Design of a JPEG Encoder using SpecC Methodology

Hanyu Yin
Haitao Du
Tzu-Chia Lee
Daniel D. Gajski

Technical Report ICS-00-23
July 25, 2000

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{hyin, hdu, tzuchial, gajski}@ics.uci.edu
http://www.ics.uci.edu/~cad

Design of a JPEG Encoder using SpecC Methodology

Hanyu Yin
Haitao Du
Tzu-Chia Lee
Daniel D. Gajski

July 25, 2000

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{hyin, hdu, tzuchial, gajski}@ics.uci.edu
http://www.ics.uci.edu/~cad

Contents

List of Figures

Design of a JPEG Encoding System using SpecC Methodology

H. Yin, H. Du, T. Lee, D.D. Gajski
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract

This report describes the design of a JPEG encoder, using the SpecC system level design methodology developed at CAD lab, UC Irvine. We first begin with an executable specification model in SpecC, and then refine the specification model into a architecture model which accurately reflects the system architecture. Based on the architecture model, a communication model where the communication protocols between the system components are defined are developed. Finally, we refine the communication model of the DCT block into the implementation model, which is the lowest level of abstraction in the SpecC methodology. This Project is a result of a course "System Tools" at Information and Computer Science Department, UC Irvine.

1 Introduction

The goal of this project is to verify SpecC methodology by applying it to the design of a JPEG encoder. In this project, the JPEG encoder was described in four models which represent four different levels of abstraction in the SpecC methodology. Figure 1 shows the SpecC methodology.

The specification model is a pure behavior description. In our project, the SepcC specification model was derived from the C description of the JPEG encoder without introducing any implementation detail. The communication between the behavioral blocks are implemented by using global variables rather than channels since up to now no concurrency and synchronization characteristic is specified.

The architecture model is an refined model from the specification model with the partition of hardware and software. The concurrency and synchronization relationships resulting from this partitioning are explicitly described by substituting the global variables with the channels, which will finally be encapsulated into a global bus. The communication between the software blocks running on the same processor is still implemented by using the global variables.

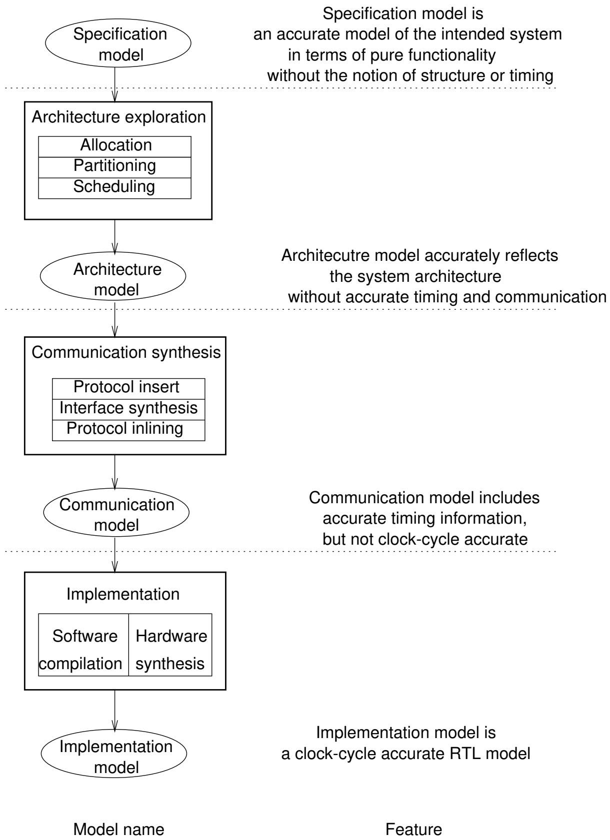

Figure 1: The SpecC methodology

The communication model is the same as the architecture model in that the blocks are mapped to the same components. However,the protocol description for the inter-block communication is refined into the timing-accurate description. This timing-accurate model can be described without knowing the internal structure of the hardware if the I/O protocols between these communicating blocks are clearly defined. But if the protocol is not explicitly given for a block, the RTL description must be analyzed to generate the corresponding I/O protocol. what's more, if the two protocols for the two communicating blocks are not compatible, a transducer has to be inserted.

Finally, the hardware block, which is DCT block in our design, is refined into the cycle-accurate description, which is the lowest level of abstraction in the SpecC methodology.

The rest of the report is organized as follows: Section 2 describes the JPEG encoder algorithm. Section 3 describes the JPEG specification model. In Section 4, the translation from the specification model to the architecture model is shown. In Section 5, a refined communication model of the JPEG encoder is given. In Section 6, the DCT block of the JPEG encoder is refined into a clock-cycle accurate RTL model. We conclude this report in Section 7.

2 Introduction of JPEG

JPEG is an standard for image compression. It applies to either the full-color images or the gray-scale images of the natural, real-world scenes. Today, parts of JPEG are already available as software-only packages or together with specific hardware support.

There are four modes of the operations in the JPEG standard: the sequential discrete cosine transform (DCT)-based mode, the progressive DCT-based mode, the lossless mode and the hierarchical mode. Our design employs the first mode, the sequential DCT-based mode, which is the simplest and the most commonly used mode.

Based on the sequential DCT-based mode, the JPEG encoder is divided into four functional blocks: the image fragmentation block, the DCT block, the quantization block and the entropy coding block. The corresponding block diagram in Figure 2 illustrates the communication relationship between these blocks.

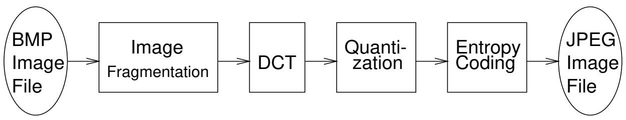

Figure 2: Block diagram of the JPEG encoder

In the image fragmentation functional block, an image is divided into the non-overlapping data blocks, each of which contains an 8*8 matrix of pixels.

