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

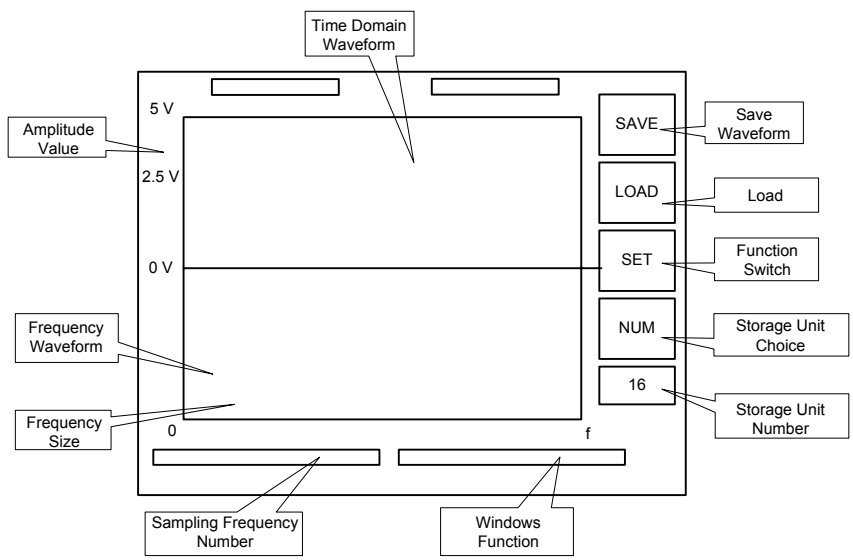
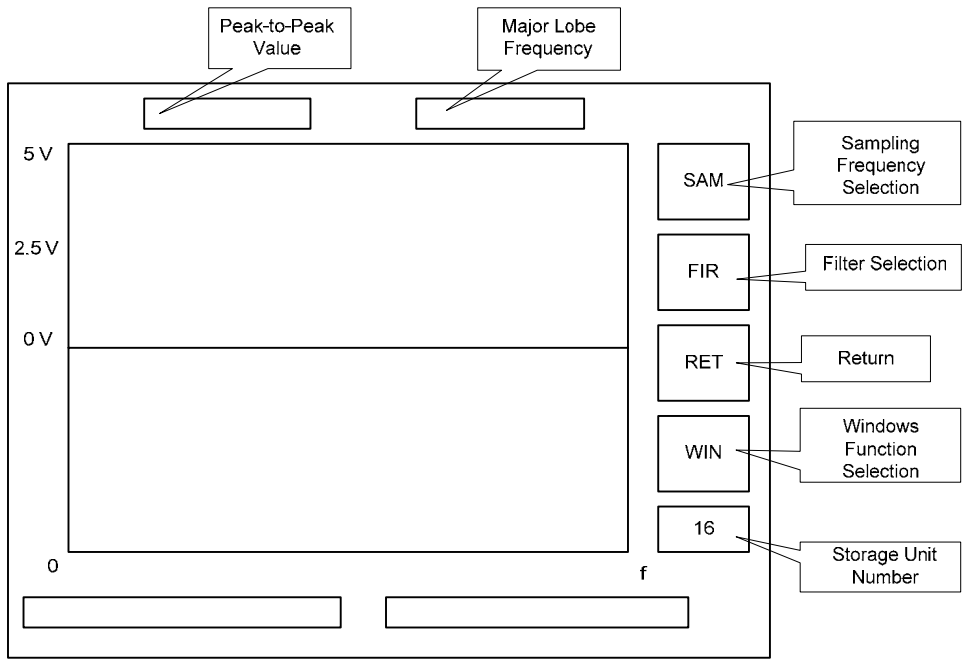


Figure 2 shows the second level menu options.

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

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:

$$w_H(n) = 0.5 - 0.5\cos(\frac{2\pi n}{N}) \ , \quad n = 0, 1, \ \ldots, N - 1 \qquad (2.1)$$

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.

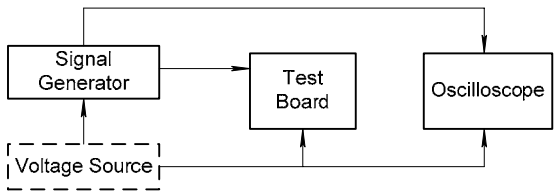
Experiment instruments:

- 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 as100 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

