EEL4783 Project 2
JPEG Huffman Decoder

1.0 Introduction

In the past few years, the use of images on computers has increased dramatically. Today electronic cameras are common, and everyone has received e-mail or viewed websites that contained images. Since transmitting bits still takes time, there is a need to compress the data in images to make them easier to store and transmit. There are a number of standards that have been created, and this project is going to look at a part of the JPEG standard. JPEG stands for Joint Photographic Experts Group, which defines a compression/decompression standard for still-life images. The JPEG standard is used in downloading graphics from the internet, in digital cameras, in medical imaging tools, and in many other interesting applications. Since the goal of this class is to teach you about VLSI design and not to make you JPEG experts, we are going to focus on a small portion of the algorithm-- a section called the Huffman decoder. Furthermore, a number of simplifications have been made to the standard to make your lives (and ours) easier. We will call our simplified version of JPEG, JPEG-lite. If you don't know much about compression don't panic. We will give you the information that you need to know, and having a real application should make it more fun to do the project. It also will give you some first hand experience on trying to figure out how to create datapaths, from stuff that does not initially look like a datapath.

The next section will give a brief overview of compression, so you get the basic idea of what is going on, and then we will focus on the Huffman decoder, the actual project.

1.1 JPEG Compression

Reducing the number of bits in an image is easy. What is hard is to reduce the number of bits, and not make the image look worse. The key to achieving the latter is to find some representation of the image where much of the information is redundant, and then not to send the redundant information.
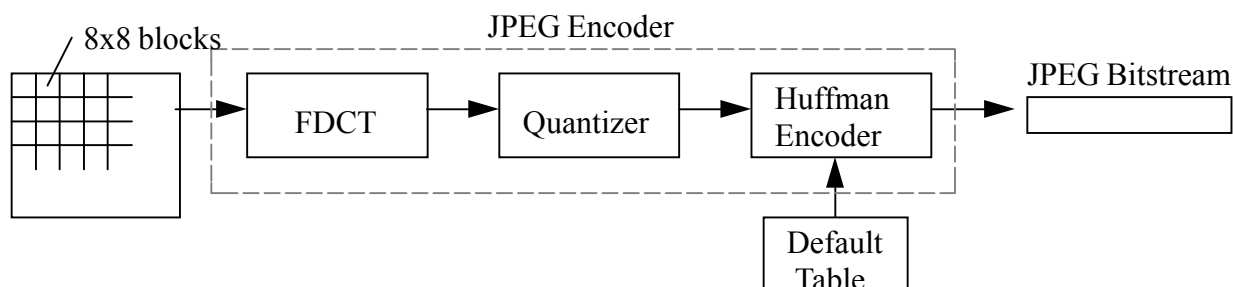

FIGURE 1. JPEG Encoder Flowchart

Figure 1 shows the processing steps needed to create a JPEG bitstream from a grayscale image. The image is divided into 8x8 blocks of 64 pixels where each pixel is represented by an 8-bit value. These blocks are sent to the forward discrete cosine transform (FDCT) which translates the pixel values of each block into spatial frequency coefficients, the rate at which the image changes from pixel to pixel (this is like a fourier transform). Images typically contain high frequency coefficients at the edges of objects and lower frequency components most everywhere else. Since the high-frequency components are generally small, often they can be approximated by zero with little error. The output of the FDCT transform are 64 values, which are arranged as another 8x8 block of values, with the DC coeficient (average value in the block) in the top-left corner and the highest-frequency coefficient at the bottom-right corner. Because edges are a small percentage of an image, most FDCT's contain a large fraction of image energy in the DC and low-frequency coefficients.

In the quantization processing step, each of the 64 coeficients in the FDCT block are divided by values in a table in order to increase the number of zero-value coefficients and to reduce the size of the JPEG bit-stream. Since the goal of quantization is to compress the image as much as possible, most of the loss in quality of the image occurs in this stage.

After this step, we have the values we want to transmit. The challenge is to transmit them using the least number of bits. One way of doing this is to use a Huffman code. The basis idea of Huffman codes is simple -- we don't need to transmit the number we want; instead, we can just transmit a code for that number. A table associating the number and its code is sent first to the decoder. Then, to send the number you want, you only need to send its code. This would not make much sense if the index was as large as the number, but usually you can make the index shorter. In fact, in Huffman codes, numbers that occur with the higher frequencies get assigned shorter codes. The less common numbers get assigned longer codes. Since all codes are not the same length, you also need to know how long your code is. The variable length is what makes Huffman codes tricky, as we will see in the project.

JPEG encoding compresses data in four ways:

- Because DC coefficients do not change significantly between adjacent blocks, they are encoded as differences. (Diff = DC_i - DC_{i-1}) This coding technique is known as Differential Pulse Code Modulation (DPCM).

- Quantized AC coefficients usually contain a run of consecutive zeroes. For this reason AC codes specify the run-length (number of consecutive zeroes preceding a non-zero coefficient) in addition to the amplitude of the coefficient.

- An end-of-block (EOB) code compresses data by indicating that the data in the rest of the scan are zeroes.

