundant logic resources of Stratix® FPGA, we were able to quickly and easily implement our design.

Function Description

This section provides a functional description of the system.

Integrated Function of Three-Axis XYZ Table System Chip

Figure 2 shows the inner structure of three-axis XYZ table system chip. There are two modules in this FPGA-based system design. The software module is implemented using software programs in the Nios II processor and includes functions that handle the communication between the control chip and PC, process control of three-axis XYZ motor movement, and movement-track computation. The hardware module is implemented in the FPGA, with functions comprising the position control of three-AC motors on three-axis XYZ table, including six-group PI controller algorithm computation, three-group QEP detection circuit, three-group current estimation circuit, and three-group SVPWM signal output.

The position controllers of the three motors are all implemented using hardware digital circuits. In this way, the controller chip can receive the PC's remote control commands and send out the position control angle of three-axis motors after its computation by the software module. After being processed by the hardware module's three-axis XYZ position loop control circuit calculation, this data is then sent to the PWM signal of each axis for a precise control of three-axis motor shift to target position. The FPGA design approach can minimize the controller size, and the Nios II processor makes the system design flexible. In this way, we can improve the three-axis XYZ table servo control performance and reduce costs.

Figure 2. Inner Structure of Three-Axis XYZ Table Servo System Integrated Chip
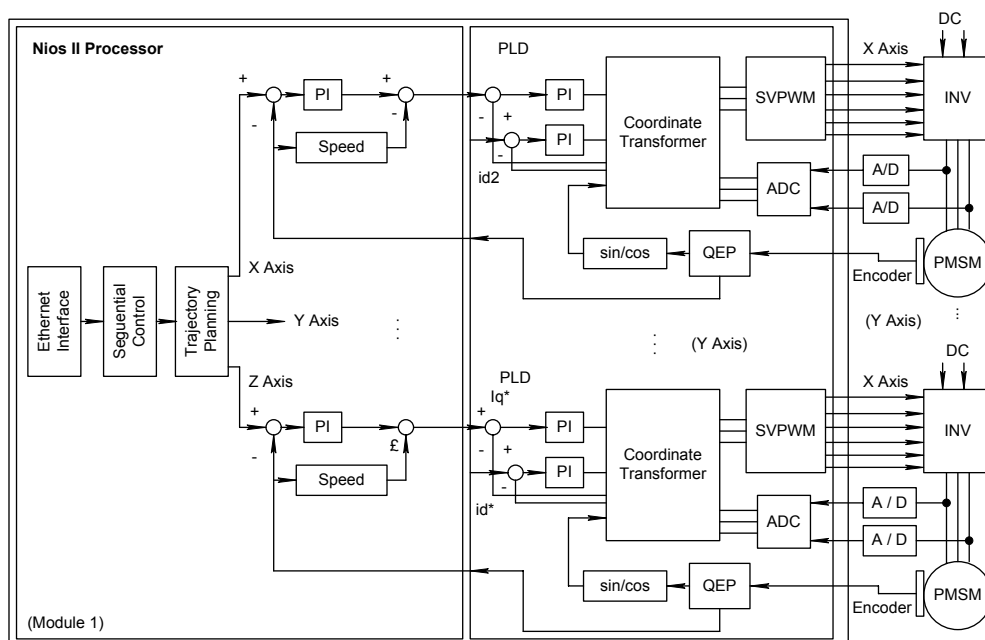
System Architecture Description

The system, Figure 1, consists of:

■ *Three-axis XYZ servo table*—This mechanism has three moving axis, and each axis is driven by a permanent magnet AC synchronous servo motor for linear movement, together with a ball screw. It features a maximum range of 300 mm. The power rating of the AC synchronous motor is 200 W, features Hall Sensor measurement of the magnetic pole position, and rotor or magnetic pole position measurement by incremental optical encoder (2,500 PPR); the transient current limit is about 10A (max). The rating speed is 2,500 rpm. The range of the ball screw is 5 mm/pitch.

■ *Converter of three-group AC motor*—As shown in Figure 1, each group driver individually drives the AC servomotor on XYZ table. The power crystal of the driver is based on Toshiba's IGBT, and we also used Toshiba's TLP250 Photo integrated circuit (IC). The converter receives the PWM signal sent by the FPGA to drive the AC motor.

■ *FPGA chip*—We used the Altera Stratix II EP2S60F672C5ES FPGA, featuring 24,176 ALM, 492 I/O, 36 DSP blocks, and a total of 2,544192 bits of on chip memory. We have also used one Nios II embedded soft core featuring 32-bit CPU, 16-Mbyte flash memory, 1-Mbyte SRAM, and 16-Mbyte SDRAM. These hardware resources can easily be adopted in the design of a three-axis XYZ table servo movement control chip.

Detailed System Description

This section describes the system in detail.

Mathematical Model of a Permanent Magnet Synchronous Motor

The equivalent electric circuit of the permanent magnet synchronous motor is shown in Figure 3, where the voltage equation's reference coordinates are settled on the synchronous rotation coordinates.

Figure 3. Motor Axis d & q Mode

(a) Axis d Mode

(b) Axis q Mode
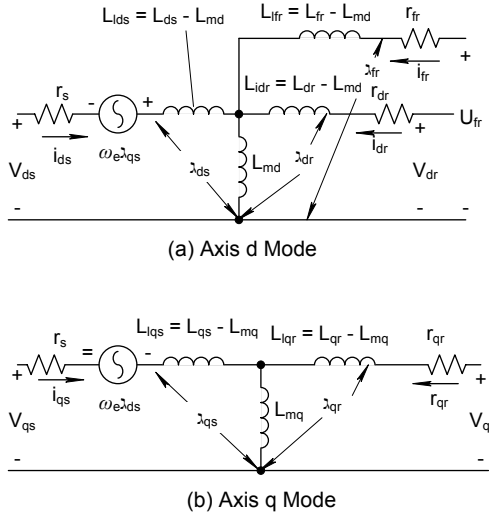
According to Figure 3, the axis d-q voltage can be shown as:

$$\begin{bmatrix} v_{qs} \\ v_{ds} \end{bmatrix} = \begin{bmatrix} r_s + sL_{qs} & \omega_e L_{ds} \\ -\omega_e L_{qs} & r_s + sL_{ds} \end{bmatrix} \begin{bmatrix} i_{qs} \\ i_{ds} \end{bmatrix} + \begin{bmatrix} \omega_e \lambda_f \\ 0 \end{bmatrix}$$

The voltage is calculated to:

$$\frac{d}{dt}\begin{bmatrix} i_{qs} \\ i_{ds} \end{bmatrix} = \begin{bmatrix} -\dfrac{r_s}{L_{qs}} & -\omega_e \dfrac{L_{ds}}{L_{qs}} \\ \omega_e \dfrac{L_{qs}}{L_{ds}} & -\dfrac{r_s}{L_{ds}} \end{bmatrix}\begin{bmatrix} i_{qs} \\ i_{ds} \end{bmatrix} + \begin{bmatrix} \dfrac{v_{qs} - \omega_e \lambda_f}{L_{qs}} \\ \dfrac{v_{ds}}{L_{ds}} \end{bmatrix}$$

The motor torque produced by axis d flux and axis q current is:

$$T_e = \frac{3}{2}\frac{P}{2}(\lambda_{ds}i_{qs} - \lambda_{qs}i_{ds})$$

Machinery dynamic equation obtained after loading is:

$$T_e - T_L = J_m \frac{d\omega_r}{dt} + B_m \omega_r$$

Combining the above three equations with the current controller of each axis, we can obtain the control block diagram during permanent magnet synchronous motor coupling, see Figure 4.

Figure 4. Control Block Diagram When Permanent Magnet Synchronous Motor Coupling
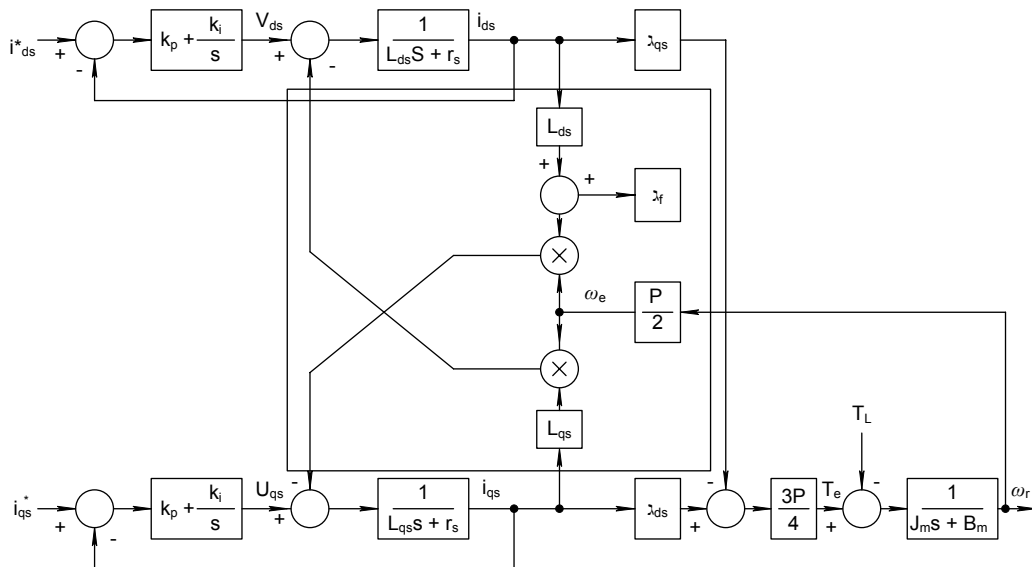

In this case, when the axis d current is set to 0, the motor torque or speed is only controlled by axis q current. This allows us to simplify the control block diagram as in Figure 5.

Figure 5. Diagram of Control Block
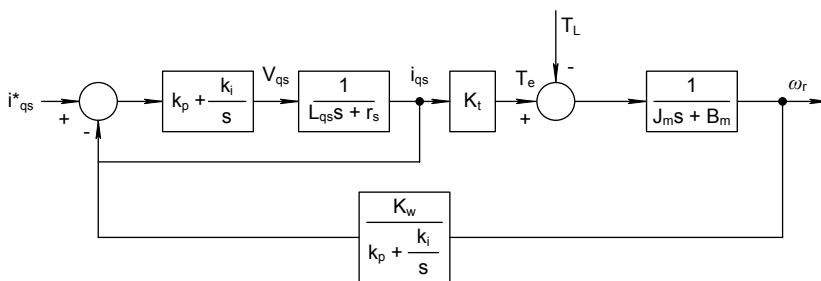
By the above permanent magnet synchronous motor control principle, the motor servo controller design is based on both the closed-circuit control and signal feedback by encoder of permanent magnet synchronous motor, making position control more accurate. Based on the above single-axis position control, we extended the system into a three-axis servo-moving controller (see Figure 6).

Figure 6. Three-Axis Servo Control Diagram
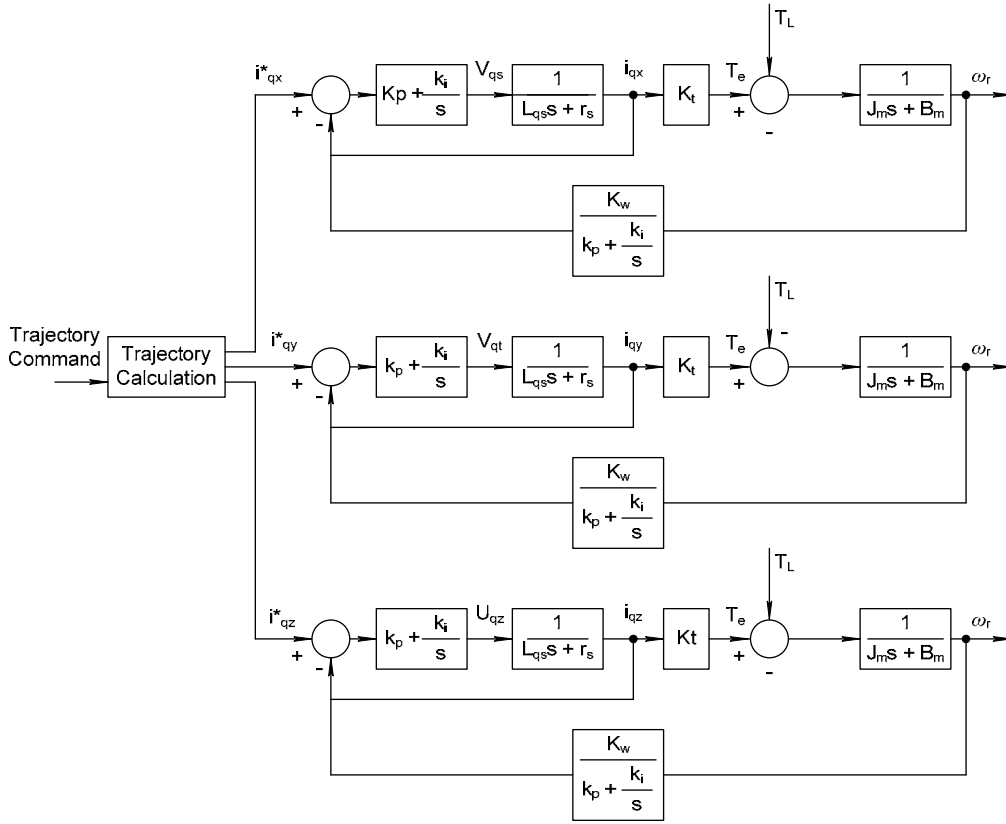
Point-to-Point Multi-Axis Track Planning & Design

For multi-position track planning, this design adopts a point-to-point track design. The point-to-point track design ignores the intermediate track and handles only the points of arrival and departure, to and from the destination. The design also takes into account the simultaneous departure and arrival, or featuring acceleration and deceleration functions upon departure and arrival. The input of point-to-point multi-axis design block diagram is equivalent to the rotation angle of each axis (θ_1, θ_2, θ_3), the maximum angular speed of each axis being (W_1, W_2, W_3), acceleration and deceleration times T_{acc} and sampling time t_d, while output is the position command θ_r^* of each axis. The design takes the following computational steps:

1. Calculate total executing time

   $T_1 = MAX(\theta_1/W_1, \theta_2/W_2, \theta_3/W_3)$

   Because total executing time should not be less than accelerating and decelerating time

   T_{acc}, therefore,

   $T = MAX(T_1, T_{acc})$

2. Revise total executing time as integral. Multiple of sampling time

    $N' = [T_{acc}/t_d]$ and $N = [T/t_d]$

    Among which, N is Interpolated point, and [ ] is Gauss function, therefore,

    $T' = N * t_d$ and $T'_{acc} = N' * t_d$

3. Revise speed

    Order $\vec{L} \underline{\underline{\Delta}} (\Delta\theta_1, \Delta\theta_2, \Delta\theta_3)$

    $\vec{W}' = \vec{L}/T'$

4. Calculate accelerating and decelerating value

    $\vec{A} \underline{\underline{\Delta}} (a_1, a_2, a_3) = \vec{W}'/T'_{acc}$

5. Calculate intermediate position command

    (1) Accelerating segment:

    $\vec{X}' = \vec{X}_0 + \frac{1}{2} * \vec{A} * t^2$

    Among which, t = n* td and 0<n N1

    (2) Even speed segment:

    $\vec{X}' = \vec{X}_1 + \vec{W} * t$

    Among which, t = n* td and 0<n N2

    (3) Decelerating segment:

    $\vec{X}' = \vec{X}_2 + (\vec{W}' * t - \frac{1}{2} * \vec{A} * t^2)$

    Among which, t = n* td and 0<n N3

    And among $\vec{X}' \underline{\underline{\Delta}} (\theta_{1r}^*, \theta_{2r}^*, \theta_{3r}^*)$, θ_{nr}^* is the position command value of the n-axis.

The design refers to Altera's Stratix II EP2S60F672C5ES, which is used in constructing three-axis XYZ table servo movement control chip. The chip includes three-axis XYZ position loop control circuit, SVPWM circuit, QEP estimation circuit, and current estimation circuit. In this way, the controller can receive optical encoder signals of three-axis motor simultaneously, compare them with command position, and then send out PWM signal for each axis after calculation by Nios II and FPGA to drive the power crystal for precise control of three-axis motor shift to target position. In general, the application

of the FPGA can not only minimize the controller size, but also improve the three-axis XYZ table servo control performance and reduce costs.

System Functions Accomplished by the Design

We used Visual Basic to develop three-axis XYZ table HCI movement and supervision software, and issue three-dimensional coordinates command through a remote PC. We were also able to set up the remote servo control mechanism through network transmission, and after track calculation by the Nios II processor we can obtain one point to another data in space, and rotation angle for each axis. The command for motor rotation position is obtained after track planning by position, computing by PI controller's position servo control, and the output of PWM command to calculate the PWM hardware circuit to drive motor driver. Then, after QEP hardware circuit computation, we were able to control motor to the required position. At this point, we sent back the sensor measurement for each axis position to Nios II CPU for conversion into three-dimensional coordinates, and transferred to HCI of remote PC through network, so the user can observe whether or not the motor has reached the required position.

Implementation of Hardware & Software Modules

The hardware and software modules are described as follows.

Hardware

Build a three-group power board, three-group motor driving circuit board, three-group analog-to-digital converter ADC converter circuit which includes the interface between three-axis motor driving board and FPGA chip, and power conversion circuit.

Software

First, we needed to write an HCI of PC that can transmit and receive through the network, and design a CPU core based on Altera's Stratix II EP2S60F672C5ES FPGA. This combination featured the necessary network transmission function and the required hardware interface for servo control, to implement a complete servo motor control chip. We developed the three-group SVPWM to produce the signal for driving AC motor circuit board, three-group QEP processor circuit to measure motor position. There are two functions involved in calculating the servo control position; the first function sets up network transmission/receiving mechanism, while the second, based on the required movement control equation for the practical use of three-axis XYZ table, writes the required response to movement. After resolving solution space vector, the space coordinates are sent back to the PC through the network to observe whether or not it has reached the required position.

