Design of a JPEG Encoding System

Lukai Cai
Junyu Peng
Chun Chang
Andreas Gerstlauer
Hongxing Li
Anand Selka
Chuck Siska
Lingling Sun
Shuqing Zhao
Daniel D. Gajski

Technical Report ICS-99-54
November 20, 1999

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

{lcai,chun,gerstl,hongli,pengj,aselka,chucks,lsun,szhao,gajski}@ics.uci.edu
http://www.ics.uci.edu/~cad

Contents

List of Figures

List of Tables

Design of a JPEG Encoding System

L. Cai, J. Peng, C. Chang, A. Gerstlauer, H. Li, A. Selka, C. Siska, L. Sun, S. Zhao, 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. The project is a result of a course "System Tools" at Information and Computer Science Department, UC Irvine. The abstract executable specification SpecC is first developed based on a public domain C implementation. Software and hardware estimation is then performed based on which a datapath architecture is selected and RTL code is implemented. Finally, to explore the method of implementing a gate level model, part of the JPEG encoder is refined to the gate level.

1 Introduction

The goal of this project is to explore the methodology of HW/SW implementation of an application from a high level executable specification. The JPEG encoder was chosen as our example throughout the project.

This project began with selecting an implementation of the JPEG encoder in C from the public domain (Figure 1, box 1). The C code was then translated into SpecC model, which is a superset of ANSI-C with constructs that facilitate system level architecture exploration (box 2).

In order to obtain insight for an efficient architecture to implement JPEG, the estimation for both software implementation and hardware implementation was performed to select the target architecture (box 3).

Based on the resultant architecture, the SpecC model was scheduled to develop the RTL behavior model (box 4). Then part of RTL behavior model was refined into a "Synopsys Model" (RTL behavioral without memory) by separating the memory from the behavioral description (box 5), from which a gate level implemention was developed by suitably using synthesis tools (box 6).

To make those models understandable, an example of different models' formats is described in Figure 2 and a detailed definition for each model will be introduced later in the relevant section.

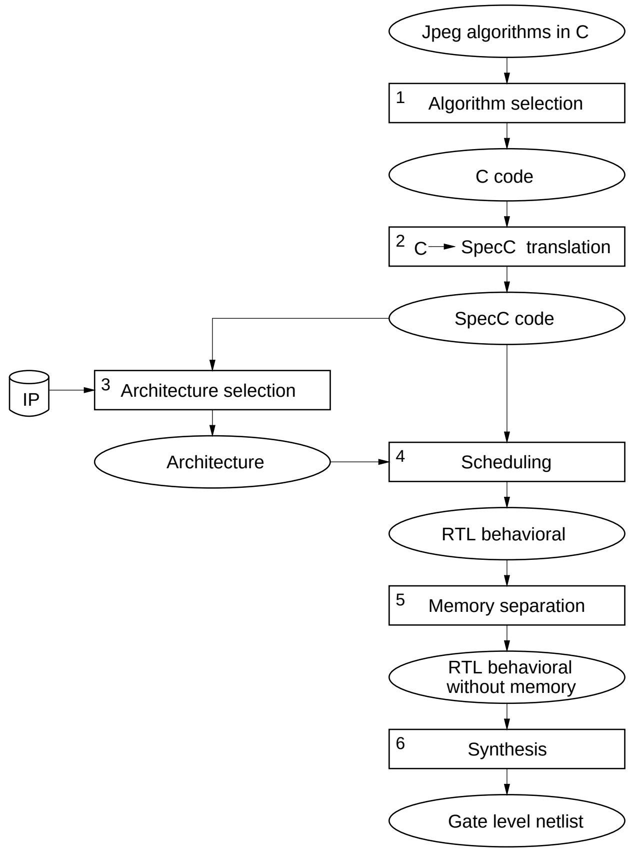


Figure 1: Design flow

The rest of the report is organized as follows: Section 2 describes the JPEG and JPEG C specification we used. In Section 3, the translation from C code to SpecC is shown. Software and hardware estimation procedures and results are described in Section 4. In Section 5, RTL behavior implementation is shown. The "Synopsys Model" design and gate level implementation is described in Section 6. We conclude the report in Section 7.

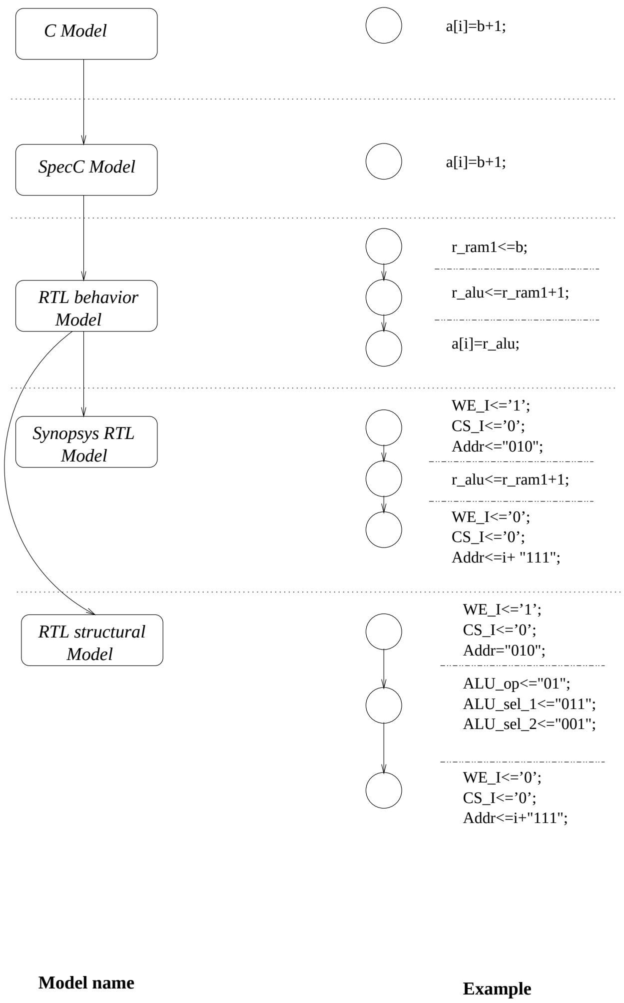


Figure 2: A example of different models

2 JPEG

JPEG (pronounced "jay-peg") is a image compression standard [1]. It is designed for compressing either full-color or gray-scale images of natural, real-world scenes. It works well on photographs, naturalistic artwork, and similar material; not so well on lettering, simple cartoons, or line drawings. JPEG handles only still images.

The JPEG standard specifies four modes of operation: a sequential discrete cosine transform (DCT)-based mode, a progressive DCT-based mode, a lossless mode and a hierarchical mode. Among these four modes, the sequential DCT-based mode is the simplest and most useful mode. In this report, we focus only on the sequential DCT-based encoder.

Figure 3 shows the block diagram of the DCT-based encoder for an gray scale image. It consists of four blocks: the image fragmentation block, the DCT block, the quantization block and the entropy coding block.

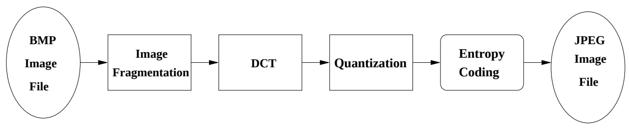


Figure 3: Block diagram of JPEG encoding

In the image fragmentation block, the image is divided into non-overlapping blocks, each of which contains an 8×8 matrix of pixels. Each block is then transformed into the frequency domain in the DCT block using a two-dimensional DCT algorithm. There are two commonly used DCT algorithms: standard DCT and ChenDCT. ChenDCT is chosen in this project for its superior performance.

The DCT output coefficients are then quantized in the quantization block before it is entropy-coded in the entropy coding block. The entropy coding block consists of two stages. The first stage is either a predictive coder for the DC coefficients or a run-length coder for the AC coefficients. The second stage is either a Huffman coder or an arithmetic coder. Huffman coder is chosen in the second stage for its simplicity.

3 Specification in SpecC

SpecC language provides special language constructs for modelling *concurrency, state transitions, structural and behavioral hierarchy, exception handling, timing, communication and synchronization* [2] which distinguish it from the standard programming languages, like C/C++ and Java, and those popular hardware description languages, like VHDL and Verilog. The initial SpecC specification should model the system at a very abstract level without introducing unnecessary implementation details. The original C code for JPEG specifies the functionality down to the bit level thus provides the basis for the specification in SpecC. For example, in Figure 2, there is no differece between C specification and SpecC specification for code "a[i]=b+1". However, a considerable amount of time was spent on analyzing and understanding the C code in order to extract the high-level structure and the global interdependencies. (For example, an analysis of how C pointers were used was performed in order to understand how to remove them.) Once this was done, a mapping into a SpecC representation was straightforward.

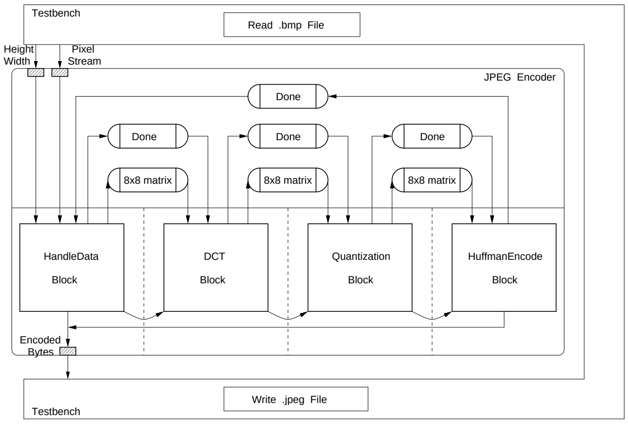


Figure 4: JPEG Encoder model in SpecC

The original C code for JPEG reads in a bitmap format picture file (.bmp), runs through the DCT and compression encoding process then writes out a JPEG format file (.jpeg). Thus, we can partition the C code into three modules (Figure 4): *Read .bmp File*, *JPEG Encoder* and *Write .jpeg File*. In the SpecC model it's reasonable to handle the *Read .bmp File* and *Write .jpeg File* within a testbench to separate them from the *JPEG Encoder* module, which is the core process that will be described below in more detail.

The top-level architecture of the *JPEG Encoder* module includes its interface to the *Testbench*, its block diagram and its inter-block communications (Figure 4).

3.1 Interface

The JPEG encoder interface includes two input ports to import data from the testbench. One is for passing in the height (*H*) and width (*W*) parameters of the image, the other is for passing in the (8-bit wide) image pixel stream. There is one output port to export the (8-bit wide) encoded bytes to the testbench.

3.2 Blocks

The JPEG encoder module is further divided into four blocks loosely based on the functionality observed in the C code. The first block, the *HandleData Block*, reads the inputs *H, W* and pixel stream through the input ports, then groups the pixel stream into 8 by 8 pixel matrices (called MCUs) for later processing. The second block, the *DCT Block*, reads each MCU passed from the *HandleData Block*, preshifts the MCU, then performs the DCT on the MCU. This results in a transformed MCU, also 8 by 8 bytes in size. The third block, the *Quantization Block*, uses a quantization table to quantize each element of the resulting MCU passed in from the *DCT Block*. This results in yet another transformed MCU. The last block, the *HuffmanEncode Block*, performs a Huffman entropy-encoding and a run-length-encoding (RLE) on the successive bytes from the input MCU. 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 output port. As we can see, these 4 concurrent blocks correspond to 4 stages in a pipeline. At any time, they operate in parallel but each on a different MCU. (This pipelining idea is not yet implemented in the current SpecC model.)

3.3 Communications

The inter-block communication is realized via buffered synchronized channels. The *Done* signal indicating the completion of the final MCU is implemented with a single bit channel between two neighbouring blocks. The data passing between adjacent blocks is implemented with a channel for an 8 by 8 byte array which stores data from the local array of the previous block and provides data for the local array of the next block.

4 Estimation

There are two methods to implement the JPEG encoder: software implementation, which is to run the SpecC model of the JPEG encoder on the targeted processor; and hardware implementation, which is to design proper architecture for the JPEG encoder and generate the RTL VHDL model of the JPEG encoder. This section first makes an estimation of the software as well as the hardware. Design decisions are made at the end of this section by comparing the results of these estimations.

4.1 Software Estimation

Software estimation was performed by compiling and simulating the SpecC model of the JPEG encoder on the targeted processor. In summary, this involves the following three steps:

1. Synthesizing valid C code for the target processor from the given SpecC description and the initial C reference model;

2. Compiling the C code into assembly code for the target processor;

3. Executing the compiled code in a customized instruction set simulator (ISS) to obtain a count of execution cycles.

Note that as part of software estimation a complete software implementation (in C) of the JPEG encoder based on the more hardware-friendly SpecC-based architectural design became available.

For the purpose of this project we chose the Motorola DSP56600 family of digital signal processors as the target for the software implementation [4]. The reasons for choosing this processor include its popularity and the availability of a tool chain (compilers, simulators, etc).

In order to synthesize the C model for the DSP56600, the functions of the SpecC model were scheduled and serialized, and each behavior was converted into a corresponding C function. The process was guided by the original C implementation of the JPEG encoder (which was the initial basis for the SpecC model). However, due to certain oddities of the Motorola architecture some modifications had to be made to the C code in order to get correct results. For example, the original C code assumed a size of one byte for the data type `char` at certain points but on the DSP56600 a `char` data type is 16 bits wide.

In addition, the C model was extended with code for I/O to the external world according to a specially defined interface (see Section 4.1.1). Direct file I/O is not available on the DSP56600 and therefore, a strategy had to be developed for getting data in and out of the (simulated) DSP in connection with the simulator.

In the second step, the synthesized C model was compiled into assembly code for the DSP56600. For this task we used the standard C compiler supplied by Motorola for their processor. It should be noted, however, that this is simply a retargeted GNU C compiler (gcc) which is not very well optimized for DSP architectures. Therefore, even though compilation was performed with all optimizations (including an assembly-level post-optimizer) enabled, results would be expected to be significantly better with a specialized compiler (e.g., a commercial compiler for the DSP56600).

Finally, an instruction set simulator (ISS) was used to run the compiled program and to cycle-accurately simulate execution of the code on a DSP56600. We used the ISS library provided by Motorola for the DSP56600 family and created a customized version of the simulator on top of that. Customization became necessary in order to implement the processor I/O interface (see Section 4.1.1) on the ISS side, feeding the simulated program with the correct data and reading

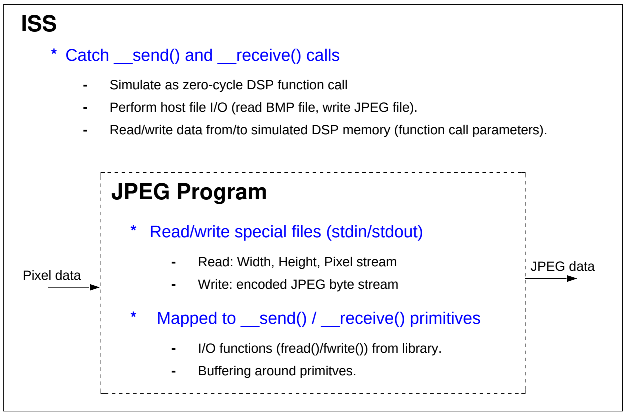


Figure 5: JPEG/ISS interface

the encoding results back. In addition, the simulator was extended to collect detailed results about execution cycles and instructions needed for each part of the code. The results will be presented and discussed in Section 4.1.2.

4.1.1 ISS Interface

As mentioned previously, the JPEG encoding module running on the simulated DSP core inside the ISS had no direct access to files stored on the simulation host. Therefore, a customized way of feeding the input data into the program and reading the results back was developed to test the encoder.

Figure 5 shows the interfacing between the JPEG program and the simulator. The C code of the JPEG encoder was wrapped in a special testbench and modified to read and write data to/from the special file handles `stdin` and `stdout` which are part of the standard C libraries supplied with the compiler. On the input side the JPEG program receives information about the width and height of the picture (2 bytes each) followed by the picture's pixels (1 byte per pixel) in raster order. On the output side, the JPEG program writes the encoded byte stream directly to standard output.

The file access functions (`fread()` and `fwrite()`) as supplied by Motorola with their C compiler are mapped internally to calls of `__send()` and `__receive()` primitives. The library implementations of the file access functions simply provide some additional buffering around the primitive calls.

This special program is then executed inside the simulator. During simulation the ISS monitors the instruction stream for calls to the I/O primitives `__send()` and `__receive()`. If such a call is detected (i.e., a jump to the corresponding subroutine) the ISS

Routine	Calls	Cycles		
		Total	Max.	Avg.
HandleData	180	3,527,471	360,887	19,597
DCT	180	8,045,786	45,001	44,698
Quantization	180	1,000,440	5,558	5,558
HuffmanEncode	180	2,084,886	20,060	11,582
fread	11,136	1,886,757	280	169
fwrite	1,693	444,028	437	262
Total	1	14,668,976	14,668,976	14,668,976

Table 1: Execution cycles on DSP56600 for software implementation

performs the corresponding file I/O operation on the host side (i.e., reading from the input file or writing to the output file) reading or writing the necessary data directly to/from the simulated DSP memory. For the program running in the ISS this simulates a zero-cycle function call with the required semantics.

4.1.2 Performance Analysis

For estimation of the software execution times on the DSP56600 we ran the sample image with a size of 116 by 96 pixels (corresponding to 180 blocks of 8x8 pixels) through the JPEG encoder executing in the simulator as described in the previous section. Table 1 summarizes the total cycles and the maximum and average number of cycles per call spent in the different functions of the C code at the first three levels of hierarchy. (A detailed table with all results can be found in Appendix A.)

This table also shows the number of cycles of communication overhead needed for buffering around the `__send()` and `__receive()` primitives in the file `fread()` and `fwrite()` functions. Read cycles are completely included in the cycles for *HandleData*. That is, in order to get the correct number of computational cycles for *HandleData* the file-read cycles had to be subtracted from the given values. The write cycles, however, are distributed between part of the *HandleData* block (which writes the header and footer for a JPEG picture format) and part of the *HuffmanEncode* block (which writes the body of the transformed picture).

Table 2 finally shows the execution times without the above mentioned communication overhead for a single block of 8 by 8 pixels, for the sample image of size 116x96. It also shows projected values for an image of standard (1024x768) screen resolution. The execution times are based on the maximum clock frequency of 60 MHz for the DSP56600 processor.

Note that since communication is not included in

	8x8 Block	116x96	1024x768
HandleData	142 us	25 ms	1.75 s
DCT	745 us	134 ms	9.15 s
Quantization	93 us	17 ms	1.14 s
HuffmannEncode	162 us	29 ms	1.98 s
Total	1142 us	205 ms	14.02 s

Table 2: Estimated execution times for different picture sizes for software implementation

these values, the time needed for reading and writing data items (at least one cycle per transfer) has to be added to the execution times. The communication overhead includes

- 4 reads plus 324 writes (or at least 5.4 ms) for the JPEG header (*HandleData*);
- 64 reads (in *HandleData*) plus 7.6 writes (in huffman encoding on average) per 8x8 block (at least 1.1 ms per block).

In reality, the communication overhead will probably be more than one cycle per transfer. The communication delays given above (based on 60 MHz clock frequency) would have to be scaled accordingly in this case.

4.2 Hardware Estimation

Hardware estimation was performed by estimating the simulation of the RTL VHDL model of the JPEG encoder using Motorola technology library cells. Basically the procedure involves 2 steps as following:

1. select components from certain library and design a proper architecture from the given SpecC description;
2. analyze the critical path and manually calculate

the number of states and clock cycles to determine the execution time of the hardware.

4.2.1 Architecture model

Resource allocation and scheduling are two major tasks of architecture design.

Resource allocation determines the number and types of RT (register transfer) components to be used in the design. Components are taken out of a library which may contain multiple types of functional units, each with different characteristics. These components include functional units, storage units and communication units. In this project, we chose them from a Motorola component library.

Scheduling is the key to determining whether architecture design will satisfy the timing constraints. It assigns operations in the behavioral description to control steps which correspond to clock cycles. The number of clock cycles along with the clock period thus determine the execution time of the hardware. Scheduling also affects resource allocation. Remember that within a control step a separate functional unit is require to execute each operation assigned to that step. Hence, the total number of functional units required in a control step directly corresponds to the number of operations scheduled in it.

The first phase of designing an architecture starts with an initial set of resources, i.e. storage elements for every variable, functional units for every operator and connections for every data transfer. We initially chose the fastest resource or the one with the maximum number of stages in case of a pipelined resource to get the best performance.

Next, the feasibility analysis phase was started by gradually exploiting the parallelism in the specification. Simple scheduling algorithms were performed to expose the parallelism based on data dependencies. The goal was to find a schedule which utilizes the components maximally therefore requiring a minimal number components. We achieved this objective by uniformly distributing operations of the same type into all available control steps. A uniform distribution ensures that resources allocated to operations in one control step are used efficiently in other control steps, leading to a high resource utilization rate.

After resource allocation and scheduling, binding then need to be performed. Binding is the process of mapping the variables and operations in the scheduled RTL model onto functional, storage and interconnection units while ensuring that the design functions correctly on the selected set of components. Binding helps to decide what components are selected and how to connect between them. After binding, the final architecture is developed. The binding strategy is as follows:

- Each operation in the behavioral description must be mapped onto one of the selected functional units.
- Storage binding maps constants, variables, and arrays in the behavioral description to storage elements (e.g. registers, RF and memory units) in the datapath. Constants, such as the quantization matrix which was used by the Huffman encoding algorithm, are usually stored in memory. Variables are stored in memory and functional unit output temporary registers or in a register file (RF). Variables whose lifetime intervals do not overlap with each other may share the same temporary register or RF location.
- Every data transfer needs an interconnection path from its source to its sink. In this project, we used a MUX to select the correct input path to an unit.

Figure 6 shows the architecture used in this project. (see appendix B for detail architecture). The components used here are shown in Table 3. The architecture consists of the control unit (left part) and the datapath (right part). The control unit consists of a control logic unit, a state register and a next state logic unit. The datapath consists of counters, address-generators, a memory, a comparator, a multiplier, a shifter, an ALU, registers, and a register file. MUXs are used to connect components in this project (In Figure 6, buses are used to represent all the MUXs).

In this architecture, because memory access time is slower than reigister file access time, both memory and a register file are used to speed up the execution time. Memory is used to store constant, input and output arrays. The register file is used to store temporal variables and intermediate results. The memory and the register file, each of which has two read ports and one write port, were selected based on efficiency considerations.

4.2.2 Performance Analysis

Performance metrics can be classified into three categories: clock cycle, control steps and execution times. The execution time can be computed as follows:

$$\text{execution\_time} = \text{num\_cycles} \times \text{clock\_cycle}$$

Component	Operations	#	Delay(ns)
ALU	add,sub	1	3.02
Shifter	shl,shr	1	2.25
Multiplier	mult	1	4.09
CMP	comp	1	1.22
MUX16	choose data	6	.66
MUX8	choose data	10	.50
MEM	storage access	1	4.20
COUNTERS	array index generation	3	.40
ADR-GEN	memeory address generation	3	1.94
REG32	temp. storage access	5	.75
REG32(setup)			.59
Register File	frequent storage access	1	1.46
Control Unit	next state generation	1	3.85

Table 3: List of Functional Units

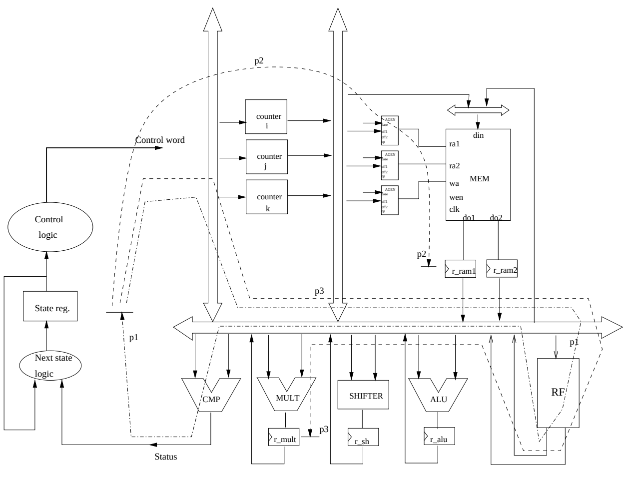


Figure 6: RTL structural diagram & critical path candidates

Given the FSMD design shown in Figure 6 the minimum reasonable clock cycle can be determined as the maximum of the critical path candidates as follows:

$$\begin{aligned} T_{p1} &= \text{delay}(SR) + \text{delay}(CL) + \text{delay}(RF) \\ &\quad + \text{delay}(MUX16) + \text{delay}(CMP) \\ &\quad + \text{setup}(SR) + \text{delay}(NL) \\ &= 12.38\text{ns} \end{aligned}$$

$$\begin{aligned} T_{p2} &= \text{delay}(SR) + \text{delay}(CL) \\ &\quad + \text{delay}(AGEN) + \text{delay}(MUX8) \\ &\quad + \text{delay}(MRD) + \text{setup}(MR) \\ &= 11.83\text{ns} \end{aligned}$$

$$\begin{aligned} T_{p3} &= \text{delay}(SR) + \text{delay}(CL) \\ &\quad + \text{delay}(RF) + \text{delay}(MUX16) \\ &\quad + \text{delay}(MUL) + \text{setup}(RR) \\ &= 11.40\text{ns} \end{aligned}$$

Here delay(SR) is the delay of reading the state register, delay(CL) is the delay of the control logic, delay(CMP) is the delay of the comparator, delay(NL) is the delay of the next state logic, setup(SR) is the setup time of the state register, delay($AGEN$) is the delay of memory address generation, delay(MRD) is the delay of reading the memory, delay(MUL) is the delay of the multiplier, delay(MUX) is the delay of the multiplexer, delay(RF) is the delay of reading data from the register file, setup(MR) is the setup time of the register connected to the memory read port, setup(RR) is the setup time of the registers storing the functional unit results. Hence, the minimum clock cycle is:

$$\text{clock\_cycle} = \max(T_{p1}, T_{p2}, T_{p3}) = 12.38\text{ns}.$$

Function	Cycle formula[1]
HandleData	$1417wh + 71h + 4748$
DCT	$1056wh$
Quantization	$449wh$
HuffmanEncode	$3935wh$
Total	$7309wh + 71h + 4748$

Table 4: Execution cycles for hardware implementation

[1] w equals 15 and h equals 12. w and h are the MCU width and MCU height. Since our image size is 116x96 pixels, the width and height are obtained by dividing the image width and height by 8 and rounding up.

	8x8 Block	116x96	1024x768
HandleData	77.19 us	3.23 ms	0.21 s
DCT	13.07 us	2.35 ms	0.16 s
Quantization	5.56 us	1.00 ms	0.07 s
HuffmannEncode	48.71 us	8.77 ms	0.59 s
Total	144.43 us	15.35 ms	1.03 s

Table 5: Estimated execution times for different picture sizes for hardware implementation

The number of execution cycles is estimated based on SpecC code and the architecture. Table 4 shows the formula of clock cycles for each top level module as a function of image width and height. The cycles needed for reading and writing data are excluded in the formula, but the cycles for writing the output file header are included.

Table 5 finally shows the execution times for images with different sizes. The operating frequency of the hardware design will be 80.8Mhz.

	8x8 Block	116x96	1024x768
Software	1142 us	205 ms	14.02 s
Hardware	144.43 us	15.35 ms	1.03 s
Ratio(H/S)	7.9	13.4	13.6

Table 6: Comparison of Software and Hardware execution times for different picture sizes.

4.3 Result

The values in Table 2 and Table 4 are compared in Table 6 which shows that the hardware implementation is roughly 10 times faster than the software implementation. Table 2 also shows that the DCT accounts for more than half of the execution cycles. Therefore, speeding up the DCT part will provide the largest gain in execution time. However, even if the DCT could be executed in zero time, the speedup will not be large enough. It is evident that only a pure hardware solution promises to deliver the necessary improvement of a factor of 10.

