Data Encoding Module on AHB by Huffman Algorithm

Final Report

Submission Date: 4/02/2017

Thursday 11:30am Lab (Lab Section 4)

TA: Chuan Yean Tan

Prepared by:

Yuqin Duan, Alan Han

1. Executive Summary

Nowadays, lossless compression plays an essential role in data compression and data reconstruction in graphic, video and cryptography. Among a lot of different algorithms, Huffman coding is one of the more commonly used algorithms in data compression and decompression. It is a particular type of optimal prefix code which operates fast and simple. The Huffman algorithm is useful as an ASIC because it requires high speed and accuracy of performance. Additionally, its frequent usage could benefit greatly from the low power requirements of ASIC.

To compress a file with Huffman coding, we must first find the frequency of the different characters our message is made up of. All this will happen in a computation core. Based on these frequencies, a tree of these characters is created, in descending order (the root of the tree represents the most frequent character). This tree is known as a Huffman tree. Accessing each leaves closer to the root takes less information than accessing leaves further from the root; it is from this rule that we compress our message.

Using the Avalon Bus Interface for inter-system communication, the communication bus can send this encoded message to another device. Additionally, the Avalon Bus Interface has settings that allow you to optimize interfacing timing, which can improve speed of operation and minimize problems from noise. Other settings of the Interface allow the user to customize the Huffman Encoder for the rest of their system.

Successful design of a human encoding module will require the following resource:

- Avalon SoC bus standard documentation
- Systemverilog simulation and Design Synthesis Tool Chain
- Huffman Encoding Algorithm documentation

The following document contents will describe:

- Intended main implementation architecture. (Session 2)
- The details of how avalon bus work collaboratively with huffman tree

module

2. Design Specifications

2.1. System Usage

2.1.1. System Usage Diagram

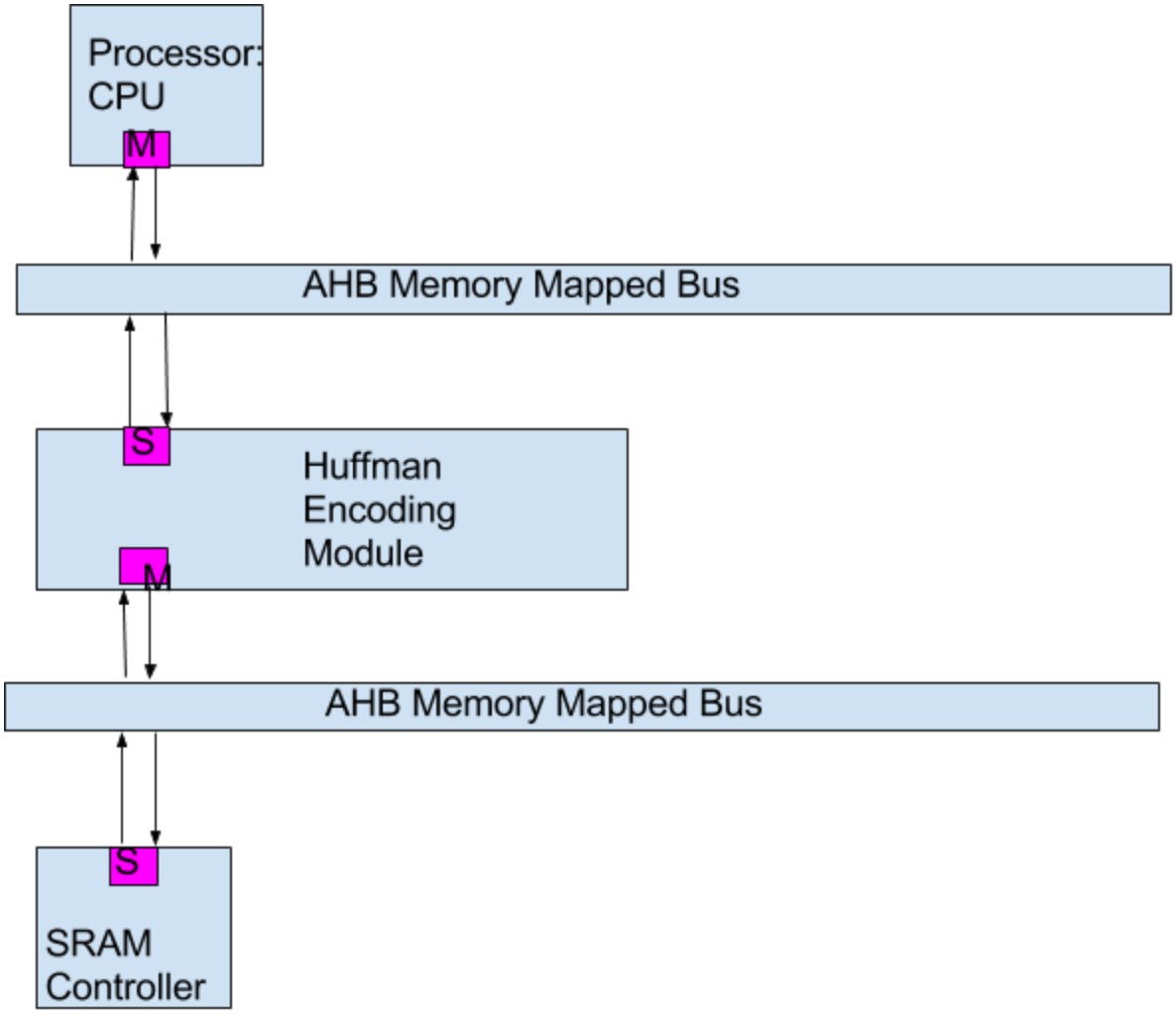


Figure 2.1.1: System Usage Diagram for Huffman Encoding Module

Figure 2.1.1 describes the intended use of Avalon Interface. In this system the main processor CPU is able to execute any software, so the CPU is the master which will

guide around all signals. SRAM Controller and our huffman encoding module will communicate between slaves. The key operation ideas are the following steps:

1. Avalon Memory Mapped Bus gets the character address from CPU, and then sends to Huffman Encoding Module
2. SRAM is active and will minimize the read-to-write delay.
3. If there is a time or resource constraint SRAM will be active
4. Cleared the previous sample data
5. Repeat step 1-4 multiply times until finish reading all data.

2.1.2. Implemented Standards and Algorithms

Huffman Tree:

- 32-bits Read:
    - Read characters in chunk (32 characters every time probably)
    - Default read sets to 0.
- 16-bits counter:
    - Extracted from 32-bits Read
    - Count the number of each ASCII characters in a data file.
    - Default count for each characters is 0.
- 32-bits Tree builder
    - Extracted from 32-bits counter
    - Find the maximum counter
    - Add the first max to the tree root
    - Otherwise add the current max to the tree leaf (based on the frequency of its root)
    - Remove current max. Move pointer the next max.
    - Default tree is null.
- 32-bits Output Tree
    - Extracted Tree from 32-bits Tree builder
    - Output the tree root first
    - Output the leaves from left to right, from top to down.

Avalon Memory Mapped Interface Standard Slave

- 32-bit data bus

- Read and Write Transfer
- SRAM Control

2.2. Pinout

2.2.1. Abridged Address Mapping

Slave Word Address	Read/Write	Data Size (B)	Description
0x000 -> 0x0FF	R/W	4	Input Sample Locations (Most Recent Sample to Oldest)
0x100 -> 0x1FF	R/W	2	Filter/Effect Value Locations (Value 0 -> Value 255)

Table 1: Huffman Encoder Local Address Mapping

2.2.2. AHB Bus Interface Pinout

Signal Name	Type (In/Out/Bidir)	Number of Bits	Description
vcc	PWR		Power pin
gnd	GND		Ground pin
clk	IN	1	Huffman Encoder clock signal (100 MHz) Note: Is also used as Avalon-MM bus clock.
n_rst	IN	1	Huffman Encoder Asynchronous reset signal. (Active Low)

Table 2a: Miscellaneous Pinout Table

Signal Name	Type (In/Out/Bidir)	Number of Bits	Description
address	IN	10	Slave Namespace 32-bit Word Address
byteenable	IN	4	Per bye enables for signaling which bytes of the transfer are active. (Active High)
burstcount	IN	9	Designates the burst size (Max of 256 values supported) (Minimum value of 1 required)
response	OUT	2	"00": OKAY -> Successful Transfer "01": RESERVED -> Not used by Huffman Encoder "10": SLAVEERROR -> Slave side error "11": DECODEERROR -> Bad Address

			Supplied
waitrequest	OUT	1	Asserted when slave cannot immediately respond to transfer request. (Active High)
read	IN	1	Asserted for Read Transfers (Active High)
readdata	OUT	32	32-bit data bus for read transfers
readdatavalid	OUT	1	Asserted when readdata bus has valid data.
write	IN	1	Asserted for Write Transfers (Active High)
writedata	IN	32	32-bit data bus for write transfers
writeresponsevalid	OUT	1	Asserted when a response to a write request is valid
debugaccess	IN	1	Ignored by design since Accelerator does not have internal ROMs

Table 3a: AHB Memory Mapped Slave Interface Pins

Signal Name	Type (In/Out/Bidir)	Number of Bits	Description
address	OUT	10	Slave Namespace 32-bit Word Address
byteenable	OUT	4	Per bye enables for signaling which bytes of the transfer are active. (Active High)
burstcount	OUT	9	Designates the burst size (Max of 256 values supported) (Minimum value of 1 required)
response	IN	2	"00": OKAY -> Successful Transfer "01": RESERVED -> Not used by Huffman Encoder "10": SLAVEERROR -> Slave side error "11": DECODEERROR -> Bad Address Supplied
waitrequest	IN	1	Asserted when slave cannot immediately respond to transfer request. (Active High)
read	OUT	1	Asserted for Read Transfers (Active High)

readdata	IN	32	32-bit data bus for read transfers
readdatavalid	IN	1	Asserted when readdata bus has valid data.
write	OUT	1	Asserted for Write Transfers (Active High)
writedata	OUT	32	32-bit data bus for write transfers
writeresponsevalid	IN	1	Asserted when a response to a write request is valid
debugaccess	OUT	1	Ignored by design since Accelerator does not have internal ROMs

Table 3b: AHB Memory Mapped Master Interface Pins

2.3. Operational Characteristics

2.3.1. Data Encoder Total Design Operation

Normal usage of the data encoder design:

1. Single Word Write(s) to set configuration registers (including compression scheme)
2. Single Word Write to clear prior message
3. Burst Write of message to be encoded
4. Single Word read of bytes encoded count
5. If bytes encoded count < bytes in message, then delay and go to step 4>
6. Burst Word Read of results
7. If not done with message processing, go to step 3

2.3.2. Huffman Tree Encoding Algorithm

The simplest tree construction algorithm uses a priority queue where the node with the highest probability(or frequency) is given highest priority. These data structures require O(log n) time per insertion. A tree with n leaves has 2n-1 nodes. This algorithm is designed to operate in O(n log n) time, where n is the number of ASCII characters. If the symbols are sorted by probability, there is a linear-time (O(n)) method to create a

Huffman tree using a big array, where the first 128 elements are the encoding digits, the second 128 elements are leaves' frequencies and the last 256 are nodes frequency and addresses of its own left and right nodes . Primitive operations are all assumed to be of O(1) time complexity. Provided that we have n items to create a tree from, the algorithm is as follows:

1. Begin with null.
2. Enqueue all the leaf nodes into the second 128 elements
3. As long as there is more than one node in the queues: ((n - 1) times)
    a. Go through the whole array and pick up two lowest frequencies
        I. (1 operation: compare frequencies)
    b. Create a new internal node, with the two just-removed nodes as children and the sum of their weights as the new weight.
        i. (3 operations: Create summed node, add 2 children)
    c. Clear the minimum frequencies nodes
        i. (1 operation: Clear)
4. The tree has now been generated and the remaining node is the root node.
    a. (Total operations: n + (n - 1)(2 + 3 + 1) = 7n - 6 operations)

For decompression, the process is simply a matter of translating the stream of prefix codes to individual byte values, by just traversing the Huffman tree node by node as each bit is read from the input stream. But before this can happen, the same Huffman tree used in Compression must be used for Decompression. So it would be a requirement for each compressed file or filestream to have a header with some of this critical information.

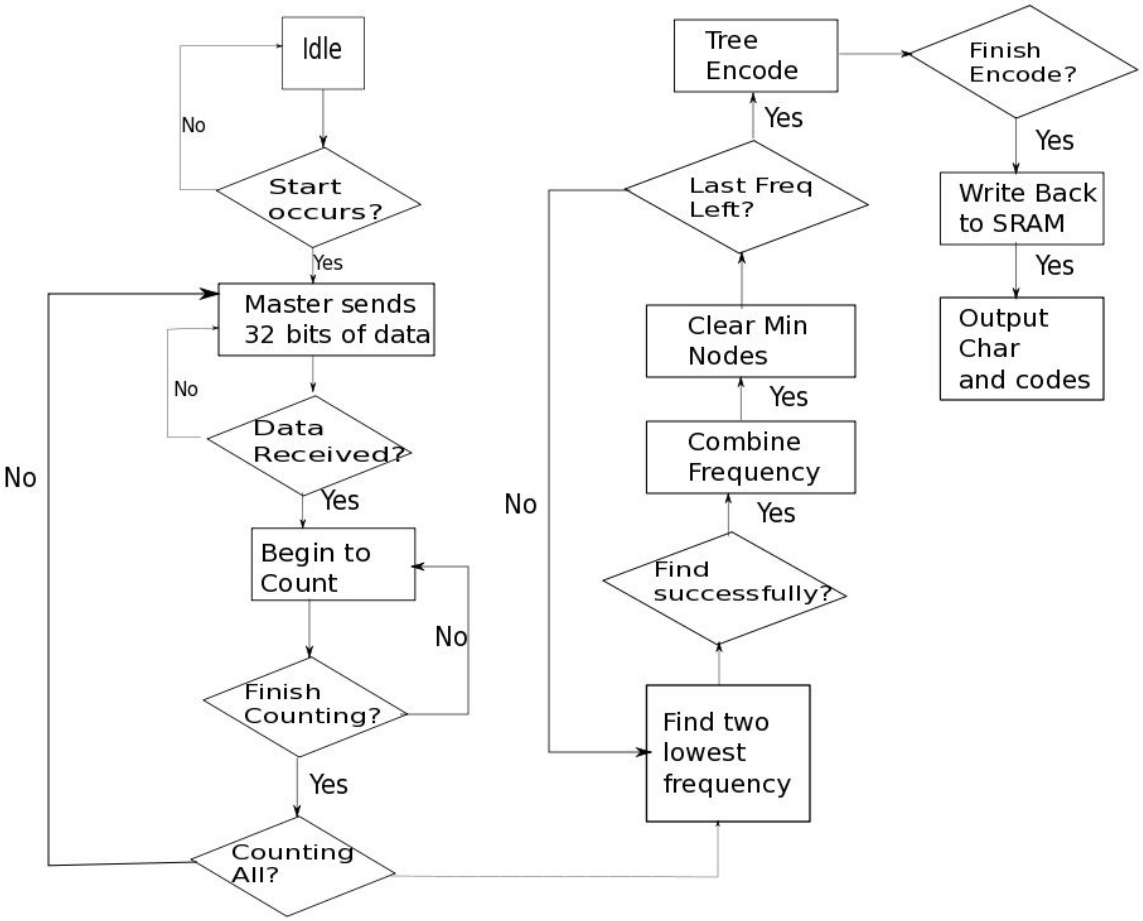
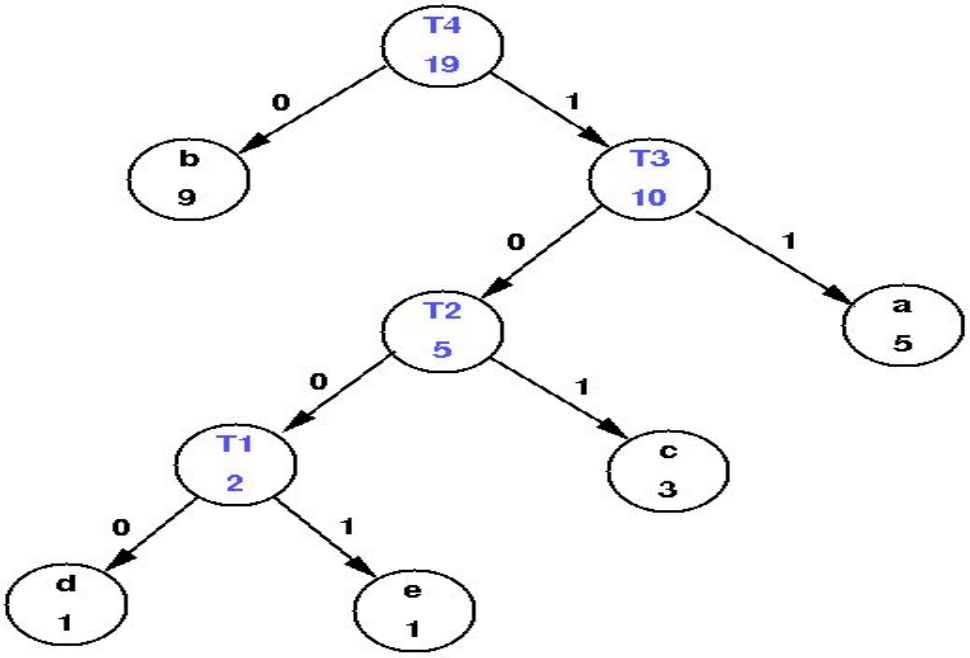