- Variable-length Huffman codes are selected such that shorter codes are used for frequently occurring run-length/coefficient sizes and longer codes are used for less-frequently occurring run-length/coefficient sizes. There is a unique Huffman code for each combination of run-length and coefficient size. There are separate tables for AC and DC Huffman codes because they exhibit different characteristics. (These terms will be defined later, so don't worry about these now.)
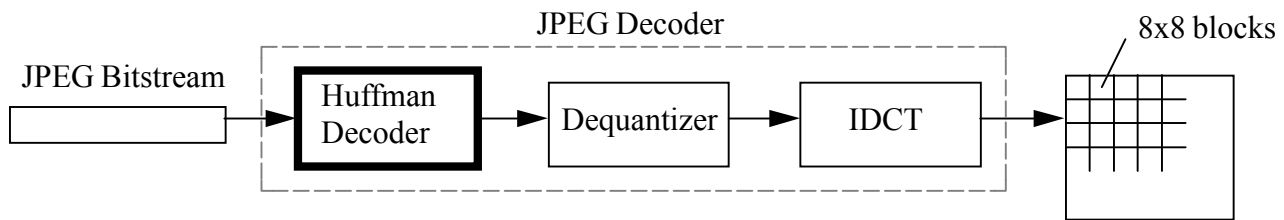
FIGURE 2. JPEG Decoder Flowchart

Figure 2 shows a JPEG decode flowchart. The JPEG bitstream contains the image size, dequantization tables, Huffman AC tables, Huffman DC tables, Huffman encoded data, and other information needed to decode the image. Huffman encoded data is decoded, multiplied by the values in the dequantization table, and then translated to pixel values by the inverse discrete cosine transform (IDCT). The 8x8 pixel blocks are then placed together to create the decompressed image.

We will look at making the Huffman decoder for the project. JPEG-lite simplifies JPEG by doing the following:

- The input has been modified to 4x4 blocks as opposed to the 8x8 blocks used in the JPEG standard in order to reduce the layout effort of hardware elements. (we sacrifice our compression ratio by a factor of 2)

- A simplified Huffman table will be used by the Encoder and Decoder that contains 10-bit Huffman codes and allows a maximum run-length of 3. The JPEG-baseline standard contains 16-bit Huffman codes and supports a maximum run-length of 15.

- Only a single AC and a single DC Huffman table will be used. In the JPEG baseline standard, two AC and DC tables were supported.

- 1-bit Huffman codes are not allowed in JPEG-lite.

- Only grayscale images will be decoded in JPEG-lite.

2.0 Project Overview

The final project will be dealing with the Huffman decoder portion of JPEG-lite. As mentioned above in Section 1.0, JPEG-lite takes an image and breaks it into 4x4 blocks of 16 pixels each, and after a number of processing steps, places the transformed values bit-by-bit on a bitstream in the form of Huffman codes. Your job will be to decode the bitstream and recreate the sequence of 4x4 blocks (and thus the entire image) by filling in missing portions of the verilog code which will be provided to you. The output of the decoder will be taken by post-processing software which will then display the decoded (mystery) image.

2.1 What's in the Bitstream?

The JPEG-lite bitstream contains many different types of information (See Appendix A). The portion of the bitstream that we will be focusing on is the part that encodes the pixel values of the image. The bit-

stream corresponding to this portion is a sequence of Huffman code/coefficient pairs. The following sections will explain what the Huffman code and the coefficients are, and how they are to be decoded.

2.2 Huffman Code Example

Huffman codes are variable-length codes that compress data by representing the more-frequently occurring data with shorter codes and less-frequently occurring data with longer codes. The following contrived example will illustrate this concept.

Consider the following set of numbers and their frequencies:

Frequency	Number	Huffman code
45	1000	00
20	100	01
10	10	100
5	5	1010
1	1	1011

As you can see in the above example, if we were to transmit the actual numbers, it would take a lot of bits because the ones occurring most frequently have larger values. Instead, Huffman codes are assigned to the numbers in the order of their frequencies, and the Huffman codes are transmitted in place of the number. In order to communicate the mapping between the numbers and the codes, the table containing this information is first sent to the decoder so that it can start decoding the Huffman codes that follow.

2.3 What's are JPEG-lite Huffman Codes?

Whereas in our previous example, the Huffman codes were associated with the actual numbers, in JPEG, the Huffman codes are associated with the sizes of the numbers and their run-lengths (to be discussed later). If each number were to have a unique Huffman code, we would end up with lots of Huffman codes, which would lead to longer code lengths. Furthermore, the events that occur most frequently in the image bitstream are the sizes of the numbers, not the actual numbers. Also, since the transformed values of the image contain numerous runs of consecutive zero values (referred to as run-length), not having to pass these values improves the compression rate. Thus, JPEG's Huffman code tells you two pieces of information: 1) the size of the number (coefficient) that immediately follows the Huffman code in the bitstream, and 2) the run-length value. The mapping between the Huffman codes and their coefficient size and run-length are stored in tables that are used by both the encoder and the decoder.

