Pipelined Implementation of the Baseline JPEG Encoder

Final Report

Abstract

In this report, we describe the design and implementation of a fully pipelined architecture for implementing the JPEG baseline image compression standard. The architecture exploits the principles of pipelining and parallelism in order to obtain high speed and throughput. The design was synthesized to Altera's FLEX 10K series FPGAs, and synthesis was carried out using Altera's Max+PlusII environment. It has been estimated that the entire architecture can be implemented on a single FPGA to yield a clock rate of about 20 Mhz which would allow an input rate of 20 mega-samples/sec.

Acknowledgements

The group would like to thank the following people for their kind and impartial support throughout the project design process:

- Mr. Muhammad Nauman, our project advisor, for providing us with much needed advice, arranging for the necessary tools, and of course, for having introduced us to the digital design field in the first place.
- Mr. Tariq Muhammad, Principal Engineer, Hughes Network Systems, for supplying the necessary tools and equipment, and his valuable encouragement.
- Mr. Khurram Hasan Shafiq, for providing us with most of the articles upon which our work has been based, and especially for his prompt replies to our queries.

We would also like to thank the other Verilog project groups for all the brain storming sessions, suggestions and exchanges of ideas and study materials.

Last but not least, we would like to thank our families, for their untiring support and understanding.

Contents

1 The Significance of Image Compression

Over the years, the need for image compression has grown steadily. Currently, it is recognized as an 'enabling technology'. For example, image compression has been and continues to be crucial to the growth of multimedia computing. In addition, it is the natural technology for handling the increased spatial resolutions of today's imaging sensors, and evolving broadcast television standards. Furthermore, image compression plays a crucial role in many important and diverse applications, including video-conferencing, remote sensing, document and medical imaging, facsimile transmission (FAX), and the control of remotely piloted vehicles in military, space, and hazardous waste control applications. In short, an ever-expanding number of applications depend on the efficient manipulation, storage, and transmission of binary, gray-scale, or color images.

1.1 Data Compression Basics

Data compression is the reduction or elimination of redundancy in data representation in order to achieve savings in storage and communication costs. Data compression techniques can be broadly classified into two categories: lossless and lossy. In lossless methods, the exact original data can be recovered while in lossy schemes a close approximation of the original data can be obtained.

Lossless compression consists of those techniques guaranteed to generate an exact duplicate of the input data stream after a compress/expand cycle. The lossless methods are also called entropy-coding schemes, since there is no loss of information

content during the process of compression. This type of compression is used in certain environments such as compression of text, database records, spreadsheets, word processing files, or medical and military imaging medical imaging where no loss of information is tolerated, or could even be catastrophic. Typical compression ratios for lossless data compression are around 3:1.

Lossy data compression concedes a certain loss of accuracy in exchange for greatly increased compression. Lossy compression methods are commonly applied to digitized image and audio. By their very nature, these digitized representations of analog phenomena are not perfect to begin with, so the idea of output and input not matching exactly is a little more acceptable. Most lossy compression techniques can be adjusted to different quality levels gaining higher accuracy in exchange for less effective compression, and depending upon the fidelity required, compression ratios of even up to 100:1 can be obtained.

As an example, consider the compression of digital images, which require an enormous amount of storage space in their uncompressed form. For example, a color image with a resolution of 1024 ∗ 1024 pixels with 24 bits per pixel would require 3.15M bytes in uncompressed form. At a video rate of 30 frames per second, this requires a data rate of 94M bytes per second. With the recent advances in video applications such as video teleconferencing, HDTV, home entertainment systems, interactive visualization and multimedia, there is an increasing demand for even higher bandwidth computing and communication systems. Very high-speed implementation of efficient image compression techniques will significantly help in meeting that challenge.

In the late 1980s, a standardization effort known by the acronym JPEG, for Joint Photographic Experts Group, started working towards establishing an international digital image compression standard for continuous-tone (multilevel) still images, both grayscale and color. The 'joint' in JPEG refers to collaboration between three international standard organizations, International Telegraph and Telephone Consultative Committee (CCITT), International Organization for Standardization (ISO), and the International Electrotechnical Commission (IEC).

JPEG has defined an international standard for coding and compression of continuous tone still images. This standard is commonly referred to as the JPEG standard. The primary aim of the JPEG standard is to propose an image compression algorithm that would be generic, application independent and aid VLSI implementation of data compression.

To meet the differing needs of many applications, the JPEG standard includes two basic compression methods, each with various modes of operation. A DCT (Discrete Cosine Transform) based method is specified for 'lossy' compression, and a predictive method for 'lossless' compression. JPEG features a simple lossy technique known as the baseline method, a subset of the other DCT-based modes of operation.

The Baseline method has been by far the most widely implemented JPEG method to date, and is sufficient in its own right for a large number of applications.

2 Outline of the JPEG Standard

The basic model for the JPEG encoder is shown in Figure 2.1.

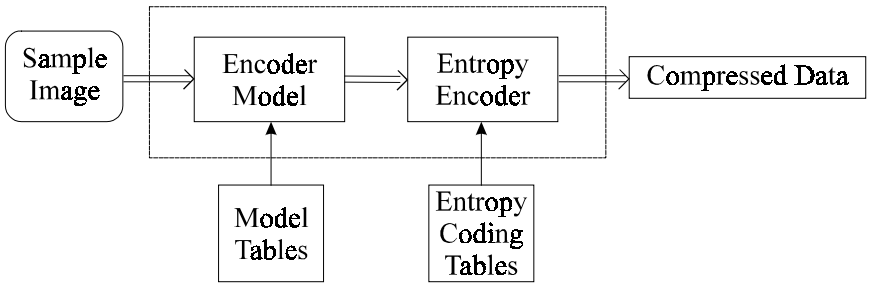


Figure 2.1: The JPEG Baseline Encoder

The encoder model transforms the input image into an abstract representation more suitable for further processing. The encoder model may require parameters stored in some model tables for achieving this transformation. The entropy encoder is a compression procedure that converts the output of the encoder model into a compressed form. Also, the entropy encoder may use tables for storing the entropy codes. Four distinct coding processes were derived based on the above-described JPEG model:

- Baseline process,
- Extended DCT-based process,
- Lossless process,
- Hierarchical process.

The baseline and the extended processes are also known as DCT-based processes since they use DCT within the encoder model. The lossless process uses prediction-based methods within the encoder model. The hierarchical process uses the encoder model form the extended process or the lossless process. The baseline process uses Huffman

codes for entropy encoding, while the other three processes use either Huffman or Arithmetic coding. Since the focus of our project is on the VLSI/firmware implementation of the baseline process, the remainder of this section deals only with the baseline process in detail.

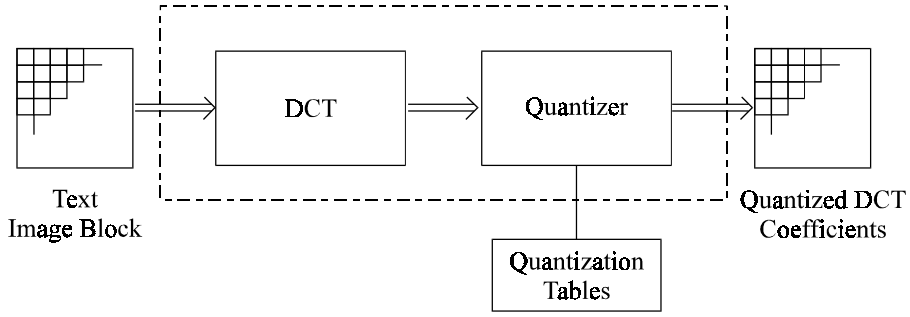


Figure 2.2: The JPEG Encoder Model

The encoder model for the baseline process is shown in Figure 2.2. The input image is divided into non-overlapping blocks of 8 x 8 pixels, and input to the baseline encoder. The pixel values are converted from unsigned integer format to signed integer format, and DCT computation is performed on each block. DCT transforms the pixel data into a block of spatial frequencies that are called the DCT coefficients. Since the pixels in the 8 x 8 neighborhood typically have small variations in gray levels, the output of the DCT will result in most of the block energy being stored in the lower spatial frequencies. On the other hand, the higher frequencies will have values equal to or close to zero and hence, can be ignored during encoding without significantly affecting the image quality. The selection of frequencies based on which frequencies are most important and which ones are less important can affect the quality of the final image. JPEG allows for this by letting the user predefine the quantization tables used in the quantization step that follows the DCT computation. The selection of quantization

values is critical since it affects both the compression efficiency, and the reconstructed image quality.

The block of DCT coefficients output by the encoder model is rearranged into one dimensional data by using zigzag reordering as shown in Figure 2.3.

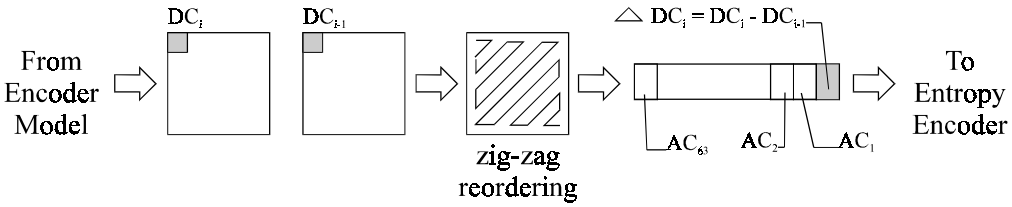


Figure 2.3: Zigzag Reordering of DCT Output

The location (0,0) of each block 'i' contains the DC coefficient for the block represented as DC_i. This DC coefficient is replaced by the value ΔDC_i which is the difference between the DC coefficients of block 'i' and block 'i – 1'. Since the pixels of adjacent blocks are likely to have similar average energy levels, only the difference between the current and the previous DC coefficients is used, which is commonly known as Differential Pulse Code Modulation (DPCM) technique. It should be noted that the high frequency coefficients that are more likely to be zeroes get grouped at the end of the one-dimensional data due to the zigzag reordering.

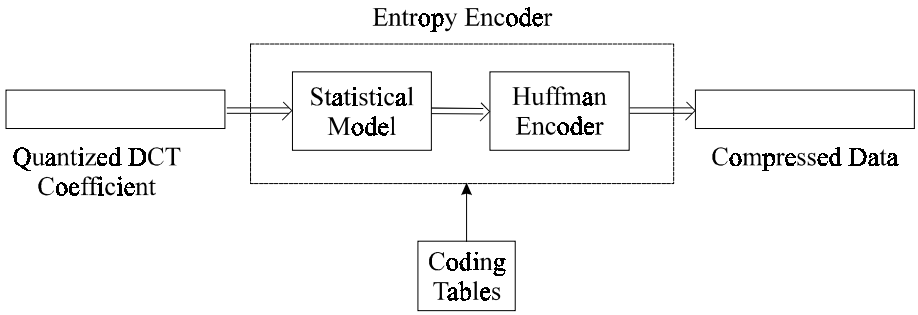


Figure 2.4: The JPEG Baseline Entropy Encoder

The entropy encoder details are shown in Figure 2.4. The entropy encoder uses variable length encoding based on a statistical model in order to encode the rearranged DCT coefficients. In the entropy encoder, the quantized DCT coefficients are converted into a stream of [runlength count, category] pairs. For each pair, there is a corresponding variable length Huffman code which will be used by the Huffman encoder to perform the compression. The Huffman codes are stored in a table. A detailed description of the various steps in the entropy encoder is given later in Section four.

In order to achieve better compression results, very often, input images are transformed to a different color space (or color coordinates) representation before being input to the encoder. Although the JPEG algorithm is unaffected by color, since it processes each color independently, it has been shown that by changing the color space, the compression ratio can be significantly improved. This is due to the perception of the human visual system and the less perfect characteristics of the display devices. One of the most appropriate color spaces for the JPEG algorithm has been shown to be YCbCr, where Y is the luminance component and Cb and Cr are the two chrominance components. Since the luminance component carries much more information compared to the chrominance components, JPEG allows different tables to be used during compression.

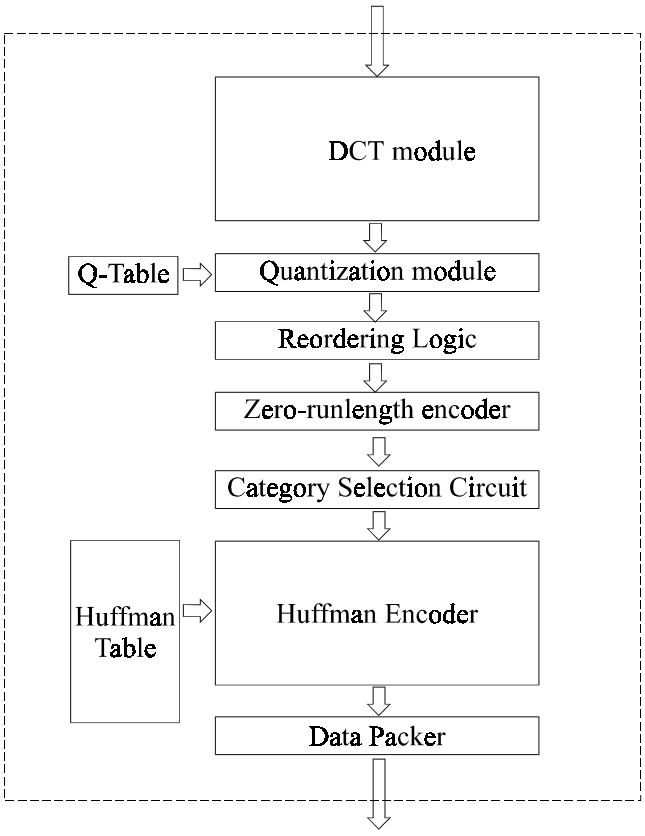
2.1 The JPEG System Architecture

The system architecture for our implementation is shown in Figure 2.5. The entire architecture is organized as a linear multistage pipeline in order to achieve high

throughput. This figure reflects the sequence of computation in the JPEG Baseline process. The architecture consists of the encoder model, and the entropy encoder.

The encoder model consists of a DCT module, a quantization module, and reordering logic. The entropy encoder consists of several modules such as the zero runlength encoder, category selection unit, Huffman encoder and data packer. The image to be

Figure 2.5: JPEG System Architecture



compressed is input to the architecture at the rate of one pixel per clock cycle. The input data is processed by the various modules in a linear fashion, where each module itself is organized internally as a multistage linear pipe. The compressed data is output by the system at a variable rate depending on the amount of compression achieved. The design and implementation of each module is described in detail in the next section.

3 The Encoder Model

The encoder model has three main components: 1.) DCT module, 2.) Quantization module, and 3.) Zigzag reorder buffer. In this section, we describe thoroughly the implementations of each of these modules. Since the research on the various modules of JPEG is rich and mature, the descriptions presented here are limited only to cover the requirements for JPEG implementation in hardware.

3.1 DCT Module

The Discrete Cosine Transform is the most complex operation that needs to be performed in the baseline JPEG process. This subsection starts with an introduction to our chosen DCT architecture, followed by a detailed mathematical explanation of the principles involved. We then present a comparison with several other proposed architectures.

3.1.1 Basis for the Design.

Our implementation of the Discrete Cosine Transform stage is based on an architecture proposed in [1]. Our choice of this particular architecture was due to a multitude of reasons. The design uses a concurrent architecture that incorporates distributed arithmetic and a memory oriented structure to achieve high speed, high accuracy, and efficient hardware realization of the 2-D DCT.

3.1.2 Mathematical Description of the Architecture.

The Discrete Cosine Transform is an orthogonal transform consisting of a set of basis vectors that are sampled cosine functions. The 2-D DCT of a data matrix is defined as

$$Z = C^t XC.....(1)$$

where X is the data matrix, C is the matrix of DCT Coefficients, and C^t is the Transpose of C. The normalized *N*th order DCT matrix is defined as

$$C = \begin{bmatrix} c_{1,1}, c_{1,2}, ..., c_{1,l} \\ c_{2,1}, c_{2,2}, ..., c_{2,l} \\ . \\ c_{k,1}, c_{k,2}, ..., c_{k,l} \end{bmatrix} .....(2)$$

where, for an *N* x *N* data matrix,

$$c_{k,l} = \sqrt{\frac{2}{N}} \cos\left[\frac{(2k-1)(l-1)\pi}{2N}\right] .....(3)$$

for $k = 1, 2, \ldots, N$, $l = 2, 3, \ldots, N$, and $c_{k,l} = N^{-1/2}$ for $l = 1$.

The 2-D DCT (8 x 8 DCT) is implemented by the row-column decomposition technique. We first compute the 1-D DCT (8 x 1 DCT) of each column of the input data matrix X to yield X^tC. after appropriate rounding or truncation, the transpose of the resulting matrix, C^tX, is stored in an intermediate memory. We then compute another 1-D DCT (8 x 1 DCT) of each row of C^tX to yield the desired 2-D DCT as defined in equation (1). A block diagram of the design is shown in Figure 3.1.

Denoting the 1-D DCT of an N x N data matrix X by Y = XC, then the (*k*, l)th element of Y is

$$y_{k,l} = x_k^t c_l .....(4)$$

where x^t_k is the kth row vector of X and c_l is the lth column vector of C. The expression in equation (4) suggests that for each x_k, a row vector ($y_{k,1}$, $y_{k,2}$, …, $y_{k,N}$) can be generated concurrently.

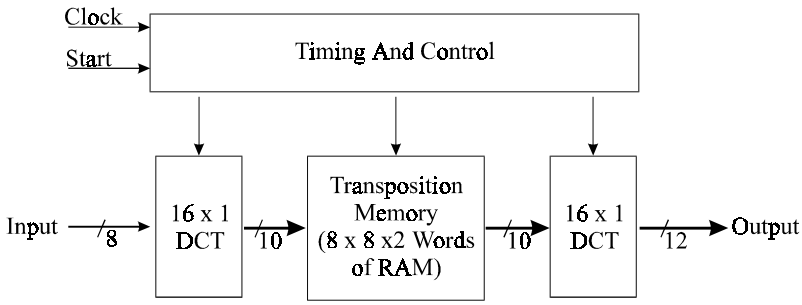


Figure 3.1: 2-D DCT Architecture

Let the element of the data matrix X be represented by the 2's complement code as follows:

$$x_{k,m} = -x_{k,m}^{(n-1)} 2^{n-1} + \sum_{j=0}^{n-2} x_{k,m}^{(j)} 2^j \text{.....(5)}$$

where $x^{(j)}_{k,\,m}$ is the jth bit of $x_{k,\,m}$ which has a value of either 0 or 1, n is the number of bits $x_{k,\,m}$ carries, and $x^{(n-1)}_{k,\,m}$ is the sign bit. Substituting equation (5) into equation (4), we have

$$y_{k,l} = -\sum_{m=1}^{N} c_{m,l} x_{k,m}^{(n-1)} 2^{n-1} + \sum_{j=0}^{n-2} \sum_{m=1}^{N} c_{m,l} x_{k,m}^{(j)} 2^j$$

OR

$$y_{k,l} = -F_{k,l}(c_l, x_k^{(n-1)}) 2^{n-1} + \sum_{j=0}^{n-2} F_{k,l}(c_l, x_k^{(j)}) 2^j \text{.....(6)}$$

where

$$F_{k,l}(c_l, x_k^{(j)}) = \sum_{m=1}^{N} c_{m,l} x_{k,m}^{(j)} \text{.....(7)}$$

for $j = 0, 1, \ldots, n\text{-}1$.

$F_{k,l}$ is a function of the coefficients, $c_{m,l}$, and bit patterns of $x_{k,m}$. Since the $c_{m,l}$ values are fixed numbers, $F_{k,l}$ can be generated and stored in memory for all possible bit patterns of $x_{k,m}$. Consequently $y_{k,l}$ defined in equation (4) can be calculated concurrently for $l = 1, 2, \ldots, N$ by shifts and adds of values of $F_{k,l}$ stored in memory. This is the basic principle of distributed arithmetic with a memory lookup approach to compute the row vectors of a 1-D DCT concurrently.

A straightforward implementation of the above approach on a single FPGA would be difficult for an 8 x 8 (2-D) DCT because it would require 2 x 8 x $2^8 = 2^{12}$ words of ROM. This value is arrived at in the following manner. One bit of each data element in a row (which is N elements long) is multiplied by a column of the DCT coefficient matrix (also N elements long), and the summation of the N results is to be stored. Since there are 2^N possible combinations of the data bits, and there are N columns in the DCT coefficient matrix, the number of data words that need to be stored for a 1-D DCT transformation are N x 2^N. A 2-D DCT process would therefore require 2 x N x 2^N words of ROM. For 8 x 8 matrices, this would equal 4096 words, or 4k words. However, certain techniques can be used to reduce the required size of the ROM (lookup table).

Before discussing these techniques, it may be helpful to consider the schematic diagram of the architecture, shown in Figure 3.2. It must be noted, however, that this

schematic does not represent the actual implementation that we have used. The actual diagram, along with an explanation of the differences, and their reasons, will be given later.

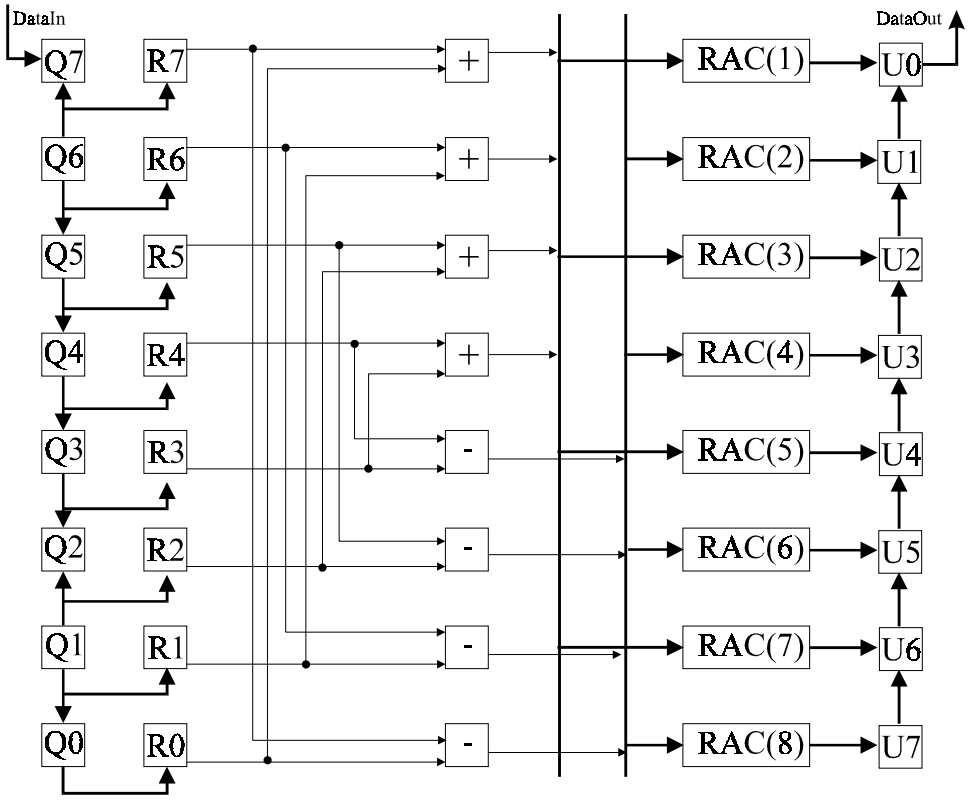


Figure 3.2: Block Diagram of the 1-D DCT

As can be seen, the data sequence $x_{k,1}, \ldots, x_{k,8}$ is shifted sequentially in time with bit parallel structure into the 8-stage Q shift registers at the input data rate $1/T$. After every $8T$ time interval, the contents of the Q shift registers are concurrently bit parallel loaded into the R shift registers. The data in the R shift registers are then concurrently bit serial shifted out with the least significant bit first. Now, instead of applying the bit patterns shifted out of the R shift registers directly to the ROM and Accumulator (RAC), a special technique is used to reduce the size of the lookup table.

By taking advantage of the specific pattern of the DCT matrix, it can be shown that

$$y_{k,l} = \sum_{m=1}^{N/2} u_{k,m} c_{m,l} \ldots\ldots(8)$$

for $l = 1, 3, \ldots, N\text{-}1$ with $u_{k,m} = x_{k,m} + x_{k,N-m+1}$ and

$$y_{k,l} = \sum_{m=1}^{N/2} v_{k,m} c_{m,l} \ldots\ldots(9)$$

for $l = 2, 4, \ldots, N$, with $v_{k,m} = x_{k,m} - x_{k,N-m+1}$.

Equations (8) and (9) imply that with the variables u and v replacing the original data sequence x, the summation from 1 to N in (6) becomes a summation from 1 to $N/2$. Therefore, the number of data lines used to address a ROM is reduced by a factor of 2, and the number of required words for each $F_{k,l}$ is reduced by a factor of $2^{N/2}$. This is the same as the first- stage butterfly used in most fast algorithms. It does not require multiplications and can be implemented using serial adders and subtractors, which require much less logic resources (for an FPGA) or chip area (for ASIC implementation). The values of $F_{k,l}$ for storage in ROM were generated using a simple software routine written in C. The code has been included in the appendix.

The output from the ROM's, or lookup tables, is then added to the output of the accumulator, which has been shifted right by 1 bit. The same operations are repeated n+1 times, except for the last time (sign bit), a subtraction instead of addition is

executed. Then, the outputs $y_{k,m}$, for m = 1, 2, …, 8, are bit parallel loaded into the U shift registers simultaneously as shown in the schematic above, and then shifted out sequentially in time. These intermediate data are stored in the transpose memory for matrix transposition, followed by another 1-D (8 x 1) DCT.

It is important to note that the values for $F_{k,l}$ obtained from equation (7) are very small. Since in this architecture, we are using fixed-point arithmetic, this can result in serious truncation errors. To avoid this, the values of $F_{k,l}$ are left shifted 8 times before storage (i.e. multiplied by 2^8). The effect of this shift is cancelled out by the 8 subsequent right shifts in the shift-and-accumulate register (equivalent to divide by 2^8).

3.1.3 Differences in our implementation.

The key difference between our implementation of the DCT module and the design proposed by [1], is that they have considered data blocks of 16 x 16 words in size (i.e. N = 16), whereas the block size that we must use is restricted to 8 x 8 words (i.e. N = 8). This detail prevented a direct implementation of the design presented in [1]. Thus several changes had to be made in order to accommodate for the 8 x 8 DCT.

The design proposed in [1] handles 16 x 16 blocks of 8-bit data (with internal precision of 16 bits), whereas the JPEG specification DCT operates on 8 x 8 blocks of 8-bit pixel data. At first the conversion to the 8 x 1 DCT seemed straightforward (by use of eight 8-bit Q and R-shift registers instead of sixteen 16-bit registers). However, overflow and loss of precision soon became hindrances.

Overflow. Addition of two signed 8-bit values could in some instances result in overflow conditions, which would necessitate the use of greater precision than 8-bits. However this means that more than 8 shifts would be required in order to obtain the correct results.

Loss of precision. Also, the outputs of the RAC registers after the first 1-D DCT are of 10 bits precision, thus the second 8 x 1 DCT would have to handle at least 11 bits of data.

Both these factors necessitate the use of greater precision than 8-bits. It was not possible to merely add extra bits to the R shift registers, as they have to be emptied within the 8*T* time interval, after which the contents of the Q registers are transferred to the R shift registers. On the other hand, extending the precision was absolutely necessary for proper operation.

Our solution thus involved the use of 16-bit internal precision, as well as shifting 2 bits out of the R shift registers after every *T* time interval. This modification had the desired result, as the R shift registers were cleared after every 8*T* time interval, due to the double shifting.

In the next sub-section is detailed the actual implemented 2-D DCT architecture and an extensive description of its working.

3.1.4 Architecture Description

The detailed schematic diagram of the actual implemented 1-D (8 x 1) DCT is shown in Figure 3.3. This design is used for both the row, and the column 8 x 1 DCT stages.

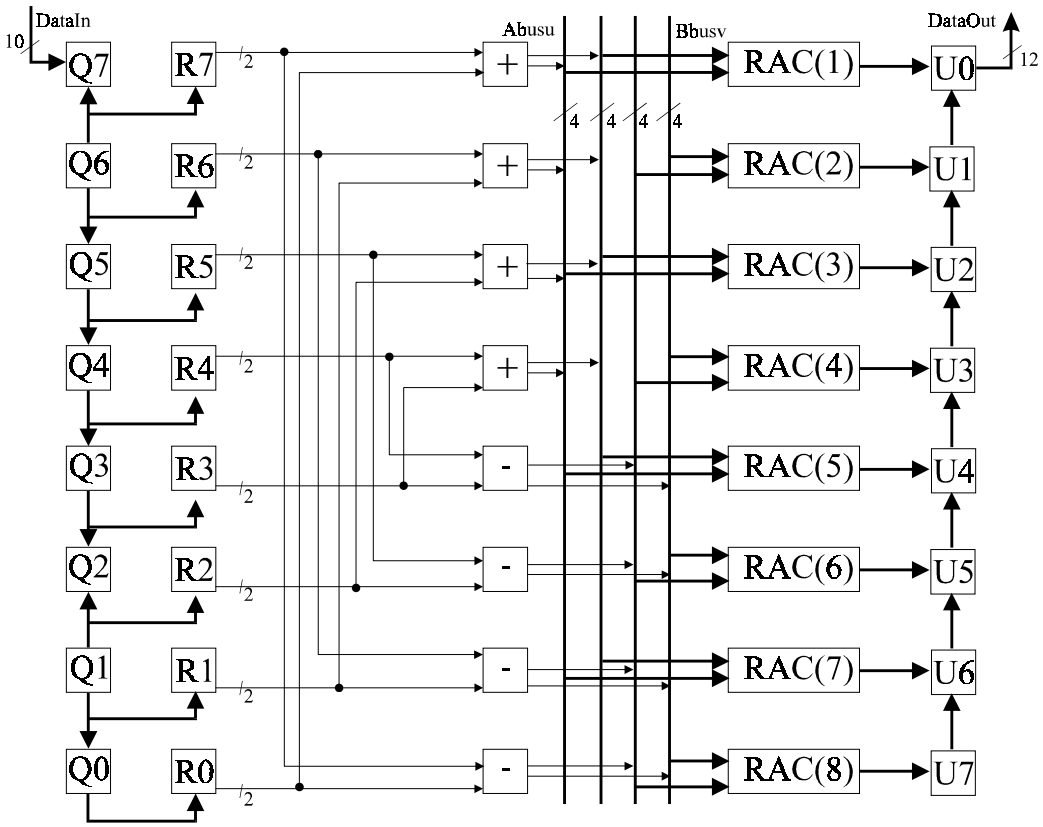


Figure 3.3: Schematic of actual 8 x 1 DCT

The input values to the first 1-D DCT stage are 8-bit, level shifted pixel values. Both the Q and the R shift registers are 16 bits wide. Thus the 8-bit input is sign extended to 16 bits, and then enters the pipeline as before. After 8*T*, or 8 clock cycles, the Q registers contain a new set of data, which is transferred to the R registers at the next clock pulse. Now, since there are 16 bits of data that need to be shifted out in 8 clock cycles, two bits are shifted out at a time. In order to handle two bits from each R register at a time, the consecutive adder and subtractor stages have also been modified: instead of four 1-bit adders and subtractors, we have used four 2-bit adders and four 2-

bit subtractors. The schematic diagram describing the operation of one such adder is given in Figure 3.4.

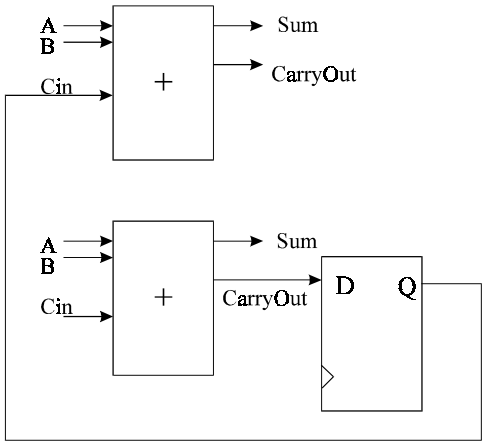


Figure 3.4: 2-bit Adder

Similarly the number of buses to the RACs, as well as the number of ROM's have been doubled. There are now four buses, uA, uB, vA and vB, each providing 4-bit data to 16 ROM's. The 16 ROM's are organized as 8 pairs, as shown in Figure 3.5.

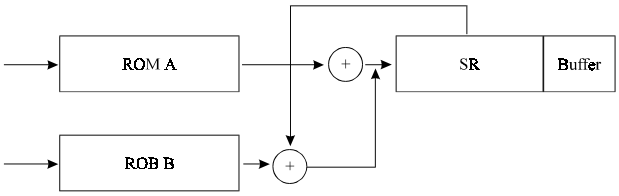


Figure 3.5: Internal Organization of RAC

Four of the pairs are addressed by the two u buses, while the other four are addressed by the two v buses. In each pair, the first ROM (ROMA) is addressed by either uA or vA, and the second (ROMB) is addressed by either uB or vB. The ROM's in each pair are identical, since they both handle data from the same source, and all produce

outputs 11-bit long. Each of the pairs is connected to a shift-and-accumulate register (SAR). Combined as shown, these blocks form the RAC unit described before.

The RAC block functions in the following manner: in each clock cycle, addresses are applied to the inputs of both the ROM's, ROMA and ROMB, from the four u and v buses, and the contents of the shift-and-accumulate register (SAR) are arithmetic-right shifted.

