Using the NIOS II Processor for HW/SW Codesign of the JPEG2000 standard

Mike Dyer, Amit Kumar Gupta and Natalia Galin

1. Design Introduction

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

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

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

2. Function Description

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

- Lossless and lossy compression
- Region of interest (ROI) coding
- Compression of color and gray-scale images

(Please refer to the Kakadu documentation [2] for more details.)

Figure 1 JPEG2000 Encoding Flow

The compression system of the JPEG2000 algorithm is illustrated in Figure 1 JPEG2000 Encoding Flow. The image is compressed in the following steps:
1. Image samples are first separated into color components (if any)
2. Image color components are then optionally decomposed into rectangular tiles, with each tile to be compressed independently
3. Discrete Wavelet Transform is used to decompose each tile into four frequency subbands. JPEG2000-Part 1 specifies two wavelet kernels for lossy and lossless compression, 9/7 and 5/3 wavelet kernels respectively.
4. The output from the wavelet transform is then quantized and separated into rectangular `code-blocks', to be processed by EBCOT unit.
5. Each code-block is then processed independently by the Block Coder (BC). The BC may be sub-divided into Bit Plane Coder (BPC) and Arithmetic Coder (AC) modules. The BPC encodes a code-block in bit-plane by bit-plane order generating Context-Data (CxD) pairs. CxD pairs are then encoded by the AC module to generate the compressed bit stream.
6. Rate-distortion optimization selects optimal contribution of a code-block to the compressed bitstream for a given target bit-rate such that the reconstructed image has minimum distortion. Kakadu uses the Post Compression Rate Distortion (PCRD) optimization algorithm [3].
7. Finally markers are added to the output bit-stream to increase error resilience and packed into the JPEG2000 compressed bit stream

The DWT and BC are two most resource intensive components of JPEG2000. A detailed study and analysis is performed to determine the strategy to best partition JPEG2000 into software and hardware components to optimize the compression stream using the rich feature set of the NIOS2 processor. We made the following changes to Kakadu.

-- Use of custom instructions to implement DWT.

-- Implementation of EBCOT in hardware. BPC, AC and Distortion estimation module are implemented as hardware peripherals.

-- Use of RTOS ((µCos-II)) to instantiate multiple block-coders to increase throughput.

3. Performance Parameters

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

Property	Value
Image	café (ISO test image)
Image Dimensions	2560 × 2048
Image Format	pgm;8-bit samples
DWT kernel	CDF 9/7
DWT levels	5
Block Coder mode	Normal
Code-block size	64×64

Table 1 Test environment parameters

A. Profile Results

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

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

Figure 2 Kakadu profiling results

B. Block Encode Time

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

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

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

$$T_{BPC} = \frac{23\times10^{6}}{1.1} \cdot \frac{1}{50\times10^{6}} = 4.182\times10^{-4}\ \text{sec}$$

For each code-block, the internal code-block RAM must be loaded via DMA. This time will accumulate in systems that use multiple block coders. Thus,

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

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

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

The average time taken to code a code-block in a system can be given approximately by

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

It should be noted that this equation is only accurate while the utilization of the data bus is low. When the time spent performing DMA transfers is greater than to the time required to code a single code-block, the bus will become the limiting factor. This would indicate that the number of block coders should be $N \le \left\lfloor \frac{T_{BPC}}{T_{DMA}} \right\rfloor = 10$. Proper simulation is invaluable when determining the true value of N.

Since block coding accounts for 103.02 of the total 167.08 seconds used to compress the cafe test image (Figure 2 Kakadu profiling results) we expect that the implementation of BC in hardware should give the highest performance improvement. We also note that the block coder must process 1286 blocks. Table 2 Average block coding times for parallel block coders shows the average block coding time when using N parallel hardware block coders, and the time taken to code 1286 blocks.

The speedup factor using multiple block coders is shown in Table 3 Speed-up achievable with multiple block coders.