15	0	2	1
1	1	0	0
0	4	0	0
0	0	5	0

▢ DC Term

▢ AC Terms

Definitions:
Coefficient: Pixel value after having been transformed by the JPEG
algorithm
DC term: The first coefficient in the upper-left corner.
AC terms: The remaining coefficients in the 4x4 block.
Coeff_size: The number of binary bits needed to represent the
coefficient. (0-10 bits)
Run-length: The number of zeros preceding a non-zero coefficient.
(Range of 0-3 zeroes allowed)
EOB: End-of-block. If the remaining coefficients are all zero, a
special Huffman code indicates that the end of the 4x4 block
has been reached.

FIGURE 3. Example 4x4 block and Definitions

There are two sets of Huffman codes, DC codes and AC codes. DC Huffman codes are used to represent the first coefficient (DC term in Figure 3) in the 4x4 block, and AC Huffman codes are used to represent the remaining coefficients. Coefficients are the transformed values of the pixels in the 4x4 block. Only the non-zero coefficients are explicitly passed in the bitstream which improves the compression ratio.

What happens to the AC coefficients equal to zero? The notion of run-length is used to take care of these coefficients. Run-length is defined as the number of zero coefficients preceding a non-zero coefficient. The zero coefficients are counted in the order the block is processed: left to right, top to bottom. For example, in Figure 3 the run-length of the coefficient=2 is 1, the run-length of the coefficient=4 is 3, and the run-lengths of the coefficients=1 are zero. What is the run-length of the coefficient=5? Looking back at the definition of run-length we can see that the run-length can be from 0 to 3. Any sequence of four zeroes is represented by a coefficient size of zero (implying a zero coefficient) and a run-length of three. Since the 4 zeroes have now been processed, the run-length of the coefficient=5 is zero.

It may be observed that DC codes will always have a run-length of zero since they are the first code in the 4x4 block. In addition, DC coefficients are differentially encoded; in other words, the difference between the previous DC term and the current DC term is encoded, not the actual value of the current DC term. For example, if the DC coefficient in the previous 4x4 block was 12 and the current DC coefficient is 15, the encoded value on the bitstream is 3 (15-12).

2.4 Encoder Example

Before discussing the details of the Huffman decoder it is helpful to show an example of a 4x4 block encoded on a bitstream.

15	0	2	1
1	1	0	0
0	4	0	0
0	0	5	0

DC Coefficient of previous block=12

Run-length	Coeff_size	Huffman Code	Coefficient
0	2	011	11 (3=15-12)
1	2	11011	10 (2)
0	1	00	1 (1)
0	1	00	1 (1)
0	1	00	1 (1)
3	3	111110101	100 (4)
3	0	111001	none
0	3	100	101 (5)
0	0	1010	none - EOB

Bitstream = 0111111011100010010011111101011001110011001011010

FIGURE 4. Encoder example

In Figure 4, the encoded bitstream is shown for the given 4x4 block. In following this example, use the Huffman Code Tables in Appendix A. Table 1 will be explained as we go through the decoder example so ignore it for the moment. To determine the Huffman code for a DC coefficient of run-length=0 and coeff_size=2 we look at Table 2 (DC). The DC code that corresponds to a coeff_size of 2 is 011. This Huffman code specifies that a 2-bit coefficient will follow on the bitstream. After encoding the DC coefficient the bitstream contains 01111. To encode the AC coefficients we follow a similar process. Special attention should be given to the AC Huffman codes 111001 (4 zeroes in a row) and 1010 (EOB). The end-of-block (EOB) code indicates that the remaining coefficients in the 4x4 block are zeroes. In both cases, the coeff_size is zero so a coefficient is not placed on the bitstream after the Huffman code.

3.0 Huffman Decoder Project

(This is the important part, since this is the stuff you will need to build)

The hard part of Huffman coded data is decoding it. The encoded bitstream in Section 2.4 appears to be a random string of 0's and 1's. The difficulty in decoding the bitstream arises in determining how long a Huffman code is (remember that Huffman codes are variable-length). Fortunately, the JPEG standard provides a clever algorithm that takes advantage of the way in which Huffman codes are created. All Huffman codes must adhere to the following rule:

> The most significant (n-1) bits of the smallest Hufman code of length n are greater
> in value than the largest Huffman code of length (n-1).

For example, the smallest AC Huffman code of length 5 is 11010 and the greatest AC Huffman code of length 4 is 1100. It is easily seen that the most significant 4 (n-1) bits of the Huffman code of length 5 (1101) are greater than the greatest Huffman code of length 4 (1100). If the smallest code of length 5

were 11000, it would be ambiguous because it cannot be distinguished from the Huffman code 1100 of length 4.

In other words, you can always tell how long a code word should be by assuming its current length is its final length. You make this guess, and look up in a table (lookup table 1) for what the max code word is of this length. If the value you have is less than the max value in the table, you are correct, and you have a valid code. If not (if the code word you have so far is larger than the max value for this length) the code word is at least one bit longer, and you have to shift in another bit and try again. The algorithm in Figure 5 does exactly this, but it does one more thing as well. Once we know the length, we still need to lookup the actual value (the variable length data is really only an index). The actual value that we look up in lookup table 2 is the coefficient size and the runlength.