5 Behavioral RTL Design

On the RT level, time is divide into intervals called control states. The register-transfer description, which specifies for each control state the condition to be tested, all register transfers to be executed, and

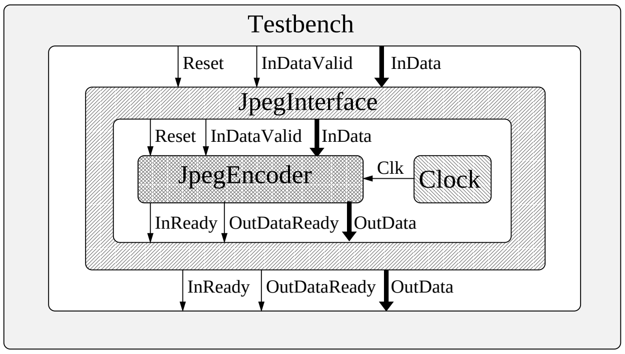


Figure 7: Behavioral RTL model of JPEG

the next control state to be entered, is called behavioral RTL [5]. For example, in Figure 2, three lines of VHDL codes beside "RTL behavior Model" block represent RTL behavior specification of "a[i]=b+1". Among these codes, signal "r_ram1" and "r_alu" represents registers; signal 'b' and array 'a' represent memory; signal 'i' represents a counter (see Figure 6). These three lines of codes belong to three sequential states. In Section 4 we decided on a purely hardware implementation for the Jpeg encoder. The next step is to manually refine SpecC specification into behavioral RTL description based on the RTL architecture developed in Section 4.

Figure 7 shows the structure of RTL code of the JPEG Encoder. *JpegEncoder* is the actual JPEG core which converts an input stream of the image pixels into a compressed format, called JFIF. *Clock* is a clock generator. *JpegInterface* is the asynchronous interface between the *JpegEncoder* and the *Testbench*. The *Testbench* provides the image data to *JpegInterface* which in turn provides the data to *JpegEncoder*. As one can see, *JpegEncoder* itself is synchronous, having a clock input. However, the communication between the *Testbench* and *JpegEncoder* is asynchronos as described below.

The input protocol for *JpegEncoder* is a simple handshake. When *JpegEncoder* needs some data, it asserts *InReady*. Once *InReady* is asserted, the *Testbench* puts the data into *InData* and asserts *InDataValid*. Then, *JpegEncoder* reads the data from *InData*.

The output of *JpegEncoder* to the *Testbench* is straightforward and does not involve a handshake since the *Testbench* is always assumed to be ready to accept output. To send the output, *JpegEncoder* asserts *OutDataReady* and puts the data into *OutData*.

Simulation was performed on the behavioral RTL description to verify that it has the same functionality as the SpecC specification. This simulation also gives roughly the same amount of execution time as predicted in Table 5 thus showing the validity of the hardware estimation.

6 Gate Level Design

In Section 5, RTL model of the JPEG Encoder is described. This model is based on the target architecture shown in Figure 7. Since at this stage the behavior is only scheduled but not bound, we refer this style of coding as RTL behavior.

There are two methods to refine the RTL behavior into synthesizable VHDL code.

The first method explicitly partitions the design into a controller and a datapath. The datapath unit performs complex numerical computation or data manipulations based on the value of the control signals issued by the control unit at every cycle. In order to make it synthesizable, The RTL behavioral model has to be refined into a structural model which consists of both the control unit and the datapath unit communicating via a set of control signals. For example, in Figure 2, the codes beside "RTL structural Model" block represents RTL structural specification. "WE_I" is memory's write enable signal, "CS_I" is the memory's chip select signal, "Addr" is the memory's address. "ALU_op", "ALU_sel_1", "ALU_sel_2" are the ALU's control signals. The datapath unit is, in turn, specified as a netlist of RTL components such as adders and registers. The control unit is specified as a finite state machine. Both units are then synthesizable given the current capability of synthesis tools.

The second method exploits the power of the synthesis tools as much as possible. While in theory the RTL behavioral code should be synthesizable in first method, one problem makes it impossible given the limitation of the Synopsys tool set, which blindly binds all array variables into registers. In order to bind arrays into the memory, the memory unit has to be separated from the rest of logic. The resultant architecture consists of two units. One is the memory unit, whose inputs consist of control signals and data written to memory and outputs consist of data read from memory. The other unit (FSMD unit) represents the rest of logic. It not only inherits all the inputs and outputs of previous model, but also adds new inputs and outputs to communicate with the memory unit. Since we use Synopsys tools to synthesize codes, we call this model the "Synopsys RTL Model".

This second method not only avoids the memory binding problem, but also is considerably easier to implement than the first one. Hence, in this project, the second method was chosen.

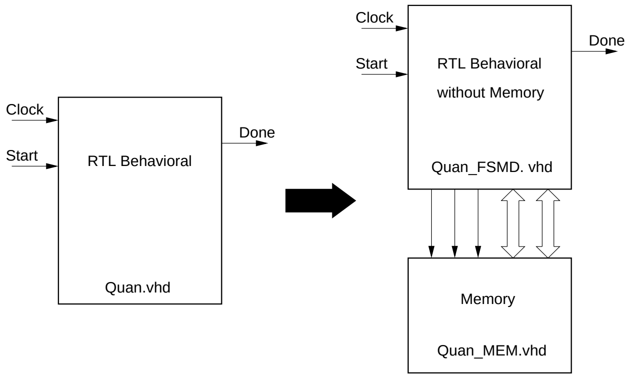


Figure 8: FSMD model with memory

Quantization (Quan.vhd), one of the four top level modules, was chosen to illustrate our model.

Quan.vhd was manually partitioned into 2 parts: memory (Quan_MEM.vhd) and synthesizable FSMD (Quan_FSMD.vhd) (Figure 8). The memory part, Quan_MEM.vhd, was modeled after SAMSUNG's asynchronous memory with memory access time 12ns. The FSMD part, Quan_FSMD.vhd, is basically the same as the original RTL code Quan.vhd except that memory accesses are now replaced with memory control signals. For example, in Figure 2, seven lines of codes beside "Synopsys RTL Model" block represent the Synopsys RTL model's specification. Compared with the RTL behavior model's specification, the code of the first state (r_ram1=b) and the third state (a[i]=r_alu) which involve memory accesses have been replaced with control signal changes. Simulation was then performed on the modified code to ensure the correctness of the partitioning.

One important issue here is the synchronization between the FSMD and memory when the FSMD has a shorter clock period than the memory access time. In this case, some idle states must be inserted into the FSMD to wait for the completion of a memory access. Since the exact clock period of the FSMD is unknown at this point, the exact number of idle states to insert cannot be determined. Hence the FSMD clock period was first estimated before we modified the RTL code. After the FSMD part was synthesized, the real clock period was then found. In case the synchronization timing constraint is not satisfied, the number of idle cycles is adjusted. While such a sequence of steps may need to be repeated to converge on the correct number of idle cycles, this rarely occurs in practice.

Then Quan_FSMD.vhd was fed into the Synopsys

Design Compiler for logic synthesis and optimization using lsi_10k.db as the target library. The output style of synthesized and optimized design is a gate level netlist which has little resemblance to the original RTL architecture. The final design uses 5400 gates and has a longest delay of 16ns, which is taken as the FSMD clock period. Simulation on VHDL format output was performed to test the correctness of the synthesis.

7 Conclusion

In this project, the performance of software implementation and hardware implementation of a JPEG encoder was estimated and compared, and the hardware implementation was chosen because it is roughly 10 times faster than the software implementation.

In the hardware implementation process, the original specification, about 1,100 lines of C code, was translated into SpecC description of 1,400 lines. Based on the SpecC code and the designed architecture, an RTL behavioral specification was produced of about 5,000 lines.

The RTL behavioral specification provides a high level of understandability but it cannot be synthesized with the current synthesis tools because of memory-handling problems. To make this specification synthesizable, the RTL behavioral specification can be translated to an RTL structural specification which is still fairly readable, but this type of translation requires a considerable amount of time.

A new approach to work around the memory problem, using Synopsys RTL, was taken in this project to refine the RTL behavior into synthesizable code where only memory was set aside to work around the memory-handling problem with current synthesis tools. It has better readability than the RTL structural specification and it also requires less time for the translation from an RTL behavioral specification. The problem of synchronization between the memory partition and the rest of the RTL architecture was then addressed and the required memory idle time was folded into the final design.

References

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

[2] D. Gajski, R. Dömer, J. Zhu. "IP-centric Methodology and Design with the SpecC Language," in *System Level Synthesis*, Proceedings of the NATO ASI on System Level Sythesis for Electronic Design, Il Ciocco, Lucca, Italy, August 1998, edited by A. Jerraya, J. Mermet, Kluwer Academic Publishers, 1999.

[3] R. Dömer, J. Zhu, D. Gajski, *The SpecC Language Reference Manual*, University of California, Irvine, Technical Report ICS-TR-98-13, March 1998.

[4] Motorola, Inc., Semiconductors Products Sector, DSP Division, *DSP56600 16-bit Digital Signal Processor Family Manual*, DSP56600FM/AD, 1996.

[5] D. Gajski, A. Wu, N. Dutt, S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluver Academic Publishers, 1992.

A Software Estimation Results

This appendix contains the results for simulating execution of the JPEG encoding program on the Motorola DSP56600 processor. The table lists both instruction and cycle counts (total and maximum/average per call) for each function in the call graph of the C program.

Routine		calls	cycles			instructions		
			max	total	avg.	max	total	avg
Fmain	1d22	1	14675166	14675166	14675166	8869428	8869428	8869428
FReadWord	1d05	2	286	460	230	158	260	130
FJpeg	1cbe	1	14668976	14668976	14668976	8866670	8866670	8866670
FHandleData	1b7f	180	360887	3527471	19597	180630	1838619	10214
FWriteByte	177f	290	468	85501	294	241	43495	149
FWriteWord	174c	11	593	6515	592	299	3289	299
FJpegDefaultHuffman	1a49	1	21281	21281	21281	10380	10380	10380
FSpecifiedHuffman	14e5	2	18616	21221	10610	9055	10351	5175
FWriteMarker	188d	7	588	4068	581	297	2061	294
FSizeTable	1529	2	4525	5420	2710	2255	2695	1347
FCodeTable	1568	2	5732	6433	3216	2499	2804	1402
FOrderCodes	15ac	2	5849	6298	3149	2760	2970	1485
FWriteAPP0	18ba	1	5375	5375	5375	2726	2726	2726
FWriteSOF	194e	1	3894	3894	3894	1984	1984	1984
FWriteDQT	18f2	1	22301	22301	22301	11170	11170	11170
FWriteDHT	1927	1	69713	69713	69713	34753	34753	34753
FWriteHuffman	15cd	2	58387	67876	33938	29061	33822	16911
FWriteSOS	1985	1	2985	2985	2985	1523	1523	1523
FDCT	1b65	180	45001	8045786	44698	29837	5346377	29702
FPreshiftDctMatrix	19af	180	782	140760	782	456	82080	456
FChenDct	103f	180	42496	7594886	42193	28515	5108417	28380
FBoundDctMatrix	19c0	180	1690	304200	1690	845	152100	845
FQuantization	1b5a	180	5558	1000440	5558	3413	610758	3393
FQuantize	19e0	180	5542	997560	5542	3404	609138	3384
FHuffmanEncode	1b39	180	20060	2084886	11582	10776	1066597	5925
FEncodeHuffman	1628	2017	1111	700158	347	572	393052	194
FWriteBits	1797	3854	1048	982970	255	537	561394	145
FZigzagMatrix	1a3c	180	716	128880	716	452	81360	452
FEncodeDC	1a63	180	1105	128661	714	586	71335	396
FUseDCHuffman	1675	182	12	2184	12	5	910	5
FEncodeAC	1aac	180	18199	1818525	10102	9714	909042	5050
FUseACHuffman	166c	182	12	2184	12	5	910	5
Ffread	1e3d	11136	280	1886757	169	138	915560	82
Ffgetc	1de2	4	180	383	95	93	204	51
Ffwrite	1ee3	1693	437	444028	262	234	216076	127
Ffree	20f3	1	11	11	11	4	4	4
Fmalloc	1feb	1	160	160	160	90	90	90
Fmemcpy	20fb	12840	647	668734	52	304	308656	24
Ffflush	2e3f	3	165	251	83	87	141	47
Fsprintf	215f	1	3884	3884	3884	1605	1605	1605
Fstrlen	2e93	1	233	233	233	117	117	117
Ffprintf	2126	1	1761	1761	1761	840	840	840
Ffill_decimal_word	289d	2	236	407	203	134	225	112
F__doprnt	2351	2	3818	5552	2776	1570	2397	1198
F__read	2135	44	73	3212	73	36	1584	36
F__receive	100e	103	5	412	4	1	103	1
F__send	100b	74	4	266	3	1	74	1
F__write	21ba	15	76	1140	76	40	600	40

B SpecC Code

B.1 JPEG Encoder

B.1.1 global.sc

```
//----------------------------------------------------------------------
// Global definitions for testbench
//----------------------------------------------------------------------
//
// 05/10/99  A. Gerstlauer
//

#include <stdio.h>
#include <stdlib.h>


// Error messages
void error(const char *Format, const char* Name)
{
  fprintf(stderr, Format, Name);
  exit(1);
}


FILE* openStdout()
{
  return stdout;
}

```

B.1.2 chann.sc

```
#ifndef __CHANNEL_
#define __CHANNEL_

typedef char BYTE;

interface iBlckSendByte {
        void send(BYTE val);
};

interface iBlckRecvByte {
        BYTE receive(void);
};

interface iBlckSendInt {
        void send(int val);
};

interface iBlckRecvInt {
        int receive(void);
};

interface iBlckSendBlock {
        void send(int val[64]);
};

interface iBlckRecvBlock {
        void receive(int val[64]);
};

channel cSyncByte(void) implements iBlckSendByte, iBlckRecvByte
{
BYTE message;
```

```
bool valid=false;
event sent, received;

void send(BYTE val){
        message = val;
        valid=true;
        notify(sent);
        if(valid)
                wait(received);
}

BYTE receive(void){
        BYTE local_message;

        if(!valid)
                wait(sent);
        local_message = message;
        valid=false;
        notify(received);
        return local_message;
}
};

channel cSyncInt(void) implements iBlckSendInt, iBlckRecvInt
{
int message;
bool valid=false;
event sent, received;

void send(int val){
        message = val;
        valid=true;
        notify(sent);
        if(valid)
                wait(received);
}

int receive(void){
        int local_message;

        if(!valid)
                wait(sent);
        local_message = message;
        valid=false;
        notify(received);
        return local_message;
}
};

channel cSyncBlock(void) implements iBlckSendBlock, iBlckRecvBlock
{
int message[64], i;
bool valid=false;
event sent, received;

void send(int val[64]){
        for(i=0; i<64; i++)
                message[i] = val[i];
        valid=true;
        notify(sent);
        if(valid)
                wait(received);
}

void receive(int val[64]){
        if(!valid)
```

```
                wait(sent);
        for(i=0; i<64; i++)
                val[i] = message[i];
        valid=false;
        notify(received);
    }
};
#endif
```