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

Table 2 Average block coding times for parallel block coders

N	1	2	3	4
Speedup	2.58	2.59	2.60	2.60

Table 3 Speed-up achievable with multiple block coders

C. DWT Flow

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

$$\text{speedup} = \frac{1}{(1-P) + \frac{P}{S}}$$

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

4. Design Architecture

The system structure is illustrated by Figure 3 JPEG2000 co-design configuration on Altera EP1S40 FPGA:

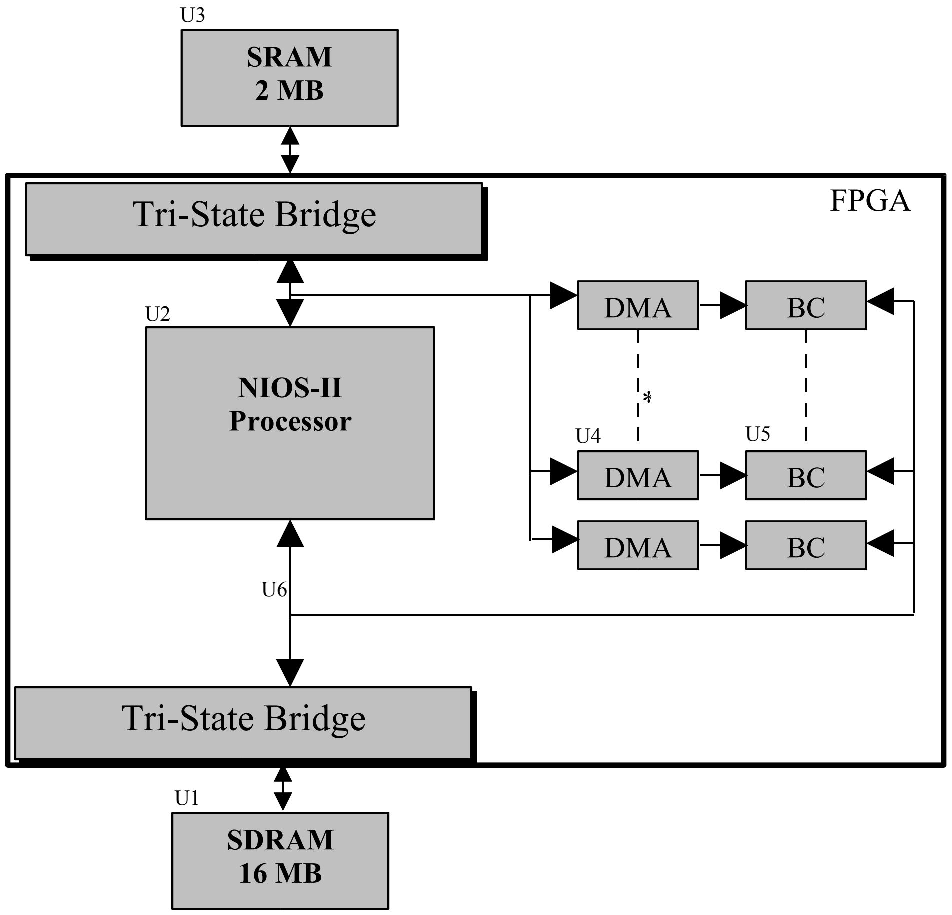

Figure 3 JPEG2000 co-design configuration on Altera EP1S40 FPGA

Where,
U1 -- 16Mbytes of SDRAM memory containing the modified Kakadu program and image to be compressed
U3 -- 2Mbytes of SRAM memory which is loaded with code block data as it is created by Kakadu
U2 -- NIOS II processor
U4 -- DMA controller configured to feed code block data at the rate of 16-bits/clock cycle to the BPC (U5)
U5 -- Block encoder hardware peripheral as described in detailed by Figure 4 Block coder detailed system configuration.
* - the number of DMAs and BPCs is determined by the available bandwidth on the Avalon bus
We have chosen to load the image directly onto the SDRAM so that the speed of our system is not limited by data transfer rates of external data IOs. In this way a fair comparison is made

between the baseline and our proposed system. In future it would be possible to integrate the system with a fast bus fabric which will not saturate the speed advantages provided by it.