The index to the lookup table 2 is calculated by adding an offset (base) to the code word that has been identified above.

Huffman Code Identification Algorithm
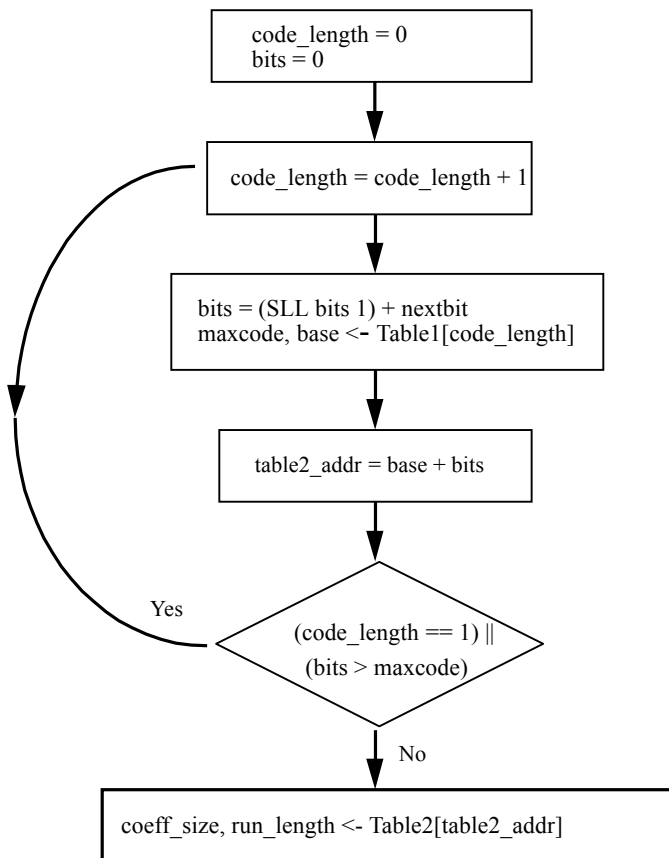
Example:

DC Huffman code = 011

code_length	1	2	3
bits	000000000	000000001	000000**011**
maxcode	undefined	000000000	000000110
base	undefined	000000	111111
table2_addr	undefined	000001	000010
loop condition?	YES	YES	NO
coeff_size	N/A	N/A	2
run_length	N/A	N/A	0

Note that the maxcode and base are undefined when the code_length == 1.

FIGURE 5. Huffman Code Identification

The above flowchart provides an overall picture of how the Huffman codes are recognized and how the coefficient size and the run-length corresponding to the Huffman code are retrieved.

In the initial state, code_length and bits are initialized to zero. (bits is a shift register used to receive the JPEG bitstream, and code_length counts the number of bits that have been shifted into bits.)

The middle 4 steps comprise a loop. First, the code_length counter is incremented. Then, the next bit is shifted into bits, and maxcode and base are looked up in Table 1. Maxcode is the maximum Huffman code for a given Huffman code_length and base is used to speculatively compute the index (table2_addr) for Table 2 (see Appendix A). Finally, a check is made to see whether bits > maxcode or code_length is one (there is no Huffman code of length one). If so, we know that bits does not match any Huffman code of length code_length so we need to keep iterating through the loop.

If bits is less than or equal to maxcode, a valid Huffman code has been found and a lookup is performed on Table 2 using table2_addr to retrieve the coeff_size and run_length for this Huffman code.

The example above shows how a Huffman code of 011 is recognized. The bits shifted in from the bitstream are displayed in bold font. The values for each iteration through the loop is shown until we find a valid Huffman code and its corresponding coeff_size and run_length.

4.0 The Big Picture

After all this talk about definitions and algorithms you are probably wondering how all these pieces fit together. The top-level diagram of the Huffman decoder is shown below in Figure 6. The inputs to the Huffman Decoder are the bitstream, initialization signals, and a reset signal. The initialization signals are used to initialize the lookup tables (maxcode,base, run_length, and coeff_size).

The output of the Huffman Decoder is the block number, the coefficient, the position within the block and a signal specifying that the coefficient and position are valid. The coefficient has already been defined as the pixel value transformed by the JPEG-lite algorithm and the position is a number from 0-15 which specifies the location within the 4x4 block as shown below. The output is used by the post-processing software that transforms the coefficients back into their original pixel values so that they can be viewed as a digital image.
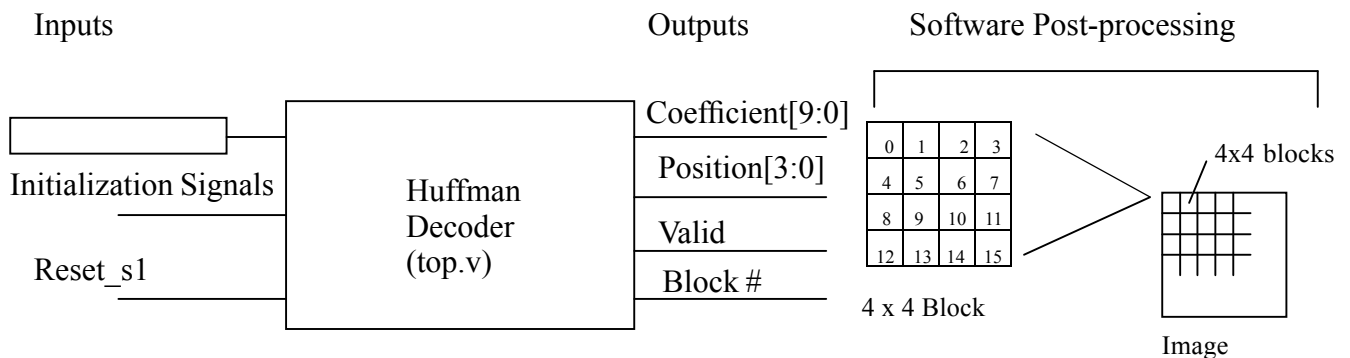