B.1.3 jpeg.sc

```
/***********************************************************************
* IO functions are changed to communicate with testbench
* and the useless functions are erased
* by Hongxing Li in May 13, 1999
***********************************************************************/

/* JPEG encoder */

import "global";
import "chann";
/***********************************************************************
* DCT
***********************************************************************/

#define NO_MULTIPLY

#ifdef NO_MULTIPLY
#define LS(r,s) ((r) << (s))
#define RS(r,s) ((r) >> (s))        /* Caution with rounding... */
#else
#define LS(r,s) ((r) * (1 << (s)))
#define RS(r,s) ((r) / (1 << (s))) /* Correct rounding */
#endif

#define MSCALE(expr)  RS((expr),9)

/* Cos constants */

#define c1d4 362L
#define c1d8 473L
#define c3d8 196L
#define c1d16 502L
#define c3d16 426L
#define c5d16 284L
#define c7d16 100L

/*
  VECTOR_DEFINITION makes the temporary variables vectors.
  Useful for machines with small register spaces.
*/

#ifdef VECTOR_DEFINITION
#define a0 a[0]
#define a1 a[1]
#define a2 a[2]
#define a3 a[3]
#define b0 b[0]
#define b1 b[1]
#define b2 b[2]
#define b3 b[3]
#define c0 c[0]
#define c1 c[1]
#define c2 c[2]
#define c3 c[3]
#endif
```

```
/*START*/
/*BFUNC

ChenDCT() implements the Chen forward dct. Note that there are two
input vectors that represent x=input, and y=output, and must be
defined (and storage allocated) before this routine is called.

EFUNC*/

void ChenDct(int *x,int *y)
{
  register int i;
  register int *aptr,*bptr;
#ifdef VECTOR_DEFINITION
  register int a[4];
  register int b[4];
  register int c[4];
#else
  register int a0,a1,a2,a3;
  register int b0,b1,b2,b3;
  register int c0,c1,c2,c3;
#endif

  /* Loop over columns */

  for(i=0;i<8;i++)
    {
      aptr = x+i;
      bptr = aptr+56;

      a0 = LS((*aptr+*bptr),2);
      c3 = LS((*aptr-*bptr),2);
      aptr += 8;
      bptr -= 8;
      a1 = LS((*aptr+*bptr),2);
      c2 = LS((*aptr-*bptr),2);
      aptr += 8;
      bptr -= 8;
      a2 = LS((*aptr+*bptr),2);
      c1 = LS((*aptr-*bptr),2);
      aptr += 8;
      bptr -= 8;
      a3 = LS((*aptr+*bptr),2);
      c0 = LS((*aptr-*bptr),2);

      b0 = a0+a3;
      b1 = a1+a2;
      b2 = a1-a2;
      b3 = a0-a3;

      aptr = y+i;

      *aptr = MSCALE(c1d4*(b0+b1));
      aptr[32] = MSCALE(c1d4*(b0-b1));

      aptr[16] = MSCALE((c3d8*b2)+(c1d8*b3));
      aptr[48] = MSCALE((c3d8*b3)-(c1d8*b2));

      b0 = MSCALE(c1d4*(c2-c1));
      b1 = MSCALE(c1d4*(c2+c1));

      a0 = c0+b0;
      a1 = c0-b0;
      a2 = c3-b1;
      a3 = c3+b1;
```

```
      aptr[8] = MSCALE((c7d16*a0)+(c1d16*a3));
      aptr[24] = MSCALE((c3d16*a2)-(c5d16*a1));
      aptr[40] = MSCALE((c3d16*a1)+(c5d16*a2));
      aptr[56] = MSCALE((c7d16*a3)-(c1d16*a0));
    }

  for (i=0;i<8;i++)
    {         /* Loop over rows */
      aptr = y+LS(i,3);
      bptr = aptr+7;

      c3 = RS((*(aptr)-*(bptr)),1);
      a0 = RS((*(aptr++)+*(bptr--)),1);
      c2 = RS((*(aptr)-*(bptr)),1);
      a1 = RS((*(aptr++)+*(bptr--)),1);
      c1 = RS((*(aptr)-*(bptr)),1);
      a2 = RS((*(aptr++)+*(bptr--)),1);
      c0 = RS((*(aptr)-*(bptr)),1);
      a3 = RS((*(aptr)+*(bptr)),1);

      b0 = a0+a3;
      b1 = a1+a2;
      b2 = a1-a2;
      b3 = a0-a3;

      aptr = y+LS(i,3);

      *aptr = MSCALE(c1d4*(b0+b1));
      aptr[4] = MSCALE(c1d4*(b0-b1));
      aptr[2] = MSCALE((c3d8*b2)+(c1d8*b3));
      aptr[6] = MSCALE((c3d8*b3)-(c1d8*b2));

      b0 = MSCALE(c1d4*(c2-c1));
      b1 = MSCALE(c1d4*(c2+c1));

      a0 = c0+b0;
      a1 = c0-b0;
      a2 = c3-b1;
      a3 = c3+b1;

      aptr[1] = MSCALE((c7d16*a0)+(c1d16*a3));
      aptr[3] = MSCALE((c3d16*a2)-(c5d16*a1));
      aptr[5] = MSCALE((c3d16*a1)+(c5d16*a2));
      aptr[7] = MSCALE((c7d16*a3)-(c1d16*a0));
    }

  /* We have an additional factor of 8 in the Chen algorithm. */

  for (i=0,aptr=y;i<64;i++,aptr++)
    *aptr = (((*aptr<0) ? (*aptr-4) : (*aptr+4))/8);
}

/**********************************************************
huffman.c

This file represents the core Huffman routines, most of them
implemented with the JPEG reference. These routines are not very fast
and can be improved, but comprise very little of software run-time.

**********************************************************/

#define         XHUFF           struct huffman_standard_structure
#define         EHUFF           struct huffman_encoder

XHUFF {
        int bits[36];
```

```
        int huffval[257];
};

EHUFF {
        int ehufco[257];
        int ehufsi[257];
};

/*PUBLIC*/
int WriteBits(int, int, iBlckSendByte);
int WriteByte(int, iBlckSendByte);
static void SizeTable();
static void CodeTable();
static void OrderCodes();

void PrintHuffman();
void PrintTable(int *);

static int huffsize[257];
static int huffcode[257];
static int lastp = 0;
static XHUFF *Xhuff=0;
static EHUFF *Ehuff=0;

static XHUFF ACXhuff, DCXhuff;
static EHUFF ACEhuff, DCEhuff;

/*START*/
void SpecifiedHuffman(int *bts,int *hvls)
{
        int i;
        int accum;

        for(accum=0,i=0;i<16;i++)
        {
                accum+= bts[i];
                Xhuff->bits[i+1] = bts[i];       /* Shift offset for internal specs.*/
        }
        for(i=0;i<accum;i++)
        {
                Xhuff->huffval[i] = hvls[i];
        }
        SizeTable();                             /*From Xhuff to Ehuff */
        CodeTable();
        OrderCodes();
}

static void SizeTable()
{
  int i,j,p;

  for(p=0,i=1;i<17;i++)
    {
      for(j=1;j<=Xhuff->bits[i];j++)
        {
          huffsize[p++] = i;
        }
    }
  huffsize[p] = 0;
  lastp = p;
}

static void CodeTable()
{
  int p,code,size;
```

```
  p=0;
  code=0;
  size = huffsize[0];
  while(1)
    {
      do
        {
          huffcode[p++] = code++;
        }
      while((huffsize[p]==size)&&(p<257)); /* Overflow Detection */
      if (!huffsize[p]) /* All finished. */
        {
          break;
        }
      do                  /* Shift next code to expand prefix. */
        {
          code <<= 1;
          size++;
        }
      while(huffsize[p] != size);
    }
}

static void OrderCodes()
{
  int index,p;

  for(p=0;p<lastp;p++)
    {
      index = Xhuff->huffval[p];
      Ehuff->ehufco[index] = huffcode[p];
      Ehuff->ehufsi[index] = huffsize[p];
    }
}

void WriteHuffman(iBlckSendByte Data_Ch)
{
  int i,accum;

  if (Xhuff)
    {
      for(accum=0,i=1;i<=16;i++)
        {
          WriteByte(Xhuff->bits[i], Data_Ch);
          accum += Xhuff->bits[i];
        }
      for(i=0;i<accum;i++)
        {
          WriteByte(Xhuff->huffval[i], Data_Ch);
        }
    }
  else
    {
      printf("Null Huffman table found.\n");
    }
}

void EncodeHuffman(int value, iBlckSendByte Data_Ch)
{
        if (Ehuff->ehufsi[value]) {
                WriteBits(Ehuff->ehufsi[value],Ehuff->ehufco[value], Data_Ch);
        }
        else {
                printf("Null Code for [%d] Encountered:\n", value);
                printf("*** Dumping Huffman Table ***\n");
                PrintHuffman();
```

```
                printf("***\n");
                exit(-1);
            }
}

void UseACHuffman()
{
  Xhuff = &ACXhuff;
  Ehuff = &ACEhuff;
}

void UseDCHuffman()
{
  Xhuff = &DCXhuff;
  Ehuff = &DCEhuff;
}

void PrintHuffman()
{
  int i;

  if (Xhuff)
    {
      printf("Bits: [length:number]\n");
      for(i=1;i<9;i++)
        {
          printf("[%d:%d]",i,Xhuff->bits[i]);
        }
      printf("\n");
      for(i=9;i<17;i++)
        {
          printf("[%d:%d]",i,Xhuff->bits[i]);
        }
      printf("\n");

      printf("Huffval:\n");
      PrintTable(Xhuff->huffval);
    }
  if (Ehuff)
    {
      printf("Ehufco:\n");
      PrintTable(Ehuff->ehufco);
      printf("Ehufsi:\n");
      PrintTable(Ehuff->ehufsi);
    }
}

void PrintTable(int *table)
{
  int i,j;

  for(i=0;i<16;i++)
    {
      for(j=0;j<16;j++)
        {
          printf("%2x ",*(table++));
        }
      printf("\n");
    }
}

/************************************************************************
** JPEG encoder
**              -> no resync, chen DCT
**
************************************************************************/
```

```
#define         BLOCKSIZE       64
#define         QuantizationMatrix  LuminanceQuantization

static int LuminanceQuantization[] = {
16, 11, 10, 16, 24, 40, 51, 61,
12, 12, 14, 19, 26, 58, 60, 55,
14, 13, 16, 24, 40, 57, 69, 56,
14, 17, 22, 29, 51, 87, 80, 62,
18, 22, 37, 56, 68, 109, 103, 77,
24, 35, 55, 64, 81, 104, 113, 92,
49, 64, 78, 87, 103, 121, 120, 101,
72, 92, 95, 98, 112, 100, 103, 99};

static int csize[] = {
0,
1,
2, 2,
3, 3, 3, 3,
4, 4, 4, 4, 4, 4, 4, 4,
5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8};

int ZigzagIndex[] =  /* Is zig-zag map for matrix -> scan array */
      {0,  1,  5,  6, 14, 15, 27, 28,
       2,  4,  7, 13, 16, 26, 29, 42,
       3,  8, 12, 17, 25, 30, 41, 43,
       9, 11, 18, 24, 31, 40, 44, 53,
      10, 19, 23, 32, 39, 45, 52, 54,
      20, 22, 33, 38, 46, 51, 55, 60,
      21, 34, 37, 47, 50, 56, 59, 61,
      35, 36, 48, 49, 57, 58, 62, 63};

int IZigzagIndex[] =
      {0,  1,  8, 16,  9,  2,  3, 10,
      17, 24, 32, 25, 18, 11,  4,  5,
      12, 19, 26, 33, 40, 48, 41, 34,
      27, 20, 13,  6,  7, 14, 21, 28,
      35, 42, 49, 56, 57, 50, 43, 36,
      29, 22, 15, 23, 30, 37, 44, 51,
      58, 59, 52, 45, 38, 31, 39, 46,
      53, 60, 61, 54, 47, 55, 62, 63};

int LastDC;

/* Default huffman table */
static int LuminanceDCBits[] = {
0x00, 0x01, 0x05, 0x01, 0x01, 0x01, 0x01, 0x01,
0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

static int LuminanceDCValues[] = {
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b};

static int LuminanceACBits[] = {
```

```
0x00, 0x02, 0x01, 0x03, 0x03, 0x02, 0x04, 0x03,
0x05, 0x05, 0x04, 0x04, 0x00, 0x00, 0x01, 0x7d};

static int LuminanceACValues[] = {
0x01, 0x02, 0x03, 0x00, 0x04, 0x11, 0x05, 0x12,
0x21, 0x31, 0x41, 0x06, 0x13, 0x51, 0x61, 0x07,
0x22, 0x71, 0x14, 0x32, 0x81, 0x91, 0xa1, 0x08,
0x23, 0x42, 0xb1, 0xc1, 0x15, 0x52, 0xd1, 0xf0,
0x24, 0x33, 0x62, 0x72, 0x82, 0x09, 0x0a, 0x16,
0x17, 0x18, 0x19, 0x1a, 0x25, 0x26, 0x27, 0x28,
0x29, 0x2a, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39,
0x3a, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49,
0x4a, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59,
0x5a, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69,
0x6a, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79,
0x7a, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89,
0x8a, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98,
0x99, 0x9a, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7,
0xa8, 0xa9, 0xaa, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6,
0xb7, 0xb8, 0xb9, 0xba, 0xc2, 0xc3, 0xc4, 0xc5,
0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xd2, 0xd3, 0xd4,
0xd5, 0xd6, 0xd7, 0xd8, 0xd9, 0xda, 0xe1, 0xe2,
0xe3, 0xe4, 0xe5, 0xe6, 0xe7, 0xe8, 0xe9, 0xea,
0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8,
0xf9, 0xfa};

/************************************************************************
**              I/O routines
************************************************************************/

int WriteWord(int code, iBlckSendByte Data_Ch)
{
        Data_Ch.send((char)(code>>8));
        Data_Ch.send((char)(code & 0xff));
        return 2;
}

int WriteByte(int code, iBlckSendByte Data_Ch)
{
        Data_Ch.send((char)code);
        return 0;
}

int WriteBits(int n, int code, iBlckSendByte Data_Ch)
{
        static unsigned char write_byte = 0;
        static left_bits = 8;
        int p;

        unsigned lmask[] = {
                0x0000,
                0x0001, 0x0003, 0x0007, 0x000f,
                0x001f, 0x003f, 0x007f, 0x00ff,
                0x01ff, 0x03ff, 0x07ff, 0x0fff,
                0x1fff, 0x3fff, 0x7fff, 0xffff
        };

        // synchronize buffer value
        if (n < 0) {
                if (left_bits < 8) {
                        n = left_bits;
                        Data_Ch.send(write_byte);
                        if (write_byte == 0xff) {
                                Data_Ch.send(0);
                        }
                        write_byte = 0;
```

```
				left_bits = 8;
			}
			else	n = 0;

			return n;
		}

		code &= lmask[n];
		p = n - left_bits;

		if (n == left_bits) {
			write_byte |= code;
			Data_Ch.send(write_byte);
			if (write_byte == 0xff) {
				Data_Ch.send(0);
			}

			write_byte = 0;
			left_bits = 8;
		}
		else if (n > left_bits) {
			write_byte |= (code >> p);
			Data_Ch.send(write_byte);
			if (write_byte == 0xff) {
				Data_Ch.send(0);
			}

			if (p > 8) {
				write_byte = (0xff & (code >> (p - 8)));
				Data_Ch.send(write_byte);
				if (write_byte == 0xff) {
					Data_Ch.send(0);
				}
				p -= 8;
			}

			write_byte = (code & lmask[p]) << (8 - p);
			left_bits = 8 - p;
		}
		else {
			write_byte |= (code << -p);
			left_bits -= n;
		}

		return n;
}

/***************************************************************************
** JPEG output stream
***************************************************************************/
#define		M_APP0		0xe0
#define		M_COM		0xfe
#define		M_DAC		0xcc
#define		M_DHP		0xde
#define		M_DHT		0xc4
#define		M_DNL		0xdc
#define		M_DQT		0xdb
#define		M_DRI		0xdd
#define		M_EOI		0xd9
#define		M_EXP		0xdf
#define		M_JPG		0xc8
#define		M_RST0		0xd0
#define		M_SOI		0xd8
#define		M_SOS		0xda
#define		M_SOF0		0xc0
```

```
void WriteMarker(int m, iBlckSendByte Data_Ch)
{
        Data_Ch.send(0xff);
        Data_Ch.send((char)m);
}

void WriteAPP0(iBlckSendByte Data_Ch)
{
        WriteMarker(M_APP0, Data_Ch);

        /* length */
        WriteWord(16, Data_Ch);

        /* identifier */
        WriteByte('J', Data_Ch);
        WriteByte('F', Data_Ch);
        WriteByte('I', Data_Ch);
        WriteByte('F', Data_Ch);
        WriteByte(0, Data_Ch);

        /* version */
        WriteWord(0x0102, Data_Ch);

        /* units */
        WriteByte(2, Data_Ch);

        /* Xdensity */
        WriteWord(0x001d, Data_Ch);

        /* Ydensity */
        WriteWord(0x001d, Data_Ch);

        /* Xthumbnail, Ythumbnail */
        WriteWord(0x0000, Data_Ch);
}

void WriteDQT(iBlckSendByte Data_Ch)
{
        int i;

        WriteMarker(M_DQT, Data_Ch);

        /* Lq */
        WriteWord(67, Data_Ch);

        /* define each quantization table */
        WriteByte(0, Data_Ch);

        for (i=0; i<64; i++) {
                WriteByte(QuantizationMatrix[IZigzagIndex[i]], Data_Ch);
        }
}

void WriteDHT(iBlckSendByte Data_Ch)
{
        WriteMarker(M_DHT, Data_Ch);

        WriteWord(0x4+0x20+0xc+0xa2, Data_Ch);

        /* Write DC Huffman */
        /* Tc, Th */
        WriteByte(0, Data_Ch);
        UseDCHuffman();
        WriteHuffman(Data_Ch);

        /* Write AC Huffman */
```

```
        /* Tc, Th */
        WriteByte(0x10, Data_Ch);
        UseACHuffman();
        WriteHuffman(Data_Ch);
}

void WriteSOF(int ImageHeight, int ImageWidth, iBlckSendByte Data_Ch)
{

        WriteMarker(M_SOF0, Data_Ch);

        /* Lf: frame header length */
        WriteWord(11, Data_Ch);

        /* P: precision */
        WriteByte(8, Data_Ch);

        /* Y: # lines, X: # samples/line */
        WriteWord(ImageHeight, Data_Ch);
        WriteWord(ImageWidth, Data_Ch);

        /* Nf: number of component in frame */
        WriteByte(1, Data_Ch);

        /* Frame component specification */
        /* Ci: component identifier */
        WriteByte(1, Data_Ch);
        /* Hi: horizontal sampling factor, Vi: vertical sampling factor */
        WriteByte(0x11, Data_Ch);
        /* Tqi: quantization table destination selector */
        WriteByte(0, Data_Ch);
}

void WriteSOS(iBlckSendByte Data_Ch)
{
        WriteMarker(M_SOS, Data_Ch);

        /* Ls: scan header length */
        WriteWord(8, Data_Ch);

        /* Ns: number of components in scan */
        WriteByte(1, Data_Ch);

        /* Csk: scan component selector */
        WriteByte(1, Data_Ch);

        /* Tdk, Tak: DC/AC entropy coding table selector */
        WriteByte(0, Data_Ch);

        /* Ss: start of spectral selection or predictor selection */
        WriteByte(0, Data_Ch);

        /* Se: end of spectral selection */
        WriteByte(63, Data_Ch);

        /* Ah, Al: successssive approximation bit position high/low */
        WriteByte(0, Data_Ch);
}

/*********************************************************************
**
*********************************************************************/
void PreshiftDctMatrix(int *matrix, int shift)
{
        int i;
        for(i=0; i<BLOCKSIZE; i++)
```

```
                matrix[i] -= shift;
}

void BoundDctMatrix(int *matrix, int bound)
{
        int i;
        for (i=0; i<BLOCKSIZE; i++) {
                if (matrix[i] < -bound)                 matrix[i] = -bound;
                else if (matrix[i] > bound)             matrix[i] = bound;
        }
}

void Quantize(int *matrix, int *qmatrix)
{
        int i, m, q;
        for (i=0; i<BLOCKSIZE; i++) {
                m = matrix[i];
                q = qmatrix[i];
                if (m > 0) {
                        matrix[i] = (m + q/2) / q;
                }
                else {
                        matrix[i] = (m - q/2) / q;
                }
        }
}

void ZigzagMatrix(int *imatrix, int *omatrix)
{
        int i, z;

        for (i=0; i<BLOCKSIZE; i++) {
                z = ZigzagIndex[i];
                omatrix[z] = imatrix[i];
        }
}

void JpegDefaultHuffman()
{
        UseDCHuffman();
        SpecifiedHuffman(LuminanceDCBits, LuminanceDCValues);
        UseACHuffman();
        SpecifiedHuffman(LuminanceACBits, LuminanceACValues);
}

void EncodeDC(int coef, iBlckSendByte Data_Ch)
{
        int s, diff, cofac;

        UseDCHuffman();

        diff = coef - LastDC;
        LastDC = coef;
        cofac = abs(diff);
        if (cofac < 256) {
                s = csize[cofac];
        }
        else {
                cofac = cofac >> 8;
                s = csize[cofac] + 8;
        }

        EncodeHuffman(s, Data_Ch);
        if (diff < 0) {
                diff--;
        }
```

```
        WriteBits(s, diff, Data_Ch);
}

void EncodeAC(int *matrix, iBlckSendByte Data_Ch)
{
        int i, k, r, ssss, cofac;

        UseACHuffman();

        for (k=r=0; ++k<BLOCKSIZE; ) {
                cofac = abs(matrix[k]);
                if (cofac < 256)        ssss = csize[cofac];
                else {
                        cofac = cofac >> 8;
                        ssss = csize[cofac] + 8;
                }

                if (matrix[k] == 0) {
                        if (k == BLOCKSIZE-1) {
                                EncodeHuffman(0, Data_Ch);
                                break;
                        }
                        r++;
                }
                else {
                        while (r > 15) {
                                EncodeHuffman(240, Data_Ch);
                                r -= 16;
                        }
                        i = 16 * r + ssss;
                        r = 0;
                        EncodeHuffman(i, Data_Ch);
                        if (matrix[k] < 0) WriteBits(ssss, matrix[k]-1, Data_Ch);
                        else WriteBits(ssss, matrix[k], Data_Ch);
                }
        }
}

/**************************************************************
* Coded by Hongxing Li
**************************************************************/

behavior HuffmanEncode(iBlckRecvBlock Q_H_Ch, iBlckRecvInt QDone,
        iBlckSendInt HDone, iBlckSendByte Data_Ch)
{
        int input[BLOCKSIZE];
        int output[BLOCKSIZE];
        int done;

    void main(void) {
        do {
                Q_H_Ch.receive(input);
                done = QDone.receive();

                ZigzagMatrix(input, output);
                EncodeDC(output[0], Data_Ch);
                EncodeAC(output, Data_Ch);

                // Coding finishs, use HDone to synchronize with HandleData
                if (done == 1)
                        HDone.send(done);
        } while(done != 1);
    }
};
```

```
behavior Quantization(iBlckRecvBlock D_Q_Ch, iBlckRecvInt DDone,
        iBlckSendBlock Q_H_Ch, iBlckSendInt QDone)
{
        int data[BLOCKSIZE];
        int done;

    void main(void) {
        do {
                D_Q_Ch.receive(data);
                done = DDone.receive();

                Quantize(data, QuantizationMatrix);

                Q_H_Ch.send(data);
                QDone.send(done);
        } while(done != 1);
    }
};


behavior DCT(iBlckRecvBlock H_D_Ch, iBlckRecvInt HDDone,
        iBlckSendBlock D_Q_Ch, iBlckSendInt DDone)
{
        int DCTBound=1023;
        int DCTShift=128;

        int input[BLOCKSIZE];
        int output[BLOCKSIZE];
        int done;

    void main(void) {
        do {
                H_D_Ch.receive(input);
                done = HDDone.receive();

                PreshiftDctMatrix(input, DCTShift);
                ChenDct(input, output);
                BoundDctMatrix(output, DCTBound);

                D_Q_Ch.send(output);
                DDone.send(done);
        } while(done != 1);
    }
};

behavior HandleData(iBlckRecvInt Header_Ch, iBlckRecvByte Pixel_Ch,
        iBlckRecvInt HDone, iBlckSendBlock H_D_Ch, iBlckSendByte Data_Ch,
        iBlckSendInt HDDone)
{
        int data[BLOCKSIZE];
        unsigned char *stripe;
        int m, i, j, k, done;
        int MDUWide, MDUHigh, ImageHeight, ImageWidth;

    void main(void) {

        LastDC = 0;

        ImageWidth = Header_Ch.receive();
        ImageHeight = Header_Ch.receive();

        MDUWide = (ImageWidth+7) >> 3;
        MDUHigh = (ImageHeight+7) >> 3;

        JpegDefaultHuffman();
```

```
        WriteMarker(M_SOI, Data_Ch);
        WriteAPP0(Data_Ch);
        WriteSOF(ImageHeight, ImageWidth,  Data_Ch);
        WriteDQT(Data_Ch);
        WriteDHT(Data_Ch);
        WriteSOS(Data_Ch);

        stripe = (unsigned char *)calloc(64*MDUWide, sizeof(char));

        for (m=0; m<MDUHigh; m++) {
                // Read a stripe from testbench
                for (i=0; i<8; i++) {
                        for (j=0; j<MDUWide*8; j++) {
                                if ((j<ImageWidth) && (m*8+i<ImageHeight)) {
                                        stripe[i*MDUWide*8+j]
                                                = Pixel_Ch.receive();
                                }
                                else {
                                        stripe[i*MDUWide*8+j]
                                                = stripe[ImageWidth-1];
                                }
                        }
                }

                // fetch a block and send it to DCT
                for (i=0; i<MDUWide; i++) {
                        for(j=0; j<8; j++) {
                                for(k=0; k<8; k++) {
                                        data[j*8+k] = stripe[j*MDUWide*8+i*8+k];
                                }
                        }
                        H_D_Ch.send(data);
                        if ((m == MDUHigh-1) && (i == MDUWide-1))
                                HDDone.send(1);
                        else
                                HDDone.send(0);
                }
        }

        done = HDone.receive();
        WriteBits(-1, 0, Data_Ch);
        WriteMarker(M_EOI, Data_Ch);
    }
};


behavior Jpeg(iBlckRecvInt header_ch, iBlckRecvByte pixel_ch,
        iBlckSendByte data_ch)
{
        cSyncInt        hddone, ddone, qdone, hdone;
        cSyncBlock      h_d_ch, d_q_ch, q_h_ch;

        HandleData      handledata(header_ch, pixel_ch, hdone, h_d_ch,
                                data_ch, hddone);
        DCT             dct(h_d_ch, hddone, d_q_ch, ddone);
        Quantization    quantization(d_q_ch, ddone, q_h_ch, qdone);
        HuffmanEncode   huffmanencode(q_h_ch, qdone, hdone, data_ch);

    void main(void) {
        par {
                handledata.main();
                dct.main();
                quantization.main();
                huffmanencode.main();
        }
    }
```

```
};
```

B.2 Testbench

B.2.1 io.sc

```
//------------------------------------------------------------------------
// Input/Output behaviors for testbench
//------------------------------------------------------------------------
//
// 05/17/99  A. Gerstlauer
//   * Fixed a bug in BMP file reading: scan lines are aligned to
//     LONG boundaries.
//
// 05/10/99  A. Gerstlauer
//

import "global";
import "chann";

//------------------------------------------------------------------------
// Input stimuli generator

// Type definitions for BMP file format

typedef short WORD;
typedef long DWORD;

typedef struct tagBITMAPFILEHEADER {
        WORD bfType;
        DWORD bfSize;
        WORD bfReserved1;
        WORD bfReserved2;
        DWORD bfOffBits;
} BITMAPFILEHEADER;

typedef struct tagBITMAPINFOHEADER {
        /* BITMAP core header info -> OS/2 */
        DWORD biSize;
        DWORD biWidth;
        DWORD biHeight;
        WORD biPlanes;
        WORD biBitCount;

        /* BITMAP info -> Windows 3.1 */
        DWORD biCompression;
        DWORD biSizeImage;
        DWORD biXPelsPerMeter;
        DWORD biYPelsPerMeter;
        DWORD biClrUsed;
        DWORD biClrImportant;
} BITMAPINFOHEADER;

typedef struct tagRGBTRIPLE {
        BYTE B, G, R;
} RGBTRIPLE;


// Input behavior
// Read BMP file and send data to JPEG behavior.
// NOTE: Only able to handle certain BMP files: grayscale, non-compressed...
behavior Input (in char* ifname,
                iBlckSendInt Header,
                iBlckSendByte Pixel)
{
  FILE* ifp;
```

```
BITMAPFILEHEADER BmpFileHeader;
BITMAPINFOHEADER BmpInfoHeader;
RGBTRIPLE *BmpColors;
int BmpScanWidth, BmpScanHeight;

int ReadRevWord()
{
  int c;
  c = fgetc(ifp);
  c |= fgetc(ifp) << 8;

  return c;
}

int ReadWord()
{
  int c;
  c = fgetc(ifp) << 8;
  c |= fgetc(ifp);

  return c;
}

int ReadByte()
{
  return fgetc(ifp);
}

long ReadRevDWord()
{
  long c;
  c = fgetc(ifp);
  c |= fgetc(ifp) << 8;
  c |= fgetc(ifp) << 16;
  c |= fgetc(ifp) << 24;

  return c;
}

long ReadDWord()
{
  long c;
  c = fgetc(ifp) << 24;
  c |= fgetc(ifp) << 16;
  c |= fgetc(ifp) << 8;
  c |= fgetc(ifp);

  return c;
}

int  IsBmpFile()
{
  int t = ('B'<<8) | 'M';
  int c;
  c = ReadWord();
  fseek(ifp, -2, 1);

  return t == c;
}

void ReadBmpHeader()
{
  int i, count;

  if (!IsBmpFile()) {
    error("This file is not compatible with BMP format.\n", 0);
```

```
  }

  /* BMP file header */
  BmpFileHeader.bfType = ReadWord();
  BmpFileHeader.bfSize = ReadRevDWord();
  BmpFileHeader.bfReserved1 = ReadRevWord();
  BmpFileHeader.bfReserved2 = ReadRevWord();
  BmpFileHeader.bfOffBits = ReadRevDWord();

  /* BMP core info */
  BmpInfoHeader.biSize = ReadRevDWord();
  BmpInfoHeader.biWidth = ReadRevDWord();
  BmpInfoHeader.biHeight = ReadRevDWord();
  BmpInfoHeader.biPlanes = ReadRevWord();
  BmpInfoHeader.biBitCount = ReadRevWord();

  if (BmpInfoHeader.biSize > 12) {
    BmpInfoHeader.biCompression = ReadRevDWord();
    BmpInfoHeader.biSizeImage = ReadRevDWord();
    BmpInfoHeader.biXPelsPerMeter = ReadRevDWord();
    BmpInfoHeader.biYPelsPerMeter = ReadRevDWord();
    BmpInfoHeader.biClrUsed = ReadRevDWord();
    BmpInfoHeader.biClrImportant = ReadRevDWord();

    /* read RGBQUAD */
    count = BmpFileHeader.bfOffBits - ftell(ifp);
    count >>= 2;

    BmpColors = (RGBTRIPLE*) calloc(sizeof(RGBTRIPLE), count);

    for (i=0; i<count; i++) {
      BmpColors[i].B = ReadByte();
      BmpColors[i].G = ReadByte();
      BmpColors[i].R = ReadByte();
      ReadByte();
    }
  }
  else {
    /* read RGBTRIPLE */
    count = BmpFileHeader.bfOffBits - ftell(ifp);
    count /= 3;

    BmpColors = (RGBTRIPLE*) calloc(sizeof(RGBTRIPLE), count);

    for (i=0; i<count; i++) {
      BmpColors[i].B = ReadByte();
      BmpColors[i].G = ReadByte();
      BmpColors[i].R = ReadByte();
    }
  }

  /* BMP scan line is aligned by LONG boundary */
  if (BmpInfoHeader.biBitCount == 24) {
    BmpScanWidth = ((BmpInfoHeader.biWidth*3 + 3) >> 2) << 2;
    BmpScanHeight = BmpInfoHeader.biHeight;
  }
  else {
    BmpScanWidth = ((BmpInfoHeader.biWidth + 3) >> 2) << 2;
    BmpScanHeight = BmpInfoHeader.biHeight;
  }
}


void main(void)
{
  int c, r;
```

```
    char buf;

    // Open file
    ifp = fopen(ifname, "rb");
    if (!ifp) {
      error("Cannot open input file %s\n", ifname);
    }

    // Read BMP file header
    ReadBmpHeader();

    // Send header (size) information
    Header.send (BmpInfoHeader.biWidth);
    Header.send (BmpInfoHeader.biHeight);

    // Loop over rows
    for (r = BmpInfoHeader.biHeight - 1; r >= 0; r--)
    {
      // Position file pointer to corresponding row
      fseek (ifp, BmpFileHeader.bfOffBits + r * BmpScanWidth, 0);

      // Loop over columns
      for (c = 0; c < BmpInfoHeader.biWidth; c++) {
        // Read pixel
        if (ferror(ifp) || (fread(&buf, 1, 1, ifp) != 1)) {
          error("Error reading data from file %s\n", ifname);
        }

        // Send off
        Pixel.send(buf);
      }
    }

    fclose (ifp);
  }
};


//---------------------------------------------------------------------------
// Output monitor.

// End of image marker
#define MARKER     0xFF
#define MARK_EOI   0xD9


// Output behavior
// Listen to JPEG output and write it into a file.
behavior Output (in char* ofname,
                 iBlckRecvByte Data)
{
  FILE* ofp;

  void main(void)
  {
    bool running, marker;
    unsigned char buf;

    // Open file
    if (ofname) {
      ofp = fopen(ofname, "wb");
      if (!ofp) {
        error("Cannot open output file %s\n", ofname);
      }
    }
    else {
```

```
      ofp = openStdout();
    }

    // Read JPEG data, write to file
    running = true;
    marker = false;
    while (running)
    {
      buf = Data.receive();

      if ((fwrite(&buf, 1, 1, ofp) != 1) || ferror(ofp)) {
        error("Error writing to file %s\n", ofname);
      }

      // Is it a marker?
      if (marker) {
        // End of image?
        running = (buf != MARK_EOI);
        marker = false;
      }
      else {
        marker = (buf == MARKER);
      }

    }

    fclose(ofp);
  }
};
```

B.2.2 tb.sc

```
//------------------------------------------------------------------------
// JPEG Testbench
//------------------------------------------------------------------------
//
// 05/10/99  A. Gerstlauer
//

import "io";
import "jpeg";


behavior Main
{
  char* ifname;
  char* ofname;

  // Channels
  cSyncInt   header;
  cSyncByte  pixel;
  cSyncByte  data;

  Input    input(ifname, header, pixel);
  Jpeg     jpeg(header, pixel, data);
  Output   output(ofname, data);


  int main (int argc, char** argv)
  {
    // Command line arguments
    if (argc < 2) {
      error("Usage: %s infile [ outfile ]\n", argv[0]);
```

```
    }
    ifname = argv[1];
    if (argc >= 3) {
      ofname = argv[2];
    } else {
      ofname = 0;
    }

    // And now run the stuff ...
    par {
      input.main();
      jpeg.main();
      output.main();
    }

    return 0;
  }

};
```

C RTL Code

C.1 JPEG Encoder

C.1.1 typedef.vhd

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

package typedef is
        type INT_ARRAY is array(natural range<>) of INTEGER;
end typedef;
```

C.1.2 basic_function.vhd

```
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL;
USE IEEE.Std_Logic_arith.All;
USE IEEE.Std_Logic_signed.ALL;
USE work.typedef.ALL;

PACKAGE Basic_Function IS
function JLS(L: INTEGER; R: INTEGER) return INTEGER;
function JRS(L: INTEGER; R: INTEGER) return INTEGER;
function JAND(L: INTEGER; R: INTEGER) return INTEGER;
function JOR(L: INTEGER; R: INTEGER) return INTEGER;
END Basic_Function;

PACKAGE BODY Basic_Function IS

function JLS(L: INTEGER; R: INTEGER) return INTEGER is
        variable Result :INTEGER;
begin
        Result := L;
        for count in 0 to R-1 loop
                Result := Result*2;
        end loop;
        return Result;
end;

function JRS(L: INTEGER; R: INTEGER) return INTEGER is
        variable Result :INTEGER;
begin
        Result := L;
        for count in 0 to R-1 loop
                Result := Result/2;
        end loop;
        return Result;
end;

function JAND(L: INTEGER; R: INTEGER) return INTEGER is
        variable Result :INTEGER;
begin
        return CONV_INTEGER(CONV_STD_LOGIC_VECTOR(L, 32) and
                CONV_STD_LOGIC_VECTOR(R, 32));
end;

function JOR(L: INTEGER; R: INTEGER) return INTEGER is
        variable Result :INTEGER;
begin
        return CONV_INTEGER(CONV_STD_LOGIC_VECTOR(L, 32) or
                CONV_STD_LOGIC_VECTOR(R, 32));
end;

END Basic_Function;
```

C.1.3 JpegEncoder.vhd

```
---------------------------------------------------------------------
--      File            :       JpegEncoder.vhd
--      Designer        :       Anand Selka, Junyu Peng, Chuck Siska,
--                              Lingling Sun, Lukai Cai, Hongxing Li
--      Date            :       June 9, 1999
---------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;
use work.typedef.all;
use work.basic_function.all;
use std.textio.all;
use work.read_pgm.all;

entity JpegEncoder is
        port (
                Reset : in STD_LOGIC;
                Clk : in STD_LOGIC;
                InReady : out STD_LOGIC;
                InDataValid : in STD_LOGIC;
                OutDataReady : out STD_LOGIC;
                InData : in unsigned(7 downto 0);
                OutData : out unsigned(7 downto 0)
                );
end JpegEncoder;

architecture sequential of JpegEncoder is

constant C_1 : INTEGER := -1;
constant C0 : INTEGER := 0;
constant C1 : INTEGER := 1;
constant C2 : INTEGER := 2;
constant C3 : INTEGER := 3;
constant C4 : INTEGER := 4;
constant C5 : INTEGER := 5;
constant C6 : INTEGER := 6;
constant C7 : INTEGER := 7;
constant C8 : INTEGER := 8;
constant C9 : INTEGER := 9;
constant C15 : INTEGER := 15;
constant C16 : INTEGER := 16;
constant C24 : INTEGER := 24;
constant C32 : INTEGER := 32;
constant C40 : INTEGER := 40;
constant C48 : INTEGER := 48;
constant C56 : INTEGER := 56;
constant C63    : INTEGER := 63;
constant C64    : INTEGER := 64;
constant C100 : INTEGER := 100;
constant C196 : INTEGER := 196;
constant C210 : INTEGER := 210;
constant C240   : INTEGER := 240;
constant C255 : INTEGER := 255;
constant C256 : INTEGER := 256;
constant C284 : INTEGER := 284;
constant C362 : INTEGER := 362;
constant C426 : INTEGER := 426;
constant C473 : INTEGER := 473;
constant C502 : INTEGER := 502;
constant M_APP0 : INTEGER := 224;
constant M_COM : INTEGER := 254;
constant M_DAC : INTEGER := 204;
constant M_DHP: INTEGER := 222;
```