One important fact is that in our implementation, an 8-bit buffer is concatenated to the LSB of the SAR, and all arithmetic-right shift operations are performed on the whole {SAR, buffer} combination. This buffer only helps to preserve the last 8 bits shifted out of the SAR, and is not involved in any of the addition/subtraction operations. The reason for this preservation is that in our implementation, 16 shifts are performed instead of just 8. As was mentioned before, the first 8 right-shifts are sufficient for canceling the effect of the 8 bit left-shifted $F_{k,l}$ values (refer to last paragraph, section 3.1.2), but the next 8 right-shifts can cause loss of data due to truncation, unless accounted for.

The output from ROMA is then added to the shifted contents of the SAR. The 8-bit buffer is not included in this addition. The result is then placed back in the SAR, and then arithmetic-right shifted *combinationally* (i.e. within the same clock cycle). This arithmetic shifted result is then added to the output from ROMB. At the next clock pulse, this twice shifted and added result is reentered into the SAR.

At the end of 8 clock cycles the least significant 12 bits of the {SAR, Buffer} combination are bit parallel loaded into the U registers simultaneously, and then shifted out sequentially in time. New outputs are produced from the U registers every clock cycle, and are input to the transpose buffer for reordering prior to the second 1-D DCT.

3.1.5 The Transpose Buffer

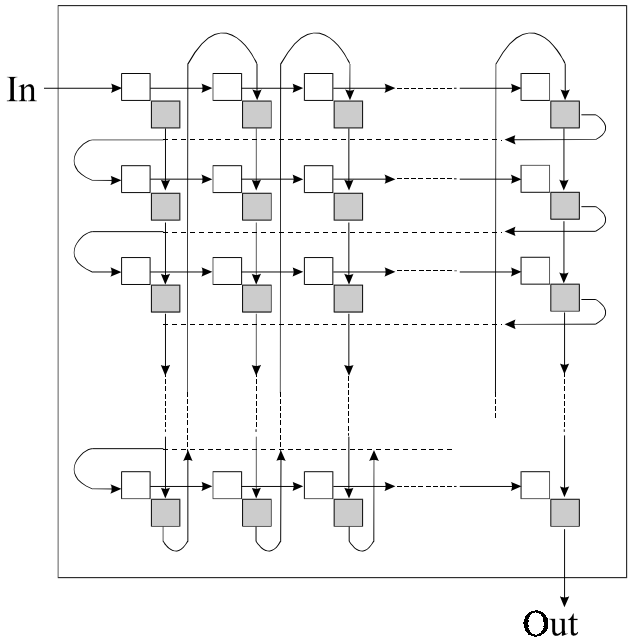


Figure 3.6: Structure of the Transpose Buffer

The transpose buffer is shown in Figure 3.6. The implementation is based on a design suggested in [2]. The buffer consists of an 8 x 8 array of register pairs organized as shown. The data is input to the transpose buffer in row-wise fashion until all the 64 registers are loaded. The data in those registers are copied in parallel onto the corresponding adjacent registers, which are connected in column wise fashion. While the data is being read out from the column registers, the row registers will keep receiving further data from the DCT module. Thus, the output of the row-wise DCT computation is transposed for the column-wise DCT computation. The transpose

buffer has a latency of 65 clock cycles. Thus the 8 x 8 DCT pipeline has a total latency of 103 clock cycles, as both the 1-D DCT stages have a latency of 19 clock cycles.

3.1.6 Comparison to Other Approaches

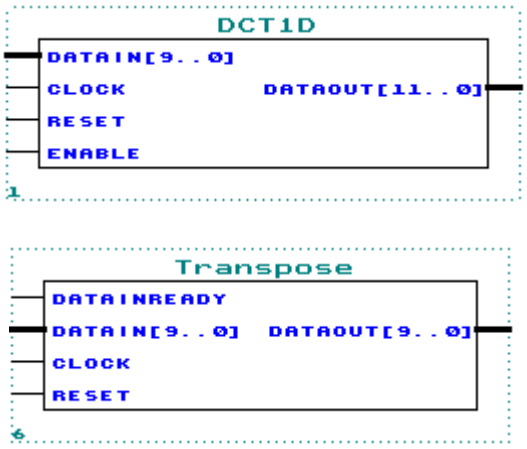
Due to the wide spectrum of applications in which the DCT is employed, the research on DCT circuits is rich and mature, resulting in a vast amount of literature. For the implementation of the DCT stage, several architectures were considered. The direct implementation of equation (1) (Sec 3.1.3) is computationally intensive, requiring 1024 multiplications and 1024 accumulations to calculate an 8x8 DCT. In order to reduce the number of required multiplications many implementations in the literature use various forms of butterfly structures with fewer number of multipliers. However, many multipliers are still required to maintain high throughput. Multipliers require a relatively large amount of logic resources. Moreover, the butterfly approach often results in an irregular architecture and complicated routing which may also result in a large circuit area. Also, since multiple stages of multiplications are accompanied by rounding and truncation in finite-precision arithmetic, fixed internal precision can cause resulting accuracy to be seriously degraded.

The architecture that we choose for our implementation uses distributed arithmetic and a memory-oriented structure. The merits of this architecture are: (1) saving of circuit area by replacement of multipliers by memory look-up tables; (2) the expectation of higher accuracy results given the same internal precision because the accumulated results undergo fewer rounding/truncation stages than the other structures; (3) more structural regularity which allows modular design; and (4) area saving and high speed

operations resulting from the combined advantages of bit-serial and bit-parallel structures. These features lead to a high performance design composed of memories, adders and registers only. No multipliers are required.

Compared to the DCT stage used in the baseline JPEG architecture proposed in [2], the latency of this DCT approach is less. The 1-D DCT latency in [2] is 59 cycles, compared to 19 for the architecture implemented here.

3.1.7 Module Block Diagrams



For description of ports and the verilog source code, refer Appendix A.

3.2 Quantization Module

The quantization module is shown in Figure 3.7. It consists of a ROM to store the quantization table and an 11 x 8 bit multiplier. The quantization step in the JPEG algorithm involves multiplying the output of the DCT stage with a set of predefined values from a quantization table. Since the DCT architecture is organized as a linear multistage pipeline, in order to maintain the same throughput throughout the pipeline,

we needed to implement a high speed, pipelined multiplier. The multiplier we have implemented has a Wallace-tree design. A schematic is shown in Figure 3.8. The 8-bit multiplier value is retrieved from the Quantization Table each clock cycle, and the coefficient values from the DCT stage are input as the 12-bit multiplicands.

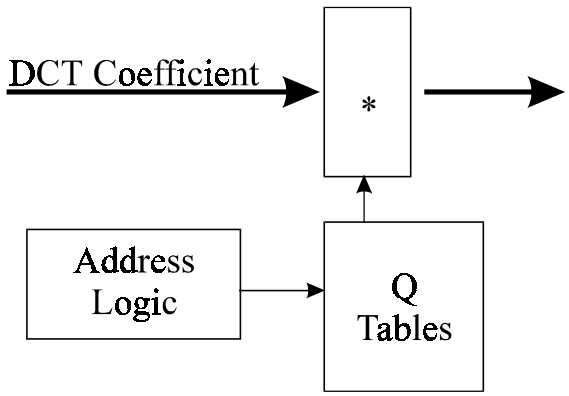


Figure 3.7: Structure of Quantization Module

Quantization is basically a division process that is converted into multiplication by simply inverting the quantization table values. In order to maintain precision, these inverted values are biased by multiplying them with 2^8 (i.e. 8 times left shift) prior to storage.

The product of multiplication is a 20-bit value, whose least significant 8 bits are discarded. This has a reverse biasing effect (i.e. 8 times right shift) on the out put.

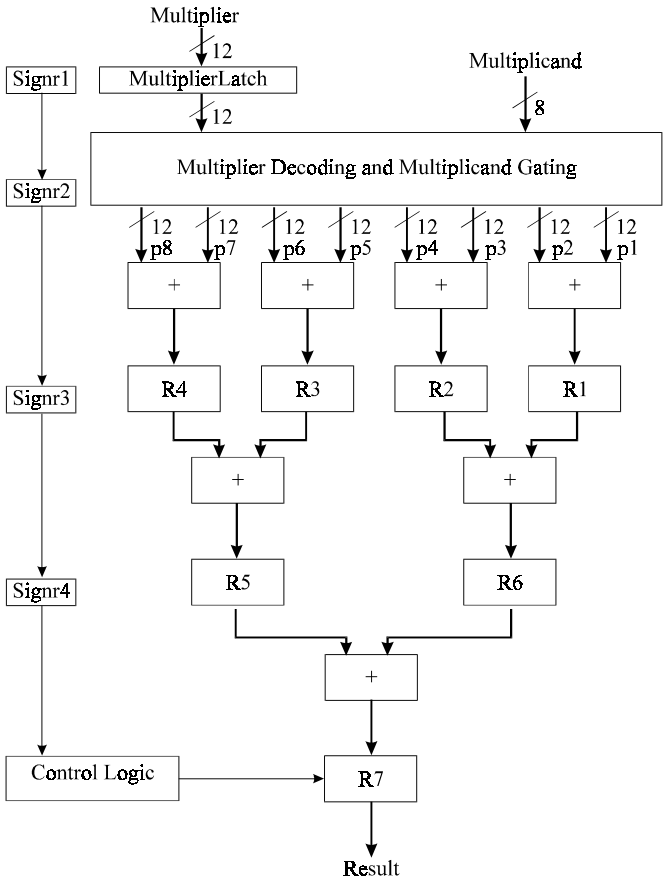
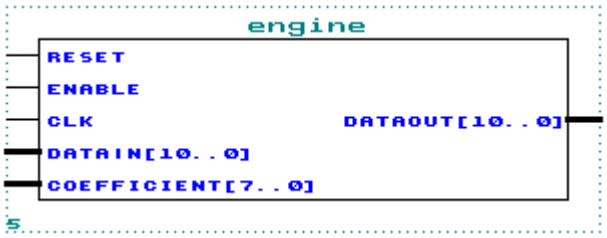


Figure 3.8: Wallace Tree Multiplier for the Quantization Module

3.2.1 Module Block Diagram

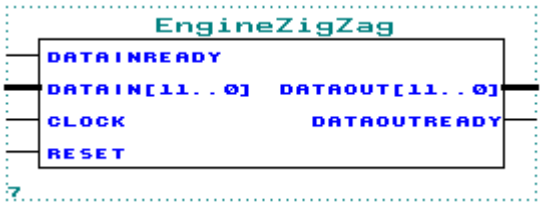


For description of ports and the verilog source code, refer Appendix A.

3.3 Zigzag Reordering Buffer

Each block of data that is output by the quantization module needs to be reordered in a zigzag fashion before being forwarded to the entropy encoder. This reordering is achieved using an 8 x 8 array of register pairs organized in a fashion similar to the transpose buffer. This implementation is also based on a design suggested in [2].

3.3.1 Module Block Diagram



For description of ports and the verilog source code, refer Appendix A.

4 The Entropy Encoder

The function of the entropy encoder is to code the quantized coefficients from the encoder model using variable length encoding. Our implementation of the entropy encoder is based on a design suggested in [2]. The reasons for this choice are several: first and foremost was the need for a pipelined architecture that consumes input data at the same rate as the Encoder Model. Another reason was the unavailability of literature on JPEG specific architectures for entropy encoders.

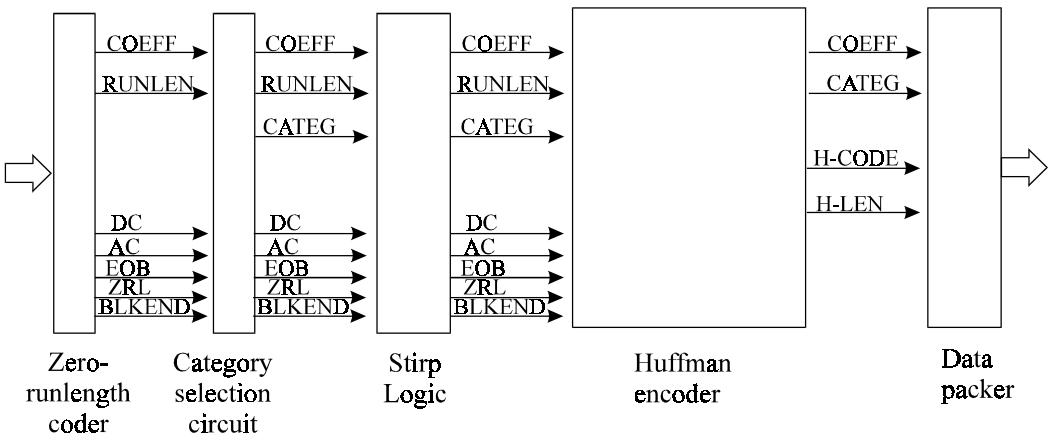


Figure 4.1: Entropy Encoder Architecture

The architecture of the entropy encoder is shown in Figure 4.1. As can be seen, the entropy encoder consists of 1.) Zero runlength coder, 2.) Category selection unit, 3.) Strip logic, 4.) Huffman encoder, and 5.) Data packer. With the exception of stages (1) and (3), all the stages were either heavily modified, or completely redesigned, in order to reduce design complexity. The main aim behind this approach to the implementation of the entropy encoder is to achieve a linear pipe with a small clock period for each stage.

4.1 Brief Outline of the Entropy Encoder

The various steps of the entropy encoder algorithm are briefly outlined as follows. Each block of quantized pixel data consists of one DC coefficient, followed by 63 AC coefficients.

The first step is to calculate ΔDC, which is the difference between the current DC coefficient and the DC coefficient of the previous block. Also, the JPEG algorithm requires that the DC/AC coefficients are decremented by one if the sign of the coefficient is negative.

The next step is to extract the zero runlength count from the stream of the AC coefficients within that block. The block data is thus converted into a stream of AC coefficients with an associated count value, indicating the number of zeros preceding that coefficient. The runlength count is represented as a 4-bit field. When the runlength is greater than 16, two special symbols, ZRL and EOB are used to code the data depending on certain conditions. A zero runlength symbol ZRL (represented in JPEG as [F,0]) is inserted within the data whenever a runlength of 16 zeroes is encountered. The end-of-block symbol EOB (represented in JPEG as [0,0]) is inserted whenever it is detected that the rest of the AC coefficients until the end of the block are zeroes. A 4-bit status field is generated corresponding to each coefficient, which indicates if the data being output is a DC or AC coefficient, ZRL or EOB symbol. The above steps are performed within the zero runlength coder.

Within the category selection circuit, each DC and AC coefficient is associated with a corresponding category depending on the magnitude of the coefficient. The definition

of categories as defined by the JPEG standard is shown in Table 4.1. Each element in the stream of data coming out of the category selection unit consists of coefficients, the corresponding category, the runlength count and the four-bit status. It should be noted that the data stream still contains all 64 coefficients including the streaks of zero coefficients that have been encoded as zero runlength counts. Also if an EOB symbol follows one or more ZRL symbols within the data stream, the ZRL symbols are redundant and must be stripped off the data stream. The above functions are performed within the strip logic.

Category	DC Difference	AC Coefficient
0	0	0
1	-1, 1	-1, 1
2	-3, -2, 2, 3	-3, -2, 2, 3
3	-7, …, -4, 4, …, 7	-7, …, -4, 4, …, 7
4	-15, …, -8, 8, …, 15	-15, …, -8, 8, …, 15
5	-31, …, -16, 16, …, 31	-31, …, -16, 16, …, 31
6	-63, …, -32, 32, …, 63	-63, …, -32, 32, …, 63
7	-127, …, -64, 64, …, 127	-127, …, -64, 64, …, 127
8	-255, …, -128, 128, …, 255	-255, …, -128, 128, …, 255
9	-511, …, -256, -256, …, 511	-511, …, -256, -256, …, 511
10	-1023, …, -512, 512, …, 1023	-1023, …, -512, 512, …, 1023
11	-2047, …, -1024, 1024, …, 2047	

Table 4.1:JPEG Category Definitions

During the next step, each data element consisting of {AC/DC coefficient, runlength count, category, status} output by the strip logic is converted into a corresponding element: {AC/DC coefficient, category, Huffman code, Huffman code length}. The Huffman code is selected based on the runlength count, category and status fields. The sets of Huffman codes are pre-stored in a table and can be changed depending on the application. The category and the Huffman code length fields are used in the data packer unit to pack the variable length compressed data (comprised of the DC/AC coefficient and the Huffman code) into a stream of fixed length compressed data units to be output by the compression chip.

The implementation of each module within the entropy encoder architecture is described below.

4.2 Zero-Runlength Coder

The zero-runlength coder module performs the functions as described in the earlier part of this section. The module consists of three stages and thus a latency of 3 cycles. The first stage consists of logic for computing ΔDC while the second stage derives the runlength count and the third stage is used for decrementing negative coefficients. The various stages of the zero runlength coder are shown in Figure 4.2.

4.2.1 Architectural Description

Before we describe the flow of data through this stage, it would help to explain the working of the control logic. The control logic block is primarily responsible for two things: differentiating between DC and AC coefficients, and maintaining the zero-counter. The control logic is also responsible for generating signals that indicate

conditions such as the occurrence of 16 consecutive zeros, end-of-block, and whether the output data value represents a DC or an AC coefficient.

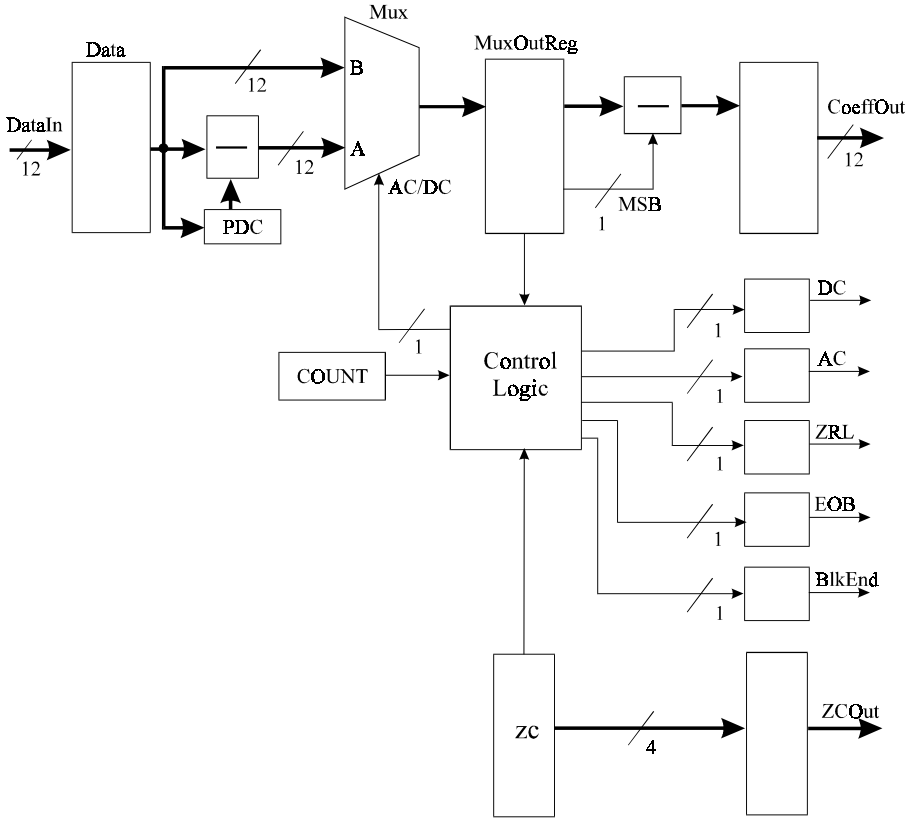


Figure 4.2: Zero-runlength coder

The control logic differentiates between DC and AC coefficients by use of a 'coefficient counter' that counts to a maximum value of 63, and then resets. The zero value in this counter means that a DC coefficient is present at the input. An AC coefficient is present otherwise. This counter increments every time new data is made available as input to the runlength coder.

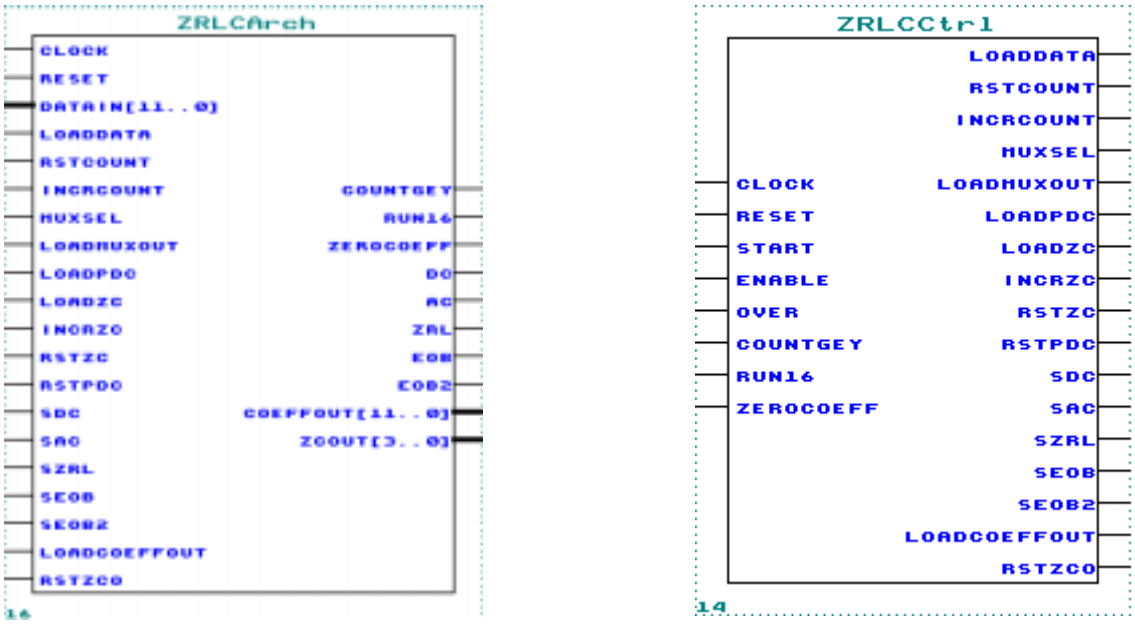
As can be seen, the design consists of three stages. The runlength coder receives quantized input data from the Zigzag Reordering buffer. The value of the DC

coefficient of the previous 8 x 8 block is stored in the 'pDC' register. In case this is the first block if data, pDC contains zero. The data received from the input latch is routed along two paths to a multiplexer - one path for AC coefficients and the other for DC. Based on the contents of the coefficient counter, the control logic selects the appropriate input of the multiplexer. In the case of a DC coefficient, ΔDC appears at the B input to the multiplexer, which is selected by the control logic. At the next clock pulse, the value of the current DC coefficient is placed in the pDC register.

The values in the coefficient counter and the intermediate latch are used to generate the DC, AC and EOB signals. The AC signal is set high whenever both the current coefficient and the coefficient counter have non-zero values. Thus no signals are set for zero-valued coefficients. The contents of the intermediate latch and the zero-runlength counter are used to generate the ZRL signal. The zero-counter register counts the number of consecutive zero valued coefficients appearing in the intermediate latch. When the count reaches 16 consecutive zeroes, the ZRL signal is set to high. The EOB signal line is set to high if the coefficient counter reaches a count of 63, and the corresponding coefficient value is zero. Also, when the EOB signal is generated, the runlength counter is reset. This is to indicate that the current block has been completely runlength encoded and a new block is about to start. It is important to note that these four status signals are mutually exclusive.

The BLKEND signal is used to identify the end of a block, regardless of the current coefficient value. This signal is used in subsequent stages. Before the final output latch is a simple mechanism for decrementing the negative coefficients in the data.

4.2.2 Module Block Diagram



For description of ports and the verilog source code, refer Appendix A.

4.3 Category Selection Unit

This stage of the JPEG process is implemented using our original design. Category selection is defined in the JPEG compression standard as shown in Table 4.1.

A straightforward implementation of category selection would require storing the ranges in memory and comparing the input data with those pre-stored values which requires complex address decoding and control logic. However, the table memory can be avoided and the entire category selection can be achieved with a simple combinational circuit. This circuit operates like an encoder that converts the given coefficient into the corresponding category in a single clock cycle. The circuit is given in Figure 4.3. However, it should be noted that the negative coefficients must be

decremented by one before applying the conversion logic as per the JPEG standard. This decrementing is performed by the last stage of the zero runlength coder in our implementation.

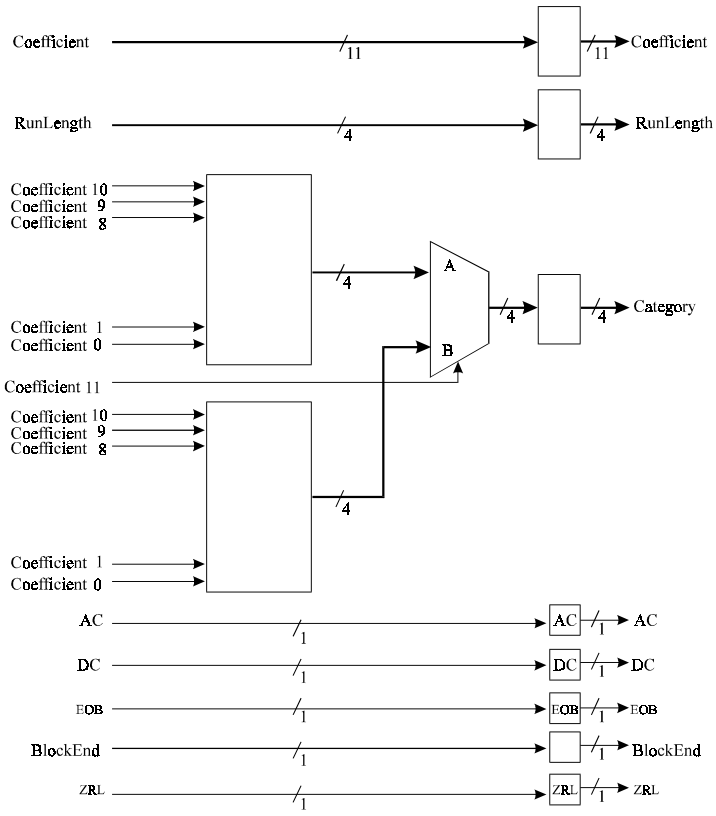


Figure 4.3: Category Selection Unit

4.3.1 Architectural Description

As can be seen from the schematic, the category selection unit has been organized in the form of two modified priority encoders, whose outputs are connected to a multiplexer. The outputs are selected on the basis of the MSB of the coefficient.

4.3.2 Module Block Diagram

For description of ports and the verilog source code, refer Appendix A.

4.4 Strip Logic

The strip logic shown in Figure 4.4 is a slightly modified version of that presented in [2] and consists of five stages instead of four. The two main aims of this stage are to discard the zero-valued coefficients, as well as the redundant ZRL symbols occurring before an EOB symbol.

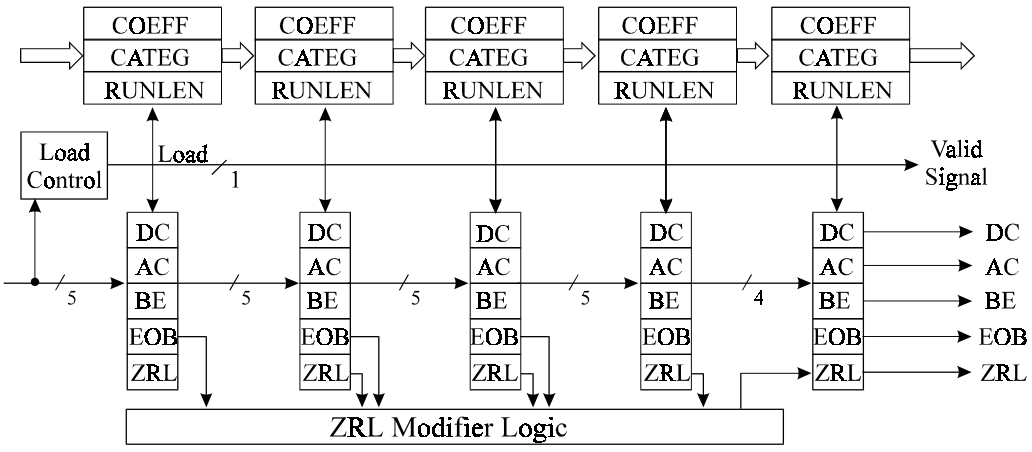


Figure 4.4: Strip Logic structure

Each stage has three registers to hold the coefficient, runlength count and category fields corresponding to a data element output by the category selection unit and a set of

1-bit registers to hold the corresponding status. The status bits are decoded and used to strip the zero-valued coefficients and also to strip off the ZRL symbols that precede an EOB symbol. It should be noted that there could be a maximum of three ZRL symbols preceding an EOB symbol. The strip logic acts as a five-stage buffer through which the compressed data elements, after the removal of zero coefficients travel, before being forwarded to the Huffman encoder. The valid bit signal is set to high whenever valid data is being output by the strip logic for Huffman encoding. It must be noted that the ZRL bit needs to be reset whenever a ZRL symbol has been deleted from the data stream.

4.4.1 Architectural Description

The inputs to the strip logic are the AC/DC coefficients, the category of the coefficient, the corresponding zero runlength counts, and the five status signals generated at the runlength coder. The operation of this stage proceeds in the following manner: the load control logic is simply the logical ORing of the DC, AC, ZRL, and EOB signals. Thus whenever a zero coefficient appears without the ZRL signal being set, it is discarded. Upon the occurrence of nonzero coefficients, or ZRL symbols, the contents of each stage propagate to the next. The second task of discarding redundant ZRL symbols is performed in this manner: the first of a series of consecutive ZRL symbols (at most 3) propagates through to the fourth stage. If an EOB signal follows the last ZRL symbol in the series, all the preceding ZRL symbols will be ignored by the ZRL Modifier Logic.

4.4.2 Module Block Diagram

For description of ports and the verilog source code, refer Appendix A.

4.5 Huffman Encoder Module

The Huffman encoder module that we have implemented is a heavily modified version of the design suggested in [2]. It consists of Huffman code tables stored in random access memory modules and logic for replacing the category, runlength count pairs with the corresponding Huffman codes. The table is accessed by using the {runlength, category} pair for addressing. The input data passes through each of the two stages, and depending on the address, the corresponding Huffman code and the code length are output. The hardware organization is shown in Figure 4.5.

4.5.1 Architectural Description

The Huffman encoder receives category, runlength, coefficient, and status signal inputs from the strip logic. The encoder also receives the 'valid input' signal generated by the strip logic. The two AC and DC memory modules have as inputs the {runlength} and {category, runlength} data fields respectively, and based on these values, the modules output the appropriate Huffman codes. The output of these

modules is a 20 bit value where the most significant 4 bits represent the actual length of the Huffman code present in the remaining 16 bits.

The outputs are connected to the inputs of a multiplexer. The multiplexer's input select pins are connected to the DC status signal, and the ORed result: AC+ZRL+EOB. Therefore in the case of a DC input signal, the Huffman code generated by the DC memory module is output to the 20 bit output latch, in the case of either an AC, ZRL or EOB signal, the AC memory module output is chosen.

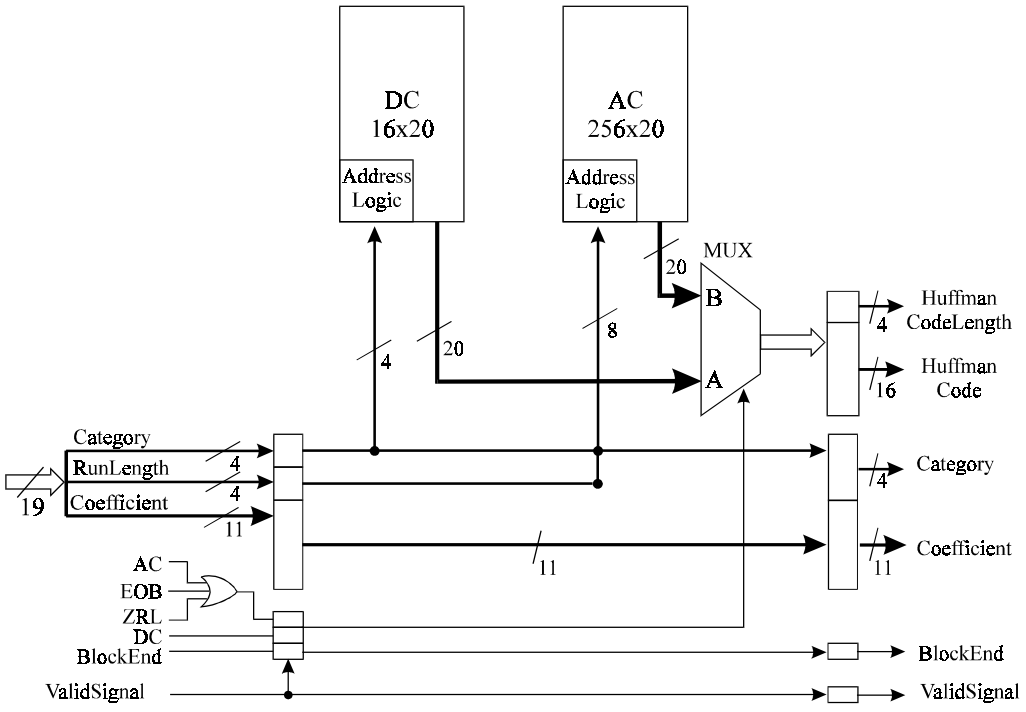


Figure 4.5: The Huffman Encoder Module

The control logic has two purposes: determining whether valid data is present at the inputs, and generating the valid output signal. Valid data is identified by the use of the status signals and the 'valid input' signal. In case valid data is present at the input

terminals, the control logic enables the output latches to receive the data, and generates the valid output signal.