FIGURE 6. The Big Picture

4.1 Verilog Hierarchy

The Huffman decoder's top-level verilog, top.v, contains 4 sub-units: stimulus.v, datapath.v, control.v, and writeoutput.v. Stimulus.v and Writeouput.v are complete and do not need to be modified. You need to implement the Huffman Code Identification Algorithm in datapath.v and the finite-state machine in control.v. The interactions between these units are shown below in Figure 7.
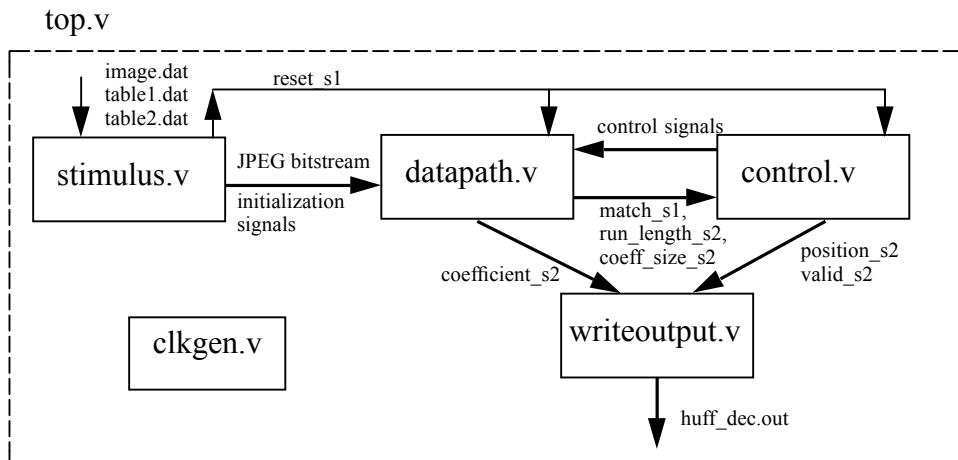

FIGURE 7. Verilog Hierarchy

Stimulus.v provides the input signals to the Huffman decoder. It receives its inputs from three files: image.dat, table1.dat and table2.dat. Table1.dat and table2.dat contain the necessary information to initialize the lookup tables used in the datapath. After initialization, stimulus.v sends bits from image.dat to datapath.v as the JPEG bitstream.

Datapath.v contains the lookup tables, the shift register and other hardware elements needed to implement the Huffman Code Identification Algorithm described in Section 3.0. The primary inputs are the JPEG bitstream from stimulus.v, and control signals from control.v that reset and initialize the shift register and code_length counter. These control signals will be discussed in more detail in Section 5.2. The primary outputs of datapath.v are match_s1, run_length_s2, coeff_size_s2, and coefficient_s2. Match_s1=1 indicates that a complete Huffman code has been shifted in from the JPEG bitstream. The results of the table2 lookup: run_length_s2 and coeff_size_s2, provide information to the control about the run_length and size of the coefficient that follow the Huffman code in the JPEG bitstream. Coefficient_s2 contains the bits that are currently in the datapath shift register. The block diagram of the datapath is contained in Section 6.1

Control.v contains a finite state machine, logic that counts how many coefficient bits have been shifted into the datapath shift register, and logic that calculates position_s2. The FSM will be described in more detail in Section 5.1. The primary inputs are match_s1, run_length_s2, and coeff_size_s2. The control block's primary outputs are position_s2, valid_s2 and control signals needed by the datapath (see Section 5.2). As described above, position_s2 indicates the location within the 4x4 block and valid_s2 is asserted when the entire coefficient has been shifted in from the bitstream.

Writeoutput.v takes coefficient_s2, position_s2, and writes them to the file, huff_dec.out, when valid_s2 is high. This is the file that is used by the post-processing software to transform the coeffcients into pixel values. The output format of the file is as follows: <4x4 block number><coefficient><position>. This file may be useful as you're debugging your verilog.