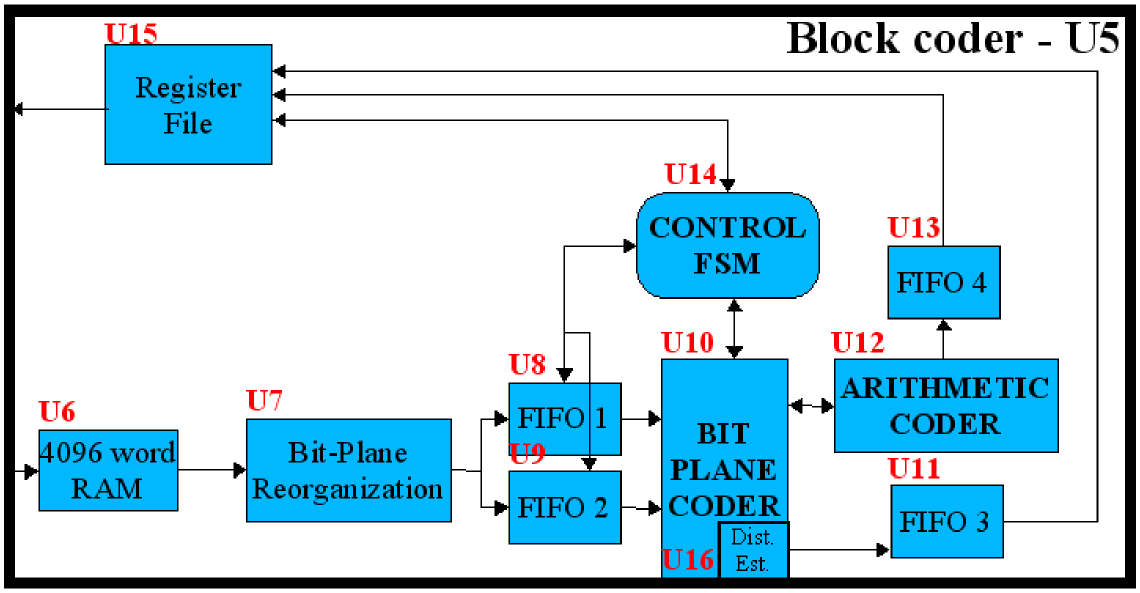

Figure 4 Block coder detailed system configuration

Figure 4 Block coder detailed system configuration shows a detailed block diagram for the block coder hardware peripheral, where:
U6 -- 4096 word RAM to buffer the code-block data (entire code-block if necessary)
U7 -- reorganizes subband samples in a bit-plane by bit-plane fashion to feed the BPC
U8 -- 16x4-bit FIFO to store the data bits from U7
U9 -- 16x4-bit FIFO to store the sign bits from U7
U10 -- BPC module (Section 6.C.II)
U11 -- 22x13-bit FIFO used to buffer distortion estimation data
U12 -- AC module (Section 6.C.III)
U13 -- 16x17-bit FIFO used to store compressed data from the AC module.
U14 -- Finite State Machine to control the information flow between the various components of the system
U15 -- Register file, containing 32x32-bit registers, 16 for read, and 16 for write, to interface between the BPC (U5) system and the control bus
U16 - Distortion estimation module (Section 6.C.IV)

5. Design Methodology

The HW/SW codesign of the JPEG2000 was carried through the following design steps:

1. Alteration to Kakadu to support multi-threading
2. Development of Bit-accurate software to verify the functionality of the proposed hardware peripherals.
3. Implementation of Hardware peripherals using HDLs.
4. Use of ModelSim to verify the hardware peripherals' functionality. The testbench vectors are generated using the bit-accurate software.

5. Use of Leonardo Spectrum tool to synthesize the hardware peripherals.
6. Use of Quartus development suite to produce the layout and timing analysis of the hardware peripherals.
7. Performing post-synthesis simulation in ModelSim using the Quartus timing results.
8. Load Kakadu software into the NIOS IDE and development of glue software to interface it to the hardware peripherals. Custom instructions also were added to Kakadu at this point.
9. Build and load of the Quartus project onto the FPGA.

In order to improve the performance of the DWT function in Kakadu, we augmented the instruction set in the NIOS processor with two new custom instructions (Section 6.A). The block diagram for the custom instructions is shown in Figure 5 NIOS arithmetic logic unit [4], and custom logic layout to perform a lifting step.

Figure 5 NIOS arithmetic logic unit [4], and custom logic layout to perform a lifting step

6. Design Features and Details

A. DWT-Custom Instructions

As can be seen from Figure 6 9/7 DWT lifting state machine of the state machine for the CDF 9/7 Lifting DWT implementation, the four lifting steps are very similar to each other, and the only difference being the value of the multiplier coefficient. Hence, it was clear that the DWT would best be implemented using the capability provided by NIOS of adding custom logic to its ALU unit. To perform a lifting step, two custom instructions were needed: one to augment two 16-bit samples into one, and one to perform the lifting step, as shown in Figure Figure 5 NIOS arithmetic logic unit [4], and custom logic layout to perform a lifting step.
Upon compilation, it was seen that the number of assembly instructions to perform the lifting step decreased from 5 (in the pure C implementation), to 2 (using NIOS custom instructions).