In the DCT functional block, each data block is transformed into a frequency representation. There are two commonly used DCT algorithms for this translation process: the standard DCT and the ChenDCT [DCT]. The ChenDCT algorithm is employed in our design.

In the quantization block, the DCT output coefficients are quantized.

Finally, in the entropy coding block, the AC coefficients are encoded by using a predictive coder and the DC coefficients are encoded by using a run-length coder. Then the Huffman coding algorithm is employed to generate the JPEG image.

3 Specification Model

The SpecC specification model models the functionality of the system without introducing any unnecessary implementation details. No timing characteristic is modeled for the computation and communication behaviors. When simulated, the specification model is assumed to be executed in zero time. The specification model is as Figure 3.

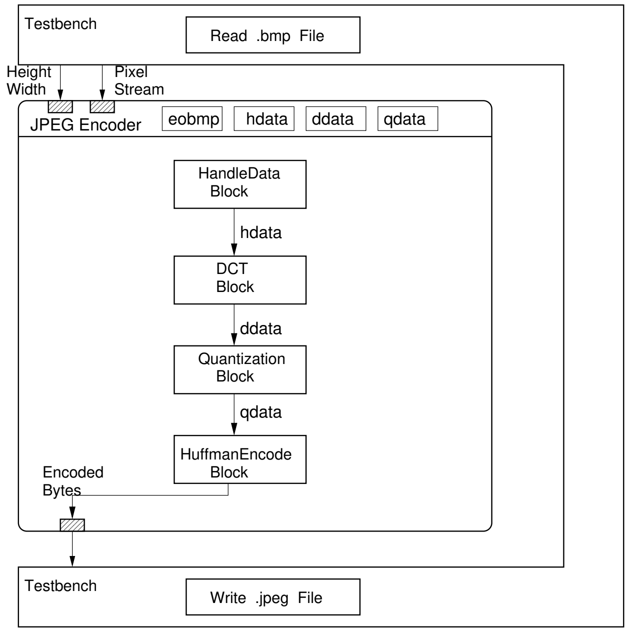

Figure 3: Specification model of the JPEG encoder

The specification model for the JPEG encoder is illustrated in Figure 3 in which there are three top-level behaviors: Read.Bmp File, Write.Bmp File, and JPEG Encoder. Right now, the Height&Width, Pixel stream and Encoded Byte are channels for the communication between these top-level behaviors. The JPEG Encoder is further divided into four sub behaviors: HandleData, DCT, Quantization, and HuffmanEncode. There are three global variables (hdata,ddata,qdata) are introduced for the data exchange between these sub behaviors since these sub behaviors are sequentially executed. Another global variable (eobmp) is used to indicate the end of the data transfer for the current image. The eobmp is public to all the four behaviors in the JPEG Encoder. The functionality of each behavior is described in the following subsections. The complete SpecC model is given in Appendix A.

Before going to the functionality descriptions, we make one note here: channels are used in the SpecC specification model only if some concurrency and synchronization relationships are explicitly defined. However, later implementation may be different depending on the selected architecture.

3.1 Test Bench: Read.Bmp File and Write.Bmp File

Read.Bmp File and Write.Bmp File can be modeled as the testbench for the JPEG Encoder. First, the height (H) and width (W) of the image are passed from the Read.Bmp File to the JPEG Encoder, which uses them to determine the number of the iterations for the block transfer (H*W). Then serials of the image pixel streams (8-bit wide) are passed to the JPEG Encoder where the image is packeted into blocks and then processed block by block. Each block consists of 8*8 bytes of image information. The processed image is then sent out byte by byte to the Write.Bmp file block where the final JPG image file is generated.

3.2 Blocks in the JPEG encoder

The JPEG encoder includes four blocks: HandleData, DCT, Quantization and HuffmanEncoder.

3.2.1 HandleData Block

The first block, the HandleData block, reads the inputs H, W and pixel stream from the Read.Bmp File, calculates the number of the iterations, groups the pixel stream into 8*8 pixel matrix (MCUs) and sends the MCUs to the DCT block.

3.2.2 DCT Block

The second block, the DCT block, reads the MCUs passed in from the HandleData block, preshifts the MCUs, performs the Chen forward DCT algorithm on them, and sends the result (still 8*8 matrix called transformed MCUs) to the Quantization block.

3.2.3 Quantization Block

The third block, the Quantization block, uses a quantization table to quantize each element of the resulting MCUs passed in from the DCT block and sends the result to the HuffmanEncoder block.

3.2.4 HuffmanEncoder Block

The last block, the HuffmanEncoder block, performs a Huffman entropy-encoding and a run-length-encoding (RLE) on the successive bytes in the incoming MCUs. Each byte is transformed into a bit-sequence (often smaller than the 8 bits of the input byte). The sequence of encoded bits is then packed into bytes and written to the Write.Bmp File block.

4 Architecture Model

The architecture model is an refinement of the specification model in that the behaviors are partitioned into the hardware behaviors and the software behaviors, which can execute concurrently. Because of this, channels are used in the architecture model to describe the concurrency and synchronization relationship between these concurrent behaviors. The use of channels implies that the final implementation of the communication relationship would be the buses. So at the end of the architecture modeling, these channels are encapsulated into one bus. All the communication that go through the channels will be implemented in this bus. Global variables are still used for the communication between the behavioral blocks running on the same processor.

In the architecture model, neither the software blocks nor the hardware blocks are timing-accurate. The communication between concurrent blocks is also not timing-accurate.