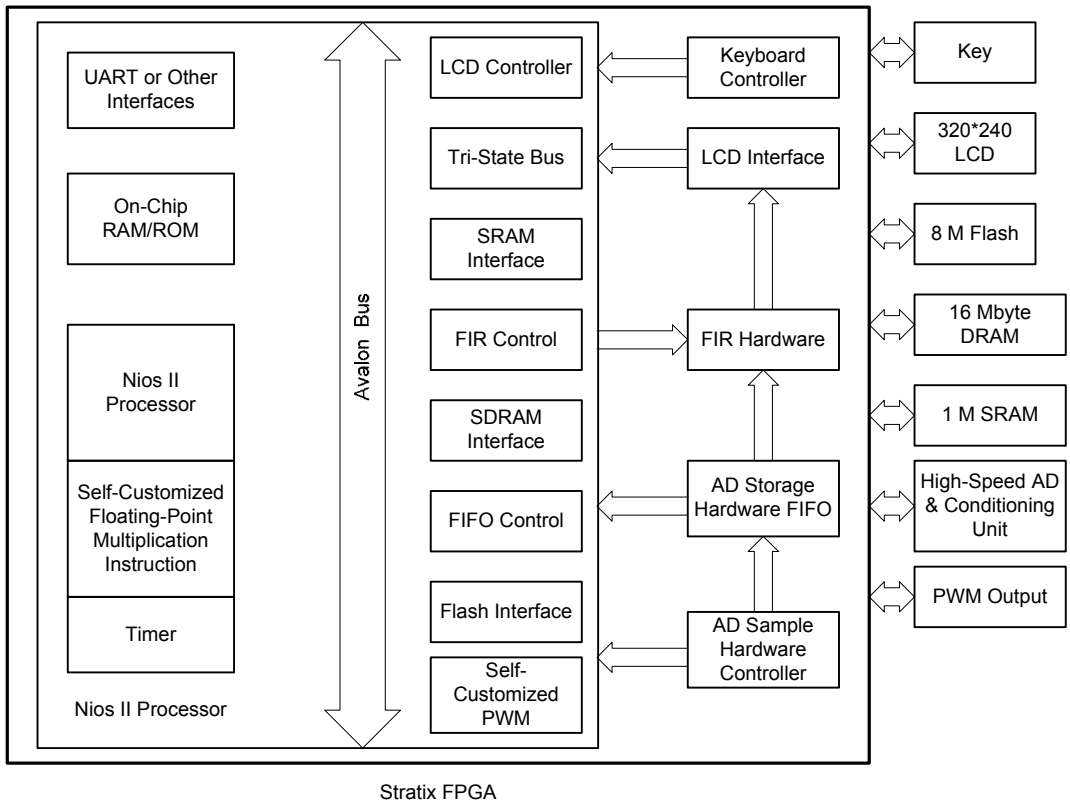
Conclusion: With the experiment, the repeat (copy) accuracy of time domain waveform meets the requirements of our design.

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

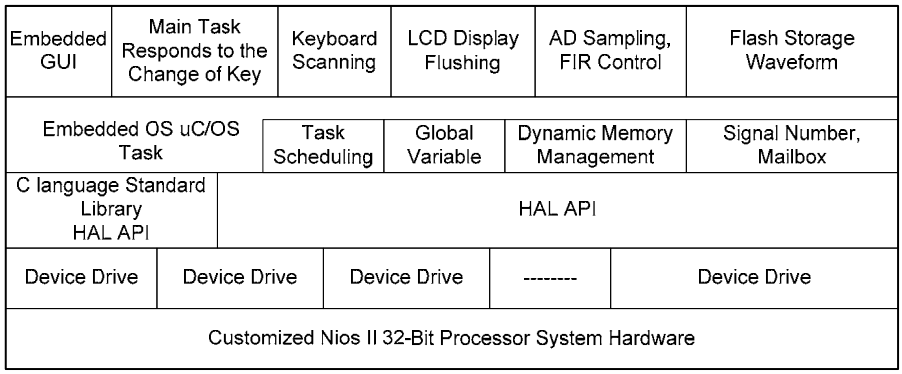


The software is implemented based on a Hardware Abstraction Layer (HAL) provided by the Nios II IDE. The software tasks are handled by a multitasking real-time operating system, the μC/OS II, which improves program readability and simplifies program development. We added a graphical user interface, μC/GUI 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