Figure 2.3.2a: The Flow Diagram of Huffman Tree Encoding



Output Tree:

- *a: 11*
- *b: 0*
- *c: 101*
- *d: 1000*
- *e: 1001*

Figure 2.3.2b: The Simple Sample of Huffman Tree Encoding

2.3.3. Supported Avalon-MM SoC Bus Transactions

1. Typical (Non-Pipelined) Single Word Transfer

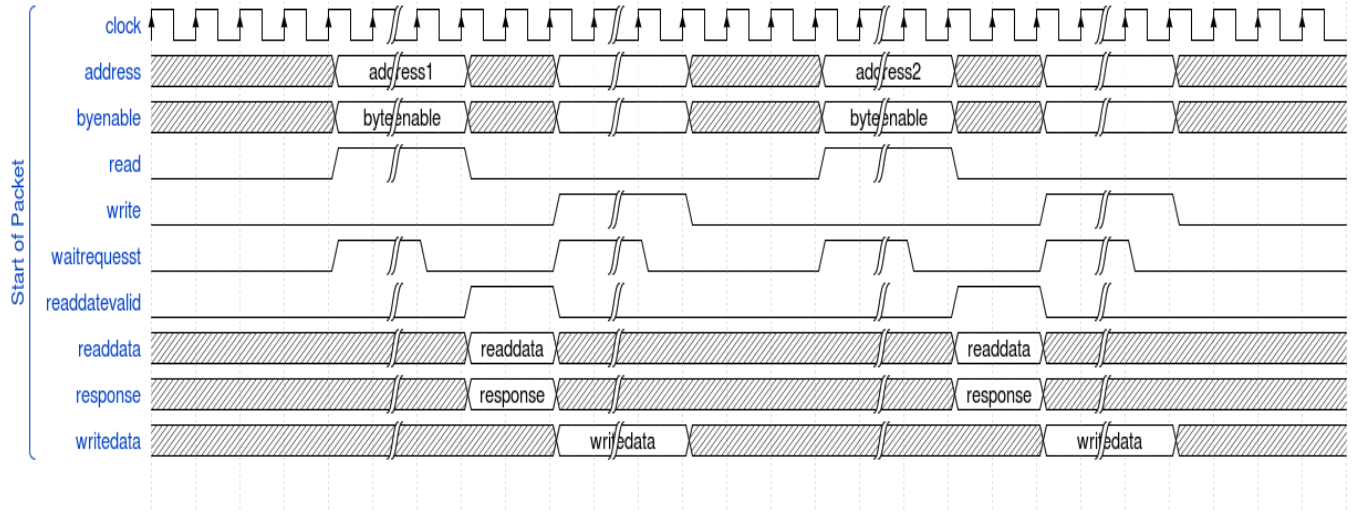


Figure 2.3.3.1: Avalon Memory Mapped Interface Single Word Transfers Example Timing Diagram

Read Transfers

During single word read transfers, the master module will assert the address bus to the desired address value, byteenable to the proper value based on the number of active bytes and their position within the word, and the read signal to a logic ‘1’ during the initial cycle of the transfer. If the slave module needs the signals from the master to remain constant for more than the initial cycle it must assert the waitrequest signal to a logic ‘1’ before the end of that cycle and hold for the 1 cycle less than the required duration. Once the slave module has the correct data value asserted on the readdata bus, it must assert the readdatavalid signal to a logic ‘1’ for 1 cycle in order to signal its response to the master. If the slave supports the usage of the response bus, it must also assert its response value during the cycle that readdatavalid is a logic ‘1’.

Write Transfers

During single word write transfers, the master module will assert the address bus to the desired address value, byteenable to the proper value based on the number of active bytes and their position within the word, the writedata bus to the data value to be written, and the write signal to a logic ‘1’ during the initial cycle of the transfer. If the slave module needs the signals from the master to remain constant for more than the initial cycle it must assert the waitrequest signal to a logic ‘1’ before the end of that cycle and hold for the 1 cycle less than the required duration. No response bus values are issued for typical writes.

2. Burst Word Read

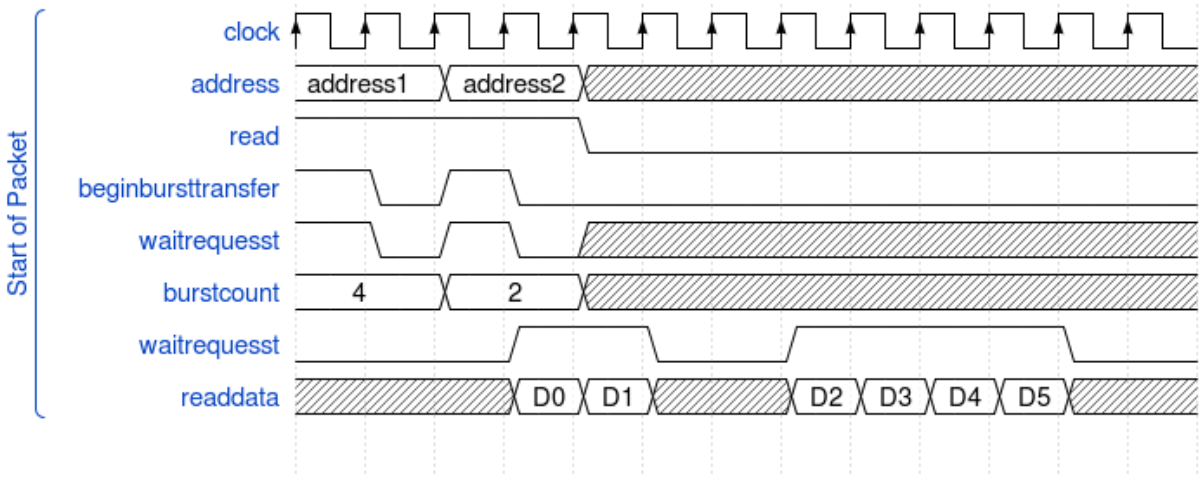


Figure 2.3.3.2: Burst Read Transfer Example Timing Diagram

Burst Read transfers are similar to pipelined read transfers in that they have defined address and data phases that may overlap between different requests (as shown in the above diagram). The address phase is identical to the initial cycle setup for a typical read, except that the beginbursttransfer is asserted to a logic '1' and the burstcount bus is asserted to be the number of data words to return for the transfer. Since burstcount value is the number of words the transfer is for, it must always be at least a value a one. Normally the burst transfers are used with the address supplied being the address of the first word, and all subsequent words being a respective word address offset from the initial address. However, to simplify the use of this accelerator's internal FIFO queues burst read transfers that start with the result word address will return successively computed results from the internal result FIFO queue.

3. Burst Word Write

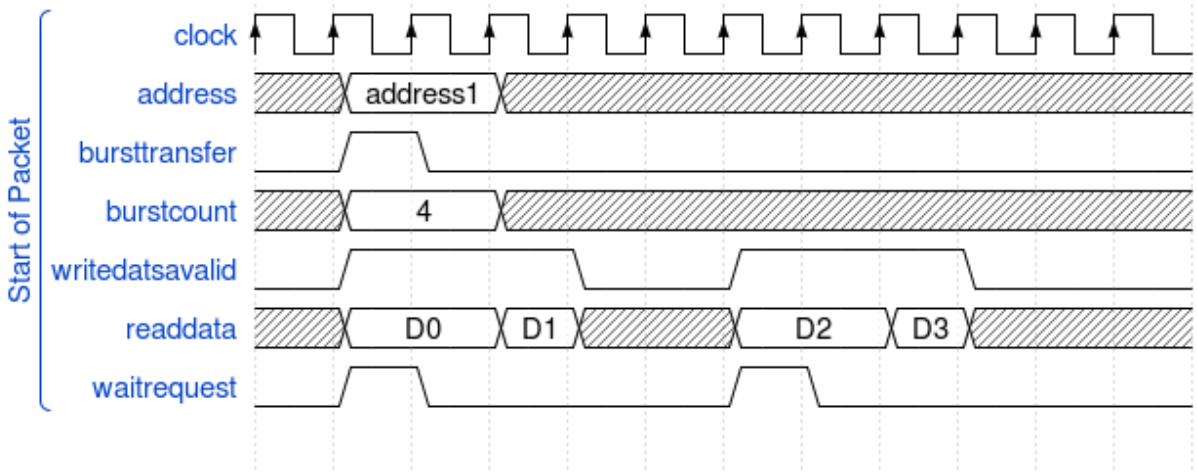


Figure 2.3.3.3: Burst Write Transfer Example Timing Diagram

In the Avalon standard writes are never pipelined since the first data value is always sent during the initial cycle that handles the 'address phase' for transfers. However, the 'address phase' related signals and the behavior of the beginbursttransfer and burstcount signals are handled identically in write bursts and read bursts. The main difference is that since the write signal is used as the writedata valid signal as well as the transfer mode indicator, it is asserted with each new word and deasserted during gaps between valid write data words during the burst transfer.

2.4. Requirements for Design

The development of this ASIC entails many design choices. The intended application of the Huffman Encoder targets efficient transmission of ASCII-based data, namely human-readable messages, with reasonable latency (sub-second operation). Specifically, the transmitted ASCII data will be 8-bit ASCII encoding or UTF-8 (both encoding formats can be fitted into 7-bit words) . Additionally, messages can support up to 65,536(2^16) ASCII characters. As previously mentioned, the Huffman Tree can be created in O(n) time complexity, with a possible maximum of 2^7 nodes (128 nodes).

Provided with these specifications, and according to the operation calculations in Section 2.3.2, the maximum number of operations we would have to handle would be 7n - 6, where n (number of ASCII chars in message) is equal to 65,536. This results in 7(65,536) - 6 = 458,746 operations, which must be processed in under a second.

Operating at a clock frequency of 100MHz would satisfy this requirement. Our throughout is 65,536 samples/sec * 4 bytes/channel * 1 channel = 262,144.

Optimization: In order to save on size of design, we decided to use burst read and write so that we can read chunks of data at a time to save memory. Additionally, burst read and write provides us with a more flexible mode so the data will continue processing as long as it is available. We use only one flip flop as timer to increase the count by 1. All character frequencies after being counted will be saved to sRAM. With this design, we greatly reduce the number of flip flops required.
Our algorithm is designed to accelerate the whole procedure. After receiving frequencies for all characters, the module will deal with them by a two step process instead of one. By one for loop we will easily find the lowest two frequencies and the corresponding characters. Those will be sent to another combine frequency block. It will also take a for loop and continue to build the tree. Two separate blocks may take one more clock cycle but a O(n) time complexity save a lot than O(n^2) or O(n*logn).

Our Area budget is 5mm * 5mm due to huge chunks of data the module has to deal with in one second. Our timing budget is 10 ns clock period, which is 100MHZ.

3. Design Implementation

3.1. Design Architecture

3.1.1. Architectural Block Diagram

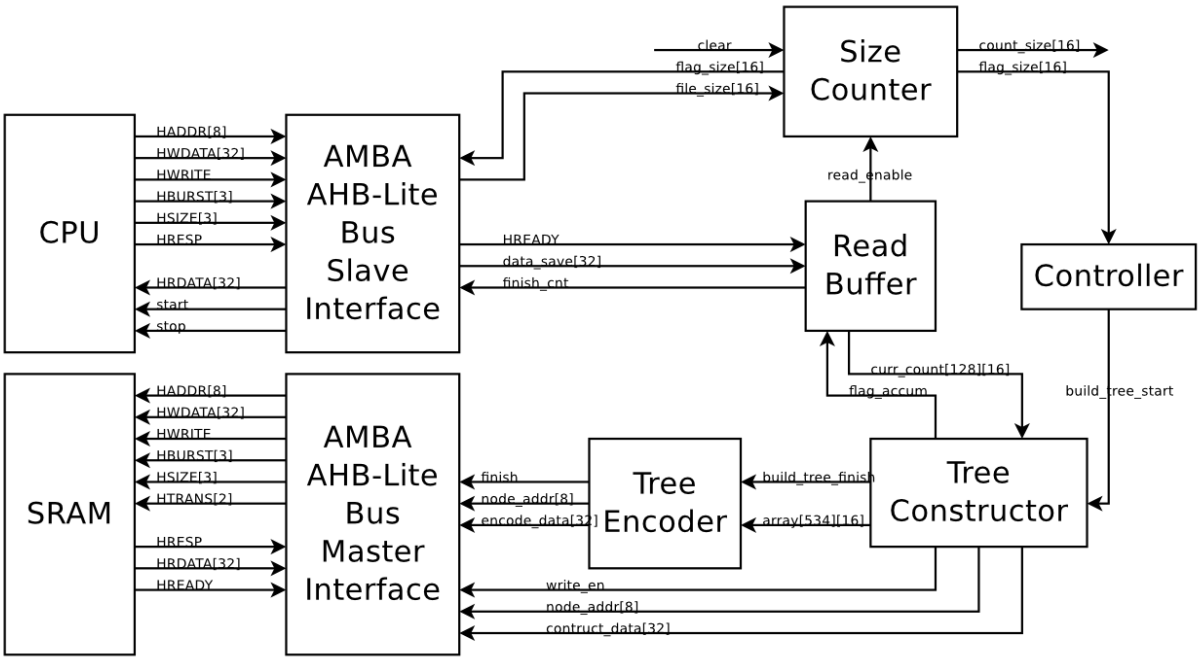


Figure 3.1.1: Huffman Encoding Module Architecture Diagram

The intended implementation diagram is shown above in Figure 3.1.1. The avalon bus interface device builds a connection between the storage and operating blocks. In general is the following: Avalon will get the a chunk of data's address from processor and then send it to huffman computation core. It will execute huffman filter and rearrange each character inside the data file. All the data will then flow to Huffman Frequency Tree. The main control unit is more like a wrapper file, where will give access to execute all functions.

3.1.2. Operations

1. Receive Operation

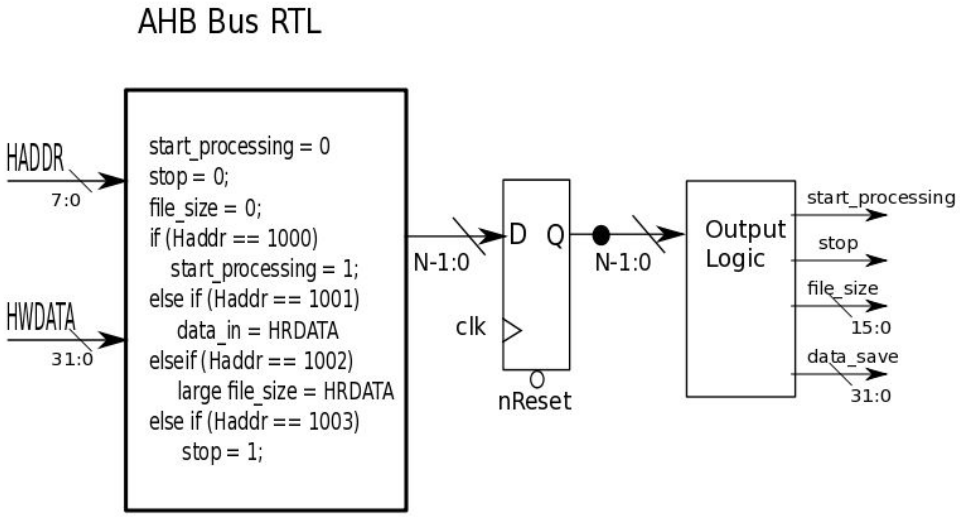


Figure 3.1.2.1a: The RTL diagram of AHB Bus

Description:

AHB Bus is the media that enable our huffman tree module to communicate with CPU and SRAM in a fast and efficient way. When CPU begins to send data address to AHB, Avalon takes it and compare in order to know when to start process data, the size of data and also stop sending the data.

