Third Prize

Multimedia Decoder Using the Nios II Processor

Institution: **Indian Institute of Science**

Participants: **Mythri Alle, Naresh K. V., Svatantra Singh**

Instructor: **S. K. Nandy**

Design Introduction

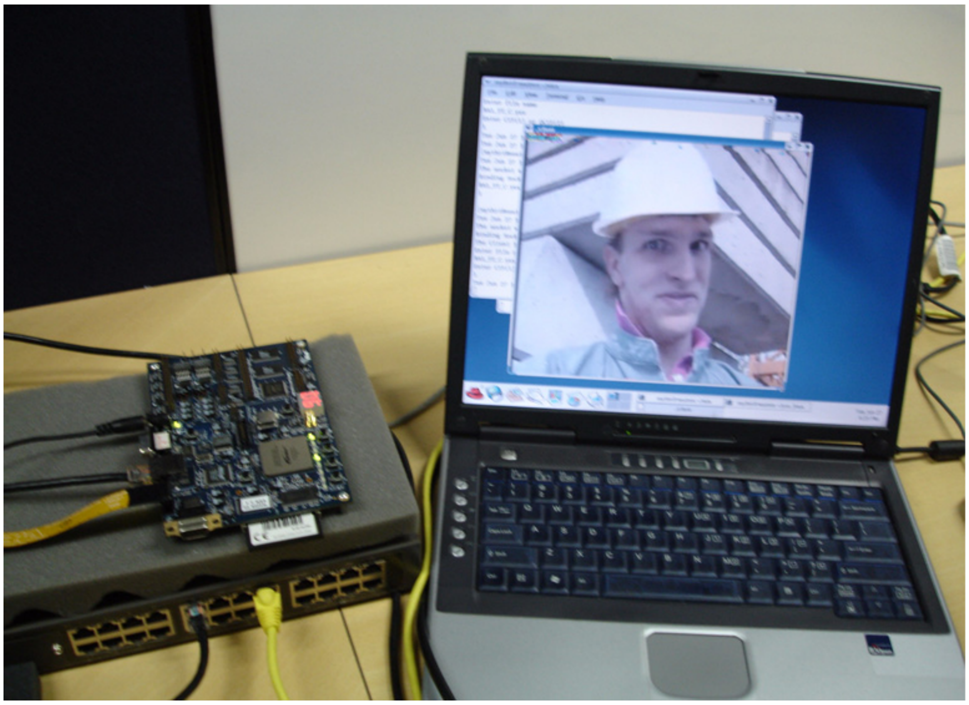
Our design target was to build a low-cost, high-performance H.264 decoder with a prototype H.264 decoder created using multiple small FPGAs. H.264 is a computationally complex, advanced video standard for achieving high compression ratios. To cater to the needs of high throughput applications such as HDTV, which requires 216,000 macroblocks/second of throughput, designers usually advocate a hardware implementation. Our proposed design is targeted at consumer electronics such as digital cameras or video players. Specifically:

- We created a novel display mechanism that is driver independent. The display output is in luminance-bandwidth-chrominance (YUV) format, which allows us to use any player that plays YUV data for any frame size.
- The design meets high throughput requirements, even when operating at 50 MHz. This speed is possible because we designed the application to use as few as 250 clock cycles to decode one macroblock.
- Operating at low frequencies enabled us to use multiple small, low-cost FPGAs to implement the decoder.
- The design is partitioned into multiple modules. The modules form different stages of the decoder pipeline.

- The proposed decoder implementation outputs the decoded frame in YUV format. We use a software player that can play streaming YUV data to display the video sequence.
- Our design is intended to meet the performance requirements of the main profile of H.264. Because we do not have an implementation of B-frame prediction, we are limited to a baseline profile.

Figure 1 shows the implementation setup.

Figure 1. Implementation Setup



High-Level Implementation Overview

The high-level implementation involved the following items:

- The H.264 decoder is added as custom user logic to the Nios® II processor.
- The user logic and Nios II processor communicate using the Avalon® bus.
- The user logic is an Avalon slave.
- The Avalon bus width is configurable and is set to 32 bits.
- To meet real-time video transfer from the FPGA, high-speed communication is essential. Therefore, we used the Avalon burst mode for the data transfer.

- For good-quality video, we used Ethernet cables to provide high-speed communication with external devices.
- We implemented Reference Picture Management, a non-compute-intensive part of the H.264 decoder, in software.

Why Use the Nios II Processor?