```
constant M_DHT: INTEGER := 196;
constant M_DNL: INTEGER := 220;
constant M_DQT: INTEGER := 219;
constant M_DRI: INTEGER := 221;
constant M_EOI: INTEGER := 217;
constant M_EXP: INTEGER := 223;
constant M_JPG: INTEGER := 200;
constant M_RST0: INTEGER := 208;
constant M_SOI: INTEGER := 216;
constant M_SOS: INTEGER := 218;
constant M_SOF0: INTEGER := 192;
constant C0XFF : INTEGER := 255;

signal rf : INT_ARRAY(0 to 31);
signal i, j, k, r_alu, r_div, r_sh, r_mul, r_ram1, r_ram2 : INTEGER;

type STATE_VALUE is (
        -- HandleData
        HD_S0, HD_S1, HD_S2, HD_S2_1, HD_S2_2, HD_S2_3, HD_S2_4,
        HD_S2_5, HD_S2_6, HD_S3, HD_S3_1, HD_S3_2, HD_S3_3, HD_S3_4,
        HD_S3_5, HD_S3_6, HD_S4, HD_S5, HD_S6, HD_S7, HD_S8, HD_S9,
        HD_S10, HD_S11, HD_S12, HD_S12_0, HD_S12_0_1,
        HD_S12_0_2, HD_S12_0_3, HD_S13, HD_S14, HD_S15,
        HD_S16, HD_S17, HD_S18, HD_S19, HD_S20, HD_S21, HD_S21_1,
        HD_S22, HD_S22_1, HD_S23,HD_S24, HD_S25, HD_S26, HD_S27,
        HD_S28, HD_S29, HD_S30, HD_S31, HD_S31_1, HD_S31_2, HD_S31_3,
        HD_S32, HD_S33, HD_S34, HD_S35, HD_S36, HD_S36_1, HD_S37,
        HD_S38, HD_S39, HD_S40, HD_S40_1, HD_S40_2, HD_S40_3,
        HD_S40_4, HD_S40_4_1, HD_S40_4_2, HD_S40_4_3, HD_S40_5, HD_S41,
        HD_S42, HD_S43, HD_S44, HD_S44_1, HD_S45, HD_S46, HD_S47,
        HD_S48, HD_S49, HD_S50, HD_S51, HD_S52, HD_S53, HD_S54,
        HD_S54_0, HD_S54_0_1, HD_S54_1, HD_S54_2, HD_S54_3,
        HD_S54_4, HD_S54_4_1, HD_S54_4_2, HD_S54_4_3, HD_S54_4_4,
        HD_S55, HD_S56, HD_S57, HD_S58, HD_S59, HD_S60,
        HD_S61, PDM_S0,

        -- PreShift
        PD_S1, PD_S2, PD_S3, PD_S4, PD_S5,

        -- Chen DCT
        CDCT_S1, CDCT_S2, CDCT_S3, CDCT_S4, CDCT_S5, CDCT_S6,
        CDCT_S7, CDCT_S8,CDCT_S9, CDCT_S10, CDCT_S11, CDCT_S12, CDCT_S13,
        CDCT_S14, CDCT_S15, CDCT_S16, CDCT_S17, CDCT_S18, CDCT_S19,
        CDCT_S20, CDCT_S21, CDCT_S22, CDCT_S23, CDCT_S24, CDCT_S25,
        CDCT_S26, CDCT_S27, CDCT_S28,CDCT_S29, CDCT_S30, CDCT_S31,
        CDCT_S32, CDCT_S33, CDCT_S34, CDCT_S35, CDCT_S36, CDCT_S37,
        CDCT_S38, CDCT_S39, CDCT_S40, CDCT_S41, CDCT_S42, CDCT_S43,
        CDCT_S44, CDCT_S45, CDCT_S46, CDCT_S47, CDCT_S48, CDCT_S49,
        CDCT_S50, CDCT_S51, CDCT_S52, CDCT_S53, CDCT_S54, CDCT_S55,
        CDCT_S56, CDCT_S57, CDCT_S58,CDCT_S59, CDCT_S60, CDCT_S61,
        CDCT_S62, CDCT_S63, CDCT_S64, CDCT_S65, CDCT_S66, CDCT_S67,
        CDCT_S68, CDCT_S69, CDCT_S70, CDCT_S71, CDCT_S72, CDCT_S73,
        CDCT_S74, CDCT_S75, CDCT_S76, CDCT_S77, CDCT_S78, CDCT_S79,
        CDCT_S80, CDCT_S81, CDCT_S82, CDCT_S83, CDCT_S84, CDCT_S85,
        CDCT_S86, CDCT_S87, CDCT_S88,CDCT_S89, CDCT_S90, CDCT_S91,
        CDCT_S92, CDCT_S93, CDCT_S94, CDCT_S95, CDCT_S96, CDCT_S97,
        CDCT_S98, CDCT_S99, CDCT_S100, CDCT_S101, CDCT_S102, CDCT_S103,
        CDCT_S104, CDCT_S105, CDCT_S106, CDCT_S107, CDCT_S108, CDCT_S109,
        CDCT_S110, CDCT_S111, CDCT_S112, CDCT_S113, CDCT_S114, CDCT_S115,
        CDCT_S116, CDCT_S117, CDCT_S118,CDCT_S119, CDCT_S120, CDCT_S121,
        CDCT_S122, CDCT_S123, CDCT_S124, CDCT_S125, CDCT_S126, CDCT_S127,
        CDCT_S128, CDCT_S129, CDCT_S130, CDCT_S131, CDCT_S132, CDCT_S133,
        CDCT_S134, CDCT_S135, CDCT_S136, CDCT_S137, CDCT_S138, CDCT_S139,
        CDCT_S140, CDCT_S141, CDCT_S142, CDCT_S143, CDCT_S144, CDCT_S145,
        CDCT_S146, CDCT_S147,
```

```
-- Bound
BD_S0,BD_S1, BD_S2, BD_S3, BD_S4, BD_S5, BD_S6, BD_S7,

-- Quantization
QT_S1, QT_S2, QT_S3, QT_S4, QT_S5, QT_S6, QT_S7, QT_S8,

-- ZigzagMatrix states:
ZM_S0, ZM_S1, ZM_S2, ZM_S3, ZM_S4,

-- EncodeDC states:
ED_S0, ED_S1, ED_S2, ED_S3, ED_S4, ED_S5, ED_S6,
ED_S7, ED_S8, ED_S9, ED_S10, ED_S11, ED_S12, ED_S13,

-- EncodeDC states:
EA_S0, EA_S1, EA_S2, EA_S3, EA_S4, EA_S5, EA_S6,
EA_S7, EA_S8, EA_S9, EA_S10, EA_S11, EA_S12,
EA_S13, EA_S13a, EA_S14, EA_S15, EA_S16, EA_S17, EA_S18,
EA_S18a, EA_S19, EA_S20, EA_S21, EA_S22, EA_S23, EA_S24,
EA_S25,

----------------------------------------
--              EncodeHuffman
----------------------------------------
EH_S_0,          -- void EncodeHuffman( int value, bool ac_dc )
EH_S_REG_SAVE,      -- store rf_21
EH_S_REG_SAVE2,
EH_S_REG_SAVE3,     -- store rf_20
EH_S_REG_SAVE4,     -- prep to load with index [value]
EH_S_IF_AC_DC,      -- using AC or DC Huff table?
EH_S_DC_IF_EHUFF,   -- if (Ehuff->ehufsi[value]) {
EH_S_DC_IF_EHUFF2,
EH_S_AC_IF_EHUFF,   -- if (Ehuff->ehufsi[value]) {
EH_S_AC_IF_EHUFF2,
EH_S_CALL,          -- call WriteBits
EH_S_RETURN,        -- restore rf_1
EH_S_REG_RESTORE,   -- restore rf_1
EH_S_REG_RESTORE2,
EH_S_REG_RESTORE3,  -- restore rf_2
EH_S_END,           -- Return to Caller

----------------------------------------
--              WriteBits
----------------------------------------
WB_S_0,          -- int WriteBits( n, code )
WB_S_REG_SAVE,      -- store rf_n
WB_S_REG_SAVE2,     -- prep
WB_S_REG_SAVE3,     -- store rf_code
WB_S_REG_SAVE4,     -- prep
WB_S_REG_SAVE5,     -- store rf_3
WB_S_REG_SAVE6,     -- prep
WB_S_REG_SAVE7,     -- store rf_4
WB_S_REG_SAVE8,     -- prep
WB_S_REG_SAVE9,     -- store rf_5
WB_S_INIT,          -- prep to load write_byte
WB_S_INIT2,         -- load write_byte
WB_S_INIT3,         -- prep to load left_bits
WB_S_INIT4,         -- load left_bits
WB_S_IF_N_LT0,      -- if (n < 0) {
WB_S_IF_LB_LT,        -- if (left_bits < 8) {
WB_S_LB_LT8,            -- n = left_bits;
WB_S_LT8_SEND,          -- Data_Ch.send(write_byte);
WB_S_LT8_IF_WB,         -- if (write_byte == 0xff) {
WB_S_LT8_WB_FF,           -- Data_Ch.send(0); }
WB_S_LT8_WB,            -- write_byte = 0;
WB_S_LT8_WB_LB,         -- left_bits = 8; }
WB_S_ELSE_LB_LT,      -- else { n = 0; }
```

```
                              -- return n; }
        WB_S_N_GE0,           -- #1 code &= lmask[n];
        WB_S_NGE0_CO2,        -- #2 code &= lmask[n];
                              --:: p = n - left_bits;
        WB_S_NGE0_CO3,        -- #3 code &= lmask[n];
        WB_S_NGE0_CO4,        -- #4 code &= lmask[n];
        WB_S_IF_N_EQ,         -- if (n == left_bits) {
        WB_S_N_EQ_LB,           -- write_byte |= code;
        WB_S_EQLB_SEND,         -- Data_Ch.send(write_byte);
        WB_S_EQLB_IF_WB,        -- if (write_byte == 0xff) {
        WB_S_EQLB_WB_FF,          -- Data_Ch.send(0); }
        WB_S_EQLB_WB,           -- write_byte = 0;
        WB_S_EQLB_WB_LB,        -- left_bits = 8; }
        WB_S_ELIF_N_GT,       -- else if (n > left_bits) {
        WB_S_N_GT_LB,           -- #1 write_byte |= (code >> p);
        WB_S_GTLB_WB2,          -- #2 write_byte |= (code >> p);
        WB_S_GTLB_SEND,         -- Data_Ch.send(write_byte);
        WB_S_GTLB_IF_WB,        -- if (write_byte == 0xff) {
        WB_S_GTLB_WB_FF,          -- Data_Ch.send(0); }
        WB_S_IF_P_GT8,          -- if (p > 8) {
        WB_S_P_GT8,               -- #1 write_byte = (0xff & (code >> (p - 8)));
        WB_S_GT8_WB2,             -- #2 write_byte = (0xff & (code >> (p - 8)));
        WB_S_GT8_WB3,             -- #3 write_byte = (0xff & (code >> (p - 8)));
        WB_S_GT8_SEND,            -- Data_Ch.send(write_byte);
        WB_S_GT8_IF_WB,           -- if (write_byte == 0xff) {
        WB_S_GT8_WB_FF,             -- Data_Ch.send(0); }
        WB_S_GT8_P,               -- p -= 8; }
        WB_S_P_LE8,             -- #1 write_byte = (code & lmask[p]) << (8 - p);
        WB_S_PLE8_WB2,          -- #2 write_byte = (code & lmask[p]) << (8 - p);
                                --:: left_bits = 8 - p; }
        WB_S_PLE8_WB3,          -- #3 write_byte = (code & lmask[p]) << (8 - p);
        WB_S_PLE8_WB4,          -- #4 write_byte = (code & lmask[p]) << (8 - p);
        WB_S_PLE8_WB5,          -- #5 write_byte = (code & lmask[p]) << (8 - p);
        WB_S_ELSE_N_LT,       -- else { #1 write_byte |= (code << -p);
        WB_S_LTLB_WB2,          -- #2 write_byte |= (code << -p);
                                --:: left_bits -= n; }
        WB_S_LTLB_WB3,          -- #3 write_byte |= (code << -p);
        WB_S_LTLB_WB4,          -- #4 write_byte |= (code << -p);
        WB_S_LTLB_LB,           -- left_bits -= n; }
        WB_S_LTLB_LB2,          -- #2 left_bits -= n; }
        WB_S_RETURN,          -- store write_byte
        WB_S_RETURN2,
        WB_S_RETURN3,         -- store left_bits
        WB_S_RETURN4,
        WB_S_REG_RESTOR,      -- restore rf_n;
        WB_S_REG_RESTOR2,     -- #2 restore rf_n;
        WB_S_REG_RESTOR3,     -- restore rf_code;
        WB_S_REG_RESTOR4,     -- #2 restore rf_code;
        WB_S_REG_RESTOR5,     -- restore rf_3;
        WB_S_REG_RESTOR6,     -- #2 restore rf_3;
        WB_S_REG_RESTOR7,     -- restore rf_4;
        WB_S_REG_RESTOR8,     -- #2 restore rf_4;
        WB_S_REG_RESTOR9,     -- restore rf_5;
        WB_S_END,               -- Return to Caller

        ------------------------------------------------------------------
        -- WriteMarker, WriteAPP0, WriteSOF, WRiteDQT, WriteDHT, WriteSOS
        ------------------------------------------------------------------
        WH_S0, WH_S1, WH_S2, WH_S3, WH_S4,
        WH_S5, WH_S6, WH_S7, WH_S8, WH_S9,
        WH_S10, WH_S11, WH_S12, WH_S13, WH_S14,
        WH_S15, WH_S16, WH_S17, WH_S18, WH_S19,
        WH_S20, WH_S21, WH_S21_1, WH_S21_2, WH_S22, WH_S23, WH_S24,
        WH_S25, WH_S26, WH_S27, WH_S28, WH_S29,
        WH_S30, WH_S31, WH_S32, WH_S33, WH_S34,
        WH_S35, WH_S36, WH_S37, WH_S38, WH_S39,
```

```
        WH_S40, WH_S41, WH_S42, WH_S43, WH_S44,
        WH_S45, WH_S46, WH_S47, WH_S48, WH_S49,
        WH_S50, WH_S51, WH_S52, WH_S53, WH_S54,
        WH_S55, WH_S56, WH_S57, WH_S58, WH_S59,
        WH_S60, WH_S61, WH_S62, WH_S63, WH_S64,
        WH_S65, WH_S66, WH_S67, WH_S68, WH_S69,
        WH_S70, WH_S71, WH_S72, WH_S73, WH_S74,
        WH_S75, WH_S76, WH_S77, WH_S78, WH_S79,
        WH_S80, WH_S81, WH_S82, WH_S83, WH_S84,
        WH_S85, WH_S86, WH_S87, WH_S88, WH_S89,
        WH_S90, WH_S91, WH_S92, WH_S93, WH_S94,
        WH_S95, WH_S96, WH_S97, WH_S98, WH_S99,
        WH_S100, WH_S101, WH_S102, WH_S103, WH_S104,
        WH_S105, WH_S106, WH_S107, WH_S108, WH_S109,
        WH_S110, WH_S111, WH_S112, WH_S113, WH_S114,
        WH_S115, WH_S116, WH_S117, WH_S118, WH_S119,
        WH_S120, WH_S121, WH_S122, WH_S123, WH_S124,
        WH_S125, WH_S126, WH_S127, WH_S128, WH_S129,
        WH_S130, WH_S131, WH_S132, WH_S133, WH_S134,
        WH_S135, WH_S136, WH_S137, WH_S138, WH_S139,
        WH_S140, WH_S141, WH_S142, WH_S143, WH_S144,
        WH_S145, WH_S146, WH_S147, WH_S148, WH_S149,
        WH_S150, WH_S151, WH_S152, WH_S153, WH_S154,
        WH_S155, WH_S156, WH_S157, WH_S158, WH_S159,
        WH_S160, WH_S161, WH_S162, WH_S163, WH_S164,
        WH_S165, WH_S166, WH_S167, WH_S168, WH_S169,
        WH_S170, WH_S171, WH_S172, WH_S173, WH_S174,
        WH_S175, WH_S176, WH_S177, WH_S178

        );

signal State : STATE_VALUE := HD_S0;
signal JDH_Return_State, WM_Return_State, WA_Return_State,
        WSF_Return_State, WDQ_Return_State, WDH_Return_State,
        WB_Return_State, WSS_Return_State, PDM_Return_State,
        ED_Return_State, EA_Return_State, EH_Return_State : STATE_VALUE;

begin
        process(Reset, Clk)
        variable stripe : INT_ARRAY(0 to 8192) := (others => 0); --7424
        variable input, output: INT_ARRAY(0 to 63);
        variable mMDUWide, mMDUHigh, mImageHeight, mImageWidth,
                Pq, LastDC, temp : INTEGER := 0;

        variable QuantizationMatrix: INT_ARRAY(0 to 63) := (
                16, 11, 10, 16, 24, 40, 51, 61,
                12, 12, 14, 19, 26, 58, 60, 55,
                14, 13, 16, 24, 40, 57, 69, 56,
                14, 17, 22, 29, 51, 87, 80, 62,
                18, 22, 37, 56, 68, 109, 103, 77,
                24, 35, 55, 64, 81, 104, 113, 92,
                49, 64, 78, 87, 103, 121, 120, 101,
                72, 92, 95, 98, 112, 100, 103, 99 );

        variable LuminanceDCBits: INT_ARRAY(0 to 15) := (
                0, 1, 5, 1, 1, 1, 1, 1,
                1, 0, 0, 0, 0, 0, 0, 0);

        variable LuminanceDCValues: INT_ARRAY(0 to 11) := (
                0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11);

        variable LuminanceACBits: INT_ARRAY(0 to 15) := (
                0, 2, 1, 3, 3, 2, 4, 3,
                5, 5, 4, 4, 0, 0, 1, 125);

        variable LuminanceACValues: INT_ARRAY(0 to 161) := (
```

```
        1, 2, 3, 0, 4, 17, 5, 18,
        33, 49, 65, 6, 19, 81, 97, 7,
        34, 113, 20, 50, 129, 145, 161, 8,
        35, 66, 177, 193, 21, 82, 209, 240,
        36, 51, 98, 114, 130, 9, 10, 22,
        23, 24, 25, 26, 37, 38, 39, 40,
        41, 42, 52, 53, 54, 55, 56, 57,
        58, 67, 68, 69, 70, 71, 72, 73,
        74, 83, 84, 85, 86, 87, 88, 89,
        90, 99, 100, 101, 102, 103, 104, 105,
        106, 115, 116, 117, 118, 119, 120, 121,
        122, 131, 132, 133, 134, 135, 136, 137,
        138, 146, 147, 148, 149, 150, 151, 152,
        153, 154, 162, 163, 164, 165, 166, 167,
        168, 169, 170, 178, 179, 180, 181, 182,
        183, 184, 185, 186, 194, 195, 196, 197,
        198, 199, 200, 201, 202, 210, 211, 212,
        213, 214, 215, 216, 217, 218, 225, 226,
        227, 228, 229, 230, 231, 232, 233, 234,
        241, 242, 243, 244, 245, 246, 247, 248,
        249, 250);

variable ZigzagIndex : INT_ARRAY(0 to 63) := (
        0,  1,  5,  6, 14, 15, 27, 28,
        2,  4,  7, 13, 16, 26, 29, 42,
        3,  8, 12, 17, 25, 30, 41, 43,
        9, 11, 18, 24, 31, 40, 44, 53,
        10, 19, 23, 32, 39, 45, 52, 54,
        20, 22, 33, 38, 46, 51, 55, 60,
        21, 34, 37, 47, 50, 56, 59, 61,
        35, 36, 48, 49, 57, 58, 62, 63);

variable IZigzagIndex: INT_ARRAY(0 to 63) := (
        0,  1,  8, 16,  9,  2,  3, 10,
        17, 24, 32, 25, 18, 11,  4,  5,
        12, 19, 26, 33, 40, 48, 41, 34,
        27, 20, 13,  6,  7, 14, 21, 28,
        35, 42, 49, 56, 57, 50, 43, 36,
        29, 22, 15, 23, 30, 37, 44, 51,
        58, 59, 52, 45, 38, 31, 39, 46,
        53, 60, 61, 54, 47, 55, 62, 63);

variable csize : INT_ARRAY(0 to 255) := (
        0,
        1,
        2, 2,
        3, 3, 3, 3,
        4, 4, 4, 4, 4, 4, 4, 4,
        5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
        7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
        7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
        7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
        8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
        8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
        8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
        8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
        8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
        8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
        8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
        8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8);

        --==============================================
        --              EncodeHuffman Locals
```

```
        --===============================================
variable encodehuffman_reg_save: INT_ARRAY(0 to 1) := ( 0, 0 );

        --===============================================
        --              WriteBits Locals
        --===============================================

variable writebits_static: INT_ARRAY(0 to 7) := (
        0, -- local write_byte
        8, -- local left_bits
        0, 0, 0, 0, 0, 0 -- callee-save register area
        );
variable lmask: INT_ARRAY(0 to 16) := (
        0,
        1, 3, 7, 15,
        31, 63, 127, 255,
        511, 1023, 2047, 4095,  -- (1+ (* 2 32767))
        8191, 16383, 32767, 65535
        );


        --===============================================
        --              Huffman Encoding Tables
        --===============================================

variable DC_EHUFF_CO: INT_ARRAY(0 to 256) := (
        0, 2, 3, 4, 5, 6, 14, 30, 62, 126,
        254, 510, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0
        );
variable DC_EHUFF_SI: INT_ARRAY(0 to 256) := (
        2, 3, 3, 3, 3, 3, 4, 5, 6, 7,
        8, 9, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0
            );
    variable AC_EHUFF_CO: INT_ARRAY(0 to 256) := (
            10, 0, 1, 4, 11, 26, 120, 248, 1014, 65410,
            65411, 0, 0, 0, 0, 0, 0, 12, 27, 121,
            502, 2038, 65412, 65413, 65414, 65415, 65416, 0, 0, 0,
            0, 0, 0, 28, 249, 1015, 4084, 65417, 65418, 65419,
            65420, 65421, 65422, 0, 0, 0, 0, 0, 0, 58,
            503, 4085, 65423, 65424, 65425, 65426, 65427, 65428, 65429, 0,
            0, 0, 0, 0, 0, 59, 1016, 65430, 65431, 65432,
            65433, 65434, 65435, 65436, 65437, 0, 0, 0, 0, 0,
            0, 122, 2039, 65438, 65439, 65440, 65441, 65442, 65443, 65444,
            65445, 0, 0, 0, 0, 0, 0, 123, 4086, 65446,
            65447, 65448, 65449, 65450, 65451, 65452, 65453, 0, 0, 0,
            0, 0, 0, 250, 4087, 65454, 65455, 65456, 65457, 65458,
            65459, 65460, 65461, 0, 0, 0, 0, 0, 0, 504,
            32704, 65462, 65463, 65464, 65465, 65466, 65467, 65468, 65469, 0,
            0, 0, 0, 0, 0, 505, 65470, 65471, 65472, 65473,
            65474, 65475, 65476, 65477, 65478, 0, 0, 0, 0, 0,
            0, 506, 65479, 65480, 65481, 65482, 65483, 65484, 65485, 65486,
            65487, 0, 0, 0, 0, 0, 0, 1017, 65488, 65489,
            65490, 65491, 65492, 65493, 65494, 65495, 65496, 0, 0, 0,
            0, 0, 0, 1018, 65497, 65498, 65499, 65500, 65501, 65502,
            65503, 65504, 65505, 0, 0, 0, 0, 0, 0, 2040,
            65506, 65507, 65508, 65509, 65510, 65511, 65512, 65513, 65514, 0,
            0, 0, 0, 0, 0, 65515, 65516, 65517, 65518, 65519,
            65520, 65521, 65522, 65523, 65524, 0, 0, 0, 0, 0,
            2041, 65525, 65526, 65527, 65528, 65529, 65530, 65531, 65532, 65533,
            65534, 0, 0, 0, 0, 0, 0
            );
    variable AC_EHUFF_SI: INT_ARRAY(0 to 256) := (
            4, 2, 2, 3, 4, 5, 7, 8, 10, 16,
            16, 0, 0, 0, 0, 0, 0, 4, 5, 7,
            9, 11, 16, 16, 16, 16, 16, 0, 0, 0,
            0, 0, 0, 5, 8, 10, 12, 16, 16, 16,
            16, 16, 16, 0, 0, 0, 0, 0, 0, 6,
            9, 12, 16, 16, 16, 16, 16, 16, 16, 0,
            0, 0, 0, 0, 0, 6, 10, 16, 16, 16,
            16, 16, 16, 16, 16, 0, 0, 0, 0, 0,
            0, 7, 11, 16, 16, 16, 16, 16, 16, 16,
            16, 0, 0, 0, 0, 0, 0, 7, 12, 16,
            16, 16, 16, 16, 16, 16, 16, 0, 0, 0,
            0, 0, 0, 8, 12, 16, 16, 16, 16, 16,
            16, 16, 16, 0, 0, 0, 0, 0, 0, 9,
            15, 16, 16, 16, 16, 16, 16, 16, 16, 0,
            0, 0, 0, 0, 0, 9, 16, 16, 16, 16,
            16, 16, 16, 16, 16, 0, 0, 0, 0, 0,
            0, 9, 16, 16, 16, 16, 16, 16, 16, 16,
            16, 0, 0, 0, 0, 0, 0, 10, 16, 16,
            16, 16, 16, 16, 16, 16, 16, 0, 0, 0,
            0, 0, 0, 10, 16, 16, 16, 16, 16, 16,
            16, 16, 16, 0, 0, 0, 0, 0, 0, 11,
            16, 16, 16, 16, 16, 16, 16, 16, 16, 0,
```

```
            0, 0, 0, 0, 0, 16, 16, 16, 16, 16,
            16, 16, 16, 16, 16, 0, 0, 0, 0, 0,
            11, 16, 16, 16, 16, 16, 16, 16, 16, 16,
            16, 0, 0, 0, 0, 0, 0
            );

    begin
    if(Reset = '1') then
            State <= HD_S0;
    end if;
    if(Reset /= '1' and Clk = '1') then
            case State is
                    when HD_S0 =>
                            r_ram1 <= C0;
                            State <= HD_S1;

                    when HD_S1 =>
                            LastDC := r_ram1;
                            InReady <= '1';
                            State <= HD_S2;

                    when HD_S2 =>
                            if(InDataValid = '1') then
                                    State <= HD_S2_1;
                            else
                                    State <= HD_S2;
                            end if;

                    when HD_S2_1 =>
                            rf(5) <= CONV_INTEGER(InData);
                            InReady <= '0';
                            State <= HD_S2_2;

                    when HD_S2_2 =>
                            InReady <= '1';
                            if(InDataValid = '1') then
                                    State <= HD_S2_3;
                            else
                                    State <= HD_S2_2;
                            end if;

                    when HD_S2_3 =>
                            rf(7) <= CONV_INTEGER(InData);
                            r_ram1 <= C256;
                            InReady <= '0';
                            State <= HD_S2_4;

                    when HD_S2_4 =>
                            r_mul <= rf(5) * r_ram1;
                            State <= HD_S2_5;

                    when HD_S2_5 =>
                            r_alu <= r_mul + rf(7);
                            State <= HD_S2_6;

                    when HD_S2_6 =>
                            rf(5) <= r_alu;
                            State <= HD_S3;

                    when HD_S3 =>
                            InReady <= '1';
                            if(InDataValid = '1') then
                                    State <= HD_S3_1;
                            else
                                    State <= HD_S3;
                            end if;
```

```
                when HD_S3_1 =>
                        rf(6) <= CONV_INTEGER(InData);
                        InReady <= '0';
                        State <= HD_S3_2;

                when HD_S3_2 =>
                        InReady <= '1';
                        if(InDataValid = '1') then
                                State <= HD_S3_3;
                        else
                                State <= HD_S3_2;
                        end if;

                when HD_S3_3 =>
                        rf(7) <= CONV_INTEGER(InData);
                        r_ram1 <= C256;
                        InReady <= '0';
                        State <= HD_S3_4;


                when HD_S3_4 =>
                        r_mul <= rf(6) * r_ram1;
                        State <= HD_S3_5;

                when HD_S3_5 =>
                        r_alu <= r_mul + rf(7);
                        State <= HD_S3_6;

                when HD_S3_6 =>
                        rf(6) <= r_alu;
                        State <= HD_S4;

                when HD_S4 =>
                        r_ram1 <= C7;
                        State <= HD_S5;

                when HD_S5 =>
                        r_alu <= rf(5) + r_ram1;
                        r_ram2 <= C8;
                        State <= HD_S6;

                when HD_S6 =>
                        r_sh <= r_alu / r_ram2;
                        State <= HD_S7;

                when HD_S7 =>
                        rf(2) <= r_sh;
                        r_alu <= rf(6) + r_ram1;
                        State <= HD_S8;

                when HD_S8 =>
                        r_sh <= r_alu / r_ram2;
                        State <= HD_S9;

                when HD_S9 =>
                        rf(3) <= r_sh;
                        State <= WH_S0;