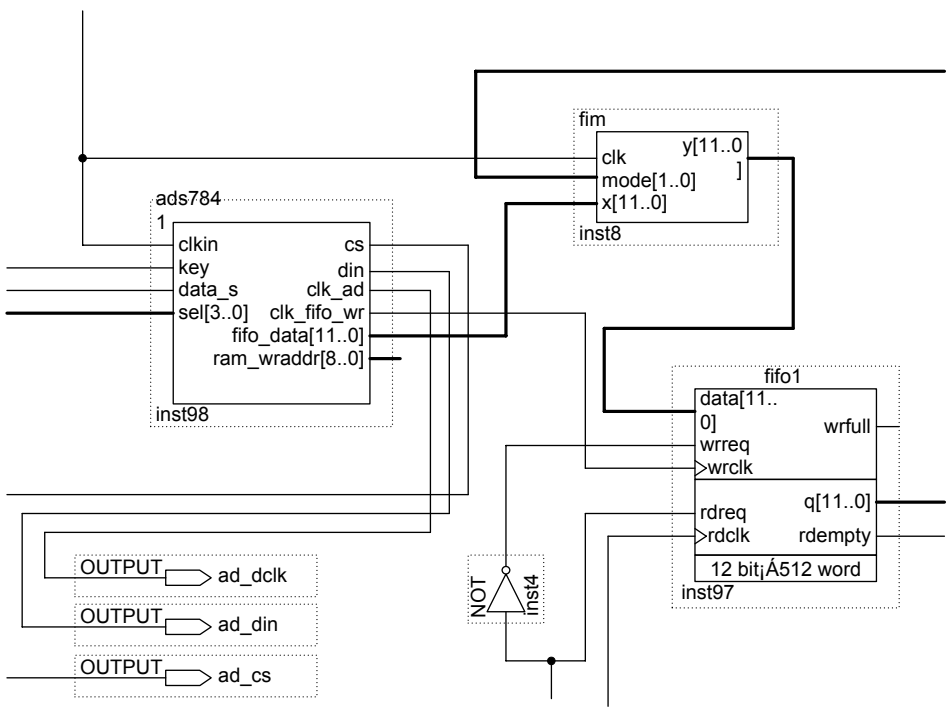
Design Methodology

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

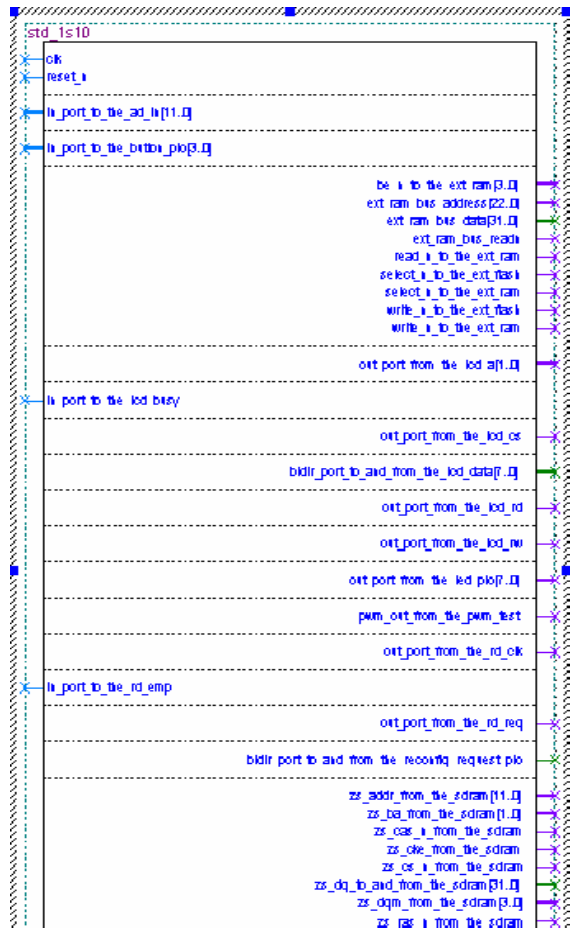
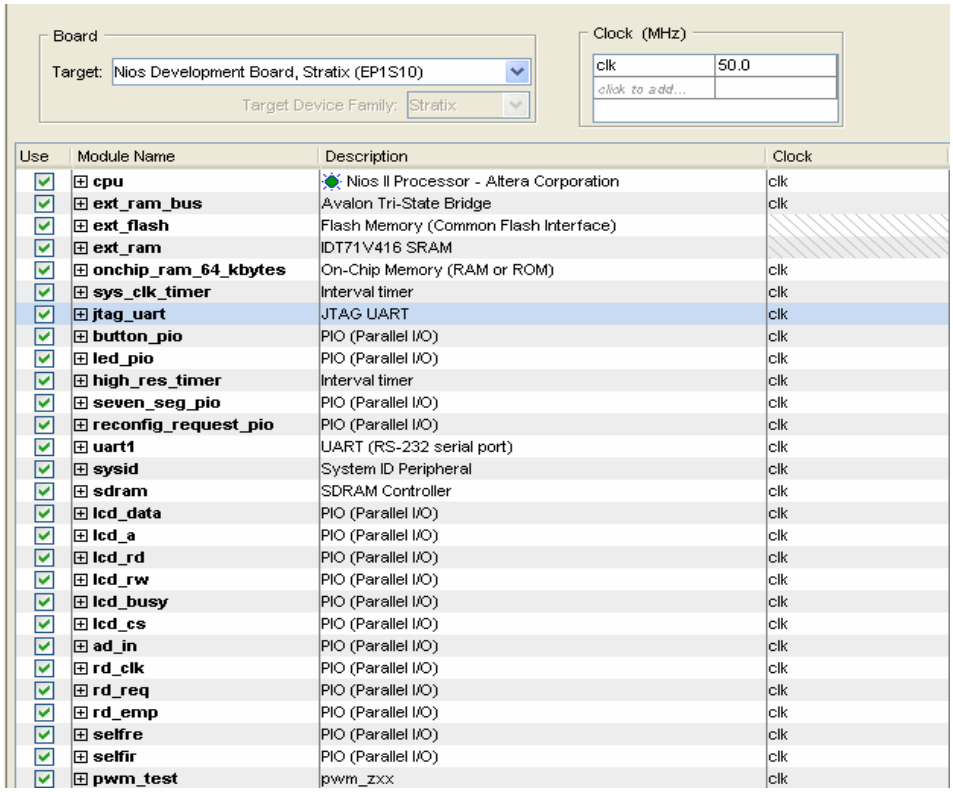


Figure 8 shows the integrated IP modules of SOPC Builder.

Figure 8. Integrated IP Modules



Use	Module Name	Description	Clock
☑	cpu	Nios II Processor - Altera Corporation	clk
☑	ext_ram_bus	Avalon Tri-State Bridge	clk
☑	ext_flash	Flash Memory (Common Flash Interface)	
☑	ext_ram	IDT71V416 SRAM	
☑	onchip_ram_64_kbytes	On-Chip Memory (RAM or ROM)	clk
☑	sys_clk_timer	Interval timer	clk
☑	jtag_uart	JTAG UART	clk
☑	button_pio	PIO (Parallel I/O)	clk
☑	led_pio	PIO (Parallel I/O)	clk
☑	high_res_timer	Interval timer	clk
☑	seven_seg_pio	PIO (Parallel I/O)	clk
☑	reconfig_request_pio	PIO (Parallel I/O)	clk
☑	uart1	UART (RS-232 serial port)	clk
☑	sysid	System ID Peripheral	clk
☑	sdram	SDRAM Controller	clk
☑	lcd_data	PIO (Parallel I/O)	clk
☑	lcd_a	PIO (Parallel I/O)	clk
☑	lcd_rd	PIO (Parallel I/O)	clk
☑	lcd_rw	PIO (Parallel I/O)	clk
☑	lcd_busy	PIO (Parallel I/O)	clk
☑	lcd_cs	PIO (Parallel I/O)	clk
☑	ad_in	PIO (Parallel I/O)	clk
☑	rd_clk	PIO (Parallel I/O)	clk
☑	rd_req	PIO (Parallel I/O)	clk
☑	rd_emp	PIO (Parallel I/O)	clk
☑	selfre	PIO (Parallel I/O)	clk
☑	selfir	PIO (Parallel I/O)	clk
☑	pwm_test	pwm_zxx	clk

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.