We used the Nios II processor for the following reasons:

- It was easy to load µClinux into the Nios II processor, letting us use many devices.
- It provided a high-speed communication channel using sockets to transfer real-time video.
- It was easy to combine FPGAs using an Ethernet cable and socket programming.
- The Avalon bus burst mode allowed fast transfer of data to custom logic.
- It efficiently integrated the hardware and software portions of the application.
- We could run parallel processes with µClinux using the clone system call.
- Because the peripherals are memory mapped, it was very easy to perform input/output for the custom peripherals using simple C programs.

Function Description

H.264 is a video coder/decoder (CODEC) standard defined jointly by Video Coding Experts Group (VCEG) and ISO/IEC Moving Picture Experts Group (MPEG). It is aimed at achieving high-resolution video at low bit rates.

In H.264, video is compressed by exploiting similarity in video frames. The current frame is predicted from previously encoded frame(s). The frame can be predicted from other parts of the same frame by exploiting spatial similarity (intra), or it can be predicted from a different frame by exploiting temporal similarity (inter). Residual data is the difference between the actual frame and the predicted frame. This residual data is further quantized and transformed to compress it efficiently. A loop deblocking filter removes the blocking artifacts. The user selects the parameters and the residual data is encoded using different entropy encoders and is sent as an encoded bitstream. All of the units are computation-intensive. The following sections describe each module and its implementation.

High-Level Decoder Design

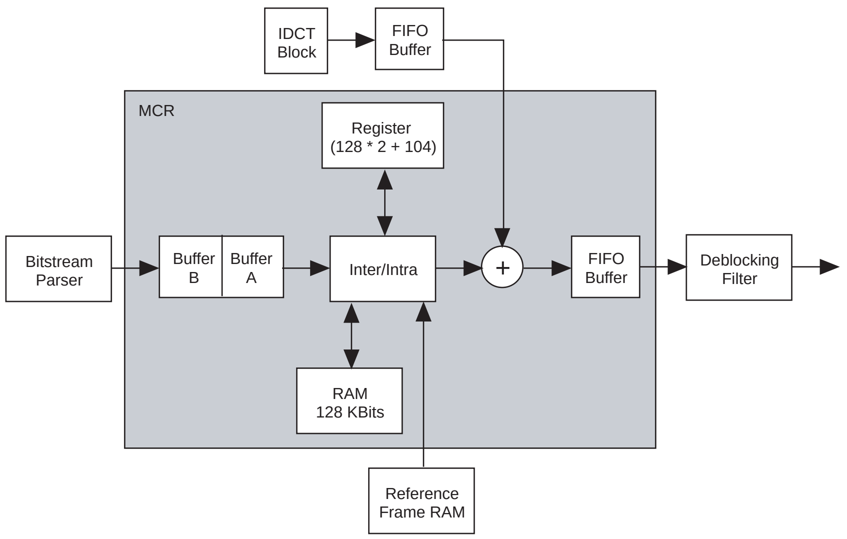
We implemented a macroblock-level pipeline to achieve high throughput. The encoded bitstream contains data per macroblock. Therefore, the macroblock-level pipeline provides an elegant interface between the parser and the rest of the decoder. Figure 2 shows the parser's high-level block diagram.

The parser processes the incoming bitstream. The motion compensation and reconstruction (MCR) module takes the prediction modes and its parameters as the input from the bitstream parser.

The residual data obtained after inverse discrete cosine transform (IDCT) serves as input to the MCR module. Residual data is added to the predicted frame and is then computed by the MCR module before it passes the data to the deblocking filter.

The communication between the IDCT and MCR modules is through a FIFO buffer. When the FIFO buffer is full, the IDCT stalls; when it is empty, the MCR stalls, ensuring synchronization. The parser and IDCT modules share two buffers in a mutually exclusive manner. The buffer access is switched for every macroblock. The MCR and deblocking module also synchronize through a FIFO buffer.

Figure 2. Parser High-Level Block Diagram



Module Descriptions

The following sections provide a short summary of each module.

Parser

The decoding first step is to parse the encoded bitstream. In the baseline profile, H.264 uses context-adaptive variable-lenght coding (CAVLC) and exponential-Golomb (Exp-Golomb) entropy encoders to encode various syntax elements. Both encoding schemes are variable length codes and the length of the encoding element is not in multiples of bytes. Using variable length codes makes the bitstream processing inherently sequential. To achieve high throughput, the entropy decoders must be efficiently implemented.

The parser buffers the incoming bitstream and provides a byte-aligned bitstream for different computing elements. The number of bits required to decode a syntax element is not a multiple of eight, which means data can be read from any position.