Performance Parameters

For this experiment, we set the control interrupt frequency at 1 kHz in the Nios II processor, which is the most basic and important working frequency in the controller. The following is the measured interrupt time by Nios II CPU. In Figure 7, we can see that after several computing intensive steps, the Nios II CPU can precisely obtain the 1-kHz interrupt frequency required for the experiment, without any mismatch of control frequency. (1 stands for entering interrupt cycle, 0 stands for leaving interrupt cycle.)
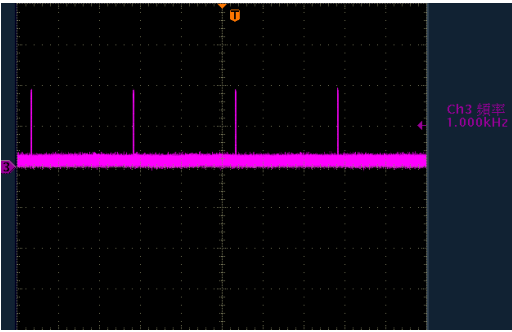
Figure 7. Obtaining the Frequency

Figure 8 is a point-to-point echelon speed-change track after Nios II CPU computation. The blue line stands for the command and the green line stands for the feedback signal. The working frequency is 100 Hz after the interrupt frequency is issued, subject to acceleration, uniform velocity, and deceleration.

Figure 8. Echelon Track Designing Position Command & Feedback
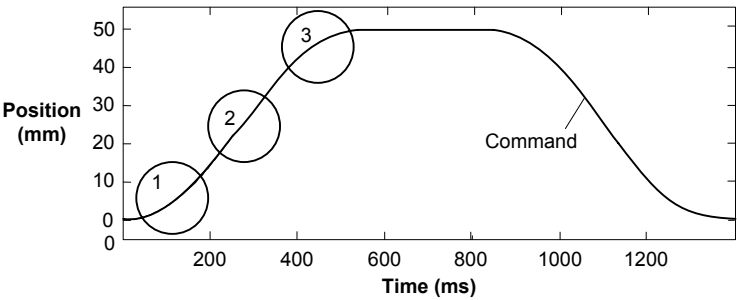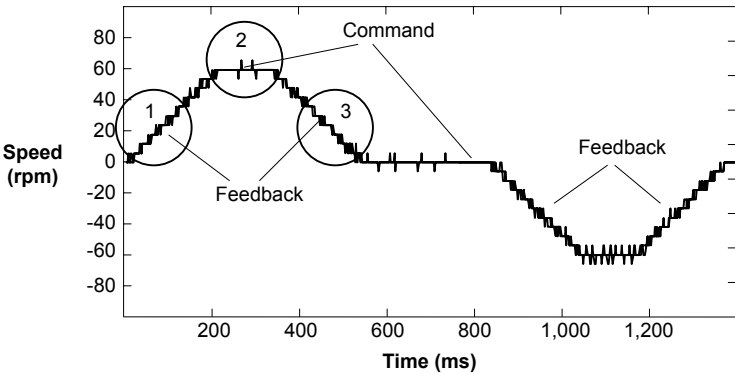

Figure 9 shows a designing position command with 5-cm radius circular track after calculations. The working frequency is 50 Hz after the interrupt frequency is removed.

Figure 9. Echelon Track Designing Speed & Feedback

Figures 10, 11, and 12 show the circular track position and its sine waveforms.

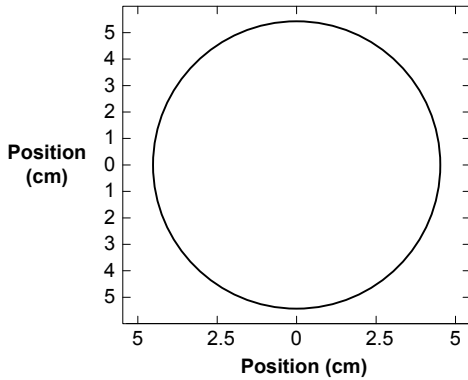*Figure 10. Circular Track Position Command*

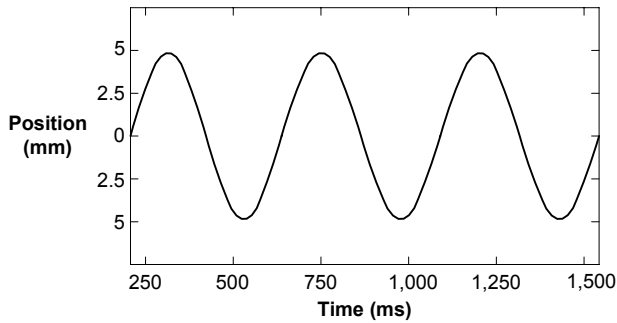*Figure 11. Sine Waveform (Command & Feedback) of Circular Track Position Command*

Figure 12. Sine Waveform (Position Error) of Circular Track Position Command
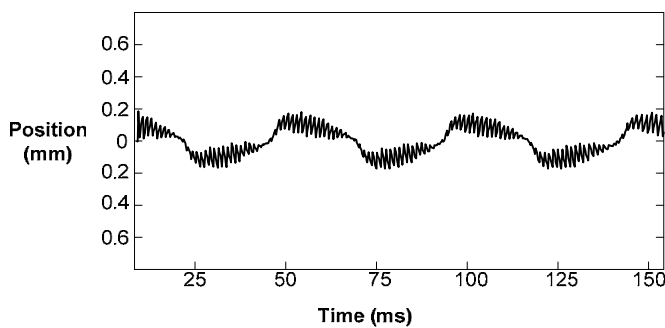
We can see from the above response identify that the response on the echelon track design is quite good, and the circular track position response can be seen from the sine waveform position error. The motor position control is also quite efficient.

Figure 13 shows the whole required track based on the computation of the CPU module's point-to-point track design, building the PI controller, speed, and position estimation module through the CPU. It draws Altera's logo using the XYZ table, with a working frequency set to 100 Hz, 50 Hz, and 25 Hz.

Figure 13. Required Track

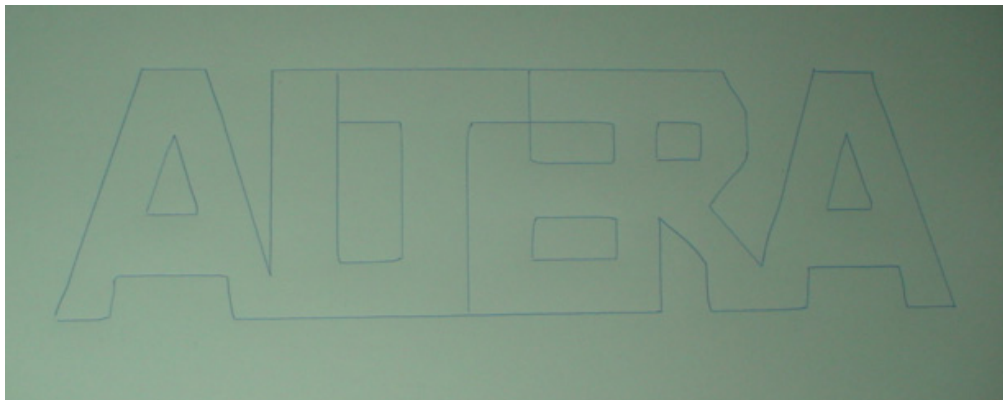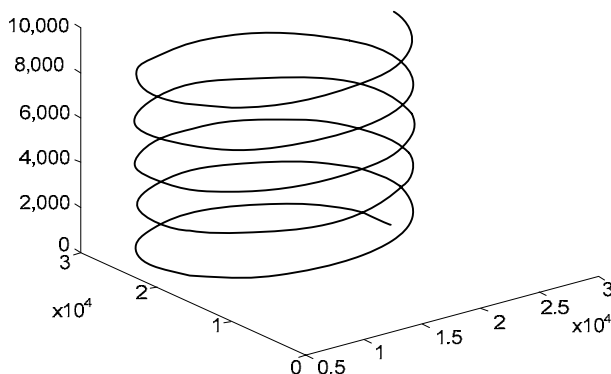Figure 14 shows the 3-D circular track position command after computation by the Nios II CPU with the working frequency set to 50 Hz.

Figure 14. Circular Track Position Command

In this experiment, the Nios II processor was of crucial importance. The Nios II processor provided powerful debugging during the design process, which is an inseparable key function for system design. While we used the Nios II processor mainly for the controller design, we also used it to design multi-group PI controller, speed and position estimating module, 3-D circular track, point-to-point track, echelon speed-change track, and one-group 1-kHz interrupt program. Also, after filtering out the interrupt frequency, we obtained the various requires working frequencies.

Design Architecture

The three-axis XYZ table servo control system based is shown in Figure 15, and its experimental system is shown in Figure 16. The system consists of the following parts:

■ *Three-axis XYZ table*—There are three moving axis, and each axis driven by permanent ac synchronous servo motor in linear movement together with a ball bearing guiding screw, as shown in Figure 15.

■ *FPGA development board*—This board is the system core. We used Altera's Stratix II EP2S60F672C5ES FPGA to develop a control chip of the three-axis XYZ table.

284

■ *Three-group ac motor converter*—The converter can receive the output PWM signal of control chip, and invert it into different voltages to control AC motor.

■ *PC*—Develop supervision software for the man-machine interface.

Figure 15. Three-Axis XYZ Servo Table
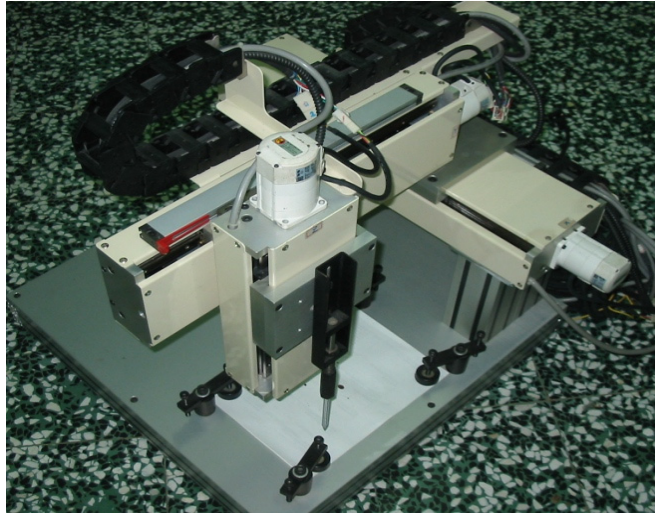
Figure 16. Three-Axis XYZ Table Servo Control Experimental System
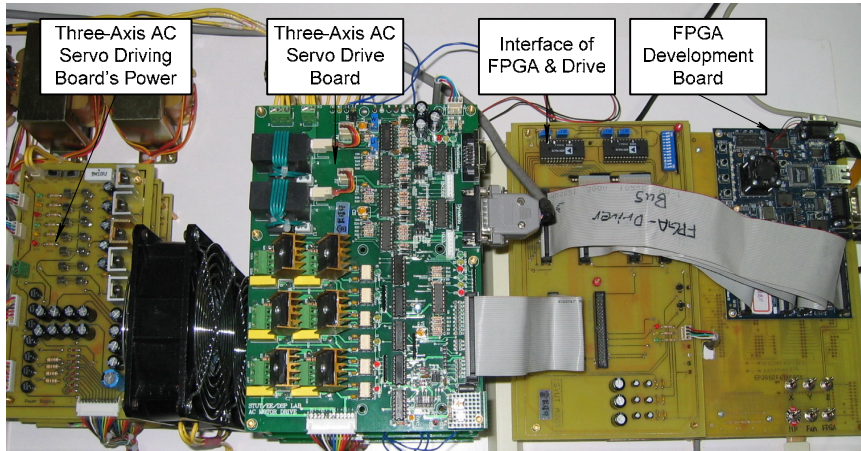
The inner hardware circuit of the three-axis XYZ table server system integrated chip is shown in Figure 17, which includes two modules. Module 1 is implemented in the Nios II processor by software, with functions including communication between the control chip and PC, process control of the three-axis XYZ table movement, and computation of the moving track. Module 2 is implemented in the FPGA, with the execution of three-axis servo controller for the table. The detailed circuit diagram of module 2 is shown in Figure 17, which includes six-group controller arithmetic, three-group QEP detecting circuit, three-group current estimating circuit, and three-group SVPWM signal output. Circuits are shown in Figures 17 and 18.

Figure 17. Inner Hardware Circuit of Three-Axis XYZ Table System Integrated Chip
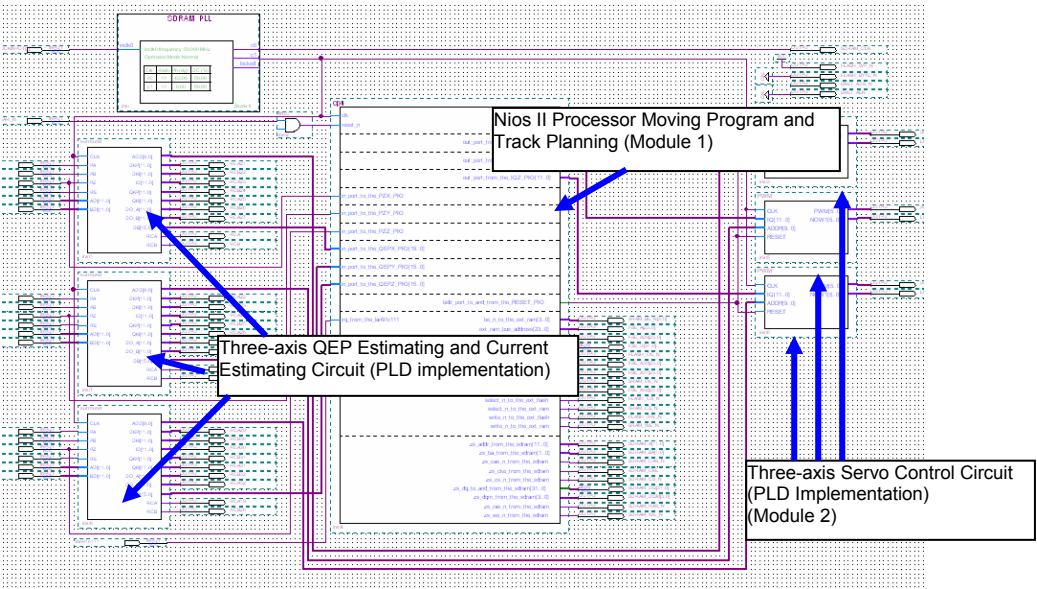
Figure 18. Three-Axis XYZ Table System Integrated Chip Inner Module 2—Three-Axis Servo Control Circuit
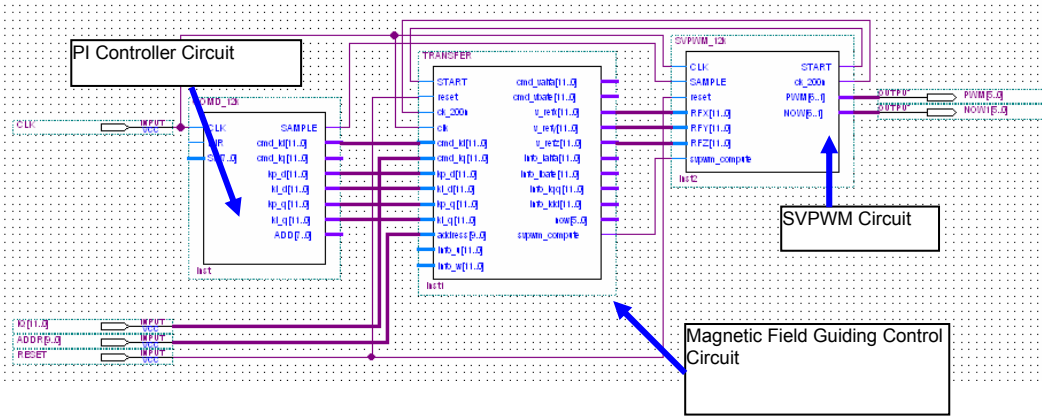
Figure 19 shows the proportion integral (PI) controller.

Figure 19. PI Controller Circuit Block Diagram
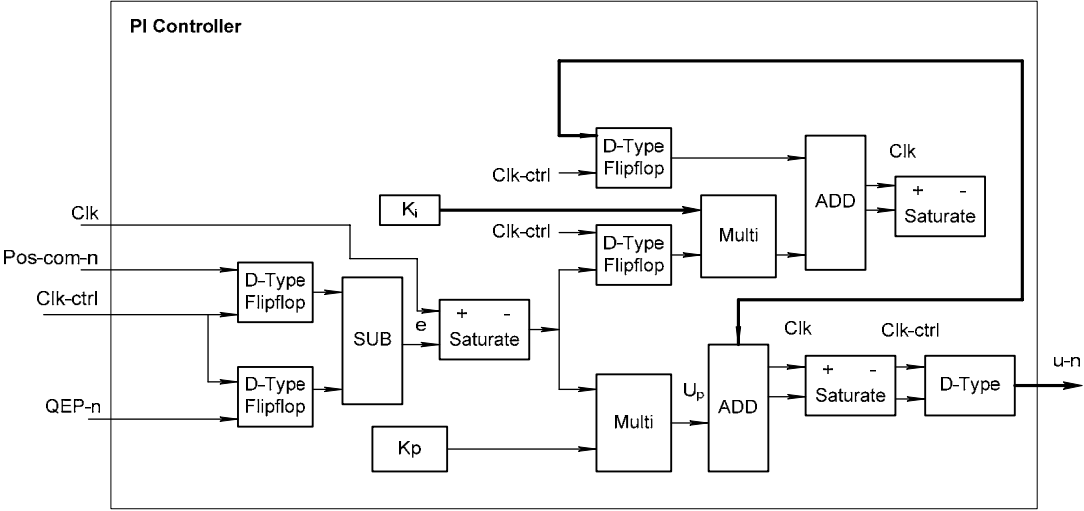
Figure 20 shows the QEP treatment circuit.

Figure 20. QEP Estimating Circuit Block Diagram
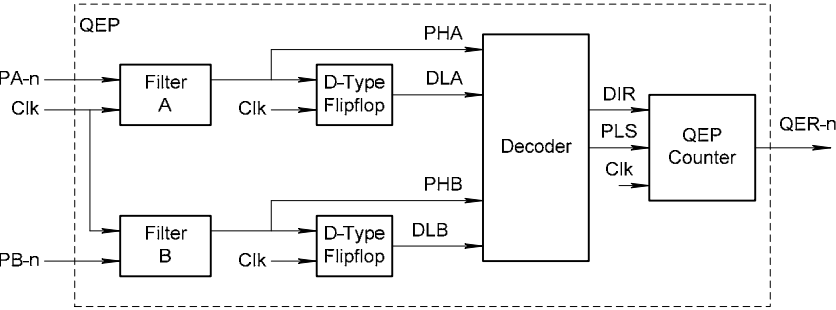
Figure 21 shows the SVPWM circuit.