In the SpecC design methodology, two steps must be performed to get the final architecture model: HW/SW partition and Channel partition, both of which are described in the following two subsections.

4.1 HW/SW Partition

To get the better trade-off between the performance and cost, HW/SW partition is performed, which involves the estimation of the different partitions. Based on the estimation [CPGC], the JPEG Encoder in our project is partitioned into two parts: 1) the software model, which includes three blocks: HandleData, Quantization, and HuffmanEncode; 2) the hardware model, which includes the block DCT. Even if the

DCT block is called the hardware block, its model is still in the behavioral description in the specC architecture model. The refined model after behavior partitioning is illustrated in Figure 4.

The software model is implemented on the selected processor. In our design, the ColdFire processor is employed. The hardware model can be synthesized into a custom hardware. The channels are employed to model the communication between the software block and the hardware block because concurrency and synchronization relationship can be efficiently described in channels in SpecC. In our model, these channels includes two data channels: chdata, cddata, and one control channel: ceobmp. The functionality of these channels is to convey the corresponding data and control information. Behavior OEOBmp, IEOBmp, and channel ceobmp are used to transfer eobmp. Behavior OHData, IHData, and channel chdata are used to transfer hdata. Behavior IDData, ODData, and channel cddata are used to transfer ddata. The SpecC source code is given in Appendix B.

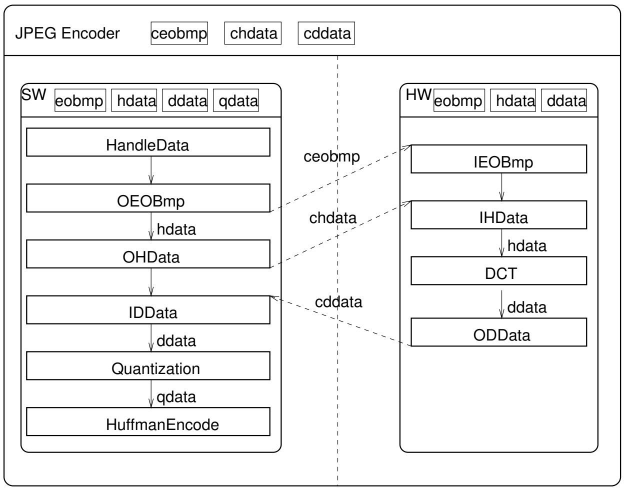

Figure 4: Model after behavior partitioning

4.2 Channel Partition

The goal of the channel partitioning is to group and encapsulate all the channels existing between the two communicating blocks into one bus. The bus is also a type of channel in SpecC, and it implies that the future implementation would be the wired buses. There is some difference between the channel and the bus. That is: channels connect the concurrent behaviors while buses connect the corresponding components into which these behaviors are mapped. Only one bus can exist between two components. The refined architecture model after the channel partitioning is given in Appendix B.

In our model, the channel ceobmp, chdata, and cddata are encapsulated into one global channel Bus. All the implementations of these channels are also encapsulated into this bus. The bus can refer the corresponding channel for the communication between two behavior blocks. The SpecC source code for the bus channel is shown in Appendix B.

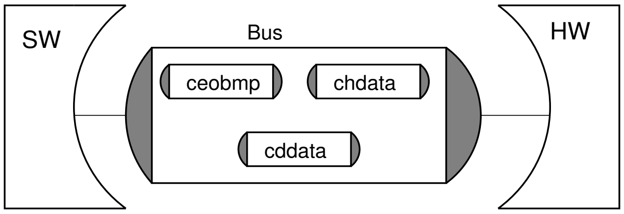

Figure 5: Model after channel partitioning

5 Communication Model

The communication model is the same as the architecture model in that all blocks are mapped to the same components. However, the bus model between the two communicating blocks is timing-accurate but not necessarily to be cycle-accurate. To do this, the time-accurate bus protocols for both of the communicating blocks should be defined. What's more, if these two protocols are not compatible, a transducer should be inserted. Finally, the implementation of the bus communication is inlined into the two communicating components. For the software component, this implementation is represented by an Assembly language I/O procedure which is called a bus driver. For the hardware component, this implementation is a time-accurate behavior description such as a Super FSMD in which each state represents a set of operations that are executed in one or more clock cycles. This time-accurate description will later be refined into a clock-accurate FSMD model.

In our model, the ColdFire protocol is employed for the software part to input/output the data. However, no protocol is defined for the hardware (DCT) part so we need to analyze the DCT datapath to generate the I/O protocol. As we will see, the protocol for the ColdFire and the one for the DCT are not compatible, so a transducer is inserted. The bus protocols for the ColdFire and the DCT as well as the transducer description are described in the following subsections.

5.1 Bus Protocol

5.1.1 ColdFire Master Bus

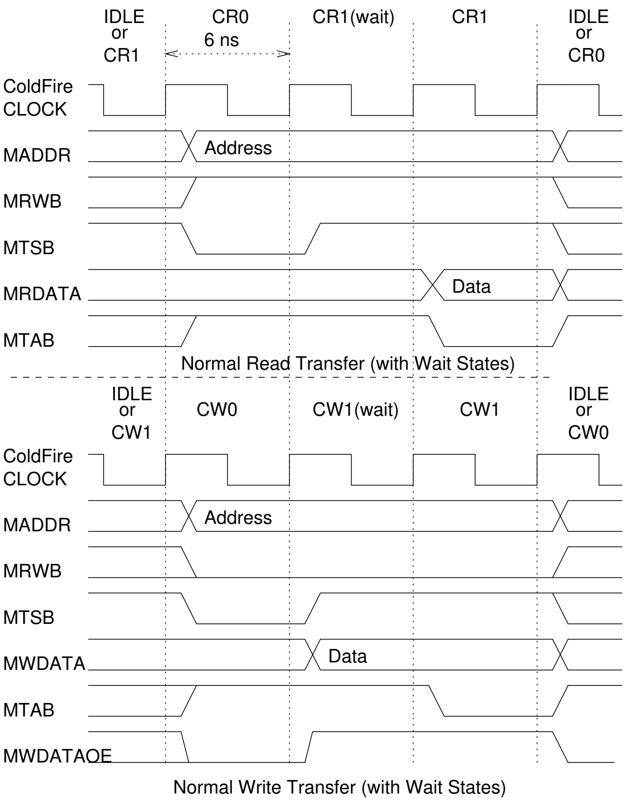

Figure 6: ColdFire master bus timing diagram

The communication between the ColdFire processor and an external device is based on the ColdFire Master Bus protocol. A subset of the master bus signals used in our model are listed below.