Sending Operation

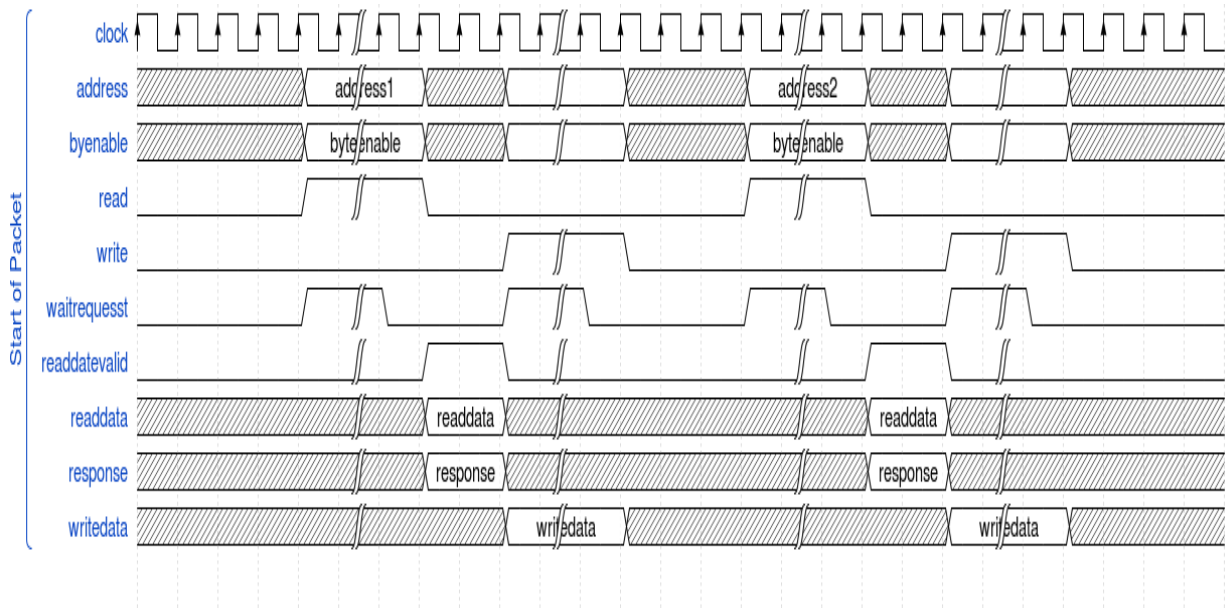


Figure 3.1.2.1b: AHB Memory Mapped Interface Single Word Transfers Example Timing Diagram

AHB Area:

```
****************************************
Report : area
Design : AHB
Version: K-2015.06-SP1
Date   : Wed May  3 19:26:52 2017
****************************************

Library(s) Used:

    osu05_stdcells (File: /package/eda/cells/OSU/v2.7/synopsys/lib/ami05/osu05_stdcells.db)

Number of ports:                          322
Number of nets:                           439
Number of cells:                          212
Number of combinational cells:            106
Number of sequential cells:               104
Number of macros/black boxes:               0
Number of buf/inv:                         24
Number of references:                       2

Combinational area:              28872.000000
Buf/Inv area:                     3600.000000
Noncombinational area:           82368.000000
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:                111240.000000
Total area:                 undefined
1
```

Area: 1.112 mm^2

Description

During single word read transfers, the master module will assert the address bus to the desired address value, byteenable to the proper value based on the number of active bytes and their position within the word, and the read signal to a logic ‘1’ during the initial cycle of the transfer. If the slave module needs the signals from the master to remain constant for more than the initial cycle it must assert the waitrequest signal to a logic ‘1’ before the end of that cycle and hold for the 1 cycle less than the required duration. Once the slave module has the correct data value asserted on the readdata bus, it must assert the readdatavalid signal to a logic ‘1’ for 1 cycle in order to signal its response to the master. If the slave supports the usage of the response bus, it must also assert its response value during the cycle that readdatavalid is a logic ‘1’.

3.1.3. Area Budget

Core Area Calculations				
Name of Block	**Category**	**Gate/FF Count**	**Area (um2)**	**Comments**
On-chip SRAM	Reg. w/ Reset	776	1,862,400	tree storage: max nodes = equal frequencies for all 128 chars (128 leaves, so 128*2 - 1 total nodes, which means probably (128*2 - 1) * (1 + 2) (due to char (8bit) and frequency (16bit)), in addition, unencoded message must be stored to be encoded later
Controller Next-State Logic	Combinational	24	18,000	# of states + # of inputs to get to each state
Controller State Register	Reg. w/ Reset	4	9,600	11 states = 4 FF
Controller Output Logic	Combinational	5	3,750	# of states with outputs
Accumulator Counter Next Count Logic	Combinational	4	3,000	bits * 2 (1 bit is a full adder, 2 gates)
Accumulator Counter Count Register	Reg. w/ Reset	2	4,800	2-bit
Accumulator Counter Output Logic	Combinational	2	1,500	
Size Counter Next Count Logic	Combinational	32	24,000	bits * 2
Size Counter Count	Reg. w/ Reset	16	38,400	16-bit

Register				
Size Counter Output Logic	Combinational	16	12,000	
Read Buffer Logic	Combinational	1007	755,250	for a 2^n:1 mux, there are (n + 2^n + 1) gates. muxes added first. assume adder is 5 gates/bit
Read Buffer Register	Reg. w/ Reset	4352	10,444,800	
Tree Constructor Next-State Logic	Combinational	16	12,000	# of conditional input (if/else) * 4 + next state assignments
Tree Constructor State Register	Reg. w/ Reset	4	9,600	10 states = 4 bits
Tree Constructor Other Register	Reg. w/ Reset	4248	10,195,200	10 states = 4 bits
Tree Constructor Output Logic	Combinational	31	23,250	# outputs + # of conditional statements (4 per if/else)
Tree Encoder Next-State Logic	Combinational	130	97,500	# of conditional input (if/else) * 4 + next state assignments
Tree Encoder State Register	Reg. w/ Reset	6	14,400	33 states = 6 bits
Tree Encoder Other Register	Reg. w/ Reset	146	350,400	
Tree Encoder Output Logic	Combinational	137	102,750	# outputs + # of conditional statements (4 per if/else)
Total Core Area			23,982,600	

Chip Area Calculations (units in um or um2)

Number of I/O Pads:	-		
I/O Pad Dimensions:	-	by	-
I/O Based Padframe Dimensions:	-	by	-
Core Dimensions	4,897	by	4,897
Core Based Padframe Dimensions:	4,897	by	4,897
Final Padframe Dimensions:	4,897	by	4,897
Final Chip Area:	23,982,600		

Area Budget Discussion

The bulk of our area requirements seems to originate from the Read Buffer and Tree Construct modules, unsurprisingly. This is due to the enormous amount of storage capabilities required to count frequencies and construct the Huffman Tree in the first place.

3.2. Functional Block Diagrams

3.2.1. Controller

Top Level

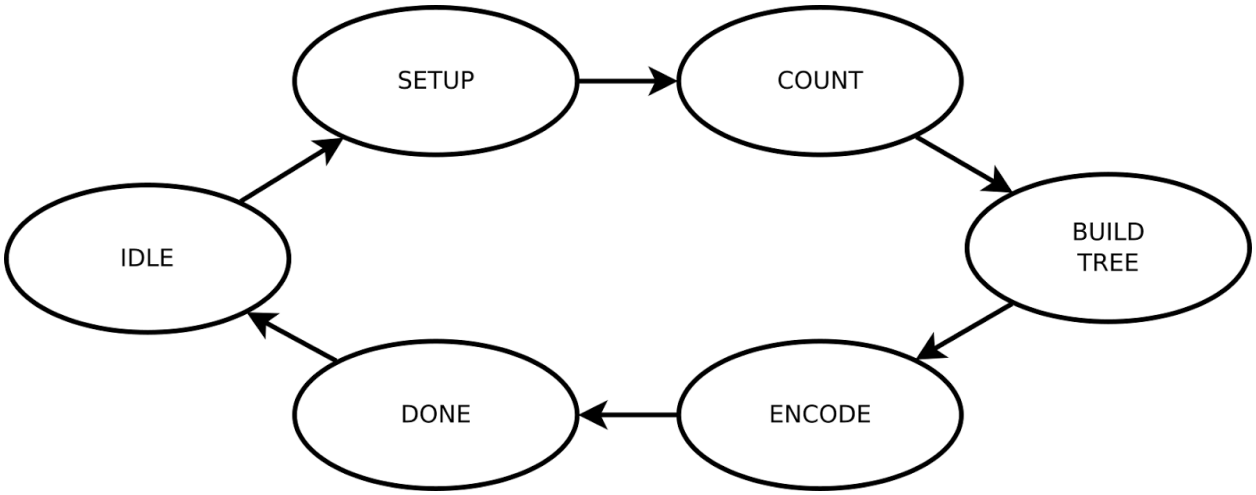


Figure 3.2.1a: Abstract top level flowchart of controller.

State Diagram

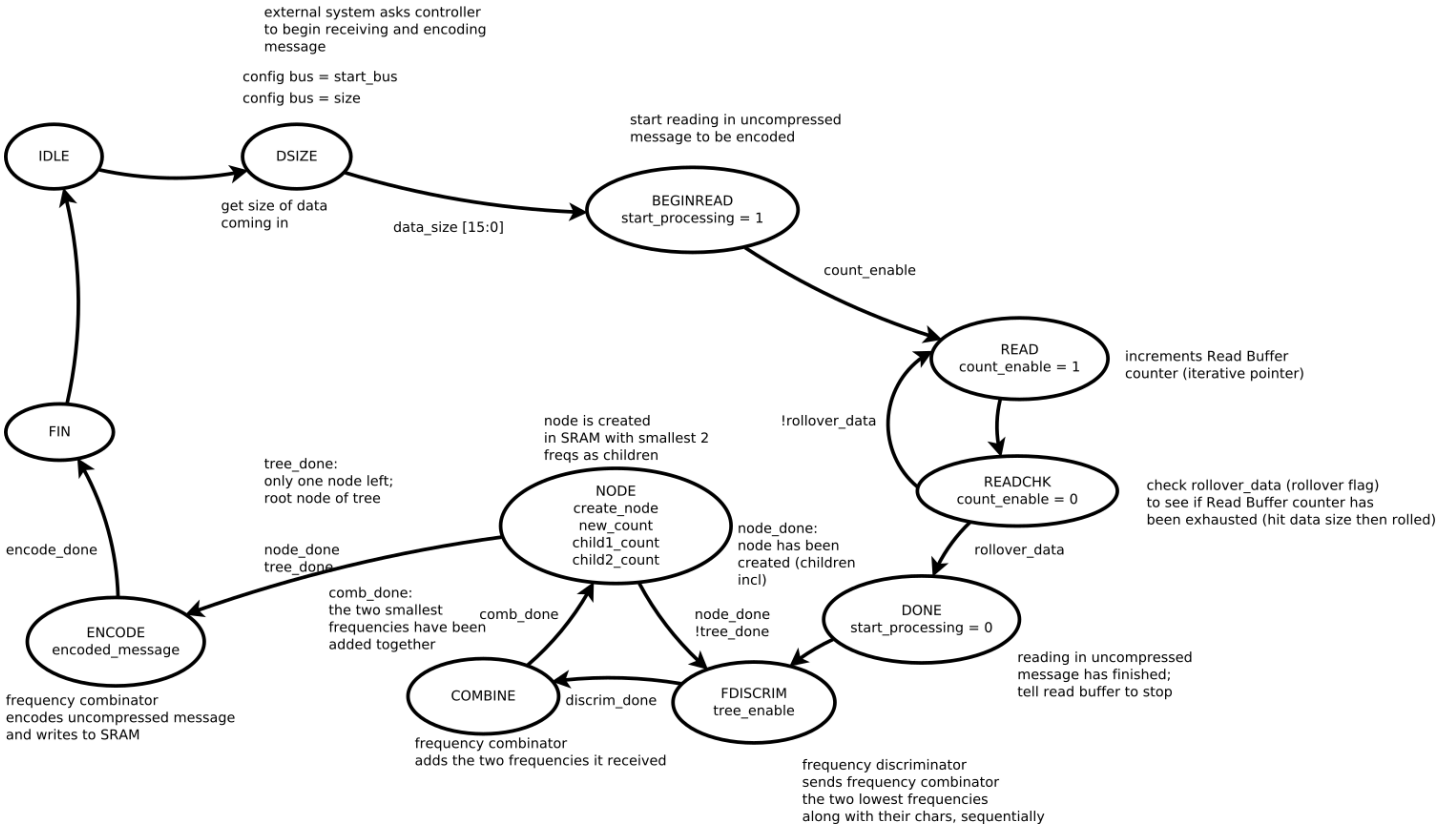


Figure 3.2.1b: State Diagram of controller.

Timing Waveform Diagram

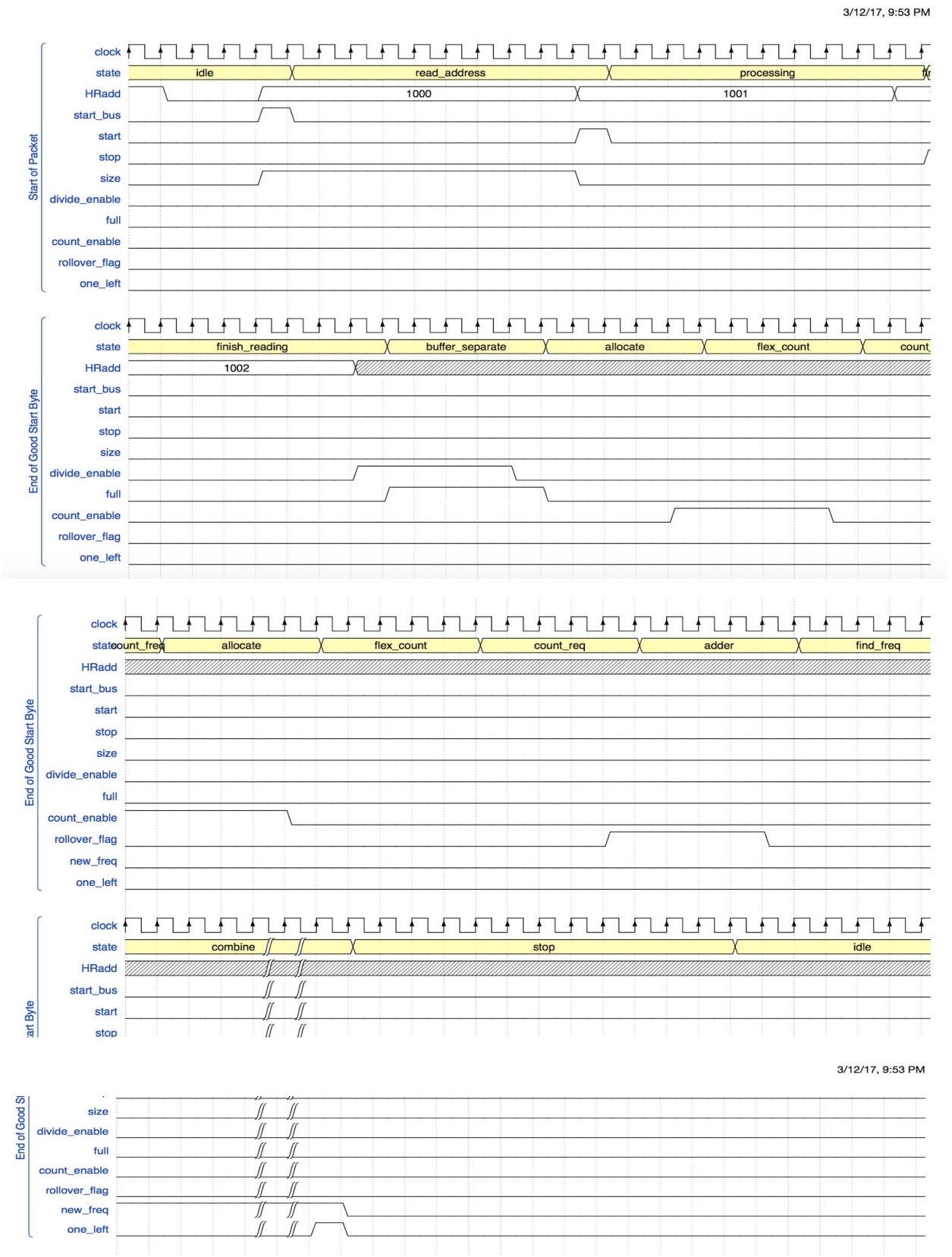


Figure 3.2.1c: The time diagram of controller.

Area Budget

Controller Next-State Logic	Combinational	24	18,000	# of states + # of inputs to get to each state
Controller State Register	Reg. w/ Reset	4	9,600	11 states = 4 FF
Controller Output Logic	Combinational	5	3,750	# of states with outputs

Table 3.2.1: Area Budget of controller.

Area Budget Description

The controller has a straightforward area budget. Having 11 states, the register only requires 4 ff. The nsl is determined by counting the number of if statements required to realize the logic, which amounts to the number of states with the number of inputs to get between states. If the same input signal is used in multiple places, it is counted multiple times. Similarly, the number of gates for the output logic is determined from the number of if statements, or number of states with outputs.