Figure 21. SVPWM Circuit Block Diagram
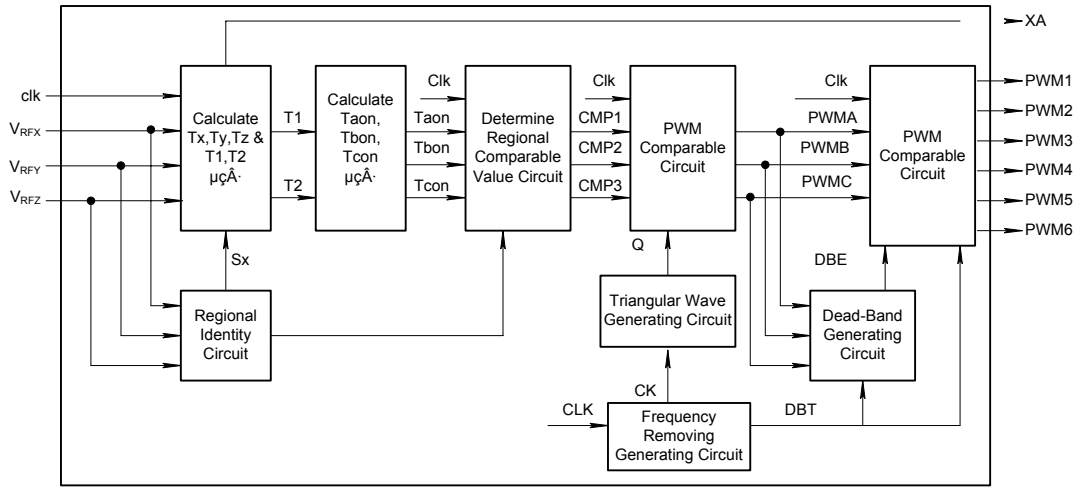
Figure 22 shows the software design flow chart for the control chip.

Figure 22. Nios II Three-Axis Servo Application Control Flow Chart
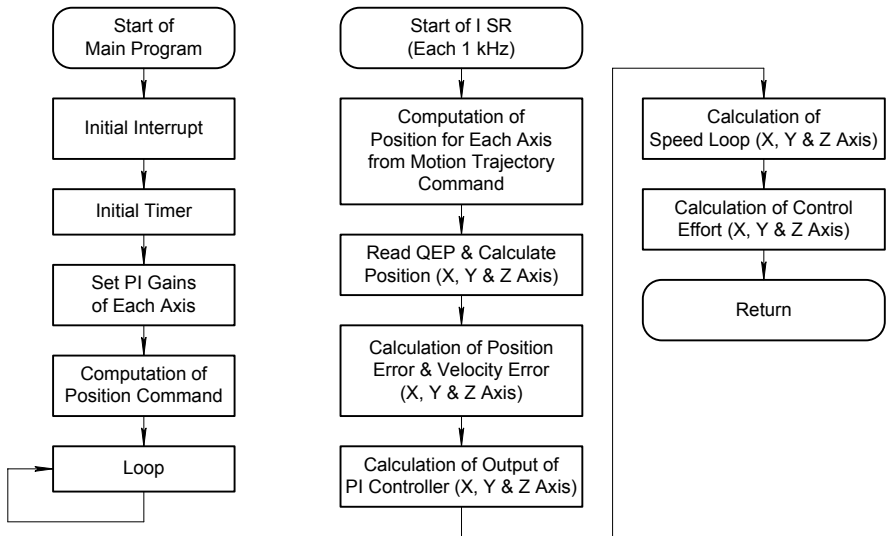
Design Methodology

The design methodology involved the following steps:

1. Develop the IP core of motor drive and design three-group AC servo motor drive using the VHDL language in the Quartus II integrated development environment (IDE), including magnetic field guiding control module, QEP estimation circuit, linear-optical scale position estimation, and current estimation circuit, ADC chip driver circuit, and SVPWM circuit.

2. Modify the Nios II processor, using SOPC Builder, to be the controller of the motor drive, design EMS memory unit flash and SDRAM, Avalon® tri-state bridge, system ID peripherals, JTAG UART, timer, and Avalon PIO. Next, design the reset address and exception address in flash memory and SDRAM, respectively. Finally, design the PLL for the CPU and SDRAM clock. Program the value of the motor drive obtained by the PIO using the Nios II processor, the output value of which is used for drawing waveforms in the MATLAB software for validation.

3. Design three-axis servo controlling system hardware which includes three-group power, three-group motor drive board, three-group motor required DC power, FPGA development board, and interface circuits for hardware including ADC circuit, linear optical scale signal receiving circuit, limit switch signal receiving circuit, and so on.

4. Design the motor controller and design one-group, 1-kHz interrupt program using the Nios II IDE. Develop many sub-programs: echelon track design, rotundity track design, PI controller, speed estimator, position estimator, and so on. First, calculate the required track by the main program, and then execute the PI controller, speed, and position estimator, moving the track program and the control program by a 1-kHz interrupt.

5. Integrate the three-axis servo control system, expand the original CPU to the Nios II processor to handle a three-axis servo control system, and combine the newly modified CPU with the three-axis motor drive and position estimating circuit. Next, output the previously written moving track program to the Nios II processor, whose motor drive circuit calculates the obtained command value and acquires the PWM signal. The signal is sent to the motor drive board via the FPGA development board and the necessary circuits to drive the AC Servo Motor.

6. Verify the program and check system whether the system is correct and efficient. Apply the previously designed program and control system on XYZ table to the Nios II processor-based system. Obtain the motor position and the speed feedback signal, output the command value compared with the feedback value, and verify and modify the controller's arithmetic parameter, until the controller's efficiency improves.

Design Features

We implemented our three-axis permanent AC synchronous servo motor control system design by using a complex arithmetic three-dimensional movement control. We converted the coordinates using the Nios II CPU and realized the remote control and monitoring through network mechanisms. While developing this project, we used the Nios II processor's powerful functions to quickly modify the system's computing parameters, verify its correctness, and were able to greatly reduce the design time.

Conclusion

This contest enabled us to develop a better understanding of the Nios II processor. By using the Nios II processor, we could easily design our system, which includes many embedded processors, on-chip and off-chip EMS memory, high-speed I/O ports, and network functions. The Altera development tools let

us develop our own multi-functional IC quickly. Additionally, we could modify the CPU hardware at any time for multi-purpose development using the SOPC Builder tool. As for the Nios II IDE part, we hope to use the Nios II debug function to shorten the software development time significantly. Altera's ability to develop and update the Nios II IP and functions was extremely important. For example, using custom instructions we can accelerate the hardware computation speed, which can improve our system's efficiency. We hope that Nios II users across the globe will exchange ideas more frequently for the purpose of improving future designs. This exchange will promote the advantages of the Nios II processor and help to boost the development of SOPC technology.

First Prize

Networking Remote-Controlled Moving Image Monitoring System

Institution: **National Chung Hsing University**

Participants: **Cai Jingtao**

Instructor: **Cai Qingchi**

Design Introduction

Our design focuses on establishing a real-time image monitoring system that can be operated under user commands to achieve real-time network monitoring. Users navigate a browser-like user interface in the manner of web browsing to remote control real-time image inspection. The application scope is security, home usage, and unmanned exploration. See Figure 1.

Figure 1. Motion Image Monitoring System
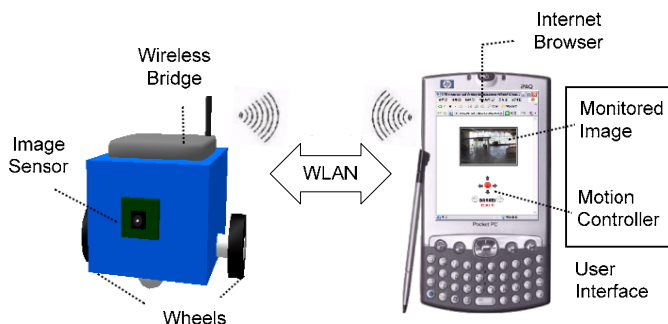
This design is different from fixed-monitor image inspection systems because it realizes integrated monitoring without any dead angle. For instance, parents can keep an eye on their kids using remote controls and a monitoring interface instead of using computer monitors at fixed locations in their homes.

The system design uses the Nios® II processor for implementing the basic peripheral components. It takes advantage of the powerful flexibility provided by the Nios II processor, as well as the support for real-time operating system (RTOS) to establish an integrated system. Additionally, the Cyclone™ FPGA supports low-cost performance by allowing hardware customization.

Function Description

The functional description of our design is as follows.

■ *Real-time image acquisition*—Connect the CMOS image sensor with the programmable I/O (PIO) interface circuit to get the value of picture points (gray scale) and then compose RGB images with a simple de-mosaic algorithm.

■ *Real-time image compression*—The algorithm compresses the image into JPEG format to improve the processing efficiency, and then uses hardware/software co-design to implement the Discrete Cosine Transform (DCT) algorithm in the JPEG compression system in hardware. Additionally, we designed a DCT algorithm as a custom peripheral meeting the requirements of the Avalon® bus standard. We used software to implement other algorithms.

■ *Graphical user interface (GUI)*—The design uses a web server as a simple GUI in a web-browser format. The GUI monitors picture display and control interface. The common gateway interface (CGI) programs handle control and communication.

■ *Motion control*—The mobile part of the design integrates the driver's integrated circuit (IC) and its circuitry with two DC motors on a custom-designed circuit board. The board also supports a pulse width modulator (PWM) component on the Nios II processor to control the motor through the PWM-signal output.

■ *Wireless network transmission*—Because the Cyclone FPGA does not provide a wireless network interface, our design uses the available interface with a wireless bridge to accomplish wireless-network transmission.

■ *System integration*—The system's motherboard is based on the Nios II development board and the Cyclone FPGA. We designed a circuit board to integrate the drive circuit and power supply. Further, we used a lead-acid battery as the system's power supply to meet embedded system application requirements.

Performance Parameters

The major performance parameters of the design depend on the frame rate of the JPEG encoder. The process of composing a frame includes:

1. Image acquisition: the PIO controls the CMOS image sensor to capture a 320 x 240 gray-level image.

2. Image processing: realize de-mosaic motion in an easy way to synthesize RGB images.

3. Image compression: carried out in accordance with the encoder output, in standard JPEG format.

In the JPEG standard baseline mode, our design uses hardware to implement the DCT algorithm. Software programs fill the other related performance functions. We implemented the JPEG compression in software by migrating the IJG Library. Then, the hardware/software co-design of the JPEG encoder is

complete. Finally, the compressed data is delivered to the remote user through the network transmission by reading and displaying the received JPEG images through a browser interface.

The following table shows the image of QVGA resolution (320 x 240) processed by our system as well as the time spent in various processes.

Process	Image Fetching	De-Mosaic Algorithm	JPEG Encoder
Time	0.2 sec	0.11 sec	0.83 sec

For testing purposes, the design uses the Cyclone FPGA with an operating frequency of 50 MHz and 0.87 FPS efficiency.

Because the DCT operations are time consuming, the DCT is more easily created in hardware when using the JPEG-compiled code system. However, planar DCT can be realized with current high-speed algorithms such as the IJG Library that provides fixed-point and floating-point DCT functions. However, the DCT is still more compute-intensive than the JPEG operation.

The following table shows time spent on processing one block (8 x 8 pixels) per frequency period. Software refers to the fixed-point fast DCT function provided by the IJG library and hardware is the DCT component that is executed through the Avalon bus interface.

Note: Data processing includes execution time of a single measurement function and accounts for data transfer into memory.

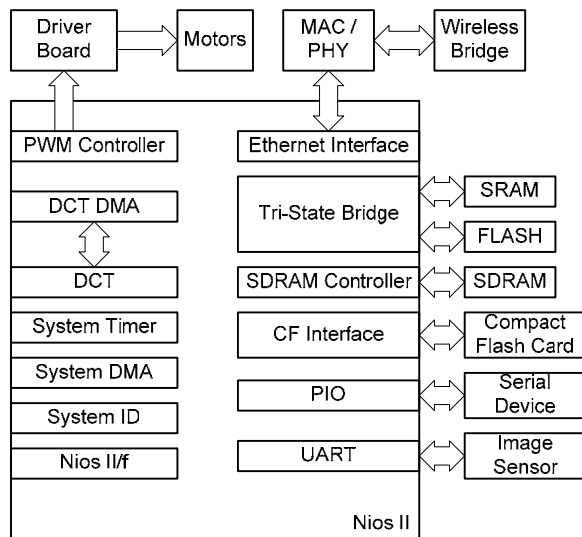| Method	Software	Hardware
Time	4,000 clocks	450 clocks

Using the table above, the design successfully implements the JPEG image compression system on the Nios II processor using hardware/software co-design, freeing up system resources during DCT computation.

Design Architecture

Figure 2 shows the system architecture.

Figure 2. System Architecture

The main system is based on the Nios II Development Board, Cyclone Edition, and we used the peripherals of the development board in this design. The system comprises the following functions:

■ It adopts Nios II/f (fast) core and adds hardware multiplier/divider/shifter selectively.

■ It stores the hardware and software programming files into flash.

■ It adopts SDRAM as memory storage for bigger programs.

■ Storage devices of the system use a compact flash (CF) card and two SRAM devices.

■ UART interface is used as system-console interface.

■ It applies a PHY/MAC chip of the development board to implement wireless-network transmission, connecting with the wireless bridge through an RJ-45 network interface.

■ For image acquisition, the system connects to an external CMOS image sensor and uses the PIO to realize a picture-element read interface.

■ It uses hardware to implement the DCT, and we designed custom peripherals to work with the Avalon bus and added DMA for data read/write.

■ In the motion mechanism, we added a PWM controller to output the PWM signal to the IC motor drive and implement the normal/reverse control for the DC motor.

Figure 3 shows the software flow chart.

Figure 3. Software Flow Chart
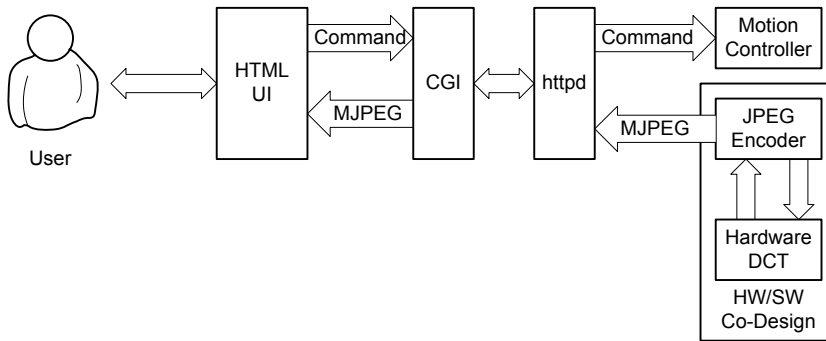
A major feature of the design is its simple user interface. The remote user merely needs to connect to the self-defined IP address of the system using the network browser. The web server then provides service for the user, including JavaScript programs to handle real-time image monitoring of pictures in motion JPEG mode.

Additionally, the mobile control interface can execute forward, backward, and stop motions easily. The software design and development is based on the Microtronix Nios II Linux distribution version 1.3 RTOS, using a CF card to qualify the system as a read/write file.

The communication between web control interfaces is created with CGI programs. When the user executes an image-monitoring function, the system acquires an image immediately, performs JPEG compression with the combination of software and hardware, and delivers the images to the web. At the same time, the user can control the image mobility and adjust the brightness to achieve real-time and interactive motion-monitoring function. Figure 4 shows the GUI.

Figure 4. GUI
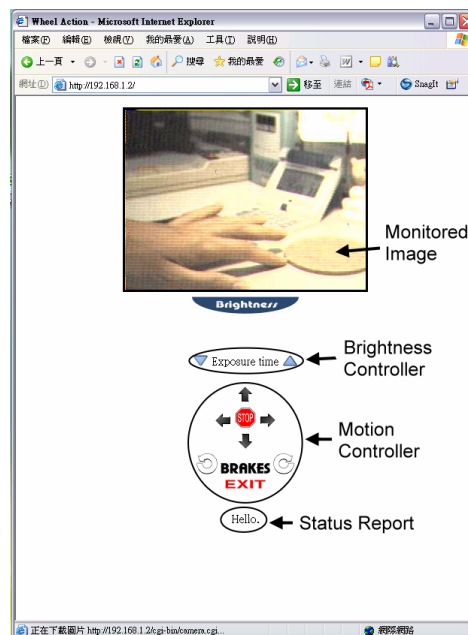
Design Methodology

Figure 5 shows the detailed design steps, various system tasks, and their inter-working:

Figure 5. Design Steps Flowchart
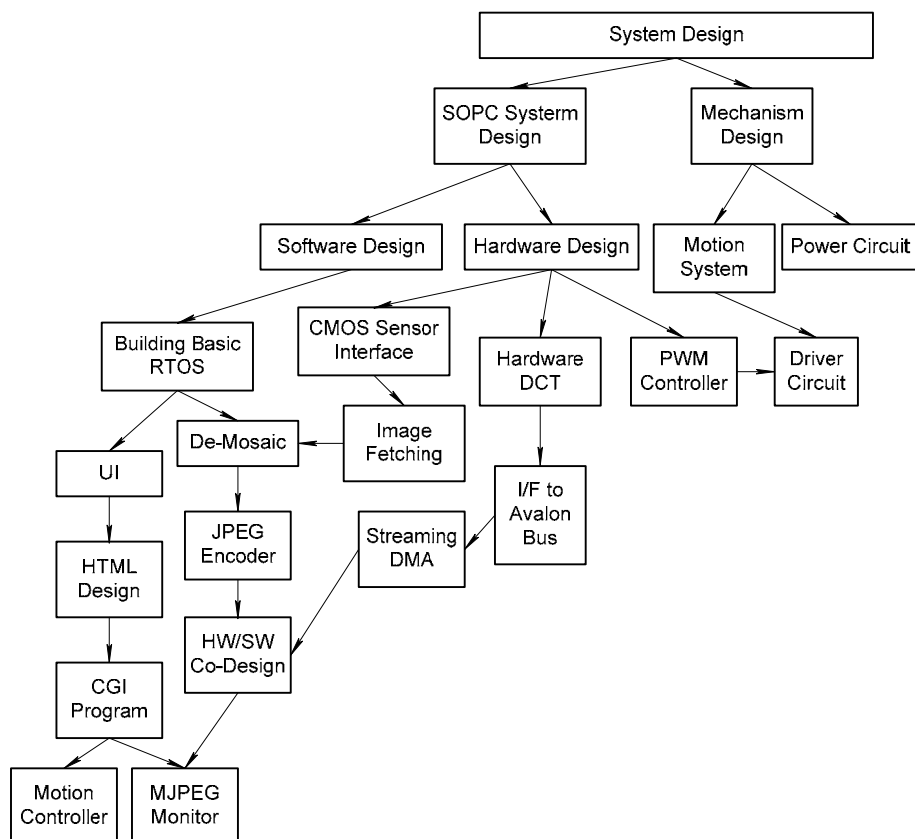
The flowchart is based on a top-down integrated system design concept, which is divided into two parts: system design and architecture design.