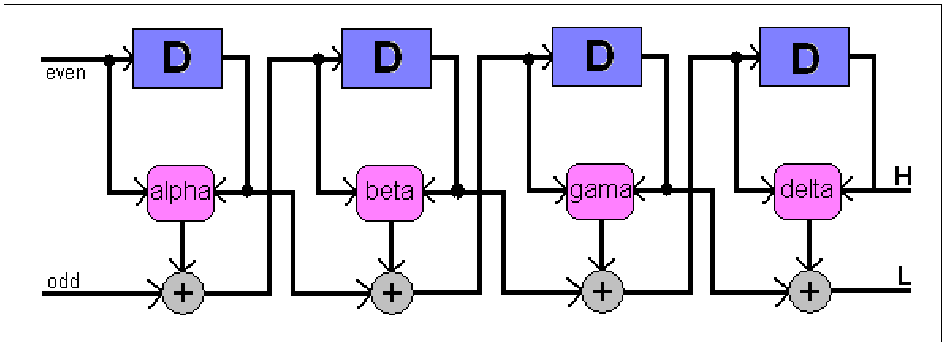

Figure 6 9/7 DWT lifting state machine

B. Multi-threading of Kakadu

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

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

C. Hardware Peripheral

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

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

I. Bit-plane Reorganization

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

II. BPC Module

1) Generic: It handles all modes of BPC\ operation for all nominal code-block dimensions.
2) High processing throughput: It generates an average of 1.1 CxD/clock-cycle (Existing generic block coder architectures only generate 0.7CxD/clock-cycle) [5].
3) It is based on 2 state memory system [6].
4) Uses our proposed optimal 2 sub-bank memory architecture for internal memories [7].
5) It has the minimum memory cost (16 Kbits dual port RAM) currently reported for a generic BPC architecture.
6) Uses an efficient intermediate buffer: The BPC has varying CxDs/clock-cycle output (anywhere between 0 to 10 CxDs/clock-cycle) depending on image statistics. Due to the fact that our AC module has a maximum input rate of 2 CxD/clock-cycle, we use an intermediate buffer [8] to integrate the BPC and AC\ module. The buffer is optimized for its hardware cost vs throughput performance using real image statistics.

III. AC Module

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

IV. Distortion Estimation

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

We designed a novel hardware module for distortion estimation which uses 1 fractional bit (in comparison to 5 fractional bits as used by Kakadu). Our simulation results show that it results in average 0.02 dB PSNR degradation in comparison to Kakadu for a given target rate (the reported architectures achieve an average 0.3-0.7 dB PSNR degradation [10]).

D. Implementation Results

Figure 7 The simulation flow showing block coder-avalon bus transfer and Figure 8 Simulation flow showing DMA transfer in place between bit-plane reorganizer and BPC show the simulation waveforms of our system. The following points in time are of particular interest:

1.82100ns Sample DMA finishes
2.82200ns Register file access asserts operating parameters and starts system
3.82400ns Bitplane reorganization started
4.132550ns Register file access checking status and reading compressed data bytes
5.132500ns onwards demonstrates normal operation with bit plane coder providing contexts to the arithmetic coder.