Master Address Bus (MADDR[31:0]). If these output signals are used, the memory space of the external device is implicitly mapped into the memory space in the ColdFire processor. So the ColdFire processor can explicitly specify the target address of the external device during the I/O process.

Master Read/Write (MRWB). This output signal indicates a data read/write operation for the current bus communication. A high level '1' indicates a read and a low level '0' indicates a write.

Master Transfer Start (MTSB). This output signal indicates the start of a bus transfer when it is asserted to be '0'.

Master Read/Write Data Bus (MRDATA[31:0] and MWDATA[31:0]). These input/output bus signals provide the datapaths for the data I/O. We can choose to use 8, 16, 32 bits of the data bus per data bus transfer.

Master Transfer Acknowledge (MTAB). This input signal is asserted to '0' by the slave to indicate the successful completion of a bus transfer. It is sampled by the ColdFire Processor at the end of each clock cycle. If it is not asserted, the ColdFire inserts one or more wait states.

Master Write Data Output Enable (MWDATAOE). when asserted to '1', this output signal indicates that the ColdFire is driving the master write data bus. This is used to control optional bidirectional data bus three-state drivers.

Besides these, there is a special bus INTC coming into the ColdFire from outside. Please note that this bus is not in the ColdFire Protocol.

Figure 6 shows the timing diagram for the master bus read (with wait states) and write (with wait states) I/O operations. Figure 7 shows the corresponding FSMD for the ColdFire normal read transfer (with wait states) and Figure 8 shows the ColdFire normal write transfer (with wait state).

The ColdFire master bus protocol is a two-cycle process which include the data transfer start cycle and the data transfer acknowledge cycle. However, one or more wait cycles may be inserted if the acknowledgment of the slave cannot arrive in two ColdFire clock cycles. At first (when there is no I/O request), the ColdFire is in the IDLE state. Once an I/O instruction is decoded and executed, a bus I/O is activated and the ColdFire starts the data transfer. Please note in the usual data transfer, the state before the current data transfer can be an IDLE state or the last state for the previous data transfer. This is shown in the timing diagram Figure 6.

For the ColdFire read I/O, in the first clock cycle (CR0), the address (MADDR) and control information (MTSB, MRWB) are driven onto the bus. In the next cycle (CR1), the address MADDR remains on the bus. The MRWB is still asserted. But MTSB is deasserted to '1'. At this time, one wait states may be inserted if the slave is not ready to send the data. In this case, the slave will keep (desserting) MTAB as '1'. The ColdFire samples the MTAB at the end of the current clock cycle. Because it finds that the MTAB is deasserted ('1'), the ColdFire will insert one wait state (CR1 wait). In other words, the ColdFire will stay one more clock cycle at this state (CR1). In the next cycle (the ColdFire is still at state CR1), if the slave is ready to send the data, it will assert the MTAB to '0'. At the end of this clock cycle, the ColdFire samples the MTAB. It finds that the MTAB is asserted to '0', so at the end of this clock cycle, the data at the data bus

MRDATA is latch into the ColdFire data buffer. If there are more I/O requests suspending, the ColdFire will immediately go back to the state CR0 and start the next data I/O. Otherwise, if there is no more I/O requests, the ColdFire will go to the IDLE state.

For the ColdFire write I/O, in the first clock cycle (CW0), the ColdFire asserts the MTSB to '0', drives the address to address bus MADDR, asserts the MRWB to '0' indicating a write I/O. The MWDATAOE is kept deasserted as '0' indicating that data will be not ready on the MWDATA bus until the next clock cycle. In the next clock cycle (CW1), the data is driven onto the data bus by the ColdFire. Also in this clock cycle, the address remains at the MADDR. The MRWB is still asserted as '0'. But the MTSB is deasserted to '1', and the MWDATAOE is asserted to '1' indicating that data is now on the data bus MWDATA. If the slave is not ready to latch the data, the MTAB will not be asserted, that is, the MTAB is '1' in this clock cycle(CW1). At the end of the current clock cycle (CW1), the ColdFire samples the MTAB, and if the MTAB is still deasserted ('1'), one more wait state (CW1 wait) will be inserted. In the next clock cycle (now the ColdFire is still in the state CW1), if the slave is ready to latch the data (which is not shown in the timing diagram), it will assert the MTAB to '0'. At the end of this clock cycle (CW1), the ColdFire samples the MTAB, and it finds that the MTAB is asserted to '0', so it removes the data from MWDATA and jumps out of the wait state (CW1). If there is more I/O requests suspending, the ColdFire will immediately go to the state CW0 and starts a new data transfer. Otherwise, it will go to the IDLE state.