3.2.2. Read Buffer

Block Diagram

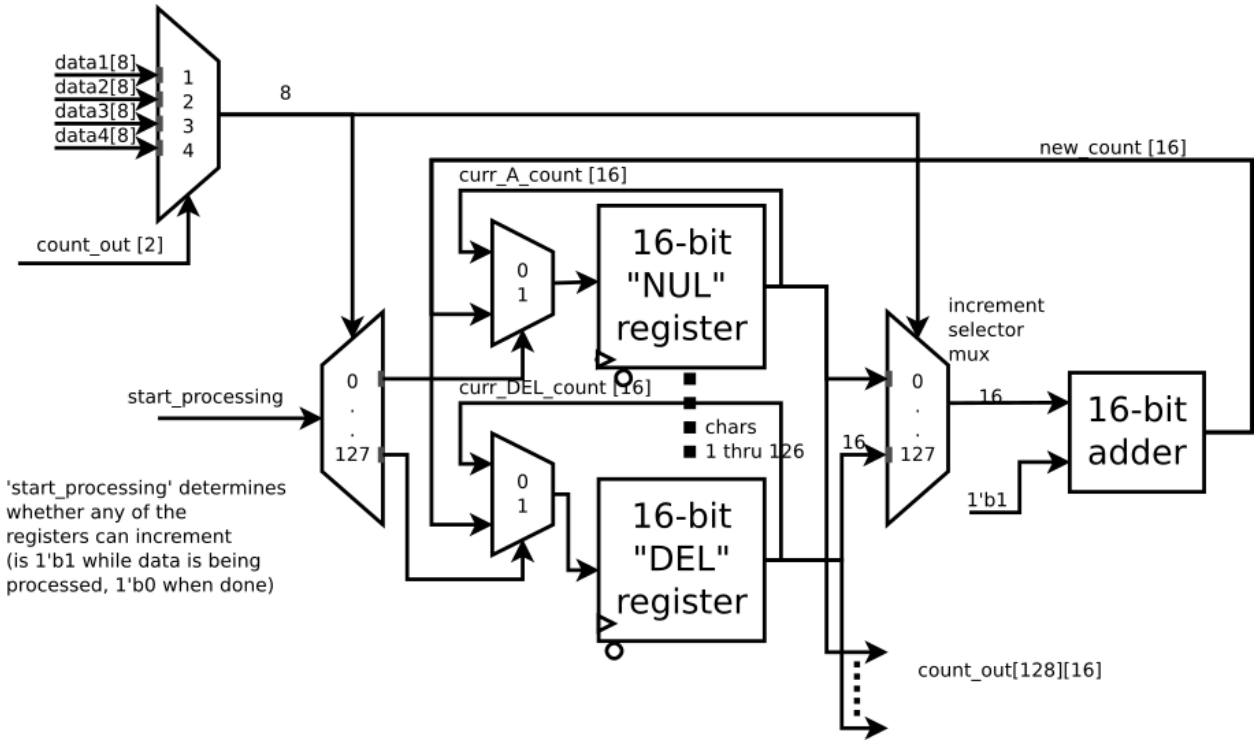
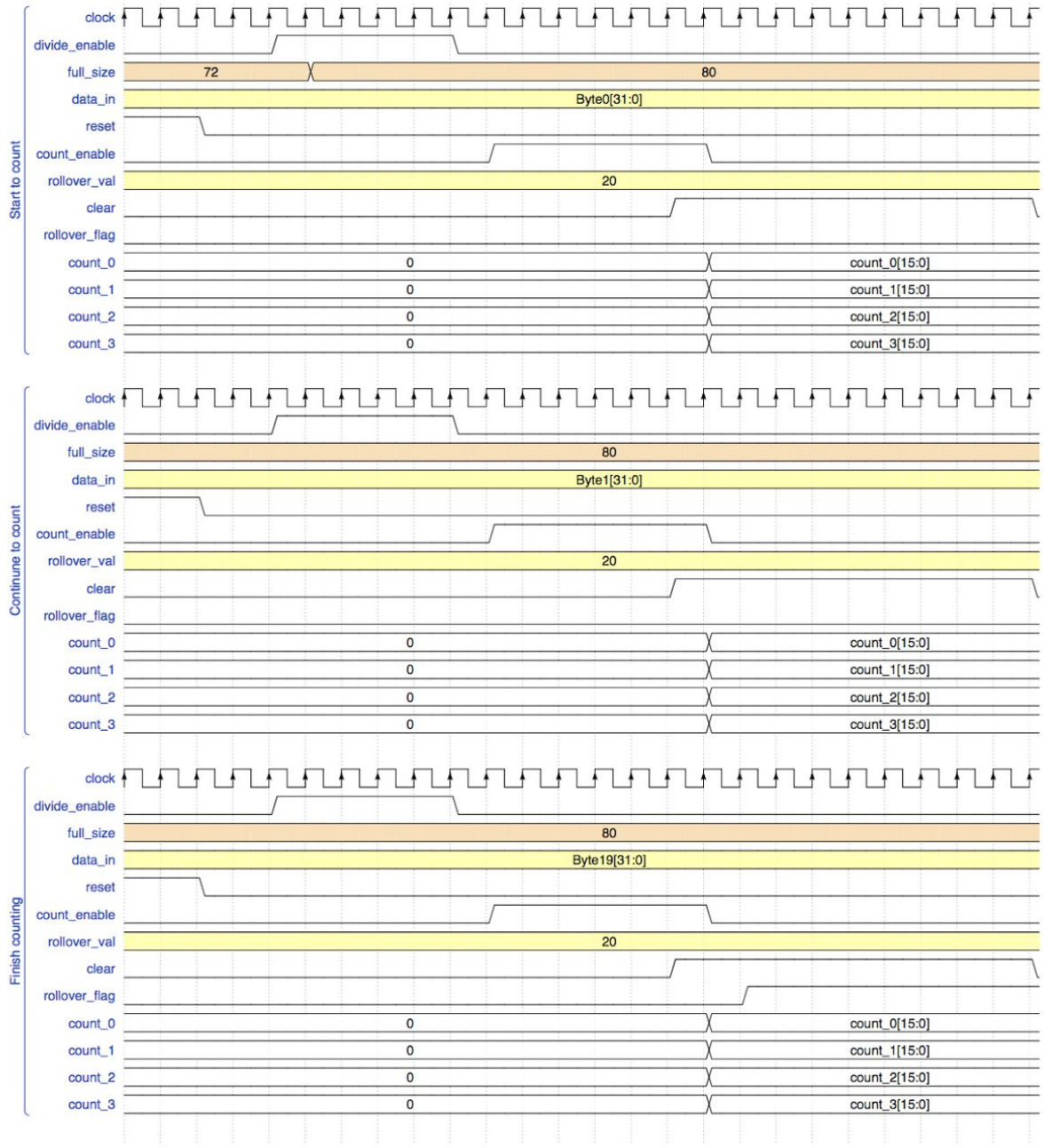
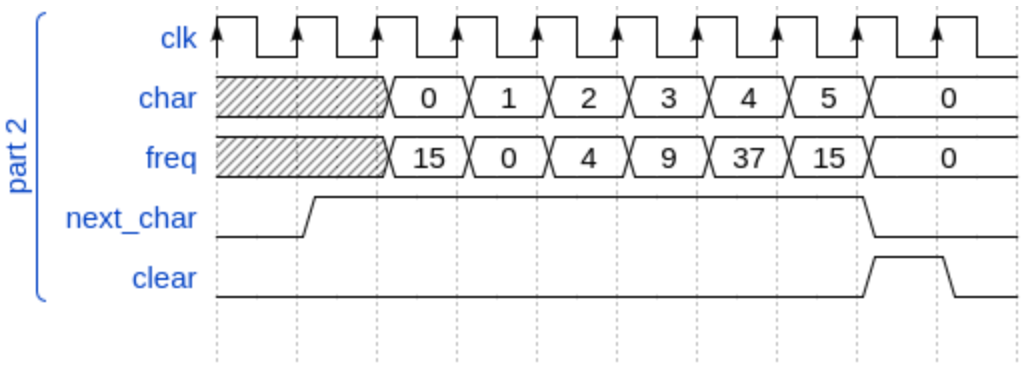


Figure 3.2.2a: The RTL diagram of read buffer

Timing Diagram:


Start to count
clock
divide_enable
full_size
72
80
data_in
Byte0[31:0]
reset
count_enable
rollover_val
20
clear
rollover_flag
count_0
0
count_0[15:0]
count_1
0
count_1[15:0]
count_2
0
count_2[15:0]
count_3
0
count_3[15:0]
Continune to count
clock
divide_enable
full_size
80
data_in
Byte1[31:0]
reset
count_enable
rollover_val
20
clear
rollover_flag
count_0
0
count_0[15:0]
count_1
0
count_1[15:0]
count_2
0
count_2[15:0]
count_3
0
count_3[15:0]
Finish counting
clock
divide_enable
full_size
80
data_in
Byte19[31:0]
reset
count_enable
rollover_val
20
clear
rollover_flag
count_0
0
count_0[15:0]
count_1
0
count_1[15:0]
count_2
0
count_2[15:0]
count_3
0
count_3[15:0]



Area Budget

Read Buffer Logic	Combinational	1007	755,250	for a 2^n:1 mux, there are (n + 2^n + 1) gates. muxes added first. assume adder is 5 gates/bit
Read Buffer Register	Reg. w/ Reset	4352	10,444,800	

Area Estimate: 11.1 mm^2

Area Budget Description

The Read Buffer is composed of a number of muxes, 3 of which are 128:1. for a 2^n:1 mux, there are (n + 2^n + 1) gates. Additionally, it is assumed that an adder is 5 gates/bit. The number of ff required is due to the 128 16-bit registers holding counts. Temp registers are included.

Exact Area consumption:

```
****************************************
Report : area
Design : read_buffer
Version: K-2015.06-SP1
Date   : Wed May  3 19:37:04 2017
****************************************

Library(s) Used:

    osu05_stdcells (File: /package/eda/cells/OSU/v2.7/synopsys/lib/ami05/osu05_stdcells.db)

Number of ports:                         6183
Number of nets:                         16510
Number of cells:                        10712
Number of combinational cells:           6486
Number of sequential cells:              4098
Number of macros/black boxes:               0
Number of buf/inv:                       2325
Number of references:                     136

Combinational area:            2482668.000000
Buf/Inv area:                   334800.000000
Noncombinational area:         3245616.000000
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:               5728284.000000
Total area:                 undefined
1
```

Area_read_buffer = 5.728 mm^2

Description

The Read Buffer is responsible for reading in a data chunk of 4 bytes and processing 4 bytes (serially) every clock cycle; processing of the 4 bytes occurs concurrently with the burst read of the 4 byte data chunk. A mux controlled by a 2-bit counter will serially output 1 of the 4 bytes. This output, a char byte, will control 2 more muxes. These 2 muxes are responsible for gating the appropriate char and freq count so that if detected, the value is incremented by 1 and resaved. The design in place also ensures that the rest of the undetected chars retain their current count.

3.2.3. Tree Constructor

State Diagram

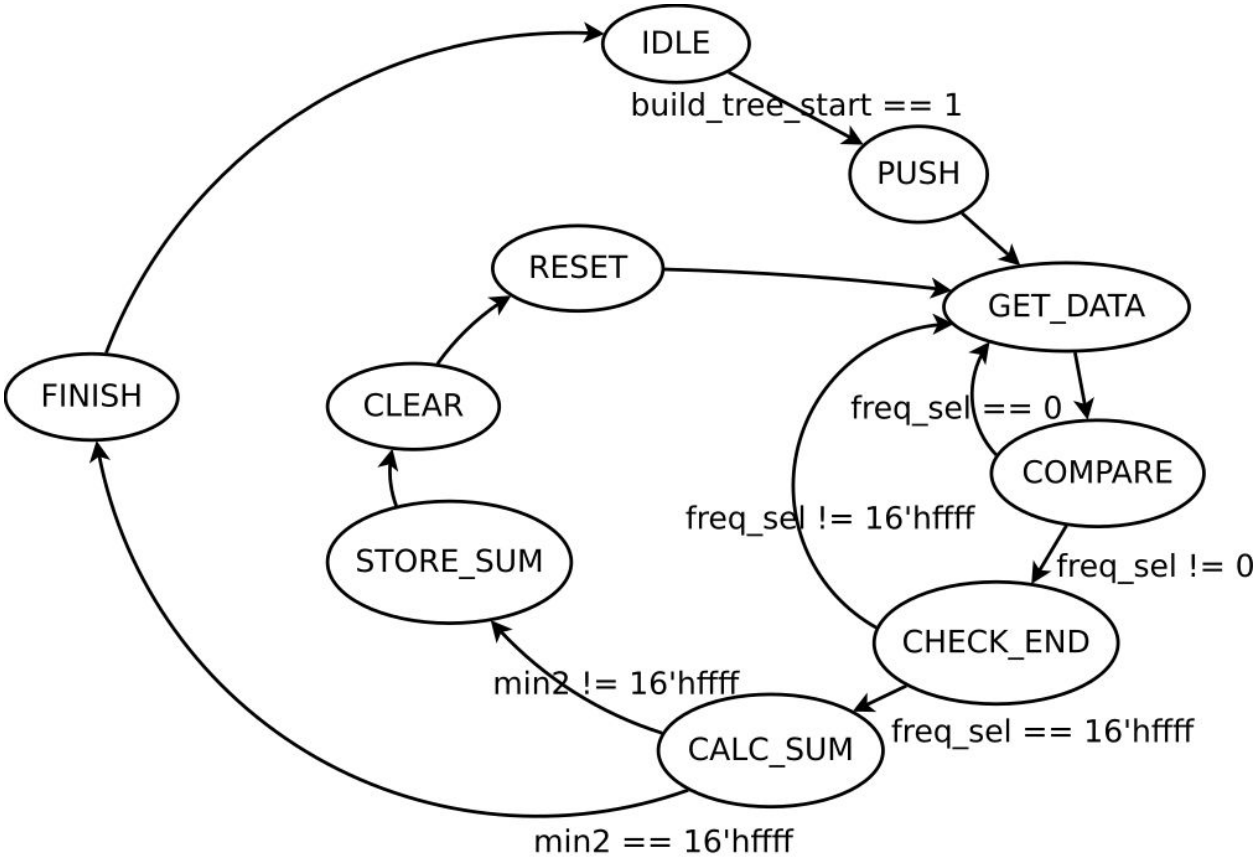


Figure 3.2.3a: The STD diagram of Tree Constructor

Description

Tree Constructor is responsible to build the tree based on all characters' frequency. This function block was designed majorly by FSM. All characters' frequencies will be saved into array[128] to array[255]. By going through the whole big array, every frequency (leaf frequencies and node frequencies) is compared to get the two lowest frequency. Combine them and write it to the tail (starts from 256). After the sum node is saved successfully, the next two elements are this node's left child address and right child address. Then, the two nodes which hold lowest frequencies will be clear, by setting the frequency to zero. This processing will keep going until the end sign "16'hffff" is reached.

Area Budget:

Tree Constructor Next-State Logic	Combinational	16	12,000	# of conditional input (if/else) * 4 + next state assignments
Tree Constructor State Register	Reg. w/ Reset	4	9,600	10 states = 4 bits
Tree Constructor Other Register	Reg. w/ Reset	4248	10,195,200	10 states = 4 bits
Tree Constructor Output Logic	Combinational	31	23,250	# outputs + # of conditional statements (4 per if/else)

Area estimate: 10 mm^2

Area Budget Description

This module takes a lot of space because it has to house the constructed tree (due to SRAM not coming to fruition).

```
****************************************
Report : area
Design : tree_construct
Version: K-2015.06-SP1
Date   : Wed May  3 19:47:18 2017
****************************************

Library(s) Used:

    osu05_stdcells (File: /package/eda/cells/OSU/v2.7/synopsys/lib/ami05/osu05_stdcells.db)

Number of ports:                        10740
Number of nets:                         67782
Number of cells:                        65579
Number of combinational cells:          48224
Number of sequential cells:             17351
Number of macros/black boxes:               0
Number of buf/inv:                      12316
Number of references:                      20

Combinational area:           12924450.000000
Buf/Inv area:                  1993824.000000
Noncombinational area:        13738032.000000
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:              26662482.000000
Total area:                 undefined
1
```

Area: 26.662 mm^2

3.2.4. Tree Encode

State Diagram

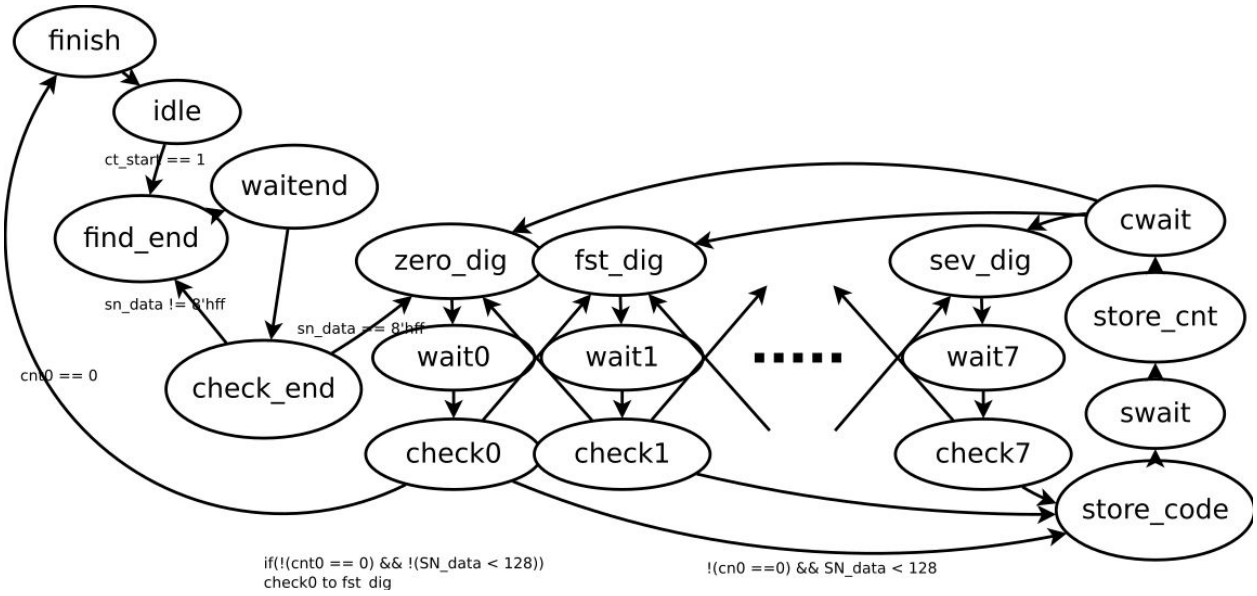


Figure 3.2.4a: The STD diagram of Tree Encoder

Description

The tree is encoded based on the tree construction. The digit for each node is initialized by 2. There are totally 8 digits. The encoding process always begins from the end of the array. First the pointer moves to the last node's left child, then the digit will be modified to 1. Going to left node's address, and check if it is a node or character. If it is a node, goes to the left node's address again and change the second digit to 1. Keep this process until the last address represents a character(index from 128 to 255). Count how many digits have been modified and that is the encode of one character. After finishing left branch, go to the right branch and change the digit from 1 to 0. After going through all right branches, clear all digits back to 2. All the process is controlled by a FSM(state machine).