------------------------------------------------------------------------
--WriteMarker, WriteAPP0, WriteSOF, WriteDQT, WriteDHT, WriteSOS
------------------------------------------------------------------------

                -- WriteMarker(M_SOI)

                when WH_S0 =>
```

```
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(C255, C8);
                State <= WH_S1;

        when WH_S1 =>
                OutDataReady <= '0';
                State <= WH_S2;

        when WH_S2 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(M_SOI, C8);
                State <= WH_S3;

        when WH_S3 =>
                OutDataReady <= '0';
                State <= WH_S4;

        -- WriteAPP0()
                -- WriteMarker(M_APP0)

        when WH_S4 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(C255, C8);
                State <= WH_S5;

        when WH_S5 =>
                OutDataReady <= '0';
                State <= WH_S6;

        when WH_S6 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(M_APP0, C8);
                State <= WH_S7;

        when WH_S7 =>
                OutDataReady <= '0';
                State <= WH_S8;

                -- WriteWord(16)

        when WH_S8 =>
                r_alu <= JRS(16, C8);
                State <= WH_S9;

        when WH_S9 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(r_alu, C8);
                State <= WH_S10;

        when WH_S10 =>
                OutDataReady <= '0';
                State <= WH_S11;

        when WH_S11 =>
                r_alu <= JAND(16, C255);
                State <= WH_S12;

        When WH_S12 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(r_alu, C8);
                State <= WH_S13;

        when WH_S13 =>
                OutDataReady <= '0';
                State <= WH_S14;
```

```vhdl
                -- The ASCII code of 'JFIF' is 74, 70, 73, 70

        when WH_S14 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(74, C8);
                State <= WH_S15;

        when WH_S15 =>
                OutDataReady <= '0';
                State <= WH_S16;

        when WH_S16 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(70, C8);
                State <= WH_S17;

        when WH_S17 =>
                OutDataReady <= '0';
                State <= WH_S18;

        when WH_S18 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(73, C8);
                State <= WH_S19;

        when WH_S19 =>
                OutDataReady <= '0';
                State <= WH_S20;

        when WH_S20 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(70, C8);
                State <= WH_S21;

        when WH_S21 =>
                OutDataReady <= '0';
                State <= WH_S21_1;

        when WH_S21_1 =>
                r_ram1 <= C0;
                State <= WH_S21_2;

        when WH_S21_2 =>
                OutData <= CONV_UNSIGNED(r_ram1, C8);
                OutDataReady <= '1';
                State <= WH_S22;

        when WH_S22 =>
                OutDataReady <= '0';
                r_alu <= JRS(258, C8);
                State <= WH_S23;

        when WH_S23 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(r_alu, C8);
                State <= WH_S24;

        when WH_S24 =>
                OutDataReady <= '0';
                State <= WH_S25;

        when WH_S25 =>
                r_alu <= JAND(258, C255);
                State <= WH_S26;
```

```
when WH_S26 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(r_alu, C8);
        State <= WH_S27;

when WH_S27 =>
        OutDataReady <= '0';
        State <= WH_S28;

        -- WriteByte(2)

when WH_S28 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(2, C8);
        State <= WH_S29;

when WH_S29 =>
        OutDataReady <= '0';
        State <= WH_S30;

        -- WriteWord(0x001d)

when WH_S30 =>
        r_alu <= JRS(29, C8);
        State <= WH_S31;

when WH_S31 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(r_alu, C8);
        State <= WH_S32;

when WH_S32 =>
        OutDataReady <= '0';
        State <= WH_S33;

when WH_S33 =>
        r_alu <= JAND(29, C255);
        State <= WH_S34;

when WH_S34 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(r_alu, C8);
        State <= WH_S35;

when WH_S35 =>
        OutDataReady <= '0';
        State <= WH_S36;

        -- WriteWord(0x001d)

when WH_S36 =>
        r_alu <= JRS(29, C8);
        State <= WH_S37;

when WH_S37 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(r_alu, C8);
        State <= WH_S38;

when WH_S38 =>
        OutDataReady <= '0';
        State <= WH_S39;

when WH_S39 =>
        r_alu <= JAND(29, C255);
        State <= WH_S40;
```

```
when WH_S40 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(r_alu, C8);
        State <= WH_S41;

when WH_S41 =>
        OutDataReady <= '0';
        State <= WH_S42;

        -- WriteWord(0x0000)

when WH_S42 =>
        r_alu <= JRS(0, C8);
        State <= WH_S43;

when WH_S43 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(r_alu, C8);
        State <= WH_S44;

when WH_S44 =>
        OutDataReady <= '0';
        State <= WH_S45;

when WH_S45 =>
        r_alu <= JAND(0, C255);
        State <= WH_S46;

when WH_S46 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(r_alu, C8);
        State <= WH_S47;

when WH_S47 =>
        OutDataReady <= '0';
        State <= WH_S48;

-- WriteSOF()
        -- WriteMarker(M_SOF0)

when WH_S48 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(C255, C8);
        State <= WH_S49;

when WH_S49 =>
        OutDataReady <= '0';
        State <= WH_S50;

when WH_S50 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(M_SOF0, C8);
        State <= WH_S51;

when WH_S51 =>
        OutDataReady <= '0';
        State <= WH_S52;

        -- WriteWord(11)

when WH_S52 =>
        r_alu <= JRS(11, C8);
        State <= WH_S53;

when WH_S53 =>
```

```
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(r_alu, C8);
                State <= WH_S54;

        when WH_S54 =>
                OutDataReady <= '0';
                State <= WH_S55;

        when WH_S55 =>
                r_alu <= JAND(11, C255);
                State <= WH_S56;

        when WH_S56 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(r_alu, C8);
                State <= WH_S57;

        when WH_S57 =>
                OutDataReady <= '0';
                State <= WH_S58;

                -- WriteByte(8)

        when WH_S58 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(8, C8);
                State <= WH_S59;

        when WH_S59 =>
                OutDataReady <= '0';
                State <= WH_S60;

                -- WriteWord(rf(6))

        when WH_S60 =>
                r_alu <= JRS(rf(6), C8);
                State <= WH_S61;

        when WH_S61 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(r_alu, C8);
                State <= WH_S62;

        when WH_S62 =>
                OutDataReady <= '0';
                State <= WH_S63;

        when WH_S63 =>
                State <= WH_S64;

        when WH_S64 =>
                r_alu <= JAND(rf(6), C255);
                State <= WH_S65;

        when WH_S65 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(r_alu, C8);
                State <= WH_S66;

        when WH_S66 =>
                OutDataReady <= '0';
                State <= WH_S67;

                -- WriteWord(rf(5))

        when WH_S67 =>
```

```
                r_alu <= JRS(rf(5), C8);
                State <= WH_S68;

        when WH_S68 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(r_alu, C8);
                State <= WH_S69;

        when WH_S69 =>
                OutDataReady <= '0';
                State <= WH_S70;

        when WH_S70 =>
                r_alu <= JAND(rf(5), C255);
                State <= WH_S71;

        when WH_S71 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(r_alu, C8);
                State <= WH_S72;

        when WH_S72 =>
                OutDataReady <= '0';
                State <= WH_S73;

                -- WriteByte(1)

        when WH_S73 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(1, C8);
                State <= WH_S74;

        when WH_S74 =>
                OutDataReady <= '0';
                State <= WH_S75;

                -- WriteByte(1)

        when WH_S75 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(1, C8);
                State <= WH_S76;

        when WH_S76 =>
                OutDataReady <= '0';
                State <= WH_S77;

                -- WriteByte(0x11)

        when WH_S77 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(17, C8);
                State <= WH_S78;

        when WH_S78 =>
                OutDataReady <= '0';
                State <= WH_S79;

                -- WriteByte(0)

        when WH_S79 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(0, C8);
                State <= WH_S80;

        when WH_S80 =>
```

```
                OutDataReady <= '0';
                State <= WH_S81;

        -- WriteDQT()

        when WH_S81 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(C255, C8);
                State <= WH_S82;

        when WH_S82 =>
                OutDataReady <= '0';
                State <= WH_S83;

        when WH_S83 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(M_DQT, C8);
                State <= WH_S84;

        when WH_S84 =>
                OutDataReady <= '0';
                State <= WH_S85;

                -- WriteWord(67)

        when WH_S85 =>
                r_alu <= JRS(67, C8);
                State <= WH_S86;

        when WH_S86 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(r_alu, C8);
                State <= WH_S87;

        when WH_S87 =>
                OutDataReady <= '0';
                State <= WH_S88;

        when WH_S88 =>
                r_alu <= JAND(67, C255);
                State <= WH_S89;

        when WH_S89 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(r_alu, C8);
                State <= WH_S90;

        when WH_S90 =>
                OutDataReady <= '0';
                State <= WH_S91;

                -- for loop

        when WH_S91 =>
                i <= 0;
                State <= WH_S92;

        when WH_S92 =>
                if (i < 64) then
                        State <= WH_S93;
                else
                        State <= WH_S98;
                end if;

        when WH_S93 =>
                r_ram1 <= QuantizationMatrix(i);
```

```
                State <= WH_S94;

        when WH_S94 =>
                if (r_ram1 > 255) then
                        State <= WH_S95;
                else
                        State <= WH_S96;
                end if;

        when WH_S95 =>
                Pq := 16;
                State <= WH_S97;

        when WH_S96 =>
                Pq := 0;
                State <= WH_S97;

        when WH_S97 =>
                i <= i + 1;
                State <= WH_S92;

                -- WriteByte(Pq|0)

        when WH_S98 =>
                r_alu <= JOR(Pq, 0);
                State <= WH_S99;

        when WH_S99 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(r_alu, C8);
                State <= WH_S100;

        when WH_S100 =>
                OutDataReady <= '0';
                State <= WH_S101;

                -- for loop

        when WH_S101 =>
                i <= 0;
                State <= WH_S102;

        when WH_S102 =>
                if (i < 64) then
                        State <= WH_S103;
                else
                        State <= WH_S115;
                end if;

        when WH_S103 =>
                r_ram1 <= IZigzagIndex(i);
                State <= WH_S104;

        when WH_S104 =>
                r_ram1 <= QuantizationMatrix(r_ram1);
                State <= WH_S105;

        when WH_S105 =>
                if (Pq = 0) then
                        State <= WH_S112;
                else
                        State <= WH_S106;
                end if;

                -- WriteWord(QuantizationMatrix[IZigzagIndex[i]])
```

```
when WH_S106 =>
        r_alu <= JRS(r_ram1, C8);
        State <= WH_S107;

when WH_S107 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(r_alu, C8);
        State <= WH_S108;

when WH_S108 =>
        OutDataReady <= '0';
        State <= WH_S109;

when WH_S109 =>
        r_alu <= JAND(r_ram1, C255);
        State <= WH_S110;

When WH_S110 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(r_alu, C8);
        State <= WH_S111;

when WH_S111 =>
        OutDataReady <= '0';
        State <= WH_S114;

        -- WriteByte(QuantizationMatrix[IZigzagIndex[i]])

when WH_S112 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(r_ram1, C8);
        State <= WH_S113;

when WH_S113 =>
        OutDataReady <= '0';
        State <= WH_S114;

when WH_S114 =>
        i <= i + 1;
        State <= WH_S102;

-- WriteDHT()

when WH_S115 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(C255, C8);
        State <= WH_S116;

when WH_S116 =>
        OutDataReady <= '0';
        State <= WH_S117;

when WH_S117 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(M_DHT, C8);
        State <= WH_S118;

when WH_S118 =>
        OutDataReady <= '0';
        State <= WH_S119;

        -- WriteWord(0x4+0x20+oxc+0xa2)

when WH_S119 =>
        r_alu <= JRS(210, C8);
        State <= WH_S120;
```

```
when WH_S120 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(r_alu, C8);
        State <= WH_S121;

when WH_S121 =>
        OutDataReady <= '0';
        State <= WH_S122;

when WH_S122 =>
        r_alu <= JAND(210, C255);
        State <= WH_S123;

when WH_S123 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(r_alu, C8);
        State <= WH_S124;

when WH_S124 =>
        OutDataReady <= '0';
        State <= WH_S125;

        -- Write DC Huffman
        -- WriteByte(0)

when WH_S125 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(0, C8);
        State <= WH_S126;

when WH_S126 =>
        OutDataReady <= '0';
        State <= WH_S127;

        -- UseDCHuffman(); WriteHuffman();
                -- for loop 1

when WH_S127 =>
        i <= 0;
        State <= WH_S128;

when WH_S128 =>
        if (i < 16) then
                State <= WH_S129;
        else
                State <= WH_S133;
        end if;

                -- WriteByte(Xhuff->bits[i])

when WH_S129 =>
        r_ram1 <= LuminanceDCBits(i);
        State <= WH_S130;

when WH_S130 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(r_ram1, C8);
        State <= WH_S131;

when WH_S131 =>
        OutDataReady <= '0';
        State <= WH_S132;

when WH_S132 =>
        i <= i + 1;
```

```
                State <= WH_S128;

                        -- for loop 2

        when WH_S133 =>
                i <= 0;
                State <= WH_S134;

        when WH_S134 =>
                if (i < 12) then
                        State <= WH_S135;
                else
                        State <= WH_S139;
                end if;

                        -- WriteByte(Xhuff->huffval[i])

        when WH_S135 =>
                r_ram1 <= LuminanceDCValues(i);
                State <= WH_S136;

        when WH_S136 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(r_ram1, C8);
                State <= WH_S137;

        when WH_S137 =>
                OutDataReady <= '0';
                State <= WH_S138;

        when WH_S138 =>
                i <= i + 1;
                State <= WH_S134;

                -- Write AC Huffman
                -- WriteByte(0x10)

        when WH_S139 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(16, C8);
                State <= WH_S140;

        when WH_S140 =>
                OutDataReady <= '0';
                State <= WH_S141;

                -- UseACHuffman(); WriteHuffman();
                        -- for loop 1
        when WH_S141 =>
                i <= 0;
                State <= WH_S142;

        when WH_S142 =>
                if (i < 16) then
                        State <= WH_S143;
                else
                        State <= WH_S147;
                end if;

                        -- WriteByte(Xhuff->bits[i])

        when WH_S143 =>
                r_ram1 <= LuminanceACBits(i);
                State <= WH_S144;

        when WH_S144 =>
```

```
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(r_ram1, C8);
                State <= WH_S145;

        when WH_S145 =>
                OutDataReady <= '0';
                State <= WH_S146;

        when WH_S146 =>
                i <= i + 1;
                State <= WH_S142;

                        -- for loop 2

        when WH_S147 =>
                i <= 0;
                State <= WH_S148;

        when WH_S148 =>
                if (i < 162) then
                        State <= WH_S149;
                else
                        State <= WH_S153;
                end if;

                        -- WriteByte(Xhuff->huffval[i])

        when WH_S149 =>
                r_ram1 <= LuminanceACValues(i);
                State <= WH_S150;

        when WH_S150 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(r_ram1, C8);
                State <= WH_S151;

        when WH_S151 =>
                OutDataReady <= '0';
                State <= WH_S152;

        when WH_S152 =>
                i <= i + 1;
                State <= WH_S148;

        -- WriteSOS()

        when WH_S153 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(C255, C8);
                State <= WH_S154;

        when WH_S154 =>
                OutDataReady <= '0';
                State <= WH_S155;

        when WH_S155 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(M_SOS, C8);
                State <= WH_S156;

        when WH_S156 =>
                OutDataReady <= '0';
                State <= WH_S157;

                -- WriteWord(8)
```

```
when WH_S157 =>
        r_alu <= JRS(8, C8);
        State <= WH_S158;

when WH_S158 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(r_alu, C8);
        State <= WH_S159;

when WH_S159 =>
        OutDataReady <= '0';
        State <= WH_S160;

when WH_S160 =>
        r_alu <= JAND(8, C255);
        State <= WH_S161;

when WH_S161 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(r_alu, C8);
        State <= WH_S162;

when WH_S162 =>
        OutDataReady <= '0';
        State <= WH_S163;

        -- WriteByte(1)

when WH_S163 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(1, C8);
        State <= WH_S164;

when WH_S164 =>
        OutDataReady <= '0';
        State <= WH_S165;

        -- WriteByte(1)

when WH_S165 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(1, C8);
        State <= WH_S166;

when WH_S166 =>
        OutDataReady <= '0';
        State <= WH_S167;

        -- Writebyte(0)

when WH_S167 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(0, C8);
        State <= WH_S168;

when WH_S168 =>
        OutDataReady <= '0';
        State <= WH_S169;

        -- WriteByte(0)

when WH_S169 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(0, C8);
        State <= WH_S170;
```

```
when WH_S170 =>
        OutDataReady <= '0';
        State <= WH_S171;

        -- WriteByte(63)

when WH_S171 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(63, C8);
        State <= WH_S172;

when WH_S172 =>
        OutDataReady <= '0';
        State <= WH_S173;

        -- WriteByte(0)

when WH_S173 =>
        OutDataReady <= '1';
        OutData <= CONV_UNSIGNED(0, C8);
        State <= WH_S174;

when WH_S174 =>
        OutDataReady <= '0';
        State <= HD_S10;

----------------------------------------------------------------

when HD_S10 =>
        r_ram1 <= C0;
        State <= HD_S11;

when HD_S11 =>
        k <= r_ram1;
        State <= HD_S12;

when HD_S12 =>
        if(k < rf(3)) then
                State <=  HD_S13;
        else
                WB_Return_State <= WH_S175;
                State <=  HD_S12_0;
        end if;

when HD_S12_0 =>
        r_ram1 <= C_1;
        State <= HD_S12_0_1;

when HD_S12_0_1 =>
        rf(21) <= r_ram1;
        State <= HD_S12_0_2;

when HD_S12_0_2 =>
        r_ram1 <= C0;
        State <= HD_S12_0_3;

when HD_S12_0_3 =>
        rf(20) <= r_ram1;
        State <= WB_S_0;

------------------------------------------------------------

        -- WriteMarker(M_EOI)

when WH_S175 =>
        OutDataReady <= '1';
```

```
                OutData <= CONV_UNSIGNED(C255, C8);
                State <= WH_S176;

        when WH_S176 =>
                OutDataReady <= '0';
                State <= WH_S177;

        when WH_S177 =>
                OutDataReady <= '1';
                OutData <= CONV_UNSIGNED(M_EOI, C8);
                State <= WH_S178;

        when WH_S178 =>
                OutDataReady <= '0';
                State <= HD_S0;

        ------------------------------------------------

        when HD_S13 =>
                r_ram1 <= C0;
                State <= HD_S14;

        when HD_S14 =>
                i <= r_ram1;
                State <= HD_S15;

        when HD_S15 =>
                r_ram2 <= C8;
                State <= HD_S16;

        when HD_S16 =>
                if(i < r_ram2) then
                        State <= HD_S17;
                else
                        State <= HD_S34;
                end if;

        when HD_S17 =>
                r_ram1 <= C0;
                State <= HD_S18;

        when HD_S18 =>
                j <= r_ram1;
                State <= HD_S19;

        when HD_S19 =>
                r_ram2 <= C8;
                State <= HD_S20;

        when HD_S20 =>
                r_mul <= rf(2) * r_ram2;
                State <= HD_S21;

        when HD_S21 =>
                if(j < r_mul) then
                        State <= HD_S22;
                else
                        State <= HD_S21_1;
                end if;

        when HD_S21_1 =>
                i <= i + 1; -- counter
                State <= HD_S15;

        when HD_S22 =>
                r_mul <= k * r_ram2;
```

```
            State <= HD_S22_1;

        when HD_S22_1 =>
            if(j < rf(5)) then
                State <= HD_S23;
            else
                State <= HD_S30;
            end if;

        when HD_S23 =>
            r_alu <= r_mul + i;
            State <= HD_S24;

        when HD_S24 =>
            if(r_alu < rf(6)) then
                State <= HD_S25;
            else
                State <= HD_S30;
            end if;

        when HD_S25 =>
            r_mul <= rf(2) * i;
            State <= HD_S26;

        when HD_S26 =>
            r_mul <= r_mul * r_ram2;
            State <= HD_S27;

        when HD_S27 =>
            r_alu <= r_mul + j;
            State <= HD_S28;
            InReady <= '1';

        when HD_S28 =>
            if(InDataValid = '1') then
                State <= HD_S29;
            else
                State <= HD_S28;
            end if;

        when HD_S29 =>
            stripe(r_alu) := CONV_INTEGER(InData);
            InReady <= '0';
            j <= j + 1; -- counter
            State <= HD_S19;

        when HD_S30 =>
            r_mul <=rf(2)* i;
            r_ram1 <= C1;
            State <= HD_S31;

        when HD_S31 =>
            r_alu <= rf(5) - r_ram1;
            State <= HD_S31_1;

        when HD_S31_1 =>
            r_mul <= r_mul * r_ram2;
            State <= HD_S31_2;

        when HD_S31_2 =>
            rf(4) <= r_alu;
            State <= HD_S31_3;

        when HD_S31_3 =>
            r_alu <= r_mul + j;
            State <= HD_S32;
```

```
when HD_S32 =>
        r_ram1 <= stripe(rf(4));
        State <= HD_S33;

when HD_S33 =>
        stripe(r_alu) := r_ram1;
        j <= j + 1; -- counter
        State <= HD_S19;

when HD_S34 =>
        r_ram1 <= C0;
        State <= HD_S35;

when HD_S35 =>
        i <= r_ram1;
        State <= HD_S36;

when HD_S36 =>
        if(i < rf(2)) then
                State <= HD_S37;
        else
                State <= HD_S36_1;
        end if;

when HD_S36_1 =>
        k <= k + 1; -- counter
        State <= HD_S12;

when HD_S37 =>
        r_ram1 <= C0;
        State <= HD_S38;

when HD_S38 =>
        j <= r_ram1;
        State <= HD_S39;

when HD_S39 =>
        r_ram2 <= C8;
        State <= HD_S40;

when HD_S40 =>
        if(j < r_ram2) then
                State <= HD_S41;
        else
                State <=  HD_S40_1;
        end if;

when HD_S40_1 =>
        rf(28) <= i;
        State <= HD_S40_2;

when HD_S40_2 =>
        rf(29) <= j;
        State <= HD_S40_3;

when HD_S40_3 =>
        rf(31) <= k;
        State <= HD_S40_4;

when HD_S40_4 =>
        mMDUHigh := rf(3);
        State <= HD_S40_4_1;

when HD_S40_4_1 =>
        mImageWidth := rf(5);
```

```
                State <= HD_S40_4_2;

        when HD_S40_4_2 =>
                mImageHeight := rf(6);
                State <= HD_S40_4_3;

        when HD_S40_4_3 =>
                temp := rf(4);
                State <= HD_S40_5;

        when HD_S40_5 =>
                mMDUWide := rf(2);
                PDM_Return_State <= HD_S54_1;
                State <= PDM_S0;

        when HD_S41 =>
                r_ram1 <= C0;
                State <= HD_S42;

        when HD_S42 =>
                rf(30) <= r_ram1;
                State <= HD_S43;

        when HD_S43 =>
                r_ram2 <= C8;
                State <= HD_S44;

        when HD_S44 =>
                if (rf(30) < r_ram2) then
                        State <= HD_S45;
                else
                        State <= HD_S44_1;
                end if;

        when HD_S44_1 =>
                j <= j + 1; -- counter
                State <= HD_S39;

        when HD_S45 =>
                r_mul <= j * r_ram2;
                State <= HD_S46;

        when HD_S46 =>
                r_alu <= r_mul + rf(30);
                State <= HD_S47;

        when HD_S47 =>
                rf(4) <= r_alu;
                State <= HD_S48;

        when HD_S48 =>
                r_mul <= i * r_ram2;
                State <= HD_S49;

        when HD_S49 =>
                r_alu <= r_mul + rf(30);
                State <= HD_S50;

        when HD_S50 =>
                r_mul <=  j * r_ram2;
                State <= HD_S51;

        when HD_S51 =>
                r_mul <= r_mul * rf(2);
                State <= HD_S52;
```

```
when HD_S52 =>
        r_alu <= r_alu + r_mul;
        State <= HD_S53;

when HD_S53 =>
        r_ram1 <= stripe(r_alu);
        State <= HD_S54;

when HD_S54 =>
        input(rf(4)) := r_ram1;
        r_ram2 <= C1;
        State <= HD_S54_0;

when HD_S54_0 =>
        r_alu <= rf(30) + r_ram2;
        State <= HD_S54_0_1;

when HD_S54_0_1 =>
        rf(30) <= r_alu;
        State <= HD_S43;

when HD_S54_1 =>
        i <= rf(28);
        j <= rf(29);
        State <= HD_S54_2;

when HD_S54_2 =>
        k <= rf(31);
        State <= HD_S54_3;

when HD_S54_3 =>
        r_ram1 <= mMDUWide;
        r_ram2 <= mMDUHigh;
        State <= HD_S54_4;

when HD_S54_4 =>
        rf(2) <= r_ram1;
        rf(3) <= r_ram2;
        State <= HD_S54_4_1;

when HD_S54_4_1 =>
        r_ram1 <= mImageWidth;
        r_ram2 <= mImageHeight;
        State <= HD_S54_4_2;

when HD_S54_4_2 =>
        rf(5) <= r_ram1;
        rf(6) <= r_ram2;
        State <= HD_S54_4_3;

when HD_S54_4_3 =>
        r_ram1 <= temp;
        State <= HD_S54_4_4;

when HD_S54_4_4 =>
        rf(4) <= r_ram1;
        State <= HD_S55;

when HD_S55 =>
        r_ram1 <= C1;
        State <= HD_S56;

when HD_S56 =>
        r_alu <= rf(3) - r_ram1;
        State <= HD_S57;
```

```
when HD_S57 =>
        if(k = r_alu) then
                State <= HD_S58;
        else
                State <=  HD_S61;
        end if;

when HD_S58 =>
        r_alu <= rf(2) - r_ram1;
        State <= HD_S59;

when HD_S59 =>
        if(i = r_alu) then
                State <= HD_S60;
        else
                State <= HD_S61;
        end if;

when HD_S60 =>
        i <= i + 1; -- counter
        State <= HD_S36;

when HD_S61 =>
        i <= i + 1; -- counter
        State <= HD_S36;

when PDM_S0 =>
        i <= 0;
        State <= PD_S1;

--------------------------------------------
--        State PD_S1
--------------------------------------------
when PD_S1 =>
        if (i < 64) then
                State <= PD_S2;
        else
                State <= PD_S5;
        end if;

--------------------------------------------
--        State PD_S2
--------------------------------------------
when PD_S2 =>
        r_ram1 <= input(i);
        State <= PD_S3;

--------------------------------------------
--        State PD_S3
--------------------------------------------
when PD_S3 =>
        r_alu <= r_ram1 - 128;
        State <= PD_S4;

--------------------------------------------
--        State PD_S4
--------------------------------------------
when PD_S4 =>
        input(i) := r_alu;
        i<=i+1;
        State <= PD_S1;

--------------------------------------------
--        State PD_S5
--------------------------------------------
when PD_S5 =>
```

```
            i<=0;
            State <=CDCT_S1;

------------------------------------------------
--          State  CDCT_S1
------------------------------------------------
when CDCT_S1 =>
            if ( i<8) then
                        State <=CDCT_S2;
            else
                        State <=CDCT_S72;
            end if;


------------------------------------------------
--          State  CDCT_S2
------------------------------------------------
when CDCT_S2 =>
            rf(0)<=i;
            r_alu<=i+C56;
            State <=CDCT_S3;


------------------------------------------------
--          State  CDCT_S3
------------------------------------------------
when CDCT_S3 =>
            rf(1)<=r_alu;
            State <=CDCT_S4;


------------------------------------------------
--          State  CDCT_S4
------------------------------------------------
when CDCT_S4 =>
            r_ram1<=input(rf(0));
            r_ram2<=input(rf(1));
            State<=CDCT_S5;