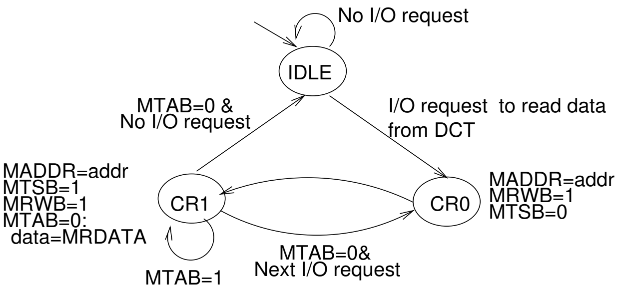

Figure 7: FSMD for the ColdFire normal read transfer

Figure 7 shows the corresponding FSMD for the ColdFire normal read transfer (with wait states).

Figure 8 shows the corresponding FSMD for the ColdFire normal write transfer (with wait states).

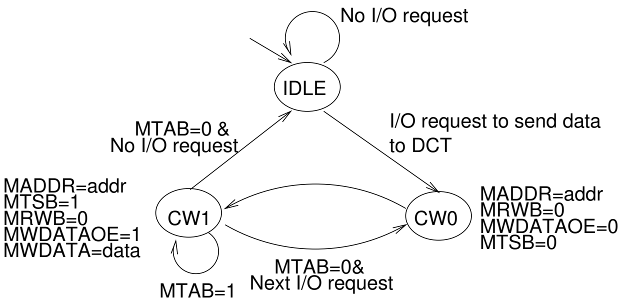

Figure 8: FSMD for the ColdFire normal write transfer

5.1.2 DCT Bus

In our design, the internal structure of the DCT block need to be analyzed to generate the I/O protocol because no protocol is explicitly defined for the DCT. Figure 9 shows the RTL structural diagram of the DCT block.

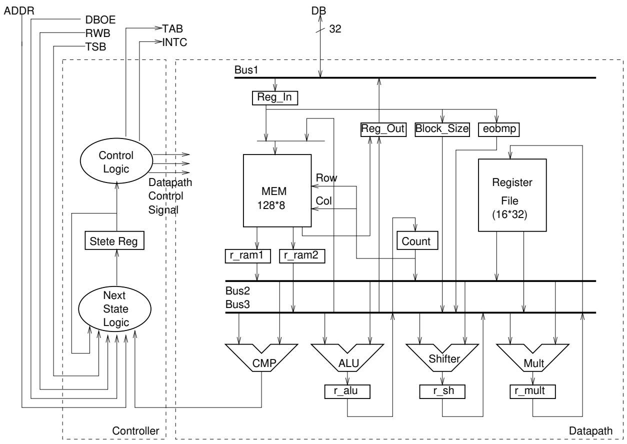

Figure 9: RTL structural diagram of DCT

The bus signals used for the DCT I/O are listed below.

Address bus (ADDR). This input address signal is decoded by the DCT to indicate the corresponding source/destination for data I/O.

Read/Write (RWB). This input signal indicates the data transfer direction in the current bus cycle. A high level indicates a cycle to write to the DCT and a low level indicates a cycle to read from the DCT.

Transfer Start (TSB). This input signal indicates the start of a bus I/O when it is asserted to be '0'.

Data Bus (DB[31:0]). These input/output 32-bit bi-directional signals provide the data paths for the DCT I/O.

Data Bus Data Enable (DBOE). This input signal indicates that the data on the DB[31:0] is ready for latching when it is asserted to be '1'.

Transfer Acknowledge (TAB). This output signal acknowledges the successful completion of the current bus I/O when asserted to be '0'.

Besides these, there is a output interrupt signal interrupt INTC, which is used to interrupt the ColdFire processor. Please note that this signal is not in the DCT bus protocol.

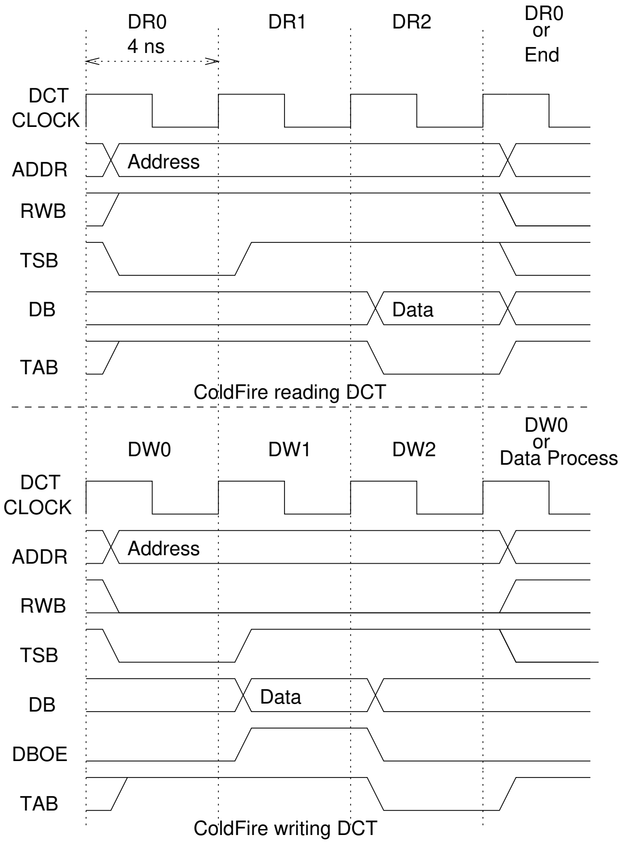

Figure 10: DCT bus operation timing diagram

The DCT clock cycle is assumed to be 4 ns. Figure 10 shows the timing diagram for the DCT bus I/Os. We can see that the DCT bus protocol is a three-cycle process.