Area Budget:

Tree Encoder Next-State Logic	Combinational	130	97,500	# of conditional input (if/else) * 4 + next state assignments
Tree Encoder State Register	Reg. w/ Reset	6	14,400	33 states = 6 bits
Tree Encoder Other Register	Reg. w/ Reset	146	350,400	
Tree Encoder Output Logic	Combinational	137	102,750	# outputs + # of conditional statements (4 per if/else)

Area Estimate:

0.55 mm^2

Area Budget Description

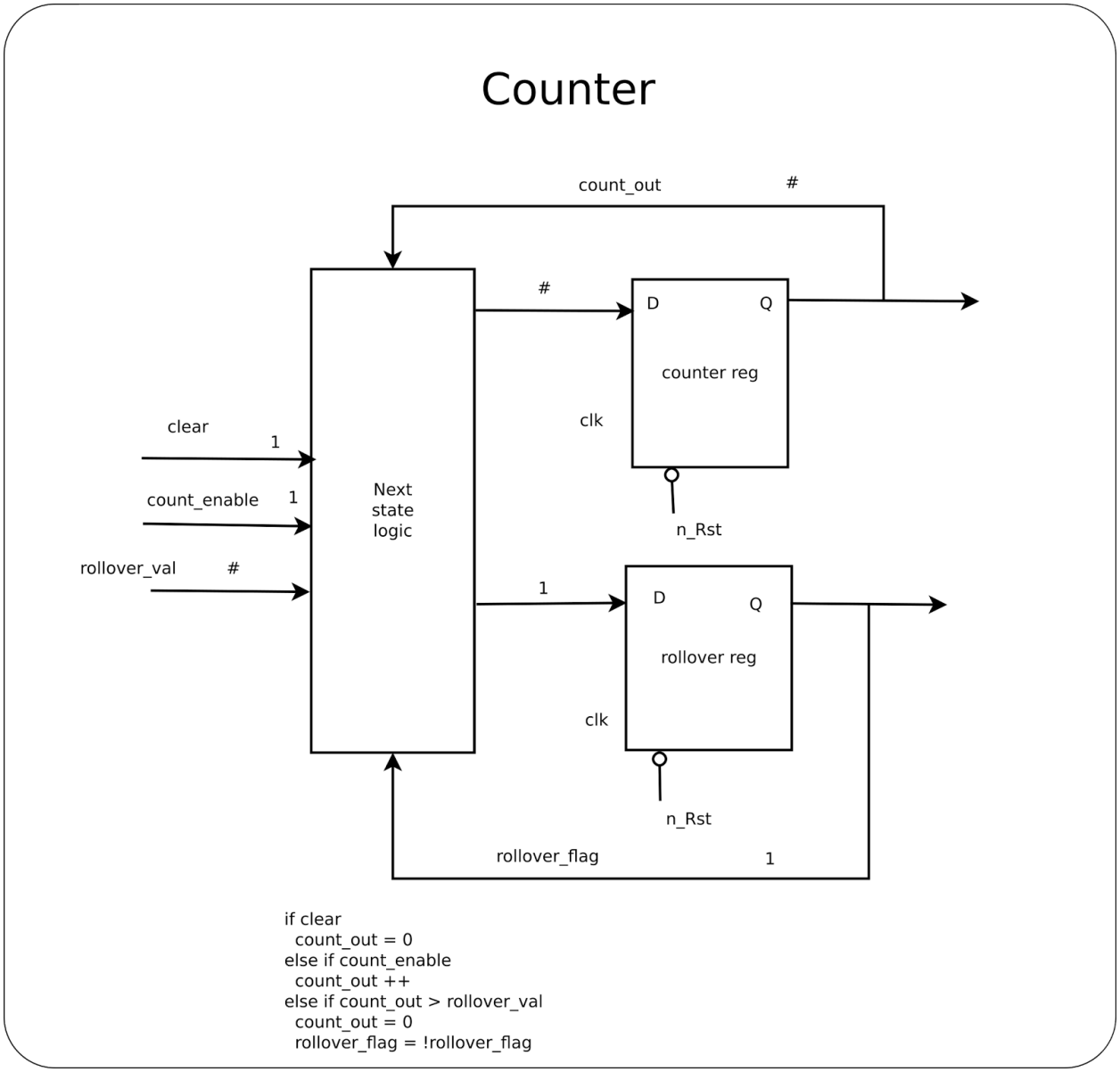
This module doesn't take too many resources because it simply has to interpret all the computed information from Tree Construction.

```
****************************************
Report : area
Design : tree_encode
Version: K-2015.06-SP1
Date   : Wed May  3 20:02:26 2017
****************************************

Library(s) Used:

    osu05_stdcells (File: /package/eda/cells/OSU/v2.7/synopsys/lib/ami05/osu05_stdcells.db)

Number of ports:                           48
Number of nets:                          1296
Number of cells:                         1193
Number of combinational cells:           1012
Number of sequential cells:               180
Number of macros/black boxes:               0
Number of buf/inv:                        153
Number of references:                      20

Combinational area:             329382.000000
Buf/Inv area:                    22320.000000
Noncombinational area:           85536.000000
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:                414918.000000
Total area:                 undefined
```

Area Actual: 0.415 mm^2

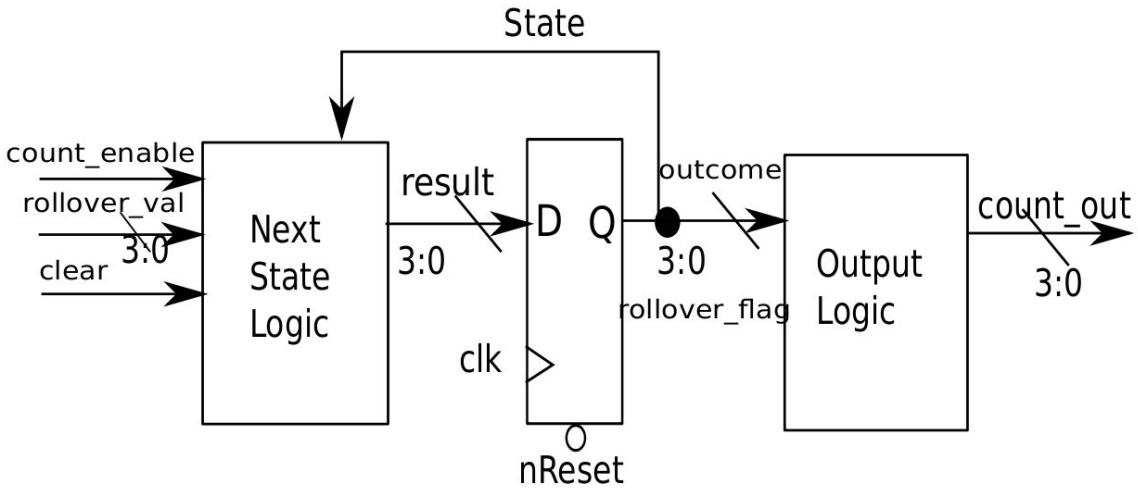
3.2.5. Size Counter

RTL Diagram



Size counter keeps track of whether the incoming uncompressed data has been finished interpreting. The input rollover_value is the file size. Every time it will be increase by 4 until it reaches the rollover_value.

Accumulate Counter:



Accumulate counter is used to keep track if all 4 bytes data have been counted.

Area:

```
****************************************
Report : area
Design : accumu_counter
Version: K-2015.06-SP1
Date   : Wed May  3 19:32:33 2017
****************************************

Library(s) Used:

    osu05_stdcells (File: /package/eda/cells/OSU/v2.7/synopsys/lib/ami05/osu05_stdcells.db)

Number of ports:                           19
Number of nets:                            69
Number of cells:                           55
Number of combinational cells:             44
Number of sequential cells:                10
Number of macros/black boxes:               0
Number of buf/inv:                         10
Number of references:                       1

Combinational area:              11052.000000
Buf/Inv area:                     1440.000000
Noncombinational area:            6336.000000
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:                 17388.000000
Total area:                 undefined
1
```

Area: 0.17388 mm^2

3.3. Design Timing Analysis

3.3.1 Block Category

Block Name	Category
Avalon Bus	Combinational
Flex Counter	Register
Controller Next-State Logic	Combinational
Controller State Logic	Register
Controller Output Logic	Combinational
Read Buffer adder	Combinational
Read Buffer Detector	Register
Read Buffer election	Combinational
Tree frequency election	Combinational
Tree combine frequency	Combinational
Tree build Tree	Register
Tree encode	Register

Table 13: The Block Category

3.3.2 Assumption

Assumption	Value (ns)
Input Pad Propagation Delay	0.1
Output Pad Propagation Delay	0.2
Flip-flop Propagation Delay	0.4
Flip-flop Setup Delay	0.2
On-chip SRAM Delay	5
Off-chip SRAM Delay	10

Table 14: The Assumption of delay time

3.3.3 Timing Estimation per Path

Starting Component	Tp	Combination Logic	Tp	Ending Component	Tp	Total Delay
AHB Input Data	0.1 ns	AHB read address	1 ns	Flex Counter Input	0.2	1.3
Size Counter Input	0.2 ns	Size counter counting	1.8 ns	count_size	0.2	2.2
AHB Interface	0.2 ns	Read 4 bytes Data	1.4 ns	data_pro	0.2	1.8
Accumu counter	0.2 ns	4 bytes counting	1.5 ns	count_out	0.2	1.9
data_pro	0.2 ns	Read buffer count	4.1 ns	curr_count	0.2	4.5

curr_count	0.2 ns	Find mini freqs	4.3ns	2 low freqs	0.2	4.7
Min freqs	0.2 ns	combinator	8.1 ns	Sum freqs	0.2	8.5
Sum freqs	0.2 ns	Construct node	4.8ns	node	0.2	5.2
Node address	0.2 ns	Tree Encoding	4.8 ns	Const tree addr	0.2	5.2

Table 15: Time Estimation of Each Path Based on Assumptions

Explanation:

There are five critical paths in our module. The longest path is when we build combine frequencies. First it will take 0.2 nanosecond to get the two lowest frequencies. The combinational block is to combine frequencies. It contains two 128 to 1 muxes for characters and address, each one equals to eight 2 to 1 muxes in cascade, which bring up to 16 * 2 muxes in total. We also need 16 muxes to combine the frequencies. Therefore, it's 48 muxes in total, which will cause 48 * 0.1 = 4.8 ns delay. Additionally, to combine two 16 bits frequencies, 32 more gates are needed for adder. Therefore, in total it is 4.8 + 3.2 = 8.1 ns delay during combinational logic. Therefore the total time cycle is 8.1 + 0.2 + 0.2 = 8.5 ns.

The second longest path is when we need to write combined nodes to a specific array. It will take 0.2 ns propagation delay from register. The combining frequency is 16 bits wide, with address of 8 bits. In order to write it back to array, 24 gates are needed to pass the address and frequency. This is will cause 24 * 0.1 = 2.4 ns delay. Besides, to search the array position, 533 to 1 multiplexer are needed, which are 8 2 to 1 multiplexers. Then I need 16 gates to save the location. In general, it will bring up to (24 + 8 + 16) * 0.1 = 4.8 ns. So the total time cycle is 4.8 + 0.2 + 0.2 = 5.2 ns.

The third longest path is to encode a character. Input character from register will have a delay of 0.2 ns. The combinational block purpose is to find the encode of a character. In order to reach to the end of array, 16 gates will be need to save the location. After knowing the location, another 16 gates will be used to find the left nodes address. At the same time, 16 AND and XOR gates are used to keep track of encode digits. In total 48 * 0.1 = 4.8 ns. Finally total time cycle is 4.8 + 0.2 + 0.2 = 5.2 ns

The fourth one is the reading buffer. Input data from register will have a delay of 0.2ns. Then logic 1 is passing through an one to 128 mux. After this, the logic 1 will go to character detector to find out the corresponding char. Lastly it will go into a 2 by 1 mux to select frequency. Each frequency is 16 bits, so that I will need 32 gates to add frequencies together. In summary there are 41 muxes with couple gates, which will bring the total delay to 0.2 + 4.1 + 0.2 = 4.5 ns.

Similar to frequency counter, input data from register will have a delay of 0.2ns. Input of 127 * 16 frequencies will be saved in an array. This requires log2(127 * 16) = 11 gates. And then, 16 gates are needed to keep the address. Finally, 16 gates are used to combine two frequencies. In general there are (11 + 16 + 16) * 0.1 + 0.2 + 0.2 =4.7 ns.

Project Management

3.4. Week by Week Schedule

3.4.1. Week 9

1. Phase 3

3.4.2. Week 10 (Spring Break)

1. Finish Phase 3
2. Discuss more details of the timing of controller and computation

3.4.3. Week 11

1. Start work on Design Review
2. Start to code different blocks.

3.4.4. Week 12

1. Finish and submit Design Review
2. Finish codes and start to write test bench of each block
3. Test each block and make sure every block works
4. Start Verification plan

3.4.5. Week 13

1. Finish and submit Verification plan

2. Write test case for the wrapper huffman tree module and fully test it.

3.4.6. Week 14

1. Start work for presentations/demonstrations
2. Debugging the source and map version

3.4.7. Week 15

1. Continue work for presentations/demonstrations

3.4.8. Week 16 (Dead week)

1. Finish work for demonstration

3.4.9. Week 17 (Finals week)

1. Present Mon or Tues
2. Finish Final report for Wed

3.5. Individual Task Assignments

3.5.1. Yuqin Sophia

1. Write and design AHB Bus read
2. Design and finish read buffer block
3. Design and finish tree construct block
4. Design and finish tree encode block
5. Finish all test benches
6. Combine AHB, read buffer and tree construct together and test
7. Timing Budget
8. Reports

3.5.2. Samanth

1.

3.5.3. Alan

1. Frequency Accumulator
2. Read Buffer RTL Diagram
3. Frequency Min RTL Diagram
4. Controller State Diagram
5. Area Budget

4. Success Criteria

4.1. Fixed Criteria (12 points total)

4.1.1. (2 points) Test benches exist for all top-level components and the entire design. The test benches for the entire design can be demonstrated or documented to cover all of the functional requirements given in the design specific success criteria.

- Huffman_Tree.sv covers AHB, read buffer, controller and tree construct.
- tb_Huffman_Tree.sv is the test bench to test the top level.

4.1.2. (4 points) Entire design synthesizes completely, without any inferred latches, timing arcs, and, sensitivity list warnings.