------------------------------------------------
--          State  CDCT_S5
------------------------------------------------
when CDCT_S5 =>
            r_alu<=r_ram1+r_ram2;
            State<=CDCT_S6;

------------------------------------------------
--          State  CDCT_S6
------------------------------------------------
when CDCT_S6 =>
            r_sh<=JLS(r_alu,C2);
            r_alu<=r_ram1-r_ram2;
            State<=CDCT_S7;

------------------------------------------------
--          State  CDCT_S7
------------------------------------------------
when CDCT_S7 =>
            rf(2)<=r_sh;
            r_sh<=JLS(r_alu,C2);
            State<=CDCT_S8;

------------------------------------------------
--          State  CDCT_S8
------------------------------------------------
when CDCT_S8 =>
```

```
                rf(13)<=r_sh;
                State<=CDCT_S9;

--------------------------------------------------------------
--      State CDCT_S9
--------------------------------------------------------------
when CDCT_S9 =>
                rf(0)<=rf(0)+C8;
                State<=CDCT_S10;

--------------------------------------------------------------
--      State CDCT_S10
--------------------------------------------------------------
when CDCT_S10 =>
                rf(1)<=rf(1)-C8;
                State<=CDCT_S11;


---------------------------------------------------------------
---------------------------------------------------------------
---------------------------------------------------------------

--------------------------------------------------------------
--      State CDCT_S11
--------------------------------------------------------------
when CDCT_S11 =>
                r_ram1<=input(rf(0));
                r_ram2<=input(rf(1));
                State<=CDCT_S12;

--------------------------------------------------------------
--      State CDCT_S12
--------------------------------------------------------------
when CDCT_S12 =>
                r_alu<=r_ram1+r_ram2;
                State<=CDCT_S13;

--------------------------------------------------------------
--      State CDCT_S13
--------------------------------------------------------------
when CDCT_S13 =>
                r_sh<=JLS(r_alu,C2);
                r_alu<=r_ram1-r_ram2;
                State<=CDCT_S14;

--------------------------------------------------------------
--      State CDCT_S14
--------------------------------------------------------------
when CDCT_S14 =>
                rf(3)<=r_sh;
                r_sh<=JLS(r_alu,C2);
                State<=CDCT_S15;

--------------------------------------------------------------
--      State CDCT_S15
--------------------------------------------------------------
when CDCT_S15 =>
                rf(12)<=r_sh;
                State<=CDCT_S16;

--------------------------------------------------------------
--      State CDCT_S16
--------------------------------------------------------------
when CDCT_S16 =>
                rf(0)<=rf(0)+C8;
                State<=CDCT_S17;
```

```
--------------------------------------------------
--          State  CDCT_S17
--------------------------------------------------
when CDCT_S17 =>
            rf(1)<=rf(1)-C8;
            State<=CDCT_S18;
--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
--          State  CDCT_S18
--------------------------------------------------
when CDCT_S18 =>
            r_ram1<=input(rf(0));
            r_ram2<=input(rf(1));
            State<=CDCT_S19;

--------------------------------------------------
--          State  CDCT_S19
--------------------------------------------------
when CDCT_S19 =>
            r_alu<=r_ram1+r_ram2;
            State<=CDCT_S20;

--------------------------------------------------
--          State  CDCT_S20
--------------------------------------------------
when CDCT_S20 =>
            r_sh<=JLS(r_alu,C2);
            r_alu<=r_ram1-r_ram2;
            State<=CDCT_S21;

--------------------------------------------------
--          State  CDCT_S21
--------------------------------------------------
when CDCT_S21 =>
            rf(4)<=r_sh;
            r_sh<=JLS(r_alu,C2);
            State<=CDCT_S22;

--------------------------------------------------
--          State  CDCT_S22
--------------------------------------------------
when CDCT_S22 =>
            rf(11)<=r_sh;
            State<=CDCT_S23;

--------------------------------------------------
--          State  CDCT_S23
--------------------------------------------------
when CDCT_S23 =>
            rf(0)<=rf(0)+C8;
            State<=CDCT_S24;

--------------------------------------------------
--          State  CDCT_S24
--------------------------------------------------
when CDCT_S24 =>
            rf(1)<=rf(1)-C8;
            State<=CDCT_S25;

--------------------------------------------------
--------------------------------------------------
--------------------------------------------------
--          State  CDCT_S25
--------------------------------------------------
```

```
when CDCT_S25 =>
          r_ram1<=input(rf(0));
          r_ram2<=input(rf(1));
          State<=CDCT_S26;

-------------------------------------------------------
--        State  CDCT_S26
-------------------------------------------------------
when CDCT_S26 =>
          r_alu<=r_ram1+r_ram2;
          State<=CDCT_S27;

-------------------------------------------------------
--        State  CDCT_S27
-------------------------------------------------------
when CDCT_S27 =>
          r_sh<=JLS(r_alu,C2);
          r_alu<=r_ram1-r_ram2;
          State<=CDCT_S28;

-------------------------------------------------------
--        State  CDCT_S28
-------------------------------------------------------
when CDCT_S28 =>
          rf(5)<=r_sh;
          r_sh<=JLS(r_alu,C2);
          State<=CDCT_S29;

-------------------------------------------------------
--        State  CDCT_S29
-------------------------------------------------------
when CDCT_S29 =>
          rf(10)<=r_sh;
          r_alu<=rf(2)+rf(5);
          State<=CDCT_S30;

-------------------------------------------------------
--        State  CDCT_S30
-------------------------------------------------------
when CDCT_S30 =>
          rf(6)<=r_alu;
          r_alu<=rf(3)+rf(4);
          State<=CDCT_S31;


-------------------------------------------------------
--        State  CDCT_S31
-------------------------------------------------------
when CDCT_S31 =>
          rf(7)<=r_alu;
          r_alu<=rf(3)-rf(4);
          State<=CDCT_S32;

-------------------------------------------------------
--        State  CDCT_S32
-------------------------------------------------------
when CDCT_S32 =>
          rf(8)<=r_alu;
          r_alu<=rf(2)-rf(5);
          State<=CDCT_S33;

-------------------------------------------------------
--        State  CDCT_S33
-------------------------------------------------------
when CDCT_S33 =>
          rf(9)<=r_alu;
```

```
        State<=CDCT_S34;

------------------------------------------------
--      State  CDCT_S34
------------------------------------------------
when CDCT_S34 =>
        rf(0)<=i;
        State<=CDCT_S35;

------------------------------------------------
--      State  CDCT_S35
------------------------------------------------
when CDCT_S35 =>
        r_alu<=rf(6)+rf(7);
        State<=CDCT_S36;

------------------------------------------------
--      State  CDCT_S36
------------------------------------------------
when CDCT_S36 =>
        r_alu<=rf(6)-rf(7);
        r_mul<=C362*r_alu;
        State<=CDCT_S37;

------------------------------------------------
--      State  CDCT_S37
------------------------------------------------
when CDCT_S37 =>
        r_sh<=JRS(r_mul,C9);
        r_mul<=C362*r_alu;
        State<=CDCT_S38;


------------------------------------------------
--      State  CDCT_S38
------------------------------------------------
when CDCT_S38 =>
        r_sh<=JRS(r_mul,C9);
        output(rf(0)) := r_sh;
        State<=CDCT_S39;

------------------------------------------------
--      State  CDCT_S39
------------------------------------------------
when CDCT_S39 =>
        r_alu<=i+C32;
        State<=CDCT_S40;

------------------------------------------------
--      State  CDCT_S40
------------------------------------------------
when CDCT_S40 =>
        output(r_alu) := r_sh;
        State<=CDCT_S41;

------------------------------------------------
--      State  CDCT_S41
------------------------------------------------
when CDCT_S41 =>
        r_mul<=C196*rf(8);
        State<=CDCT_S42;

------------------------------------------------
--      State  CDCT_S42
------------------------------------------------
when CDCT_S42 =>
```

```
            r_mul<=C473*rf(9);
            rf(15)<=r_mul;
            State<=CDCT_S43;

-------------------------------------------------
--          State  CDCT_S43
-------------------------------------------------
when CDCT_S43 =>
            r_alu<=r_mul+rf(15);
            r_mul<=C196*rf(9);
            State<=CDCT_S44;

-------------------------------------------------
--          State  CDCT_S44
-------------------------------------------------
when CDCT_S44 =>
            r_sh<=JRS(r_alu,C9);
            r_mul<=C473*rf(8);
            rf(14)<=r_mul;
            State<=CDCT_S45;

-------------------------------------------------
--          State  CDCT_S45
-------------------------------------------------
when CDCT_S45 =>
            rf(15)<=r_sh;
            r_alu<=rf(14)-r_mul;
            State<=CDCT_S46;

-------------------------------------------------
--          State  CDCT_S46
-------------------------------------------------
when CDCT_S46 =>
            r_alu<=i+C16;
            r_sh<=JRS(r_alu,C9);
            State<=CDCT_S47;


-------------------------------------------------
--          State  CDCT_S47
-------------------------------------------------
when CDCT_S47 =>
            output(r_alu) := rf(15);
            r_alu<=i+C48;
            State<=CDCT_S48;

-------------------------------------------------
--          State  CDCT_S48
-------------------------------------------------
when CDCT_S48 =>
            output(r_alu):=r_sh;
            State<=CDCT_S49;

-------------------------------------------------
--          State  CDCT_S49
-------------------------------------------------
when CDCT_S49 =>
            r_alu<=rf(12)-rf(11);
            State<=CDCT_S50;

-------------------------------------------------
--          State  CDCT_S50
-------------------------------------------------
when CDCT_S50 =>
            r_alu<=rf(12)+rf(11);
            r_mul<=r_alu*C362;
```

```
            State<=CDCT_S51;

-----------------------------------------------
--          State  CDCT_S51
-----------------------------------------------
when CDCT_S51 =>
            r_sh <=JRS(r_mul ,  C9);
            r_mul<=r_alu *C362;
            State<=CDCT_S52;

-----------------------------------------------
--          State  CDCT_S52
-----------------------------------------------
when CDCT_S52 =>
            r_sh <=JRS(r_mul ,  C9);
            rf(6)<=r_sh ;
            State<=CDCT_S53;

-----------------------------------------------
--          State  CDCT_S53
-----------------------------------------------
when CDCT_S53 =>
            rf(7)<=r_sh ;
            r_alu<=rf(10)+rf(6);
            State<=CDCT_S54;

-----------------------------------------------
--          State  CDCT_S54
-----------------------------------------------
when CDCT_S54 =>
            r_alu<=rf(10)-rf(6);
            rf(2)<=r_alu ;
            State<=CDCT_S55;

-----------------------------------------------
--          State  CDCT_S55
-----------------------------------------------
when CDCT_S55 =>
            r_alu<=rf(13)-rf(7);
            rf(3)<=r_alu ;
            State<=CDCT_S56;

-----------------------------------------------
--          State  CDCT_S56
-----------------------------------------------
when CDCT_S56 =>
            r_alu<=rf(13)+rf(7);
            rf(4)<=r_alu ;
            State<=CDCT_S57;

-----------------------------------------------
--          State  CDCT_S57
-----------------------------------------------
when CDCT_S57 =>
            rf(5)<=r_alu ;
            r_mul<=C100*rf(2);
            State<=CDCT_S58;
-----------------------------------------------
-----------------------------------------------
-----------------------------------------------

-----------------------------------------------
--          State  CDCT_S58
-----------------------------------------------
when CDCT_S58 =>
            r_mul<=C502*rf(5);
```

```
            rf(15)<=r_mul;
            State<=CDCT_S59;

-------------------------------------------------
--          State  CDCT_S59
-------------------------------------------------
when CDCT_S59 =>
            r_alu<=r_mul+rf(15);
            r_mul<=C426*rf(4);
            State<=CDCT_S60;

-------------------------------------------------
--          State  CDCT_S60
-------------------------------------------------
when CDCT_S60 =>
            r_sh<=JRS(r_alu,C9);
            r_mul<=C284*rf(3);
            rf(14)<=r_mul;
            State<=CDCT_S61;

-------------------------------------------------
--          State  CDCT_S61
-------------------------------------------------
when CDCT_S61 =>
            rf(15)<=r_sh;
            r_alu<=rf(14)-r_mul;
            State<=CDCT_S62;

-------------------------------------------------
--          State  CDCT_S62
-------------------------------------------------
when CDCT_S62 =>
            r_alu<=i+C8;
            r_sh<=JRS(r_alu,C9);
            State<=CDCT_S63;


-------------------------------------------------
--          State  CDCT_S63
-------------------------------------------------
when CDCT_S63 =>
            output(r_alu) := rf(15);
            r_alu<=i+C24;
            State<=CDCT_S64;

-------------------------------------------------
--          State  CDCT_S64
-------------------------------------------------
when CDCT_S64 =>
            output(r_alu):=r_sh;
            r_mul<=C426*rf(3);
            State<=CDCT_S65;
-------------------------------------------------
-------------------------------------------------
-------------------------------------------------
-------------------------------------------------
--          State  CDCT_S65
-------------------------------------------------
when CDCT_S65 =>
            r_mul<=C284*rf(4);
            rf(15)<=r_mul;
            State<=CDCT_S66;

-------------------------------------------------
--          State  CDCT_S66
-------------------------------------------------
```

```
when CDCT_S66 =>
        r_alu<=r_mul+rf(15);
        r_mul<=C100*rf(5);
        State<=CDCT_S67;

------------------------------------------------------------
--      State CDCT_S67
------------------------------------------------------------
when CDCT_S67 =>
        r_sh<=JRS(r_alu,C9);
        r_mul<=C502*rf(2);
        rf(14)<=r_mul;
        State<=CDCT_S68;

------------------------------------------------------------
--      State CDCT_S68
------------------------------------------------------------
when CDCT_S68 =>
        rf(15)<=r_sh;
        r_alu<=rf(14)-r_mul;
        State<=CDCT_S69;

------------------------------------------------------------
--      State CDCT_S69
------------------------------------------------------------
when CDCT_S69 =>
        r_alu<=i+C40;
        r_sh<=JRS(r_alu,C9);
        State<=CDCT_S70;


------------------------------------------------------------
--      State CDCT_S70
------------------------------------------------------------
when CDCT_S70 =>
        output(r_alu) := rf(15);
        r_alu<=i+C56;
        State<=CDCT_S71;

------------------------------------------------------------
--      State CDCT_S71
------------------------------------------------------------
when CDCT_S71 =>
        output(r_alu):=r_sh;
        i<=i+1;
        State<=CDCT_S1;

------------------------------------------------------------
--      State CDCT_S72
------------------------------------------------------------
when CDCT_S72 =>
        i<=0;
        State<=CDCT_S73;

------------------------------------------------------------
--      State CDCT_S73
------------------------------------------------------------
when CDCT_S73 =>
        if (i<C8) then
                State <=CDCT_S74;
        else
                State <=CDCT_S139;
        end if;

------------------------------------------------------------
--      State CDCT_S74
```

```
--------------------------------------------------
when CDCT_S74 =>
        r_sh <=JLS(i , C3);
        State<=CDCT_S75;

--------------------------------------------------
--      State  CDCT_S75
--------------------------------------------------
when CDCT_S75 =>
        j<=r_sh ;
        r_alu<=r_sh +C7;
        State<=CDCT_S76;

--------------------------------------------------
--      State  CDCT_S76
--------------------------------------------------
when CDCT_S76 =>
        k<=r_alu ;
        State<=CDCT_S77;

--------------------------------------------------
--      State  CDCT_S77
--------------------------------------------------
when CDCT_S77 =>
        r_ram1<=output(j);
        r_ram2<=output(k);
        State<=CDCT_S78;

--------------------------------------------------
--      State  CDCT_S78
--------------------------------------------------
when CDCT_S78 =>
        r_alu<=r_ram1-r_ram2;
        State<=CDCT_S79;

--------------------------------------------------
--      State  CDCT_S79
--------------------------------------------------
when CDCT_S79 =>
        r_alu<=r_ram1+r_ram2;
        r_sh <=JRS(r_alu , C1);
        j<=j+C1;
        k<=k-C1;
        State<=CDCT_S80;

--------------------------------------------------
--      State  CDCT_S80
--------------------------------------------------
when CDCT_S80 =>
        rf(13)<=r_sh ;
        r_sh <=JRS(r_alu , C1);
        State<=CDCT_S81;

--------------------------------------------------
--      State  CDCT_S81
--------------------------------------------------
when CDCT_S81 =>
        rf(2)<=r_sh ;
        State<=CDCT_S82;
--------------------------------------------------
--------------------------------------------------

--------------------------------------------------
--      State  CDCT_S82
--------------------------------------------------
when CDCT_S82 =>
```

```
            r_ram1<=output(j);
            r_ram2<=output(k);
            State<=CDCT_S83;

------------------------------------------------------------
--          State CDCT_S83
------------------------------------------------------------
when CDCT_S83 =>
            r_alu<=r_ram1-r_ram2;
            State<=CDCT_S84;

------------------------------------------------------------
--          State CDCT_S84
------------------------------------------------------------
when CDCT_S84 =>
            r_alu<=r_ram1+r_ram2;
            r_sh<=JRS(r_alu,C1);
            j<=j+1;
            k<=k-1;
            State<=CDCT_S85;

------------------------------------------------------------
--          State CDCT_S85
------------------------------------------------------------
when CDCT_S85 =>
            rf(12)<=r_sh;
            r_sh<=JRS(r_alu,C1);
            State<=CDCT_S86;

------------------------------------------------------------
--          State CDCT_S86
------------------------------------------------------------
when CDCT_S86 =>
            rf(3)<=r_sh;
            State<=CDCT_S87;

------------------------------------------------------------
------------------------------------------------------------
--          State CDCT_S87
------------------------------------------------------------
when CDCT_S87 =>
            r_ram1<=output(j);
            r_ram2<=output(k);
            State<=CDCT_S88;

------------------------------------------------------------
--          State CDCT_S88
------------------------------------------------------------
when CDCT_S88 =>
            r_alu<=r_ram1-r_ram2;
            State<=CDCT_S89;

------------------------------------------------------------
--          State CDCT_S89
------------------------------------------------------------
when CDCT_S89 =>
            r_alu<=r_ram1+r_ram2;
            r_sh<=JRS(r_alu,C1);
            j<=j+1;
            k<=k-1;
            State<=CDCT_S90;

------------------------------------------------------------
--          State CDCT_S90
------------------------------------------------------------
when CDCT_S90 =>
```

```
            rf(11)<=r_sh;
            r_sh<=JRS(r_alu,C1);
            State<=CDCT_S91;

------------------------------------------------
--          State CDCT_S91
------------------------------------------------
when CDCT_S91 =>
            rf(4)<=r_sh;
            State<=CDCT_S92;

------------------------------------------------
--          State CDCT_S92
------------------------------------------------
when CDCT_S92 =>
            r_ram1<=output(j);
            r_ram2<=output(k);
            State<=CDCT_S93;

------------------------------------------------
--          State CDCT_S93
------------------------------------------------
when CDCT_S93 =>
            r_alu<=r_ram1-r_ram2;
            State<=CDCT_S94;

------------------------------------------------
--          State CDCT_S94
------------------------------------------------
when CDCT_S94 =>
            r_alu<=r_ram1+r_ram2;
            r_sh<=JRS(r_alu,C1);
            j<=j+1;
            k<=k-1;
            State<=CDCT_S95;

------------------------------------------------
--          State CDCT_S95
------------------------------------------------
when CDCT_S95 =>
            rf(10)<=r_sh;
            r_sh<=JRS(r_alu,C1);
            State<=CDCT_S96;

------------------------------------------------
--          State CDCT_S96
------------------------------------------------
when CDCT_S96 =>
            rf(5)<=r_sh;
            r_alu<=rf(2)+r_sh;
            State<=CDCT_S97;

------------------------------------------------
--          State CDCT_S97
------------------------------------------------
when CDCT_S97 =>
            rf(6)<=r_alu;
            r_alu<=rf(3)+rf(4);
            State<=CDCT_S98;


------------------------------------------------
--          State CDCT_S98
------------------------------------------------
when CDCT_S98 =>
            rf(7)<=r_alu;
```

```
                r_alu<=rf(3)-rf(4);
                State<=CDCT_S99;

--------------------------------------------
--          State  CDCT_S99
--------------------------------------------
when CDCT_S99 =>
                rf(8)<=r_alu;
                r_alu<=rf(2)-rf(5);
                State<=CDCT_S100;


--------------------------------------------
--          State  CDCT_S100
--------------------------------------------
when CDCT_S100 =>
                rf(9)<=r_alu;
                r_sh<=JLS(i,C3);
                State<=CDCT_S101;

--------------------------------------------
--          State  CDCT_S101
--------------------------------------------
when CDCT_S101 =>
                rf(0)<=r_sh;
                State<=CDCT_S102;

--------------------------------------------
--          State  CDCT_S102
--------------------------------------------
when CDCT_S102 =>
                r_alu<=rf(6)+rf(7);
                State<=CDCT_S103;

--------------------------------------------
--          State  CDCT_S103
--------------------------------------------
when CDCT_S103=>
                r_alu<=rf(6)-rf(7);
                r_mul<=C362*r_alu;
                State<=CDCT_S104;

--------------------------------------------
--          State  CDCT_S104
--------------------------------------------
when CDCT_S104 =>
                r_sh<=JRS(r_mul,C9);
                r_mul<=C362*r_alu;
                State<=CDCT_S105;


--------------------------------------------
--          State  CDCT_S105
--------------------------------------------
when CDCT_S105 =>
                r_sh<=JRS(r_mul,C9);
                output(rf(0)) := r_sh;
                State<=CDCT_S106;

--------------------------------------------
--          State  CDCT_S106
--------------------------------------------
when CDCT_S106 =>
                r_alu<=rf(0)+C4;
                State<=CDCT_S107;
```

```
--------------------------------------------------
--        State  CDCT_S107
--------------------------------------------------
when CDCT_S107 =>
          output(r_alu) := r_sh;
          State<=CDCT_S108;

--------------------------------------------------
--        State  CDCT_S108
--------------------------------------------------
when CDCT_S108 =>
          r_mul<=C196*rf(8);
          State<=CDCT_S109;

--------------------------------------------------
--        State  CDCT_S109
--------------------------------------------------
when CDCT_S109 =>
          r_mul<=C473*rf(9);
          rf(15)<=r_mul;
          State<=CDCT_S110;

--------------------------------------------------
--        State  CDCT_S110
--------------------------------------------------
when CDCT_S110 =>
          r_alu<=r_mul+rf(15);
          r_mul<=C196*rf(9);
          State<=CDCT_S111;

--------------------------------------------------
--        State  CDCT_S111
--------------------------------------------------
when CDCT_S111 =>
          r_sh<=JRS(r_alu,C9);
          r_mul<=C473*rf(8);
          rf(14)<=r_mul;
          State<=CDCT_S112;

--------------------------------------------------
--        State  CDCT_S112
--------------------------------------------------
when CDCT_S112 =>
          rf(15)<=r_sh;
          r_alu<=rf(14)-r_mul;
          State<=CDCT_S113;

--------------------------------------------------
--        State  CDCT_S113
--------------------------------------------------
when CDCT_S113 =>
          r_alu<=rf(0)+C2;
          r_sh<=JRS(r_alu,C9);
          State<=CDCT_S114;


--------------------------------------------------
--        State  CDCT_S114
--------------------------------------------------
when CDCT_S114 =>
          output(r_alu) := rf(15);
          r_alu<=rf(0)+C6;
          State<=CDCT_S115;

--------------------------------------------------
--        State  CDCT_S115
```

```
--------------------------------------------------------------------------------
when CDCT_S115 =>
        output(r_alu):=r_sh;
        State<=CDCT_S116;


--------------------------------------------------------------------------------
--      State CDCT_S116
--------------------------------------------------------------------------------
when CDCT_S116 =>
        r_alu<=rf(12)-rf(11);
        State<=CDCT_S117;

--------------------------------------------------------------------------------
--      State CDCT_S117
--------------------------------------------------------------------------------
when CDCT_S117 =>
        r_alu<=rf(12)+rf(11);
        r_mul<=r_alu*C362;
        State<=CDCT_S118;

--------------------------------------------------------------------------------
--      State CDCT_S118
--------------------------------------------------------------------------------
when CDCT_S118 =>
        r_sh<=JRS(r_mul, C9);
        r_mul<=r_alu*C362;
        State<=CDCT_S119;

--------------------------------------------------------------------------------
--      State CDCT_S119
--------------------------------------------------------------------------------
when CDCT_S119 =>
        r_sh<=JRS(r_mul, C9);
        rf(6)<=r_sh;
        State<=CDCT_S120;

--------------------------------------------------------------------------------
--      State CDCT_S120
--------------------------------------------------------------------------------
when CDCT_S120 =>
        rf(7)<=r_sh;
        r_alu<=rf(10)+rf(6);
        State<=CDCT_S121;

--------------------------------------------------------------------------------
--      State CDCT_S121
--------------------------------------------------------------------------------
when CDCT_S121 =>
        r_alu<=rf(10)-rf(6);
        rf(2)<=r_alu;
        State<=CDCT_S122;

--------------------------------------------------------------------------------
--      State CDCT_S122
--------------------------------------------------------------------------------
when CDCT_S122 =>
        r_alu<=rf(13)-rf(7);
        rf(3)<=r_alu;
        State<=CDCT_S123;

--------------------------------------------------------------------------------
--      State CDCT_S123
--------------------------------------------------------------------------------
when CDCT_S123 =>
        r_alu<=rf(13)+rf(7);
```

```
          rf(4)<=r_alu;
          State<=CDCT_S124;

-----------------------------------------------------------
--        State  CDCT_S124
-----------------------------------------------------------
when CDCT_S124 =>
          rf(5)<=r_alu;
          r_mul<=C100*rf(2);
          State<=CDCT_S125;

-----------------------------------------------------------
--        State  CDCT_S125
-----------------------------------------------------------
when CDCT_S125 =>
          r_mul<=C502*rf(5);
          rf(15)<=r_mul;
          State<=CDCT_S126;

-----------------------------------------------------------
--        State  CDCT_S126
-----------------------------------------------------------
when CDCT_S126 =>
          r_alu<=r_mul+rf(15);
          r_mul<=C426*rf(4);
          State<=CDCT_S127;

-----------------------------------------------------------
--        State  CDCT_S127
-----------------------------------------------------------
when CDCT_S127 =>
          r_sh<=JRS(r_alu,C9);
          r_mul<=C284*rf(3);
          rf(14)<=r_mul;
          State<=CDCT_S128;

-----------------------------------------------------------
--        State  CDCT_S128
-----------------------------------------------------------
when CDCT_S128 =>
          rf(15)<=r_sh;
          r_alu<=rf(14)-r_mul;
          State<=CDCT_S129;

-----------------------------------------------------------
--        State  CDCT_S129
-----------------------------------------------------------
when CDCT_S129 =>
          r_alu<=rf(0)+C1;
          r_sh<=JRS(r_alu,C9);
          State<=CDCT_S130;


-----------------------------------------------------------
--        State  CDCT_S130
-----------------------------------------------------------
when CDCT_S130 =>
          output(r_alu) := rf(15);
          r_alu<=rf(0)+C3;
          State<=CDCT_S131;

-----------------------------------------------------------
--        State  CDCT_S131
-----------------------------------------------------------
when CDCT_S131 =>
          output(r_alu):=r_sh;
```

```
            r_mul<=C426*rf(3);
            State<=CDCT_S132;
------------------------------------------------------------
------------------------------------------------------------
------------------------------------------------------------
------------------------------------------------------------
--          State  CDCT_S132
------------------------------------------------------------
when CDCT_S132 =>
            r_mul<=C284*rf(4);
            rf(15)<=r_mul;
            State<=CDCT_S133;

------------------------------------------------------------
--          State  CDCT_S133
------------------------------------------------------------
when CDCT_S133 =>
            r_alu<=r_mul+rf(15);
            r_mul<=C100*rf(5);
            State<=CDCT_S134;