When the DCT is read (after the data is processed in the DCT), it will keep TAB deasserted ('1') in the first clock cycle (DR0). The DCT will check the input signal TSB at the end of the first clock cycle (DR0) to see if the TSB is asserted ('0'). If the TSB is not asserted to '0', the DCT will insert a wait state. It means that the DCT will stay at DR0 for one more DCT clock cycle. If the TSB is asserted to be '0', in

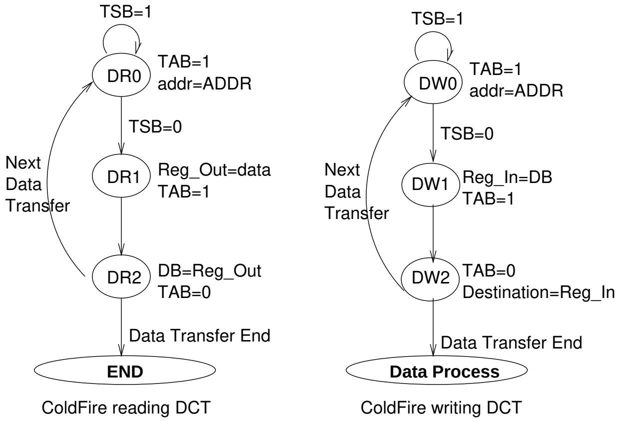

Figure 11: FSMD for DCT bus transfer

the next clock cycle (DR1), the decoded ADDR signals will be used as the memory source address to pull the data out of the MEM into the default output register Reg_Out. In this clock cycle (DR1), the TAB is kept deasserted as '1'. In the third clock cycle (DR2), the data in the Reg_Out is driven onto the DB[31:0]. At the same time (DR2), the TAB is asserted to '0' to acknowledge the successful completion of the current bus I/O.

When the DCT is written, it will keep the TAB deasserted ('1') in the first clock cycle (DW0). The DCT will check the TSB at the end of the first clock cycle (DW0) to see if the TSB is asserted to '0'. If the TSB is not asserted to '0', the DCT will insert a wait state. It means that the DCT will stay at DW0 for one more clock cycle. If the TSB is asserted to '0', the DCT will jump out of the wait state(DW0) in the next clock cycle. The decoded ADDR signals will be accepted as the valid memory destination address, and will be used later in the third clock cycle (DW2). In the clock cycle (DW1) after the DCT jump out of the wait state, the data on the DB[31:0] is latched into the default input register Reg_In. In the third clock cycle (DW2), the data in the Reg_In is sent to the destination. At the same time (DW2), the TAB is asserted to be '0' to acknowledge the successful completion of the current bus I/O. The next state can be either the following data transfer start state (DW0) if the data transfer is not ended, or the data processing start state (Data Processing) if the data transfer is ended.

Figure 11 shows the corresponding FSMD for the DCT bus transfer.

5.2 Transducer Design

The functionality of a transducer includes 1) re-packet the data to the type which can be recognized by the other communicating block. 2) adjust the data arrival and leaving time such that data is safely transfered. So at least, there should be a storage unit in a transducer.

A transducer is inserted into our model since the ColdFire protocol and the DCT protocol are not compatible. The datapath and the corresponding control units are shown in Figure 12. The signals for the communication between the ColdFire and the DCT are shown in this figure.

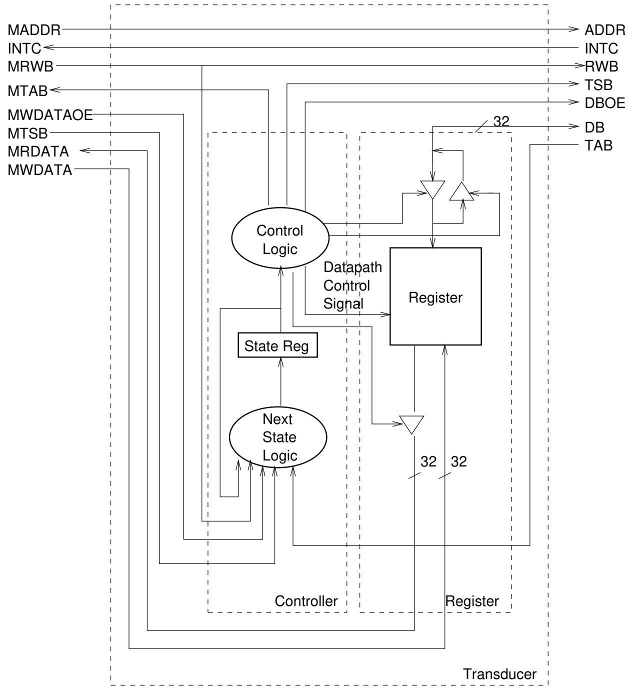

Figure 12: Transducer between ColdFire and DCT

Figure 13 shows the FSMD for the transducer which illustrates the communication process between the ColdFire and the DCT.

The Figure 14 is the corresponding timing diagram for data transfer from the DCT to the ColdFire.

The Figure 15 is the corresponding timing diagram for data transfer from the ColdFire to the DCT. Please note that in this timing diagram, TW2 can be removed if the rising of the MWDATAOE is faster enough such that it is sampled to be '1' at the end of the TW1.