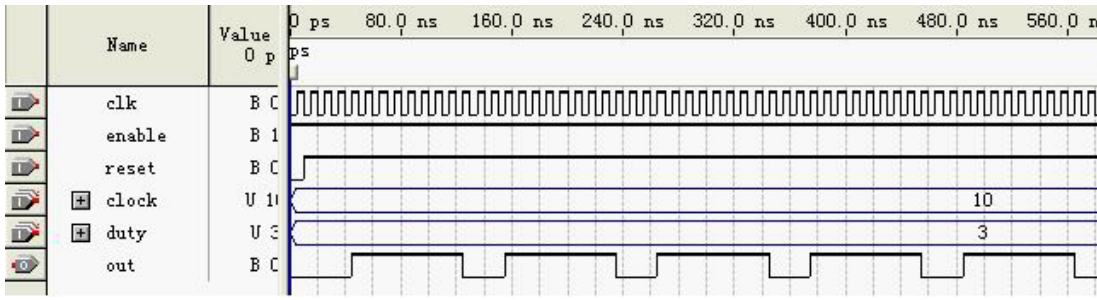
Here is the Verilog HDL program source code for programming control cycle and pulse width.

```
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:

$$N=(-1)\ s*M*2E-'z' \qquad (5.1)$$

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:

$$M=1+m2\}2-'+mz\}2-z+m\}o2-'\ \}\ \} - mot \qquad (5.2)$$

m in the formula indicates the i^{th} 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

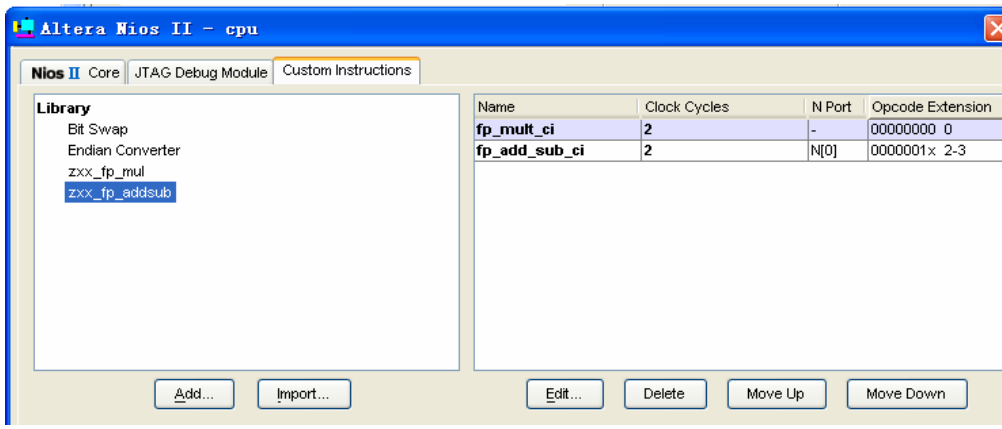
According to IEEE754 standard, the algorithm of single precision floating-point number multiplication can be de divided into sign bit calculation, index bit calculation, and remainder multiplication calculation. The first two are easy to realize. The sign bit is two multiplicands sign bit; the index 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

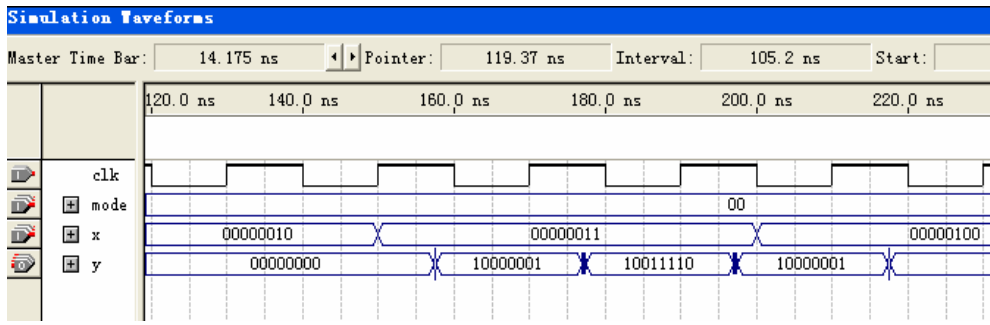
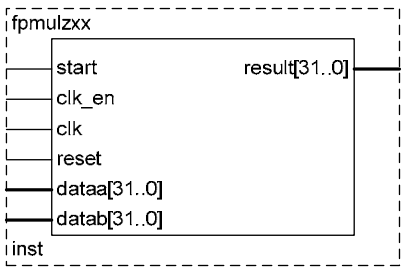


Figure 12 shows the symbol generated by the system.

Figure 12. Symbol



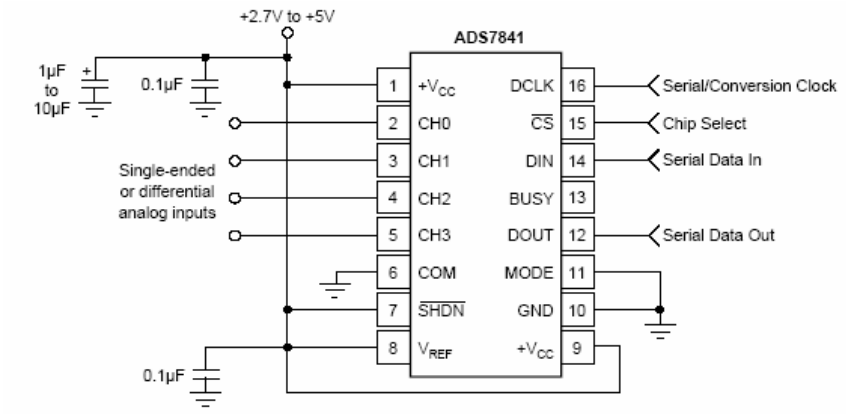
Design of A/D Sample Controller

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	

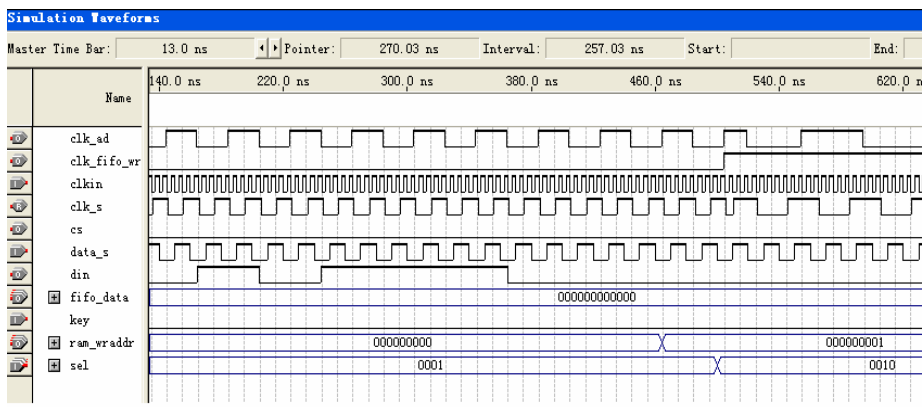
The CHO channel can be selected for sample, therefore, A2=0, A1=0, A0=1 and SGL/ $\overline{DIF}$ =1 should be set in control word input from DIN. In order to output 12-bit data after conversion, the MODE pin out should be made low. In order to ensure normal operation of the ADS7841 device during data 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.

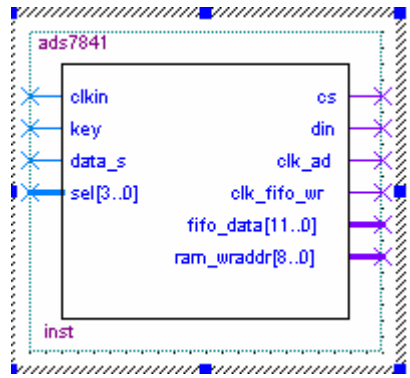
To effect changes in sample frequency, we designed a 4-bit control port that receives the control word sent from Nios II soft-core processor. Then, based on the received control word, the A/D controller changes sample frequency.

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;
```