■ System-on-a-Programmable-Chip (SOPC) System Design:

> The hardware design includes configuring the basic Nios II SOPC environment and customizing peripherals that are compatible with the Avalon bus. These peripherals include CMOS image interface circuit, hardware DCT, and PWM controller.

> The software module comprises RTOS tasks including the file system routines, the GUI design, and various system tasks. Refer to Figure 5 for more details.

■ Architecture Design:

> The main mobile device comprises a chassis with two Maxon DC motors and wheels to handle movements such as forward, backward, and circumrotation (in the manner of differential rotation).

296

A motor driver and a power circuit comprises driver and optocoupler devices. We designed a board that integrates all IC and driver circuits.

The system's power supply uses two 12-V lead-acid secondary batteries. This circuit features a simple filtering-protection circuit, switch, and connection.

■ System Integration:

Figure 6 shows a fully integrated system.

Figure 6. Fully Integrated System
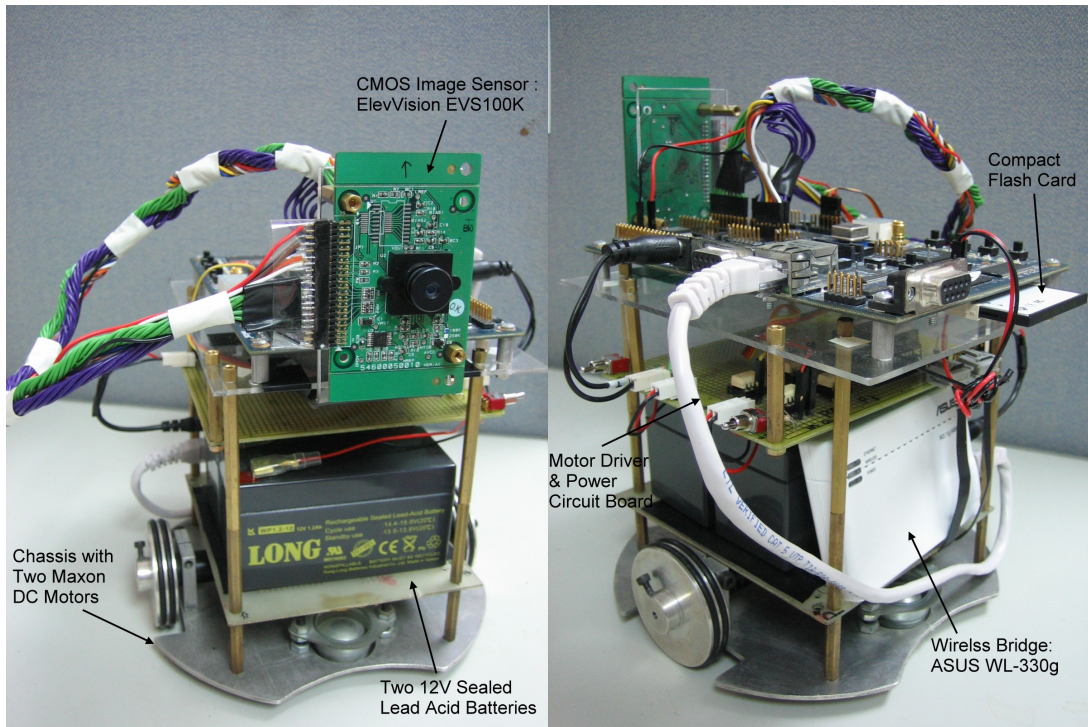
Front View   **Back View**

Design Features

Using the Nios II processor, you can simplify the customized SOPC design concept and verify your designs realistically. Furthermore, FPGA resources can be applied creatively to implement various functions depending on user's special requirements. These requirements could include functions such as hardware DCT and a PWM controller. In this way, the designer can integrate most functions into one FPGA.

The core module of the design is the JPEG encoder, which we implemented through hardware/software co-design. This design also showcases the FPGA advantage at its best. We created the DCT in hardware to realize the JPEG encoder and designed a custom interface that followed Avalon bus requirements. Next, we wrote the DCT output data in memory for processing in software with streaming DMA to save operation time.

The general features of the system design are described in the following three points:

Technical Standards

We have successfully used the Nios II processor to fulfill the IP camera function, thus meeting market needs. Additionally, we have added the remote control mobile function for extra benefit.

Our system boasts a highly integrated design and features an easy-to-use GUI and RTOS environment.

We used nearly all peripherals of the Nios II development board. We integrated most functions into the low-cost Cyclone EP1C20 FPGA and the logic element (LE) utilization exceeds 90%.

Design Creativity

Our system integrates a mobile mechanism that can be controlled easily using existing fixed monitoring systems available in the market to produce an innovative design. This system is a balanced product between consumer electronics and a robot.

Adaptability

The no-dead-angle remote monitoring function in this design is flexible and can be used to handle many applications. Potential users include home, factory, and commercial establishments.

Our design's modular architecture is similar to the common IP camera. We implemented our system with the low-cost Cyclone FPGA. Moreover, the mobile system attachment is very economical and can be deployed in consumer applications requiring no-dead-angle remote control. Additionally, the system does not require expensive interface devices, and all its functions can be realized using semiconductor ICs, which can be mass-produced easily.

The outstanding features of the system design:

- *Easy-to-use GUI*—It can monitor the situation on-site through the web.

- *Mobility*—Control the car with a remote network and extend the monitoring scope of the original fixed-monitoring system to any location.

- *Utilize Nios II and FPGA features to implement the functionality, keeping the total cost lower than many competing solutions*—Meet the IP Camera functions based on market expectations with the low cost Nios II processor and Cyclone FPGA.

- *Design uses RTOS*—Adopt Microtronix Nios II Linux Distribution v1.3 as the OS system.

- *Custom peripheral for hardware acceleration*—Added a DCT custom peripheral to accelerate the JPEG encoder.

- *Two custom peripherals without hardware acceleration*— Added Altera's PWM controller for handling motor control functions. Added Microtronix's Compact Flash Component to access the CF card data.

- *Performs graphic acceleration/uses the DSP algorithm*—Through hardware/software co-design, managed to increase the frame rate of motion JPEG.

- *Used DMA as another master device on Avalon bus*—Acquired DCT-output data with streaming DMA.

■ Greater than 90% LE/memory utilization.

■ Connected over 15 main peripherals on Avalon bus.

Conclusion

When we used the Nios II processor to implement our design, we were very impressed with the convenience it provided. Our design benefited greatly from its features. For example:

■ *Complete and coherent development environment*—The SOPC design process, software design, and verification were made easy by using the Nios II processor, thanks to Altera's convenient development tools.

■ *Abundant and flexible peripheral support*—Peripherals can be fully integrated and peripheral components customized by designers easily and quickly.

■ *Quick verification*—Because the system architecture is based on digital logic design, we were able to get clear and detailed information by simulation software during the system verification.

Altera's Nios II processor platform using FPGAs can be adapted for teaching, experimenting, and quick product development in many diverse applications.

Third Prize

Embedded Electric Power Network Monitoring System

Institution: **Jiangsu University**

Participants: **Xu Leijun, Guo Wenbin, and Sun Zhiquan**

Instructor: **Zhao Buhui**

Design Introduction

Electric power is the mainstay of a nation's economy and the lifeline of industrial production and social life. All parameters of an electric grid, especially harmonic parameters, are related to the quality of electricity generated, which guarantees safe operation of electric equipment. Therefore, this design combines a web-based electric grid parameter measurement system with a video-monitoring system to arrive at an integrated monitoring system of embedded electric grid/station. In the measurement of electric grid parameters, we check the voltage, current, harmonics, etc. in the target areas. The video section monitors the key instruments and environment and transfers these results to a web interface in real time. Therefore, no matter where the monitoring personnel are, they can observe the electric grid parameters and inspect both equipment and environment in real time, as long as they are connected to the Internet. Our design applies to electric stations at all levels, enterprises with requirements of a reliable network, as well as applications requiring remote monitoring.

The Altera® system-on-a-programmable-chip (SOPC) solution is a flexible, efficient solution, integrating the function modules necessary to system design such as CPU, memory, and I/O interface into an FPGA. The SOPC design approach makes for a flexible system design allowing you to modify, expand, and upgrade system modules using Altera-supplied software and hardware tools. We chose the Nios® II soft core embedded processor because of its low cost and abundant FPGA logic resources, which can cater to the demands of different applications. In addition, Altera provides a complete solution based on the Nios II processor, which includes the Quartus® II and integrated development environment (IDE) tools, further reducing product development cost.

Function Description

The system design consists of two parts: the measurement and delivery of electric grid parameters and video monitoring**.**

Measurement & Delivery of Electric Grid Parameters

Our measurement circuit uses an application-specific DSP chip to collect, store, and send the following data:

■   A measurement of valid values

■   The harmonic component of three-phase voltage/current and neutral current

■   The voltage imbalance factor

The DSP chip also measures real power, reactive power, power factor, power supply frequency, and power cut times, calibrated against the national standard precision.

During measurement, the three-phase voltage and current circuitry was isolated using voltage and current sensors for data collection and attenuation, and then sent to A/D converter with 128 sample points in every phase. The DSP chip performed FIR filtering, fast Fourier transform (FFT) operation, storage, and display of sample data. The communication between the circuit being measured and the Nios II processor was conducted through a serial port, based on the Modbus protocol.

Remote measurement of electric grid parameters can be divided into two parts: measurement of real-time electric grid data and monitoring power cut data. Measuring the real-time electric grid data is handled through a web interface. When the user clicks the **Measure Now** button in a web browser, a measurement command is sent to the system based on the Nios II processor, which in turn instructs the DSP chip to start real-time measurement. Then, the DSP chip returns the measurement results back to the web browser for users to verify. For example, the application software displays history of the most recent 25 blackouts.

The measurement functions of the electric grid parameters are realized in a dynamic web page display, using the Boa web server under uClinux and common gateway interface (CGI) technology. When the user sends a measure/check data command through the web browser, CGI scripts compiled with C language send the user command to the web server (Boa), which interprets and executes the CGI script and sends the results back to the browser.

Video Monitoring

Video monitoring is needed to guarantee safe, normal operation of important measuring instruments in real time. To implement video monitoring, our system has two critical components: video data collection and video data transmission.

Video data collection is realized using both hardware and software. The hardware part comprises a web camera and USB adaptor. Today, web cameras are a popular low-cost network video capture devices that are convenient for use in embedded applications. Because there is no support for USB devices on the Cyclone™ EP1C20 board included in the Altera Nios II development kit, we designed a USB interface to read video data from a web camera. We developed the software modules by modifying open source software like vgrabbj and xawtv, whose main function is to capture video from a web camera and convert it to the JPEG format. Next, this video data is sent to the network in real time using CGI.

We implemented video transmission using CGI and the web server. Working with the available electric grid parameters, the application software combines video information with electric grid information in an easy format that remote users can view through a web browser.

Performance Parameters

The performance is based on electric grid parameters and video image parameters.

■ *Electric Grid Parameters*—The measurement accuracy of the electric frequency is 0.01 Hz. For the three-phase voltage and current measurement, the accuracy is 0.5%. For the three-phase voltage imbalance factor, the accuracy is 0.2%. For the three-phase current imbalance factor, the accuracy is 1%. The harmonic measurement accuracy meets the GB/T 14549-93 B standard. The shortest measurement interval is 3 seconds.

■ *Image Parameters*—The screen-capture function adopts a 320[x]240 JPEG format with a 1-Hz refresh frequency. This performance meets the requirement of normal video monitoring.

Design Methodology

Our design comprises the following parts: software/hardware modules for measuring electric-grid parameters, uClinux OS kernel and file system configuration, web server configuration, CGI program development, USB interface board design, and development of an image capture program.

Hardware Design

The hardware circuitry is based on Texas Instrument's TMS320LF2407A DSP chip as the computing engine. The TMS320LF2407A features a 40-MHz clock and a single-cycle instruction execution time of 25 ns. Because the DSP chip operates at 3.3 V, and because all other chips in the system operate at 5 V, we needed to perform voltage-level conversion. Therefore, we chose the Altera EPM7128S device to implement level conversion and logic control. We also took advantage of the device's compatibility with 3.3-V and 5-V levels for use in communication control and address decoding functions between the DSP chip and other chips in the system.

To measure the three-phase voltage and current, we designed a protective circuit that deploys voltage sensors and current transformers for safe isolation from high voltage and currents. This circuit performs AC attenuation, acts as an anti-aliasing low-pass filter, performs A/D conversion, and passes data to the DSP chip for computation. The results are displayed and then stored for further analysis. At this point, all operations, including synchronous sampling by software, measuring signal frequency before sampling, and calculating sampling frequency based on the signal frequency, must be handled carefully to avoid FFT errors caused by frequency fluctuations.

Software Design

We can code DSP programs in C or Assembly language; C programs are easily readable, changeable, and are good for porting. However, their executable code has low efficiency. In contrast, Assembly language routines yield highly efficient executable code. To improve code efficiency and meet the requirements of a real-time system, we deployed the C2xxAssembly language routines for each software module and interrupt program. For example, we used Assembly language to take advantage of DSP special instructions in the FFT subprogram of the DSP data processing software module: bit-reversed indirect addressing, which is designed for real-time implementation of FFT arithmetic.

Configuration of uClinux OS Kernel & File System

We configured the uClinux OS kernel and file system as described in the following section.

■ *uClinux Kernel Configuration*—The embedded uClinux utilizes a customized Linux kernel with high flexibility, and is an open-source code that is stable and reducible. In this design, we adopted the uClinux version 1.3 edition, which was developed by Microtronix for the Nios/Nios II processor. Further, we reconfigured the kernel according to the requirements of the contest, utilizing the least system resources and keeping the function execution in mind.

■ *File System Configuration*—We chose the minimal install option in the interface configuration, and then added agetty, boa, dhcpcd, ftpd, inetd, init, ping, route, and telnetd.

■ *Modification of Kernel Source Code*—We modified the USB driver source code, USB controller SL811HS driver, web camera OV511 driver, and system files such as Kconfig and Makefile.

Web Server Configuration

The web server used in the project belongs to the uClinux file system. uClinux's own file system includes two web server programs, httpd and Boa. In our design, we chose the Boa web server because of its support for CGI. Boa configuration includes the following steps:

1. Open the **boa.conf** file in the **/target/etc/config** folder in the established file system.

2. Change **ChRoot** to **/mnt/ide0/www** to make the **/www** file on the compact flash card the main folder of the web server. When you type IP system, the server automatically analyzes and searches the web page named index.htm in the main directory, which is also the homepage of this design.

3. Add the command **ScriptAlias /mnt/ide0/www/cgi-bin/ /cgi-bin/** in the **ScriptAlias** section to map the forefront folder address with a complete path. This setting saves time for entering an address in the address column and enhances the security of the system. Defaults can be selected for other options.

4. Save the configured files. After downloading the file system, create a **cgi-bin** directory in the **/mnt/ide0/www** file for storing CGI scripts.

CGI Program Design

The CGI program enables monitoring from the client, who can issue measurement commands to the system through the web browser. Upon receiving commands, the CGI program performs the task of parameter monitoring and image supervision of the electric grid. Because the CGI program is written in C and embedded in an HTML page, when it is executed, the system can perform measurement operations on specific ports display the result on a web page for further action.

First, the CGI script called GET receives the analytical QUERY_STRING passed to the web page from web server, which acts as the customer's monitoring command center. After receiving the command, the CGI program decodes it and sends the collected command to the electric grid parameter data collecting module. Next, it receives the returned data and immediately passes it to the web interface, which is browsed by supervisors. It is important to note that after the CGI script is programmed and executed, it should be moved to the relevant directory on the compact flash card. It should also have its suffix changed to **cgi**, and attribute changed to executable for the web server to recognize and execute the CGI program correctly.

USB Interface Board Design

The SL811HS is a widely used USB controller in embedded systems supporting the USB1.1 protocol. Based on this controller, we designed a USB interface board and used the J16 pin on the Cyclone development board as the connecting interface to SL811HS hardware. Simultaneously, we programmed the timing conversion using HDL to conform to Avalon® bus timing and SL811HS timing specifications, and added it to the user-defined logic in the SOPC Builder tool.

Image Capture Program Design

Our image capture program in uClinux mainly uses video4linux application programming interfaces (APIs) provided by the kernel, which can capture images using image codecs. The APIs also support the USB interface and web camera's driver programs. The main functions of the program include an image collection device (web camera) initialization, mapping and capture of the current video frame, converting an image into JPEG format, and saving it to a file. We access this file through CGI, and display it on the supervisor's web page, which denotes the completion of the image capture and display process. Our software program uses several video collection programs, which observe the GPL license and run on Linux. These programs include vgrabbj, xawtv, and webcam server programs.

Design Features

Our design combines both electric grid parameter measurement and video monitoring tasks, which is perhaps a first of its kind in the market. We anticipate many application possibilities for our system with a huge market potential.

Our design is very economical because it uses Altera's FPGA as the core of the whole system. We adopted the Nios II processor, a JTAG-based hardware-debugging module that supported JTAG debugging online, and omitted an external emulator. By using a web camera to perform video capture, we saved a lot of money. Furthermore, our design is flexible enough to accommodate changes based on the customer's actual requirements, and can cater to different applications.

