NCTU-EE IC LAB – Spring 2025

Lab01 Exercise

Design: Huffman Coding (HF)

Data Preparation

1. Extract files from TA's directory:
   % tar -xvf ~iclabTA01/Lab01.tar

Design Description and Examples

Huffman Coding is a widely used lossless data compression algorithm that assigns variable-length binary codes to input symbols based on their frequencies. The most frequently occurring symbols receive shorter codes, while less frequent symbols receive longer codes, ensuring optimal compression efficiency.

1. Algorithm Description(Rule)

Huffman Coding operates as follows:

1. **Frequency Calculation**
   1. Count the frequency of each symbol in the input data.
2. **Priority Queue Construction**
   1. A min-heap priority queue is used, where symbols with smaller frequencies have higher priority and are dequeued first
3. **Huffman Tree Construction**
   1. Extract the two symbols (nodes) with the smallest frequencies from the queue
   2. Create a new parent node with a frequency equal to the sum of these two nodes
   3. Insert the new parent node back into the queue.
   4. Repeat until only one node (the root) remains.
4. **Code Assignment**
   1. Assign binary codes to each symbol by traversing the tree:

      Left branch (smaller) → Append 0

      Right branch(bigger) → Append 1
   2. The final code is prefix-free, ensuring no code is a prefix of another.
5. **If two symbols have the same frequency, follow these tie-breaking rules**
   1. Newly merged nodes take precedence over original symbols and should be placed in the 0 branch.
   2. If both are original symbols (a–e), they should be ordered alphabetically from 'a' to 'e',

with the higher-priority symbol placed in the 0 branch.

2. Description of this lab

- **Input (Rule 1):**

You will receive a 25-bit variable named symbol_freq, which contains the frequencies of five different inputs (in order: a, b, c, d, e). Each frequency is represented using 5 bits, starting from the least significant bit (LSB) and represented as an unsigned value.

	a freq	b freq	c freq	d freq	e freq
symobl_freq	00100	00010	01010	00100	01011

In this example the frequency of {a,b,c,d,e} is {4,2,10,4,11}.

- **Construct the Huffman Tree (Rule 2,3):**

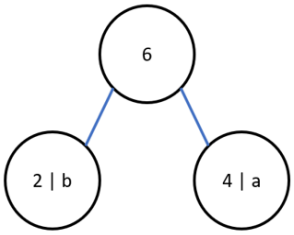
Step 1 : Sort the frequency in ascending order

Sorted Freq: {b,a,d,c,e} is {2,4,4,10,11} (follow rule 5.1 for same frequency)

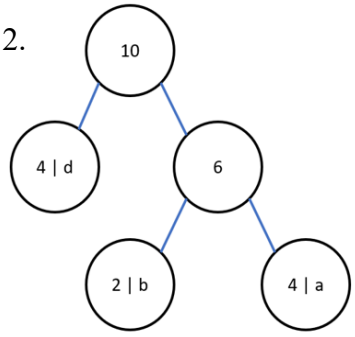
Step 2 : Combine the two nodes with the smallest frequencies and replace the original nodes until only one node remains in the queue.

1. 2 + 4 = 6 → {6,4,10,11}
2. 4 + 6 = 10 → {10,10,11} (follow rule 5.2 for same frequency)
3. 10 + 10 = 20 → {20,11}
4. 20 + 11 = 31 → {31}

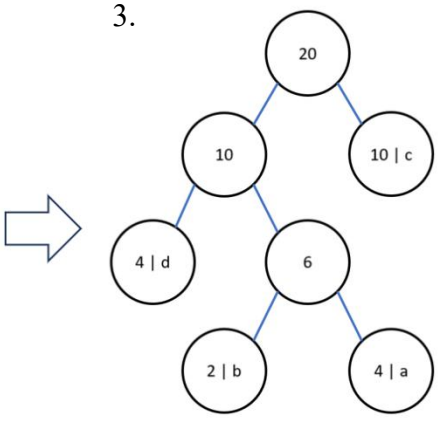
1.



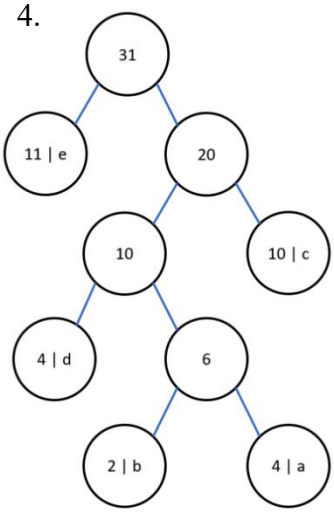
2.



3.