- Huffman_Tree can be mapped successfully without any latches, timing arcs and hurtful warnings.

```
Line #:596 -- Warning:
 In design 'tree_construct', a pin on submodule 'gt_121' is connected to logic 1
 or logic 0. (LINT-32)
Line #:598 -- Warning:
 In design 'accumu_counter', a pin on submodule 'CORE' is connected to logic 1 o
r logic 0. (LINT-32)
Line #:600 -- Warning:
 In design 'accumu_counter', a pin on submodule 'CORE' is connected to logic 1 o
r logic 0. (LINT-32)
Line #:602 -- Warning:
 In design 'accumu_counter', a pin on submodule 'CORE' is connected to logic 1 o
r logic 0. (LINT-32)
Line #:604 -- Warning:
 In design 'flex_counter_NUM_CNT_BITS16', a pin on submodule 'sub_39' is connect
ed to logic 1 or logic 0. (LINT-32)
Line #:606 -- Warning:
 In design 'flex_counter_NUM_CNT_BITS16', a pin on submodule 'r300' is connected
 to logic 1 or logic 0. (LINT-32)
Line #:608 -- Warning:
 In design 'flex_counter_NUM_CNT_BITS16', a pin on submodule 'r300' is connected
 to logic 1 or logic 0. (LINT-32)
Line #:610 -- Warning:
 In design 'flex_counter_NUM_CNT_BITS16', a pin on submodule 'r300' is connected
 to logic 1 or logic 0. (LINT-32)
Line #:612 -- Warning:
 In design 'tree_construct', the same net is connected to more than one pin on s
ubmodule 'gt_127'. (LINT-33)
Line #:614 -- Warning:
 In design 'tree_construct', the same net is connected to more than one pin on s
ubmodule 'gt_121'. (LINT-33)
Line #:616 -- Warning:
 In design 'accumu_counter', the same net is connected to more than one pin on s
ubmodule 'CORE'. (LINT-33)
Line #:618 -- Warning:
 In design 'flex_counter_NUM_CNT_BITS16', the same net is connected to more than
 one pin on submodule 'r300'. (LINT-33)
mg105 ~/ece337/project >
```

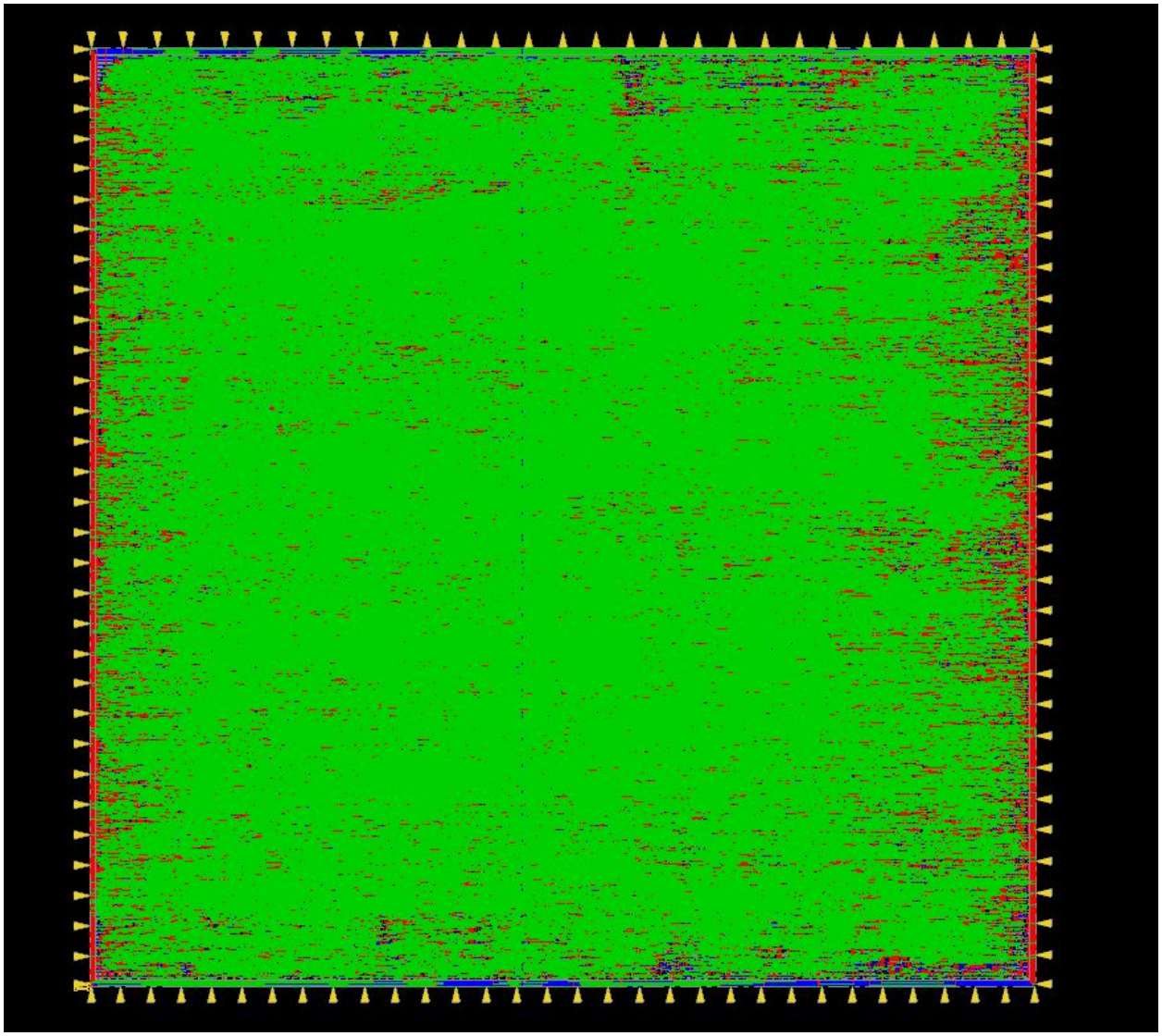
4.1.3. (2 points) Source and mapped version of the complete design behave the same for all test cases. The mapped version simulates without timing errors except at time zero.

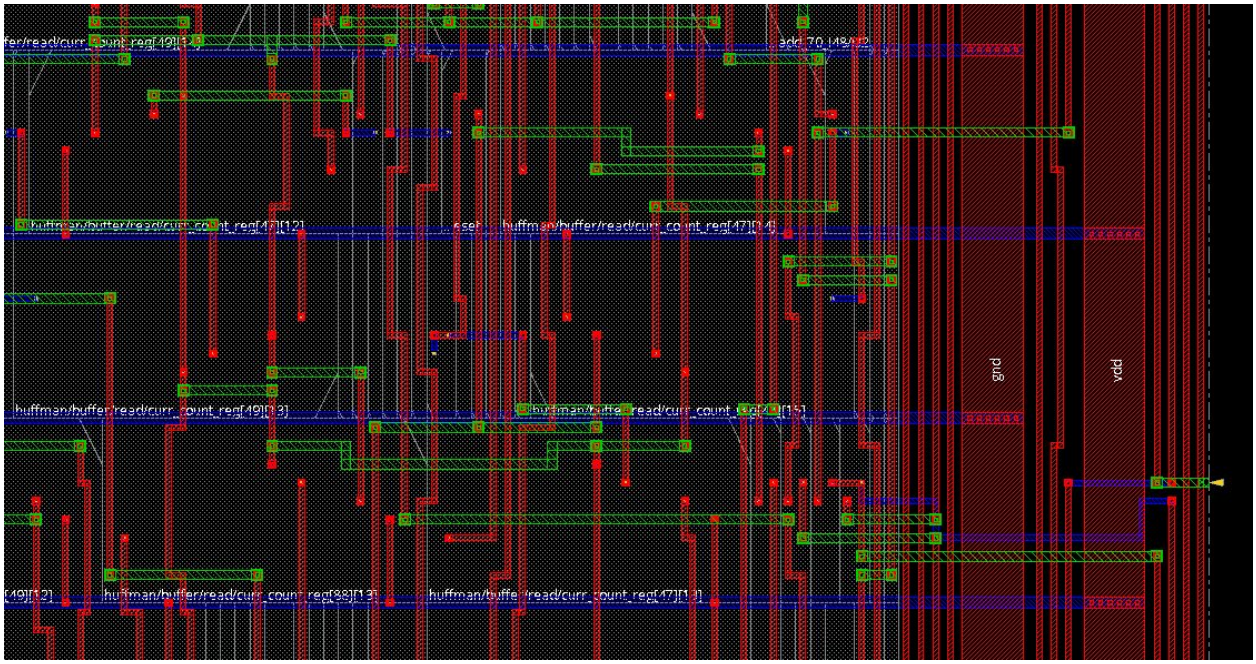
```
# vsim -Lf /home/ecegrid/a/ece337/Course_Prod/IP_Libs/Lab_IP_Lib/Vsim -
# Start time: 23:26:12 on May 03,2017
# ** Note: (vsim-8009) Loading existing optimized design _opt
# Loading sv_std.std
# Loading work.tb_Huffman_Tree(fast)
# Loading work.Huffman_Tree(fast)
# Loading work.Huffman_Read(fast)
# Loading work.AHB(fast)
# Loading work.ahb(fast)
# Loading work.ahb_resp(fast)
# Loading work.READ_BUFFER(fast)
# Loading work.control_read(fast)
# Loading work.accumu_counter(fast)
# Loading work.flex_counter(fast)
# Loading work.read_buffer(fast)
# Loading work.sizeCounter(fast)
# Loading work.flex_counter(fast__1)
# Loading work.tree_construct(fast)

VSIM 1>
```

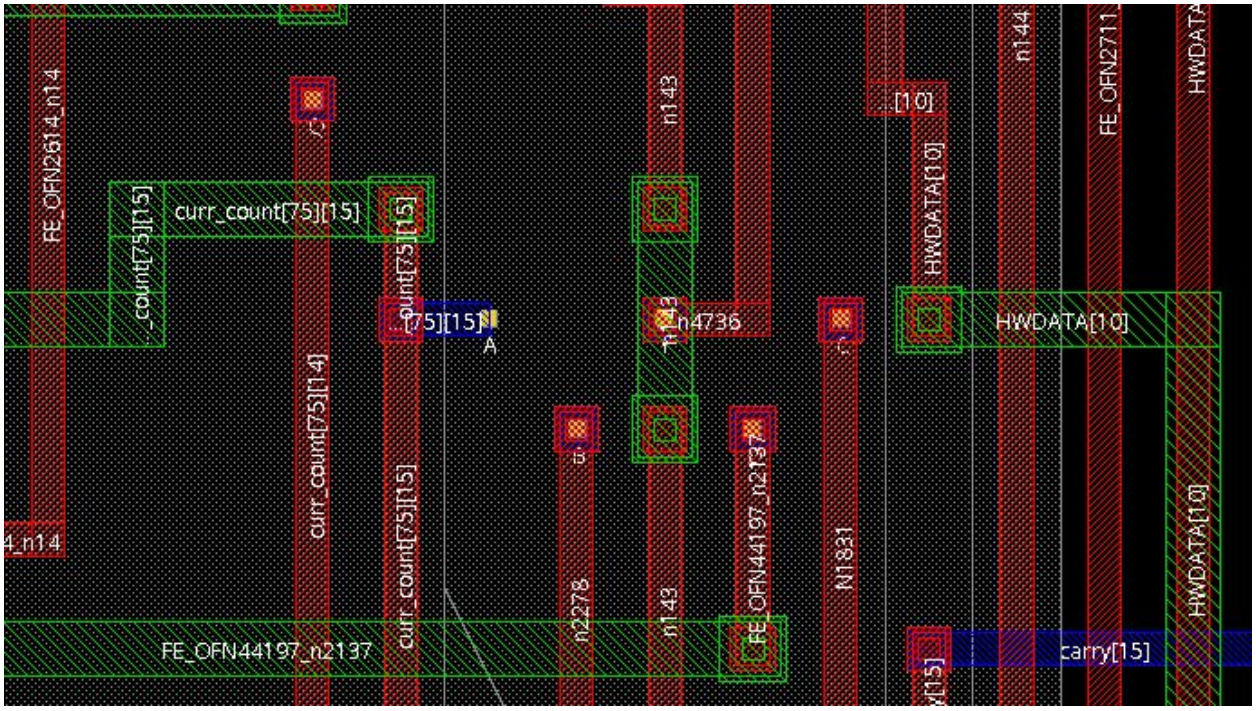
The Huffman_Tree can be synthesized without any timing errors.

4.1.4. (2 points) A complete IC layout is produced that passes all geometry and connectivity checks.

- The IC layout with 122 pins is produced that passes all geometry and connectivity checks.

- 





We zoom in, HWDATA, curr_count signal represents this is our layout

```
End Time: Wed May  3 23:31:09 2017
Time Elapsed: 0:00:06.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:05.0  MEM: -0.613M)
```

No Connectivity error.

```
 VERIFY GEOMETRY ..... Sub-Area : 166 complete 0 Viols. 0 Wrngs.
 VERIFY GEOMETRY ..... SubArea : 167 of 169
 VERIFY GEOMETRY ..... Cells          :  0 Viols.
 VERIFY GEOMETRY ..... SameNet        :  0 Viols.
 VERIFY GEOMETRY ..... Wiring         :  0 Viols.
 VERIFY GEOMETRY ..... Antenna        :  0 Viols.
 VERIFY GEOMETRY ..... Sub-Area : 167 complete 0 Viols. 0 Wrngs.
 VERIFY GEOMETRY ..... SubArea : 168 of 169
 VERIFY GEOMETRY ..... Cells          :  0 Viols.
 VERIFY GEOMETRY ..... SameNet        :  0 Viols.
 VERIFY GEOMETRY ..... Wiring         :  0 Viols.
 VERIFY GEOMETRY ..... Antenna        :  0 Viols.
 VERIFY GEOMETRY ..... Sub-Area : 168 complete 0 Viols. 0 Wrngs.
 VERIFY GEOMETRY ..... SubArea : 169 of 169
 VERIFY GEOMETRY ..... Cells          :  0 Viols.
 VERIFY GEOMETRY ..... SameNet        :  0 Viols.
 VERIFY GEOMETRY ..... Wiring         :  0 Viols.
 VERIFY GEOMETRY ..... Antenna        :  0 Viols.
 VERIFY GEOMETRY ..... Sub-Area : 169 complete 0 Viols. 0 Wrngs.
G: elapsed time: 49.00
egin Summary ...
 Cells       : 0
 SameNet     : 0
 Wiring      : 0
 Antenna     : 0
 Short       : 0
 Overlap     : 0
nd Summary

 Verification Complete : 0 Viols.  0 Wrngs.

*********End: VERIFY GEOMETRY**********
*** verify geometry (CPU: 0:00:48.8  MEM: 29.1M)
```

No geometry error or warning

4.1.5. (2 points) The entire design complies with targets for area, pin count, throughput (if applicable), and clock rate.

Estimate Area: 24 mm^2

```
****************************************
Report : area
Design : Huffman_Tree
Version: K-2015.06-SP1
Date   : Wed May  3 17:03:57 2017
****************************************

Library(s) Used:

    osu05_stdcells (File: /package/eda/cells/OSU/v2.7/synopsys/lib/ami05/osu05_stdcells.db)

Number of ports:                        30380
Number of nets:                        101511
Number of cells:                        78072
Number of combinational cells:          56314
Number of sequential cells:             21611
Number of macros/black boxes:               0
Number of buf/inv:                      15936
Number of references:                       2

Combinational area:           15767217.000000
Buf/Inv area:                  2601432.000000
Noncombinational area:        17108784.000000
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:              32876001.000000
Total area:                 undefined
```

Real Area: 32.876 mm^2

Huffman Read: 6 mm^2 + Huffman Tree: 26 mm^2

Ratio: 32/24 = 1.33

Explain: Our area is 32.876 mm^2, which is 1.31 times bigger than the estimation. The reason is because we didn't use SRAM to save the tree and encodes. If we were using SRAM, 8000 registers can be reduced since [533:0][15:0] array doesn't need to go into registers. Besides, only using registers will reduce critical path timing a lot.

Estimate Clock Cycle: 10 ns

```
Startpoint: TREE/min1_reg[0]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: TREE/sum_reg[15]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max

Point                                                    Incr       Path
-----------------------------------------------------------------------------
clock clk (rise edge)                                    0.00       0.00
clock network delay (ideal)                              0.00       0.00
TREE/min1_reg[0]/CLK (DFFSR)                             0.00 #     0.00 r
TREE/min1_reg[0]/Q (DFFSR)                               0.83       0.83 f
TREE/add_151/A[0] (tree_construct_DW01_add_3)            0.00       0.83 f
TREE/add_151/U1/Y (AND2X2)                               0.41       1.23 f
TREE/add_151/U1_1/YC (FAX1)                              0.42       1.65 f
TREE/add_151/U1_2/YC (FAX1)                              0.46       2.11 f
TREE/add_151/U1_3/YC (FAX1)                              0.46       2.57 f
TREE/add_151/U1_4/YC (FAX1)                              0.46       3.02 f
TREE/add_151/U1_5/YC (FAX1)                              0.46       3.48 f
TREE/add_151/U1_6/YC (FAX1)                              0.46       3.94 f
TREE/add_151/U1_7/YC (FAX1)                              0.46       4.40 f
TREE/add_151/U1_8/YC (FAX1)                              0.46       4.85 f
TREE/add_151/U1_9/YC (FAX1)                              0.46       5.31 f
TREE/add_151/U1_10/YC (FAX1)                             0.46       5.77 f
TREE/add_151/U1_11/YC (FAX1)                             0.46       6.22 f
TREE/add_151/U1_12/YC (FAX1)                             0.46       6.68 f
TREE/add_151/U1_13/YC (FAX1)                             0.46       7.14 f
TREE/add_151/U1_14/YC (FAX1)                             0.46       7.59 f
TREE/add_151/U1_15/YS (FAX1)                             0.42       8.02 r
TREE/add_151/SUM[15] (tree_construct_DW01_add_3)         0.00       8.02 r
TREE/U26162/Y (NAND2X1)                                  0.07       8.08 f
TREE/U26161/Y (OAI21X1)                                  0.14       8.22 r
TREE/sum_reg[15]/D (DFFSR)                               0.00       8.22 r
data arrival time                                                   8.22

clock clk (rise edge)                                   10.00      10.00
clock network delay (ideal)                              0.00      10.00
TREE/sum_reg[15]/CLK (DFFSR)                             0.00      10.00 r
library setup time                                      -0.24       9.76
data required time                                                  9.76
-----------------------------------------------------------------------------
data required time                                                  9.76
data arrival time                                                  -8.22
-----------------------------------------------------------------------------
slack (MET)                                                         1.54
```

Actual clock cycle: 8.22 ns < 10 ns

The timing is good since it can work over 100 MHZ.