The transducer is always waiting for an data I/O request. Thus In T0, the MTAB is kept '1' for deassertion. The transducer is waiting for the assertion signal of '0' from the MTSB, which indicates the start of a bus I/O. If the MTSB is asserted to be '0', the next

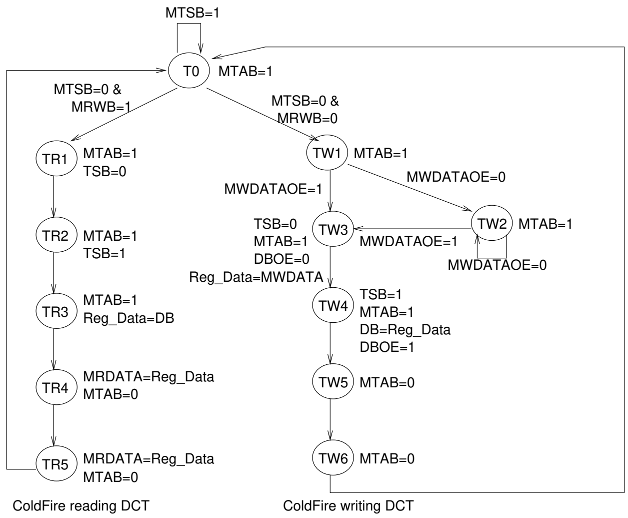

Figure 13: FSMD for the transducer

state might be TR1 or TW1 depending on the value of the MRWB (1: Read, 0: Write). The FSMD for the data transfer from DCT to ColdFire and the one from ColdFire to DCT are described in the following two subsections.

5.2.1 Transducer FSMD for data transfer from DCT to ColdFire

In TR1, the MTAB is not asserted (that is '1'), but the TSB is asserted to be '0', telling the DCT that a new bus I/O begins. In TR2, the DCT loads the out-going (to ColdFire) data into the Reg_Out from the MEM. The MTAB is still not asserted at this state (TR2). During TR3, the DCT drives the data from Reg_Out onto the DCT data bus (DB[31:0]). In TR4, the data is loaded into the Register in the transducer and driven onto the ColdFire's master read data bus (MRDATA[31:0]). At the same time (TR4), the MTAB is asserted to '0'. The data remains on the data bus and the MTAB is still asserted '0' in TR5 because the clock rate of the transducer is faster than that of the ColdFire. After that, the transducer goes back to the wait state (T0) for the next data I/O.

5.2.2 Transducer FSMD for data transfer from ColdFire to DCT

In TW1, the MTAB is not asserted (that is '1'). At the end of TW1, the transducer samples the MWDATAOE and latches the current value on the MWDATA[31:0]. If the MWDATAOE is asserted to '1', the transducer goes to TW3. If MWDATAOE is not

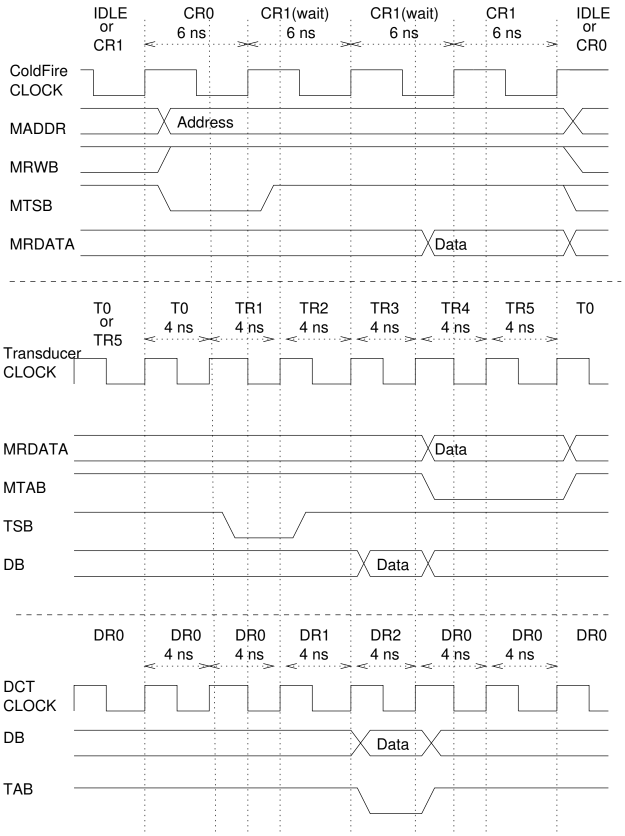

Figure 14: ColdFire reading DCT

asserted (that is '0'), the transducer goes to TW2. In TW2, the transducer keeps sampling the MWDATAOE and latches the current value on the MWDATA[31:0] until the MWDATAOE is asserted to '1', then it goes to TW3. In TW3, the transducer asserts the TSB to '0', telling the DCT that a new bus writing I/O begins. During TW4, the transducer drives the data onto the DCT data bus (DB[31:0]). Concurrently, the DCT asserts the transfer acknowledge (TAB) to '0'. In TW4, the DCT latches data from the data bus into the Reg_In. In TW5, the data is sent to the destination. At the same time, the transducer asserts the MTAB to '0' indicating the ColdFire the completion of this bus I/O. The MTAB remains '0' in TW6 to ensure that the ColdFire samples the correct value of the MTAB.

6 Implementation Model of DCT Block in JPEG Encoder

In our design, the DCT block is the only block which is refined into the Implementation model (RTL model),

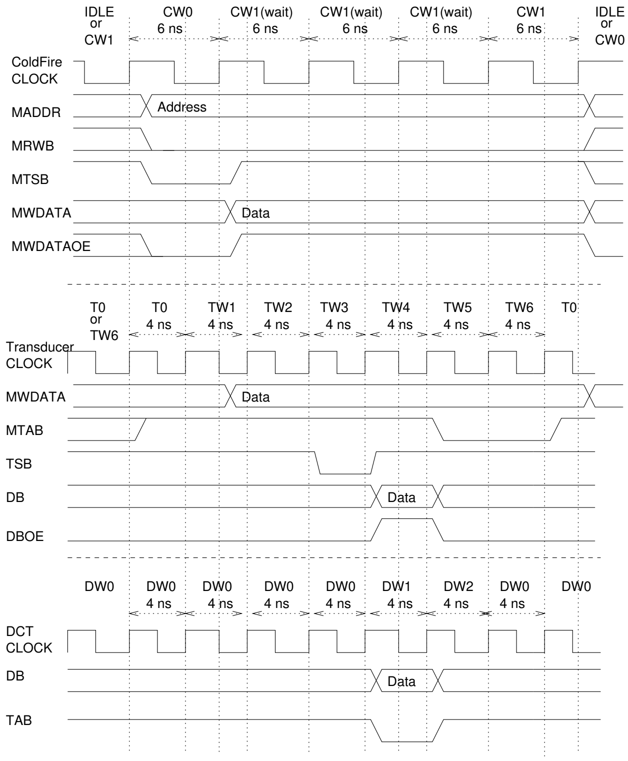

Figure 15: ColdFire writing DCT

and which will later be synthesized into a custom hardware.