We ported the uClinux real-time operating system (RTOS) to the Nios II processor to take full advantage of RTOS resources and to facilitate faster system development.

Video capture and compression tasks use complicated computing. Thanks to the Nios II processor's powerful data-processing capability, we could accomplish all of the video capture and compression tasks of our system. Also, the Nios II processor has a superior price/performance ratio, and is a good choice for most embedded developers.

We can add peripherals to our system without making any modifications to system hardware. For example, we could use the Cypress SL811HS device as the main USB controller, and can easily interface the SL811HS chip onto the Avalon bus through user-defined peripherals in the SOPC Builder tool. We cannot make these types of modifications using other embedded systems based on the ARM MCU, etc.

By expanding the compact flash card memory as external storage for our embedded system, we can store dynamic web pages and update data at any time. Again, the enhancement and interconnection of the compact flash card module can be made quickly using SOPC Builder.

Conclusion

We learned a lot from this three-month design contest. We gradually developed a through understanding of the Nios II processor, which enhanced our confidence in it. We also gained a lot of knowledge about embedded design techniques. In our opinion, Altera's SOPC design methodology is a highly flexible, efficient solution. Using SOPC Builder, we could customize the Nios II CPU for our design requirements and use system resources efficiently. Compared to other 32-bit processors with the same functions, the Nios II processor offers a very good price/performance ratio. Additionally, the Nios II IDE development environment has a user-friendly interface, facilitating fast system design and program debugging. Furthermore, Altera provided many reference designs, some of which we used with some modifications.

As Altera has improved the Nios II processor, other intellectual (IP) and software companies have perfected their support for it. For example, we used the Microtronix's uClinux RTOS, which is an embedded OS especially tailored for the Nios II processor, in our design. Using the Nios II IDE environment, we could easily make modifications to the RTOS: we modified the USB driver code to support our USB web camera better, developed a CGI program for communication with the web server, and implemented dynamic web page technology. We also edited the programs and grafted them onto GPL in IDE.

Because of the need for real-time arithmetic computation, we used a special DSP device for the electric grid parameter measurement. However, we believe we could have implemented the same measurement circuitry totally in the FPGA if we had used the Stratix® FPGA. Using this device, we could have integrated all of the system functions, and realized the whole system as a single-chip implementation. This implementation is what we intend to do in future.

During debugging of the USB web camera function, there appeared to be a collision (in communication) between the USB controller and the web camera hardware. Although we had tried our best, we could not solve this problem because of time considerations. Although our video capture software ran very well on Red Hat Linux 9.0 and Fedora 4, and after a successful edit in IDE 1.1 and 5.0, we failed to collect images on the EP1C20 board because of this hardware collision. Although the contest has come to an end, we will keep trying to address this problem to develop a more perfect system.

Third Prize

TCP/IP Offload Engine (TOE) for an SOC System

Institution: **Institute of Computer & Communication Engineering, National Cheng Kung University**

Participants: **Zhan Bokai and Yu Chengye**

Instructor: **Chen Zhonghe**

Design Introduction

Today, the Internet plays an important role in everyone's life. The 100-Mbps network system has become very popular in schools and offices, while the 1-Gbps network system is deployed in server host networks such as large portal websites and Storage Area Networks (SAN). Meanwhile, the Internet SCSI (iSCSI) protocol is recognized as the new standard for the emerging network storage technology—networking of storage component. However, the iSCSI protocol will place heavy demands on server processing when applications run on high-speed networks. According to recent studies, you need 100% efficiency from a Pentium III, 1-GHz processor or 30% efficiency from a Pentium 4 2.4 GHz processor to process the 1-Gbps TCP protocol. Therefore, it is necessary to accelerate network-processing capability and reduce the CPU load by adding extra hardware.

Accelerating system processing has a bearing on the built-in system featuring network functions. The network communication protocol performance depends on memory (memory access times) and data volume. For example, you need lots of memory access to read data when computing checksums, which leads to an efficiency bottleneck. Additionally, when the network interface controller (NIC) receives large volumes of data, the data is split into many different segments based on protocol restrictions. Therefore, subsequent interrupts affect the CPU efficiency. If protocols are processed through additional hardware and with an enhanced memory management mechanism, the above problems can be solved effectively.

The left side of Figure 1 shows a standard TCP/IP stack. To implement TCP/IP in an embedded system, we need two important blocks: the PHY and media access controller (MAC) that function in the physical layer and data-link layer.

These function blocks also operate under the local real-time operating system (RTOS) in the third layer (network layer, IP layer) and the fourth layer (transport layer, TCP layer). Thus, all TCP/IP protocols can be implemented in a software-only solution. We replaced the network protocol with our TCP/IP Offload Engine (TOE), which you can see in the system's architecture on the right side of Figure 1. TOE uses hardware modules to implement protocols, and uses drivers to communicate with upper-layer space and operating systems.

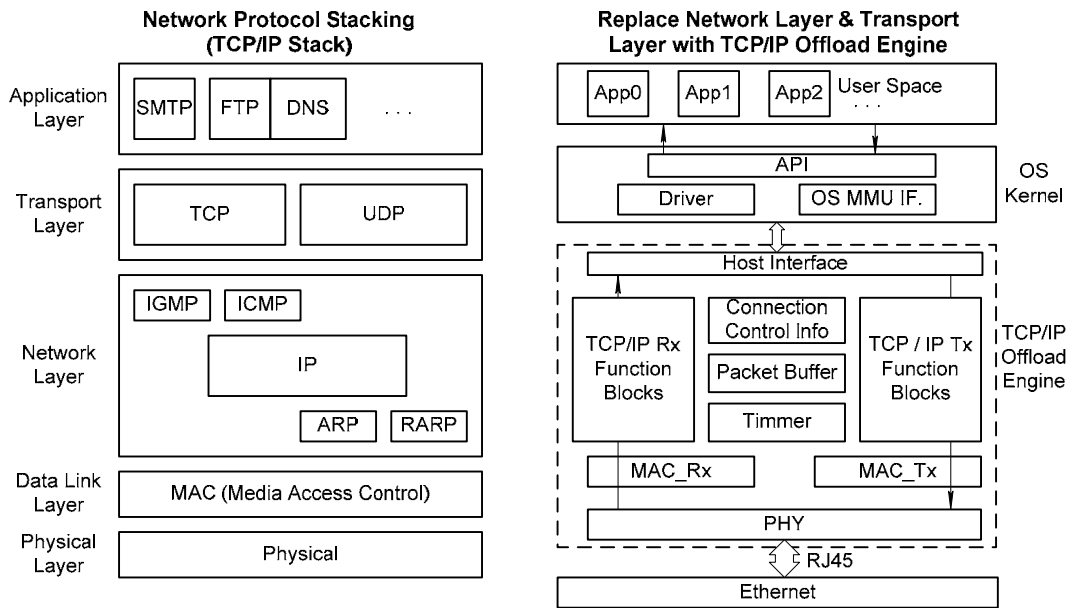*Figure 1. Network Protocol Stacking (TCP/IP Stack) & TOE Engine*

Figure 2 shows the system overview of our design in the FPGA. The system adopts the Avalon® system bus. In addition to the TOE architecture, the design includes system main memory, a Nios® II processor, DMA, the LAN91C111, and a PHY/MAC chip controller. We will describe the TOE's major functions and blocks in a later section.

In this system, the Nios II processor functions as the primary controller for data communication between the MAC, TOE, and main memory. The MAC receives a message box data and sends it to a data buffer space of TOE (TCP/IP Offload Engine) through firmware, which processes the data and sends it to the main memory. While transmitting, MAC first sends the data to the internal data buffer space of the TOE, which then adds headers and transfers it back to the MAC for transmission.

Figure 2. Hardware Overview
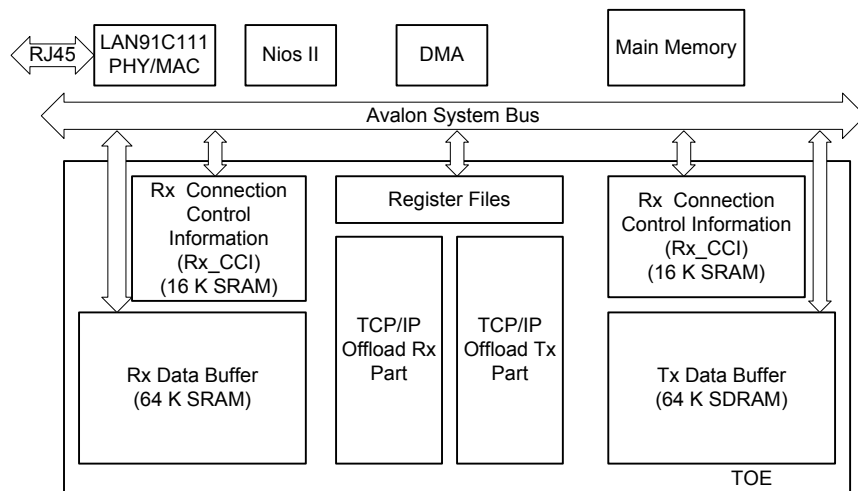
We used the Nios II processor because the Altera® PCI development board provides peripherals such as the PCI interface and SDRAM, which are required by the system. Although we were unable to achieve the final goal in this contest, that is to use SDRAM as a data buffer to support more connections and use the PCI interface to communicate with the upper-layer operating system (OS), we selected the above development board and related Altera integrated development environment (IDE) to facilitate easy development and scalability in the future.

Function Description

Although we designed the general TCP function block, we could not complete it due to time constraints. Therefore, the final product does not provide the TCP function. The paragraphs below will mark this part as "Unfinished".

Major Functions & Blocks

Figure 1 shows a complete TCP/IP stack. However, because the whole network protocol is too complicated, we have only implemented the most used functions of TCP/IP protocols. The white shaded blocks (ARP, ICMP, IP, UDP, and TCP) in Figure 1 were implemented in our design. Three major functions are included in this design:

■ Send pings and respond to echo request packets (ARP & ICMP).

■ Provide UDP transmission capability of up to eight connections simultaneously.

■ Establish and manage up to eight TCP connections simultaneously (unfinished).

In the hardware scheme shown in Figure 2, the TOE body includes modules as follows:

■ Four SRAMs (alt_syncram):

   • Two 64-KByte SRAMs are used as data buffers, which are temporary packet storage for receiving and transmitting (Rx and Tx data buffer).

- Two 16-KByte SRAMs are used as storage blocks for connection control information (CCI) of receiving end and transmitting end. The two blocks will record the status of data buffer, and the queue information of protocol processing sub-module. All data except packet data are stored in this CCI.

■ Rx and Tx protocol processing blocks:

- Rx protocol processing module.

- Tx protocol processing module.

The two protocol processing modules consist of small modules, which are responsible for partial logic functions of specific protocol layers respectively.

■ Register files (TOE internal buffer), which generally includes the following items:

- CPU control bit.

- TOE status.

- Control buffer of item addition at transmitting end (CPU is used to initiate a buffer group that sends work instruction).

- Control buffer of item addition at receiving end (CPU is used to initiate a buffer group that receives work instruction).

- ARP table control buffer.

- UDP control block control buffer.

- TCP control block control buffer (unfinished).

- Protocol modules control buffer.

- Queue substrate of protocol processing sub-module and item number control buffer.

Refer to the "Design Architecture" section for the implementation of each module.

Performance Parameters

Because the TCP module is unfinished, the buffer access times and memory data volume are used for TCP performance evaluation.

Performance Evaluation

The performance can be evaluated as follows.

■ *100-Mbps wire-speed*–In Ethernet applications, the TCP maximum segment size (MSS) is usually set to 1,460 bytes, while the IP maximum transfer unit (MTU) is set to 1,500 bytes. Therefore, the size of the message box is often 1,518 bytes (a MAC header is 14 bytes, and a CRC is 4 bytes). In addition, there are eight bytes of preamble when the MAC transmits a valid message box. Therefore, to receive a message box it needs 1526[x]8[x]10 ns = 122080 ns = 0.12179 ms, that is we need to process a frame within 0.12208 ms or 12208 clock cycles.

■ *1-Gbps wire-speed*–The timing is 1/10 of the result above, meaning we need to process a frame within 12.2 μs or 1220.8 clock cycles.

## *Practical Performance Analysis*

The test scenario is:

■ TOE: operation frequency 40 MHz

■ PC transmitting packet: 1 GHz Linux running UDP program

■ Measuring packet volume: 2 GHz Windows XP running Ethereal

TOE Performance

Figure 3 shows the number of packets in different sizes that the TOE can process per second. Rx-Tx means that when receiving a packet, the TOE always transmits a responding packet in the same size, so the number of packets processed shall include Rx packet and Tx packet. Rx means a packet in response is transmitted upon receiving 1,000 packets. Therefore, the number of packets processed is the Rx packet number. Additionally, the Nios II CPU processes all data replications.

Figure 3. TOE Performance

	Rx-Tx	Rx
⊠ 60 Byte	12,000	10,664
⊞ 518 Byte	1,970	2,912
▢ 1030 Byte	1,230	1,675
◪ 1514 Byte	996	1,220
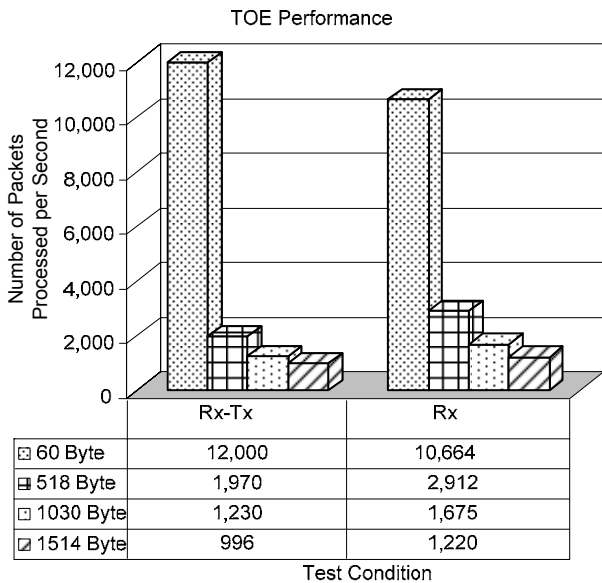
In Rx-Tx testing, when the packet size is 60 bytes, the number of packets processed by the TOE per second can go up to 1514 bytes (packet size). This is because the TOE has three data replications from receiving to transmitting: the first from MAC buffer to TOE RxBuffer, the second from TOE RxBuffer to TOE TxBuffer, and the third from TOE TxBuffer to MAC.

In Rx testing, when the packet is 60 bytes, the number of packets processed by TOE per second can go up to 1514 bytes (packet size), which is 12 times the size when compared with Rx-Tx testing. This is because the replication from TOE RxBuffer to TOE TxBuffer is reduced.

In Rx-Tx testing, when the packet is 60 bytes, the number of packets processed by TOE per second is 12,000; that is, the packets received are 6,000. However, when provided with the same packet size, TOE only receives 10,664 packets per second in the Rx testing. This is because the CPU has to process data replications of Rx packets and Tx packets simultaneously.

Provided that the time for data replication is deducted, and if we assume a packet size of 1514 bytes, the data volume received per second is 6000[x]1514/1000000=9.084 Mbytes.

In Rx testing, when the packet is 60 bytes, the number of packets processed by TOE per second is 10,664. Provided, time for data replication is deducted, and assuming packet size of 1514 bytes, the data volume received per second is: 10664[x]1514/1000000=16.145 Mbytes.

As a result, the performance bottleneck of TOE lies in the data replication. If the time of data replication is reduced but the speed is accelerated, the TOE will be able to process 100 Mbps network speed. To process 1-Gbps data, we need to improve the processor frequency of up to 1 Gbps/16.145 Mbytes =7.8 and 40 MHz x 7.8 = 312 MHz.

Design Architecture

This section describes the system's architecture.

TOE Hardware System Design Concept

As shown in Figure 4, we divided stacks into two modules: Rx and Tx, which are both ASICs and are attached to the same system bus. Therefore, they can be controlled by the same embedded CPU and share a common memory. However, the TCP module needs to communicate with input modules in full-duplex mode (the size of the sliding window is influenced by ACK reply), and the two modules can operate independently. The advantage of this design is that the transmitting and receiving allows parallel processing as long as some appropriate firmware is deployed.

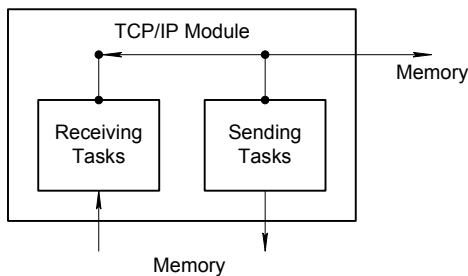*Figure 4. Parallel Processing Module Dividing (Tx & Rx)*


Figure 5 shows the relationship between the TCP packet processing module and the buffer. After the protocol processing is finished on the first layer, the processed data is discarded to the next layer in order to avoid too much memory-to-memory transfer, so that the processing module of each communication protocol will be able to read and write using a shared memory.

Figure 5. Protocol Processing Module Required for TCP Packet Processing & Shared Memory
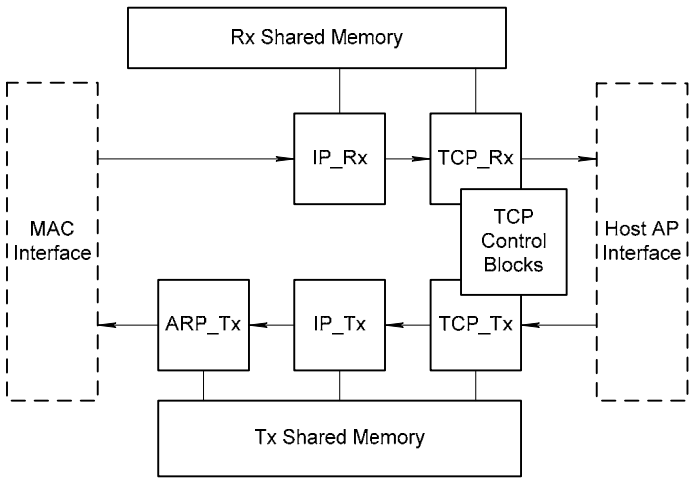
When it finishes processing a protocol packet, the module notifies the next module of a pointer in the same shared memory so that the processing flow can obtain the data with minimum memory transfer volume. However, no memory exceeding three access ports is available for use. Therefore, we need to determine which memory will come first when reading and writing the buffer via the arbiter circuit when we connect these modules with the buffer. We need to research whether the different applications or connections are related in this arbitration mechanism.