4.5.2 Module Block Diagram

For description of ports and the verilog source code, refer Appendix A.

4.6 The Data Packer

The Data Packer unit shown in Figure 4.6 is a heavily modified version of the one suggested in [2]. It is used to convert variable length compressed data into fixed length compressed data stream. The logic consists of registers A and B, two left-shift units, a masking logic unit, two ORing logic units, and control logic, which includes the two registers LENGTH and ENDP.

4.6.1 Architectural Description

The data packer works as follows. The Huffman code first enters the first left shifter. Here it is variably left shifted a number of times corresponding to the category of the coefficient. At the same time the coefficient enters the masking logic, whose function it is to set to zero the unnecessary bits of the coefficient. These unnecessary bits are determined using the category of the coefficient. The Huffman code is then bit-

aligned with the masked category data, the result being placed in Register A at the clock pulse. It should be noted that the total length of the result cannot exceed 27 bits, as the maximum length of the Huffman code is 16 bits and that of the coefficient is 11 bits. The LENGTH register contains the value 'category + code-length + 1'. 1 is added since the values of code-length start at zero, for a Huffman code 1 bit long. Another important point is that that all operations performed up till the bit alignment are performed combinationally.

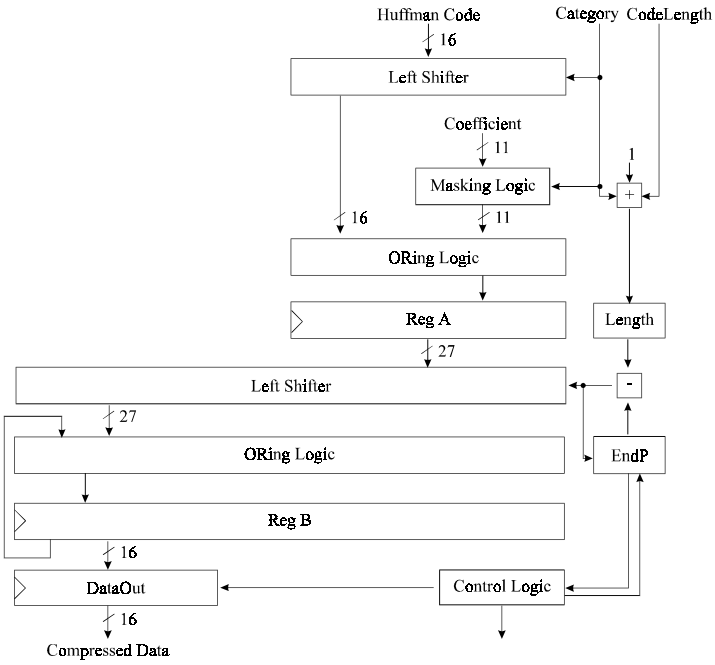
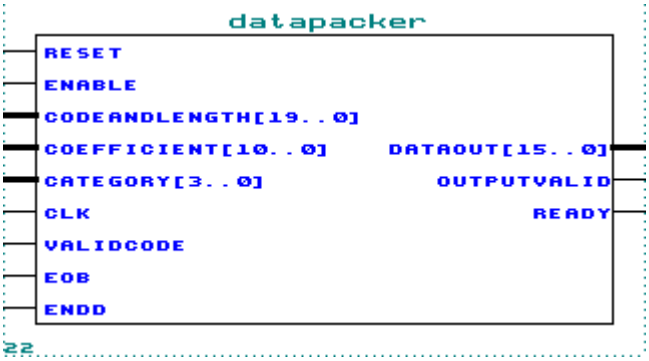


Figure 4.6: Data Packer

The ENDP register act as a pointer to positions in the B register and is used to hold the number of shifts needed to pack together successive data received from A. If B is empty, the ENDP register contains 48, which represents the length of the B register, otherwise it points to the bit position next to the least significant useful bit. At the second left shifter, the contents of A are left shifted ENDP – LENGTH times, before

being placed in B. This bit aligns the {Huffman code, masked category} combination to the right of any previously present contents of B. This number of shifts is then placed into the ENDP register. At the nest clock cycle, the next contents of A are shifted in the same manner, so that the data is concatenated to the right of the present contents of B. This occurs every clock cycle, until the value of ENDP falls below 33, at which point the most significant 16 bits of B are transferred into the 'DataOut' register, the contents of B are left shifted 16 times, and the ENDP register is incremented by 16. The process thus continues until the BLKEND signal is received, at which point, the contents of B are transferred to 'Data Out' irrespective of the value of ENDP. If the value of ENDP at this point is greater than 33, extra 1's need to be padded to the left of the useful bits in the 'Data Out' register.

4.6.2 Module Block Diagram



For description of ports and the verilog source code, refer Appendix A.

5 Project Module Hierarchy

Figure 5 illustrates the modular hierarchy of our project implementation, which is self-explanatory. The diagram portrays a top-down design scheme.

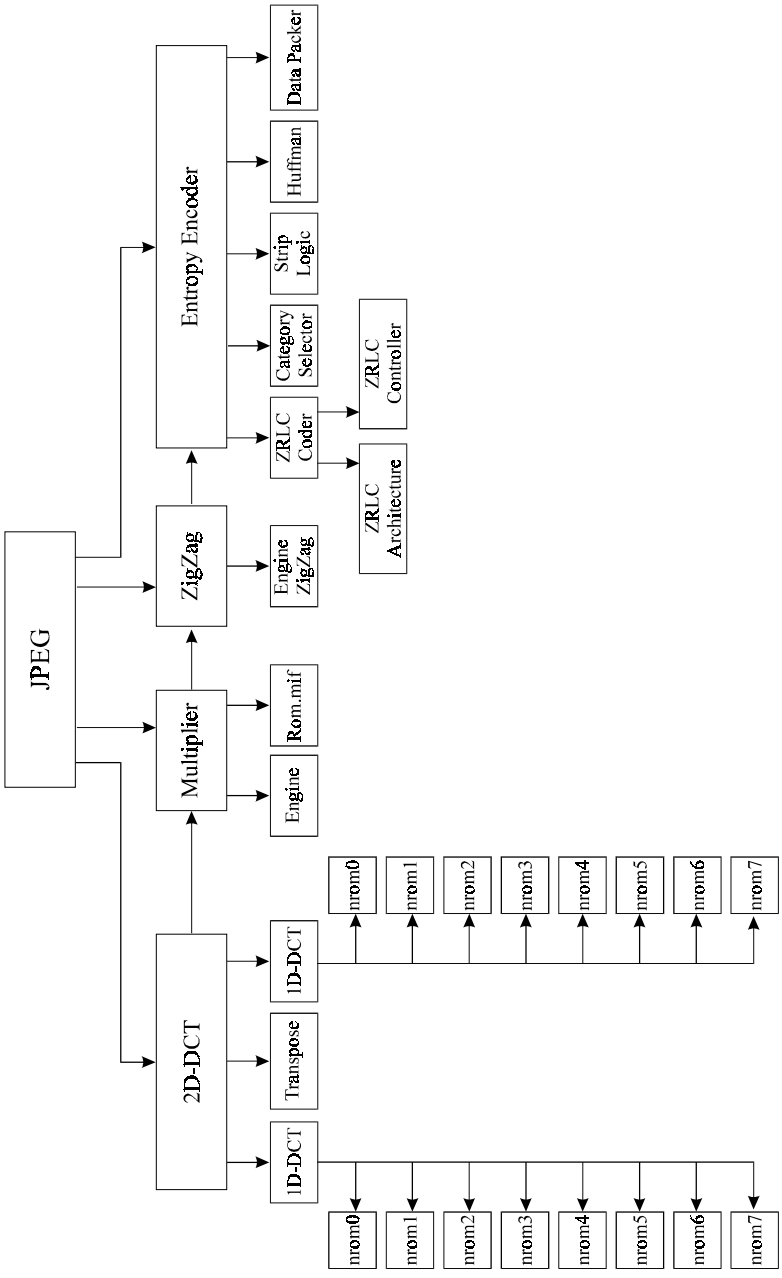
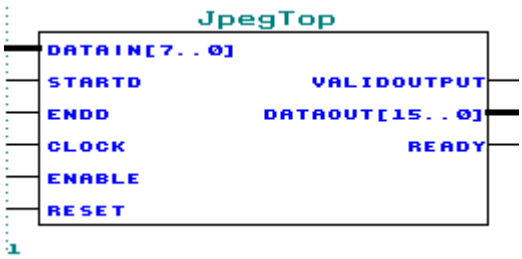


Figure 5.1: Program Module Heirarchy

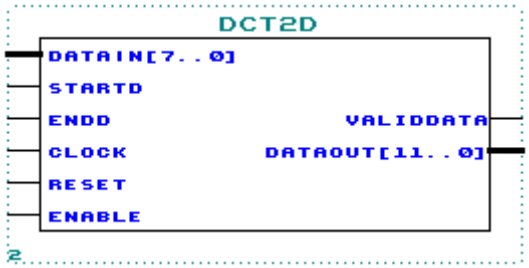
In this section we discuss the modules that provide timing and control signals to the JPEG pipeline discussed before.

5.1 The Top Level Controller - JPEG



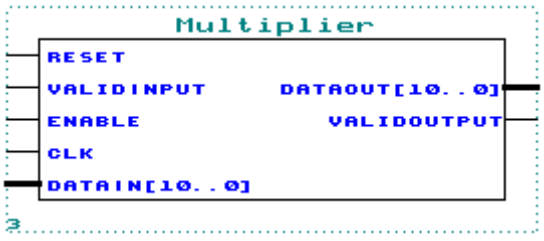
This module instantiates all four second level controllers. When the encoder is ready to receive pixel data, this module enables the READY signal. The need for this is occurs when the last pixel data from one image has entered the pipeline, and data from another image is ready to be streamed. The JPEG module disables the READY signal until the pipeline has been completely flushed of the previous image's data.

5.2 The 2-D DCT Controller



This module instantiates both the lower level 1-D DCT modules, as well as the Transpose buffer. It provides timing and control signals to these modules, handling pipeline flush operations, signaling availability of valid data, and enabling and disabling the outputs of these modules depending on the validity of the data.

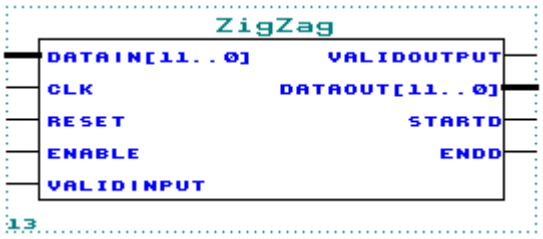
5.3 Multiplier Controller



This module instantiates both the Quantization table ROM, as well as the Wallace-tree multiplier. Upon being signaled of the availability of valid data by the DCT controller, the multiplier-controller enables Wallace-tree multiplier, and starts generating the input addresses to the Quantization table ROM. The ROM then provides the quantization coefficients to the Wallace-tree multiplier. Upon receiving

the data-invalid signal from the DCT controller, it manages the pipeline flush operation for the multiplier pipeline.

5.4 Zigzag Controller



This module instantiates the zigzag reordering module, and provides timing and control to it. It also handles pipeline flush operations, as well as signaling availability of valid data, and enabling and disabling the outputs of these modules, depending on the validity of the data. This controller receives its valid-data signals from the Multiplier-controller. It also manages the pipeline flush operation for the zigzag module.

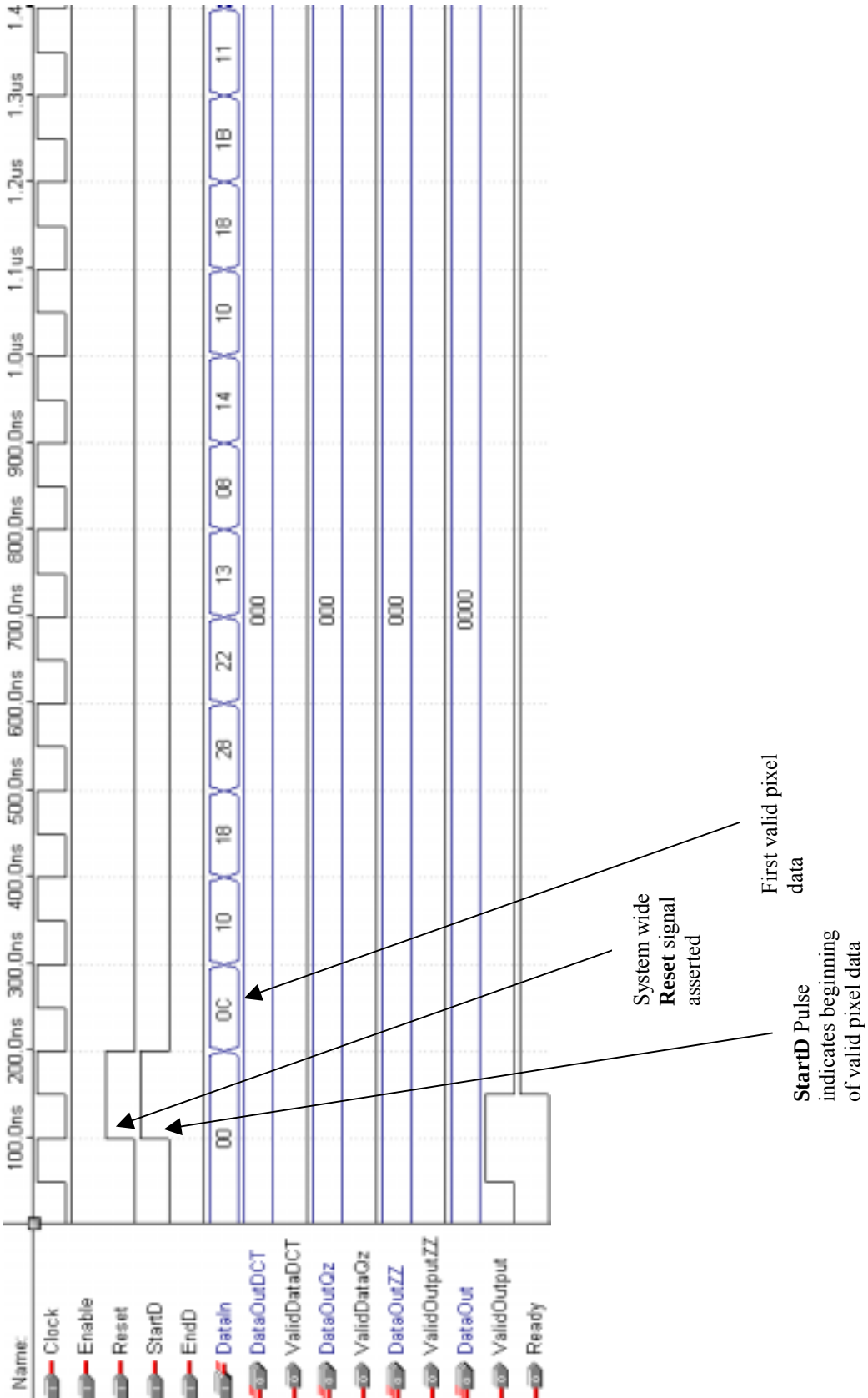
5.5 The Entropy Encoder Controller

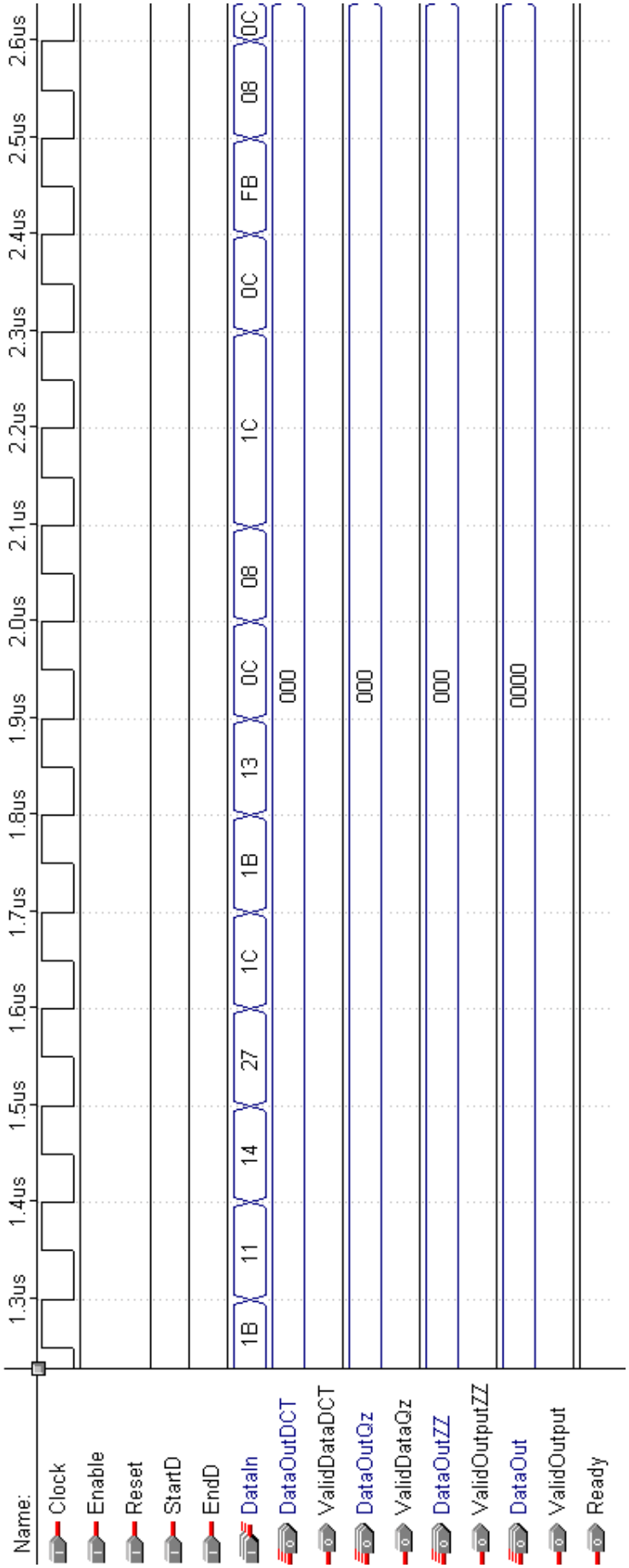
This module instantiates the 5 Entropy Encoder component modules. It also instantiates another module that is specifically responsible for handling pipeline flush operations throughout the Entropy Encoder

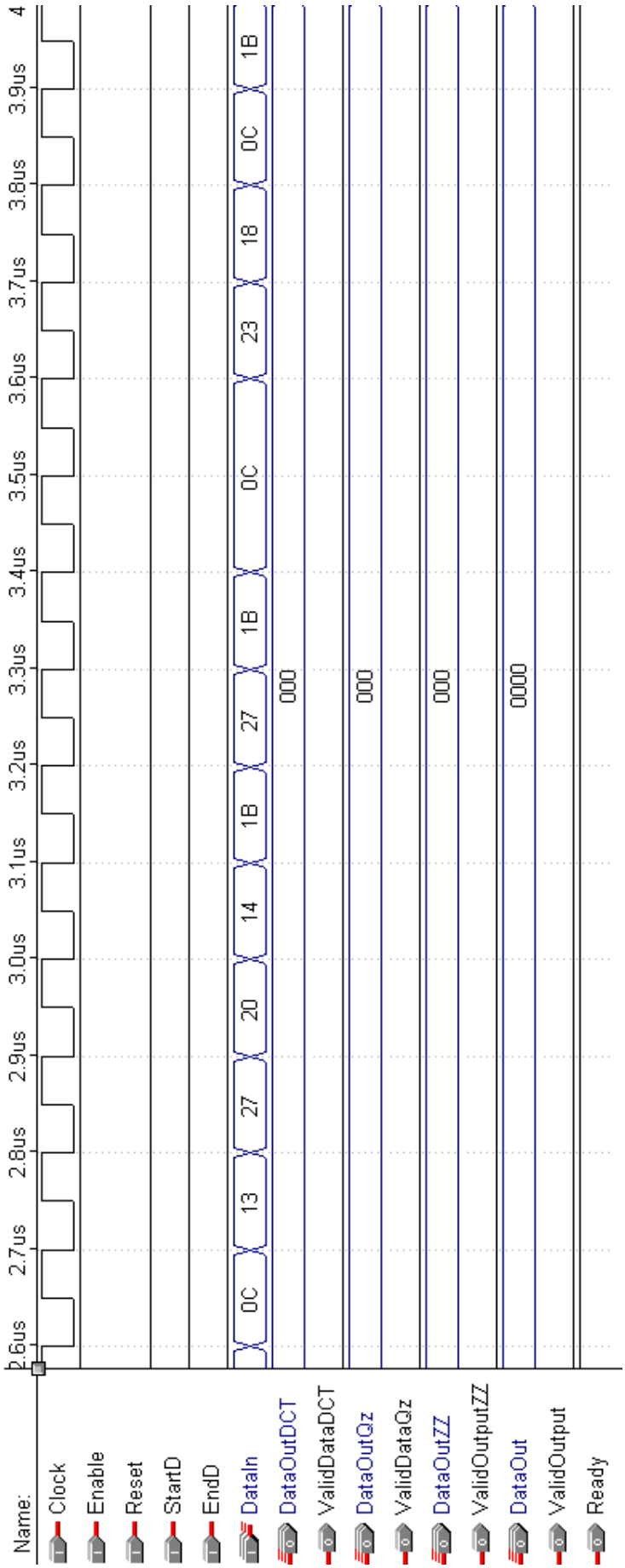
For all of these controllers, one important point to note is that control signals indicating valid or invalid data are propagated through from one controller to the next, in order to synchronize the pipeline properly, and manage the individual flush operations, in order to prevent data from being frozen in the pipeline.

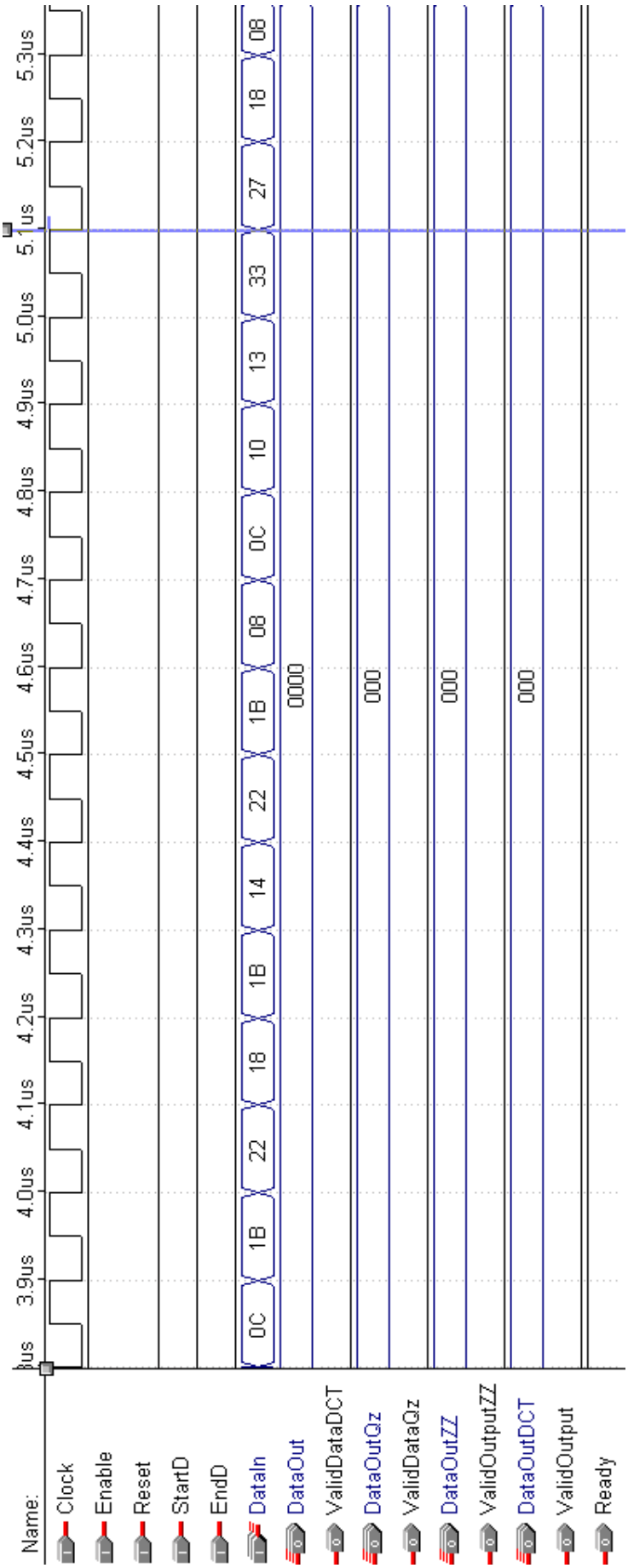
6 Simulation Results

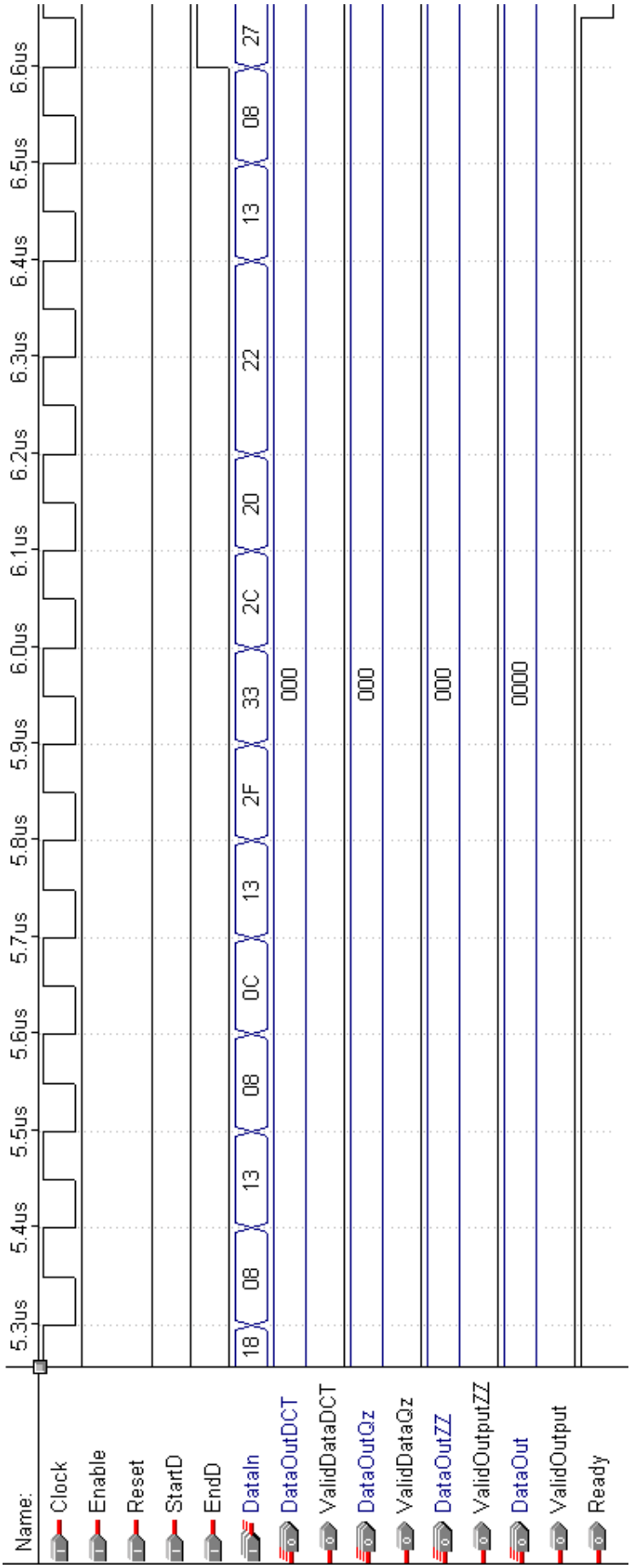
In this section are presented the simulation results of the JPEG process, when applied to one 8 X 8 block of pixels. Due to space constraints, it was not possible to include any more blocks.

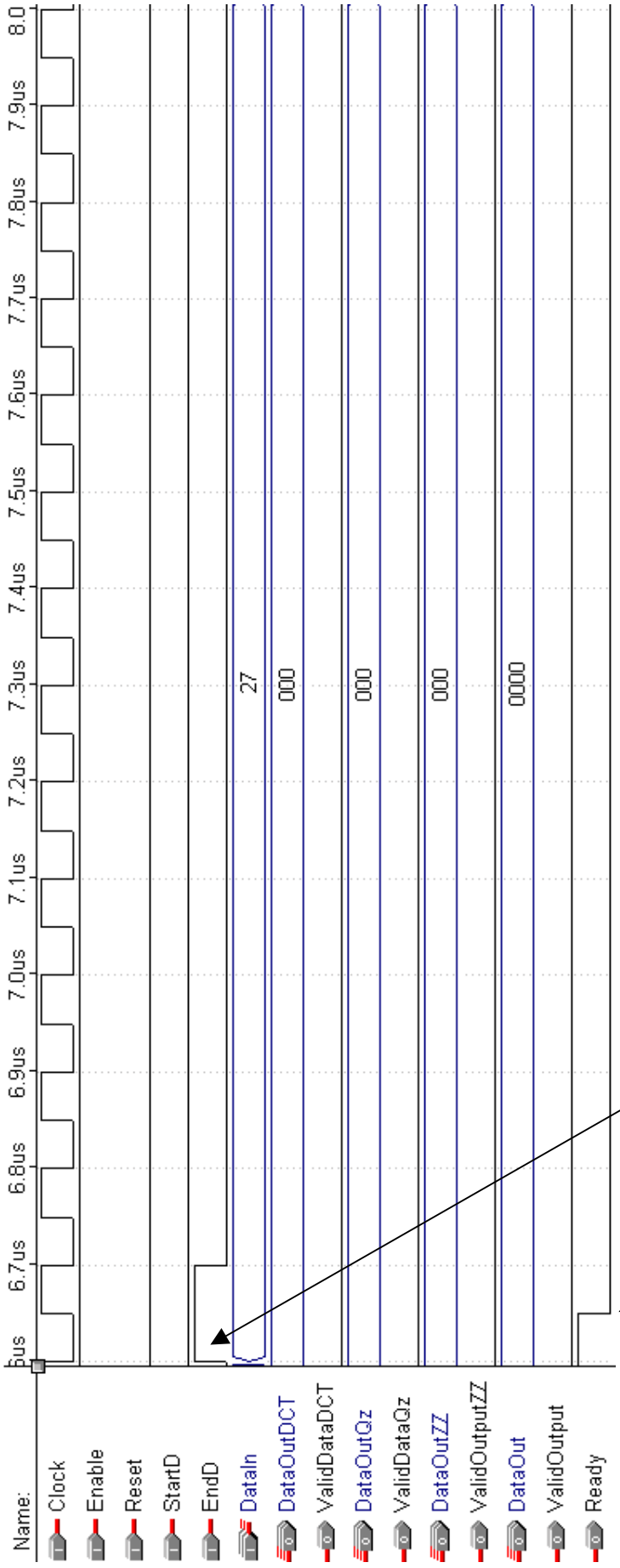


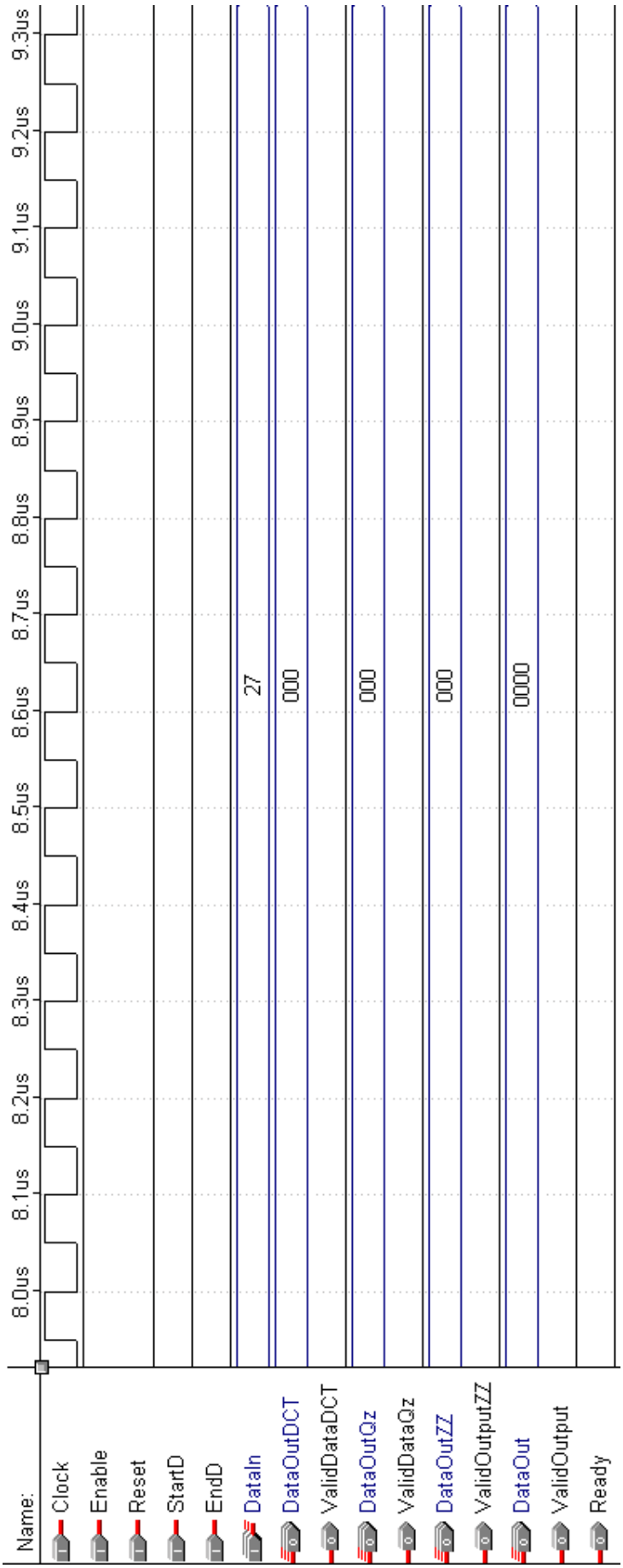

Name:
Clock
Enable
Reset
StartD
EndD
DataIn
DataOutDCT
ValidDataDCT
DataOutQz
ValidDataQz
DataOutZZ
ValidOutputZZ
DataOut
ValidOutput
Ready
1.3us
1.4us
1.5us
1.6us
1.7us
1.8us
1.9us
2.0us
2.1us
2.2us
2.3us
2.4us
2.5us
2.6us
1B
11
14
27
1C
1B
13
0C
08
1C
0C
FB
08
0C
000
000
000
0000