To avoid the overhead of a multiplexer, we use two-level buffering. The first-level buffer is circular and caches the incoming bitstream. Data is read and written in multiples of bytes. We use a read/write pointer to keep track of the data. We ensure that the buffer is never full by making the processing rate greater than maximum incoming bit rate. Therefore, w**e** do not have to maintain additional data to differentiate between full and empty conditions. A small second-stage buffer reads data from the first-stage buffer. The second stage serves data for processing. The data is also byte-aligned before it goes to various computing elements.

The entropy encoders are implemented very efficiently and take less area and clock cycles compared to the best implementation described in the literature we have read.

IDCT

This module performs an inverse integer discrete cosine transform. Unlike most other standards, IDCT is performed for each 4 x 4 block, which makes it a computationally intensive module.

We implement this module using a five-stage pipeline. We used a novel approach to split it into pipeline stages. Four pixels are inverse-transformed and de-quantized for every clock cycle. This module takes residual data as input from the parser.

MCR

This module predicts the frame using the previously decoded frames, depending on the parameters from the parser. This module contains two kinds of prediction, inter and intra prediction.

Inter Prediction

H.264 allows the capture of very fine motion at quarter-pixel resolution using a six-tap finite impulse response (FIR) filter to interpolate accurate sub-pixel values. H.264 also let the designer choose different block sizes for specifying the motion. The block size can be 16 x 16, 16 x 8, 8 x 16, 8 x 8, 8 x 4, 4 x 8, and 4 x 4. The encoded bitstream contains the reference frame number, motion vector, and size of each block.

Inter prediction is both compute- and memory-intensive. A six-tap FIR filter interpolates the reference picture pixels to obtain the predicted frame. H.264 also allows quarter-pixel granularity, which requires computation of the half pixel first and the quarter pixel second, which requires more memory.

Inter prediction has three units: a fetch unit, a compute unit, and a cache. These units function in parallel. In our design, we assigned a maximum of nine clock cycles for the fetch unit and four clock cycles for the compute unit. A cache buffers the data fetched from the reference frame to reduce the memory bandwidth required. The fetch unit takes the reference frame and the motion vector as inputs, and computes macroblock addresses required for the prediction of a 4 x 4 block. It fetches the blocks that are not available in the cache. A block is fetched only if the cache entry is free. We do not perform out-of-order fetching because it complicates the control logic. The compute unit performs the interpolation using a six-tap FIR filter. This unit contains a nine-stage pipeline. Nine stages are required because of the quarter-pixel interpolation. The first four pixels perform half-pixel interpolation. The remaining stages perform quarter-pixel interpolation.

Intra Prediction

Intra prediction is more control-intensive. Depending on the prediction type, different computations are performed on the data.

Intra prediction has three phases. The first phase is the initialization phase, where the data required for the intra prediction is obtained. The second phase, called the compute phase, uses the data fetched by the first phase and performs the computation. In the final phase, data is passed on to a deblocking filter. Intra prediction retains the required pixels that are not passed through the deblocking filter for prediction of other blocks.

Deblocking Filter

The deblocking filter removes blocking artifacts, which improves the subjective quality of the picture. This module is the most computationally expensive block, which affects the entire design’s processing rate. This block also requires a lot of memory.

We implemented this block using an eight-stage pipeline. The pipeline processes luma and chroma components. A read/write counter controls the pipeline efficiently. We used a novel transpose buffer, which allows us to start vertical processing immediately after horizontal processing. In the existing implementations, vertical processing begins only after the entire pipeline is flushed.

Performance Parameters

The performance targets for our design were:

- Use high communication bandwidth for real-time video display. We needed 100 megabits per second (Mbps) to display high quality video in real time.
- Implement the design using the smallest possible area to provide low-cost solutions.

- Provide high throughput, even when operating at low frequencies. This requirement meant we needed to use fewer clock cycles.
- Operate at low frequencies so that the design works with low-cost boards.
- Partition the functionality efficiently so that communication between these partitions is reduced. This method enables the use of multiple small FPGAs.