The implementation model of the DCT block can be globally divided into five functional parts: receive eobmp, receive blocksize, receive hdata, dct process, and send ddata. As is illustrated in Figure 16.

The recieve eobmp, receive blocksize, and receive hdata get data from SW behavior. Since the first two parts only receive one data each, the FSMD for these two parts is the same as the FSMD of ColdFire writing DCT in Figure 11. The receive hdata part gets the 8*8 matrix from SW behavior, and Figure 17 shows the superstate FSMD of the receive hdata part. Here we introduce a count 'i', which is campared with 64(blocksize) each time. RB2 is a super state which represents the FSMD of ColdFire writing DCT in Figure 11.

The dct process part processes the 8*8 matrix which is sent from SW behavior. Figure 18 shows the superstate FSMD of the dct process part. There are three super states: Preshift DCT, ChenDCT, and Bound DCT. Preshift DCT preshifts the 8*8 matrix in the range of -128 to 127. It has five states. ChenDCT implements the Chen forward dct. It has 147 states. Bound DCT bounds the output 8*8 matrix of ChenDCT in the range of -1023 to 1023. It

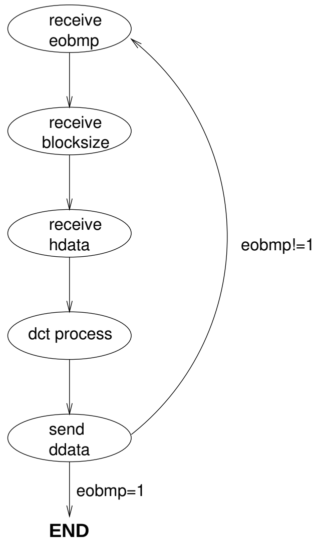

Figure 16: Superstate FSMD of HW behavior

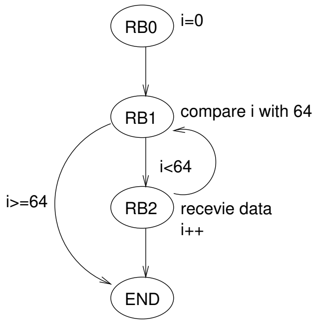

Figure 17: Superstate FSMD of the receive hdata part

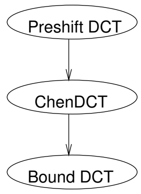

Figure 18: Superstate FSMD of the dct process part

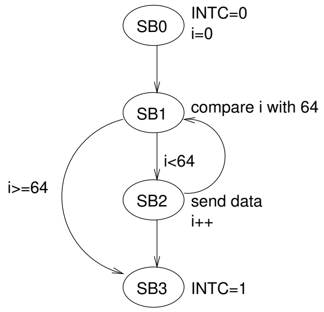

Figure 19: Superstate FSMD of the send ddata part

has seven states. The SpecC source code is given in Appendix D in two styles of Behavioral RTL (For difference, Please see the Appendix) .

The send ddata part sends the 8*8 output matrix to SW behavior, and Figure 19 shows the superstate FSMD of the send ddata part. Here we introduce a count 'i', which is campared with 64(blocksize) each time. SB2 is a super state which represents the FSMD of ColdFire reading DCT in Figure 11.

7 Conclusions

In this report, we apply the SpecC system-level design methodology into the design of a JPEG encoder. Four models are constructed based on this methodology: the specification model, the architecture model, the communication model, and the RTL (behavioral) model. For each model, we demonstrated the mechanism of refining the previous (higher level) model into the current (lower level) model. What's more, during the communication synthesis, we generated the I/O protocol for the DCT hardware block, and derived a traducer for the communication of the ColdFire and the DCT. The timing diagram shows that these two protocols match well with the translation of the traducer. We verified our SpecC models by running the testbench we mentioned in the specification model. The correct output demonstrates the correctness of our models

References

[1] [DCT] V. Bhaskaran, K. Konstantinides, *Image and Video Compression Standards*, Second Edition, Kluwer Academic Publishers, 1997.

[2] [GZDGZ] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, March, 2000.

[3] [CPCG] L. Cai, J. Peng, C. Chang, A. Gerstlauer, H. Li, A. Selka, C. Siska, L. Sun, S. Zhao, D. Gajski, *Design of a JPEG Encoding System*, Department of Information and Computer Science, University of California, Irvine, Technical Report, September 1999.

[4] [DZG] R. Dömer, J. Zhu, D. Gajski, *The SpecC Language Reference Manual*, Department of Infor-

mation and Computer Science, University of California, Irvine, Technical Report ICS-TR-99-11, February 1999.

[5] [AG1]A. Gerstlauer, *Architecture Exploration*, Department of Information and Computer Science, University of California, Irvine, Technical Report, May 2000

[6] [AG2]A. Gerstlauer, *SpecC Modeling Guidelines* Department of Information and Computer Science, University of California, Irvine, Technical Report, July 2000

[7] [ColdFire] Motorola, Inc., Semiconductors Products Sector, DSP Division, *ColdFire 2/2M Intergrated Micro-processor User's Manual, 1998*