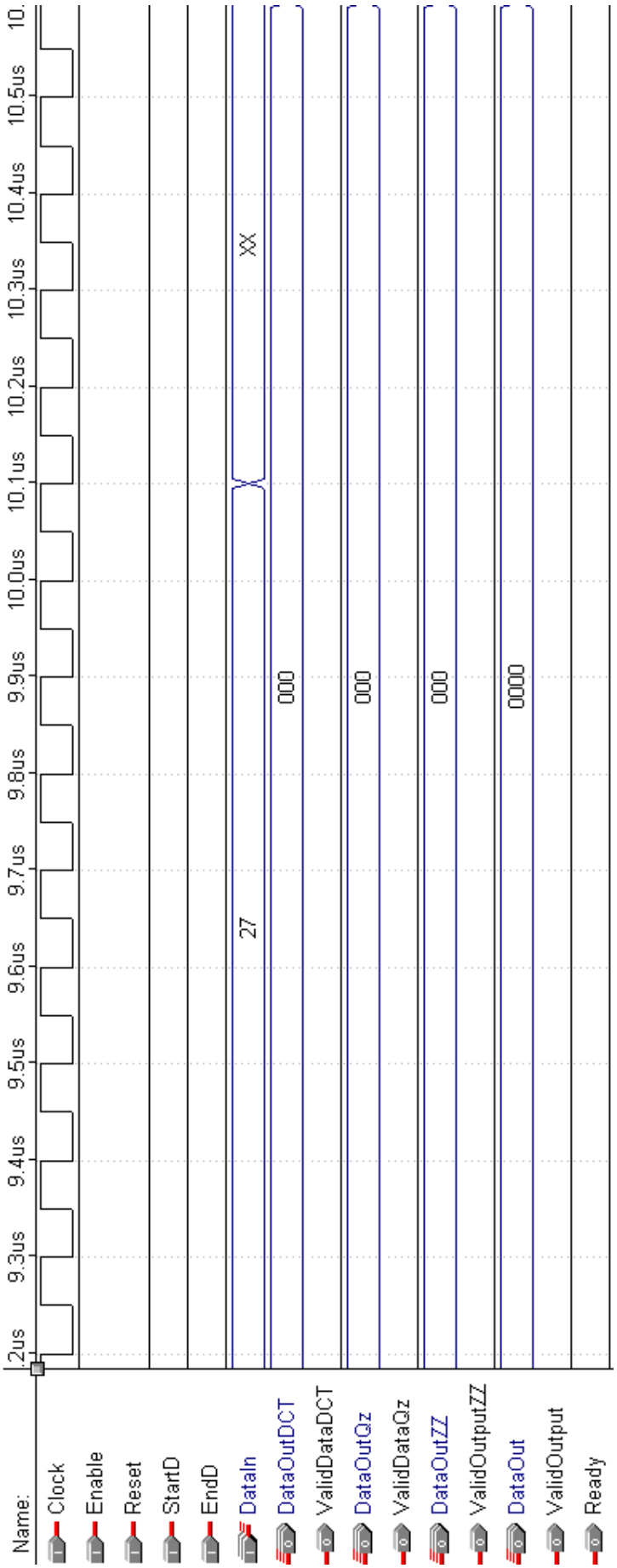

Name:
Clock
Enable
Reset
StartD
EndD
DataIn
DataOutDCT
ValidDataDCT
DataOutQz
ValidDataQz
DataOutZZ
ValidOutputZZ
DataOut
ValidOutput
Ready
2.6us
2.7us
2.8us
2.9us
3.0us
3.1us
3.2us
3.3us
3.4us
3.5us
3.6us
3.7us
3.8us
3.9us
4
0C
13
27
20
14
1B
27
1B
0C
23
18
0C
1B
000
000
000
0000

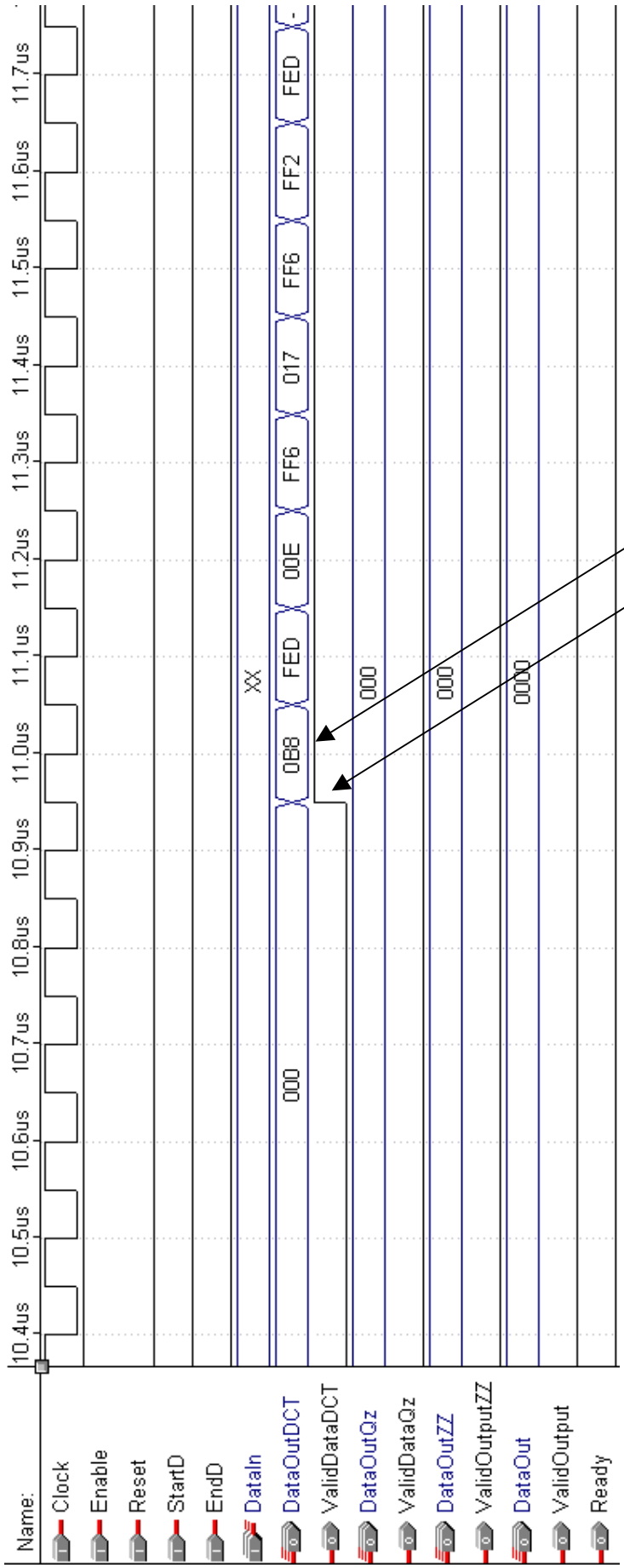

Name:
Clock
Enable
Reset
StartD
EndD
DataIn
DataOut
ValidDataDCT
DataOutQz
ValidDataQz
DataOutZZ
ValidOutputZZ
DataOutDCT
ValidOutput
Ready
0us
3.9us
4.0us
4.1us
4.2us
4.3us
4.4us
4.5us
4.6us
4.7us
4.8us
4.9us
5.0us
5.1us
5.2us
5.3us
0C
1B
22
18
1B
14
22
1B
08
0C
10
13
33
27
18
08
0000
000
000
000


Name:
Clock
Enable
Reset
StartD
EndD
DataIn
DataOutDCT
ValidDataDCT
DataOutQz
ValidDataQz
DataOutZZ
ValidOutputZZ
DataOut
ValidOutput
Ready
5.3us
5.4us
5.5us
5.6us
5.7us
5.8us
5.9us
6.0us
6.1us
6.2us
6.3us
6.4us
6.5us
6.6us
18
08
13
08
0C
13
2F
33
2C
20
22
13
08
27
000
000
000
0000

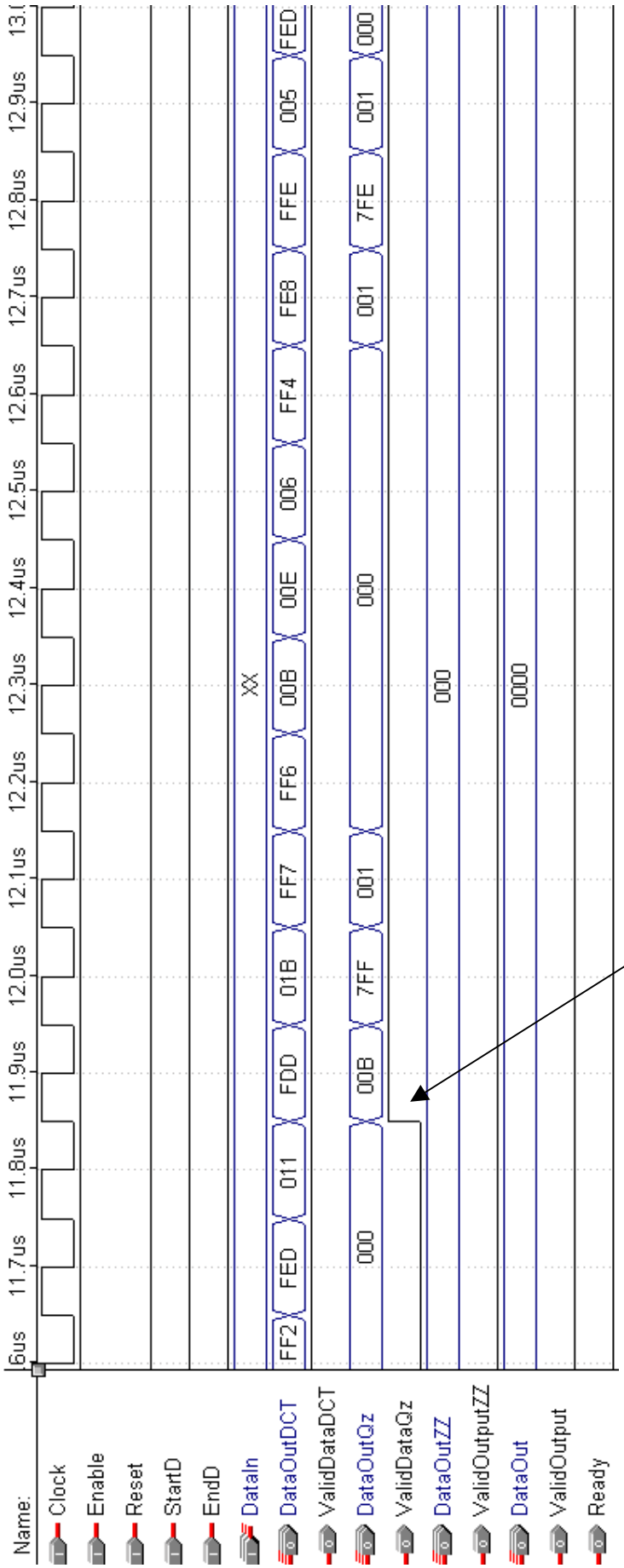

Name:
Clock
Enable
Reset
StartD
EndD
DataIn
DataOutDCT
ValidDataDCT
DataOutQz
ValidDataQz
DataOutZZ
ValidOutputZZ
DataOut
ValidOutput
Ready
6us
6.7us
6.8us
6.9us
7.0us
7.1us
7.2us
7.3us
7.4us
7.5us
7.6us
7.7us
7.8us
7.9us
8.0
27
000
000
000
0000
EndD Pulse indicates end of valid pixel data
Ready signal asserted low to indicate that new data cannot be input until the pipeline is completely flushed

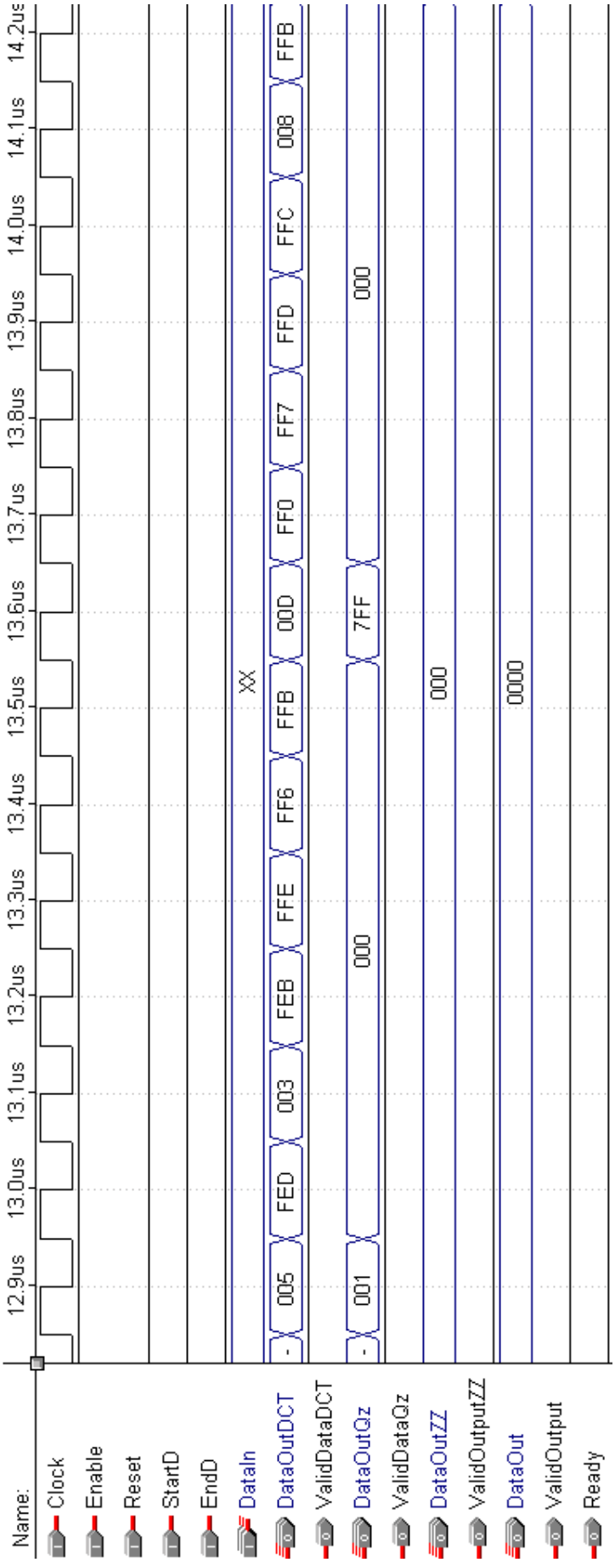

Name:
Clock
Enable
Reset
StartD
EndD
DataIn
DataOutDCT
ValidDataDCT
DataOutQz
ValidDataQz
DataOutZZ
ValidOutputZZ
DataOut
ValidOutput
Ready
8.0us
8.1us
8.2us
8.3us
8.4us
8.5us
8.6us
8.7us
8.8us
8.9us
9.0us
9.1us
9.2us
9.3us
27
000
000
000
0000

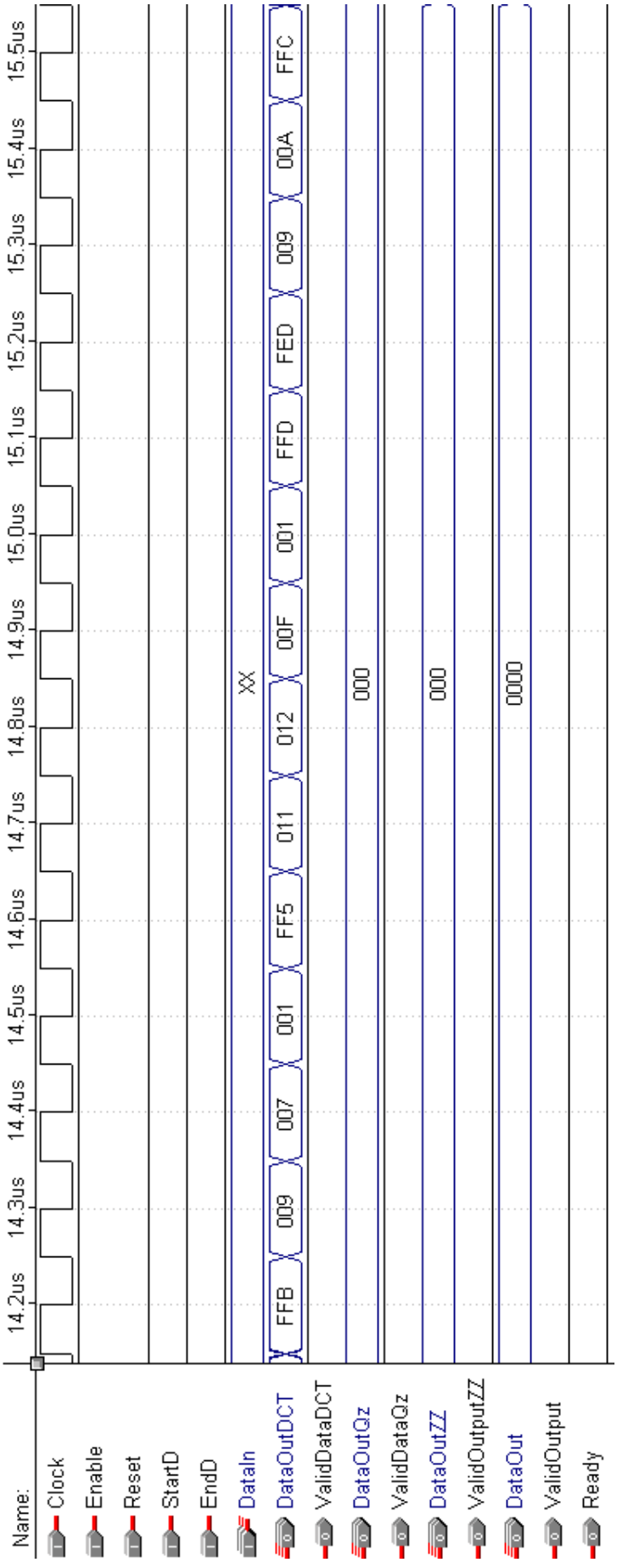

Name:
Clock
Enable
Reset
StartD
EndD
DataIn
DataOutDCT
ValidDataDCT
DataOutQz
ValidDataQz
DataOutZZ
ValidOutputZZ
DataOut
ValidOutput
Ready
.2us
9.3us
9.4us
9.5us
9.6us
9.7us
9.8us
9.9us
10.0us
10.1us
10.2us
10.3us
10.4us
10.5us
10.
27
XX
000
000
000
0000

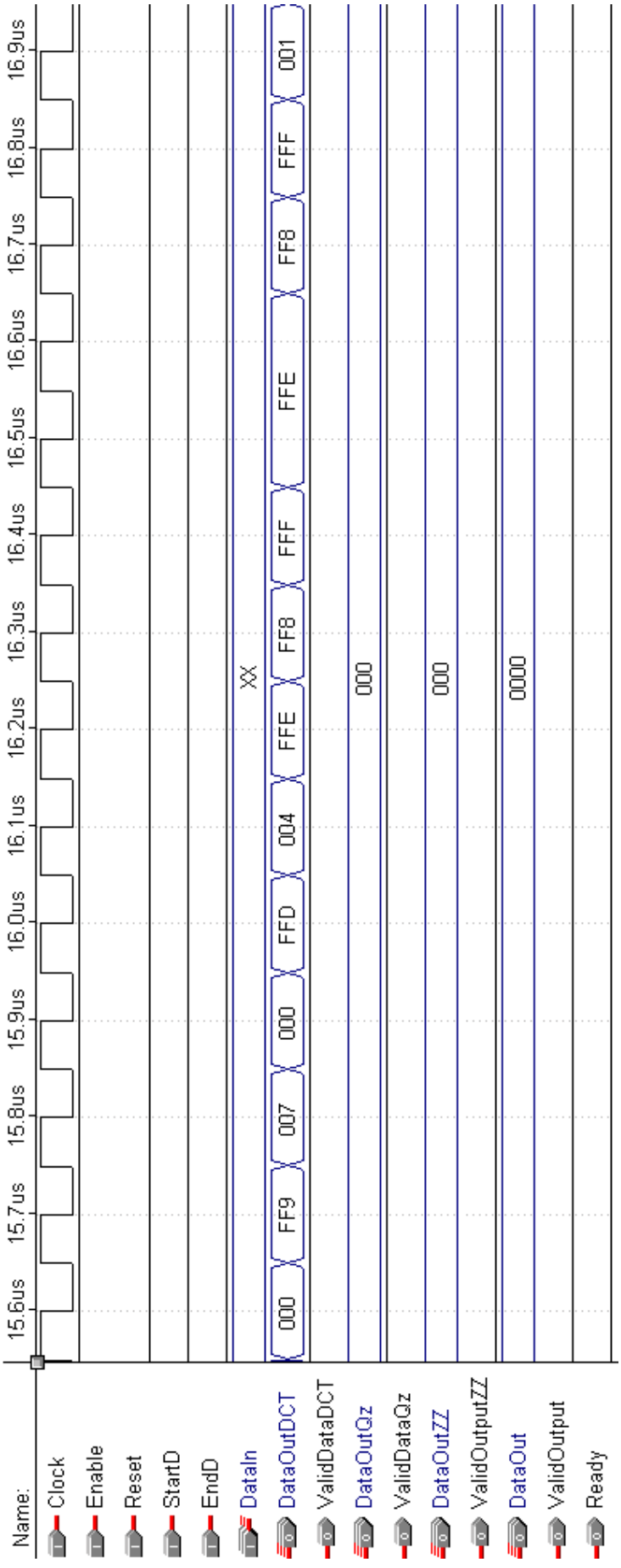


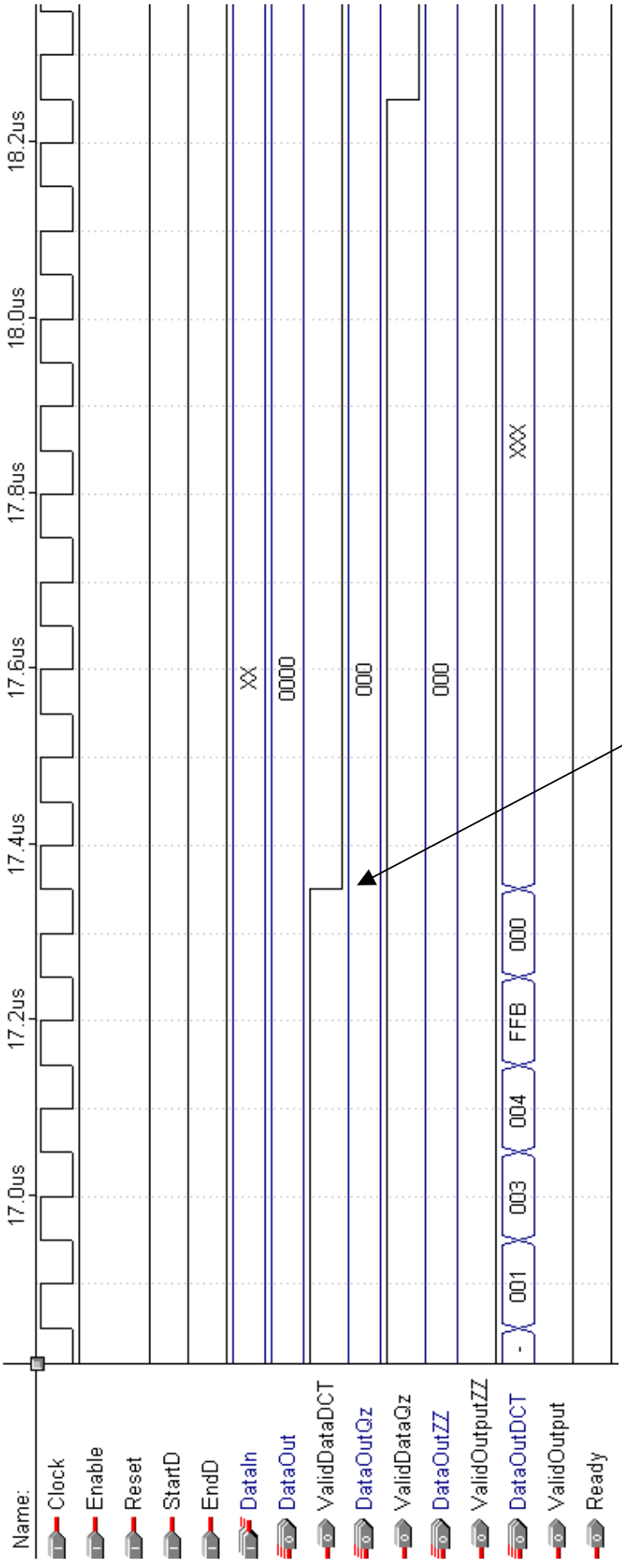
Note: All signals labeled "ValidData*XXX*" are signals that indicate the appearance of valid data at the output of the module specified by the *XXX*.

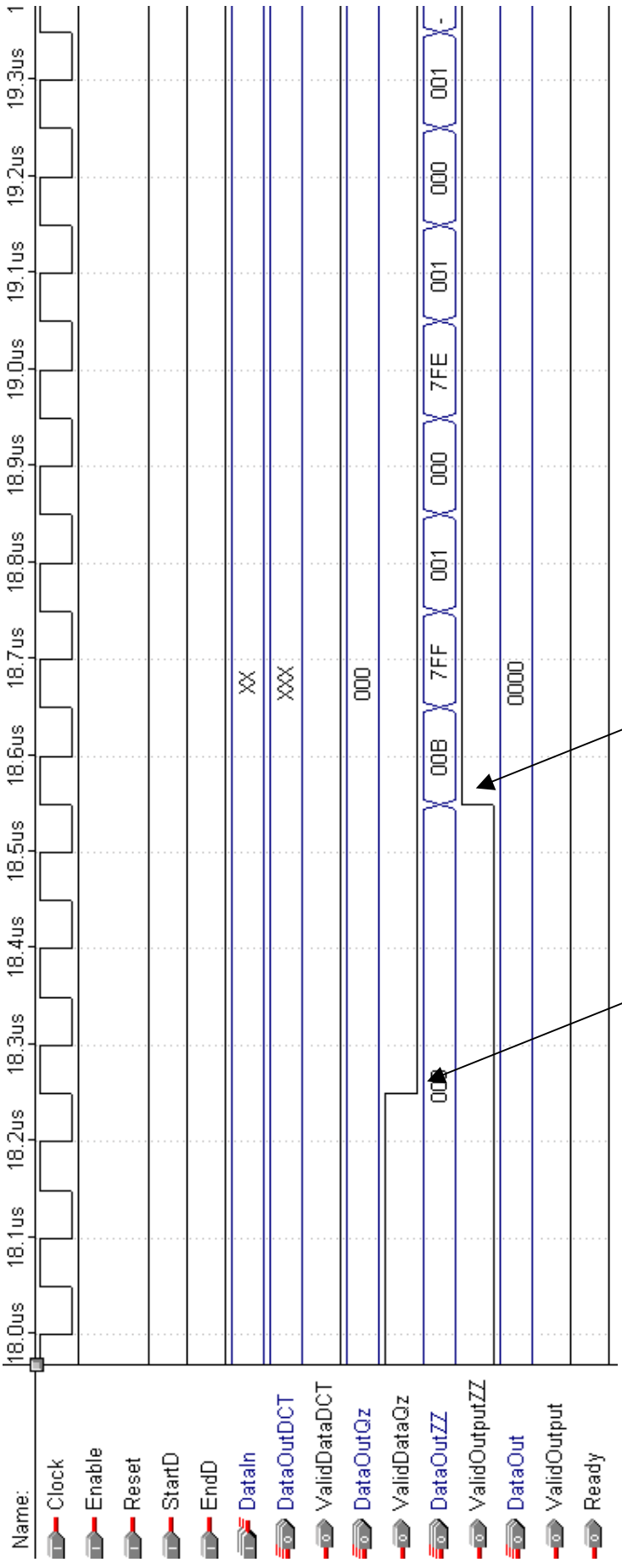

Name:
Clock
Enable
Reset
StartD
EndD
DataIn
DataOutDCT
ValidDataDCT
DataOutQz
ValidDataQz
DataOutZZ
ValidOutputZZ
DataOut
ValidOutput
Ready
6us
11.7us
11.8us
11.9us
12.0us
12.1us
12.2us
12.3us
12.4us
12.5us
12.6us
12.7us
12.8us
12.9us
FF2
FED
011
FDD
01B
FF7
FF6
00B
00E
006
FF4
FE8
FFE
005
FED
000
00B
7FF
001
000
001
7FE
001
000
000
0000
XX
After a latency of 8 clock cycles the first quantized result is output

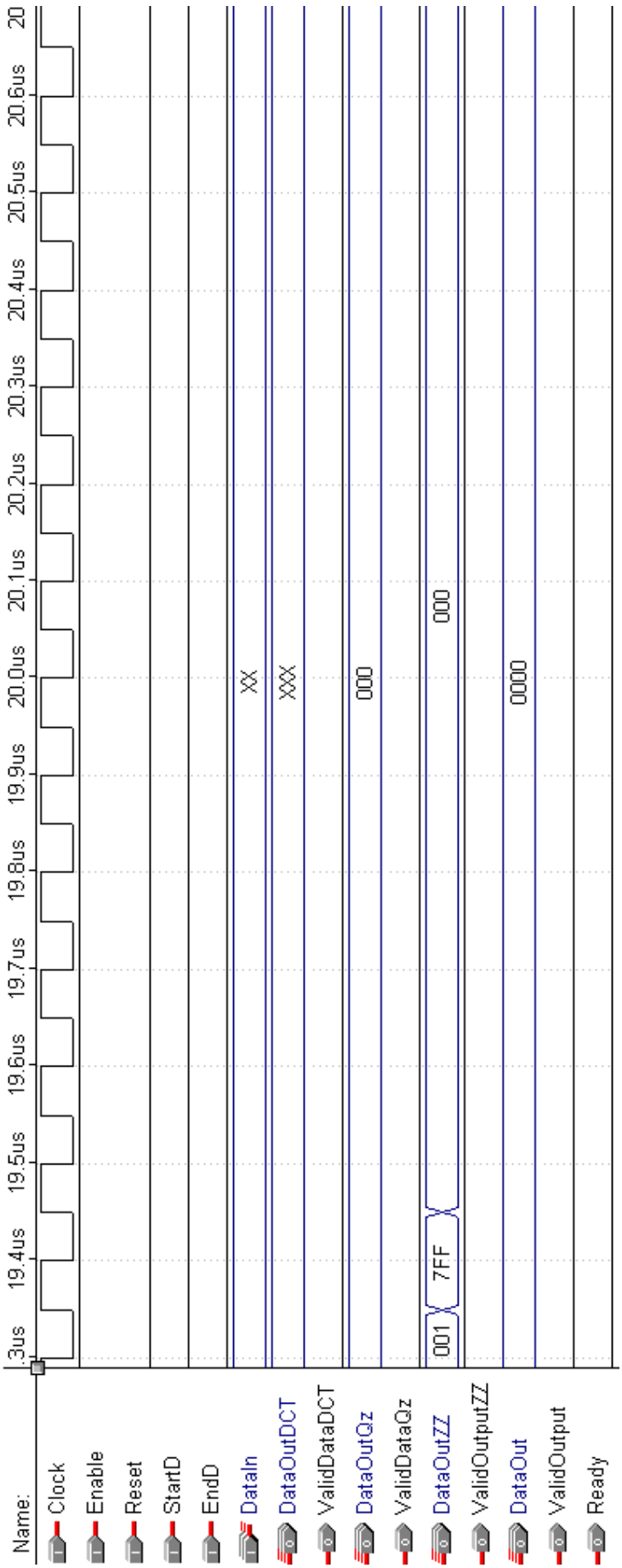

Name:
Clock
Enable
Reset
StartD
EndD
DataIn
DataOutDCT
ValidDataDCT
DataOutQz
ValidDataQz
DataOutZZ
ValidOutputZZ
DataOut
ValidOutput
Ready
12.9us
13.0us
13.1us
13.2us
13.3us
13.4us
13.5us
13.6us
13.7us
13.8us
13.9us
14.0us
14.1us
14.2us
XX
005
FED
003
FEB
FFE
FF6
FFB
00D
FF0
FF7
FFD
FFC
008
FFB
001
000
7FF
000
000
000
0000

