Second Prize

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

Institution: **Seoul National University**

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

Instructor: **Ki-Young Choi**

Design Introduction

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

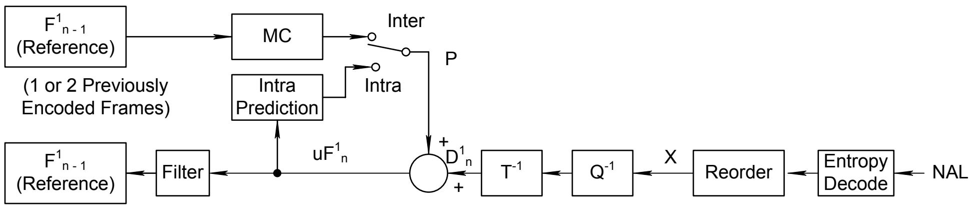
We started our design from Joint Model Reference code. Because the design requires a lot of computations with various sophisticated compression techniques, we needed a high-performance system for real-time video processing. We achieved the necessary performance for a reduced frame rate using the Nios®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

Function Description

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

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

Figure 1. H.264/AVC Decoder Block Diagram



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

Performance Parameters

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

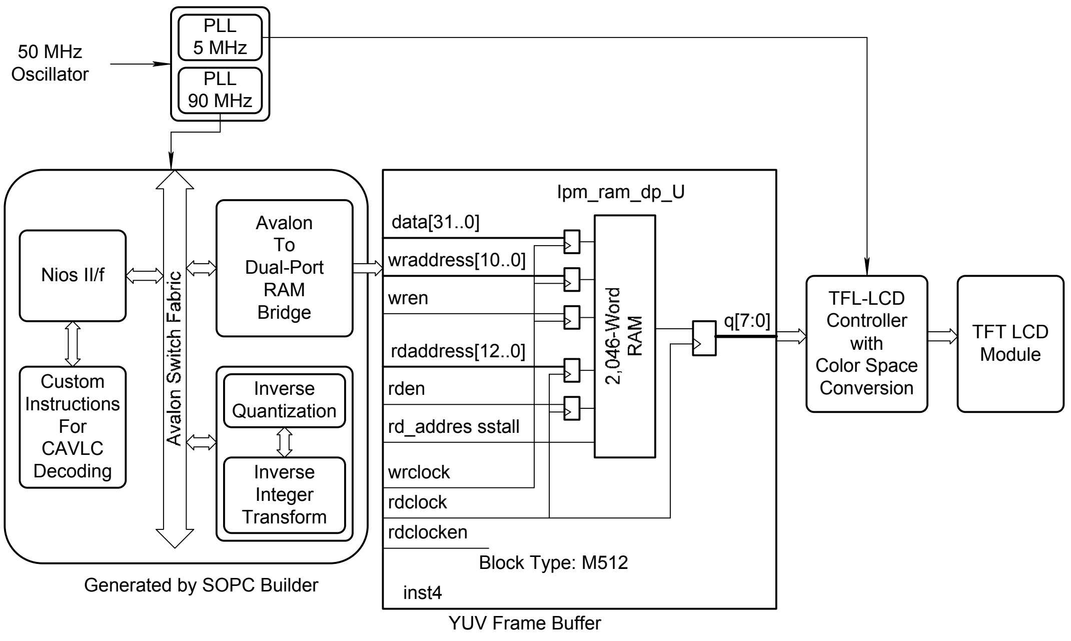
Design Architecture

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

Nios II Configuration & Memory Hierarchy

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

Figure 2. Block Diagram of Implemented H.264/AVC Decoder

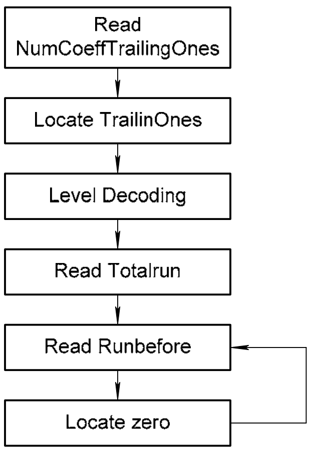


Tightly coupled data memory handles read-only data memory (.rodata), heap memory, and stack memory. Our design application uses about 16 Kbytes for read-only data, which stores frequently used coefficients. The remaining tightly coupled data memory is enough for the heap and stack. We managed to obtain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