------------------------------------------------------------
--          State  CDCT_S134
------------------------------------------------------------
when CDCT_S134 =>
            r_sh<=JRS(r_alu,C9);
            r_mul<=C502*rf(2);
            rf(14)<=r_mul;
            State<=CDCT_S135;

------------------------------------------------------------
--          State  CDCT_S135
------------------------------------------------------------
when CDCT_S135 =>
            rf(15)<=r_sh;
            r_alu<=rf(14)-r_mul;
            State<=CDCT_S136;

------------------------------------------------------------
--          State  CDCT_S136
------------------------------------------------------------
when CDCT_S136 =>
            r_alu<=rf(0)+C5;
            r_sh<=JRS(r_alu,9);
            State<=CDCT_S137;


------------------------------------------------------------
--          State  CDCT_S137
------------------------------------------------------------
when CDCT_S137 =>
            output(r_alu) := rf(15);
            r_alu<=rf(0)+C7;
            State<=CDCT_S138;

------------------------------------------------------------
--          State  CDCT_S138
------------------------------------------------------------
when CDCT_S138 =>
            output(r_alu):=r_sh;
            i<=i+1;
            State<=CDCT_S73;

------------------------------------------------------------
--          State  CDCT_S139
------------------------------------------------------------
```

```
when CDCT_S139 =>
        i<=C0;
        State<=CDCT_S140;

---------------------------------------------
--      State  CDCT_S140
---------------------------------------------
when CDCT_S140 =>
        if(i<64) then
                State<=CDCT_S141;
        else
                State<=BD_S0;
        end if;

---------------------------------------------
--      State  CDCT_S141
---------------------------------------------
when CDCT_S141 =>
        r_ram1<=output(i);
        State<=CDCT_S142;


---------------------------------------------
--      State  CDCT_S142
---------------------------------------------
when CDCT_S142 =>
        if(r_ram1<0) then
                State<=CDCT_S146;
        else
                State<=CDCT_S147;
        end if;

---------------------------------------------
--      State  CDCT_S146
---------------------------------------------
when CDCT_S146 =>
        r_alu<=r_ram1-C4;
        State<=CDCT_S143;

---------------------------------------------
--      State  CDCT_S147
---------------------------------------------
when CDCT_S147 =>
        r_alu<=r_ram1+C4;
        State<=CDCT_S143;

---------------------------------------------

--      State  CDCT_S143
---------------------------------------------
when CDCT_S143 =>
        r_div<=r_alu/C8;
        State<=CDCT_S144;

---------------------------------------------
--      State  CDCT_S144
---------------------------------------------
when CDCT_S144 =>
        output(i):=r_div;
        i<=i+C1;
        State<=CDCT_S140;

---------------------------------------------
--      State  BD_S0
---------------------------------------------
when BD_S0 =>
```

```
                i<=0;
                State<= BD_S1;

        ------------------------------------------------
        --      State  BD_S1
        ------------------------------------------------
        when BD_S1 =>
                if (i < 64) then
                        State <= BD_S2;
                else
                        State <= CDCT_S145;
                end if;

        ------------------------------------------------
        --      State  BD_S2
        ------------------------------------------------
        when BD_S2 =>
                r_ram1 <= output(i);
                State <= BD_S3;

        ------------------------------------------------
        --      State  BD_S3
        ------------------------------------------------
        when BD_S3 =>
                if (r_ram1 < -1023) then State <= BD_S4;
                else State <= BD_S5;
                end if;

        ------------------------------------------------
        --      State  BD_S4
        ------------------------------------------------
        when BD_S4 =>
                output(i) := -1023;
                i<=i+1;
                State <= BD_S1;

        ------------------------------------------------
        --      State  BD_S5
        ------------------------------------------------
        when BD_S5 =>
                if (r_ram1 > 1023) then State <= BD_S6;
                else State <= BD_S7;
                end if;

        ------------------------------------------------
        --      State  BD_S6
        ------------------------------------------------
        when BD_S6 =>
                output(i) := 1023;
                State <= BD_S7;

        ------------------------------------------------
        --      State  BD_S7
        ------------------------------------------------
        when BD_S7 =>
                i<=i+1;
                State <= BD_S1;

        ------------------------------------------------
        --      State  CDCT_S145
        ------------------------------------------------
        when CDCT_S145 =>
                i <= 0;
                State <= QT_S1;

        ------------------------------------------------
```

```
--          State  QT_S1
------------------------------------------------
when QT_S1 =>
        if (i < 64) then
                State <= QT_S2;
        else
                State <= QT_S8;
        end if;

------------------------------------------------
--          State  QT_S2
------------------------------------------------
when QT_S2 =>
        r_ram1 <= output(i);
        r_ram2 <= QuantizationMatrix(i);
        State <= QT_S3;

------------------------------------------------
--          State  QT_S3
------------------------------------------------
when QT_S3 =>
        r_div <= r_ram2/2;
        if (r_ram1 > 0) then
                State <= QT_S4;
        else
                State <= QT_S5;
        end if;

------------------------------------------------
--          State  QT_S4
------------------------------------------------
when QT_S4 =>
        r_alu <= r_ram1 + r_div;
        State <= QT_S6;

------------------------------------------------
--          State  QT_S5
------------------------------------------------
when QT_S5 =>
        r_alu <= r_ram1 - r_div;
        State <= QT_S6;

------------------------------------------------
--          State  QT_S6
------------------------------------------------
when QT_S6 =>
        r_div <= r_alu / r_ram2;
        State <=  QT_S7;

------------------------------------------------
--          State  QT_S7
------------------------------------------------
when QT_S7 =>
        output(i) := r_div;
        i <= i + 1;
        State <=  QT_S1;

------------------------------------------------
--          State  QT_S8
------------------------------------------------
when QT_S8 =>
        State <= ZM_S0;

------------------------------------------------
--          State  ZM_S0
------------------------------------------------
```

```
when ZM_S0 =>
        State <= ZM_S1;
        i <= 0;
        r_ram1 <= C64;

------------------------------------------
--      State  ZM_S1
------------------------------------------
when ZM_S1 =>
        if (i < r_ram1) then
                State <= ZM_S2;
        else
                State <= ZM_S4;
        end if;

------------------------------------------
--      State  ZM_S2
------------------------------------------
when ZM_S2 =>
        r_ram1 <= ZigzagIndex(i);
        r_ram2 <= output(i);
        State <= ZM_S3;

------------------------------------------
--      State  ZM_S3
------------------------------------------
when ZM_S3 =>
        input(r_ram1) := r_ram2;
        i <= i + 1;
        r_ram1 <= C64;
        State <= ZM_S1;

------------------------------------------
--      State  ZM_S4
------------------------------------------
when ZM_S4 =>
        State <= ED_S0;

-- ZigzagMatrix End
-- EncodeDC  Begin

------------------------------------------
--      State  ED_S0
------------------------------------------
when ED_S0 =>
        r_ram1 <= input(0);
        r_ram2 <= LastDC;
        State <= ED_S1;

------------------------------------------
--      State  ED_S1
------------------------------------------
when ED_S1 =>
        r_alu <= r_ram1 - r_ram2;
        LastDC := input(0);
        State <= ED_S2;

------------------------------------------
--      State  ED_S2
------------------------------------------
when ED_S2 =>
        rf(1) <= r_alu;
        r_alu <= abs(r_alu);
        r_ram1 <= C256;
        State <= ED_S3;
```

```
--------------------------------------------------
--        State ED_S3
--------------------------------------------------
when ED_S3 =>
        rf(2) <= r_alu;
        if(r_alu < r_ram1) then
                State <= ED_S4;
        else
                r_ram1 <= C8;
                State <= ED_S6;
        end if;

--------------------------------------------------
--        State ED_S4
--------------------------------------------------
when ED_S4 =>
        r_ram1 <= csize(rf(2));
        State <= ED_S5;

--------------------------------------------------
--        State ED_S5
--------------------------------------------------
when ED_S5 =>
        rf(0) <= r_ram1;
        State <= ED_S10;

--------------------------------------------------
--        State ED_S6
--------------------------------------------------
when ED_S6 =>
        r_sh <= JRS(rf(2), r_ram1);
        State <= ED_S7;

--------------------------------------------------
--        State ED_S7
--------------------------------------------------
when ED_S7 =>
        rf(2) <= r_sh;
        r_ram2 <= csize(r_sh);
        State <= ED_S8;

--------------------------------------------------
--        State ED_S8
--------------------------------------------------
when ED_S8 =>
        r_alu <= r_ram2 + r_ram1;
        State <= ED_S9;

--------------------------------------------------
--        State ED_S9
--------------------------------------------------
when ED_S9 =>
        rf(0) <= r_alu;
        State <= ED_S10;

--------------------------------------------------
--        State ED_S10
--------------------------------------------------
when ED_S10 =>
        rf(24) <= i;
        rf(23) <= rf(0);
        rf(22) <= 0;
        EH_Return_State <= ED_S11;
        State <= EH_S_0;

--------------------------------------------------
```

```
--      State ED_S11
------------------------------------------------------
when ED_S11 =>
        i <= rf(24);
        if(rf(1) < C0) then
                r_alu <= rf(1) - 1;
                State <= ED_S12;
        else
                State <= ED_S13;
        end if;

------------------------------------------------------
--      State ED_S12
------------------------------------------------------
when ED_S12 =>
        rf(1) <= r_alu;
        State <= ED_S13;

------------------------------------------------------
--      State ED_S13
------------------------------------------------------
when ED_S13 =>
        rf(4) <= C0;
        rf(21) <= rf(0);
        rf(20) <= rf(1);
        WB_Return_State <= EA_S0;
        State <= WB_S_0;

-- EncodeDC End
-- EncodeAC Begin

------------------------------------------------------
--      State EA_S0
------------------------------------------------------
when EA_S0 =>
        i <= 0;
        State <= EA_S1;

------------------------------------------------------
--      State EA_S1
------------------------------------------------------
when EA_S1 =>
        i <= i + 1;
        r_ram1 <= C64;
        State <= EA_S2;

------------------------------------------------------
--      State EA_S2
------------------------------------------------------
when EA_S2 =>
        if(i < r_ram1) then
                State <= EA_S3;
        else
                State <= EA_S25;
        end if;

------------------------------------------------------
--      State EA_S3
------------------------------------------------------
when EA_S3 =>
        r_ram1 <= input(i);
        State <= EA_S4;

------------------------------------------------------
--      State EA_S4
------------------------------------------------------
```

```
when EA_S4 =>
        rf(0) <= r_ram1;
        r_alu <= abs(r_ram1);
        r_ram2 <= C256;
        State <= EA_S5;

------------------------------------------------
--      State EA_S5
------------------------------------------------
when EA_S5 =>
        rf(2) <= r_alu;
        if(r_alu < r_ram2) then
                State <= EA_S6;
        else
                r_ram1 <= C8;
                State <= EA_S8;
        end if;

------------------------------------------------
--      State EA_S6
------------------------------------------------
when EA_S6 =>
        r_ram1 <= csize(rf(2));
        State <= EA_S7;

------------------------------------------------
--      State EA_S7
------------------------------------------------
when EA_S7 =>
        rf(1) <= r_ram1;
        r_ram1 <= C0;
        State <= EA_S12;

------------------------------------------------
--      State EA_S8
------------------------------------------------
when EA_S8 =>
        r_sh <= JRS(rf(2), r_ram1);
        State <= EA_S9;

------------------------------------------------
--      State EA_S9
------------------------------------------------
when EA_S9 =>
        rf(2) <= r_sh;
        r_ram2 <= csize(r_sh);
        State <= EA_S10;

------------------------------------------------
--      State EA_S10
------------------------------------------------
when EA_S10 =>
        r_alu <= r_ram2 + r_ram1;
        State <= EA_S11;

------------------------------------------------
--      State EA_S11
------------------------------------------------
when EA_S11 =>
        rf(1) <= r_alu;
        r_ram1 <= C0;
        State <= EA_S12;

------------------------------------------------
--      State EA_S12
------------------------------------------------
```

```
when EA_S12 =>
        if (rf(0) = r_ram1) then
                r_ram2 <= C63;
                State <= EA_S13;
        else
                r_ram1 <= C15;
                r_ram2 <= C16;
                State <= EA_S16;
        end if;

--------------------------------------------
--      State EA_S13
--------------------------------------------
when EA_S13 =>
        if (i = r_ram2) then

                rf(24) <= i;
                rf(23) <= 0;
                rf(22) <= 1;
                EH_Return_State <= EA_S13a;
                State <= EH_S_0;

        else
                State <= EA_S14;
        end if;

--------------------------------------------
--      State EA_S13a
--------------------------------------------
when EA_S13a =>
        i <= rf(24);
        State <= EA_S1;

--------------------------------------------
--      State EA_S14
--------------------------------------------
when EA_S14 =>
        r_alu <= rf(4) + 1;
        State <= EA_S15;

--------------------------------------------
--      State EA_S15
--------------------------------------------
when EA_S15 =>
        rf(4) <= r_alu;
        State <= EA_S1;


--------------------------------------------
--      State EA_S16
--------------------------------------------
when EA_S16 =>
        r_ram1 <= C15;
        r_ram2 <= C16;
        State <= EA_S17;

--------------------------------------------
--      State EA_S17
--------------------------------------------
when EA_S17 =>
        if (rf(4) > r_ram1) then
                r_alu <= rf(4) - r_ram2;
                r_ram1 <= C240;
                State <= EA_S18;
        else
                r_ram1 <= C0;
```

```
                r_ram2 <= C16;
                State <= EA_S19;
            end if;

    --------------------------------------------------
    --      State EA_S18
    --------------------------------------------------
    when EA_S18 =>
            rf(24) <= i;
            rf(4) <= r_alu;
            rf(23) <= r_ram1;
            rf(22) <= 1;
            EH_Return_State <= EA_S18a;
            State <= EH_S_0;

    --------------------------------------------------
    --      State EA_S18a
    --------------------------------------------------
    when EA_S18a =>
            i <= rf(24);
            State <= EA_S16;

    --------------------------------------------------
    --      State EA_S19
    --------------------------------------------------
    when EA_S19 =>
            r_mul <= r_ram2 * rf(4);
            State <= EA_S20;

    --------------------------------------------------
    --      State EA_S20
    --------------------------------------------------
    when EA_S20 =>
            r_alu <= r_mul + rf(1);
            State <= EA_S21;

    --------------------------------------------------
    --      State EA_S21
    --------------------------------------------------
    when EA_S21 =>
            rf(24) <= i;
            rf(3) <= r_alu;
            rf(4) <= r_ram1;
            rf(23) <= r_alu;
            rf(22) <= 1;
            EH_Return_State <= EA_S22;
            State <= EH_S_0;

    --------------------------------------------------
    --      State EA_S22
    --------------------------------------------------
    when EA_S22 =>
            i <= rf(24);
            r_ram1 <= C0;
            State <= EA_S23;

    --------------------------------------------------
    --      State EA_S23
    --------------------------------------------------
    when EA_S23 =>
            if (rf(0) < r_ram1) then
                r_alu <= rf(0) - 1;
                State <= EA_S24;
            else
                rf(24) <= i;
```

```
                    rf(21) <= rf(1);
                    rf(20) <= rf(0);
                    WB_Return_State <= EA_S13a;
                    State <= WB_S_0;

            end if;

        ---------------------------------------------
        --      State EA_S24
        ---------------------------------------------
        when EA_S24 =>
                rf(24) <= i;
                rf(21) <= rf(1);
                rf(20) <= r_alu;
                WB_Return_State <= EA_S13a;
                State <= WB_S_0;

        ---------------------------------------------
        --      State EA_S25
        ---------------------------------------------
        when EA_S25 =>
                State <= PDM_Return_State;

        -- EncodeAC End

    --===================================================
    --              EncodeHuffman States
    --===================================================

        ---------------------------------------------
        --      State EH_S_0
        ---------------------------------------------
        when EH_S_0 =>
          i <= C0; -- prep to save registers
          State <= EH_S_REG_SAVE;

        ---------------------------------------------
        --      State EH_S_REG_SAVE
        ---------------------------------------------
        when EH_S_REG_SAVE => -- store rf(21)
          encodehuffman_reg_save(i) := rf(21);
          State <= EH_S_REG_SAVE2;

        ---------------------------------------------
        --      State EH_S_REG_SAVE2
        ---------------------------------------------
        when EH_S_REG_SAVE2 =>
          i <= i + C1;
          State <= EH_S_REG_SAVE3;

        ---------------------------------------------
        --      State EH_S_REG_SAVE3
        ---------------------------------------------
        when EH_S_REG_SAVE3 => -- store rf(20)
          encodehuffman_reg_save(i) := rf(20);
          State <= EH_S_REG_SAVE4;

        ---------------------------------------------
        --      State EH_S_REG_SAVE4
        ---------------------------------------------
        when EH_S_REG_SAVE4 => -- prep to load with index [value]
          i <= rf(23);
          State <= EH_S_IF_AC_DC;

        ---------------------------------------------
        --      State EH_S_IF_AC_DC
```

```
---------------------------------------------------------------
when EH_S_IF_AC_DC => -- using AC or DC Huff table?
  if (C0 = rf(22)) then
    r_ram1 <= DC_EHUFF_SI(i); -- to rf(21)
    State <= EH_S_DC_IF_EHUFF;
  else
    r_ram1 <= AC_EHUFF_SI(i); -- to rf(21)
    State <= EH_S_AC_IF_EHUFF;
  end if;

---------------------------------------------------------------
--        State EH_S_DC_IF_EHUFF
---------------------------------------------------------------
when EH_S_DC_IF_EHUFF => -- if (Ehuff->ehufsi[value]) {
  rf(21) <= r_ram1;
  if (C0 = r_ram1) then
    State <= EH_S_RETURN;
  else
    State <= EH_S_DC_IF_EHUFF2;
  end if;

---------------------------------------------------------------
--        State EH_S_DC_IF_EHUFF2
---------------------------------------------------------------
when EH_S_DC_IF_EHUFF2 => -- #2 if (Ehuff->ehufsi[value]) {
  r_ram1 <= DC_EHUFF_CO(i); -- to rf(20)
  State <= EH_S_CALL; -- call WriteBits

---------------------------------------------------------------
--        State EH_S_AC_IF_EHUFF
---------------------------------------------------------------
when EH_S_AC_IF_EHUFF => -- if (Ehuff->ehufsi[value]) {
  rf(21) <= r_ram1;
  if (C0 = r_ram1) then
    State <= EH_S_RETURN;
  else
    State <= EH_S_AC_IF_EHUFF2;
  end if;

---------------------------------------------------------------
--        State EH_S_AC_IF_EHUFF2
---------------------------------------------------------------
when EH_S_AC_IF_EHUFF2 => -- #2 if (Ehuff->ehufsi[value]) {
  r_ram1 <= AC_EHUFF_CO(i); -- to rf(20)
  State <= EH_S_CALL; -- call WriteBits

---------------------------------------------------------------
--        State EH_S_CALL
---------------------------------------------------------------
when EH_S_CALL => -- call WriteBits
  rf(20) <= r_ram1;
  WB_Return_State <= EH_S_RETURN;
  State <= WB_S_0;

---------------------------------------------------------------
--        State EH_S_RETURN
---------------------------------------------------------------
when EH_S_RETURN => -- WriteBits returns here
  i <= C0; -- prep to save registers
  State <= EH_S_REG_RESTORE;

---------------------------------------------------------------
--        State EH_S_REG_RESTORE
---------------------------------------------------------------
when EH_S_REG_RESTORE => -- restore rf(21)
  r_ram1 <= encodehuffman_reg_save(i); -- to rf(21)
```

```
          State <= EH_S_REG_RESTORE2;

        -----------------------------------------------
        --        State EH_S_REG_RESTORE2
        -----------------------------------------------
        when EH_S_REG_RESTORE2 =>
          i <= i + C1;
          rf(21) <= r_ram1;
          State <= EH_S_REG_RESTORE3;

        -----------------------------------------------
        --        State EH_S_REG_RESTORE3
        -----------------------------------------------
        when EH_S_REG_RESTORE3 => -- restore rf(20)
          r_ram1 <= encodehuffman_reg_save(i); -- to rf(20)
          State <= EH_S_END;

        -----------------------------------------------
        --        State EH_S_END
        -----------------------------------------------
        when EH_S_END =>
          rf(20) <= r_ram1;
          State <= EH_Return_State; -- Return to Caller

  --=================================================
  --              WriteBits States
  --=================================================

        -----------------------------------------------
        --        State WB_S_0
        -----------------------------------------------
        when WB_S_0 => -- prep to save registers
          i <= C2;
          State <= WB_S_REG_SAVE;

        -----------------------------------------------
        --        State WB_S_REG_SAVE
        -----------------------------------------------
        when WB_S_REG_SAVE =>  -- store rf_n
          writebits_static(i) := rf(21);
          State <= WB_S_REG_SAVE2;

        -----------------------------------------------
        --        State WB_S_REG_SAVE2
        -----------------------------------------------
        when WB_S_REG_SAVE2 =>
          i <= i + C1;
          State <= WB_S_REG_SAVE3;

        -----------------------------------------------
        --        State WB_S_REG_SAVE3
        -----------------------------------------------
        when WB_S_REG_SAVE3 =>  -- store rf_code
          writebits_static(i) := rf(20);
          State <= WB_S_REG_SAVE4;

        -----------------------------------------------
        --        State WB_S_REG_SAVE4
        -----------------------------------------------
        when WB_S_REG_SAVE4 =>
          i <= i + C1;
          State <= WB_S_REG_SAVE5;

        -----------------------------------------------
        --        State WB_S_REG_SAVE5
        -----------------------------------------------
```

```
when WB_S_REG_SAVE5 =>  -- store rf(3)
                        -- rf(3) to hold write_byte
  writebits_static(i) := rf(3);
  State <= WB_S_REG_SAVE6;

----------------------------------------------
--       State WB_S_REG_SAVE6
----------------------------------------------
when WB_S_REG_SAVE6 =>
  i <= i + C1;
  State <= WB_S_REG_SAVE7;

----------------------------------------------
--       State WB_S_REG_SAVE7
----------------------------------------------
when WB_S_REG_SAVE7 =>  -- store rf(4)
                        -- rf(4) to hold left_bits
  writebits_static(i) := rf(4);
  State <= WB_S_REG_SAVE8;

----------------------------------------------
--       State WB_S_REG_SAVE8
----------------------------------------------
when WB_S_REG_SAVE8 =>
  i <= i + C1;
  State <= WB_S_REG_SAVE9;

----------------------------------------------
--       State WB_S_REG_SAVE9
----------------------------------------------
when WB_S_REG_SAVE9 =>  -- store rf(5)
                        -- rf(5) to hold temp
  writebits_static(i) := rf(5);
  State <= WB_S_INIT;

----------------------------------------------
--       State WB_S_INIT
----------------------------------------------
when WB_S_INIT => -- prep to load write_byte
  i <= C0;
  State <= WB_S_INIT2;

----------------------------------------------
--       State WB_S_INIT2
----------------------------------------------
when WB_S_INIT2 => -- load write_byte
  r_ram1 <= writebits_static(i); -- to rf(3)
  State <= WB_S_INIT3;

----------------------------------------------
--       State WB_S_INIT3
----------------------------------------------
when WB_S_INIT3 => -- prep to load left_bits
  i <= i + C1;
  rf(3) <= r_ram1;
  State <= WB_S_INIT4;

----------------------------------------------
--       State WB_S_INIT4
----------------------------------------------
when WB_S_INIT4 => -- load left_bits
  r_ram1 <= writebits_static(i); -- to rf(4)
  State <= WB_S_IF_N_LT0;

----------------------------------------------
--       State WB_S_IF_N_LT0
```

```
------------------------------------------------------------
when WB_S_IF_N_LT0 => -- if (n < 0) {
  rf(4) <= r_ram1;
  if (rf(21) < C0) then
    State <= WB_S_IF_LB_LT;
  else
    State <= WB_S_N_GE0;
  end if;

------------------------------------------------------------
--        State WB_S_IF_LB_LT
------------------------------------------------------------
when WB_S_IF_LB_LT => -- if (left_bits < 8) {
  if (rf(4) < C8) then
    State <= WB_S_LB_LT8;
  else
    State <= WB_S_ELSE_LB_LT;
  end if;

------------------------------------------------------------
--        State WB_S_LB_LT8
------------------------------------------------------------
when WB_S_LB_LT8 => -- n = left_bits;
  rf(21) <= rf(4);
  State <= WB_S_LT8_SEND;

------------------------------------------------------------
--        State WB_S_LT8_SEND
------------------------------------------------------------
when WB_S_LT8_SEND => -- Data_Ch.send(write_byte);
  OutData <= CONV_UNSIGNED(rf(3), 8);
  OutDataReady <= '1';
  State <= WB_S_LT8_IF_WB;

------------------------------------------------------------
--        State WB_S_LT8_IF_WB
------------------------------------------------------------
when WB_S_LT8_IF_WB => -- if (write_byte == 0xff) {
  OutDataReady <= '0';
  if (rf(3) = C0XFF) then
    State <= WB_S_LT8_WB_FF;
  else
    State <= WB_S_LT8_WB;
  end if;

------------------------------------------------------------
--        State WB_S_LT8_WB_FF
------------------------------------------------------------
when WB_S_LT8_WB_FF => -- Data_Ch.send(0); }
  OutData <= CONV_UNSIGNED(C0, 8);
  OutDataReady <= '1';
  State <= WB_S_LT8_WB;

------------------------------------------------------------
--        State WB_S_LT8_WB
------------------------------------------------------------
when WB_S_LT8_WB => -- write_byte = 0;
  OutDataReady <= '0';
  rf(3) <= C0;
  State <= WB_S_LT8_WB_LB;

------------------------------------------------------------
--        State WB_S_LT8_WB_LB
------------------------------------------------------------
when WB_S_LT8_WB_LB => -- left_bits = 8;
  rf(4) <= C8;
```

```
    State <= WB_S_RETURN;

  ----------------------------------------------------------------
  --       State  WB_S_ELSE_LB_LT
  ----------------------------------------------------------------
  when WB_S_ELSE_LB_LT => -- else { n = 0; }
    rf(21) <= C0;
    i <= C0; -- prep to store write_byte
    State <= WB_S_RETURN;

  ----------------------------------------------------------------
  --       State  WB_S_N_GE0
  ----------------------------------------------------------------
  when WB_S_N_GE0 => -- #1 code &= lmask[n];
    i <= rf(21);
    State <= WB_S_NGE0_CO2;

  ----------------------------------------------------------------
  --       State  WB_S_NGE0_CO2
  ----------------------------------------------------------------
  when WB_S_NGE0_CO2 => -- #2 code &= lmask[n];
                        -- p = n - left_bits ;
    r_ram1 <= lmask(i);
    r_alu <= rf(21) - rf(4); -- to rf(5)
    State <= WB_S_NGE0_CO3;

  ----------------------------------------------------------------
  --       State  WB_S_NGE0_CO3
  ----------------------------------------------------------------
  when WB_S_NGE0_CO3 => -- #3 code &= lmask[n];
    rf(5) <= r_alu;
    State <= WB_S_NGE0_CO4;

  ----------------------------------------------------------------
  --       State  WB_S_NGE0_CO4
  ----------------------------------------------------------------
  when WB_S_NGE0_CO4 => -- #4 code &= lmask[n];
    -- r_alu <= rf_code & r_ram1; -- to rf_code
    r_alu <= JAND( rf(20), r_ram1 );
    State <= WB_S_IF_N_EQ;

  ----------------------------------------------------------------
  --       State  WB_S_IF_N_EQ
  ----------------------------------------------------------------
  when WB_S_IF_N_EQ => -- if (n == left_bits) {
    rf(20) <= r_alu;
    if (rf(21) = rf(4)) then
      State <= WB_S_N_EQ_LB;
    else
      State <= WB_S_ELIF_N_GT;
    end if;