Figure 6. Protocol Module Division Design
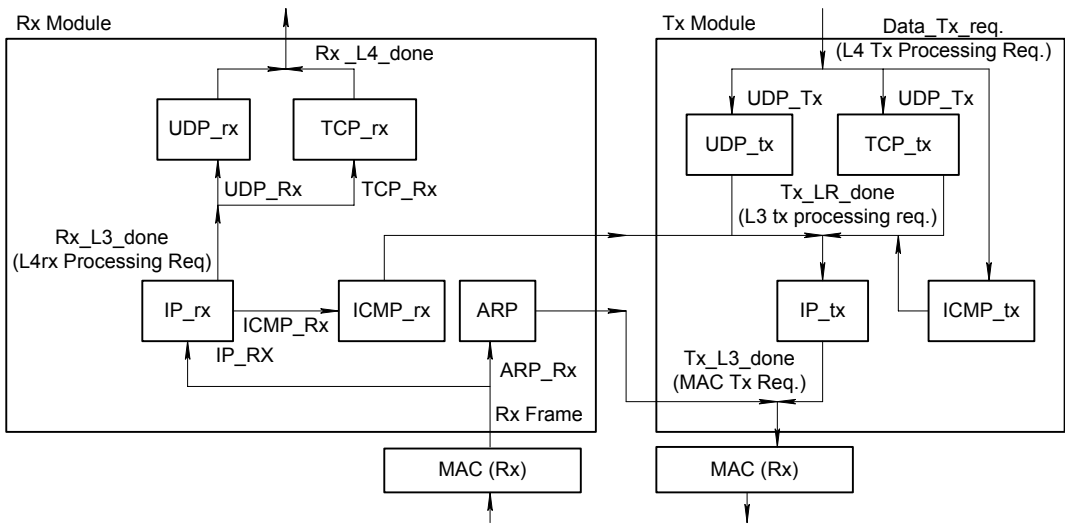
The three major functions (TCP, UDP, and ping) in the document are realized based on the *TCP/IP Protocol Suite* by Behrouz A. Forouzan and Sophia Chung Fegan. We have utilized hardware circuits to complete the logic function of each software module (using multi-cycle based finite state machine to realize function of single module by stages), and then connect the logic circuit to shared memory for communication (see Figure 5). To realize parallel processing, the whole TCP/IP stack is divided into several modules as shown in Figure 6. As mentioned above, such a design approach enables it to receive

packets while transmitting. Additionally, when a packet is processing TCP, it can accept another packet to occupy the IP processing module for implementation of protocol-related work on the IP layer. Each protocol module runs the state machine as shown in Figure 7.

Figure 7. State Machine Structure of Protocol Processing Module
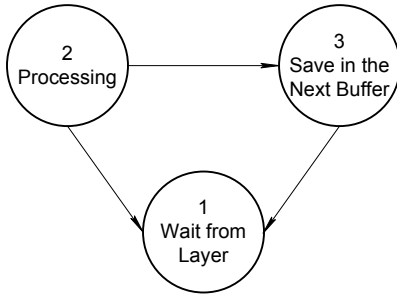
In short, these sub-modules use multi-cycle logic circuits to work as software modules. The module group responsible for receiving analyzes the header fields of the message box stored in the data buffer according to sequence, and then processes this data. The transmitting module clusters add network headers to the data segment that needs to be transmitted according to user instructions (for example, driver and system firmware).

Process Communication Queue Among Modules: Buffer Tables, Connection Control Information (CCI)

The communication among the processing modules is shown in Figure 8 and is realized through queues. In our design, we implemented it using a buffer table. The buffer table consists of pointers and information required by packet processing and is stored in the connection control information (CCI) RAM, pointing at a memory block to store complete packets into the data buffer. Each processing module is related to at least two tables; one indicates the data pointer where the data is processed, and the other holds the data pointer required to notify the module on the next layer when the module processing is finished.

In Figure 8, the lines connecting the blocks indicate table names. Each buffer table has several items, with data pointers pointing to the packet located in the data buffer. The ARP module captures the data block required for ARP protocol processing in the ARP_Rx table. It then performs the operation and store the finished data pointer in the Tx_L3_done table. Finally, it waits for the MAC_Tx module to read the pointer and output the data. Figure 8 shows the logic-buffer table and the data buffer. In our design, each data pointer has four fields as shown in Figure 9. Field description is explained following the data buffer design description.
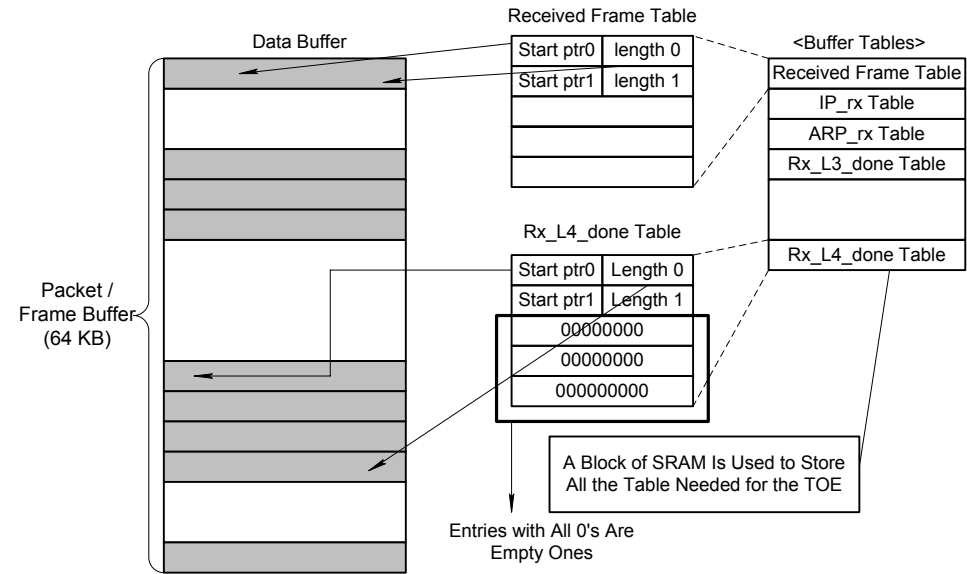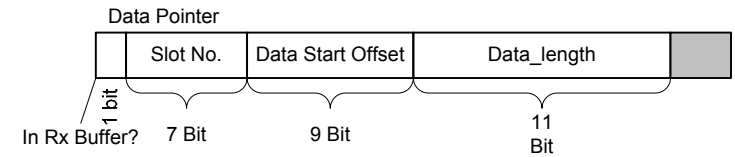
Figure 8. Buffer Table Scheme

Figure 9. Data Pointers

Data Buffer Area Design

To facilitate buffer management, the 64-Kbyte data buffer area is divided into several slots, each occupied by a message box. The larger message boxes may occupy more slots. In this design, the number of slots is set in the compiled hardware circuit. We have set 128 slots in the current trial version, where each slot takes up 512 bytes. In this way, the buffer management is less complex and requires less CCI storage.

Data pointer fields in Figure 9 can be explained as follows:

■ In the Rx buffer: 1 bit, shows whether data is at the receiving end buffer or the transmitting end buffer.

■ Slot No.: Number of the first slot occupied by the data.

■ Data start offset: Indicates the start offset of the valid data.

■ Data length: Length of the data.

The second and third bit can be combined to form a 16-bit data buffer address, pointing to the start point of the data.

314

Logically, the data buffer puts the contents of the message box into memory. However, because the packets are not processed in sequence, a linked list mechanism changes the small usable area into serial memory blocks, improving the working efficiency of the data buffer. If a frame can obtain a whole list of serial data while requiring the data buffer, the linked list mechanism is not started; instead, the serial idle segments are distributed into the frame.

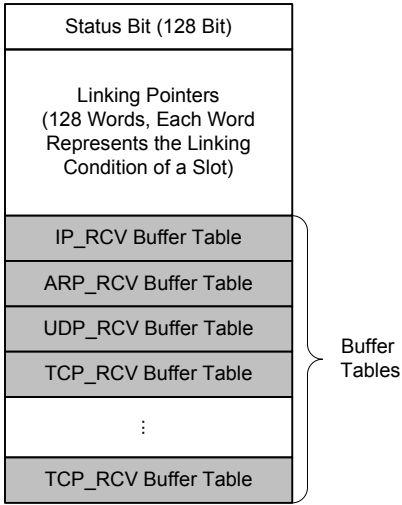*Figure 10. Data Buffer & Linked List*
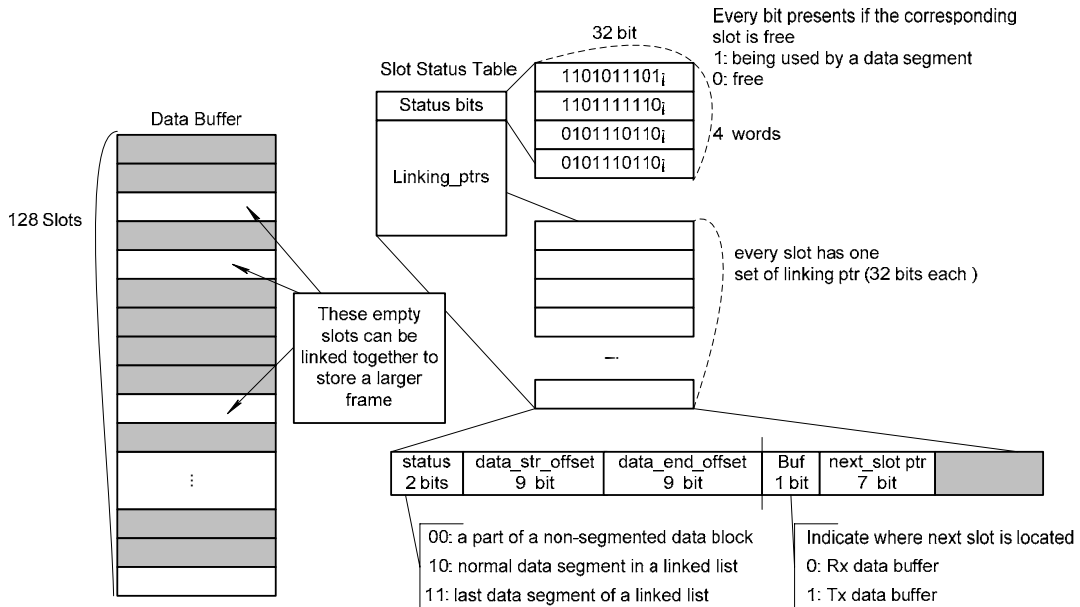
As shown in Figure 10, the idle spaces in the data buffer are not in series because of the non-sequential processing. The best way to deal with this mechanism is to connect the idle segments to a linked list. Except when transferring whole packets (for example, to transfer a message box from MAC to TOE or from TOE to main memory), other packet headers do not need to support a linked list, because the slot size is set to far exceed the overall length of all protocol headers. This means that the linked list does not span the slot while it is processing the packet header.

In Figure 10, we can see that the data buffer and linked list mechanism needs two memory blocks to support it. These memory spaces are put into CCI, and the resulting CCI memory allocation is shown in Figure 11. The first memory area is called status bits, where there are a total of 128 data bits. Each one represents the status of a slot in the data buffer—1 for occupied and 0 for idle. The second memory area is called linking ptrs (linking pointers), which records the interconnection status of occupied data slots. Here there are 128 total units, each with a 32-bit data, and each corresponds to one slot. As shown in the figure, the highest 2-bit indicates the slot status—'00' means that the slot is a part of the data that has not be segmented; '10' means the slot is a node of linked list; '11' means the slot is the last node of linked list. Next to this are two 7-bit fields, respectively, indicating the starting and ending shifts of valid data; the last 9 bits is a data pointer to the next data slot.

The linked-list mechanism relies on the Nios II processor for management. So when the hardware module processes headers, the mechanism is not used for one slot. When it needs to send data into TOE, the Nios II processor detects the idle slot by referring to status bits in the TOE; and if necessary, initiates a linked-list mechanism to store the data in the TOE. When the data is moved out of the TOE, it determines whether there is a linked list, then reconnects the processed data and transmits it to the destination.

315

Figure 11. CCI Internal Memory Allocation

Status Bit (128 Bit)
Linking Pointers (128 Words, Each Word Represents the Linking Condition of a Slot)
IP_RCV Buffer Table
ARP_RCV Buffer Table
UDP_RCV Buffer Table
TCP_RCV Buffer Table
⋮
TCP_RCV Buffer Table

Buffer
Tables

Processing Flow

Figure 12 shows the transmit packet flow.

Figure 12. Process Flow of Transmitting Packet
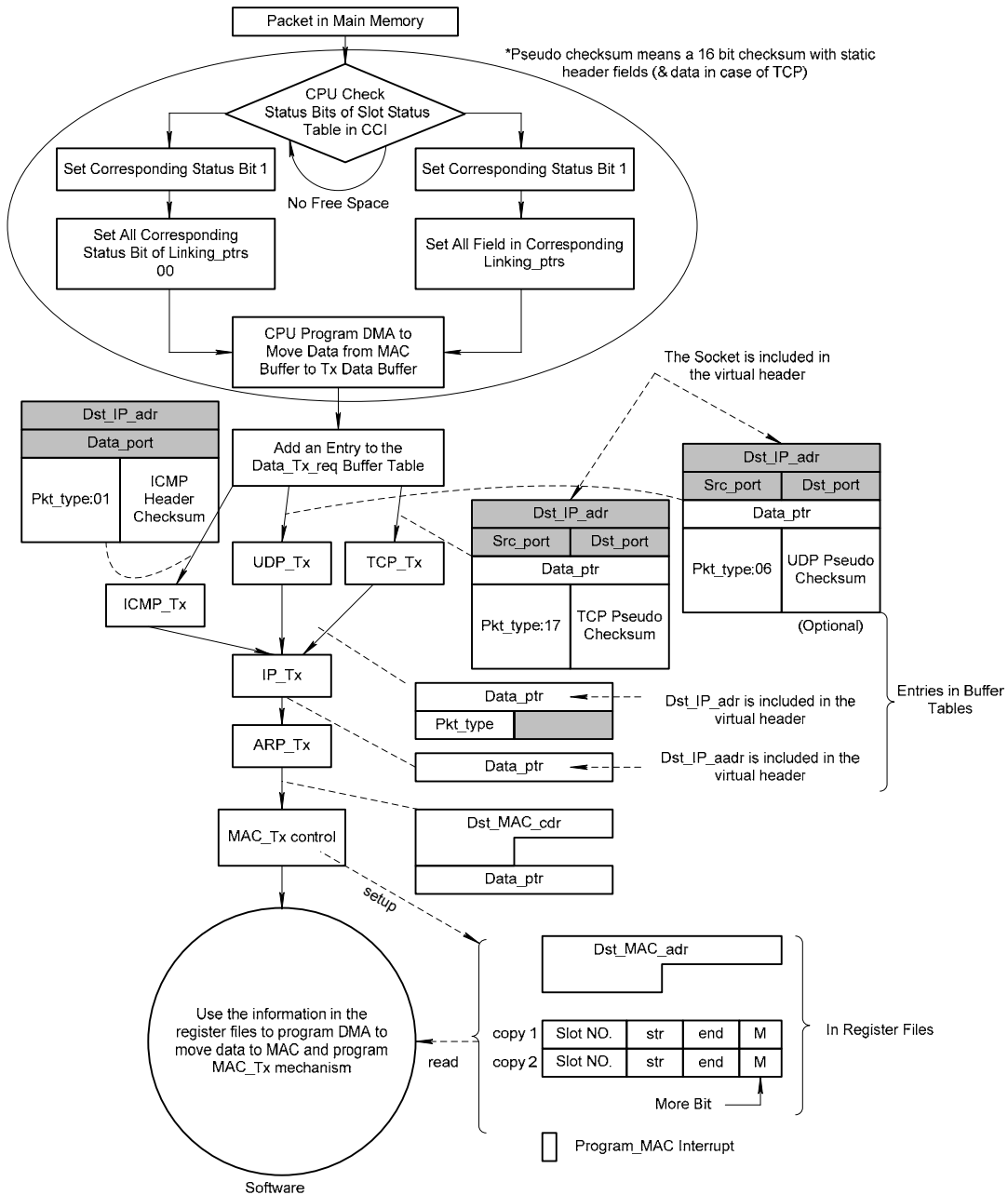
Figures 12 and 13 show the transmitting and receiving flows, respectively, in which the Nios II CPU processes the contents in circle objects (at the beginning and end of flow) via software, while the square contents are processed by a hardware protocol processing module. The squares that are connected through broken lines and bold arrows on two sides of the flow show the data structure of single items in the hardware-processing-module communication queue.

When the TCP_Tx module is transmitted to the IP_Tx module, you need the data pointer and packet type (see below).

Referring back to the data buffer design in Figures 10 and 11, the data buffers of both the receiving and transmitting ends have two fields in CCI to record the status of the data buffer. These are status bits and linking pointers: status bits use 0 and 1 to record if the slot is idle; the linking pointers record the interconnection status among each slot. When data is transferred into TOE, CPU needs to control the DMA engine for data transfer and modify the data structure that records the data buffer status. This operation ensures correct control of the processing module operation.

Figure 12 explains the process flow for packet transmission.

■ When the packet that is going to be transmitted is in the main memory, the CPU checks whether the status bits in the slot status table denote enough space. There may be three outcomes:

• *Sufficient serial space*—Setthe status bit of the corresponding space to be used to 1. Set the status bit of the corresponding Linking_ptrs of slot to be 00.

• *Sufficient non-serial space*—Set the status bit of the corresponding space to be 1. Each field of the corresponding Linking_ptrs shall be set.

• *Insufficient space*—The CPU checks continuously until there is sufficient space.

■ The CPU calls the DMA to transfer the data from the main memory to the Tx_data_buffer.

■ Add an entry into the Tx_data_req and notify the TCP/IP offload engine to transmit some data.

■ In the TCP/IP offload engine, each module has a corresponding table, and the data is processed as follows: the upper-layer module finishes data processing and adds the pointer to the data that is put into the table of next layer. In this way, the module knows which data to process by looking at the table.