Custom Instructions for CAVLC Decoding

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

Figure 3. CAVLC Decoding Process



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

Figure 4. Flow Chart of Each Block CAVLC Decoding

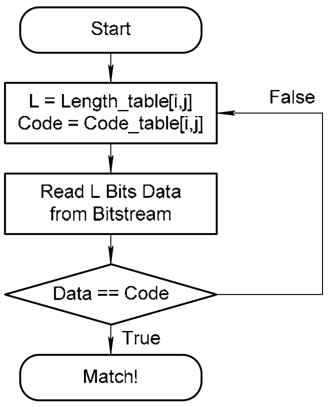
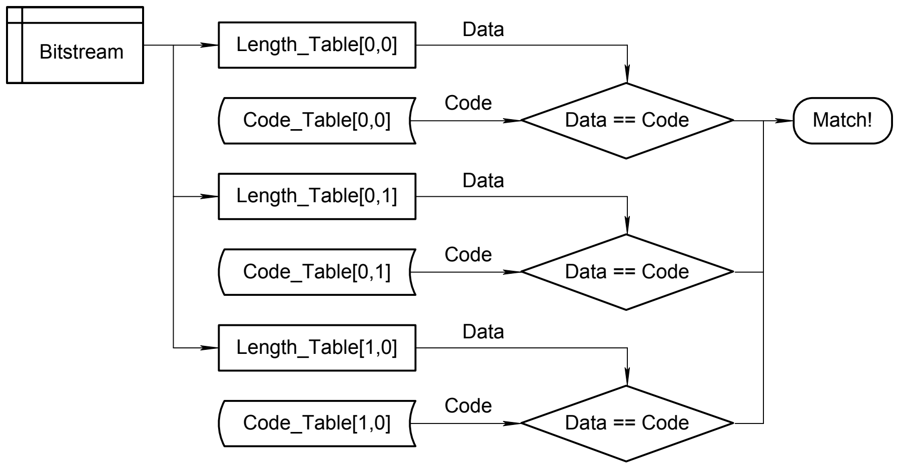


Figure 5 shows the implementation structure of each custom instruction for CAVLC decoding. By loop unrolling and parallel comparison we managed to maximize inherent parallelism and arrived at the exact match in 1 cycle. Using this custom instruction, we achieved about 13% increase in execution speed.

Figure 5. Custom Instruction Implementation of CAVLC Decoding



Inverse Quantization & Inverse Integer Transform[2]

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

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

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

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

Table 1. Rescaling Factor V

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

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

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

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

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

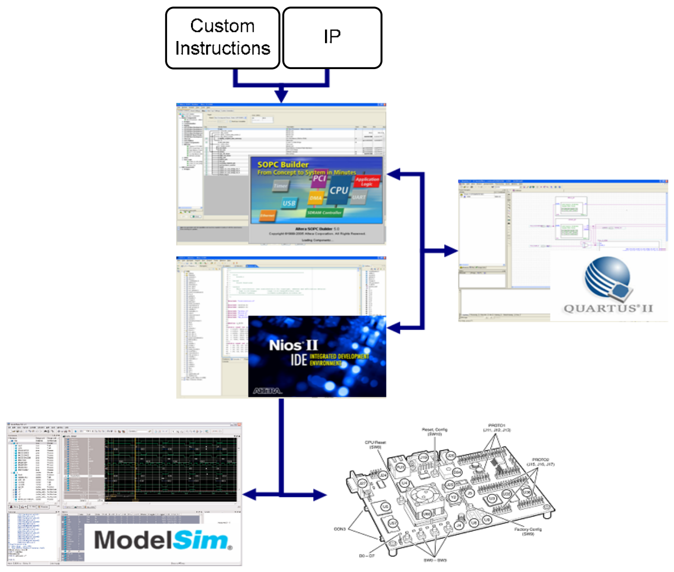
YUV Frame Buffers

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

Design Methodology

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

Figure 6. Overall Design Flow & Tools Used



Design Features

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

- Optimal configuration of memory design hierarchy and layout.
- Deployment of custom instructions for CAVLC decoding.
- Implementation of IP modules for Inverse Quantization and Inverse Integer Transform.
- Design of TFT LCD controller with YUV-to-RGB color space converter.

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

Conclusion

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