  ----------------------------------------------------------------
  --       State  WB_S_N_EQ_LB
  ----------------------------------------------------------------
  when WB_S_N_EQ_LB => -- write_byte |= code;
    -- r_alu <= rf(3) | rf_code; -- to rf(3)
    r_alu <= JOR( rf(3), rf(20) );
    State <= WB_S_EQLB_SEND;

  ----------------------------------------------------------------
  --       State  WB_S_EQLB_SEND
  ----------------------------------------------------------------
  when WB_S_EQLB_SEND => -- Data_Ch.send(write_byte);
    rf(3) <= r_alu;
    OutData <= CONV_UNSIGNED(r_alu, 8);
```

```
    OutDataReady <= '1';
    State <= WB_S_EQLB_IF_WB;

  ----------------------------------------------
  --      State  WB_S_EQLB_IF_WB
  ----------------------------------------------
  when WB_S_EQLB_IF_WB => -- if (write_byte == 0xff) {
    OutDataReady <= '0';
    if (rf(3) = C0XFF) then
      State <= WB_S_EQLB_WB_FF;
    else
      State <= WB_S_EQLB_WB;
    end if;

  ----------------------------------------------
  --      State  WB_S_EQLB_WB_FF
  ----------------------------------------------
  when WB_S_EQLB_WB_FF => -- Data_Ch.send(0); }
    OutData <= CONV_UNSIGNED(C0, 8);
    OutDataReady <= '1';
    State <= WB_S_EQLB_WB;

  ----------------------------------------------
  --      State  WB_S_EQLB_WB
  ----------------------------------------------
  when WB_S_EQLB_WB => -- write_byte = 0;
    OutDataReady <= '0';
    rf(3) <= C0;
    State <= WB_S_EQLB_WB_LB;

  ----------------------------------------------
  --      State  WB_S_EQLB_WB_LB
  ----------------------------------------------
  when WB_S_EQLB_WB_LB => -- left_bits = 8; }
    rf(4) <= C8;
    i <= C0; -- prep to store write_byte
    State <= WB_S_RETURN;

  ----------------------------------------------
  --      State  WB_S_ELIF_N_GT
  ----------------------------------------------
  when WB_S_ELIF_N_GT => -- else if (n > left_bits) {
    if (rf(21) > rf(4)) then
      State <= WB_S_N_GT_LB;
    else
      State <= WB_S_ELSE_N_LT;
    end if;

  ----------------------------------------------
  --      State  WB_S_N_GT_LB
  ----------------------------------------------
  when WB_S_N_GT_LB => -- #1 write_byte |= (code >> p);
    -- r_sh <= rf_code >> rf(5);
    r_sh <= JRS( rf(20), rf(5) );
    State <= WB_S_GTLB_WB2;

  ----------------------------------------------
  --      State  WB_S_GTLB_WB2
  ----------------------------------------------
  when WB_S_GTLB_WB2 => -- #2 write_byte |= (code >> p);
    -- r_alu <= rf_write_byte | r_sh;
    r_alu <= JOR( rf(3), r_sh ); -- to rf(3)
    State <= WB_S_GTLB_SEND;

  ----------------------------------------------
  --      State  WB_S_GTLB_SEND
```

```
--------------------------------------------
when WB_S_GTLB_SEND => -- Data_Ch.send(write_byte);
  rf(3) <= r_alu;
  OutData <= CONV_UNSIGNED(r_alu, 8);
  OutDataReady <= '1';
  State <= WB_S_GTLB_IF_WB;

--------------------------------------------
--        State WB_S_GTLB_IF_WB
--------------------------------------------
when WB_S_GTLB_IF_WB => -- if (write_byte == 0xff) {
  OutDataReady <= '0';
  if (rf(3) = C0XFF) then
    State <= WB_S_GTLB_WB_FF;
  else
    State <= WB_S_IF_P_GT8;
  end if;

--------------------------------------------
--        State WB_S_GTLB_WB_FF
--------------------------------------------
when WB_S_GTLB_WB_FF => -- Data_Ch.send(0); }
  OutData <= CONV_UNSIGNED(C0, 8);
  OutDataReady <= '1';
  State <= WB_S_IF_P_GT8;

--------------------------------------------
--        State WB_S_IF_P_GT8
--------------------------------------------
when WB_S_IF_P_GT8 => -- if (p > 8) {
  OutDataReady <= '0';
  if (rf(5) > C8) then
    State <= WB_S_P_GT8;
  else
    State <= WB_S_P_LE8;
  end if;

--------------------------------------------
--        State WB_S_P_GT8
--------------------------------------------
when WB_S_P_GT8 => -- #1 write_byte = (0xff & (code >> (p - 8)));
  r_alu <= rf(5) - C8; -- to rf(5)
  State <= WB_S_GT8_WB2;

--------------------------------------------
--        State WB_S_GT8_WB2
--------------------------------------------
when WB_S_GT8_WB2 => -- #2 write_byte = (0xff & (code >> (p - 8)));
  -- r_sh <= rf_code >> r_alu;
  r_sh <= JRS( rf(20), r_alu );
  State <= WB_S_GT8_WB3;

--------------------------------------------
--        State WB_S_GT8_WB3
--------------------------------------------
when WB_S_GT8_WB3 => -- #3 write_byte = (0xff & (code >> (p - 8)));
  -- r_alu <= C0XFF & r_sh; -- to rf(3)
  r_alu <= JAND( C0XFF, r_sh );
  State <= WB_S_GT8_SEND;

--------------------------------------------
--        State WB_S_GT8_SEND
--------------------------------------------
when WB_S_GT8_SEND => -- Data_Ch.send(write_byte);
  rf(3) <= r_alu;
  OutData <= CONV_UNSIGNED(r_alu, 8);
```

```
OutDataReady <= '1';
State <= WB_S_GT8_IF_WB;

-------------------------------------------------
--        State  WB_S_GT8_IF_WB
-------------------------------------------------
when WB_S_GT8_IF_WB => -- if (write_byte == 0xff) {
  OutDataReady <= '0';
  if (rf(3) = C0XFF) then
    State <= WB_S_GT8_WB_FF;
  else
    State <= WB_S_GT8_P;
  end if;

-------------------------------------------------
--        State  WB_S_GT8_WB_FF
-------------------------------------------------
when WB_S_GT8_WB_FF => -- Data_Ch.send(0); }
  OutData <= CONV_UNSIGNED(C0, 8);
  OutDataReady <= '1';
  State <= WB_S_GT8_P;

-------------------------------------------------
--        State  WB_S_GT8_P
-------------------------------------------------
when WB_S_GT8_P => -- p -= 8; }
  OutDataReady <= '0';
  rf(5) <= rf(5) - C8;
  State <= WB_S_P_LE8;

-------------------------------------------------
--        State  WB_S_P_LE8
-------------------------------------------------
when WB_S_P_LE8 => -- #1 write_byte = (code & lmask[p]) << (8 - p);
  i <= rf(5);
  r_alu <= C8 - rf(5); -- to rf(4)
  State <= WB_S_PLE8_WB2;

-------------------------------------------------
--        State  WB_S_PLE8_WB2
-------------------------------------------------
when WB_S_PLE8_WB2 => -- #2 write_byte = (code & lmask[p]) << (8 - p);
                      -- left_bits = 8 - p; }
  rf(4) <= r_alu;
  r_ram1 <= lmask(i);
  State <= WB_S_PLE8_WB3;

-------------------------------------------------
--        State  WB_S_PLE8_WB3
-------------------------------------------------
when WB_S_PLE8_WB3 => -- #3 write_byte = (code & lmask[p]) << (8 - p);
  -- r_alu <= rf_code & r_ram1;
  r_alu <= JAND( rf(20), r_ram1 );
  State <= WB_S_PLE8_WB4;

-------------------------------------------------
--        State  WB_S_PLE8_WB4
-------------------------------------------------
when WB_S_PLE8_WB4 => -- #4 write_byte = (code & lmask[p]) << (8 - p);
  -- r_sh <= r_alu << rf(4); -- to rf(3)
  r_sh <= JLS( r_alu, rf(4) );
  State <= WB_S_PLE8_WB5;

-------------------------------------------------
--        State  WB_S_PLE8_WB5
-------------------------------------------------
```

```
when WB_S_PLE8_WB5 => -- #5 write_byte = (code & lmask[p]) << (8 - p);
  rf(3) <= r_sh;
  i <= C0; -- prep to store write_byte
  State <= WB_S_RETURN;

----------------------------------------------------------
-- State WB_S_ELSE_N_LT
----------------------------------------------------------
when WB_S_ELSE_N_LT => -- #1 write_byte |= (code << -p);
  r_alu <= C0 - rf(5);
  State <= WB_S_LTLB_WB2;

----------------------------------------------------------
-- State WB_S_LTLB_WB2
----------------------------------------------------------
when WB_S_LTLB_WB2 => -- #2 write_byte |= (code << -p);
  -- r_sh <= rf_code << r_alu;
  r_sh <= JLS( rf(20), r_alu );
  State <= WB_S_LTLB_WB3;

----------------------------------------------------------
-- State WB_S_LTLB_WB3
----------------------------------------------------------
when WB_S_LTLB_WB3 => -- #3 write_byte |= (code << -p);
  -- r_alu <= rf(3) | r_sh; -- to rf(3)
  r_alu <= JOR( rf(3), r_sh );
  State <= WB_S_LTLB_WB4;

----------------------------------------------------------
-- State WB_S_LTLB_WB4
----------------------------------------------------------
when WB_S_LTLB_WB4 => -- #4 write_byte |= (code << -p);
  rf(3) <= r_alu;
  State <= WB_S_LTLB_LB;

----------------------------------------------------------
-- State WB_S_LTLB_LB
----------------------------------------------------------
when WB_S_LTLB_LB => -- left_bits -= n; }
  r_alu <= rf(4) - rf(21); -- to rf(4)
  State <= WB_S_LTLB_LB2;

----------------------------------------------------------
-- State WB_S_LTLB_LB2
----------------------------------------------------------
when WB_S_LTLB_LB2 => -- #2 left_bits -= n; }
  rf(4) <= r_alu;
  i <= C0; -- prep to store write_byte
  State <= WB_S_RETURN;

----------------------------------------------------------
-- State WB_S_RETURN
----------------------------------------------------------
when WB_S_RETURN => -- store write_byte
  -- rf_return <= rf_n; per hongxing, no return needed.
  -- note: 'i <= C0' set by all immediately previous states.
  writebits_static(i) := rf(3);
  State <= WB_S_RETURN2;

----------------------------------------------------------
-- State WB_S_RETURN2
----------------------------------------------------------
when WB_S_RETURN2 =>
  i <= i + C1;
  State <= WB_S_RETURN3;
```

```
-----------------------------------------------
--        State  WB_S_RETURN3
-----------------------------------------------
when WB_S_RETURN3 => -- store left_bits
  writebits_static(i) := rf(4);
  State <= WB_S_RETURN4;

-----------------------------------------------
--        State  WB_S_RETURN4
-----------------------------------------------
when WB_S_RETURN4 =>
  i <= i + C1;
  State <= WB_S_REG_RESTOR;

-----------------------------------------------
--        State  WB_S_REG_RESTOR
-----------------------------------------------
when WB_S_REG_RESTOR => -- restore rf_n;
  r_ram1 <= writebits_static(i); -- to rf_n
  State <= WB_S_REG_RESTOR2;

-----------------------------------------------
--        State  WB_S_REG_RESTOR2
-----------------------------------------------
when WB_S_REG_RESTOR2 => -- #2 restore rf_n;
  rf(21) <= r_ram1;
  i <= i + C1;
  State <= WB_S_REG_RESTOR3;

-----------------------------------------------
--        State  WB_S_REG_RESTOR3
-----------------------------------------------
when WB_S_REG_RESTOR3 => -- restore rf_code;
  r_ram1 <= writebits_static(i); -- to rf_code
  State <= WB_S_REG_RESTOR4;

-----------------------------------------------
--        State  WB_S_REG_RESTOR4
-----------------------------------------------
when WB_S_REG_RESTOR4 => -- #2 restore rf_code;
  rf(20) <= r_ram1;
  i <= i + C1;
  State <= WB_S_REG_RESTOR5;

-----------------------------------------------
--        State  WB_S_REG_RESTOR5
-----------------------------------------------
when WB_S_REG_RESTOR5 => -- restore rf(3);
  r_ram1 <= writebits_static(i); -- to rf(3)
  State <= WB_S_REG_RESTOR6;

-----------------------------------------------
--        State  WB_S_REG_RESTOR6
-----------------------------------------------
when WB_S_REG_RESTOR6 => -- #2 restore rf(3);
  rf(3) <= r_ram1;
  i <= i + C1;
  State <= WB_S_REG_RESTOR7;

-----------------------------------------------
--        State  WB_S_REG_RESTOR7
-----------------------------------------------
when WB_S_REG_RESTOR7 => -- restore rf(4);
  r_ram1 <= writebits_static(i); -- to rf(4)
  State <= WB_S_REG_RESTOR8;
```

```
-------------------------------------------------
--        State  WB_S_REG_RESTOR8
-------------------------------------------------
when WB_S_REG_RESTOR8 => -- #2  restore  rf(4);
   rf(4) <= r_ram1;
   i <= i + C1;
   State <= WB_S_REG_RESTOR9;

-------------------------------------------------
--        State  WB_S_REG_RESTOR9
-------------------------------------------------
when WB_S_REG_RESTOR9 => -- restore  rf(5);
   r_ram1 <= writebits_static(i); -- to  rf(5)
   State <= WB_S_END;

-------------------------------------------------
--        State  WB_S_END
-------------------------------------------------
when WB_S_END =>
   rf(5) <= r_ram1;
   State <= WB_Return_State; -- Return  to  Caller


                end case;
        end if;
        end process;
end sequential;
```

C.1.4 clock.vhd

```
--------------------------------------------------------------------
--        File      :        clock.vhd
--                           vhdl  model  for  a  clock  generator
--------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity clock_gr is
        generic ( cycle : TIME := 12 ns );
        port ( ck : out STD_LOGIC );
end clock_gr;

architecture behavioral of clock_gr is
begin
        process
                variable periodic : STD_LOGIC := '1';
        begin

                periodic := not periodic;

                ck <= periodic;

                wait for ( cycle / 2 );
        end process;
end behavioral;
```

C.1.5 Jpeg.vhd

```
---------------------------------------------------------------------------
--        File              :        Jpeg.vhd
--        Description       :        Test Bench for HandleData module
---------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
```

```vhdl
use IEEE.std_logic_arith.all;
use STD.textio.all;

entity JpegInterface is
        port (
                Reset : in STD_LOGIC;
                InReady : out STD_LOGIC;
                InDataValid : in STD_LOGIC;
                OutDataReady : out STD_LOGIC;
                InData : in unsigned(7 downto 0);
                OutData : out unsigned(7 downto 0)
                );
end JpegInterface;

architecture test of JpegInterface is
        component JpegEncoder
                port (
                        Reset : in STD_LOGIC;
                        Clk : in STD_LOGIC;
                        InReady : out STD_LOGIC;
                        InDataValid : in STD_LOGIC;
                        OutDataReady : out STD_LOGIC;
                        InData : in unsigned(7 downto 0);
                        OutData : out unsigned(7 downto 0)
                        );
        end component;
        for all : JpegEncoder use entity work.JpegEncoder(sequential);

        component cg
                generic (cycle : TIME := 12 ns);
                port (ck : out STD_LOGIC);
        end component;
        for all : cg use entity work.clock_gr( behavioral );

        signal Clk : STD_LOGIC;
begin
        U1 : JpegEncoder port map(Reset, Clk, InReady, InDataValid,
                OutDataReady, InData, OutData);
        U2 : cg port map(Clk);
end test;
```

C.2 Testbench

C.2.1 readpgm.vhd

```vhdl
------------------------------------------------------------------
--      File            :       readpgm.vhd
--      Description     :       Read (ASCII) Pgm image
------------------------------------------------------------------
--
-- 05/25/99  A. Gerstlauer


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use STD.textio.all;


PACKAGE read_pgm IS

   file infile : text is in "infile.pgm";

  subtype pixel is integer range 0 to 255;
```

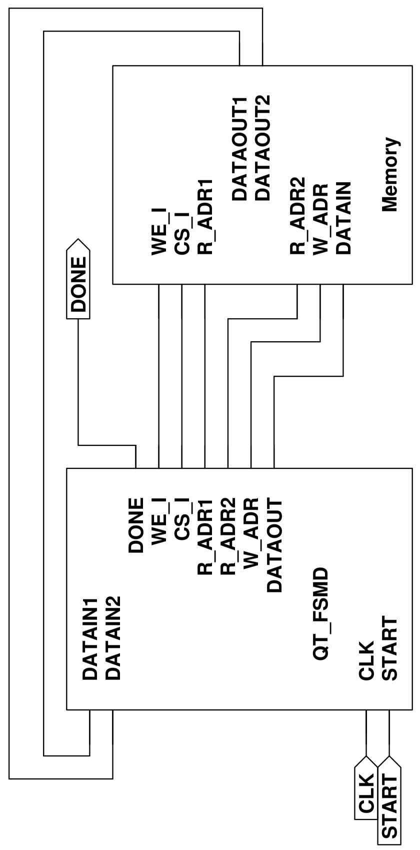
```
    -- Initialize PGM
    procedure init_pgm(pgm: inout line);

    -- Read PGM header
    procedure read_pgm_header(pgm: inout line; width: out integer; height: out integer);

    -- Read a pixel (after header)
    procedure read_pgm_pixel(pgm: inout line; data: out pixel);

END read_pgm;


PACKAGE BODY read_pgm IS


    -- Skip over whitespace and comments
    procedure skip(pgm: inout line) is
      variable ok: boolean;
      variable ch: character;
    begin
      ok := false;
      while not ok loop
        if pgm'length = 0 or pgm(pgm'left) = '#' then
          readline(infile, pgm);
        elsif pgm(pgm'left) = ' ' or pgm(pgm'left) = ht then
          read(pgm, ch);
        else
          ok := true;
        end if;
      end loop;
    end skip;


    procedure init_pgm(pgm: inout line) is
    begin
      readline(infile, pgm);
    end init_pgm;


    procedure read_pgm_header(pgm: inout line; width: out integer; height: out integer) is
      variable ch: character;
      variable colors: integer;
    begin
      read (pgm, ch);
      ASSERT (ch = 'P') REPORT "Invalid input file!" SEVERITY error;
      read (pgm, ch);
      ASSERT (ch = '2') REPORT "Invalid input file!" SEVERITY error;
      skip(pgm);
      read (pgm, width);
      skip(pgm);
      read (pgm, height);
      skip(pgm);
      read (pgm, colors);
      ASSERT (colors <= 255) REPORT "Invalid input file!" SEVERITY error;
    end read_pgm_header;


    procedure read_pgm_pixel(pgm: inout line; data: out pixel) is
    begin
      skip(pgm);
      read(pgm, data);
    end read_pgm_pixel;

END read_pgm;
```

C.2.2 tb.vhd

```
--------------------------------------------------------------------------
--      File            :       tb.vhd
--      Description     :       Test Bench for JPEG Encoder
--------------------------------------------------------------------------
--
-- 05/26/99   A. Gerstlauer

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use STD.textio.all;

use work.read_pgm.all;

entity tb is
end tb;

architecture test of tb is

  component Jpeg
    port (
      Reset: in STD_LOGIC;
      InReady: out STD_LOGIC;
      InDataValid: in STD_LOGIC;
      OutDataReady : out STD_LOGIC;
      InData: in UNSIGNED(7 downto 0);
      OutData: out UNSIGNED(7 downto 0)
    );
  end component;
  for all: Jpeg use entity work.Jpeg(dummy);

  signal reset, in_ready, in_valid, out_ready: STD_LOGIC;
  signal in_data, out_data: UNSIGNED(7 downto 0);

  file outfile: text is out "outfile.txt";

begin
  J1: Jpeg port map (reset, in_ready, in_valid, out_ready, in_data, out_data);

  Stimuli: process
    procedure write_data(data: in integer range 0 to 255) is
    begin
      wait until in_ready = '1';
      in_data <= CONV_UNSIGNED(data, 8);
      in_valid <= '1';
      wait until in_ready = '0';
      in_valid <= '0';
    end write_data;

    variable i: integer;
    variable l: line;
    variable w,h: integer;
    variable p: pixel;
  begin
    reset <= '1';
    init_pgm(l);
    wait for 0 ns;
    reset <= '0';

    read_pgm_header(l, w, h);
    write_data(w / 256);
    write_data(w rem 256);
    write_data(h / 256);
    write_data(h rem 256);
```

```
    for i in 1 to w*h loop
      read_pgm_pixel(l, p);
      write_data(p);
    end loop;

    wait;
  end process Stimuli;


  Monitor: process
    procedure read_data(data: out integer range 0 to 255) is
    begin
      wait until out_ready = '1';
      data := CONV_INTEGER(out_data);
      wait until out_ready = '0';
    end read_data;

    variable l: line;
    variable done: boolean;
    variable marker: boolean;
    variable p: integer range 0 to 255;
  begin
    done := false;
    marker := false;
    while not done loop
      read_data(p);
      write(l, p);
      writeline(outfile, l);
      if marker then
        if p = 217 then
          ASSERT FALSE REPORT "End of simulation!" SEVERITY FAILURE;
        else
          marker := false;
        end if;
      else
        if p = 255 then
          marker := true;
        end if;
      end if;
    end loop;
  end process Monitor;


end test;
```

D FSMD_MEM Architecture



E Quantization RTL code for FSMD_MEM Architecture

E.1 Quan_FSMD.vhd

```
--------------------------------------------------------------------------
--      File            :       quantization.vhd
--      Description     :       FSM model for a quantization module
--      Designer        :       Hongxing Li
--      Date            :       May 24, 1999
--------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;
use work.typedef.all;

entity QuantizationCtrl is
        port (
                Clk :           in STD_LOGIC;
                Start :         in STD_LOGIC;
                Done :          out STD_LOGIC;
                WE_I, CS_I : out std_logic;
                Addr: out INTEGER;
                DataIn :        in INTEGER;
                DataOut :       out INTEGER;
                TestOut:        out INTEGER
                );
end QuantizationCtrl;


architecture sequential of QuantizationCtrl is
-------------------------------------------------------
--      QT_S0 -- Start State
--      QT_S2-QT_S7 -- Loop State
--      QT_S8 -- End State
-------------------------------------------------------

type STATE_VALUE is (
        QT_S0, QT_S1, QT_S2,QT_S2_2,QT_S2_3, QT_S3, QT_S4, QT_S5,
        QT_S6, QT_S7, QT_S8 );

signal i, r_add, r_div, r_ram1, r_ram2 : INTEGER;
signal  State : STATE_VALUE ;

begin

        process( Clk )


        begin
        if (Start='1') Then
                Done<='0';
                WE_I<='1';
                CS_I<='1';
                Addr<=0;
                State<=QT_S0;
                TestOut<=1;
        else


        if (Clk'event and Clk = '1') Then
                case State is
                        ------------------------------------------
                        --      State QT_S0
                        ------------------------------------------
```

```vhdl
when QT_S0 =>
        TestOut <=0;
        i <= 0;
        Done <= '0';
        State <= QT_S1;

----------------------------------------------
--      State  QT_S1
----------------------------------------------
when QT_S1 =>
        TestOut <=1;
        if (i < 64) then
                State <= QT_S2;
        else
                State <= QT_S8;
        end if;

----------------------------------------------
--      State  QT_S2
----------------------------------------------
when QT_S2 =>
        WE_I<='1';
        CS_I<='0';
        Addr<=i+Matrix_Addr;
        State <= QT_S2_2;

----------------------------------------------
--      State  QT_S2_2
----------------------------------------------
when QT_S2_2 =>
        WE_I<='1';
        CS_I<='0';
        Addr<=i+QCMatrix_Addr;
        --r_ram2 <= QuantizationCtrlMatrix(i);
        State <= QT_S2_3;

----------------------------------------------
--      State  QT_S2_3
----------------------------------------------
when QT_S2_3 =>
        r_ram1<=DataIn;
        State <=QT_S3;
----------------------------------------------
--      State  QT_S3
----------------------------------------------
when QT_S3 =>
        r_div <= DataIn/2;
        r_ram2<=DataIn;
        if (r_ram1 > 0) then
                State <= QT_S4;
        else
                State <= QT_S5;
        end if;

----------------------------------------------
--      State  QT_S4
----------------------------------------------
when QT_S4 =>
        r_add <= r_ram1 + r_div;
        State <= QT_S6;

----------------------------------------------
--      State  QT_S5
----------------------------------------------
when QT_S5 =>
        r_add <= r_ram1 - r_div;
```

```
                    State <= QT_S6;

                ----------------------------------------
                --      State QT_S6
                ----------------------------------------
                when QT_S6 =>
                    r_div <= r_add * r_ram2;
                    State <=  QT_S7;

                ----------------------------------------
                --      State QT_S7
                ----------------------------------------
                when QT_S7 =>
                    --output(i) := r_div;
                    WE_I<='0';
                    CS_I<='0';
                    Addr<=i+Matrix_Addr;
                    DataOut <=r_div;
                    i <= i + 1;
                    State <=  QT_S1;

                ----------------------------------------
                --      State QT_S8
                ----------------------------------------
                when QT_S8 =>
                    Done <= '1';
                    TestOut <=2;
                    State <= QT_S0;

            end case;
        end if;

        end if; --start
        end process;
end sequential;
```

E.2 Quan_MEM.vhd

```
-- memory.vhd

-- simple memory
-- Junyu Peng

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;
use IEEE.std_logic_unsigned.all;
use STD.textio.all;
use work.typedef.all;


entity MyMemory is
        port (WE_I : in std_logic;
                Addr : in INTEGER;
                CS_I: in std_logic;
                OE_I: out std_logic;
                DataIn : in INTEGER;
                DataOut : out INTEGER;
                TestIn : in INTEGER
                );
end MyMemory;

architecture simple of MyMemory is
        type mem_array is ARRAY(0 to 191) of INTEGER;
        File load_file : text is in "data.txt";

signal mem : mem_array;

begin
        process(CS_I,WE_I,Addr,TestIn)

        variable count, data : INTEGER;
        variable in_line : LINE;
        variable ok : BOOLEAN;

        begin

        OE_I <='1';
        mem(128 to 191) <=(
                16, 11, 10, 16, 24, 40, 51, 61,
                12, 12, 14, 19, 26, 58, 60, 55,
                14, 13, 16, 24, 40, 57, 69, 56,
                14, 17, 22, 29, 51, 87, 80, 62,
                18, 22, 37, 56, 68, 109, 103, 77,
                24, 35, 55, 64, 81, 104, 113, 92,
                49, 64, 78, 87, 103, 121, 120, 101,
                72, 92, 95, 98, 112, 100, 103, 99 );


        if (TestIn=0) then
                if not endfile(load_file) then

                readline(load_file, in_line);
                for count in 0 to 63 loop
                        read(in_line, data, ok);
                        mem(count+Matrix_Addr) <= data;
                end loop;

                readline(load_file, in_line);
                for count in 0 to 63 loop
                        read(in_line, data, ok);
```

```
                    mem(count+MatrixRslt_Addr) <= data;
            end loop;

            else
--                  assert false
--                  report "End of file" severity error;
            end if;

    else
            if (TestIn=2) then
                    for count in 0 to 63 loop
                            ASSERT(mem(count+Matrix_Addr)=mem(count+MatrixRslt_Addr))
                            REPORT "error: incorrect output"
                            SEVERITY error;
                    end loop;
            else



            if (WE_I ='0' and CS_I='0') then
                    mem(Addr) <=TRANSPORT DataIn after 13 ns;
                    OE_I <='1';
            else
                    if (CS_I='0') then
                            DataOut <= TRANSPORT mem(Addr) after 13 ns;
                    else
                    OE_I <='0';
                    end if;
            end if;

            end if; --for Test=2;
    end if; --for Test=0;

    end process;
end simple;
```