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

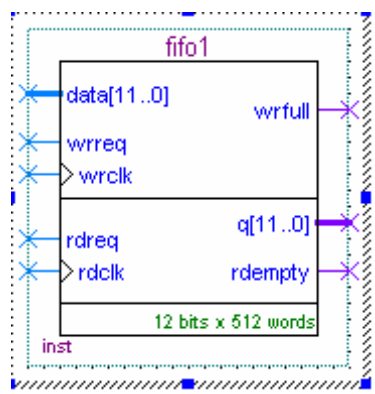
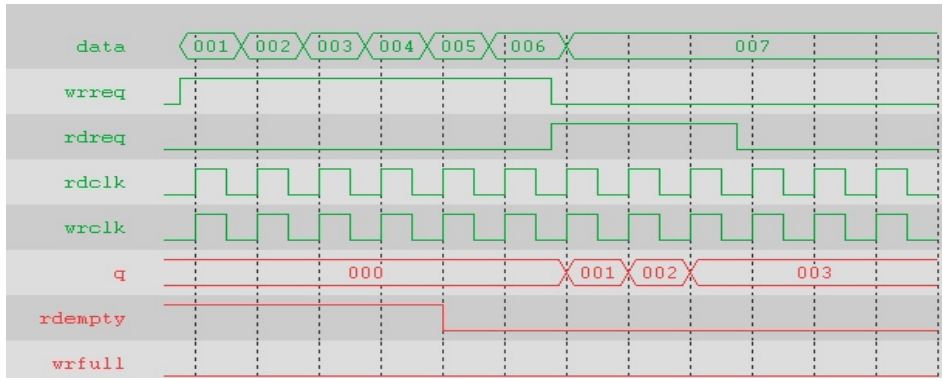


Figure 17 shows the FIFO time sequence.