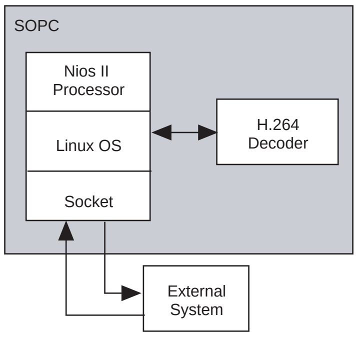
We achieved the following performance:

- The design operates at 50 MHz and meets the throughput requirements needed for large frames.
- The modules use the following numbers of logic elements (LEs) on a Stratix® II FPGA:
  - Parser—40,000 LEs
  - IDCT—2,000 LEs
  - MCR—59,000 LEs
  - Deblocking—17,000 LEs
- A maximum of 250 clock cycles is required to decode one macroblock.
- We can separate our design into various partitions. The communication between the partitions is limited and therefore it does not add overhead to the partitioning,
- Communication is through an Ethernet cable, which means the design can achieve 100 Mbps.

Design Architecture

H.264 baseline decoder is implemented in hardware. The decoder design is described earlier in this paper. The communication to and from the hardware is performed through the Nios II processor. Figure 3 shows a block diagram of the architecture.

Figure 3. Architecture Block Diagram



µClinux is loaded onto the Nios II processor, which allows us to use sockets for communication. Using sockets, the design performs high-speed data transfer that is sufficient for real-time video. The input is the encoded bitstream and the output is a decoded stream in YUV format. The output bit rate is much higher than the input bit rate. We use two sockets for communication, one for the input stream and one for the output stream.

The data exchange between the Nios II processor and the hardware is through the Avalon bus. We programmed our decoder as an Avalon slave. The bus is 32 bits wide, therefore reads and writes are 32 bits wide. We used a C application to read and write data from the hardware. We write three bytes of data and one byte of control information, which is used for synchronization. During reads, the last byte indicates control information.

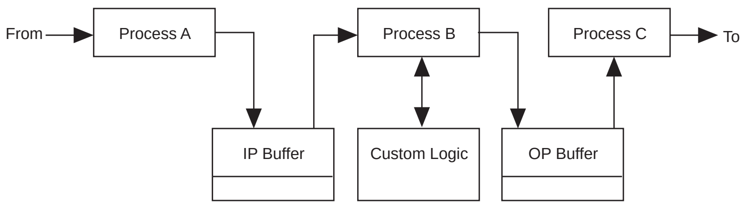
Software

Three parallel processes run in software.

- One process reads the input stream from the socket and stores it in memory.
- The second process reads from memory and provides data as the input to the custom logic. When the output of the custom logic is ready, it reads the output and writes it into a buffer. We did not make reading and writing two independent processes, because we know the approximate rate of input and output. For every two outputs from custom logic, we perform one write to the input of the custom logic. Because the peripherals are memory mapped, it is very easy to provide input and output to the custom logic through a C program.
- The third process reads data from the output buffer and writes to the socket.

Figure 4 shows the flow.

Figure 4. Software Process Flow



Design Description

This section describes the design.

Custom Logic Implementation

The H.264 decoder is added as custom logic to the Nios II processor. The decoder and Nios II processor communicate using the Avalon bus. The bus width is 32 bits. The Avalon bus requires a particular handshaking protocol for communication. We used following signals for the handshaking protocol.

- `Clk`—Clock input
- `CS`—Chip select
- `Reset`—System-wide reset
- `Wr`—Write signal, specifies input is ready
- `Din`—Data input, 32 bits
- `Rd`—Read signal
- `Dout`—Data output, 32 bits

The custom peripheral is an Avalon slave.

Synchronization of I/O

The input and output to the custom logic is 32 bits wide. 3 bytes contain data and 1 byte contains control information, which is used for synchronization. The least significant bit of the control byte indicates if the data is valid for both input and output. One more bit is used as toggle bit, which is toggled on every read (write) for input (output). This bit ensures that we are not reading old data. The input data is the encoded data and the output data is the decoded video stream. Therefore, the output rate is much higher than the input rate. This I/O is performed by a C program running on the Nios II processor.

Burst Mode

To meet our real-time video data processing requirement, we used the Avalon bus burst mode to achieve high bandwidth. Burst mode transfers are only supported in the Nios II processor version 6.0 and higher.

Combining FPGAs

The H.264 decoder is complex hardware that requires multiple small FPGAs for implementation. Because we have only one Development and Education (DE2) board, we could only implement and send a demo for the deblocking filter. The process for implementing the design in multiple FPGAs is described below:

- We separated the H.264 decoder into multiple partitions such that the communication bandwidth required between the partitions is limited.
- The partitions communicate using the Ethernet.
- The modules operate in a pipeline.
- Each pipeline module can be implemented in one FPGA.
- Because they operate in a pipeline, the communication between each module is well defined.
- We include a mechanism to stall the pipeline when data is not available.