■ After the last layer arp_tx in the TCP/IP offload engine finishes processing, it sends out an interrupt signal. The CPU calls the DMA to transfer the data from the Tx_data_buffer to the MAC buffer to complete the packet transmission.

Figure 13 shows the receive packet flow.

Figure 13. Process Flow of Receiving Packet
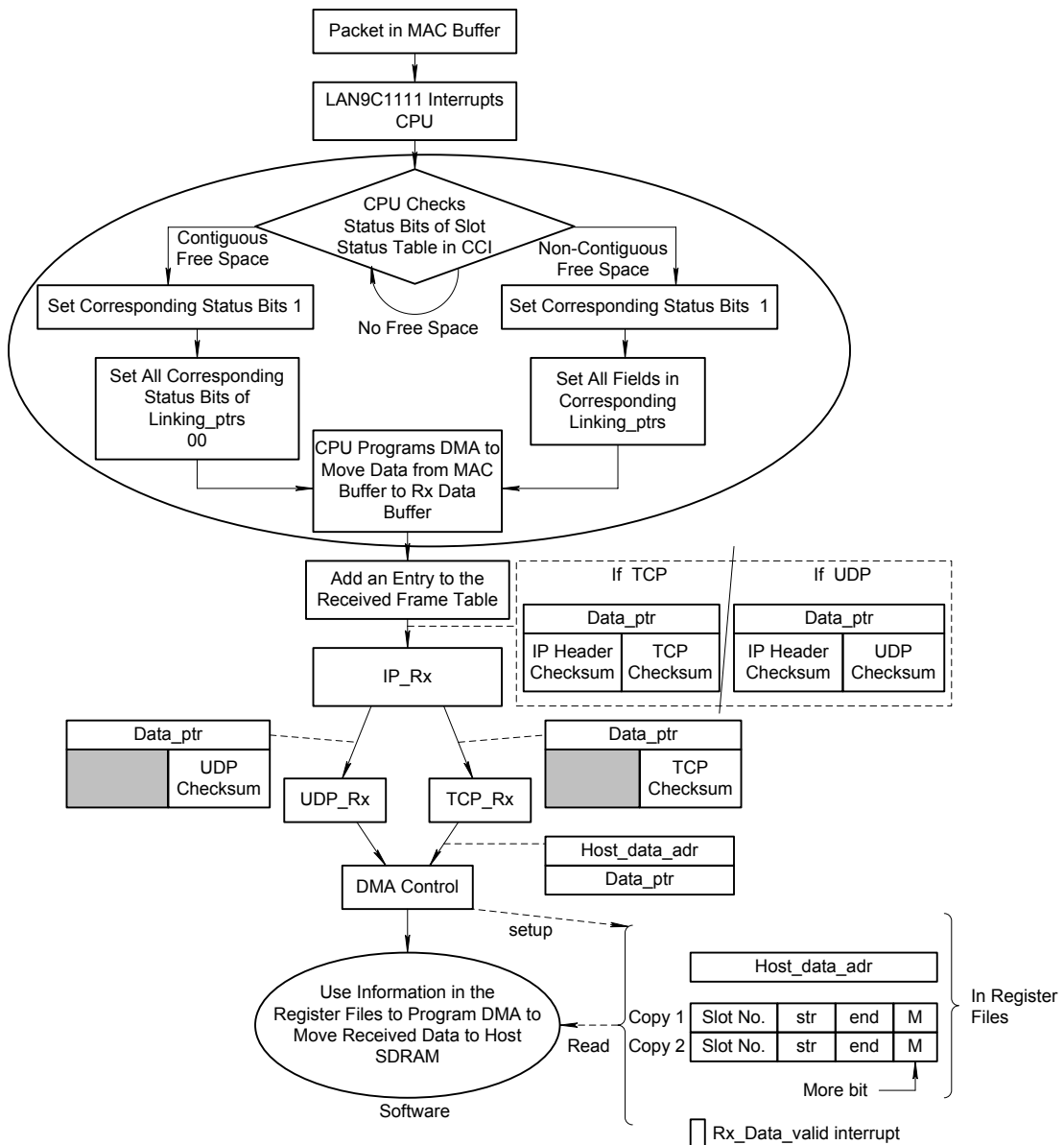
Figure 13 shows the process flow for packet receiving.

■ When the packet that is going to be transmitted is in the main memory, the CPU checks whether the status bits in the slot status table indicate enough space. There can be three results:

• *Sufficient serial space*—Set the status bit of the corresponding space to 1, and the corresponding Linking_ptrs status bit to 00.

• *Sufficient non-serial space*—Set the status bit of the corresponding space to 1. Each field of the corresponding Linking_ptrs shall be set.

- • *Insufficient space*—CPU will check continuously until there is sufficient space.

■ The CPU calls the DMA to transfer the data from the main memory to the Rx_data_buffer.

■ Add an entry into frame_table and notify the TCP/IP offload engine that the data has been received.

■ In the TCP/IP offload engine, each module has a corresponding table, and the data is processed as follows: the upper-level layer finishes the data processing and adds the pointer pointing to the data that is put into the table onto the next layer. In this way, the module knows which data to process by looking at the table.

■ After the last layer Tcp_rx and udp_rx in the TCP/IP offload engine finish processing, it sends out an interrupt signal. The CPU calls the DMA to transfer the data from the Rx_data_buffer to the main memory to complete receiving the packets.

Design Methodology

The architecture of the whole system is shown in Figure 14, detailing several main components:

■ Nios II CPU.

■ DMA engine.

■ Main memory constructed by Altera synchronous SRAM (in the future, main memory could be constructed by the development board DDR SDRAM and SDRAM controller).

■ Tri-state bridge for Lan91c111 PHY/MAC.

■ Custom ASIC TCP/IP Offload Engine.

Figure 14. Hardware Overview
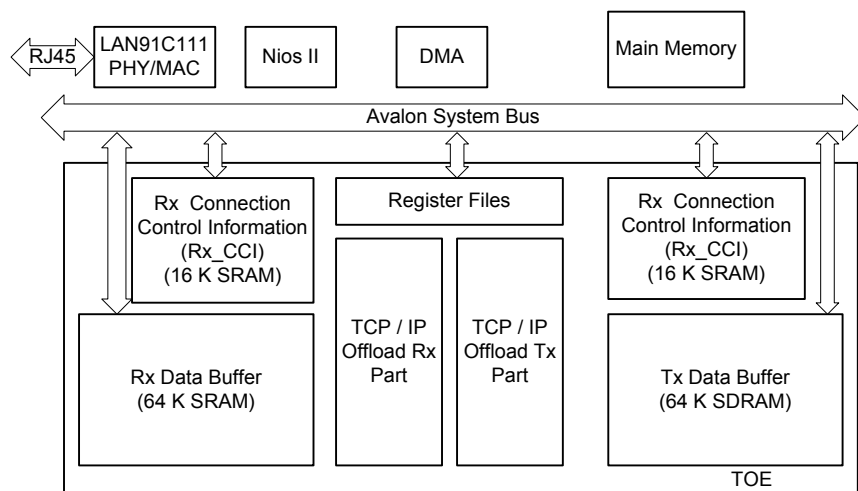
The interconnection of these components was configured using the SOPC Builder tool, and the data was transferred into each component through Avalon system bus. Our TOE features the following five external channels:

■ Access channel of CCI system at receiving end.

■ Access channel of CCI system at transmitting end.

■ Access channel of data buffer system at receiving end.

■ Access channel of data buffer system at transmitting end.

■ Register file access.

These channels connect with the Avalon system bus through the interface to user logic in SOPC Builder, which is set in slave mode; meanwhile, based on an address mechanism, the Nios II processor can access five channels of TOE through this interface.

The system is controlled by the Nios II CPU, which executes the program located in the main memory. The program initializes the lan91c111 and TOE, waits for interrupts, and then executes the corresponding interrupt service routing (ISR).

The system features the ISR described as follows.

Lan91c111 PHY/MAC ISR

When the processed packet is put into the MAC buffer and the subsequent interrupt is generated, the CPU shall do the following:

■ Check status bits in the slot status table to find if the rx_buffer has enough space for the packets.

■ Packets are transferred from the MAC buffer to the rx_buffer by calling the DMA function.

■ Add an entry to the Rx_frame.

TOE ISR

The CPU checks for the type of interrupt served by the TOE_status_reg:

■ *Interrupt receiving packet*—The CPU calls the DMA to transfer the processed packets to the main memory.

■ *Interrupt transmitting packet*—The CPU calls the DMA to transfer the processed packets to the MAC buffer.

DMA ISR

The CPU checks status bits in the slot status table to find out whether the Tx_buffer has enough space for packets.

■ Packets are transferred from the Rx_buffer to the Tx_buffer by calling the DMA.

■ Add an entry to the Tx_frame.

Design Features

The main features of our system design are as follows:

■ If we used only the Nios II CPU to process the TCP/IP network protocol, it would be impossible to reach 100 Mbps. However, by deploying both the Nios II CPU and the TOE architecture to process the TCP/IP network protocol, we can reach 100 Mbps.

■ Our architecture modularizes each protocol. Each module performs according to its own rules table. Therefore, we could easily add new protocols using the flexible TOE architecture.

■ The modules for sending and receiving are separated, and this scheme enables message-box processing in parallel.

■ Support for eight UDP connections.

■ Hardware ARP table.

■ Hardware UDP connection-management mechanism.

■ When the processing modules handle the message box, they only transfer the pointers to the message-box, reducing data transfers in memory.

■ Due to the network function created in hardware, we were able to reduce host CPU resources occupied by network tasks in embedded applications.

■ We can efficiently manage the data buffer by dividing the storage memory into numerous slots, which simplifies the mechanism of data link list. Also, you can easily change the slot size configuration to meet different requirements by simply altering the design parameters.

■ The system focuses on network storage devices, so the parameter setting in the current design is set to optimize storage-network applications.

■ If you take into account the time from the MAC operation to receiving and storing this data onto destination main memory, our system can process the data packets at 100-Mbps of network speed.

Conclusion

In our opinion, the toughest challenge for the system designer is to have the system verified by application hardware. Fortunately, Altera's PCI development board provides an environment that enables testing of the system and interconnection bus that features CPU, DMA, main memory, PC bridge, and interconnecting peripherals. You can easily construct the system by using the SOPC Builder tool, which connects its own IP with the peripherals. For example, our system uses MAC and PHY as network devices, and we developed firmware using the C language.

We constructed the test environment using SOPC Builder, emphasizing the development of the hardware processing core. If the IP is implemented, its performance can be measured by using the SOPC Builder, which will greatly reduce the IP development time. Although this board provides enough I/O, IP is generally too expensive and we need to devote more effort for developing submodules in case there are not enough resources in the lab. For instance, we could not achieve the expected results in the design because there was no proper bridge connecting the DDR SDRAM Controller and the PCI bus. If we had IP of the trial version or the simulation environment of this bridge peripheral, it would have helped us to dramatically reduce the system development time.

Appendix:
Nios II Embedded Processor Family

Today's embedded design engineers face a tough challenge: finding a processor with the perfect mix of features, cost, performance, and manageable life cycle. Altera's Nios® II processors deliver the perfect fit every time with fully customizable features and performance, low product and implementation costs, ease of use and adaptability, and obsolescence-proof design.

The Nios II family of 32-bit RISC embedded processors delivers more than 200 MIPs of performance[1] and can consume as little as US$0.35 of logic. Because the processors are soft core and flexible, you can choose from an unlimited combination of system configurations to meet your performance, feature, and cost targets. Designing with Nios II processors helps you send products to market faster, extend your product's life cycle, and avoid processor obsolescence.

Customizable Feature Set

Rather than being limited to a pre-fab processor, with Nios II processors, you choose the exact peripherals, memory, and interface features you need—customizing the processor to your specifications. In addition, you can easily integrate your own proprietary functions to give your design a unique competitive advantage.

Configurable System Performance

You want a processor with enough performance for both your current and future designs. While the future is uncertain, designers can easily modify their designs to add multiple Nios II CPUs, custom instructions, or hardware accelerators—even after the product ships—to achieve new performance goals. Your system performance can be adjusted through the Avalon® switch fabric, Altera's specialized interconnect technology, which supports unlimited parallel data paths for high-throughput applications.

Low-Cost Implementation

When choosing a processor, you may face the choice of either buying more processor than you want just to get the features you need, or buying less processor than you need to meet your cost goals. Using low-cost, customizable Nios II processors, you can include as many or as few features as you need, and you can implement them on a low-cost Altera® device, such as a Cyclone™ II FPGA, for as little as 35

---

[1] Dhrystones 2.1

cents of logic. By implementing processors, peripherals, memory, and I/O interfaces on a single FPGA, you'll also decrease your overall system costs.

Life Cycle Management

To create a successful product, you need to get it to market fast, enhance the feature set to extend its useful life, and avoid long-term processor obsolescence. Nios II embedded processors can be taken from concept to system in minutes. They are completely obsolescence-proof with their perpetual, royalty-free licenses to design and produce Nios II processor-based systems. In addition, by implementing a soft-core processor on an FPGA, in-field hardware upgrades are as easy as software upgrades, allowing products to meet the latest standards and incorporate new features.

Unparalleled Flexibility

Completely customizable and reconfigurable, the Nios II processors are adaptable to product requirements for today and the future.

Three Processor Cores

The Nios II processor family consists of three cores—fast (Nios II /f), standard (Nios II /s), and economy (Nios II /e)—each optimized for a specific price and performance range (Table 1). All three cores share a common 32-bit instruction set architecture and are 100 percent binary code compatible. You can easily add Nios II processors to your systems by using the SOPC Builder tool featured in Altera's industry-leading Quartus® II design software.

Table 1. Nios II Processor Family Members

Feature	Nios II /f (Fast)	Nios II /s (Standard)	Nios II /e (Economy)
Description	Optimized for maximum performance	Faster than the fastest and smaller than the smallest first-generation Nios CPU	Optimized for minimum logic usage
Pipeline	6 Stage	5 Stage	1 Stage
Multiplier	1 Cycle*	3 Cycle*	Emulated in software
Branch Prediction	Dynamic	Static	None
Instruction Cache	Configurable	Configurable	None
Data Cache	Configurable	None	None
Custom Instructions	Up to 256	Up to 256	Up to 256

* Using DSP Blocks in Stratix® or Stratix II FPGAs.

Peripherals

Nios II development kits include a library of commonly used peripherals and interfaces. For a complete list of SOPC Builder-ready intellectual property (IP) and peripherals, visit www.altera.com/SOPCBuilderReady.

Using the interface-to-user-logic wizard in the SOPC Builder software, you can also create your own custom peripherals and integrate them into Nios II processor systems. With SOPC Builder and Altera FPGAs, you can assemble embedded processor configurations not available in off-the-shelf processors, letting you create exactly what you need, every time.

Avalon Switch Fabric

The Avalon switch fabric enables multiple, simultaneous data transactions for unmatched system throughput. SOPC Builder automatically generates an Avalon switch fabric optimized to the specific interconnect requirements of your system's processors and peripherals.

324

In traditional bus architectures (Figure 1), a single arbiter controls communication between the bus masters and slaves. Each bus master requests control of the bus, and the arbiter then grants bus access to a single master. If multiple masters attempt to access the bus at once, the arbiter allocates bus resources to a master based on a fixed set of arbitration rules. This can lead to a bandwidth bottleneck as only one master can access the system bus and its resources at a time.

Figure 1. Traditional Bus Architecture
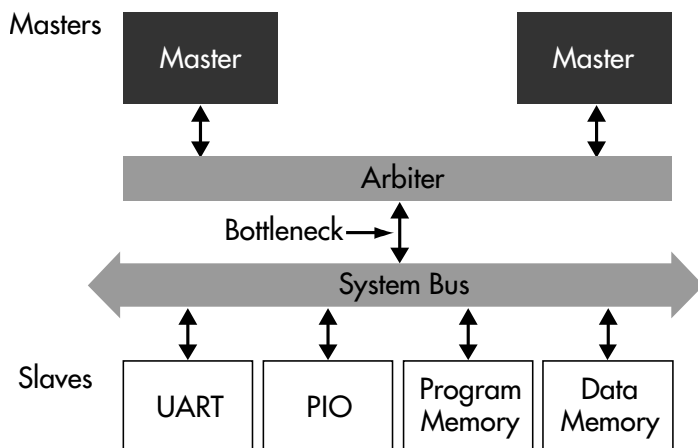
The Avalon switch fabric's simultaneous multi-master architecture increases your system's bandwidth by eliminating this bottleneck (Figure 2). Using the Avalon switch fabric, each bus master gets its own dedicated interconnect, meaning that bus masters only contend for shared slaves, not for the bus itself. Each time a component is added or the peripheral access priorities change, SOPC Builder generates a newly optimized Avalon switch fabric with a minimum of FPGA resource use.

*Figure 2. Avalon Switch Fabric Architecture*
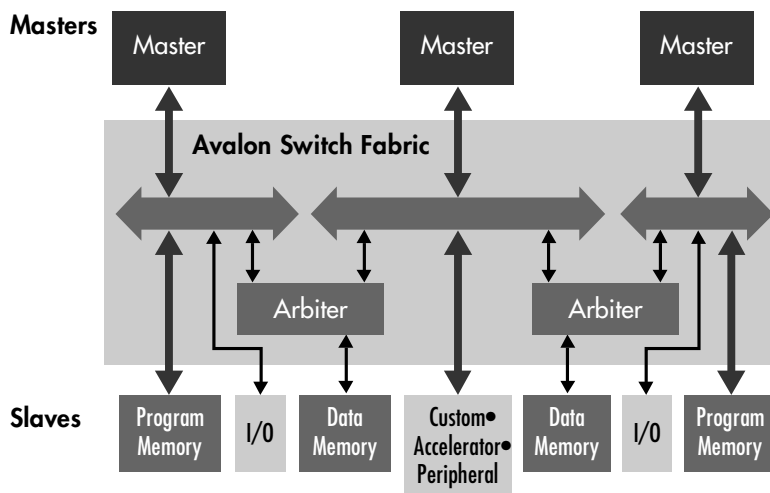
The Avalon switch fabric supports a wide range of system architectures, including single- and multiple-master systems, and allows seamless data transfers between peripherals with performance-optimized data paths. Your design's off-chip processors and peripherals are equally well supported by the Avalon switch fabric.