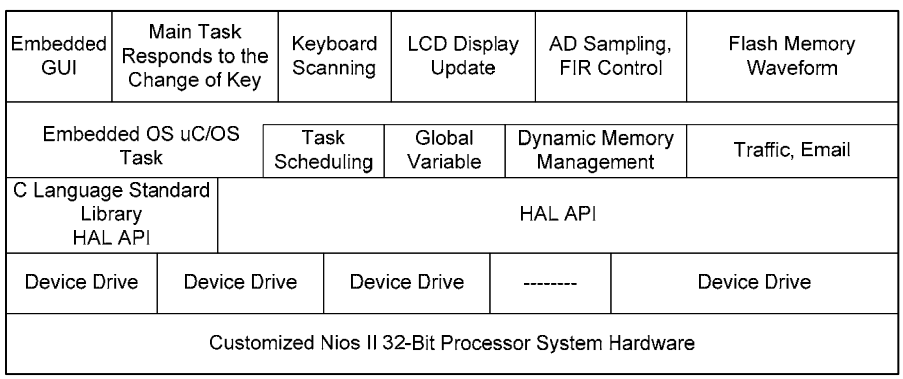
Figure 17. FIFO Time Sequence



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:

```
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:

```
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:

```
 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:

```
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.

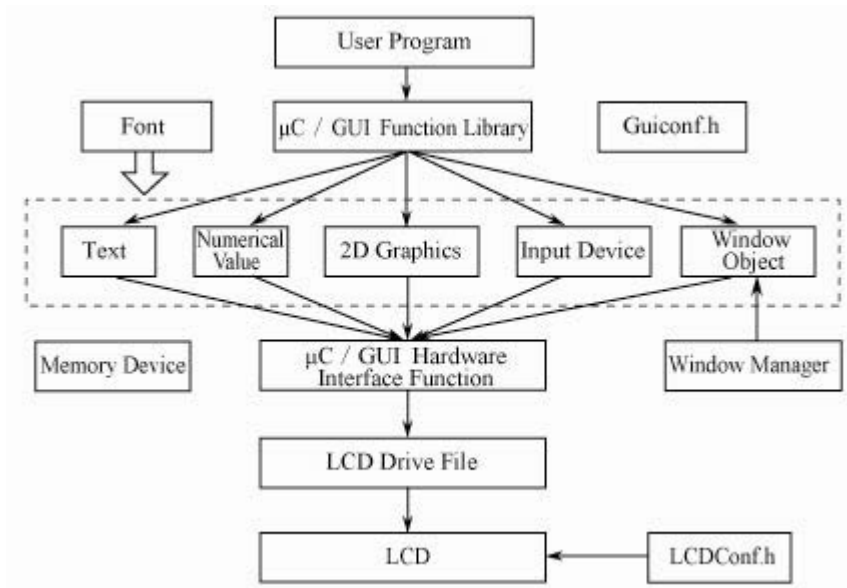
Although µC/GUI can greatly reduce the difficulty of LCD display tasks in embedded systems, we need to develop separate driver programs to handle LCDs with screens of different resolution.

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:

$$X(k) = DFT[x(n)] = \sum_{n=0}^{N-1} x(n) W_N^{nK} \quad , \quad n \le k \le N-1 \qquad (5.3)$$

$$x(k) = IDFT[X(n)] = \frac{1}{N} \sum_{n=0}^{N-1} X(n) W_N^{-nK} \quad , n \le k \le N-1 \qquad (5.4)$$

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 $X(k)$)($n \le k \le N-1$) 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 $X(k)$ ($n \le k \le N-1$) 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:

$$\begin{cases} e(r) = x(2r) \\ f(r) = x(2r+1) \end{cases} \quad 0 \le r \le N-1 \qquad (5.8)$$

DFT of N point is written as:

$$X(k) = \sum_{r=0}^{N/2-1} x(2r) W_N^{2rk} + \sum_{r=0}^{N/2-1} x(2r+1) W_N^{(2r+1)k}$$

$$= \sum_{r=0}^{N/2-1} x(2r) W_{N/2}^{rk} + W_N^k \sum_{r=0}^{N/2-1} x(2r+1) W_{N/2}^{rk}, \quad n \le k \le N-1 \tag{5.9}$$

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,

$$\begin{cases} E(k) = \sum_{r=0}^{N/2-1} e(r) W_{N/2}^{rk} \\ F(k) = \sum_{r=0}^{N/2-1} f(r) W_{N/2}^{rk} \end{cases} \quad 0 \le k \le N/2-1 \tag{5.11}$$

$E(k)$ and $F(k)$ are the result of DFT of N/2, it is known from the character of DFT:

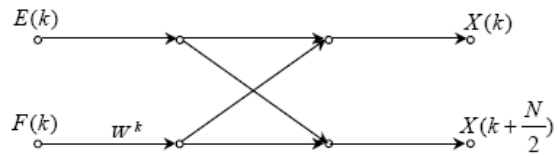
$$\begin{cases} E(k) = E(k + \frac{N}{2}) \\ F(k) = F(k + \frac{N}{2}) \end{cases}$$

Thus, this formula is written as follows:

$$\begin{cases} X(k) = E(k) + W_N^k F(k), \\ X(k + \frac{N}{2}) = E(k + \frac{N}{2}) + W_N^{(k+\frac{N}{2})} F(k + \frac{N}{2}) \\ \qquad = E(k) - W_N^k F(k) \end{cases} \quad 0 \le k \le N/2-1 \tag{5.12}$$