These features provide a very economical mechanism to synthesize an H.264 decoder that uses fewer than 80,000 LEs. An 80,000-LE FPGA costs approximately $8,000 USD. With partitioning, the decoder can be implemented using four DE2 boards, which costs about $1,200 USD, a gain of 85%. This cost savings is a very important feature of the design and has huge business impact.

Parallel Processes & Synchronization

The input/output to the external world is handled effectively using μClinux loaded in the Nios II processor. Three processes can be run on μClinux using the clone system call. Because the processes share memory, a synchronization mechanism should be used to avoid race conditions.

The buffer shared between the processes has two parts. When a process reads from one partition, the other part writes into another partition. The writing partition writes all 0s in the first four bytes of the partition before writing data to the partition. The reading process writes all 1s in the first four bytes after it finishes reading a partition. The writing process waits for all 1s before it writes data, which ensures that it is not overwriting the data before the process reads the data. The reading process waits for all 0s before reading data to ensure that it is not reading the stale data. We tested this mechanism for various video sequences and it works correctly and does not consume much overhead. Using high-level synchronization mechanisms such as semaphores would have added a lot of overhead.

Design Features

This section describes the design features.

Hardware/Software Partitioning

In our design, the external interaction is performed via software and the actual processing is performed in hardware. The system-on-a-programmable-chip (SOPC) methodology allows us to execute this process easily. The software part is executed on µClinux loaded in the Nios II processor. The processing unit is implemented in Verilog HDL and is added as a custom peripheral to the Nios II processor. SOPC Builder is very efficient and allows the user to add logic easily: we can add only the required peripherals, limiting the LE overhead required.

High-Speed Communication

It is essential to have high-speed communication to provide the user with a high-quality real-time video experience. Using the µClinux Ethernet driver makes it possible to provide a communication medium of 100 Mbps. Loading µClinux onto the Nios II processor let us use a rich set of drivers available with µClinux. With the Ethernet driver, we can use socket programming. Therefore, communication with the external system is very easy and the board can also be accessed remotely.

Fast Data I/O to the Peripheral

It is essential to have fast I/O to the peripheral to achieve high throughput. Although the processing rate is high if the I/O to the peripheral is not fast, it forms a bottleneck. The Avalon bus burst mode allows us to meet the required rate.

Combining FPGAs

The decoder is complex hardware and requires as many as 100,000 LEs on a Cyclone® FPGA. It is very economical if we can combine multiple FPGAs to implement the entire design. We proposed a novel methodology (described in an earlier section). The Ethernet driver allows us to implement our methodology.

Memory-Mapped I/O

SOPC Builder uses memory-mapped I/O. The peripherals can be addresses as we address memory. The base address of a peripheral is available from the SOPC Builder user interface. The data can be read and written from the input and the output ports of the peripheral, as we would do with any memory location. It is very easy to access the peripheral I/O using a C application running on µClinux.

Device-Independent Display

We used a very flexible, driver-independant display mechanism. We send the decoded data stream as output; the decoded data stream is in YUV format. This format is a great advantage because any software player that can play a YUV file can be used to play the data stream. There are many streaming YUV players available, making it very easy to show a real-time demo. The YUV player takes the frame size as input and can play any frame size. We can demo any frame size without any change in the setup, unlike other formats. For example, for VGA, different data drivers must be used for different frame sizes. This option is not available in any of the existing setups we researched. The VGA has a single driver and can display only one frame size. In our proposed design we can display any frame size.

Conclusion

We drew the following conclusions from this project:

- SOPC Builder
  - SOPC Builder is an ideal platform to build applications that have both software and hardware parts.
  - SOPC Builder provides three variants of the Nios II processor.
  - The user can customize the processor's peripherals.
  - It provides full control of the base address and the interrupt controller (IRQ) numbers, if desired.
  - The auto assign mode assigns the addresses automatically, which should be sufficient for most uses. The SOPC Builder flow is user-friendly and is very easy to use.
  - Custom logic is very easy to add. The user must follow the proper handshaking protocol, as specified in the documentation. If proper signals are not present, it is not possible to add the logic.
- Nios II Integrated Development Environment (IDE)
  - It allows the user to build μClinux, the file system, and other C applications.
  - It is very easy to use.
  - It is slow compared to the Linux tool flow.
  - For the DE2 board, using the Nios II IDE to build Linux is very difficult. It is very easy to build it in Linux.
- Nios II Forum
  - The Nios II forum is very helpful for Nios II and μClinux related issues.
  - We received very prompt responses to our postings.
  - The forum are already has many postings, and we can often find a solution to our problem by searching through the postings.