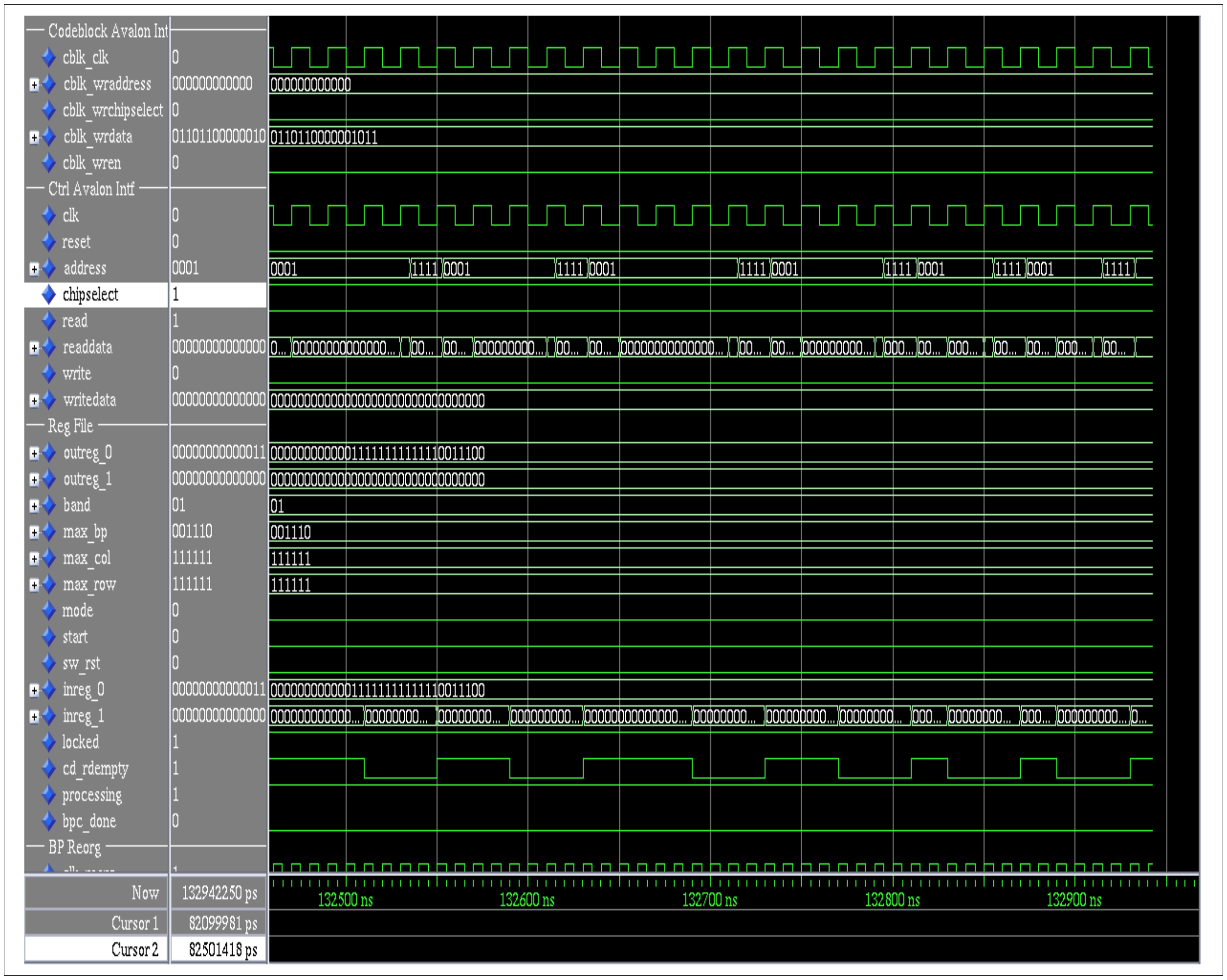
```
Codeblock Avalon Intf
cblk_clk           0
cblk_wraddress     000000000000
cblk_wrchipselect  0
cblk_wrdata        01101100000010
cblk_wren          0
Ctrl Avalon Intf
clk                0
reset              0
address            0001
chipselect         1
read               1
readdata           00000000000000
write              0
writedata          00000000000000
Reg File
outreg_0           00000000000011
outreg_1           00000000000000
band               01
max_bp             001110
max_col            111111
max_row            111111
mode               0
start              0
sw_rst             0
inreg_0            00000000000011
inreg_1            00000000000000
locked             1
cd_rdempty         1
processing         1
bpc_done           0
BP Reorg
Now                132942250 ps
Cursor 1           82099981 ps
Cursor 2           82501418 ps
```