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

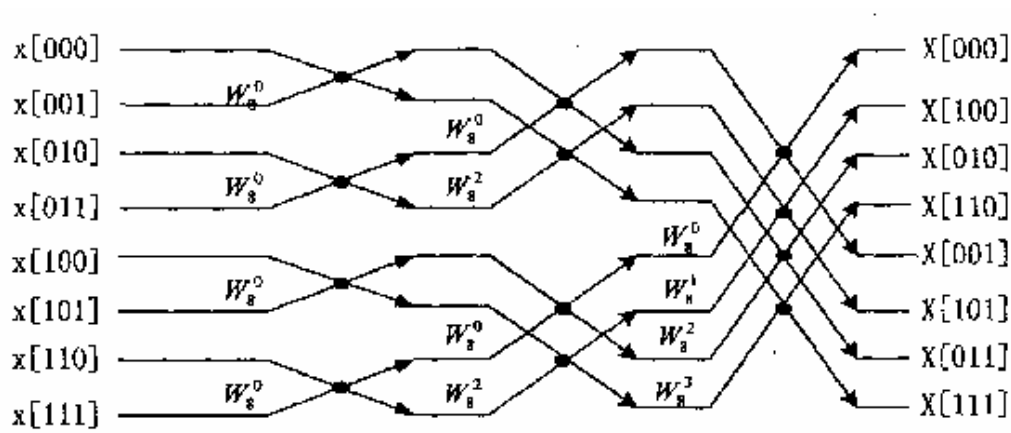
From the analysis above, $N = 2^M$ is divided into M levels, of which every level contains N/2 butterfly operation, so the number of total butterfly operations is N/2 x M.

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:

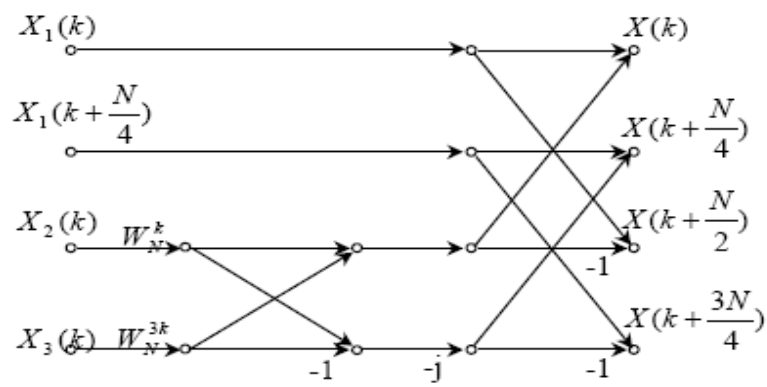
$$X(k) = \sum_{l=0}^{3} W_N^{lk} \sum_{n=0}^{N/4-1} x(4n+l) W_{N/4}^{nk} \tag{5.13}$$

make respectively $k = 4r, k = 4r + 2, k = 4r + 1¼k = 4r + 3$,and r=0,1,…,4/N-1,According to the formula (5.13), as follows:

$$\begin{aligned}
X(4r) &= \sum_{n=0}^{\frac{N}{4}-1} [(x(n) + x(n+\frac{N}{2}) + (x(n+\frac{N}{4}) + x(n+3\frac{N}{4}))] W_{N/4}^{nr} \\
X(4r+2) &= \sum_{n=0}^{\frac{N}{4}-1} [(x(n) + x(n+\frac{N}{2}) + (x(n+\frac{N}{4}) + x(n+3\frac{N}{4}))] W_N^{2n} W_{N/4}^{nr} \\
X(4r+1) &= \sum_{n=0}^{\frac{N}{4}-1} [(x(n) + x(n+\frac{N}{2}) - j(x(n+\frac{N}{4}) - x(n+3\frac{N}{4}))] W_N^{n} W_{N/4}^{nr} \\
X(4r+3) &= \sum_{n=0}^{\frac{N}{4}-1} [(x(n) + x(n+\frac{N}{2}) + j(x(n+\frac{N}{4}) - x(n+3\frac{N}{4}))] W_N^{3n} W_{N/4}^{nr}
\end{aligned} \tag{5.14}$$

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.]