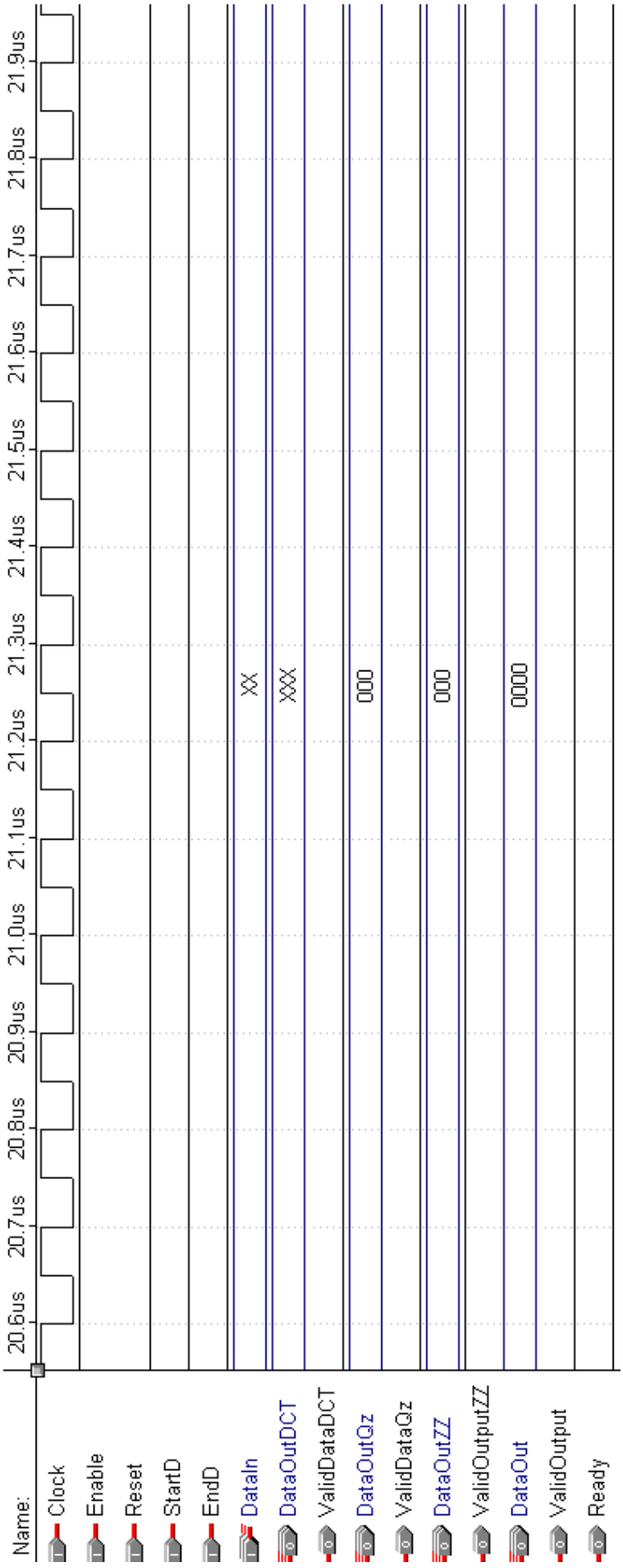

Name:
Clock
Enable
Reset
StartD
EndD
DataIn
DataOutDCT
ValidDataDCT
DataOutQz
ValidDataQz
DataOutZZ
ValidOutputZZ
DataOut
ValidOutput
Ready
14.2us
14.3us
14.4us
14.5us
14.6us
14.7us
14.8us
14.9us
15.0us
15.1us
15.2us
15.3us
15.4us
15.5us
FFB
009
007
001
FF5
011
012
00F
001
FFD
FED
009
00A
FFC
XX
000
000
0000

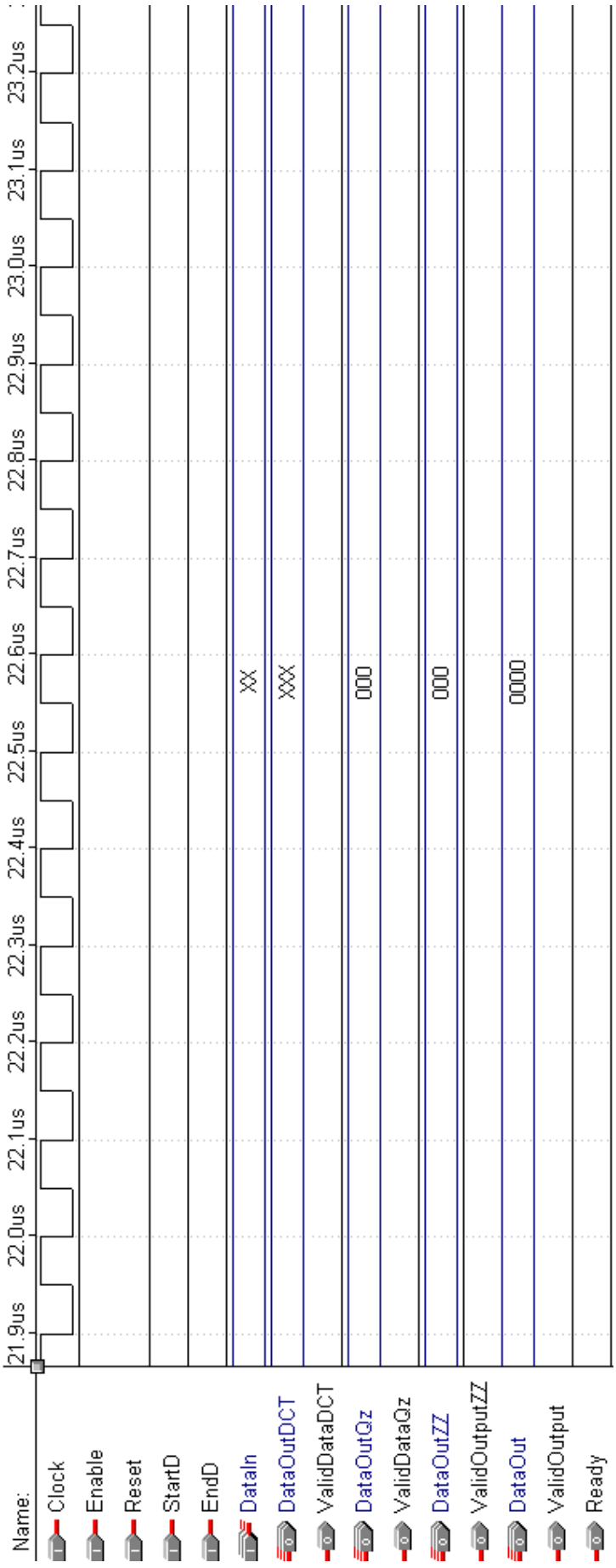

Name:
Clock
Enable
Reset
StartD
EndD
DataIn
DataOutDCT
ValidDataDCT
DataOutQz
ValidDataQz
DataOutZZ
ValidOutputZZ
DataOut
ValidOutput
Ready
15.6us
15.7us
15.8us
15.9us
16.0us
16.1us
16.2us
16.3us
16.4us
16.5us
16.6us
16.7us
16.8us
16.9us
000
FF9
007
000
FFD
004
FFE
FF8
FFF
FFE
FF8
FFF
001
XX
000
000
0000

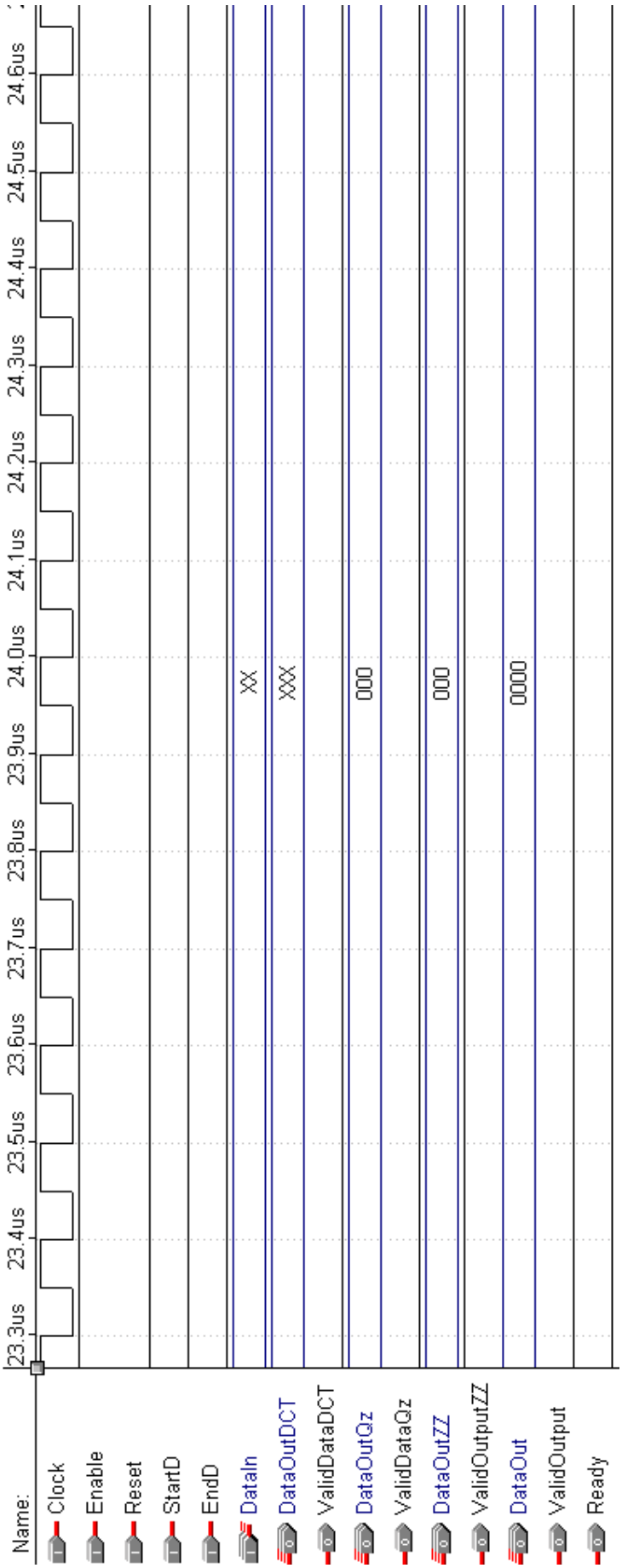

Name:
Clock
Enable
Reset
StartD
EndD
DataIn
DataOut
ValidDataDCT
DataOutQz
ValidDataQz
DataOutZZ
ValidOutputZZ
DataOutDCT
ValidOutput
Ready
17.0us
17.2us
17.4us
17.6us
17.8us
18.0us
18.2us
XX
0000
000
000
001
003
004
FFB
000
XXX
ValidDataDCT signal asserted low to indicate the end of valid 2D DCT results

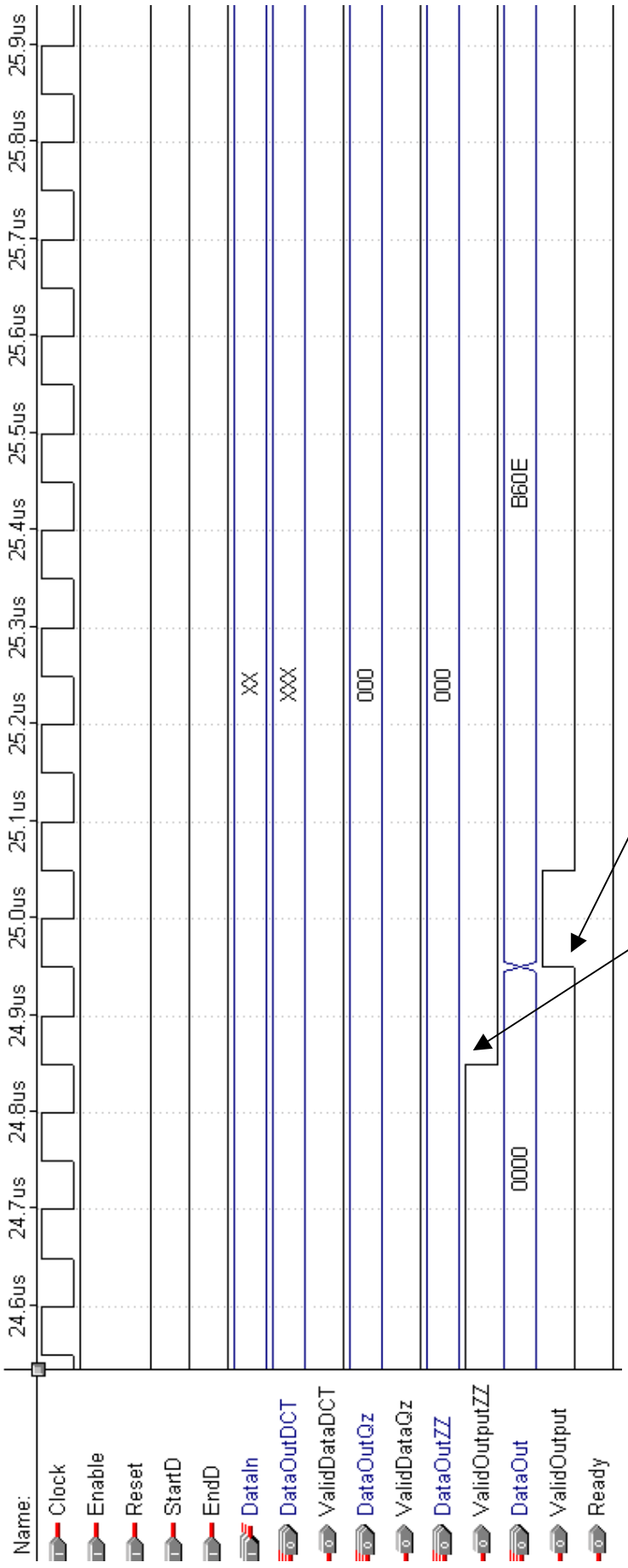

Name:
Clock
Enable
Reset
StartD
EndD
DataIn
DataOutDCT
ValidDataDCT
DataOutQz
ValidDataQz
DataOutZZ
ValidOutputZZ
DataOut
ValidOutput
Ready
18.0us
18.1us
18.2us
18.3us
18.4us
18.5us
18.6us
18.7us
18.8us
18.9us
19.0us
19.1us
19.2us
19.3us
XX
XXX
000
00B
7FF
001
000
7FE
001
000
001
0000
ValidDataQZ signal asserted low to indicate the end of valid data from quantizer
ValidOutputZZ signal asserted high to indicate the beginning of valid zig zagged results

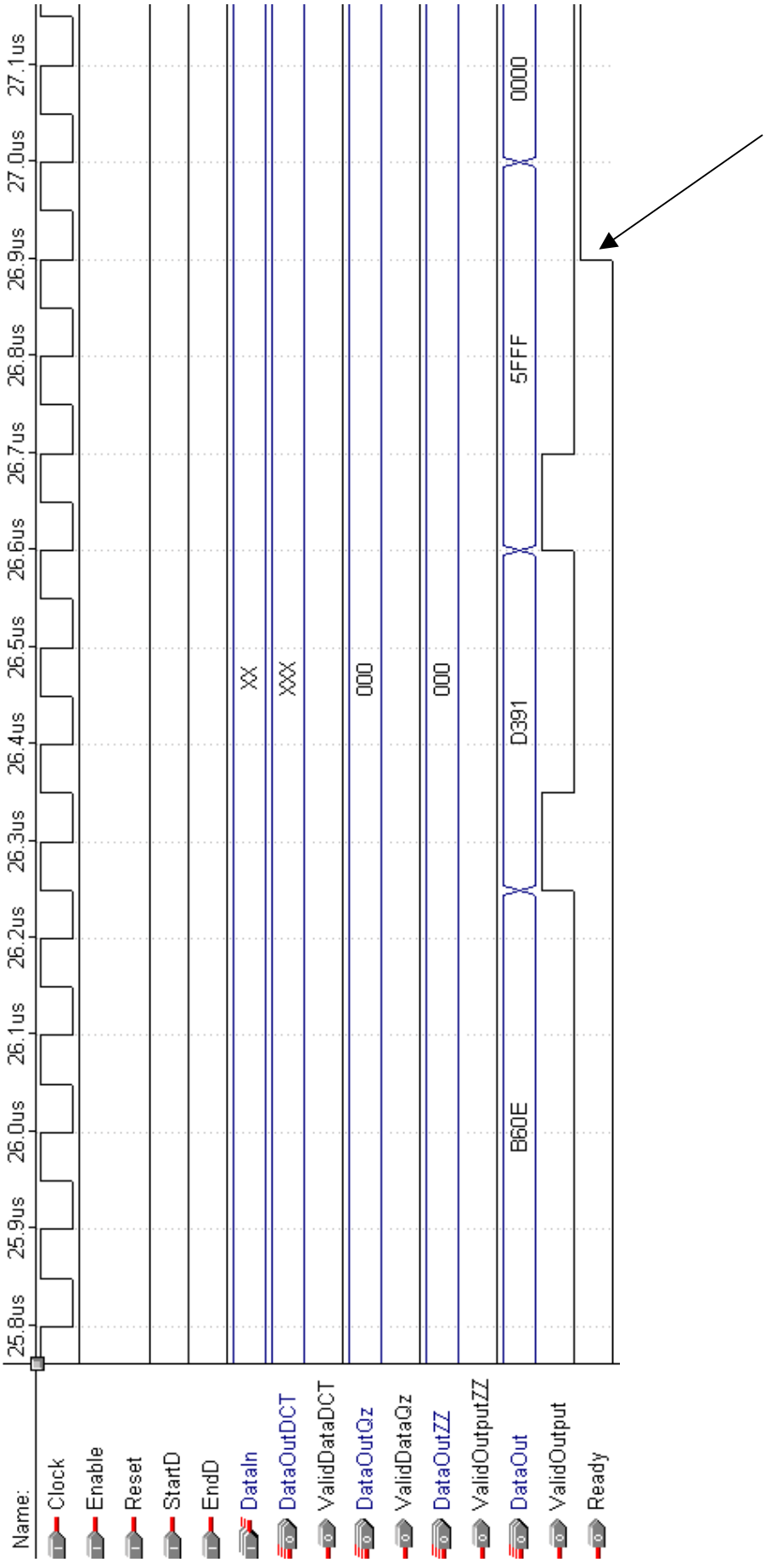

Name:
Clock
Enable
Reset
StartD
EndD
DataIn
DataOutDCT
ValidDataDCT
DataOutQz
ValidDataQz
DataOutZZ
ValidOutputZZ
DataOut
ValidOutput
Ready
.3us
19.4us
19.5us
19.6us
19.7us
19.8us
19.9us
20.0us
20.1us
20.2us
20.3us
20.4us
20.5us
20.6us
20
XX
XXX
000
001
7FF
000
0000


Name:
Clock
Enable
Reset
StartD
EndD
DataIn
DataOutDCT
ValidDataDCT
DataOutQz
ValidDataQz
DataOutZZ
ValidOutputZZ
DataOut
ValidOutput
Ready
20.6us
20.7us
20.8us
20.9us
21.0us
21.1us
21.2us
21.3us
21.4us
21.5us
21.6us
21.7us
21.8us
21.9us
XX
XXX
000
000
0000


Name:
Clock
Enable
Reset
StartD
EndD
DataIn
DataOutDCT
ValidDataDCT
DataOutQz
ValidDataQz
DataOutZZ
ValidOutputZZ
DataOut
ValidOutput
Ready
21.9us
22.0us
22.1us
22.2us
22.3us
22.4us
22.5us
22.6us
22.7us
22.8us
22.9us
23.0us
23.1us
23.2us
XX
XXX
000
000
0000


Name:
Clock
Enable
Reset
StartD
EndD
DataIn
DataOutDCT
ValidDataDCT
DataOutQz
ValidDataQz
DataOutZZ
ValidOutputZZ
DataOut
ValidOutput
Ready
23.3us
23.4us
23.5us
23.6us
23.7us
23.8us
23.9us
24.0us
24.1us
24.2us
24.3us
24.4us
24.5us
24.6us
XX
XXX
000
000
0000


Name:
Clock
Enable
Reset
StartD
EndD
DataIn
DataOutDCT
ValidDataDCT
DataOutQz
ValidDataQz
DataOutZZ
ValidOutputZZ
DataOut
ValidOutput
Ready
24.6us
24.7us
24.8us
24.9us
25.0us
25.1us
25.2us
25.3us
25.4us
25.5us
25.6us
25.7us
25.8us
25.9us
XX
XXX
000
000
0000
B60E
ValidOutputZZ signal asserted low to indicate the end of valid zig zagged data
ValidOutput pulse indicates the presense of valid compressed data at the output


Name:
Clock
Enable
Reset
StartD
EndD
DataIn
DataOutDCT
ValidDataDCT
DataOutQz
ValidDataQz
DataOutZZ
ValidOutputZZ
DataOut
ValidOutput
Ready
25.8us
25.9us
26.0us
26.1us
26.2us
26.3us
26.4us
26.5us
26.6us
26.7us
26.8us
26.9us
27.0us
27.1us
XX
XXX
000
000
B60E
D391
5FFF
0000
Ready signal asserted high to indicate that the pipeline is completely flushed and encoder is ready to accept new data

7 Future Enhancements

7.1 Incorporation of Additional JPEG Encoding Modes

At present, our JPEG implementation covers only the Baseline Sequential encoding process. One enhancement can be the incorporation of the other modes as well, namely: Progressive, Hierarchical and Lossless Encoding.

7.2 Implementing the full JPEG Codec

Implementing the decoder as well is another future upgrade worth mentioning. At present, only the encoder is implemented. It may also be possible to implement the full codecs for several modes specified in the JPEG Standard, perhaps on a high capacity FPGA.

7.3 Progression to JPEG 2000

The JPEG 2000 specification describes an image compression system that allows great flexibility, not only for the compression of images, but also for the access into the compressed data. It introduces a number of mechanisms for locating and extracting data for the purpose of retransmission, storage, display, or editing. This access allows storage and retrieval of data appropriate for a given application, without decoding.

The JPEG 2000 standard is a radical improvement over the original JPEG standard, allowing for much improved flexibility. The implementation of this standard would not simply be an enhancement, but instead warrants a major redesign. For more on the JPEG 2000 Standard, refer to [8].

7.4 Improvements to the Current Pipeline

The most basic improvement that can be made to the current design is the reduction of the maximum latency of every stage of the pipeline. The main aim behind this is to reduce the clock period, allowing for a faster implementation that processes data at an increased rate. One way of achieving this is to divide the stages with long propagation delays into two independent stages. Although this increases the length of the pipeline – and thus latency – more data is being processed per second because of the higher clock frequency.

8 Conclusion

In this report we have described the implementation of a fully pipelined architecture for JPEG baseline image compression standard. The architectures for the various stages are based on efficient and high performance designs suited for VLSI implementation. The implementation was tested for functional correctness using Verilog with Altera's tools.

There were several reasons for our choice of implementing the JPEG compression algorithm. Firstly, we wanted to work on the implementation of compute intensive algorithms in hardware using HDLs. Secondly, the JPEG compression standard was a good candidate as it does not specify any particular architecture for its implementation and in this way permits the implementers to try various innovations.

We had set some goals of our work. First was the implementation of a high throughput and pipelined design of the entire JPEG algorithm. Secondly, we wanted to test it in a practical setup. We were successful in implementing a pipelined architecture on a single chip. However, a practical setup for testing the design could not be realized due to the unavailability of necessary hardware and design tools. We were limited to the student versions of design software and a low capacity evaluation board that too was shared among various working groups. This seriously impeded our work as our design called for full-featured versions of design software and large capacity FPGA that were arranged in the final stages of our work. Although the original intent was to implement the full-color baseline JPEG encoder, it soon became apparent that this would be extremely difficult, given the constraints.

Another severe hindrance to our project work was the lack of appropriate guidance on design and implementation using HDLs. All work in this regard was carried out through self-study. Also, in the absence of any notable archetypes, most of the experience was gained through trial and error.

Our project provided us the opportunity to experiment with various design tradeoffs and on numerous occasions during our implementation we found that we needed to try different innovations to suit our requirements.

Our architecture has been completely synthesized and work for its actual verification in a practical setup is in progress.

Bibliography

[1] Ming-Ting Sun, Ting Chung Chen, and Albert M. Gottlieb, *'VLSI Implementation of a 16 X 16 Discrete Cosine Transform'*, IEEE Transactions on Circuits and Systems, Vol. 36, No. 4, APRIL 1989.

[2] Mario Kovac and N. Ranganathan, *'JAGUAR: A Fully Pipelined VLSI Architecture for JPEG Image Compression Standard'*, Proceedings of the IEEE, Vol. 83, No. 2, FEBRUARY 1995.

[3] ITU and CCITT, *'Information Technology – Digital Compression and Coding of Continuous-Tone Still Images – Requirements and Guidelines'*, ISO/IEC IS 10918-1, ITU-T T.81, SEPTEMBER 1992.

[4] Gregory K. Wallace, *'The JPEG Still Picture Compression Standard'*, Multimedia Engineering, Digital Equipment Corp., submitted for publication in IEEE Transactions on Consumer Electronics, DECEMBER 1991.

[5] Mark Nelson and Jean-Loup Gally, *'The Data Compression Handbook'*, 2ed., M&T Books.

[6] Douglas J. Smith, *'HDL Chip Design – A practical guide for designing, synthesizing, and simulating ASICs and FPGAs using VHDL or Verilog'*, Doone Publications.

[7] Samir Palnitkar, *'Verilog HDL: A Guide to Digital Design and Synthesis'*, SunSoft Press, Prentice Hall, .

[8] Mark Gordon Arnold, *'Verilog Digital Computer Design – Algorithms into Hardware'*, Prentice Hall, 1999.

[9] *'Xentec X_JPEG codec'*, Xentec Inc., 2908 South Sheridan Way, Oakville, ON L6J 7J8 Canada.

[10] *'CS6100 Motion JPEG Encoder'*, Amphion Semiconductor Inc., 2001 Gateway Place, Suite 130W, San Jose, CA 95110.

[11] *'JPEG 2000 Image Coding System'*, JPEG 2000 Final Committee Draft, ver. 1.0, MARCH 2000

[12] Y. Arai, T. Agui, and M. Nakajima, *'A fast DCT-SQ scheme for images'*, Trans IEICE, Vol. E71, No. 11, pp 1095-1097, 1998.

[13] Chen-Mie Wu and Andy Chiou, *'A SIMD-Systolic Architecture and VLSI chip for the Two Dimensional DCT and IDCT'*, IEEE Transactions on Consumer Electronics, Vol. 39, No. 4, NOVEMBER 1993

[14] William B. Pennebaker and Joan L. Mitchell, *'JPEG Still Image Data Compression Standard'*, Van Nostrand Reinhold, 1993, ISBN 0-442-01272-1.

[15] Nam Ik Cho and Sang Uk Lee, *'Fast Algorithm and Implementation of 2-D Discrete Cosine Transform'*, IEEE Transactions on Circuits and Systems, Vol. 38, No. 3, MARCH 1991.

[16] Javier Valls, Trini Sansaloni, Marcos M. Peiro, and Eduardo Boemo, '*Fast FPGA-Based Pipelined Digit-Serial/Parallel Multipliers'*, Proceedings of the 1999 IEEE International Symposium on Circuits and Systems, June 1999, Vol. 1, pages 482-485.

[17] John P. Hayes, *'Computer Architecture and Organisation'*, 2nd Edition, McGraw Hill, 1988.