4.2. Design Specific Success Criteria (8 points total)

4.2.1. (2 points) Demonstrate by simulation of Verilog test benches that the complete design is able to correctly encode a message (fully operational).

- Tree_encode.sv works properly. When I input a message, different characters can be encoded successfully since it matches the software huffman result.

4.2.2. (2 points) Demonstrate by simulation of building an encode tree.

- In design verification plan an example is showed that the tree is constructed successfully

4.2.3. (2 points) Demonstrate by simulation of Verilog test benches that the avalon bus is able to read and write from SRAM.

- Our SRAM has a good behavior of SLAVE part. It can grab the data from AHB bus correctly.
- We didn't get SRAM master work, due to timing issue, so AHB only behaves as slave.

4.2.4. (2 points) Demonstrate by simulation of Verilog test benches that the frequency accumulator module can correctly count frequencies.

- The read buffer is able to count characters' frequencies correctly, as long as the number of datas is in 65,536.

Design Verification

What to verify	Design module(s) involved	Verification procedure summary	DSSC(s) proved	Use in Final Demo	Comments
Correct chip response to	Top Level	Loopback test (trigger	DSSC 3	Yes	Use file I/O to simulate SRAM

compression request		compression request)			read & write
Correctness of Read Buffer frequency accumulation	Top Level	Compare frequency accumulation results to alternate implementation	DSSC 4	Yes	Do as part of Loopback test
Correctness of Huffman Tree Construction	Top Level	Compare tree construction results to alternate implementation	DSSC 2	Yes	Do as part of Loopback test
AMBA AHB-Lite Protocol Interfacing	Top Level	Test Bench provides samples of each type of bus transaction	DSSC 3	Yes	Should be able to reuse test bench code from AMBA bus interface controller
Correctness of Controller	Controller	Compare controller state operation to alternate implementation	DSSC 1 & 3	Only if can't show using top level	Should be able to reuse most of test bench for compression
Correctness of Read Buffer	Read Buffer	Compare frequency accumulation results to alternate implementation	DSSC 1 & 4	Only if can't show using top level	Should be able to reuse most of test bench for compression
Correctness of Size Counter	Size Counter	Compare size counter operation to alternate implementation	DSSC 4	Only if can't show using top level	Should be able to reuse most of test bench for compression
Correctness of Accumulator Counter	Accumulator Counter	Compare frequency counter operation to alternate implementation	DSSC 4	Only if can't show using top level	Should be able to reuse most of test bench for compression
Correctness of Tree Construct	Tree Construct	Compare frequency min results to alternate implementation	DSSC 2	Only if can't show using top level	Should be able to construct the tree
Correctness of Tree Encode	Tree Encode	Compare encodes to encodes generated by	DSSC 2	Only if can't show using top level	Should be able to encode the tree correctly

		software tool			
AMBA AHB-Lite Protocol Interfacing	AMBA AHB Interfacing	Test Bench provides samples of each type of bus transaction	DSSC 3	Only if can't show using top level	Should be able to reuse test bench code from AMBA bus interface controller

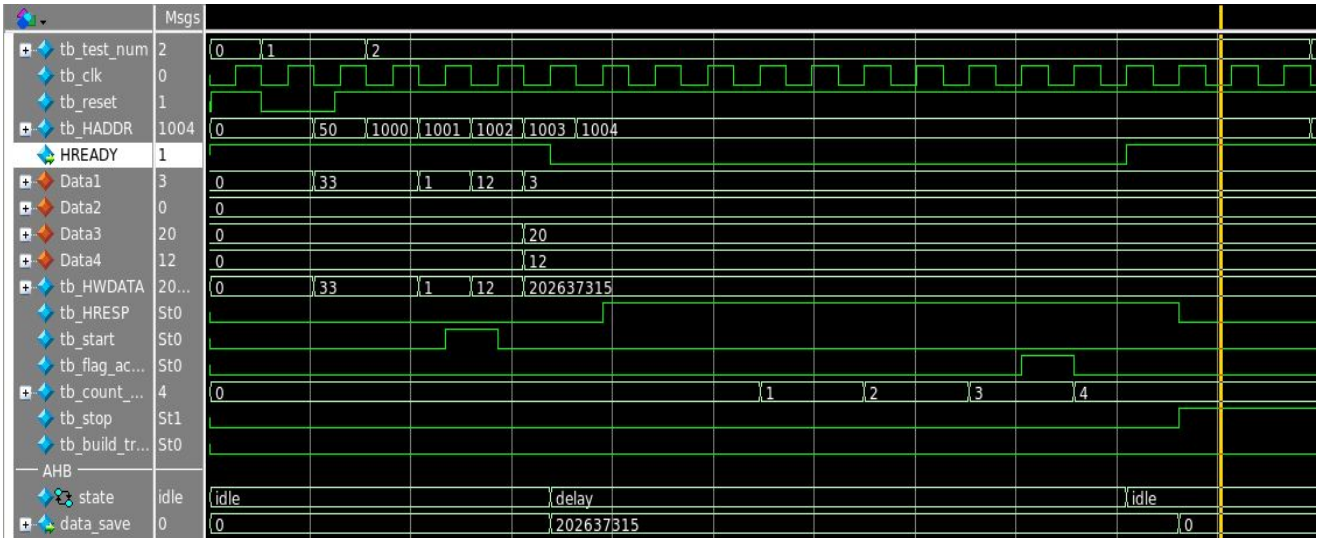
Results Details

Demo Tests

AMBA AHB-Lite Protocol Interfacing (Top Level)

- Shown in Demo: Yes
- DSSC Proved: 3
- Highest Level Design Module involved:
    - Total Design/Chip
- Test bench Expectations/Requirements
    - Have test vector of samples of each type of bus transaction and correct chip response information.
    - Chip response should be verified offline by checking against standards
- No external or premade references are needed
- No pre/post processing is needed
- Main Verification Test Steps:
    1. Emulate bus transaction sample from test vector.
    2. Check chip response against the correct response values in the test vector.
    3. Repeat steps 1-2 for all entries in test vector.

Results:



AHB Slave bus address is keeping changing in every clock cycle. When HREADY is low, the current state goes into delay and HRESP becomes high. After read buffer sends a signal to ahb, HREADY will be high again and HRESP backs to low.

AHB behavior looks good and meet our DSSC 3 slave criteria.

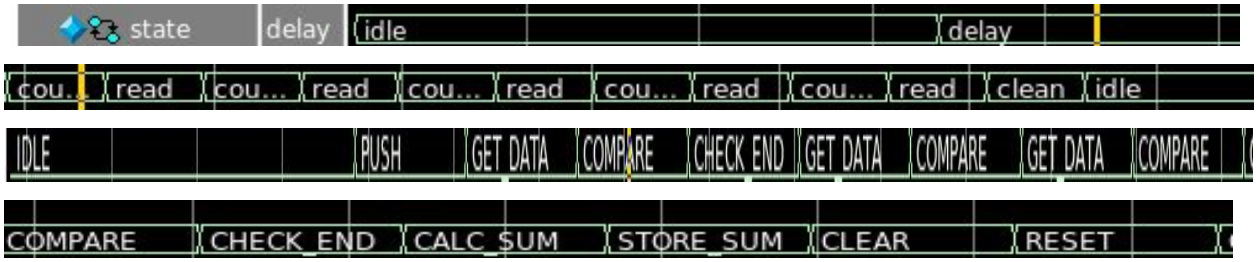
Backup and Sub-Module Tests

Correctness of Controller (Controller Block)

- Shown in Demo: Only if can't show using top level
- DSSC Proved: 1,3
- Highest Level of Design Modules involved:
    - Controller Block
- Test Bench Expectations/Requirements:
    - Every state is reachable
    - Simply implement all necessary signals to test bench
- No pre/post processing is needed
- Main Verification Test Steps:
    - 1. Check if the default state is at IDLE. Every time after reset the demo always goes back to idle state.
    - 2. If an expected input signal is implemented, the controller should move to the next state after one clock cycle
    - 3. If an unexpected signal is implemented, controller should keep its current state and output should not change either.\

Results:

States as time goes by...

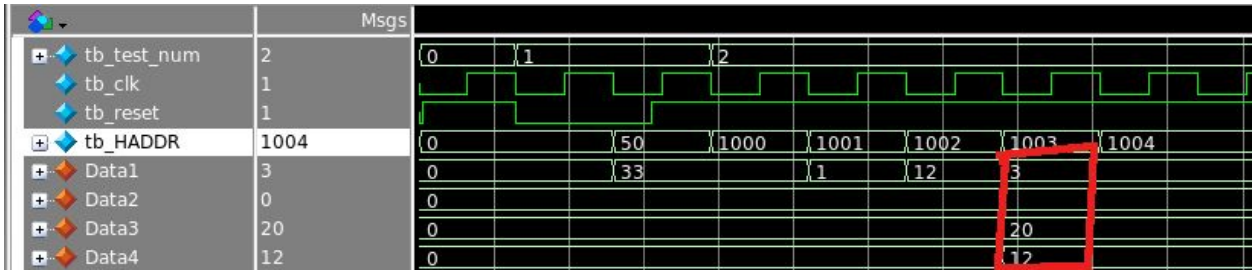


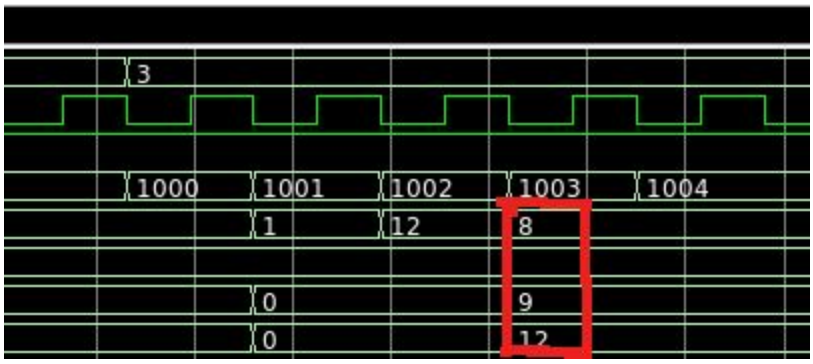
At first the controller is at AHB. When it goes to read buffer, it swings from counter and reading. After 4 cycles it will go back to AHB if it doesn't finish reading all data, otherwise it will step into tree construct. It goes from push data to array, get data when frequency is not zero, compared temporary minimum frequencies and check ending. If ending is reached the sum will be calculated and it will store in the array. After all process, temporary minimum and chars will be reset.

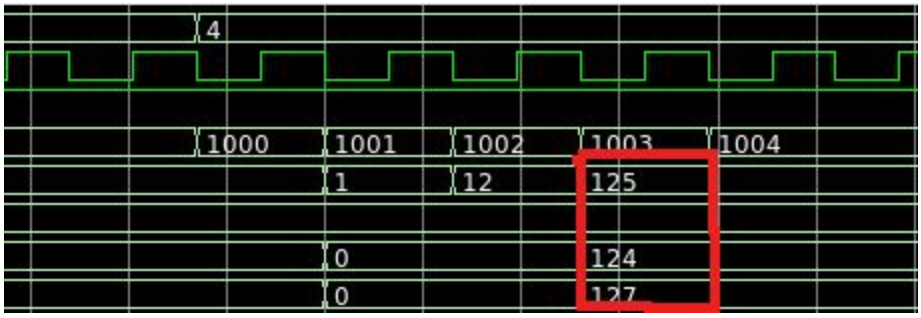
Correctness of Read Buffer (Read Buffer Block)

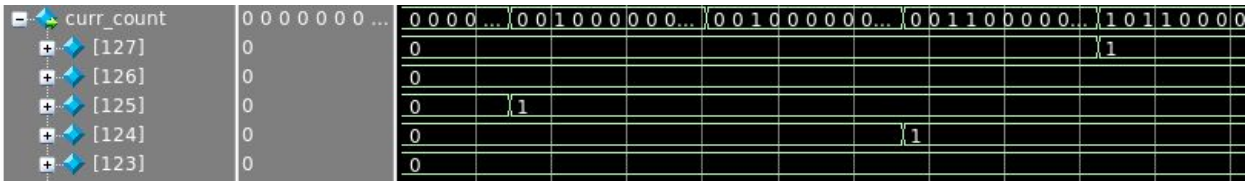
- Shown in Demo: Only if can't show using top level
- DSSC Proved: 1,4
- Highest Level of Design Modules involved:
    - Read Buffer Block
- Test bench Expectations/Requirements:
    - Simply input data demos to test bench and make sure all data will be counted correctly
- No pre/post processing is needed
- Main Verification Test Steps:
    - 1. If count_enable is not asserted, read buffer counters should maintain their value.
    - 2. When valid data is input, it will be separated successfully and after one clock cycle, the correct frequency data will be recorded.
    - 3. When new data comes in, old frequency counts won't be replaced.

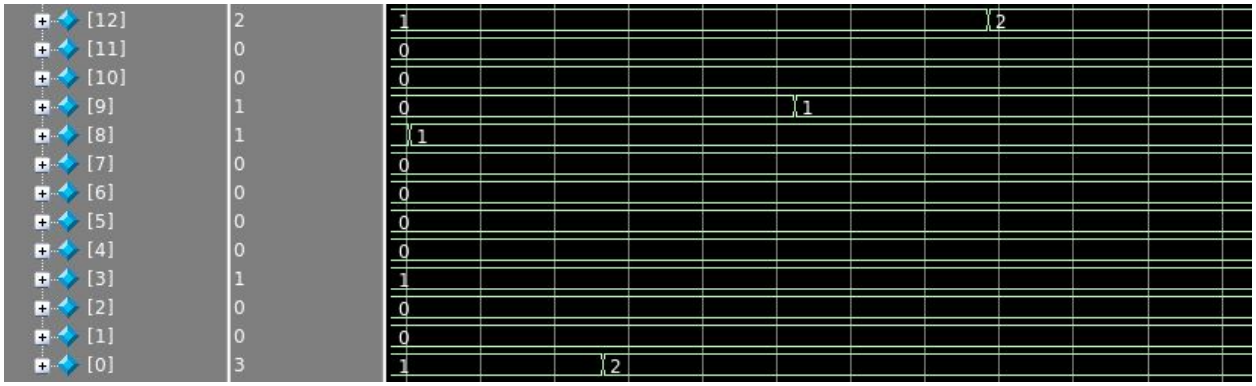
Results:
Read 4 bytes every time











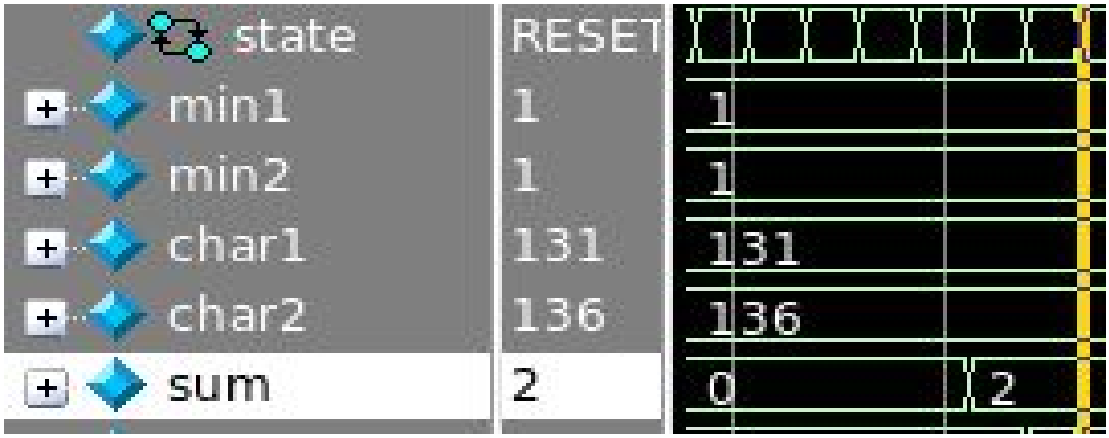
curr_count(current counts)

From AHB Interface, Read buffer block first got “3 0 20 12” “8 80 9 12” and “124 125 0 127”. Each number represents a type of character. From the curr_count, characters at 127, 125, 124, 8, 9,3 are 1s, characters at 12 and 0 are 2. The read buffer’s behavior looks good.