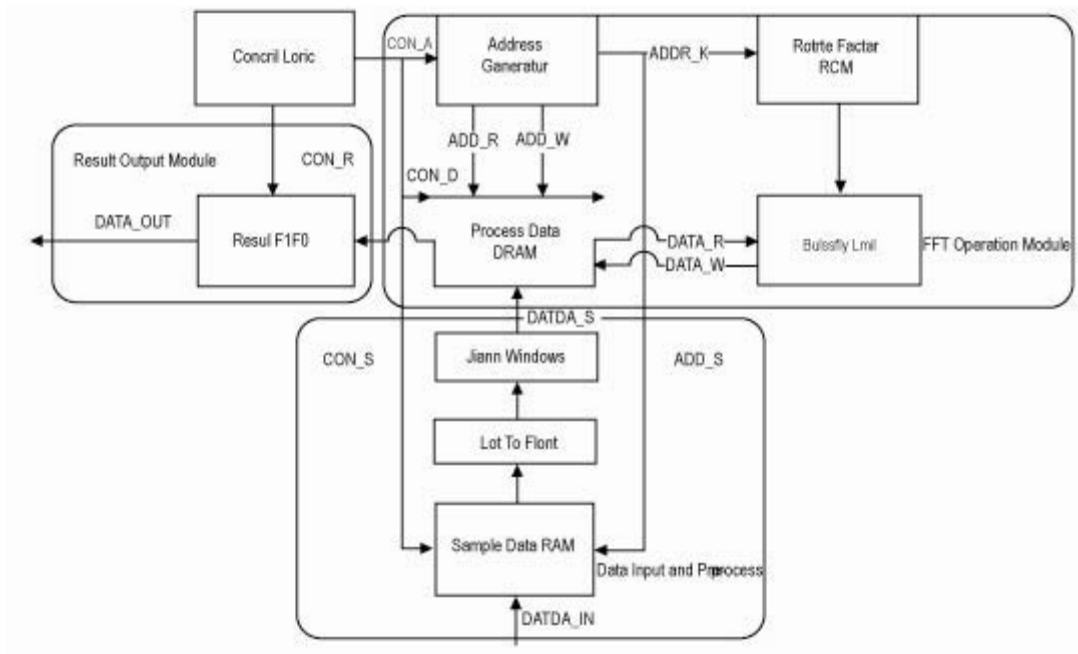
FFT Cell Design

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.

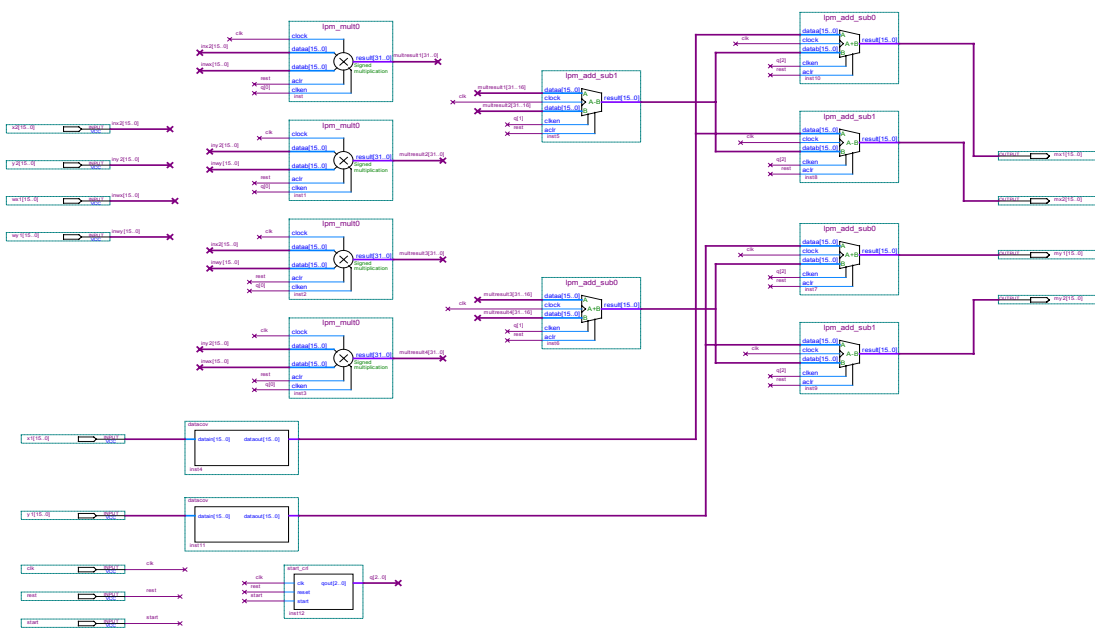
Figure 23. FFT Cell Architecture



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.

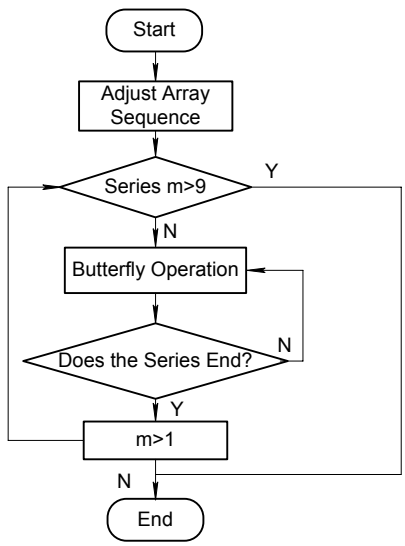
Some Issues with Hardware

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;
```

```
        } 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 is used, but $W_N^{nk} = e^{-j\frac{2\pi}{N}nk} = \cos(\frac{2\pi}{N}nk) - j\sin(\frac{2\pi}{N}nk)$ 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:

$$\hat{G}_{xx}(f_k) = \frac{2}{n_a T}\sum_{i=1}^{n_a} X_i(f_k)X_i^*(f_k)$$

$$\hat{G}_{xy}(f_k) = \frac{2}{n_a T}\sum_{i=1}^{n_a} Y_i(f_k)X_i^*(f_k) \qquad (5.16)$$

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

$$w_R(n) = \begin{cases} 1, & n = 0,\ 1, \ldots, N-1 \quad \text{»ò } n = -\dfrac{N}{2}, \ldots -1, 0, 1, \ldots, \dfrac{N}{2} \\ 0, & \text{ÆäËü} \end{cases} \tag{5.17}$$

2. Hanning Window

$$w_H(n) = 0.5 - 0.5\cos(\frac{2\pi n}{N}), \quad n = 0,\ 1,\ \ldots,\ N\text{ - }1 \tag{5.18}$$

3. Flat Top Window

$$w_F(n) = w_R(n)[a_0 + 2\sum_{k=1}^{3} a_k \cos 2\pi kn], \quad n = 0, 1, \cdots, N\text{ - }1 \tag{5.19}$$

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.

```
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

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 %)