5.0 Huffman Decoder Control

5.1 Finite State Machine

The FSM (finite-state machine) keeps track of which part of the 4x4 block is being decoded and sets the control signals accordingly. The six states shown in Figure 8 are idle, init, dc_decode, dc_coeff, ac_decode, and ac_coeff. You will need to implement this FSM in control.v.
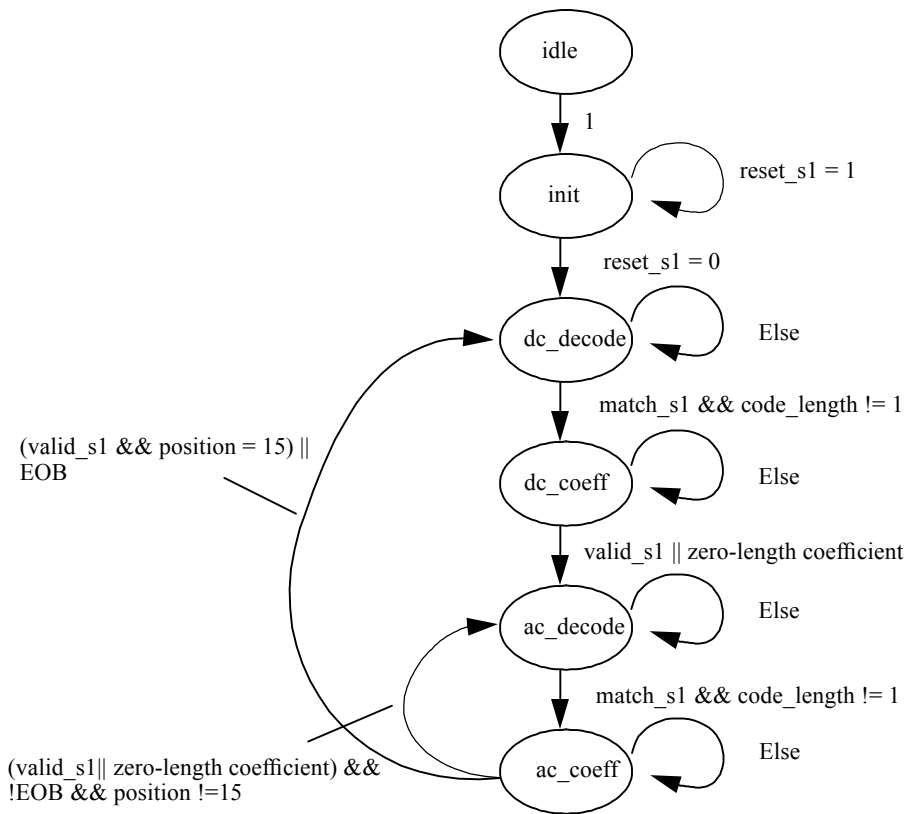

FIGURE 8. Finite State Machine

The decoder remains in the init state until reset_s1 goes low. During this state the lookup tables are initialized. After reset_s1 goes low, the next state is dc_decode and the DC Huffman code of the first entry in the 4x4 block is decoded.

The FSM remains in the dc_decode state until match_s1 goes high and code_length !=1. Match_s1 goes high when the Huffman code length has been determined. (match_s1 = bits <= maxcode) Remember that match_s1 is not valid when code_length = 1 because maxcode(code_length ==1) is undefined. <u>Hint:</u>

reset_sr_s1 is only high when code_length = 1 and may be used to qualify match_s1. (reset_sr_s1 is further described in Section 5.2)

Once the Huffman code length is been determined, the next state is dc_coeff. In this state the DC coefficient is read into the datapath shift register. If the coefficient has been read from the bitstream (valid_s1 = 1) or if the coefficient is zero-length then the next state is set to ac_decode. Hint: coeff_size_s2 and run_length_s2 are only valid one cycle after match_s1 goes high. (see Section 3.0)

The ac_decode state is very similar to dc_decode. Once the Hufman code_length is been determined the next state is set to ac_coeff.

The ac_coeff state is where things get a bit tricky. If the AC coefficient is zero or the coefficient is read from the bitstream then the next state may be either dc_decode or ac_decode. If the end of the 4x4 block is reached (position = 15) or the remaining AC coefficients are zero (EOB) then the next state is dc_decode and a new 4x4 block is started. If the coeficient is read from the bitstream and the end of the 4x4 block is not reached (position < 15) then the next state is ac_decode.

5.2 Control Signal Description

Understanding the functionality of the following control signals should help you design the datapath and debug your verilog.

Output

reset_sr_s2: This signal resets the bitstream shift register and the code_length shift register in the datapath before a new Huffman code or coefficient is read from the bitstream. The reset_sr_s2 signal is not asserted when the coeff_size is zero or the EOB code is placed on the bitstream.

dc_ac_s1: This signal selects whether the AC or DC Table 1 values are selected. The ac table is selected when dc_ac_s1=1 and the dc table is selected when dc_ac_s1=0.

rw1_en_s1: This signal is zero when table1 is read and 1 when table 1 is written to.

rw2_en_s1: This signal is zero when table2 is read and 1 when table 2 is written to.

valid_s2: This signal indicates that the outputs, coefficient_s2 and position_s2, are valid.