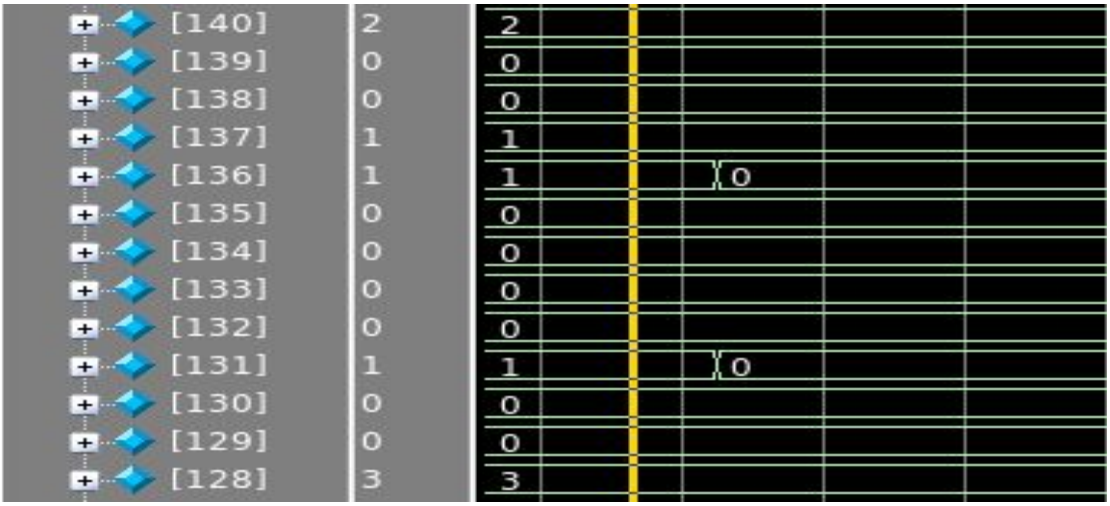
Correctness of Tree Construct (Frequency Min Block)

- Shown in Demo: Only if can’t show using top level
- DSSC Proved: 2
- Highest Level of Design Modules involved:
    - Tree Construct Block
- Test bench Expectations/Requirements:
    - Simply use loop to input different frequencies array
    - Lowest frequencies are chosen correctly every time
    - Frequencies are combined correctly
    - Node is constructed successfully
- No pre/post processing is needed
- Main Verification Test Steps:
    - Check if two lowest frequencies are chosen correctly
    - Check if frequencies are combined correctly
    - Check if node is constructed successfully.

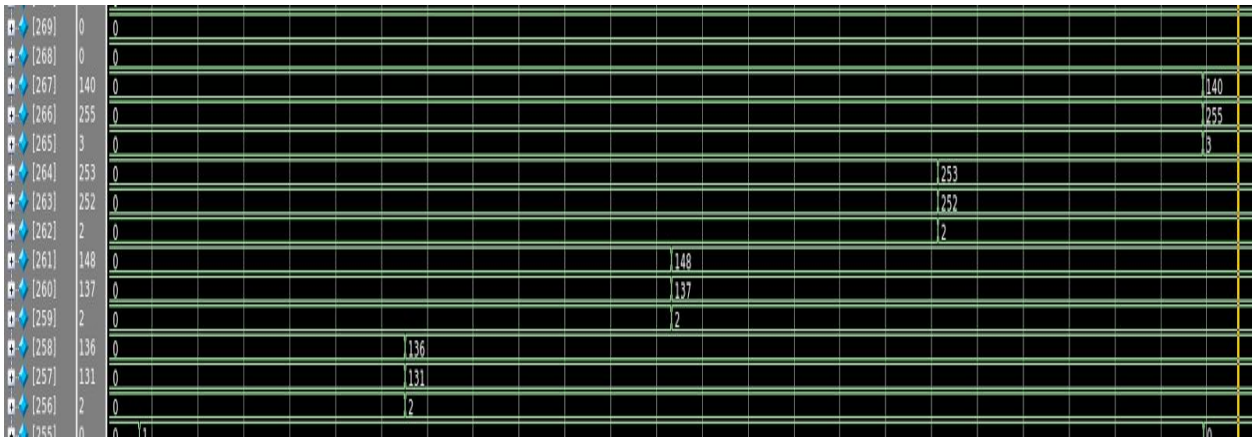
Results:



Showing the two lowest frequencies and their corresponding characters



Lowest frequencies are clear after chosen



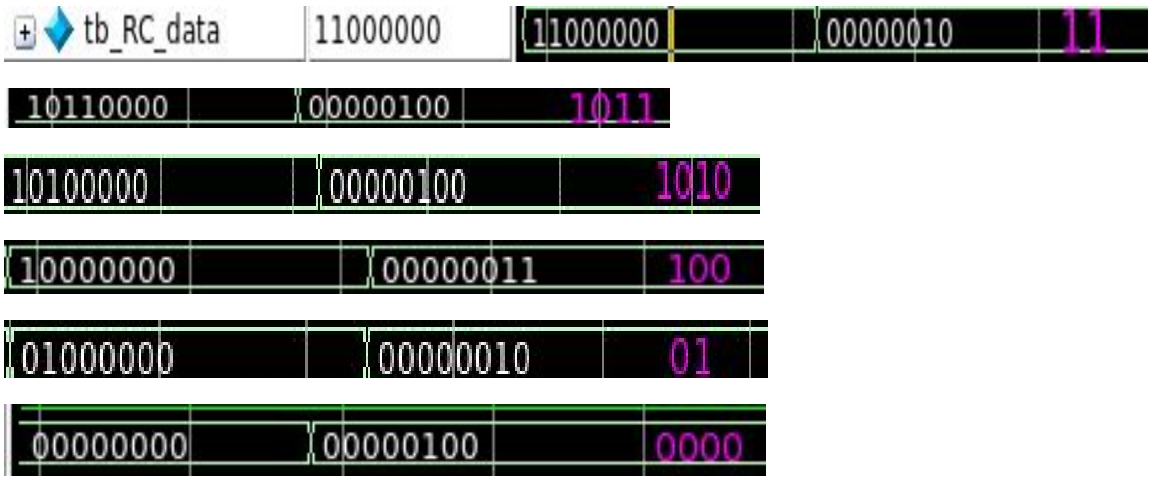
Nodes and leaves address are constructed successfully

From the waveform, you can see that at first always the lowest frequencies nodes or characters are chosen. Variables mins are used to save minimum frequencies and variable chars are used to save its character or address. Variable sum is the combining frequency of that two lowest ones. Once lowest frequencies are found successfully(two 1s), combining frequencies 2 is written into array[256] and char1 char2 are written into array[257],array[258]. Those three elements construct as a node. Once the node is written successfully, clear the min frequencies. That is why in waveform array[131], array[136] becomes 0; The last picture is about the nodes and characters that are constructed. For a node, the first two bytes are used to save frequency, the next two bytes are used to save left node address and right node address.

Correctness of Tree Encode (Tree Encode Block)

- Shown in Demo: Only if can't show using top level
- DSSC Proved: 2
- Highest Level of Design Modules involved:
  - Tree Encode Block
- Test bench Expectations/Requirements:
  - Simply use loop to input different characters with different frequencies to test bench
- No pre/post processing is needed
- Main Verification Test Steps:
  - Use Loop to input different frequencies
  - Check if the combined frequencies equals to the sum of two individual inputs
  - Check if the output written back to SRAM successfully.

Results:



Input: A 7 B 3 C 2 D 6 E 4 F 10

The output encode is showing above.

The wave shows the result when I input some characters with different frequencies.

Results from Java:

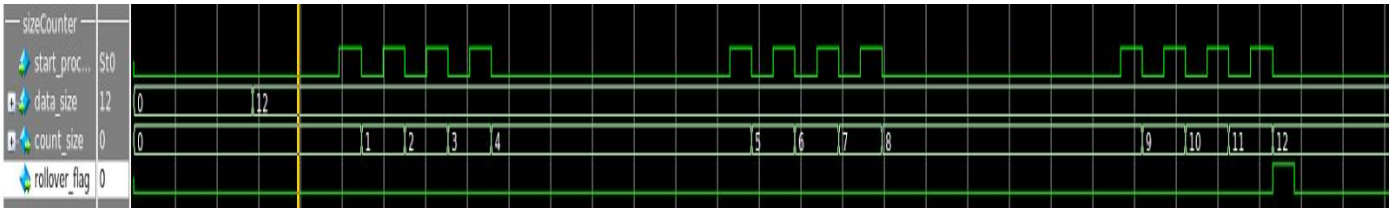
```
}

public static void main(String[] args) {
    String test = "CCBBBEEEEDDDDDDAAAAAAAFFFFFFFFFF";

    // we will assume that all our characters will have
    // code less than 256, for simplicity
    int[] charFreqs = new int[256];
    // read each character and record the frequencies
    for (char c : test.toCharArray())
        charFreqs[c]++;
```

```
fmanCode
/Library/Java/JavaVirtualMachines/jdk1.8.0_65.jdk/Contents/Home/bin/java ...
SYMBOL  WEIGHT  HUFFMAN CODE
D   6   00
A   7   01
E   4   100
C   2   1010
B   3   1011
F   10  11
```

When I input same characters with same frequencies, we are able to get the exact same encodes of each character with Huffman Module in software. DSSC 2 is reaching.

Correctness of Size Counter (Size Counter Block)

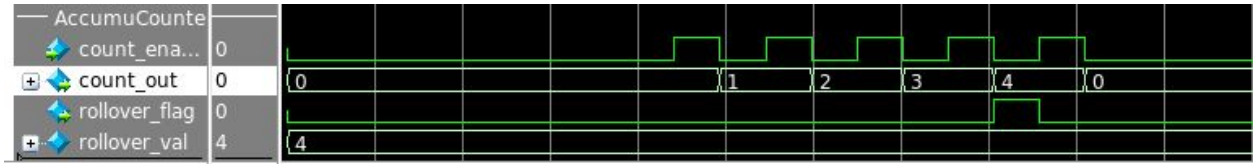
- Shown in Demo: Only if can't show using top level
- DSSC Proved: 4
- Highest Level of Design Modules involved:
    - Size Counter Block
- Test bench Expectations/Requirements:
    - Insert count enable, clear randomly
- No pre/post processing is needed
- Main Verification Test Steps:
    - 1. Check if counter starts to count if count_enable inserted
    - 2. Check if rollover_flag is high if reach out rollover value
    - 3. Check count out is 0 when clear inserted



If the data size is 12, the sizecounter will keep counting until 12 and rolloverflag will be high. This signal will trick Tree construct.

Correctness of Accumulator Counter (Accumulator Counter Block)

- Shown in Demo: Only if can't show using top level
- DSSC Proved: 4
- Highest Level of Design Modules involved:
    - Accumulator Counter Block
- Test bench Expectations/Requirements:
    - Insert count enable, clear randomly
- No pre/post processing is needed
- Main Verification Test Steps:
    - 1. Check if counter starts to count if count_enable inserted
    - 2. Check if rollover_flag is high if reach out rollover value
    - 3. Check count out is 0 when clear inserted

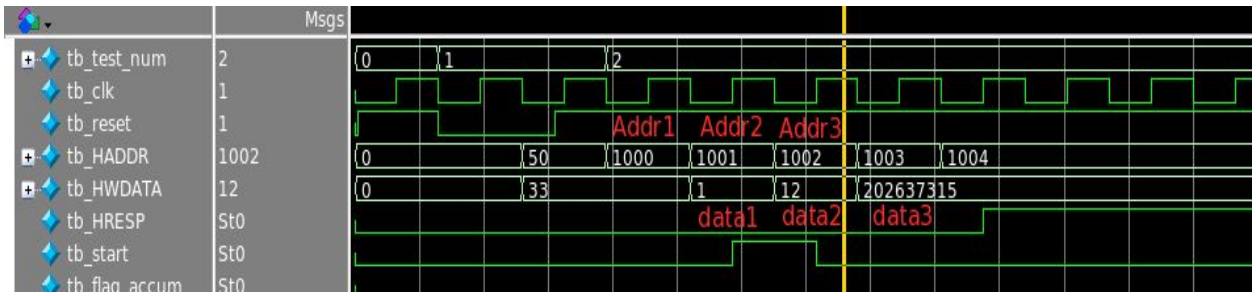


The accumu counter will always count to 4 since I am doing 32 bits(4 bytes) reading. When the counter increases, a new byte will go into the read buffer and is counted.

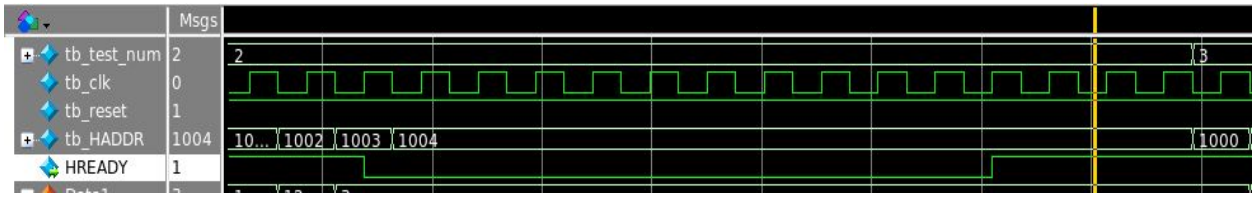
AMBA AHB-Lite Protocol Interfacing (AMBA AHB-Lite Bus Interface Block)

- Shown in Demo: Only if can't show using top level
- DSSC Proved: 3
- Highest Level Design Module involved:
    - AMBA AHB-Lite Bus Interface Block
- Test bench Expectations/Requirements
    - Have test vector of samples of each type of bus transaction and correct chip response information.
    - Chip response should be verified offline by checking against standards
- No external or premade references are needed
- No pre/post processing is needed
- Main Verification Test Steps:
    4. Emulate bus transaction sample from test vector.
    5. Check chip response against the correct response values in the test vector.
    6. Repeat steps 1-2 for all entries in test vector.

Results:



We simulate AHB bus behavior as above. First the bus will be noticed when the address is 1000. After 1 clock cycle, its corresponding data1 will come out. Address of the bus is keep changing as long as HREADY is high. The data is always one clock cycle behind the address.



HREADY signal controls the change of address.

Appendix

Account and directory where all of the files are located: mg105/ASIC_Design/Project

- Design Verilog Code Modules
  - **Top level structural Verilog code**: source/Huffman_Tree.sv
  - **Sub level structural verilog code:** source/AHB.sv
  - **Sub level structural verilog code:** source/ahb.sv
  - **Sub level structural verilog code:** source/ahb_resp.sv
  - **Sub level structural verilog code:** source/READ_BUFFER.sv
  - **Sub level structural verilog code:** source/read_buffer.sv
  - **Sub level structural verilog code:** source/control_read.sv
  - **Sub level structural verilog code:** source/flex_counter.sv
  - **Sub level structural verilog code:** source/sizeCounter.sv
  - **Sub level structural verilog code:** source/accumu_counter.sv
  - **Sub level structural verilog code:** source/tree_construct.sv

    - **Sub level structural verilog code:** source/Huffman_Read.sv
    - **Sub level structural verilog code:** source/tree_encode.sv
- Test Benches
    - **Simple top level test**: source/tb_Huffman_Tree.sv
    - **Simple sub level test**: source/tb_AHB.sv
    - **Simple sub level test**: source/tb_ahb.sv
    - **Simple sub level test**: source/tb_ahb_resp.sv
    - **Simple sub level test**: source/tb_accumu_counter.sv
    - **Simple sub level test**: source/tb_flex_counter.sv
    - **Simple sub level test**: source/tb_read_buffer.sv
    - **Simple sub level test**: source/tb_control_read.sv
    - **Simple sub level test**: source/tb_READ_BUFFER.sv
    - **Simple sub level test**: source/tb_tree_construct.sv
    - **Simple sub level test**: source/tb_Huffman_Read.sv
    - **Simple sub level test**: source/tb_tree_encode.sv
- Mapped Verilog code
    - **Simple top mapped**: /Huffman_Tree.v
    - **Simple sub mapped**: /accumu_counter.v
    - **Simple sub mapped**: /AHB.v
    - **Simple sub mapped**: /read_buffer.v
    - **Simple sub mapped**: /tree_construct.v
    - **Simple sub mapped**: /tree_encode.v
- Diagrams
    - **Simple top area**: /docs/AHB_area.png
    - **Simple sub area**: /docs/counter_area.png
    - **Simple sub area**: /docs/humman_area.png
    - **Simple sub area**: /docs/read_buffer.png
    - **Simple sub RTL**: /docs/Rollover.png
- Data Sheets
    - **Area Budget**: /docs/Area_Budget.pdf
    - **Time Budget**: /docs/Time_Budget.ods
- Final Presentation
    - **Presentation**: /docs/Final_Presentation.pptx

- Layout
    - **Layout general**: ece337/Project/timingReports
    - **Layout initial:** ece337/Project/init_pins.tcl
    - **Layout pins**: ece337/Project/innovus_pins.io
    - **Layout graph**: /docs/layout.png
    - **Layout graph zoom**: /docs/layout2.png