Appendix A

```
/*****************************************************************/
/*This is top level file that instantiates controllers of various  */
/* stages and keeps track of Ready signal                         */
/*****************************************************************/

module Jpeg(      DataIn,
                  StartD,
                  EndD,
                  Clock,
                  Enable,
                  Reset,
                  ValidOutput,
                  DataOut,
                  Ready
                  );

input             StartD;           //Signals the start of input data
input             EndD;             //Signals the end of input data
input             Clock;            //System wide Clock signal
input             Reset;            //System wide Reset signal
input             Enable;           //Chip Enable signal
input [7:0]       DataIn;           //Input data

output [15:0]     DataOut;          //Compressed data
output            ValidOutput;      //Signal for valid output
output            Ready;            //Signal to indicate if chip
                                    //is ready to take data

reg               Ready;

wire [11:0] DataOutDCT;             //Output data from DCT
wire [11:0] DataOutQz;              //Output data from Quantizer
wire [11:0] DataOutZZ;              //Output data from ZigZag
wire ValidDataDCT;                  //Signal arrival of data from DCT
wire ValidDataQz;             //Signal arrival of data from quantizer
wire ValidOutputZZ;           // Signal arrival of data from ZigZag
wire ResetIt;                 //Internal Reset

reg EndofData;                //EndD latch

dct dct_inst(DataIn, StartD, EndD, Clock, Reset, Enable,
                  ValidDataDCT, DataOutDCT);

multiplier multiplier_inst(Reset, ValidDataDCT, Enable, Clock,
      DataOutDCT, DataOutQz,  ValidDataQz);

ZigZag zigzag_inst(DataOutQz, Clock, Reset, Enable, ValidDataQz,
      ValidOutputZZ, DataOutZZ);

PHuffTop PHuffTop_inst(DataOutZZ, Clock, Reset, Enable,
      ValidOutputZZ, EndPulse, DataOut, ValidOutput, ReadyOut);

assign ResetIt = Reset & StartD;

always @(posedge Clock)
begin
```

```
            if(ResetIt)             //Reset goes low when end of data
            begin                   //is signaled and remains low
                  Ready <= 1;       //till the whole pipeline is
                  EndofData <= 0;   // flushed
            end
            else
            begin
                  if(EndD)
                  begin
                        EndofData <= 1;
                        Ready <= 0;
                  end
                  else if(EndofData & ReadyOut)
                  begin
                        Ready <= 1;
                  end
            end
end


endmodule
```

```
/*****************************************************************/
/*    This module implements the timing and control circuit      */
/*    for the 2D DCT stage                                       */
/*****************************************************************/

module DCT (      DataIn,
                  StartD,
                  EndD,
                  Clock,
                  Reset,
                  Enable,
                  ValidData,
                  DataOut
           );

input             StartD;     // Signals beginning of data arrival
input             EndD;       // Signals end of data arrival
input             Reset;
input             Enable;
input             Clock;
input [7:0]       DataIn;

output            ValidData;  // Signals the output of valid data
output [11:0]     DataOut;    // Coefficient data out

reg               ValidData;

wire [9:0]        DataOutdct; // Data out from first 1D DCT stage
wire [9:0]        DataOutTranspose; // Data out from transpose
integer           index;      // To keep track of input sequence
integer           index2;     // To keep track of output sequence
reg               DataComing; // Flag to indicate valid data i/p
reg               DataComingDct1;   // Indicates valid data from
                                    // first 1D DCT
reg               DataComingTranspose; // Indicates valid data from
                                       // transpose buffer
reg               EnableDct1;       // Signals to enable the stages
wire              Enabledct1;
wire              Enabledct2;
wire              EnableTranspose;
wire              ResetNewData;     // internal reset signal

// Instantiations of 1D DCT stages and the transpose buffer

DCT1D dct1inst    (DataIn, DataOutdct, Clock, ResetNewData,
      Enabledct1);
transpose transposeinst (EnableTranspose,DataOutdct,
      DataOutTranspose, Clock, ResetNewData);
DCT1D dct2inst    (DataOutTranspose, DataOut, Clock, ResetNewData,
      Enabledct2);

assign Enabledct1 = Enable & EnableDct1;
assign Enabledct2 = Enable & DataComingTranspose;
assign EnableTranspose = Enable & DataComingDct1;
assign ResetNewData = !Reset | StartD;

always @(posedge Clock or posedge ResetNewData)
begin
      if(ResetNewData)
      begin
            index <= 0;
```

```
            index2 <= 107;
            DataComingDct1 <= 0;
            DataComingTranspose <= 0;
            ValidData <= 0;
      end
      else if(Enable)
      begin
            if(index == 20)               // if first 1D DCT full
                  DataComingDct1 <= 1;    // Set flag
            if(index == 85)               // if transpose full
                  DataComingTranspose <= 1; // Set flag
            if(index == 107 & index2 != 0)// if second 1D DCT
                  ValidData <= 1;         // full, set valid
            else                          // data out flag
                  index <= index + 1;

      if(!DataComing)               // if end data signal
            begin                         // has been received
                  if(index2 == 0 )        // if pipeline empty
                        ValidData <= 0;   // reset valid out flag
                  else
                        index2 <= index2 - 1;
            end
      end

end

always @(posedge StartD or posedge EndD or posedge Clock)
begin
      if(StartD)              // if valid data in received
      begin
            DataComing <= 1;  // set the flag
            EnableDct1 <= 1;  // and enable first 1D DCT
      end
      else if(EndD)           // if end of valid data received
      begin
            DataComing <= 0;  // reset the flag
      end
end

endmodule
```

```
/*****************************************************************/
/*    This module implements the 8X1  one-dimensional DCT stage    */
/*****************************************************************/

module DCT1D(     DataIn,
DataOut,
Clock,
Reset,
Enable
);

input             Clock;      // Clock input signal
input             Reset;      // low asserted System-wide reset
input             Enable;     // System-wide enable signal
input [9:0]       DataIn;     // Data input

output [11:0]     DataOut;    // Coefficient data output

reg [11:0]        DataOut;

reg [15:0]        Q0;         // The 8 Q parallel-in parallel-out
reg [15:0]        Q1;         // shift registers which make up the
reg [15:0]        Q2;         // input stage of the pipeline
reg [15:0]        Q3;
reg [15:0]        Q4;
reg [15:0]        Q5;
reg [15:0]        Q6;
reg [15:0]        Q7;

reg [15:0]        R0;         // The 8 R parallel-in serial out shift
reg [15:0]        R1;         // registers
reg [15:0]        R2;
reg [15:0]        R3;
reg [15:0]        R4;
reg [15:0]        R5;
reg [15:0]        R6;
reg [15:0]        R7;

reg [10:0]        SR0;        // The 8 Shift-and-Accumulate
reg [10:0]        SR1;        // registers
reg [10:0]        SR2;
reg [10:0]        SR3;
reg [10:0]        SR4;
reg [10:0]        SR5;
reg [10:0]        SR6;
reg [10:0]        SR7;

reg [7:0]         Buff0;      // The 8-bit buffer registers appended
reg [7:0]         Buff1;      // to the Shift-and-Accumulate
reg [7:0]         Buff2;      // registers
reg [7:0]         Buff3;
reg [7:0]         Buff4;
reg [7:0]         Buff5;
reg [7:0]         Buff6;
reg [7:0]         Buff7;

reg [11:0]        U0;         // The 8 U parallel-in parallel-out
reg [11:0]        U1;         // Shift registers which make up the
reg [11:0]        U2;         // output stage of the pipeline
reg [11:0]        U3;
reg [11:0]        U4;
```

```
reg [11:0]        U5;
reg [11:0]        U6;
reg [11:0]        U7;

reg [5:0]         index1;     // Counter to keep track of the R-shift
                              // registers
reg [5:0]         index2;     // Counter to keep track of the SAR and
                              // output U registers

wire              R0b0;       // wires carrying the LSB of the
wire              R1b0;       // R-shift registers
wire              R2b0;
wire              R3b0;
wire              R4b0;
wire              R5b0;
wire              R6b0;
wire              R7b0;

wire              R0b1;       // wires carrying the second LSB of the
wire              R1b1;       // R-shift registers
wire              R2b1;
wire              R3b1;
wire              R4b1;
wire              R5b1;
wire              R6b1;
wire              R7b1;

wire              Au0;        // Wires for bus uA
wire              Au1;
wire              Au2;
wire              Au3;

wire              Av0;        // Wires for bus vA
wire              Av1;
wire              Av2;
wire              Av3;

wire              Bu0;        // Wires for bus uB
wire              Bu1;
wire              Bu2;
wire              Bu3;
wire              Bv0;        // Wires for bus vB
wire              Bv1;
wire              Bv2;
wire              Bv3;

wire              d0;         // Adder and subtractor internal
wire              d1;         // connections
wire              d2;
wire              d3;
wire              d4;
wire              d5;
wire              d6;
wire              d7;
wire              q0;
wire              q1;
wire              q2;
wire              q3;
wire              q4;
wire              q5;
wire              q6;
```

```
wire              q7;
wire              Cout0;
wire              Cout1;
wire              Cout2;
wire              Cout3;
wire              Bout0;
wire              Bout1;
wire              Bout2;
wire              Bout3;

wire [10:0]       Aromout0;   // ROMA Output buses
wire [10:0]       Aromout1;
wire [10:0]       Aromout2;
wire [10:0]       Aromout3;
wire [10:0]       Aromout4;
wire [10:0]       Aromout5;
wire [10:0]       Aromout6;
wire [10:0]       Aromout7;

wire [10:0]       Bromout0;   // ROMB Output buses
wire [10:0]       Bromout1;
wire [10:0]       Bromout2;
wire [10:0]       Bromout3;
wire [10:0]       Bromout4;
wire [10:0]       Bromout5;
wire [10:0]       Bromout6;
wire [10:0]       Bromout7;

wire [3:0]        Abusu;      // Buses uA, vA, uB, and vB
wire [3:0]        Abusv;
wire [3:0]        Bbusu;
wire [3:0]        Bbusv;

// Instantiations of two-bit adders and subtractors
fulladd    Afa0(R0b0, R7b0, q0, Au0, Cout0),
           Afa1(R1b0, R6b0, q1, Au1, Cout1),
           Afa2(R2b0, R5b0, q2, Au2, Cout2),
           Afa3(R3b0, R4b0, q3, Au3, Cout3),

           Bfa0(R0b1, R7b1, Cout0, Bu0, d0),
           Bfa1(R1b1, R6b1, Cout1, Bu1, d1),
           Bfa2(R2b1, R5b1, Cout2, Bu2, d2),
           Bfa3(R3b1, R4b1, Cout3, Bu3, d3);

fullsub    Afs0(R0b0, R7b0, q4, Av0, Bout0),
           Afs1(R1b0, R6b0, q5, Av1, Bout1),
           Afs2(R2b0, R5b0, q6, Av2, Bout2),
           Afs3(R3b0, R4b0, q7, Av3, Bout3),

           Bfs0(R7b1, R0b1, Bout0, Bv0, d4),
           Bfs1(R6b1, R1b1, Bout1, Bv1, d5),
           Bfs2(R5b1, R2b1, Bout2, Bv2, d6),
           Bfs3(R4b1, R3b1, Bout3, Bv3, d7);

// Instantiations of flip-flops for holding carry outs and borrow out
flipflop ff0(d0, clock, Reset, q0);
flipflop ff1(d1, clock, Reset, q1);
flipflop ff2(d2, clock, Reset, q2);
flipflop ff3(d3, clock, Reset, q3);
flipflop ff4(d4, clock, Reset, q4);
flipflop ff5(d5, clock, Reset, q5);
```

```
flipflop ff6(d6, clock, Reset, q6);
flipflop ff7(d7, clock, Reset, q7);


// Instantiations of ROMS in RACs

NROM0 Ainstrom0(Aromout0, Clock, Abusu);
NROM1 Ainstrom1(Aromout1, Clock, Abusv);
NROM2 Ainstrom2(Aromout2, Clock, Abusu);
NROM3 Ainstrom3(Aromout3, Clock, Abusv);
NROM4 Ainstrom4(Aromout4, Clock, Abusu);
NROM5 Ainstrom5(Aromout5, Clock, Abusv);
NROM6 Ainstrom6(Aromout6, Clock, Abusu);
NROM7 Ainstrom7(Aromout7, Clock, Abusv);

NROM0 Binstrom0(Bromout0, Clock, Bbusu);
NROM1 Binstrom1(Bromout1, Clock, Bbusv);
NROM2 Binstrom2(Bromout2, Clock, Bbusu);
NROM3 Binstrom3(Bromout3, Clock, Bbusv);
NROM4 Binstrom4(Bromout4, Clock, Bbusu);
NROM5 Binstrom5(Bromout5, Clock, Bbusv);
NROM6 Binstrom6(Bromout6, Clock, Bbusu);
NROM7 Binstrom7(Bromout7, Clock, Bbusv);

assign R0b0 = R0[0];
assign R1b0 = R1[0];
assign R2b0 = R2[0];
assign R3b0 = R3[0];
assign R4b0 = R4[0];
assign R5b0 = R5[0];
assign R6b0 = R6[0];
assign R7b0 = R7[0];

assign R0b1 = R0[1];
assign R1b1 = R1[1];
assign R2b1 = R2[1];
assign R3b1 = R3[1];
assign R4b1 = R4[1];
assign R5b1 = R5[1];
assign R6b1 = R6[1];
assign R7b1 = R7[1];

assign Abusu = {Au3, Au2, Au1, Au0};
assign Abusv = {Av3, Av2, Av1, Av0};
assign Bbusu = {Bu3, Bu2, Bu1, Bu0};
assign Bbusv = {Bv3, Bv2, Bv1, Bv0};


always@(posedge Clock or negedge Reset)
begin
if(!Reset)
begin
            index1=0;
index2=0;
end
else if(Enable)
begin
            Q0 <= Q1;         // Data flow through the input stage
            Q1 <= Q2;
            Q2 <= Q3;
            Q3 <= Q4;
```

```
            Q4 <= Q5;
            Q5 <= Q6;
            Q6 <= Q7;
Q7<={DataIn[9],DataIn[9],DataIn[9],DataIn[9],DataIn[9],
            DataIn[9],DataIn};

            if(index1 == 8)   // When the input stage is full,
                  begin       // load data into the R registers.
                  R0 <= Q0;
                  R1 <= Q1;
                  R2 <= Q2;
                  R3 <= Q3;
                  R4 <= Q4;
                  R5 <= Q5;
                  R6 <= Q6;
                  R7 <= Q7;

                  index1 <= 1;// Reset the count
            end
            else
            begin                    // else shift out the values from
                  R0 <= R0 >> 2;     // R shift registers
                  R1 <= R1 >> 2;
                  R2 <= R2 >> 2;
                  R3 <= R3 >> 2;
                  R4 <= R4 >> 2;
                  R5 <= R5 >> 2;
                  R6 <= R6 >> 2;
                  R7 <= R7 >> 2;

                  index1 <= index1 + 1; // Count the number of shifts
            end


            if(index2 == 20)          // if valid results are present
                  begin               // load the output stage
                  U0<={SR0[4],SR0[3],SR0[2],SR0[1],SR0[0],Buff0}>> 1;
                  U1<={SR1[4],SR1[3],SR1[2],SR1[1],SR1[0],Buff1}>> 1;
                  U2<={SR2[4],SR2[3],SR2[2],SR2[1],SR2[0],Buff2}>> 1;
                  U3<={SR3[4],SR3[3],SR3[2],SR3[1],SR3[0],Buff3}>> 1;
                  U4<={SR4[4],SR4[3],SR4[2],SR4[1],SR4[0],Buff4}>> 1;
                  U5<={SR5[4],SR5[3],SR5[2],SR5[1],SR5[0],Buff5}>> 1;
                  U6<={SR6[4],SR6[3],SR6[2],SR6[1],SR6[0],Buff6}>> 1;
                  U7<={SR7[4],SR7[3],SR7[2],SR7[1],SR7[0],Buff7}>> 1;

                  index2 <= 13;     // reset the count
                  end
            else
                  begin
                  U6 <= U7;         // Data flow through the output
                  U5 <= U6;         // stage
                  U4 <= U5;
                  U3 <= U4;
                  U2 <= U3;
                  U1 <= U2;
                  U0 <= U1;

                  index2 <= index2 + 1;
end
            DataOut <= U0;          // assignment to Data output
end
```

```
always@(posedge Clock or negedge Reset)
begin
      if(!Reset)
            begin
            SR0 =0;
SR1 =0;
SR2 =0;
SR3 =0;
SR4 =0;
SR5 =0;
SR6 =0;
SR7 =0;
end

else if(Enable)
begin
            if(index2 == 19 ) // Perform shift-and-subtract
begin
                  {SR0, Buff0} = {SR0[10], ({SR0, Buff0} >> 1)} ;
                  SR0 = SR0 + Aromout0;
                  {SR0, Buff0} = {SR0[10], ({SR0, Buff0} >> 1)} ;
                  SR0 = (SR0 - Bromout0);

                  {SR1, Buff1} = {SR1[10], ({SR1, Buff1} >> 1)} ;
                  SR1 = SR1 + Aromout1;
                  {SR1, Buff1} = {SR1[10], ({SR1, Buff1} >> 1)} ;
                  SR1 = (SR1 - Bromout1) >> 1;

                  {SR2, Buff2} = {SR2[10], ({SR2, Buff2} >> 1)} ;
                  SR2 = SR2 + Aromout2;
                  {SR2, Buff2} = {SR2[10], ({SR2, Buff2} >> 1)} ;
                  SR2 = (SR2 - Bromout2) >> 1;

                  {SR3, Buff3} = {SR3[10], ({SR3, Buff3} >> 1)} ;
                  SR3 = SR3 + Aromout3;
                  {SR3, Buff3} = {SR3[10], ({SR3, Buff3} >> 1)} ;
                  SR3 = (SR3 - Bromout3) >> 1;

                  {SR4, Buff4} = {SR4[10], ({SR4, Buff4} >> 1)} ;
                  SR4 = SR4 + Aromout4;
                  {SR4, Buff4} = {SR4[10], ({SR4, Buff4} >> 1)} ;
                  SR4 = (SR4 - Bromout4) >> 1;

                  {SR5, Buff5} = {SR5[10], ({SR5, Buff5} >> 1)} ;
                  SR5 = SR5 + Aromout5;
                  {SR5, Buff5} = {SR5[10], ({SR5, Buff5} >> 1)} ;
                  SR5 = (SR5 - Bromout5) >> 1;

                  {SR6, Buff6} = {SR6[10], ({SR6, Buff6} >> 1)} ;
                  SR6 = SR6 + Aromout6;
                  {SR6, Buff6} = {SR6[10], ({SR6, Buff6} >> 1)} ;
                  SR6 = (SR6 - Bromout6) >> 1;

                  {SR7, Buff7} = {SR7[10], ({SR7, Buff7} >> 1)} ;
                  SR7 = SR7 + Aromout7;
                  {SR7, Buff7} = {SR7[10], ({SR7, Buff7} >> 1)} ;
                  SR7 = (SR7 - Bromout7) >> 1;
                  end
            else        // else perform shift-and-add
                  begin
```

```
                    {SR0, Buff0} = {SR0[10], ({SR0, Buff0} >> 1)};
                    SR0 = SR0 + Aromout0;
                    {SR0, Buff0} = {SR0[10], ({SR0, Buff0} >> 1)};
                    SR0 = SR0 + Bromout0;

                    {SR1, Buff1} = {SR1[10], ({SR1, Buff1} >> 1)};
                    SR1 = SR1 + Aromout1;
                    {SR1, Buff1} = {SR1[10], ({SR1, Buff1} >> 1)} ;
                    SR1 = SR1 + Bromout1;

                    {SR2, Buff2} = {SR2[10], ({SR2, Buff2} >> 1)} ;
                    SR2 = SR2 + Aromout2;
                    {SR2, Buff2} = {SR2[10], ({SR2, Buff2} >> 1)} ;
                    SR2 = SR2 + Bromout2;

                    {SR3, Buff3} = {SR3[10], ({SR3, Buff3} >> 1)} ;
                    SR3 = SR3 + Aromout3;
                    {SR3, Buff3} = {SR3[10], ({SR3, Buff3} >> 1)};
                    SR3 = SR3 + Bromout3;

                    {SR4, Buff4} = {SR4[10], ({SR4, Buff4} >> 1)} ;
                    SR4 = SR4 + Aromout4;
                    {SR4, Buff4} = {SR4[10], ({SR4, Buff4} >> 1)} ;
                    SR4 = SR4 + Bromout4;

                    {SR5, Buff5} = {SR5[10], ({SR5, Buff5} >> 1)} ;
                    SR5 = SR5 + Aromout5;
                    {SR5, Buff5} = {SR5[10], ({SR5, Buff5} >> 1)} ;
                    SR5 = SR5 + Bromout5;

                    {SR6, Buff6} = {SR6[10], ({SR6, Buff6} >> 1)} ;
                    SR6 = SR6 + Aromout6;
                    {SR6, Buff6} = {SR6[10], ({SR6, Buff6} >> 1)} ;
                    SR6 = SR6 + Bromout6;

                    {SR7, Buff7} = {SR7[10], ({SR7, Buff7} >> 1)} ;
                    SR7 = SR7 + Aromout7;
                    {SR7, Buff7} = {SR7[10], ({SR7, Buff7} >> 1)} ;
                    SR7 = SR7 + Bromout7;
                    end
                end
end

endmodule
```

```
/*****************************************************************/
/*    This module implements the transpose buffer stage of the     */
/*    encoder model                                                */
/*****************************************************************/

module Transpose( DataInReady,
DataIn,
Clock,
Reset,
DataOut
);

input             DataInReady;      // Signal indicating valid data
input [9:0]       DataIn;           // Data from one-D DCT stage
input             Clock;            // input clock signal
input             Reset;            // low asserted system-wide reset

output [9:0]      DataOut;          // Data out from transpose buffer

reg               BufferFull;       // Internal signal indicating
// that the buffer is full and
// ready to output data

reg [6:0]         Count;            // Counter to track when the
                                    // buffer is full

// The R registers into which the input data enters before being
// transferred to the other set of registers

reg [9:0]   R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11,   R12,
      R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24,
      R25, R26, R27, R28, R29, R30, R31, R32, R33, R34, R35, R36,
      R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48,
      R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60,
      R61, R62, R63;

reg [9:0]   C0, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11,  C12,
      C13, C14, C15, C16, C17, C18, C19, C20, C21, C22, C23, C24,
      C25, C26, C27, C28, C29, C30, C31, C32, C33, C34, C35, C36,
      C37, C38, C39, C40, C41, C42, C43, C44, C45, C46, C47, C48,
      C49, C50, C51, C52, C53, C54, C55, C56, C57, C58, C59, C60,
      C61, C62, C63;

// The C registers into which data is copied in parallel from R
// registers and data is output in a transpose fashion

assign DataOut = C63;

always@(posedge Clock or posedge Reset)
  begin
  if(Reset)
    Count <= 0;
  else
    begin
    if(DataInReady)
      begin
      R0 <= DataIn;           // The following transfers represent
      R1 <= R0;               // data flow through the R registers
      R2 <= R1;
      R3 <= R2;
      R4 <= R3;
```

```
R5 <= R4;
R6 <= R5;
R7 <= R6;
R8 <= R7;
R9 <= R8;
R10 <= R9;
R11 <= R10;
R12 <= R11;
R13 <= R12;
R14 <= R13;
R15 <= R14;
R16 <= R15;
R17 <= R16;
R18 <= R17;
R19 <= R18;
R20 <= R19;
R21 <= R20;
R22 <= R21;
R23 <= R22;
R24 <= R23;
R25 <= R24;
R26 <= R25;
R27 <= R26;
R28 <= R27;
R29 <= R28;
R30 <= R29;
R31 <= R30;
R32 <= R31;
R33 <= R32;
R34 <= R33;
R35 <= R34;
R36 <= R35;
R37 <= R36;
R38 <= R37;
R39 <= R38;
R40 <= R39;
R41 <= R40;
R42 <= R41;
R43 <= R42;
R44 <= R43;
R45 <= R44;
R46 <= R45;
R47 <= R46;
R48 <= R47;
R49 <= R48;
R50 <= R49;
R51 <= R50;
R52 <= R51;
R53 <= R52;
R54 <= R53;
R55 <= R54;
R56 <= R55;
R57 <= R56;
R58 <= R57;
R59 <= R58;
R60 <= R59;
R61 <= R60;
R62 <= R61;
R63 <= R62;
if(Count==64)                       // if count has reached max
  Count <= 1;                       // value, reset it to 1
```

```
      else
        Count <= Count + 1;
      end
  end
end

always@(posedge Clock or posedge Reset)
  begin
  if(Reset)
    BufferFull <= 0;
  else
  begin
  if(Count==64)                          // if the input stage is
    begin                                // full, data is copied
    C0 <= R0;                            // to the output stage
    C1 <= R1;
    C2 <= R2;
    C3 <= R3;
    C4 <= R4;
    C5 <= R5;
    C6 <= R6;
    C7 <= R7;
    C8 <= R8;
    C9 <= R9;
    C10 <= R10;
    C11 <= R11;
    C12 <= R12;
    C13 <= R13;
    C14 <= R14;
    C15 <= R15;
    C16 <= R16;
    C17 <= R17;
    C18 <= R18;
    C19 <= R19;
    C20 <= R20;
    C21 <= R21;
    C22 <= R22;
    C23 <= R23;
    C24 <= R24;
    C25 <= R25;
    C26 <= R26;
    C27 <= R27;
    C28 <= R28;
    C29 <= R29;
    C30 <= R30;
    C31 <= R31;
    C32 <= R32;
    C33 <= R33;
    C34 <= R34;
    C35 <= R35;
    C36 <= R36;
    C37 <= R37;
    C38 <= R38;
    C39 <= R39;
    C40 <= R40;
    C41 <= R41;
    C42 <= R42;
    C43 <= R43;
    C44 <= R44;
    C45 <= R45;
    C46 <= R46;
```

```
  C47 <= R47;
  C48 <= R48;
  C49 <= R49;
  C50 <= R50;
  C51 <= R51;
  C52 <= R52;
  C53 <= R53;
  C54 <= R54;
  C55 <= R55;
  C56 <= R56;
  C57 <= R57;
  C58 <= R58;
  C59 <= R59;
  C60 <= R60;
  C61 <= R61;
  C62 <= R62;
  C63 <= R63;
  BufferFull <= 1;                // indicates that the buffer is
  end                             // ready to output data

else if(DataInReady & BufferFull) // The following represent
  begin                           // data flow through the
  C8 <= C0;                       // output stage
  C16 <= C8;
  C24 <= C16;
  C32 <= C24;
  C40 <= C32;
  C48 <= C40;
  C56 <= C48;
  C1 <= C56;
  C9 <= C1;
  C17 <= C9;
  C25 <= C17;
  C33 <= C25;
  C41 <= C33;
  C49 <= C41;
  C57 <= C49;
  C2 <= C57;
  C10 <= C2;
  C18 <= C10;
  C26 <= C18;
  C34 <= C26;
  C42 <= C34;
  C50 <= C42;
  C58 <= C50;
  C3 <= C58;
  C11 <= C3;
  C19 <= C11;
  C27 <= C19;
  C35 <= C27;
  C43 <= C35;
  C51 <= C43;
  C59 <= C51;
  C4 <= C59;
  C12 <= C4;
  C20 <= C12;
  C28 <= C20;
  C36 <= C28;
  C44 <= C36;
  C52 <= C44;
  C60 <= C52;
```

```
    C5 <= C60;
    C13 <= C5;
    C21 <= C13;
    C29 <= C21;
    C37 <= C29;
    C45 <= C37;
    C53 <= C45;
    C61 <= C53;
    C6 <= C61;
    C14 <= C6;
    C22 <= C14;
    C30 <= C22;
    C38 <= C30;
    C46 <= C38;
    C54 <= C46;
    C62 <= C54;
    C7 <= C62;
    C15 <= C7;
    C23 <= C15;
    C31 <= C23;
    C39 <= C31;
    C47 <= C39;
    C55 <= C47;
    C63 <= C55;
    end
  end
 end

endmodule
```

```
/*****************************************************************/
/*    This module controls engine of Wallace Tree Multiplier Stage */
/*****************************************************************/

module multiplier(      Reset,
ValidInput,
Enable,
Clk,
DataIn,
DataOut,
ValidOutput);


input             Reset;            // System wide Reset
input             Clk;              // System wide Clock
input             Enable;           // System wide Enable
input             ValidInput;       // Arrival of Valid Data signal
input [10:0]      DataIn;           // Input Data (Multiplier)

output [10:0]     DataOut;          // Data Output
output            ValidOutput;      // Signal of valid data

reg               ValidOutput;
reg [10:0]        DataOut;

reg               WorkEngine;       //To keep engine working
wire              EnableEngine;
reg [10:0]        Data1;            //For keeping input data till
reg [10:0]        Data2;            //Rom outputs first coefficient
reg [10:0]        Data3;
reg [18:0]        Result;           //Final output
reg [7:0]         Coefficient;      //Coefficient to be multiplied
reg [7:0]         Index;            //To keep track of valid output
reg [3:0]         Index2;           //To keep engine working equal to
                                    // its latancy
reg [6:0]         Address;


//Multiplier Engine Instantiation
engine engine_inst( Reset, EnableEngine, Clk, Data3, Coefficient,
      Result);

//Rom instantiation
Rom RomInst(Address, Clk, Clk, Coefficient);

//    Ist Clk -> Latches Address, 2nd Clk -> Gives Coefficient of
//    Latched Address.


//Enabling the engine of multiplier
assign EnableEngine = Enable & WorkEngine;


always @(posedge Clk or posedge ValidInput)
begin
      if(Reset)
      begin
            Index2 <= 0;
            WorkEngine <= 0;
      end
```

```
      else if(ValidInput)           //if valid data is coming
      begin                         //then
            WorkEngine <= 1;        //keep engine working
            Index2 <= 0;
      End
      else                          //else
      begin
            if(Index2 != 8)         //wait for the latency
                  Index2 <= Index2 + 1;
            else
                  WorkEngine <= 0;  //then stop engine
      end
      Data1 <= DataIn;              //store data
      Data2 <= Data1;               //till a coefficient is fetched
      Data3 <= Data2;               //from the rom
end

always @(posedge Clk)
begin
      if(Reset)
      begin
            Index <= 0;
            Address <= 0;
      end
      else if(EnableEngine)
      begin
            if(Index > 6 & Index2 != 8)
            begin                         //wait till pipeline of
                  ValidOutput <= 1;       // engine gets fulled
                  DataOut <= Result ;     // output the and
            end                           // then result
            else
                  ValidOutput <= 0;

            if(Address == 63)
            begin
                  Address <= 0;
                  Index <= 7;
            end
            else
            begin
                  Index <= Index + 1;
                  Address <= Address + 1; //for fetching
            End                           // next coefficient
      end
end
endmodule
```

```
/*****************************************************************/
/*    This module implements Wallace Tree Multiplier Stage       */
/*****************************************************************/

module engine(    Reset,
Enable,
Clk,
DataIn,
Coefficient,
Result7);

input Reset;            //Reset is actually stall signal.
input Clk;              //Clock Signal
input Enable;           //System wide Enable


input [10:0] DataIn;    //Multiplicand
input [7:0] Coefficient;//Multiplier
output [18:0] Result7;  //Final Result

reg [7:0]   CoefficientLatch; //Internal Copy of Coefficient
reg [10:0]  DataInLatch;      //Internal Copy of DataIn
reg [9:0]   p1;               //partial products
reg [9:0]   p2;               //These contain copies of DataIn
reg [9:0]   p3;               //depending on coefficient
reg [9:0]   p4;
reg [9:0]   p5;
reg [9:0]   p6;
reg [9:0]   p7;
reg [9:0]   p8;

reg Signr1;                   //These are registers that tell if the
reg Signr2;                   //result in various stages is in 2s
reg Signr3;                   //complement form or not
reg Signr4;
reg Signr5;


wire [9:1] carryr1, carryr2, carryr3, carryr4;
wire [10:0] carryr5, carryr6;
wire [12:0] carryr7;
wire [16:0] carryr8;

wire [11:1] R1, R2, R3, R4;   //These wires have partial results
wire [14:2] R5, R6;           //at them
wire [18:4] R7;

reg [11:0] Result1, Result2, Result3, Result4;
reg [14:0] Result5, Result6;  //The partial results are stored in
reg [18:0] Result7;           //these registers


//R1 = p1 + p2

adder r1c1(p1[1], p2[0], 0, R1[1], carryr1[1]);
adder r1c2(p1[2], p2[1], carryr1[1], R1[2], carryr1[2]);
adder r1c3(p1[3], p2[2], carryr1[2], R1[3], carryr1[3]);
adder r1c4(p1[4], p2[3], carryr1[3], R1[4], carryr1[4]);
adder r1c5(p1[5], p2[4], carryr1[4], R1[5], carryr1[5]);
adder r1c6(p1[6], p2[5], carryr1[5], R1[6], carryr1[6]);
adder r1c7(p1[7], p2[6], carryr1[6], R1[7], carryr1[7]);
```

```
adder r1c8(p1[8], p2[7], carryr1[7], R1[8], carryr1[8]);
adder r1c9(p1[9], p2[8], carryr1[8], R1[9], carryr1[9]);
adder r1c10(0, p2[9], carryr1[9], R1[10], R1[11]);


//R2 = p3 + p4

adder r2c3(p3[1], p4[0], 0, R2[1], carryr2[1]);
adder r2c4(p3[2], p4[1], carryr2[1], R2[2], carryr2[2]);
adder r2c5(p3[3], p4[2], carryr2[2], R2[3], carryr2[3]);
adder r2c6(p3[4], p4[3], carryr2[3], R2[4], carryr2[4]);
adder r2c7(p3[5], p4[4], carryr2[4], R2[5], carryr2[5]);
adder r2c8(p3[6], p4[5], carryr2[5], R2[6], carryr2[6]);
adder r2c9(p3[7], p4[6], carryr2[6], R2[7], carryr2[7]);
adder r2c10(p3[8], p4[7], carryr2[7], R2[8], carryr2[8]);
adder r2c11(p3[9], p4[8], carryr2[8], R2[9], carryr2[9]);
adder r2c12(0, p4[9], carryr2[9], R2[10], R2[11]);


//R3 = p5 + p6

adder r3c5(p5[1], p6[0], 0, R3[1], carryr3[1]);
adder r3c6(p5[2], p6[1], carryr3[1], R3[2], carryr3[2]);
adder r3c7(p5[3], p6[2], carryr3[2], R3[3], carryr3[3]);
adder r3c8(p5[4], p6[3], carryr3[3], R3[4], carryr3[4]);
adder r3c9(p5[5], p6[4], carryr3[4], R3[5], carryr3[5]);
adder r3c10(p5[6], p6[5], carryr3[5], R3[6], carryr3[6]);
adder r3c11(p5[7], p6[6], carryr3[6], R3[7], carryr3[7]);
adder r3c12(p5[8], p6[7], carryr3[7], R3[8], carryr3[8]);
adder r3c13(p5[9], p6[8], carryr3[8], R3[9], carryr3[9]);
adder r3c14(0, p6[9], carryr3[9], R3[10], R3[11]);


//R4 = p7 + p8

adder r4c7(p7[1], p8[0], 0, R4[1], carryr4[1]);
adder r4c8(p7[2], p8[1], carryr4[1], R4[2], carryr4[2]);
adder r4c9(p7[3], p8[2], carryr4[2], R4[3], carryr4[3]);
adder r4c10(p7[4], p8[3], carryr4[3], R4[4], carryr4[4]);
adder r4c11(p7[5], p8[4], carryr4[4], R4[5], carryr4[5]);
adder r4c12(p7[6], p8[5], carryr4[5], R4[6], carryr4[6]);
adder r4c13(p7[7], p8[6], carryr4[6], R4[7], carryr4[7]);
adder r4c14(p7[8], p8[7], carryr4[7], R4[8], carryr4[8]);
adder r4c15(p7[9], p8[8], carryr4[8], R4[9], carryr4[9]);
adder r4c16(0, p8[9], carryr4[9], R4[10], R4[11]);


//R5 = Result1 + Result2

adder r5c2(Result1[2], Result2[0], 0, R5[2], carryr5[0]);
adder r5c3(Result1[3], Result2[1], carryr5[0], R5[3], carryr5[1]);
adder r5c4(Result1[4], Result2[2], carryr5[1], R5[4], carryr5[2]);
adder r5c5(Result1[5], Result2[3], carryr5[2], R5[5], carryr5[3]);
adder r5c6(Result1[6], Result2[4], carryr5[3], R5[6], carryr5[4]);
adder r5c7(Result1[7], Result2[5], carryr5[4], R5[7], carryr5[5]);
adder r5c8(Result1[8], Result2[6], carryr5[5], R5[8], carryr5[6]);
adder r5c9(Result1[9], Result2[7], carryr5[6], R5[9], carryr5[7]);
adder r5c10(Result1[10], Result2[8], carryr5[7], R5[10], carryr5[8]);
adder r5c11(Result1[11], Result2[9], carryr5[8], R5[11], carryr5[9]);
adder r5c12(0, Result2[10], carryr5[9], R5[12], carryr5[10]);
adder r5c13(0, Result2[11], carryr5[10], R5[13], R5[14]);
```

```
//R6 = Resul3 + Result4

adder r6c2(Result3[2], Result4[0], 0, R6[2], carryr6[0]);
adder r6c3(Result3[3], Result4[1], carryr6[0], R6[3], carryr6[1]);
adder r6c4(Result3[4], Result4[2], carryr6[1], R6[4], carryr6[2]);
adder r6c5(Result3[5], Result4[3], carryr6[2], R6[5], carryr6[3]);
adder r6c6(Result3[6], Result4[4], carryr6[3], R6[6], carryr6[4]);
adder r6c7(Result3[7], Result4[5], carryr6[4], R6[7], carryr6[5]);
adder r6c8(Result3[8], Result4[6], carryr6[5], R6[8], carryr6[6]);
adder r6c9(Result3[9], Result4[7], carryr6[6], R6[9], carryr6[7]);
adder r6c10(Result3[10], Result4[8], carryr6[7], R6[10], carryr6[8]);
adder r6c11(Result3[11], Result4[9], carryr6[8], R6[11], carryr6[9]);
adder r6c12(0, Result4[10], carryr6[9], R6[12], carryr6[10]);
adder r6c13(0, Result4[11], carryr6[10], R6[13], R6[14]);


//R7 = Result5 + Result6

adder Result7c4(Result5[4], Result6[0], 0, R7[4], carryr7[0]);
adder Result7c5(Result5[5], Result6[1], carryr7[0], R7[5],
      carryr7[1]);
adder Result7c6(Result5[6], Result6[2], carryr7[1], R7[6],
      carryr7[2]);
adder Result7c7(Result5[7], Result6[3], carryr7[2], R7[7],
      carryr7[3]);
adder Result7c8(Result5[8], Result6[4], carryr7[3], R7[8],
      carryr7[4]);
adder Result7c9(Result5[9], Result6[5], carryr7[4], R7[9],
      carryr7[5]);
adder Result7c10(Result5[10], Result6[6], carryr7[5], R7[10],
      carryr7[6]);
adder Result7c11(Result5[11], Result6[7], carryr7[6], R7[11],
      carryr7[7]);
adder Result7c12(Result5[12], Result6[8], carryr7[7], R7[12],
      carryr7[8]);
adder Result7c13(Result5[13], Result6[9], carryr7[8], R7[13],
      carryr7[9]);
adder Result7c14(0, Result6[10], carryr7[9], R7[14], carryr7[10]);
adder Result7c15(0, Result6[11], carryr7[10], R7[15], carryr7[11]);
adder Result7c16(0, Result6[12], carryr7[11], R7[16], carryr7[12]);
adder Result7c17(0, Result6[13], carryr7[12], R7[17], R7[18]);

always @(posedge Clk)
begin
      if(Reset)
      begin

      end
      else if(Enable)
      begin
            CoefficientLatch <= Coefficient;
            if(DataIn[10])               //if input is negative
            begin                        //latch positive value
                  DataInLatch <= -DataIn; //and set sign flag
                  Signr1 <= 1;
            end
            else
            begin
                  DataInLatch <= DataIn;  //else latch it as it is
                  Signr1 <= 0;            //reset sign bit
            end
```

```
if(CoefficientLatch[0])       //latch multiplicand
      p1 <= DataInLatch;      //according to multiplier
else
      p1 <= 0;
if(CoefficientLatch[1])
      p2 <= DataInLatch ;
else
      p2 <= 0;
if(CoefficientLatch[2])
      p3 <= DataInLatch ;
else
      p3 <= 0;
if(CoefficientLatch[3])
      p4 <= DataInLatch ;
else
      p4 <= 0;
if(CoefficientLatch[4])
      p5 <= DataInLatch ;
else
      p5 <= 0;
if(CoefficientLatch[5])
      p6 <= DataInLatch ;
else
      p6 <= 0;
if(CoefficientLatch[6])
      p7 <= DataInLatch ;
else
      p7 <= 0;
if(CoefficientLatch[7])
      p8 <= DataInLatch ;
else
      p8 <= 0;

if(Signr1)                    //propogate sign bits
      Signr2 <= 1;
else
      Signr2 <= 0;

if(Signr2)
      Signr3 <= 1;
else
      Signr3 <= 0;

if(Signr3)
      Signr4 <= 1;
else
      Signr4 <= 0;

Result1 <= {R1, p1[0]};       //latch partial results
Result2 <= {R2, p3[0]};
Result3 <= {R3, p5[0]};
Result4 <= {R4, p7[0]};
Result5 <= {R5, Result1[1], Result1[0]};
Result6 <= {R6, Result3[1], Result3[0]};
```

```
            if(Signr4)              //if originally multiplicand
            begin                   //was negative then reverse the
                                    //sign after removing the
                                    //biasing and latch the result

Result7 <= - ({R7, Result5[3], Result5[2], Result5[1], Result5[0]} >>
      8 );
            end
            else
            begin
Result7  <= ( {R7, Result5[3], Result5[2], Result5[1], Result5[0]} >>
      8 );

            end
      end
end

endmodule
```

```
/*****************************************************************/
/*                This module controls Zig Zag Engine             */
/*****************************************************************/


module ZigZag(    DataIn,
Clk,
Reset,
Enable,
ValidInput,
ValidOutput,
DataOut,
StartD,
EndD);


input       Clk;              //Clock
input       Reset;            //Reset
input       Enable;           //Enable
input       ValidInput;       //Input is valid
input[11:0] DataIn;           //Input data

output      ValidOutput;      //Output is valid
output      StartD            //Signal for start of data
output      EndD;             //Signal for end of data
output[11:0] DataOut;         //Output after ZigZag

reg         StartD;
reg         EndD;
reg [11:0] DataOut;
reg ValidOutput;

reg [6:0]   Index;
reg [2:0]   CurrentState, NextState;
parameter[1:0]    ResetState = 0,         //State Machine's states
ValidOutputState = 1,
EndPulseFall = 2;
reg [11:0] DataInLatch;             //For latching input
reg [11:0] ResetEngine;             //Signal for resetting engine
reg WorkEngineZZ;                   //Internal enable for engine
reg EngineWorking;                  //Flag showing status of engine


wire [11:0] DataOutEngine;          //Output from the engine
wire EnableZZ                       //Enable for the engine
wire ResetZZ;                       //Reset for the engine

//Instantiation of the engine
EngineZigZag engine_inst_zz(EnableZZ, DataInLatch, DataOutEngine,
ValidOutputEngine, Clk, ResetZZ);

assign EnableZZ= Enable & WorkEngineZZ;
assign ResetZZ = Reset | ResetEngine;
```

```
always @(posedge Clk or posedge Reset)
begin
      if(Reset)
      begin
EngineWorking <= 0;     //a Flag for this module telling
            Index <= 0;             // that Engine is processing data
            WorkEngineZZ <= 0;      //To Enable/Disable Engine
  end
      else if(Enable)
      begin
            if(ValidInput)
            begin

                  WorkEngineZZ <= 1;      //Start the engine
                  DataInLatch <= DataIn;  //Latch Data
                  Index <= 0;
                  EngineWorking <= 1;     //Set engine working flag
            end
            else if(EngineWorking)  //Valid data has stopped coming
                                    //keep the engine working till
            begin                   //pipelined is Flush
                  if(Index == 65)   //Data has traversed the pipeline
                  begin
                        WorkEngineZZ <= 0;//Stop the engine and
                        EngineWorking <= 0;//reset the working flag
                  end
                  else
                  begin             //keep working
                        Index <= Index + 1;
                        DataInLatch <= DataIn;
                  end

            end

      end

end


always @(posedge Clk)   //State Machine for producing Start and
begin                   //End Pulses
      if(Reset)
            CurrentState <= ResetState;
      else
            CurrentState <= NextState;

end

always @(CurrentState)
begin
      case (CurrentState)
            ResetState:       //Reset State
            Begin             //reset the signals
                  ResetEngine = 0;
                  StartD = 0;
                  EndD = 0;
                  NextState = ResetState;
                  ValidOutput = 0;
                  if(ValidOutputEngine)
                  begin       //produce start data pulse
                        StartD = 1;
```

```
                        NextState = ValidOutputState;
                  end
                  else
                        NextState = ResetState;
            end

            ValidOutputState:
            Begin
                  StartD = 0;             //produce falling edge
                  if(ValidOutputEngine)
                  begin
                        NextState = ValidOutputState;
                        ValidOutput = 1;  //output is valid
                  end
                  else                    //engine has stopped
                  begin                   //giving valid output
                        NextState = EndPulseFall;
                        ValidOutput = 0;
                        EndD = 1;
                  end
            end

            EndPulseFall:                 //provide end pulse
            begin
                  EndD = 0;
                  ResetEngine = 1;
                  NextState = ResetState;
            end
      endcase
end

always @(posedge Clk)
begin
      DataOut <= DataOutEngine;     //Latch the output of the
end                                 //engine to output



endmodule
```

```
/*****************************************************************/
/*    This module implements the zig zag buffer Stage of the     */
/*    encoder model.  It has the same functionality as the       */
/*    transpose buffer except for the data transfers in the      */
/*    Q reqisters which take place in a zig zag fashion.         */
/*****************************************************************/

module EngineZigZag(    DataInReady,
DataIn,
Clock,
Reset,
DataOut,
DataOutReady
);

input                   DataInReady;
input [11:0]            DataIn;
input                   Clock;
input                   Reset;

output [11:0]           DataOut;
output                  DataOutReady;

reg                     BufferFull;

reg [6:0]               Count;

reg [11:0]  R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12,
      R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24,
      R25, R26, R27, R28, R29, R30, R31, R32, R33, R34, R35, R36,
      R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48,
      R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60,
      R61, R62, R63;

reg [11:0]  C0, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12,
      C13, C14, C15, C16, C17, C18, C19, C20, C21, C22, C23, C24,
      C25, C26, C27, C28, C29, C30, C31, C32, C33, C34, C35, C36,
      C37, C38, C39, C40, C41, C42, C43, C44, C45, C46, C47, C48,
      C49, C50, C51, C52, C53, C54, C55, C56, C57, C58, C59, C60,
      C61, C62, C63;

assign DataOut = C63;

assign DataOutReady = DataInReady & BufferFull;

always@(posedge Clock or posedge Reset)
  begin
  if(Reset)
    Count <= 0;
  else
    begin
    if(DataInReady)
      begin
      R0 <= DataIn;
      R1 <= R0;
      R2 <= R1;
      R3 <= R2;
      R4 <= R3;
      R5 <= R4;
      R6 <= R5;
```

```
      R7 <= R6;
      R8 <= R7;
      R9 <= R8;
      R10 <= R9;
      R11 <= R10;
      R12 <= R11;
      R13 <= R12;
      R14 <= R13;
      R15 <= R14;
      R16 <= R15;
      R17 <= R16;
      R18 <= R17;
      R19 <= R18;
      R20 <= R19;
      R21 <= R20;
      R22 <= R21;
      R23 <= R22;
      R24 <= R23;
      R25 <= R24;
R26 <= R25;
      R27 <= R26;
      R28 <= R27;
      R29 <= R28;
      R30 <= R29;
      R31 <= R30;
      R32 <= R31;
      R33 <= R32;
      R34 <= R33;
      R35 <= R34;
      R36 <= R35;
      R37 <= R36;
    R38 <= R37;
    R39 <= R38;
    R40 <= R39;
    R41 <= R40;
    R42 <= R41;
    R43 <= R42;
    R44 <= R43;
    R45 <= R44;
    R46 <= R45;
    R47 <= R46;
    R48 <= R47;
    R49 <= R48;
    R50 <= R49;
    R51 <= R50;
    R52 <= R51;
    R53 <= R52;
    R54 <= R53;
    R55 <= R54;
    R56 <= R55;
    R57 <= R56;
    R58 <= R57;
    R59 <= R58;
    R60 <= R59;
    R61 <= R60;
    R62 <= R61;
    R63 <= R62;
    if(Count==64)
      Count <= 1;
    else
      Count <= Count + 1;
```

```
    end
  end
end



always@(posedge Clock or posedge Reset)
  begin
  if(Reset)
    BufferFull <= 0;
  else
  begin
  if(Count==64)
    begin
    C0 <= R0;
    C1 <= R1;
    C2 <= R2;
    C3 <= R3;
    C4 <= R4;
    C5 <= R5;
    C6 <= R6;
    C7 <= R7;
    C8 <= R8;
    C9 <= R9;
    C10 <= R10;
    C11 <= R11;
    C12 <= R12;
    C13 <= R13;
    C14 <= R14;
    C15 <= R15;
    C16 <= R16;
    C17 <= R17;
    C18 <= R18;
    C19 <= R19;
    C20 <= R20;
    C21 <= R21;
    C22 <= R22;
    C23 <= R23;
    C24 <= R24;
    C25 <= R25;
    C26 <= R26;
    C27 <= R27;
    C28 <= R28;
    C29 <= R29;
    C30 <= R30;
    C31 <= R31;
    C32 <= R32;
    C33 <= R33;
    C34 <= R34;
    C35 <= R35;
    C36 <= R36;
    C37 <= R37;
    C38 <= R38;
    C39 <= R39;
    C40 <= R40;
    C41 <= R41;
    C42 <= R42;
    C43 <= R43;
    C44 <= R44;
    C45 <= R45;
    C46 <= R46;
```

```
  C47 <= R47;
  C48 <= R48;
  C49 <= R49;
  C50 <= R50;
  C51 <= R51;
  C52 <= R52;
  C53 <= R53;
  C54 <= R54;
  C55 <= R55;
  C56 <= R56;
  C57 <= R57;
  C58 <= R58;
  C59 <= R59;
  C60 <= R60;
  C61 <= R61;
  C62 <= R62;
  C63 <= R63;
  BufferFull <= 1;
  end

else if(DataInReady & BufferFull)
  begin
  C1 <= C0;
  C8 <= C1;
  C16 <= C8;
  C9 <= C16;
  C2 <= C9;
  C3 <= C2;
  C10 <= C3;
  C17 <= C10;
  C24 <= C17;
  C32 <= C24;
  C25 <= C32;
  C18 <= C25;
  C11 <= C18;
  C4 <= C11;
  C5 <= C4;
  C12 <= C5;
  C19 <= C12;
  C26 <= C19;
  C33 <= C26;
  C40 <= C33;
  C48 <= C40;
  C41 <= C48;
  C34 <= C41;
  C27 <= C34;
  C20 <= C27;
  C13 <= C20;
  C6 <= C13;
  C7 <= C6;
  C14 <= C7;
  C21 <= C14;
  C28 <= C21;
  C35 <= C28;
  C42 <= C35;
  C49 <= C42;
  C56 <= C49;
  C57 <= C56;
  C50 <= C57;
  C43 <= C50;
  C36 <= C43;
```

```
    C29 <= C36;
    C22 <= C29;
    C15 <= C22;
    C23 <= C15;
    C30 <= C23;
    C37 <= C30;
    C44 <= C37;
    C51 <= C44;
    C58 <= C51;
    C59 <= C58;
    C52 <= C59;
    C45 <= C52;
    C38 <= C45;
    C31 <= C38;
    C39 <= C31;
    C46 <= C39;
    C53 <= C46;
    C60 <= C53;
    C61 <= C60;
    C54 <= C61;
    C47 <= C54;
    C55 <= C47;
    C62 <= C55;
    C63 <= C62;
    end
  end
 end

endmodule
```

```
/*****************************************************************/
/*    This module implements the controller for the Run Length   */
/*    Encoder stage of the Entropy encoder                       */
/*****************************************************************/

module NewRLCCtrl(      Clock,
                        Reset,
                        Start,
                        Enable,
                        Done,
                        Over,
                        Countgey,
                        Run16,
                        ZeroCoeff,
                        LoadData,
                        RSTCount,
                        IncrCount,
                        MuxSel,
                        LoadMuxOut,
                        LoadPDC,
                        LoadZC,
                        IncrZC,
                        RSTZC,
                        RSTPDC,
                        sDC,
                        sAC,
                        sZRL,
                        sEOB,
                        sEOB2,
                        sDone,
                        LoadCoeffOut,
                        RSTZCO);

input           Clock;          // Clock signal
input           Reset;          // Low asserted system-wide reset
input           Start;          // Signal implying start of data
input           Enable;         // System-wide enable signal
input           EndData;        // Signal implying end of data
input           Over;           // Signal implying that last data
                                // has been processed
input           Countgey;       // Signal implying Count has
                                // its max value
input           Run16;          // Sinal implying zero count has
                                // reached its max value
input           ZeroCoeff;      // Signal implying a zero
                                // coefficient

output          LoadData;       // Signals loading of new data
output          RSTCount;       // Resets the counter value
output          IncrCount;      // Increments the counter value
output          MuxSel;         // Mux selector input
output          LoadMuxOut;     // Signals loading of mux output
output          LoadPDC;        // Signals loading of DC(I-1) value
output          LoadZC;         // Signals loading of zero count
output          IncrZC;         // Increments the zero count
output          RSTZC;          // Resets the zero count
output          RSTPDC;         // Resets the DC(I-1)
output          sDC;            // Sets the DC status signal
output          sAC;            // Sets the AC status signal
output          sZRL;           // Sets the ZRL status signal
output          sEOB;           // Sets the EOB status signal
```

```
output            sEOB2;            // Sets the BlkEnd status signal
output            LoadCoeffOut;     // Signals loading of Data out
output            RSTZCO;           // Resets the zero counter out
output            sDone;            // Sets the last data in flag

reg         LoadData;
reg         RSTCount;
reg         IncrCount;
reg         MuxSel;
reg         LoadMuxOut;
reg         LoadPDC;
reg         LoadZC;
reg         IncrZC;
reg         RSTZC;
reg         RSTPDC;
reg         sDC;
reg         sAC;
reg         sZRL;
reg         sEOB;
reg         sEOB2;
reg         sDone;
reg         LoadCoeffOut;
reg         RSTZCO;

reg [1:0]   CurrState;              // Current state register
reg [1:0]   NextState;              // Next state register

parameter [1:0]   ST0 = 2'b00,      // State names
ST1 = 2'b01,
ST2 = 2'b10,
ST3 = 2'b11;

parameter         DC = 1'b0,        // Mux inputs
AC = 1'b1;

// FSM current state logic
always@(posedge Clock or negedge Reset)
      begin: FSM_SEQ
      if(!Reset)
             CurrState = ST0;
      else
            CurrState = NextState;
      end

// FSM next state and output logic
always @ (  CurrState or  Start or Enable or Done or Over or Countgey
      or Run16 or ZeroCoeff   )
      begin: FSM_COMB
      LoadData=0;
      RSTCount=0;
      IncrCount=0;
      MuxSel=0;
      LoadMuxOut=0;
      LoadPDC=0;
      LoadZC=0;
      IncrZC=0;
      RSTZC=0;
      RSTPDC=0;
      sDC=0;
      sAC=0;
      sZRL=0;
```

```
      sEOB=0;
      LoadCoeffOut=0;
      RSTZCO = 0;
      sDone = 0;
      sEOB2 = 0;

      case(CurrState)

ST0:  if(Start & Enable)      // if arrival of data has begun
            begin
            LoadData = 1;
            NextState = ST1;
            end
      else
            begin             // else remain in reset state
            RSTCount = 1;
            RSTZC = 1;
            RSTPDC = 1;
            NextState = ST0;
            end

      ST1:  if(Enable)              // generate signals for latching
            begin             // current DC coefficient and
            LoadData = 1;     // select the difference of DC
            LoadPDC = 1;      // coefficients as the mux output
            MuxSel = DC;
            LoadMuxOut = 1;
            NextState = ST2;
            end
      else
            NextState = ST1;

      ST2:  if(Enable)              // set the DC status signal and
            Begin             // begin the count of incoming
            LoadData = 1;     // coefficients
            MuxSel = AC;
            LoadMuxOut = 1;
            LoadCoeffOut = 1;
            RSTZCO = 1;
            sDC = 1;
            IncrCount = 1;
            NextState = ST3;
            end
      else
            NextState = ST2;

      ST3:  if(Enable)
            begin
            LoadData = 1;
            MuxSel = AC;
            LoadMuxOut = 1;
            LoadCoeffOut = 1;
            IncrCount = 1;
            if(Done)                // if last data has arrived
                  sDone = 1;        // set the last data flag
            if(ZeroCoeff)           // if coefficient is zero
                  begin
                  if(Countgey)      // Count equals 64?
                        Begin
                        sEOB = 1;   // output EOB status
                        sEOB2 = 1;  // output BlkEnd status
```

```
                        if(Over)    // if this is last data
                              NextState = ST0; //finish work
                        else
                              begin
                              LoadPDC = 1;      // prepare
                              MuxSel = DC;      // for new
                              RSTZC = 1;        // block
                              RSTZCO = 1;
                              NextState = ST2;
                              end //end else
                        end  //end if(Countgey)
                  else
                        begin       // if block not ended
                        IncrZC = 1; // adjust the zero count
                        LoadZC = 1; // load ZC value
                        if(Run16)
                              sZRL = 1;
                        NextState = ST3;
                        end //end else
                  end // end if(ZeroCoeff)
            else
                  begin             // if coefficient non-zero
                  RSTZC = 1;        // reset ZC
                  LoadZC = 1;       // load current value
                  sAC = 1;          // set AC status
                  if(Countgey)      // Count equals 64?
                        begin
                        sEOB2 = 1;  // set BlkEnd status
                        if(Over)    // if this is last data
                              NextState = ST0;  // finish work
                        else
                              begin
                              LoadPDC = 1;      // else
                              MuxSel = DC;      // prepare for
                              NextState = ST2;  // new block
                              end //end else
                        end //end if(Countgey)
                  else
                        NextState = ST3;
                  end //end else
            end //end if(Enable)
      else
            NextState = ST3;

default: NextState = ST0;
endcase
end

endmodule
```

```
/*****************************************************************/
/*    This module implements the architecture for the Run Length   */
/*    Encoder stage of the Entropy encoder                         */
/*****************************************************************/
module RLCArch(   Clock,
                  Reset,
                  DataIn,
                  LoadData,
                  RSTCount,
                  IncrCount,
                  MuxSel,
                  LoadMuxOut,
                  LoadPDC,
                  LoadZC,
                  IncrZC,
                  RSTZC,
                  RSTPDC,
                  sDC,
                  sAC,
                  sZRL,
                  sEOB,
                  sEOB2,
                  sDone,
                  LoadCoeffOut,
                  RSTZCO,
                  Countgey,
                  Run16,
                  ZeroCoeff,
                  Over,
                  DC,
                  AC,
                  ZRL,
                  EOB,
                  EOB2,
                  CoeffOut,
                  Zcout
                  );

input             Clock;            // Clock signal
input             Reset;            // Low asserted system-wide reset
input             LoadData;         // Signals loading of new data
input             RSTCount;         // Resets the coefficient counter
input             IncrCount;        // Increments coefficient counter
input             MuxSel;           // Mux Select
input             LoadMuxOut;       // Signals loading of mux output
input             LoadPDC;          // Signals loading of DC value
input             LoadZC;           // Signals loading of zero count
input             IncrZC;           // Increments the zero count
input             RSTZC;            // Resets the zero count
input             RSTPDC;           // Resets the DC(I-1)
input             sDC;              // Set the DC status flag
input             sAC;              // Set the AC status flag
input             sZRL;             // Set the ZRL status flag
input             sEOB;             // Set the EOB status flag
input             sEOB2;            // Set the BlkEnd status flag
input             LoadCoeffOut;     // Signals loading of data out
input             RSTZCO;           // Resets the ZC output register
input             sDone;            // Sets the last data in flag
input [11:0]      DataIn;           // Data input from Zig Zag stage

output            Countgey;         // Indicates counter==63
```

```
output            Run16;            // Indicates zero count==16
output            ZeroCoeff;        // Indicates a zero coefficient
output            DC;               // DC status output
output            AC;               // AC status output
output            ZRL;              // ZRL status output
output            EOB;              // EOB status output
output            EOB2;             // BlkEnd status output
output            Over;             // Indicates the last data
output [11:0]     CoeffOut;         // Processed coefficient output
output [3:0]      ZCout;            // zero count output

reg               DC;
reg               AC;
reg               ZRL;
reg               EOB;
reg               EOB2;
reg               Over;
reg [11:0]        CoeffOut;
reg [3:0]         ZC;               // zero counter
reg [3:0]         ZCout;

wire              Countgey;
wire              Run16;
wire              ZeroCoeff;

reg [11:0]        Data;             // Register for input data
reg [11:0]        MuxOutReg;        // Holds the output of mux
reg [11:0]        PDC;              // Holds the DC_I-1 value
reg [5:0]         Count;            // Counter for coefficients
wire [11:0]       MuxOut;
wire [11:0]       SubPDC;
wire [11:0]       SubMSB;

assign MuxOut = MuxSel ? Data:SubPDC;     // 2:1 mux
assign SubPDC = Data - PDC;               // subtractor for DC_I - DC_I-1
assign SubMSB = MuxOutReg - MuxOutReg[11];// For decrementing –ve
                                          // valued coefficients

assign Countgey = Count<63?0:1;           // Comparator for count
assign ZeroCoeff = MuxOutReg==0?1:0;      // zero Comparator
assign Run16 = ZC==15?1:0;                // runlength comparator

always@(posedge Clock)
begin
      if(LoadData)
            Data <= DataIn;
      end

always@(posedge Clock)
begin
      if(LoadMuxOut)
            MuxOutReg <= MuxOut;
      end

always@(posedge Clock)
begin
      if(LoadCoeffOut)
            CoeffOut <= SubMSB;
      end

always@(posedge Clock)
```

```
begin
      if(RSTPDC)
            PDC <= 0;
      if(LoadPDC)
            PDC <= Data;
      end

always@(posedge Clock)
begin
      if(IncrCount)
            Count <= Count + 1;
      if(sDone)
            Over <= 1;
      if(RSTCount)
            begin
            Count <= 0;
            Over <= 0;
            end
      end

always@(posedge Clock)
      begin
      if(IncrZC)
            ZC <= ZC + 1;
      if(LoadZC)
            ZCout <= ZC;
      if(RSTZC)
            ZC <= 0;
      if(RSTZCO)
            ZCout <= 0;
      end

always@(posedge Clock)
begin
      if(sAC)
            AC <= 1;
      else
            AC <= 0;

      if(sDC)
            DC <= 1;
      else
            DC <= 0;

      if(sZRL)
            ZRL <= 1;
      else
            ZRL <= 0;

      if(sEOB)
            EOB <= 1;
      else
            EOB <= 0;


      if(sEOB2)
            EOB2 <= 1;
      else
            EOB2 <= 0;
end
endmodule
```

```
/*****************************************************************/
/*    This module implements the category selection circuit      */
/*    of the Entropy encoder                                     */
/*****************************************************************/

module categoryselect(  Coefficient,
RunlengthIn,
                        DCin,
                        ACin,
                        EOBin,
EOB2in,
                        ZRLin,
                        Clock,
                        Reset,
                        Enable,
                        Category,
                        CoefficientOut,
                        RunLengthOut,
                        DCOut,
                        ACOut,
                        EOBOut,
                        EOB2out,
                        ZRLOut
                      );

input [11:0]      Coefficient;      // Coefficient from RLC stage
input [3:0]       RunlengthIn;      // Run length from RLC stage
input             Clock;            // Clock signal
input             Reset;            // Low asserted System-wide reset
input             DCin;             // DC status signal
input             ACin;             // AC status signal
input             EOBin;            // EOB status signal
input             ZRLin;            // ZRL status signal
input             EOB2in;           // BlkEnd status signal
input             Enable;           // System-wide enable signal

output            DCOut;            // DC status signal out
output            ACOut;            // AC status signal out
output            EOBOut;           // EOB status signal out
output            ZRLOut;           // ZRL status signal out
output            EOB2out;          // BlkEnd status signal out
output [10:0]     CoefficientOut;   // Coefficient output
output [3:0]      RunLengthOut;     // Run length output
output [3:0]      Category;         // Evaluated category

reg               DCOut;
reg               ACOut;
reg               EOBOut;
reg               ZRLOut;
reg               EOB2out;
reg [10:0]        CoefficientOut;
reg [3:0]         RunLengthOut;
reg [3:0]         Category;



always @(posedge Clock or negedge Reset)
      begin
      if(!Reset)
            begin
CoefficientOut <= 0;
```

```
            RunLengthOut <= 0;
DCOut <= 0;
ACOut <= 0;
            EOBOut <= 0;
            ZRLOut <= 0;
            EOB2out <= 0;
            end
else if(Enable)
            begin
            CoefficientOut <= Coefficient;
            RunLengthOut <= RunlengthIn;
            DCOut <= DCin;
            ACOut <= ACin;
            EOBOut <= EOBin;
            ZRLOut <= ZRLin;
            EOB2out <= EOB2in;

//The following if-else block models a 2:1 multiplexer.  The nested
      //if else statements within the if-else block models the
      priority   //encoders for generating the category from the
      coefficient.  The //first set of if-else statements model an
      active-low input encoder //for negative coefficients.  The
      second set of if-else statements //model an active high input
      encoder for positive coefficients.

            if (Coefficient [11] == 1'b1)
                  begin
                  if(Coefficient [10] == 1'b0)
                        Category <= 11;
                  else if(Coefficient [9] == 1'b0)
                        Category <= 10;
                  else if(Coefficient [8] == 1'b0)
                        Category <= 9;
                  else if(Coefficient [7] == 1'b0)
                        Category <= 8;
                  else if(Coefficient [6] == 1'b0)
                        Category <= 7;
                  else if(Coefficient [5] == 1'b0)
                        Category <= 6;
                  else if(Coefficient [4] == 1'b0)
                        Category <= 5;
                  else if(Coefficient [3] == 1'b0)
                        Category <= 4;
                  else if(Coefficient [2] == 1'b0)
                        Category <= 3;
                  else if(Coefficient [1] == 1'b0)
                        Category <= 2;
                  else if(Coefficient [0] == 1'b0)
                        Category <= 1;
                  else
                        Category <= 0;
                  end
            else
                  begin
                  if(Coefficient [10] == 1'b1)
                        Category <= 11;
                  else if(Coefficient [9] == 1'b1)
                        Category <= 10;
                  else if(Coefficient [8] == 1'b1)
                        Category <= 9;
                  else if(Coefficient [7] == 1'b1)
```

```
                        Category <= 8;
                    else if(Coefficient [6] == 1'b1)
                        Category <= 7;
                    else if(Coefficient [5] == 1'b1)
                        Category <= 6;
                    else if(Coefficient [4] == 1'b1)
                        Category <= 5;
                    else if(Coefficient [3] == 1'b1)
                        Category <= 4;
                    else if(Coefficient [2] == 1'b1)
                        Category <= 3;
                    else if(Coefficient [1] == 1'b1)
                        Category <= 2;
                    else if(Coefficient [0] == 1'b1)
                        Category <= 1;
                    else
                        Category <= 0;
                    end
                end
            end
        end
    endmodule
```

```
/*****************************************************************/
/*    This module implements the strip logic stage               */
/*    of the Entropy encoder                                     */
/*****************************************************************/

module Stripper(  Coefficient,
                  Runlength,
                  Category,
                  DC,
                  AC,
                  EOB,
                  ZRL,
                  EOB20,
                  Flush,
                  Clk,
                  Reset,
                  Enable,
                  CoefficientOut,
                  RunlengthOut,
                  CategoryOut,
                  DCOut,
                  ACOut,
                  EOBOut,
                  ZRLOut,
                  EOB2Out,
                  LoadControl
);

input [10:0]      Coefficient;      // Coefficient input
input [3:0]       Runlength;        // Runlength input
input [3:0]       Category;         // Category input
input             DC;               // 4-bit status field input
input             AC;
input             EOB;
input             ZRL;
input             EOB20;            // BlkEnd input
input             Clk;              // Clock signal input
input             Flush;            // Pipeline flush signal
input             Reset;            // low asserted system-wide reset
input             Enable;           // System-wide enable signal

output [10:0]     CoefficientOut;
output [3:0]      RunlengthOut;
output [3:0]      CategoryOut;
output            DCOut;
output            ACOut;
output            EOBOut;
output            ZRLOut;
output            EOB2Out;
output            LoadControl;      // Controls loading of data into
                                    // the strip logic stages

reg [10:0]        CoefficientOut;
reg [3:0]         RunlengthOut;
reg [3:0]         CategoryOut;
reg               DCOut;
reg               ACOut;
reg               EOBOut;
reg               ZRLOut;
reg               EOB2Out;
wire              LoadControl;
```

```
reg [10:0]        Coefficient1;     // The registers making up the
reg [10:0]        Coefficient2;     // the stages of the strip logic
reg [10:0]        Coefficient3;
reg [10:0]        Coefficient4;

reg [3:0]         Runlength1;
reg [3:0]         Runlength2;
reg [3:0]         Runlength3;
reg [3:0]         Runlength4;

reg [3:0]         Category1;
reg [3:0]         Category2;
reg [3:0]         Category3;
reg [3:0]         Category4;

reg               DC1;
reg               DC2;
reg               DC3;
reg               DC4;

reg               AC1;
reg               AC2;
reg               AC3;
reg               AC4;

reg               EOB1;
reg               EOB2;
reg               EOB3;
reg               EOB4;

reg               ZRL1;
reg               ZRL2;
reg               ZRL3;
reg               ZRL4;

reg               EOB21;
reg               EOB22;
reg               EOB23;
reg               EOB24;

// The LoadControl signal is generated if any of the 4 status signals
      // are received.  Also the external controller generates the
      Flush  // signal to empty the pipeline after the previous
      stages have
// stopped receiving new data.  Since the LoadControl is not
// generated for zero coefficients, the strip action is accomplished

or or1(LoadControl, DC, AC, EOB, ZRL, Flush);

always @(posedge Clk or negedge Reset)
begin
if(!Reset)
            begin
            DC1 <= 0;
            AC1 <= 0;
            ZRL1 <= 0;
            EOB1 <= 0;
            EOB21 <= 0;

            DC2 <= 0;
```

```
            AC2 <= 0;
            ZRL2 <= 0;
            EOB2 <= 0;
            EOB22 <= 0;

            DC3 <= 0;
            AC3 <= 0;
            ZRL3 <= 0;
            EOB3 <= 0;
            EOB23 <= 0;

            DC4 <= 0;
            AC4 <= 0;
            ZRL4 <= 0;
            EOB4 <= 0;
            EOB24 <= 0;

            DCOut <= 0;
            ACOut <= 0;
            ZRLOut <= 0;
            EOBOut <= 0;
            EOB2Out <= 0;
            end
      else if(LoadControl & Enable)
            begin
            DC1 <= DC;
            AC1 <= AC;
            EOB1 <= EOB;
            ZRL1 <= ZRL;
            EOB21 <= EOB20;

            DC2 <= DC1;
            AC2 <= AC1;
            EOB2 <= EOB1;
            ZRL2 <= ZRL1;
            EOB22 <= EOB21;

            DC3 <= DC2;
            AC3 <= AC2;
            EOB3 <= EOB2;
            ZRL3 <= ZRL2;
            EOB23 <= EOB22;

            DC4 <= DC3;
            AC4 <= AC3;
            EOB4 <= EOB3;
            ZRL4 <= ZRL3;
            EOB24 <= EOB23;

            DCOut <= DC4;
            ACOut <= AC4;
            EOBOut <= EOB4;
            EOB2Out <= EOB24;

// The signal ZRLout is conditionally output depending on the
// presense of EOB in the previous stages of strip logic

            if (ZRL4 & ZRL3 & ZRL2 & EOB1)
                  ZRLOut <= 0;
            else if(ZRL3 & ZRL4 & EOB2)
                  ZRLOut <= 0;
```

```
            else if(ZRL4 & EOB3)
                  ZRLOut <= 0;
            else
                  ZRLOut <= ZRL4;

            Coefficient1 <= Coefficient;
            Runlength1 <= Runlength;
            Category1 <= Category;

            Coefficient2 <= Coefficient1;
            Runlength2 <= Runlength1;
            Category2 <= Category1;

            Coefficient3 <= Coefficient2;
            Runlength3 <= Runlength2;
            Category3 <= Category2;

            Coefficient4 <= Coefficient3;
            Runlength4 <= Runlength3;
            Category4 <= Category3;

            CoefficientOut <= Coefficient4;
            RunlengthOut <= Runlength4;
            CategoryOut <= Category4;
            end
end
endmodule
```

```
/*****************************************************************/
/*    This module implements the Huffman encoder stage of the    */
/*    Entropy encoder                                            */
/*****************************************************************/


module HuffmanEncoder(  Clock,
Reset,
Enable,
                        ValidSignal,
CoefficientIn,
CategoryIn,
RunLengthIn,
DCin,
ACin,
EOBin,
EOB2in,
ZRLin,
Huffman,
CategoryOut,
                        RunLengthOut,
CoefficientOut,
EOB2out,
ValidOutput,
);

input             Clock;
input             Reset;
input             Enable;
input             ValidSignal;      // Signal indicating new data
input [10:0]      CoefficientIn;
input [3:0]       CategoryIn;
input [3:0]       RunLengthIn;
input             Dcin;
input             Acin;
input             EOBin;
input             ZRLin;
input             EOB2in;

output [3:0]      CategoryOut;
output [3:0]      RunLengthOut;
output [10:0]     CoefficientOut;
output [19:0]     Huffman;          // Huffman code and code length
output            EOB2out;          // BlkEnd signal
output            ValidOutput;      // Signal indicating valid code
                                    // output
reg               ValidOutput;
reg [19:0]        Huffman;

reg               ResetAll;         // Signals reset of all internal
                                    // registers
reg               LoadInput;        // Signals loading of new data
reg               LoadACOut;        // Signals loading of AC ROM out
reg               LoadDCOut;        // Signals loading of DC ROM out
reg               OutputValid;      // Sets the ValidOutput flag

reg [10:0]        Coefficient1;     // Registers making up the
reg [10:0]        Coefficient2;     // internal stages of the
reg [3:0]         RunLength1;       // Huffman coder stage
reg [3:0]         RunLength2;
reg [3:0]         Category1;
```

```
reg [3:0]         Category2;
reg               EOB21;
reg               EOB22;
reg               DC;
reg               AEZ;
wire              aez;              // Signals presense of either
                                    // AC, EOB or ZRL status signal
reg               CurrState;        // Current state register of FSM
reg               NextState;
wire [19:0]       ACCodeOut;        // Output of AC codes LUT
wire [19:0]       DCCodeOut;        // Output of DC codes LUT

parameter         RESET = 1'b0,
ST1=1'b1;

//    Instantiations of LUTs
AROMDC DCLut(Category1, DCCodeOut);
AROMAC ACLut({RunLength1,Category1}, ACCodeOut);

assign CoefficientOut = Coefficient2;
assign RunLengthOut = RunLength2;
assign CategoryOut = Category2;
assign EOB2out = EOB22;

assign aez = ACin | EOBin | ZRLin;

// Current state logic of FSM
always@(posedge Clock or posedge Reset)
  begin: FSM_SEQ
  if(!Reset)
    CurrState = RESET;
  else
    CurrState = NextState;
  end

// Next state and output logic of FSM
always@(CurrState or ValidSignal or Enable)
  begin: FSM_COMB
  ResetAll = 0;
  LoadInput = 0;
  LoadACOut = 0;
  LoadDCOut = 0;
  OutputValid = 0;

  case(CurrState)

    RESET:  begin
            ResetAll = 1;
            NextState = ST1;
            end

    ST1:    begin
            if(ValidSignal & Enable)
              begin
              LoadInput = 1;
              if(AEZ)
                begin
                LoadACOut = 1;
                OutputValid = 1;
                end
              else if(DC)
```

```
                begin
                LoadDCOut = 1;
                OutputValid = 1;
                end
              else
                begin
                OutputValid = 0;
                end
              NextState = ST1;
              end
            else
              NextState = ST1;
            end

    default: NextState = RESET;

  endcase
  end

always@(posedge Clock)
  begin
    if(ResetAll)
      begin
      Coefficient1 <= 0;
      RunLength1 <= 0;
      Category1 <= 0;
      Coefficient2 <= 0;
      RunLength2 <= 0;
      Category2 <= 0;
      EOB21 <= 0;
      EOB22 <= 0;
      DC <= 0;
      AEZ <= 0;
      end
    else if(LoadInput)
      begin
      Coefficient1 <= CoefficientIn;
      RunLength1 <= RunLengthIn;
      Category1 <= CategoryIn;
      DC <= DCin;
      AEZ <= aez;
      Coefficient2 <= Coefficient1;
      RunLength2 <= RunLength1;
      Category2 <= Category1;
      EOB21 <= EOB2in;
      EOB22 <= EOB21;
      end
    else if(EOB22)
      EOB22 <= 0;
  end

always@(posedge Clock)
  begin
  if(LoadACOut)
    Huffman <= ACCodeOut;
  else if(LoadDCOut)
    Huffman <= DCCodeOut;
  else
    Huffman <= 0;
  end
```

```
always@(posedge Clock)
  if(OutputValid)
    ValidOutput <= 1;
  else
    ValidOutput <= 0;
endmodule
```