Input

match_s1: This signal is 1 when the Huffman code-length is determined (bits <= maxcode) and it is sampled by the control during the ac_decode or dc_decode state when code_length is greater than 1.

coeff_size_s2: This signal describes the coefficient length (in number of bits) and is only valid for one cycle after the complete Huffman code has been read from the bitstream

run_length_s2: This signal describes the run_length of a non-zero coefficient and is only valid for one cycle after the complete Huffman code has been read from the bitstream.

6.0 Detailed Datapath Information

6.1 Datapath Block Diagram
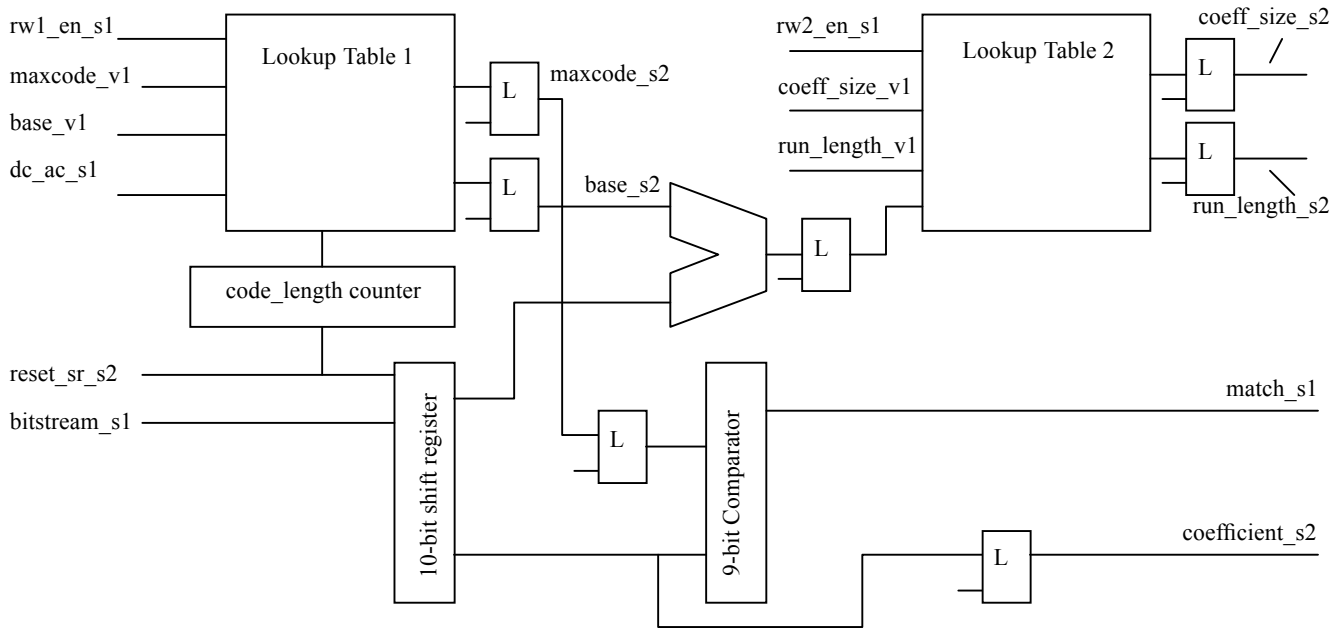
6.2 Timing Information

Your implementation of the datapath's Huffman Code Identification Algorithm must adhere to the timing shown below:
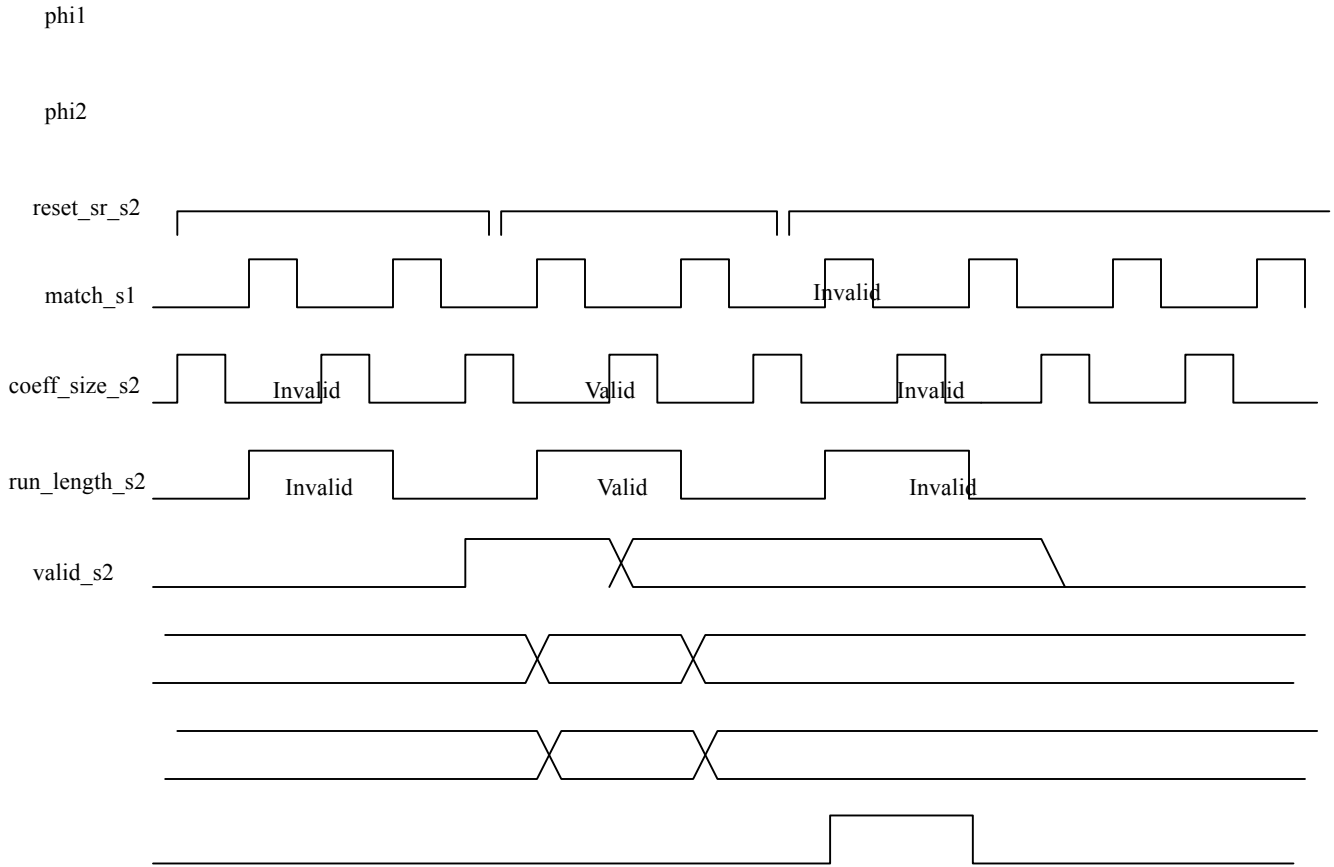
phi1

phi2

reset_sr_s2

match_s1   Invalid

coeff_size_s2   Invalid   Valid   Invalid

run_length_s2   Invalid   Valid   Invalid

valid_s2

7.0 What do I need to do?

1. Download all verilog code and documents to your own computer.

2. Fill in the datapath.v verilog after studying the datapath algorithm and timing section.

3. Code the FSM in control.v after studying the FSM section.

4. Try the test cases: test1.dat or create your own. Each test case contains two 4x4 blocks. Your output file, huff_dec.out, should match test1.out for test1.dat. See the newsgroup or web page on instructions on how to create your own test case.

   • cp test1.dat to image.dat.

   • Change the variable, NUM_BITS, as shown in stimulus.v:

   • Run your favorite verilog simulator.