Custom Instructions

Custom instructions allow developers using Nios II processors to increase system performance by extending the CPU instruction set to accelerate time-critical software. Using custom instructions, you can optimize system performance in a way not possible with traditional off-the-shelf processors.

The Nios II family of processors supports up to 256 custom instructions to accelerate logic or mathematically complex algorithms normally handled in software. For example, a block of logic that performs a cyclic redundancy code calculation on a 64-Kbyte buffer operates 27 times faster as a custom instruction than when performed by software (Figure 3). Nios II processors support fixed and variable cycle operations, include a wizard for importing user logic as a custom instruction, and automatically create software macros for use in developers' code.

Figure 3. Nios II Custom Instructions
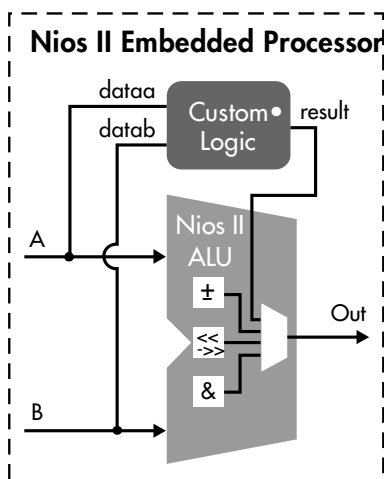
Hardware Acceleration

Large blocks of data can be processed concurrently to CPU operation by adding application-specific hardware accelerators (Figure 4) that act as custom co-processors within the FPGA. Using the cyclic redundancy code example shown in Figure 3, processing a 64-Kbyte buffer runs 530 times faster with hardware accelerators than software. SOPC Builder includes an import wizard that allows developers to add their acceleration logic and DMA channel to the system.

Figure 4. Nios II Hardware Accelerator
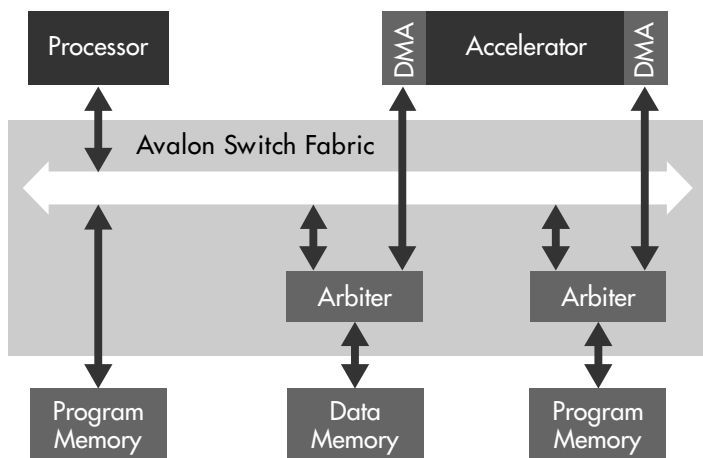
Altera also offers the Nios II C-to-Hardware Acceleration Compiler (C2H Compiler), a productivity tool that gives embedded developers push-button acceleration of performance-critical C-language software algorithms. These algorithms are automatically converted into hardware accelerators in the FPGA that act as coprocessors with a latency-aware, pipelined connection to the processor's memory map. With this tool, designers have an easy way to boost performance using a known programming language and familiar tools, improving productivity and speeding time-to-market.

The C2H Compiler is tightly integrated into the Nios II development environment (Figure 5), leveraging Altera's proven SOPC Builder tool and the Avalon switch fabric interconnect to automate the conversion of ANSI C source code to hardware (register transfer language), integrate the resulting hardware accelerator into the system's memory map, and schedule memory transactions with latency-aware pipelining. It enables developers to quickly prototype functions in software running on the processor, then easily convert the software into a hardware-accelerated implementation.

Figure 5. C2H Compiler
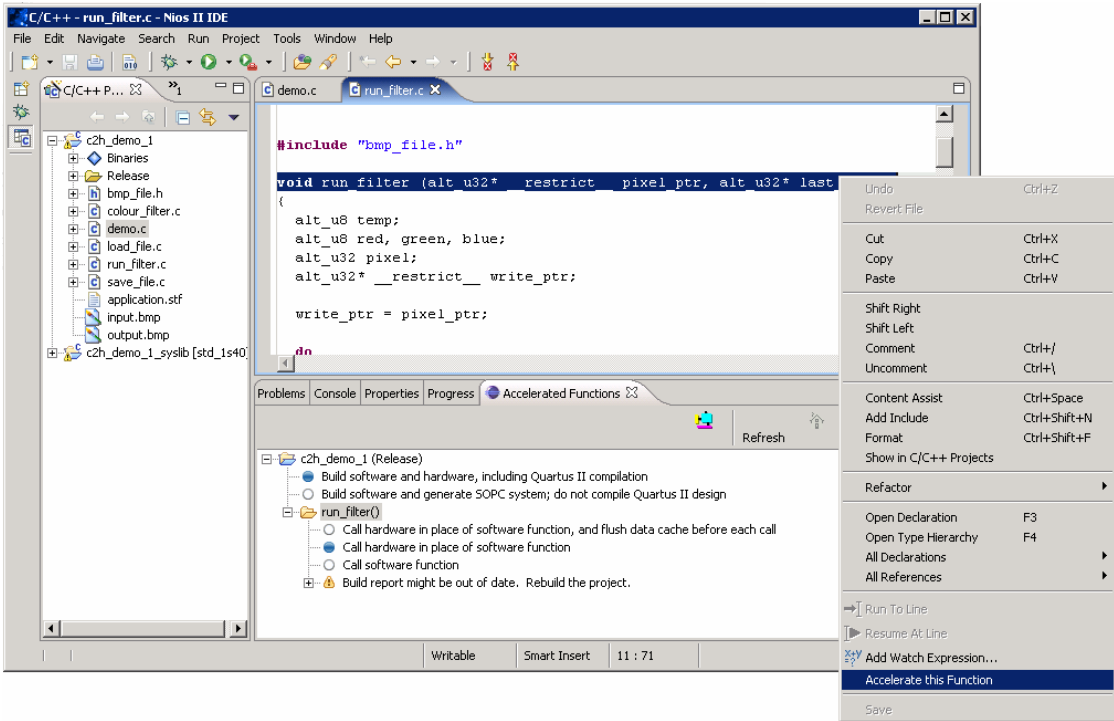
Device Family Support

You can quickly and easily implement Nios II processors in all of Altera's mainstream devices. Altera's complete range of high-performance, high-density, and low-cost devices gives you the FPGAs to fit any embedded design. For higher-volume applications, you can implement Nios II processors in HardCopy® series structured ASICs with no royalties or additional license fee requirements.

Complete Development Tool Suite

Altera's comprehensive hardware and software tools help you create powerful Nios II processor systems in minutes. Figure 6 shows the complete Nios II embedded processor design flow. From concept (at top), through hardware and software implementation, to debug, Altera offers all the tools you need to get your product to market fast.

*Figure 6. Nios II Embedded Processor Development Flow*
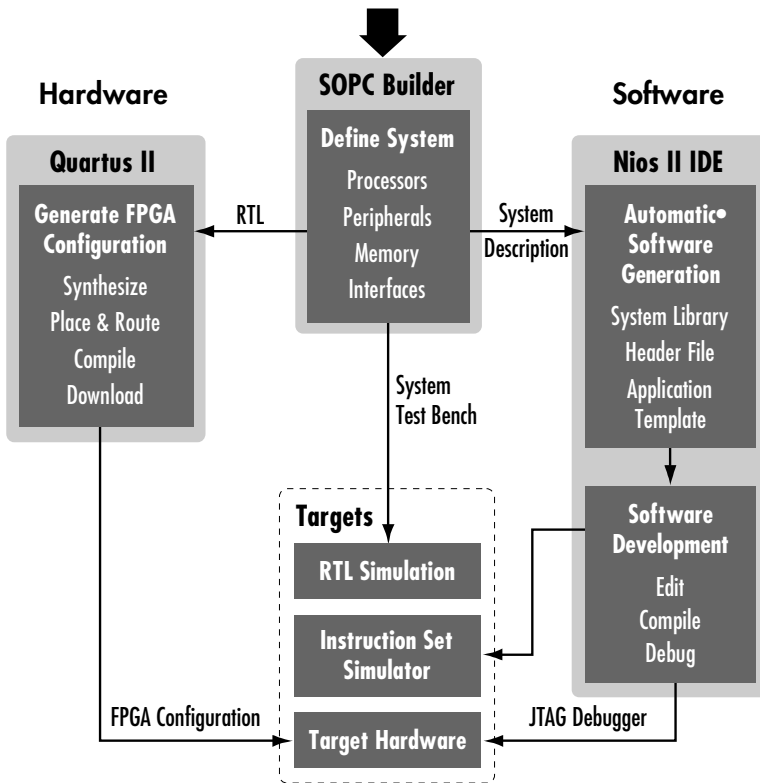
Hardware Design

Altera provides a complete set of tools for your hardware design, including the SOPC Builder system development tool, Quartus II design software, ModelSim®-Altera software, and SignalTap® II embedded logic analyzer.

SOPC Builder

Hardware design for creating Nios II processor-based systems uses the SOPC Builder system development tool to specify, configure, and generate systems. Launching from within the Quartus II design software, SOPC Builder provides an intuitive wizard-driven graphical user interface (GUI) so you can create, configure, and generate system-on-a-programmable-chip (SOPC) designs. It minimizes the time spent integrating components into a coherent system. Figure 7 shows a view of the intuitive SOPC Builder GUI.

Figure 7. SOPC Builder GUI
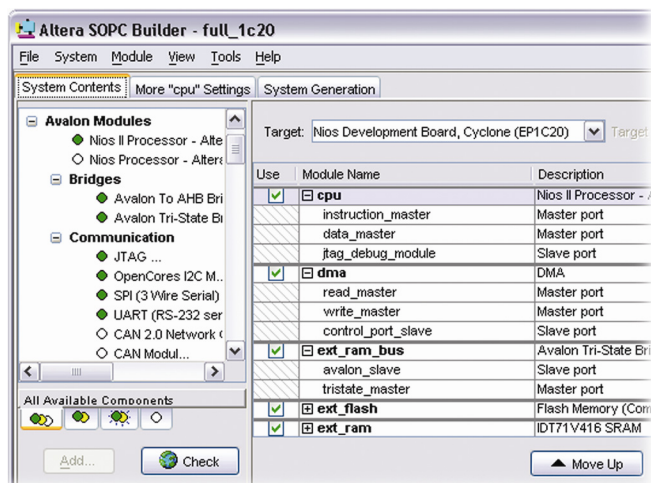
Quartus II

Altera's Quartus II design software technology leadership gives you unmatched levels of performance and ease-of-use. Using Quartus II software, you can easily design, optimize, and verify your Nios II designs in an Altera device.

When you're ready to simulate your design, SOPC Builder generates both VHDL and Verilog HDL simulation models. You can easily simulate Nios II processor-based systems using an automatically generated simulation environment created by SOPC Builder and the Nios II integrated development environment (IDE). A full Quartus II software subscription includes ModelSim-Altera software that can also be used to simulate your Nios II designs.

SignalTap II

The ultimate testbench for engineers who want to see the active processes within their design is running at speed under real-world system conditions. The challenge is in accessing nodes buried within the FPGA architecture. The SignalTap II embedded logic analyzer eliminates this challenge by providing access to nearly any node within your FPGA design through a standard Joint Test Action Group (JTAG) port to view design nodes in system and at system speeds.
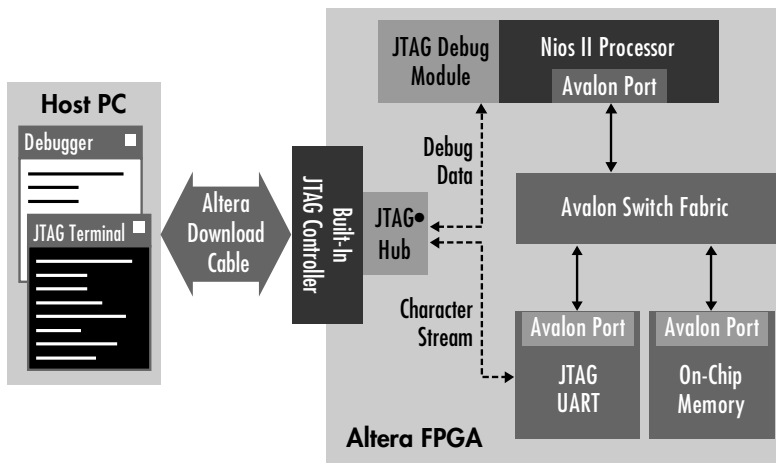
Software Design

To make the software design flow as easy as possible, you can accomplish all software development tasks within the Nios II IDE, including editing, building, and debugging programs. As part of the Nios II IDE, Altera partners with operating system and middleware providers for additional software development tools.

Nios II Integrated Development Environment

Based on the open, extensible Eclipse IDE project and the Eclipse C/C++ Development Tools project, the Nios II IDE is the primary software development tool for the Nios II family of embedded processors. You can complete all software development tasks within the Nios II IDE, including editing, compiling, downloading, debugging, and flash programming. The Nios II IDE, shown in Figure 8, provides a consistent development platform that works for all Nios II processor systems. With a PC, an

Altera device, and a JTAG download cable, you have everything you need to develop and debug Nios II processor-based systems.

Figure 8. Nios II IDE

JTAG Debug Module

The Nios II architecture supports a JTAG debug module that provides on-chip emulation features to control the processor remotely from a host PC. The Nios II IDE communicates with the JTAG debug module on one or more Nios II processors so you can:

■ Download programs to memory

■ Start and stop program execution

■ Set breakpoints and watchpoints

■ Analyze registers and memory

■ Collect real-time execution trace data

The debug module connects to the JTAG circuitry built into all Altera devices and connects to the host PC via a download cable (Figure 9). Additionally, debug support for the Nios II processor is available from several industry-standard providers.

Figure 9. Nios II JTAG Debug Module

Instruction Set Simulator

The Nios II instruction set simulator (ISS) allows you to begin developing programs before the target hardware platform is ready. The IDE allows you to run programs on the ISS as easily as running on a real hardware target.

Flash Programmer

Many designs that use Nios II processors also incorporate flash memory on the board. Any CFI-compliant flash device connected to the FPGA can be programmed using the Nios II IDE flash programmer. The Nios II IDE flash programmer can also program any Altera serial configuration device connected to the FPGA, as shown in Figure 10. The Nios II IDE flash programmer is pre-configured to work with all of the boards available with the Nios II development kits, and can be easily ported to any custom hardware.

Figure 10. Transmitting Flash Content to the Flash Device
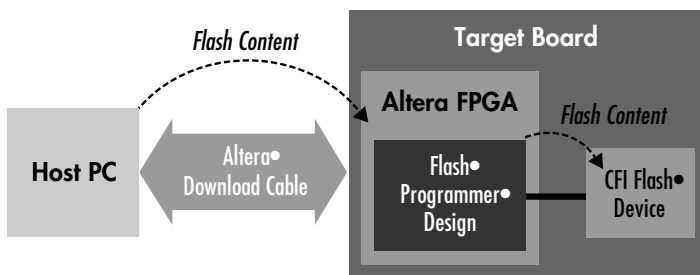
Software Templates

In addition to a project set-up wizard, the Nios II IDE provides software code examples, in the form of project templates, to help you bring up working systems as quickly as possible.

Each template is a collection of software files and project settings. You can add your own source code to the project by placing the code in the project directory or importing the files into the project. Figure 11 shows some of the available software project templates.

Figure 11. Software Project Templates
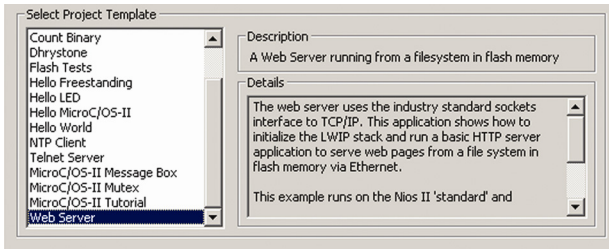
System Software

The Nios II IDE lets you customize systems quickly using system software. With system software (also referred to as "software components"), you have an easy way to painlessly configure your system for specific target hardware.

Hardware Abstraction Layer

The hardware abstraction layer (HAL) system library is a lightweight runtime environment that provides a simple device driver interface for programs to communicate with underlying hardware. As SOPC Builder and the Nios II IDE are tightly integrated, the HAL system library can be automatically generated to serve as a board support package for Nios II processor-based designs.

MicroC/OS-II

A complete, portable, ROM-able, preemptive real-time kernel, MicroC/OS-II from Micrium ships with all Nios II development kits and includes full source code, reference manual, and free developers' license. When you're ready to migrate your design to your board, you can purchase a shippers' license. A shippers' license entitles you to a license for three developers to create an unlimited number of designs for one year on MicroC/OS-II and a perpetual license to support designs created during the subscription period (to fix bugs and make minor modifications).

TCP/IP Stack

Nios II development kits ship with an open-source lwIPTCP/IP stack that is built to work with MicroC/OS-II applications and implements the standard UNIX Sockets API. The software is available as source code with complete documentation, reference designs, and technical support from Altera.

Linux

Linux designers requiring a full-featured operating system with network protocol stack, file system, and other popular device drivers can download the open source µCLinux port for the Nios II processor family from www.niosforum.org.

Nios II Development Kits

Altera and its partners offer development kits that give you everything you need to start designing the perfect processor for your system today: from documentation to download cables, from boards to design software. One example kit is shown in Figure 12. To find out more, visit Altera's development kits web site at www.altera.com/devkits.

Figure 12. Nios II Development Kit, Cyclone Edition

Learn More

There are several ways to learn more about the Nios II processors, all of which begin by navigating to the Nios II home page at the Altera web site (www.altera.com/nios2) where you can:

■   View online demonstrations

■   Read in-depth technical documentation

■   Download an evaluation version of the Nios II processors and Nios II IDE

■   Check out the latest Nios II development kits

■   Register to attend on-line or instructor-led training

When you're ready for the next step, simply order a development kit or contact a sales office. Visit www.altera.com today for details.

You can also visit the Nios design forum site (www.niosforum.org), where Nios and Nios II users around the world share ideas, design examples, and other information.