Appendix A

```
/*****************************************************************/
/*          This module implements Data packer Stage  of the       */
/*                       Entropy Encoder                           */
/*****************************************************************/

module packer(    Reset,
Enable,
CodeandLength,
Coefficient,
Category,
Clk,
ValidCode,
EOB,
EndD,
DataOut,
OutputValid,
Ready
 );


input [19:0]      CodeandLength;    //Huffman Code and Length
input [10:0]      Coefficient;      //Coefficient
input             Clk;              //System wide clock
input             ValidCode;        // Arrival of Valid Data
input             Reset;            //System wide reset
input             EOB;              //Signal for end of block
input             EndD;             //Signal for end of data
input             Enable;           //System wide enable
input [3:0]       Category;         //Category

output [15:0]     DataOut;          //Output Data
output            OutputValid,      //Signals external logic that
                                    // valid data is available
output            Ready;            //Data packer stage ready or not

reg [15:0]        DataOut;
reg               OutputValid;
reg               Ready;

reg [26:0]        RegA;
reg [47:0]        RegB;
reg [4:0]         Length;           //Length of valid data in RegA
reg [5:0]         EndP;             //Points to end of valid data in
                                    //RegB
reg               OutputLastData;   //Internal Signal
reg               ResetMe;          //Internal Reset
reg [15:0]        TempReg;          //Register for internal use
reg               EOB2, EOB3;

wire [10:0]       CoefficientIn, TempValue;
wire [3:0]        CodeLength;
wire              ResetIt;
wire              [15:0]Code;
wire              [15:0] Data;
wire [5:0]        Shift1, Shift2;

assign Code = CodeandLength[15:0];  //Take Code from CodeandLength
assign CodeLength = CodeandLength[19:16]; //Take Code Length
assign Data = RegB[47:32];          //Output is taken from 16 upper
                                    //bits of RegB
assign Shift1 = EndP - Length;      //Value of Shift for Shifting
```

```
                                    //contents of RegA in RegB

assign Shift2 = (Shift1 + 16 );     //Value of Shift for Shifting
                                    //contents of RegA in RegB
                                    //when output is valid

assign {CoefficientIn, TempValue}  = {Coefficient, 11'b 00000000000}
      & (22'b 0000000000011111111111 << Category);
                                    //Mask off extra bits of
                                    //Coefficient

assign ResetIt = Reset | ResetMe;
    //Reset = External Reset + Internal Reset



always @(posedge Clk)
begin
if(ResetIt)
begin
      EndP <= 48;
      OutputValid <= 0;


end
else if(Enable)
begin
      if(ValidCode)           //On arrival of valid code
      begin                   //take its length in Length
            Length <= Category + CodeLength + 1;
      end
      else
            Length <= 0;

      RegA <= CoefficientIn | (Code << Category);
                              //Latch Code and Coefficient as
                              //required in RegA

      if(EndP < 33)           //If there is valid data in upper
      begin                   //16 bits of RegB
            OutputValid <= 1; //Signal the output
            DataOut <= Data;  //Latch output
            RegB <= (RegB << 16) | ( RegA << Shift2 );
                              //Load new value in RegB
            EndP <= Shift2;   //Modify EndP to new end
      end
      else if(EOB3)           //If End of Block has arrived
      begin
            OutputValid <= 1; //Signal the external circuitry
            OutputLastData <= 1;    //Set internal flag
            {DataOut, TempReg} <= {RegB[47:32], 16'b
      0000000000000000} | (32'b 00000000000000001111111111111111 <<
      (EndP-16));             //Padding of 1s

      end
      else              //when there are less than 16 valid data bits
      begin             // in RegB
            RegB <= RegB | ( RegA << Shift1) ;
                  //Concatenate new value with the previous one
            EndP <= Shift1;   //Modify EndP to new value
            OutputValid <= 0; //Output is invalid
```

```
      end

 end
 end

always @(posedge Clk)
begin
if(ResetIt)                         //Reset the module
begin
      ResetMe <= 0;
      EOB2 <= 0;
      EOB3 <= 0;
end
else if(Enable)
begin
      if(EOB)
      begin
            EOB2 <= 1;
      end
      else if(EOB3)
      begin
            ResetMe <= 1;
      end
      else if(EOB2)
      begin
            EOB3 <= 1;
      end

      if(OutputLastData )
      begin                         //if last data has been output
            if(EndD)
                  Ready <= 1;       //tell this to external circuitry
            else
                  Ready <= 0;
      end

end
end



endmodule
```

```
/*****************************************************************/
/*          This module implements the logic to flush the entropy  */
/*          encoder pipeline after the last data has arrived       */
/*****************************************************************/

module FlushLogic(      Reset,
Clock,
Enable,
EndData,
ReadyIn,
Flush,
LastData,
ReadyOut
);

input             Reset;
input             Clock;
input             EndData;    // Signals the end of data arrival
input             Enable;
input             ReadyIn;    // Ready status from Data Packer

output            Flush;      // Signal used to flush the pipeline
output            LastData;   // Informs Data Packer about last data
output            ReadyOut;   // Informs the top controller of status


reg               Flush;
reg               LastData;
reg               ReadyOut;

integer           Index;      // Used to track flush count
reg               Over;       // Internal flag, set after EndData
      Pulse


always@(posedge Clock or posedge Reset)
      begin
      if(Reset)
            begin
            Flush <= 0;
            Over <= 0;
            LastData <= 0;
            ReadyOut <= 0;
            Index <= 0;
            end
      else if(Enable)
            begin
            if(EndData)       // if last data has arrived
                  Over <= 1;  // set the internal flag
            if(Over)          // if internal flag set, start counter
                  Index <= Index + 1;

            if(Index==2)      // time to set the flush signal
                  Flush <= 1;
            else if(Index==9) // time to lower the flush signal
                  Flush <= 0;
            else if(Index==14) // time to inform data packer

                  LastData <= 1;
            if(ReadyIn & Over) // if data packer has finished work
```

```
                begin
                ReadyOut <= 1;
                Over <= 0;
                Index <= 0;
                end
            else
                ReadyOut <= 0;
            end
        end
endmodule
```