   • Compare huff_dec.out to test1.out for test1.dat.

-

8.0 References

Pennebaker, William B, Joan L. Mitchell, "JPEG Still Image Data Compression Standard"
Wallace, Gregory K.   "The JPEG Still Picture Compression Standard"
Hung, Andy C. PVRG-JPEG CODEC 1.1

Appendix A

Physical Addr	Code Length	DC Maxcode	DC Base	AC Maxcode	AC Base
0	2	000000000	000000	000000001	001011
1	3	000000110	111111	000000100	001001
2	4	000001110	111000	000001100	000100
3	5	000011110	011000	000011011	110111
4	6	000111110	001010	000111010	011011
5	7	001111110	001011	001110111	100000
6	8	011111110	001100	000000000	000000
7	9	xxxxxxxxx	xxxxxx	111111100	111000

Physical Addr	Huffman Code	Run-length	Coeff Size (Hex)
0	00	0	0
1	010	0	1
2	011	0	2
3	100	0	3
4	101	0	4
5	110	0	5
6	1110	0	6
7	11110	0	7
8	111110	0	8
9	1111110	0	9
10	11111110	0	A

Physical Addr	Huffman Code	Run-length	Coeff Size (Hex)
11	00	0	1
12	01	0	2
13	100	0	3
14	1010 (EOB)	0	0
15	1011	0	4
16	1100	1	1
17	11010	0	5
18	11011	1	2
19	111000	2	1
20	111001	3	0
21	111010	3	1
22	1110110	0	6
23	1110111	1	3
24	111100000	0	7
25	111100001	0	8
26	111100010	0	9
27	111100011	0	A
28	111100100	1	4
29	111100101	1	5
30	111100110	1	6
31	111100111	1	7
32	111101000	1	8
33	111101001	1	9
34	111101010	1	A
35	111101011	2	2
36	111101100	2	3
37	111101101	2	4
38	111101110	2	5
39	111101111	2	6
40	111110000	2	7
41	111110001	2	8
42	111110010	2	9
43	111110011	2	A
44	111110100	3	2
45	111110101	3	3
46	111110110	3	4
47	111110111	3	5
48	111111000	3	6
49	111111001	3	7
50	111111010	3	8
51	111111011	3	9
52	111111100	3	A
53	111111101	unused	unused
54	111111110	unused	unused