Figure 7 The simulation flow showing block coder-avalon bus transfer

7. Conclusion

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

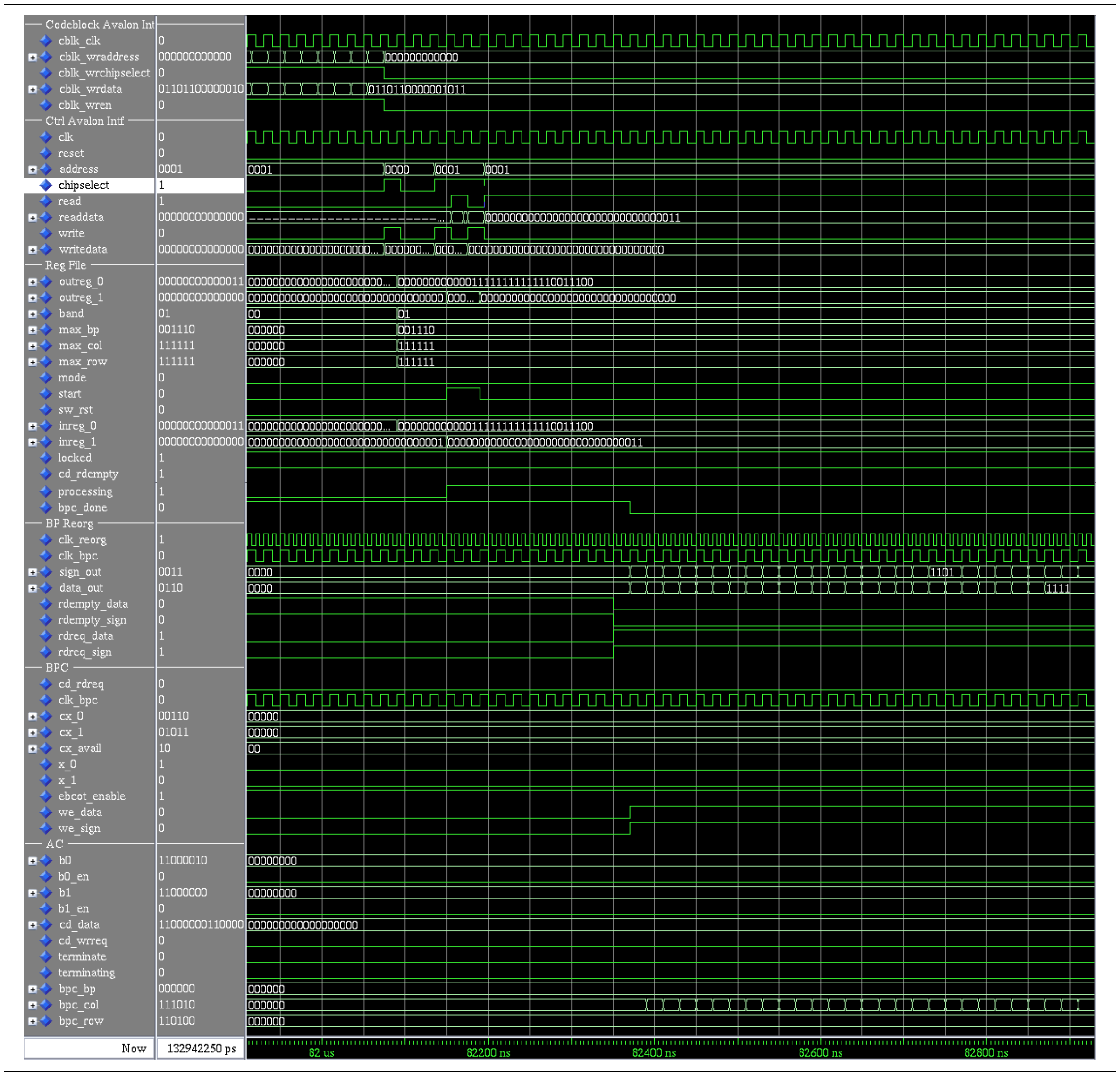

Figure 8 Simulation flow showing DMA transfer in place between bit-plane reorganizer and BPC

8. References

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

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

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

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

[5] A. K. Gupta, S. Nooshabadi, and D. Taubman, “Concurrent symbol processing capable VLSI architecture for bit plane coder of JPEG2000,” *IEICE Transactions on Information and Systems, Special Section on Recent Advances in Circuits and Systems*, vol. E88-D, pp. 1878 – 1884, 2005.
[6] Y.-T. Hsiao, H.-D. Lin, K.-B. Lee, and C.-W. Jen, “High-speed memory-saving architecture for the embedded block coding in JPEG2000,” in *IEEE International Symposium on Circuits and Systems*, vol. 5, pp. V–133 – V–136, May 2002.
[7] A. K. Gupta, S. Nooshabadi, and D. Taubman, “Optimal 2 sub-bank based memory architecture for bit plane encoder of JPEG2000,” *IEEE International Conference of Circuits and Systems (ISCAS’05)*, 2005.
[8] A. K. Gupta, S. Nooshabadi, and D. Taubman, “Efficient VLSI architecture for buffer used in EBCOT of JPEG2000 encoder,” *IEEE International Conference of Circuits and Systems (ISCAS’05)*, 2005.
[9] M. Dyer, D. Taubman, and S. Nooshabadi, “Improved throughput arithmetic coder for JPEG2000,” *IEEE international conference on Image Processing (ICIP’04)*, 2004.
[10] Y. Chang, H. Fang, C. Lian, and L. Chen, “Novel pre-compression rate distortion optimization algorithm of JPEG2000,” *SPIE Proc. of Visual Communication and Image Processing*, vol. 5308, pp. 1353–1361, 2004.