Appendix A

Appendix B

```
/*****************************************************************/
/*This program performs DCT by the original 2D-DCT equation. Input */
/*matrix is read from a file                                       */
/*****************************************************************/
#include <stdio.h>
#include <math.h>
#include <conio.h>
#define PI 3.1415926

int Pixels[8][8];

void ReadData(int Pixels[8][8]);
void DoBiasing(void);
void FDCT(void);
void DisplayResults(void);
int DCT[8][8];

void main(void)
{
      clrscr();
      ReadData(Pixels);
      DoBiasing();
      FDCT();
      DisplayResults();

}

void ReadData(int Pixels[8][8])
{
      int i,j;
      char Number[10];
      FILE *fptr;

      if((fptr = fopen("c:\\DCTdata2.txt","r"))==NULL)
      {
            printf("\nThe file does not exist.");
            getch();
            exit(0);
      }

      for(i=0;i<8;i++)
      {
            for(j=0;j<8;j++)
            {
                  fgets(Number, 10, fptr);
                  Pixels[i][j] = atoi(Number);
            }
      }
      fclose(fptr);
}


void DoBiasing(void)
{
      int i,j;
      for(i=0;i<8;i++)
      {
            for(j=0;j<8;j++)
            {
                  Pixels[i][j] -=128;
            }
      }
}
```

```
void FDCT(void)
{
      int u,v,i,j;
      double temp;

      FILE *fptr;
      char *file;
      if((fptr=fopen("c:\\Debug.txt","w"))==NULL)
                  printf("Sorry, fatal error.  Contact Agent006\n");

      for(u=0;u<8;u++)
      {
            for(v=0;v<8;v++)
            {
                  temp = 0.0;
                  for(i=0;i<8;i++)
                  {
                        for(j=0;j<8;j++)
                        {
                        temp+=cos((2*i+1)*u*PI/16)*cos((2*j+1)*v*PI/1
                        6)*Pixels[i][j];
                        fprintf(fptr, "u=%d, v=%d, i=%d, j=%d,
                        temp=%f\n",u,v,i,j,temp);
                        }
                  }
                  if(u==0)
                        temp *= 0.707;
                  else
                        temp *=1;

                  if(v==0)
                        temp *=0.707;
                  else
                        temp *=1;

                  temp /=4;
                  temp += 0.5;
                  DCT[u][v] = (int)temp;
                  fprintf(fptr, "u=%d, v=%d, temp=%f\n",u,v,temp);
                  fprintf(fptr, "DCT = %d\n\n",DCT[u][v]);
            }
      }
      fclose(fptr);
}

void DisplayResults(void)
{
      int u,v,i,j;
      for(u=0;u<8;u++)
      {
            for(v=0;v<8;v++)
            {
                  printf(" %d ",DCT[u][v]);
            }
            printf("\n");
      }
}
```

Appendix B

```
/*****************************************************************/
/*This file performs 2D-DCT by row column decomposition method     */
/*****************************************************************/

#include <stdio.h>
#include <math.h>
#include <conio.h>
#define PI 3.1415926

int Pixels[8][8] = { {140, 144, 152, 168, 162, 147, 136, 148},
                     {144, 152, 155, 145, 148, 167, 156, 155},
                     {147, 140, 136, 156, 156, 140, 123, 136},
                     {140, 147, 167, 160, 148, 155, 167, 155},
                     {140, 140, 163, 152, 140, 155, 162, 152},
                     {155, 148, 162, 155, 136, 140, 144, 147},
                     {179, 167, 152, 136, 147, 136, 140, 147},
                     {175, 179, 172, 160, 162, 162, 147, 136} };


void makeDCT_Table(double DCTTable[8][8]);
void ShowTable(int Matrix[8][8]);
void ShowTableD(double Matrix[8][8]);
void Transpose(int Matrix[8][8]);
void CalculateResult(double DCTTable[8][8], int Pixels[8][8], int
Result[8][8]);
void Biasing(int Pixel[8][8]);

void main(void)
{
      int i,j;
      double DCTTable[8][8];
      int Result[8][8];
      int Pixels2[8][8];
      clrscr();
      makeDCT_Table(DCTTable);
      ShowTable(Pixels2);
      Transpose(Pixels);
      Biasing(Pixels2);
      CalculateResult(DCTTable, Pixels2, Result);
      getch();
}

void CalculateResult(double DCTTable[8][8], int Pixels[8][8], int
Result[8][8])
{
      int i,j,k;
      double sum;
      int TempResult[8][8];
      printf("\n");
      for(i=0;i<8;i++)
      {
            for(j=0;j<8;j++)
            {
                  sum = 0;
                  for(k=0;k<8;k++)
                  {
                        sum+=Pixels[i][k]*DCTTable[k][j];

                  }
                  //printf("%d ",(int)sum);
                  TempResult[i][j] = (int)sum;
            }
      }

      ShowTable(TempResult);
      Transpose(TempResult);

      for(i=0;i<8;i++)
      {
```

```
            for(j=0;j<8;j++)
            {
                  sum = 0;
                  for(k=0;k<8;k++)
                  {
                        sum+=TempResult[i][k]*DCTTable[k][j];

                  }
                  Result[i][j] = (int)sum;
            }
      }
      printf("\n");
      ShowTable(Result);
}

void Biasing(int Pixels2[8][8])
{
      int i,j;
      for(i=0;i<8;i++)
      {
            for(j=0;j<8;j++)
            {
                  Pixels2[i][j] -=128;
            }
      }

}

void makeDCT_Table(double DCTTable[8][8])
{
      int k,l;
      double x,y;

      //y = 1.0/4;
      for(k=1;k<=8;k++)
      {
            for(l=1;l<=8;l++)
            {
                  if(l==1)
                  {
                        DCTTable[k-1][l-1] = (double)sqrt(1/8.0);
                  }
                  else
                  {
                        x=(2*k-1)*(l-1)*PI/16;
                        x = cos(x)/2;
                        DCTTable[k-1][l-1] = x;
                  }
            }
      }
}

void ShowTable(int Matrix[8][8])
{
      int i,j;
      for(i=0;i<8;i++)
      {
            for(j=0;j<8;j++)
            {
                  printf("%d ",Matrix[i][j]);
                  //getch();
            }
            printf("\n");
      }
}

void ShowTableD(double Matrix[8][8])
{
      int i,j;
```

```
	for(i=0;i<8;i++)
	{
		for(j=0;j<8;j++)
		{
			printf("%f ",Matrix[i][j]);
			//getch();
		}
		printf("\n");
	}
}

void Transpose(int Matrix[8][8])
{
	int i,j;
	int Dummy[8][8];
	for(i=0;i<8;i++)
	{
		for(j=0;j<8;j++)
		{
			Dummy[i][j] = Matrix[j][i];
		}
	}

	for(i=0;i<8;i++)
	{
		for(j=0;j<8;j++)
		{
			Matrix[i][j] = Dummy[i][j];
		}
	}
}
```

```
/*****************************************************************/
/*This program implements our DCT architecture employing u and v   */
/* decomposition                                                   */
/*****************************************************************/

#include <stdio.h>
#include <math.h>
#include <conio.h>
#define PI 3.1415926

int Pixels[8][8]; = { {140, 144, 152, 168, 162, 147, 136, 148},
                      {144, 152, 155, 145, 148, 167, 156, 155},
                      {147, 140, 136, 156, 156, 140, 123, 136},
                      {140, 147, 167, 160, 148, 155, 167, 155},
                      {140, 140, 163, 152, 140, 155, 162, 152},
                      {155, 148, 162, 155, 136, 140, 144, 147},
                      {179, 167, 152, 136, 147, 136, 140, 147},
                      {175, 179, 172, 160, 162, 162, 147, 136} };

int u[8][4], v[8][4];

void makeDCT_Table(double DCTTable[8][8]);
void ShowTable(int Matrix[8][8]);
void Transpose(int Matrix[8][8]);
void Apply1D_DCT(double DCTTable[8][8], int u[8][4], int v[8][4], int
Result[8][8]);
void Biasing(void);
void MakeuvTables(int u[8][4], int v[8][4]);

void main(void)
{
      int i,j;
      double DCTTable[8][8];
      int Result[8][8];
      clrscr();
      ReadData(Pixels);
      ShowTable(Pixels);
      makeDCT_Table(DCTTable);
      Biasing();
      ShowTable(Pixels);
      Transpose(Pixels);
      ShowTable(Pixels);
      MakeuvTables(u,v);
      Apply1D_DCT(DCTTable, u, v, Result);
      ShowTable(Result);
      Transpose(Result);
      ShowTable(Result);
      for(i=0;i<8;i++)
            for(j=0;j<8;j++)
                  Pixels[i][j]=Result[i][j];

      MakeuvTables(u,v);
      Apply1D_DCT(DCTTable, u, v, Result);
      getch();
      ShowTable(Result);
      getch();
}



void Apply1D_DCT(double DCTTable[8][8], int u[8][4], int v[8][4], int
Result[8][8])
{
      int k,l,m;
      double sum;
      for(k=0;k<8;k++)
      {
            for(l=1;l<=8;l++)
```

```
            {
                  sum = 0;
                  if(l%2==0)
                  {
                        for(m=0;m<4;m++)
                        {
                              sum += v[k][m] * DCTTable[m][l-1];
                        }
                  }
                  else
                  {
                        for(m=0;m<4;m++)
                        {
                              sum += u[k][m] * DCTTable[m][l-1];
                        }
                  }
                  Result[k][l-1] = (int)sum;
            }
      }
}

void MakeuvTables(int u[8][4], int v[8][4])
{
      int k,m;
      for(k=0;k<8;k++)
      {
            for(m=1;m<=4;m++)
            {
                  u[k][m-1] = Pixels[k][m-1] + Pixels[k][7-m+1];
                  v[k][m-1] = Pixels[k][m-1] - Pixels[k][7-m+1];
            }
      }

}

void Biasing(void)
{
      int i,j;
      for(i=0;i<8;i++)
      {
            for(j=0;j<8;j++)
            {
                  Pixels[i][j] -=128;
            }
      }

}

void makeDCT_Table(double DCTTable[8][8])
{
      int k,l;
      double x,y;

      //y = 1.0/4;
      for(k=1;k<=8;k++)
      {
            for(l=1;l<=8;l++)
            {
                  if(l==1)
                  {
                        DCTTable[k-1][l-1] = (double)sqrt(1/8.0);
                  }
                  else
                  {
                        x=(2*k-1)*(l-1)*PI/16;
                        x = cos(x)/2;
                        DCTTable[k-1][l-1] = x;
                  }
            }
```

```
	}
}

void ShowTable(int Matrix[8][8])
{
	int i,j;

	FILE *fptr;
	printf("\n");
	if((fptr=fopen("d:\\Dct.txt","a"))==NULL)
		printf("Sorry, fatal error.  Contact Agent006\n");
	for(i=0;i<8;i++)
	{
		for(j=0;j<8;j++)
		{
			printf("%d ",Matrix[i][j]);
			fprintf(fptr, "%d ", Matrix[i][j]);
			//getch();
		}
		printf("\n");
		fprintf(fptr, "\n");
	}
	fprintf(fptr, "\n");
	fclose(fptr);
}

void Transpose(int Matrix[8][8])
{
	int i,j;
	int Dummy[8][8];
	for(i=0;i<8;i++)
	{
		for(j=0;j<8;j++)
		{
			Dummy[i][j] = Matrix[j][i];
		}
	}
	for(i=0;i<8;i++)
	{
		for(j=0;j<8;j++)
		{
			Matrix[i][j] = Dummy[i][j];
		}
	}
}
```

```
/*****************************************************************/
/*This program evaluates Rom contents for our 2D-DCT architecture  */
/*****************************************************************/

#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <string.h>
#define PI 3.1415926

void Evaluate_ROM_Contents(double DCTTable[8][8], int Result[16][8]);
void makeDCT_Table(double DCTTable[8][8]);
void Make_Table(int Result[16][8]);

void main(void)
{
      int i,j;
      double DCTTable[8][8];
      int Result[16][8];
      clrscr();
      makeDCT_Table(DCTTable);
      Evaluate_ROM_Contents(DCTTable, Result);
      Make_Table(Result);

}

void Evaluate_ROM_Contents(double DCTTable[8][8], int Result[16][8])
{
      int k,m,l,value;
      int bit;
      double SOP;
      for(l=0;l<8;l++)
      {
            for(value=0;value<=15;value++)
            {
                  Result[value][l]=0;
                  SOP = 0;
                  for(m=0;m<4;m++)
                  {
                        bit = (value>>m)&0x01;
                        SOP += bit*DCTTable[m][l];
                  }
                  Result[value][l] = SOP*256;
                        printf("value=%d, l=%d, Result=%d, SOP=%f\n",
                  value, l, Result[value][l], SOP);
            }
            printf("\n");
            getch();
      }
}

void Make_Table(int Result[16][8])
{
      int l, value, integer, index;
      FILE *fptr;
      char byte,string[31];
      char *file;
      clrscr();
      if((fptr=fopen("c:\\LUT.dat","w"))==NULL)
                  printf("Sorry, fatal error.  Contact Agent006\n");

      for(l=0;l<8;l++)
      {
            for(value=0;value<=15;value++)
            {
                  integer = Result[value][l];
                  for(index=0;index<11;index++)
                  {
                        byte = integer & 0x00000001;
```

```
                        if(byte)
                              string[index] = '1';
                        else
                              string[index] = '0';
                        integer = integer >> 1;
                  }
                  string[index]='\x0';
                  printf("%s\n",strrev(string));
                  fputs(string, fptr);
                  fputs("\n",fptr);
            }
            fputs("\n",fptr);
            getch();
      }
      fclose(fptr);
}

void makeDCT_Table(double DCTTable[8][8])
{
      int k,l;
      double x,y;

      for(k=1;k<=8;k++)
      {
            for(l=1;l<=8;l++)
            {
                  if(l==1)
                  {
                        DCTTable[k-1][l-1] = (double)sqrt(1/8.0);
                  }
                  else
                  {
                        x=(2*k-1)*(l-1)*PI/16;
                        x = cos(x)/2;
                        DCTTable[k-1][l-1] = x;
                  }
            }
      }
}
```

```
/*****************************************************************/
/*This is a simulation model for huffman coder                   */
/*****************************************************************/

#include "string.h"
#include "stdio.h"
#include "stdlib.h"
#include "conio.h"

void LoadLUT(char AC[][20], char DC[][20]);
void HuffmanCoder(int Data[64], char ACcodes[][20], char
DCcodes[][20]);
int GetCategory(int Data);

void main(void)
{
      char Number[10];
      FILE *fptr;
      char Filename[80];
      int Data[64];
      char AC[256][20], DC[12][20];

      printf("\nEnter Filename: ");
      gets(Filename);

      if((fptr = fopen("c:\\tc\\output\\data.txt","r"))==NULL)
      {
            printf("\nThe file does not exist.");
            getch();
            exit(0);
      }

      for(int i=0;i<64;i++)
      {
            fgets(Number, 10, fptr);
            Data[i] = atoi(Number);
            //printf("%d\n",Data[i]);
      }
      fclose(fptr);
      LoadLUT(AC, DC);
      HuffmanCoder(Data, AC, DC);
      getch();
}

void LoadLUT(char AC[][20], char DC[][20])
{
      FILE *fptr;
      int count;

      if((fptr=fopen("c:\\tc\\output\\AC_code.txt","r"))==NULL)
      {
            printf("\nUnable to open AC LUT file");
            getch();
            exit(0);
      }

      for(count=0;count<256;count++)
            fgets(AC[count], 20, fptr);
      fclose(fptr);

      if((fptr=fopen("c:\\tc\\output\\DC_code.txt","r"))==NULL)
      {
            printf("\nUnable to open DC LUT file");
            getch();
            exit(0);
      }

      for(count=0;count<12;count++)
            fgets(DC[count], 20, fptr);
```

```
	fclose(fptr);
}

void HuffmanCoder(int Data[64], char ACcodes[][20], char
DCcodes[][20])
{
	int Category;
	int ZC=0;
	int ZRL=0;
	int k;
	char Pattern[30];
	char Coeff[15];
	FILE *fptr;
	char *ptr;

	if((fptr=fopen("ECS.txt","w"))==NULL)
	{
		printf("\nCannot create Huffman results file.  Contact
Faisal Nawaz (agent006) at nawaz@writeme.com");
		getch();
	}

	if(Data[0] & 0x8000)
			Data[0] = Data[0] - 1;
	Category = GetCategory(Data[0]);
	strcpy(Pattern,DCcodes[Category]);
	ptr=strchr(Pattern, '\n');
	Pattern[ptr-Pattern] = '\x0';
	for(int j=0;j<Category;j++)
	{
		if(Data[0] & 0x0001)
			Coeff[j] = '1';
		else
			Coeff[j] = '0';

		Data[0] = Data[0] >> 1;
	}
	Coeff[j] = '\x0';
	strrev(Coeff);
	strcat(Pattern, Coeff);
	printf(" %s ",Pattern);
	fprintf(fptr," %s ",Pattern);
	for(k=1;k<64;k++)
	{
		if(!Data[k])
		{
			ZC = ZC + 1;
			if(ZC==16)
			{
				ZRL = ZRL + 1;
				ZC = 0;
			}
		}
		else
		{
			while(ZRL>0)
			{
				strcpy(Pattern, ACcodes[0xF0]);
				ZRL--;
				ptr=strchr(Pattern, '\n');
				Pattern[ptr-Pattern] = '\x0';
				printf(" %s ",Pattern);
				fprintf(fptr," %s ",Pattern);
			}
			if(Data[k] & 0x8000)
				Data[k] = Data[k] - 1;
			Category = GetCategory(Data[k]);
			int Index = (ZC<<4) + Category;
			strcpy(Pattern, ACcodes[Index]);
```

```
                        ptr=strchr(Pattern, '\n');
                        Pattern[ptr-Pattern] = '\x0';
                        for(int j=0;j<Category;j++)
                        {
                              if(Data[k] & 0x0001)
                                    Coeff[j] = '1';
                              else
                                    Coeff[j] = '0';
                              Data[k] = Data[k] >> 1;
                        }
                        Coeff[j] = '\x0';
                        strrev(Coeff);
                        strcat(Pattern, Coeff);
                        printf(" %s ",Pattern);
                        fprintf(fptr," %s ",Pattern);
                        ZC = 0;
                  }
            }
            if(!Data[63])
            {
                  strcpy(Pattern, ACcodes[0]);
                  ptr=strchr(Pattern, '\n');
                  Pattern[ptr-Pattern] = '\x0';
                  printf(" %s ",Pattern);
                  fprintf(fptr," %s ",Pattern);
            }
      }

      int GetCategory(int Data)
      {
            char c[20];
            int cat;

            itoa(Data, c, 2);
            strrev(c);

            if(c[11]=='1')
            {
                  if(c[10]=='0')
                        cat = 11;
                  else if(c[9]=='0')
                        cat = 10;
                  else if(c[8]=='0')
                        cat = 9;
                  else if(c[7]=='0')
                        cat = 8;
                  else if(c[6]=='0')
                        cat = 7;
                  else if(c[5]=='0')
                        cat = 6;
                  else if(c[4]=='0')
                        cat = 5;
                  else if(c[3]=='0')
                        cat = 4;
                  else if(c[2]=='0')
                        cat = 3;
                  else if(c[1]=='0')
                        cat = 2;
                  else if(c[0]=='0')
                        cat = 1;
                  else
                        cat = 0;
            }
            else
            {
                  if(c[10]=='1')
                        cat = 11;
                  else if(c[9]=='1')
                        cat = 10;
```

```
            else if(c[8]=='1')
                  cat = 9;
            else if(c[7]=='1')
                  cat = 8;
            else if(c[6]=='1')
                  cat = 7;
            else if(c[5]=='1')
                  cat = 6;
            else if(c[4]=='1')
                  cat = 5;
            else if(c[3]=='1')
                  cat = 4;
            else if(c[2]=='1')
                  cat = 3;
            else if(c[1]=='1')
                  cat = 2;
            else if(c[0]=='1')
                  cat = 1;
            else
                  cat = 0;
      }
      return(cat);
}
```

```
/*****************************************************************/
/*This file produces Rom contents for Quantizer stage            */
/*****************************************************************/

#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <string.h>
#define PI 3.1415926


int QuantizationTable[8][8] = {{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, 12, 37, 56, 68, 109, 103, 77},
                               {24, 33, 55, 64, 81, 104, 113, 92},
                               {49, 64, 78, 87, 103, 121, 120, 101},
                         {72, 92, 95, 98, 112, 100, 103, 99} };


void Evaluate_ROM_Contents(int Result[8][8]);
void Make_Table(int Result[8][8]);

void main(void)
{
      int i,j;
      int Result[8][8];
      clrscr();
      Evaluate_ROM_Contents(Result);
      Make_Table(Result);

}

void Evaluate_ROM_Contents(int Result[16][8])
{
      int i, j;
      double temp,fractional;
      for (i=0;i<8;i++)
      {
            for(j=0;j<8;j++)
            {
                  temp = (1.0/QuantizationTable[i][j])*256;
                  Result[i][j] = (int)temp;
                  fractional = temp - Result[i][j];
                  if(fractional >= 0.5)
                        Result[i][j]++;
                  //printf("%3f %d ", temp, Result[i][j]);
                  //getch();
            }
      }
}

void Make_Table(int Result[16][8])
{
      int l, m,integer, index;
      FILE *fptr;
      char byte,string[31];
      char *file;
      clrscr();
      if((fptr=fopen("d:\\Quantization.dat","w"))==NULL)
                  printf("Sorry, fatal error.  Contact Agent006\n");
      for(l=0;l<8;l++)
      {
            for(m=0;m<8;m++)
            {
                  integer = Result[l][m];
                  for(index=0;index<8;index++)
                  {
```

```
                        byte = integer & 0x00000001;
                        if(byte)
                              string[index] = '1';
                        else
                              string[index] = '0';
                        integer = integer >> 1;
                  }
                  string[index]='\x0';
                  printf("%s\n",strrev(string));
                  fputs(string, fptr);
                  fputs("\n",fptr);
            }
            fputs("\n",fptr);
            getch();
      }
      fclose(fptr);
}
```