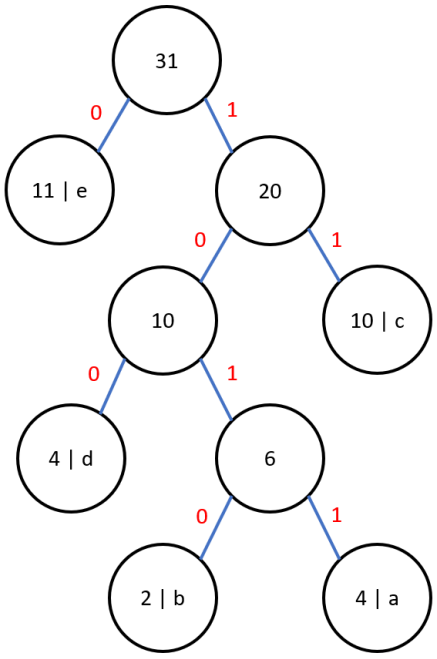
4.



➢ **Encode (Rule 4,5) :**

Traverse the Huffman tree from the root downward. Taking a as an example, the traversal path follows the nodes: 31 → 20 → 10 → 6 → 4 | a. Since the corresponding binary path is 1011, the encoded value for a is 4'b1011.

a: 4’b1011. b:4’b1010 c:2’b11 d:3’b100 e:1’b0



➢ **Output**

Expand the final five encoded values to 4 bits each, then concatenate them in the order of a to e to form a 20-bit output out_encoded , and starting from the least significant bit (LSB)

	a encoded	b encoded	c encoded	d encoded	e encoded
out_encoded	1011	1010	0011	0100	0000

Inputs

Input Signals	Bit Width	Description
symbol_freq	25	Represents the frequencies of five different symbols: a, b, c, d, and e. Each symbol's frequency is encoded using 5 bits

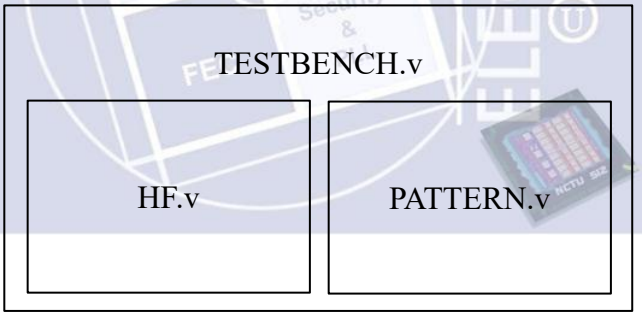
Outputs

Output Signals	Bit Width	Description
encoded_output	20	Represents the 20-bit Huffman-encoded output for symbols **a–e**, where each symbol's 4-bit code is concatenated in the order **a to e.**

Specifications

1. Top module name : HF(File name: HF.v)
2. After synthesis, check the "HF.area" and "HF.timing" in the folder "Report". **The area report is valid only when the slack in the end of "HF.timing" is "MET". if it report "violate" means you will fail.**
3. The synthesis result **cannot** contain any **latch**.
   Note: You can check if there is a latch by searching the keyword "**Latch**" in 02_SYN/**syn.log**
4. Lookup table method is prohibited. (Such as directly outputting the answer based on the pat index.)
5. IP is prohibited in this lab.

Block Diagram



Grading Policy

The performance is determined by the area of your design. The less area your design has, the higher grade you get. Try to reach better performance by thinking your architecture before coding.

Function Validity : 70%

Performance: Area: 30%

If you fail Lab01 at first demo, and pass at second demo, you will get 30% off of your original score. Get no score if you fail both first and second demo. Note that you will get 0 score if you are found plagiarism at your code.

Note

1. Tar all your design by run the command **Lab01/09_SUBMIT/00_tar**
2. Submit your design through **Lab01/09_SUBMIT/01_submit**
   a. 1st_demo deadline: 2025/03/03(Mon.) 12:00:00
   b. 2nd_demo deadline: 2025/03/05(Wed.) 12:00:00
3. If your file violates the naming rule, you will lose 5 points.
4. Don't use any wire/reg/submodule/parameter name called *error*, *Congratulations*, *latch* or *FAIL* otherwise you will fail the lab. Note: * means any char in front of or behind the word. e.g: error_note is forbidden.
5. Do not modify the syn.tcl.

Be careful about all details!

Template folders and reference commands:

In demo, the reference commands is:

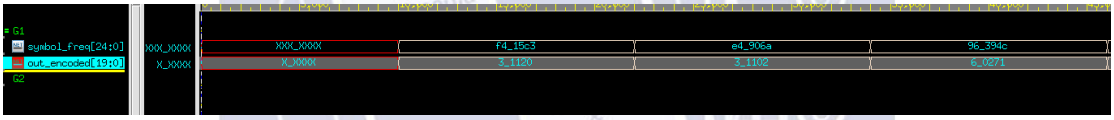
1. 01_RTL (RTL simulation):
   ./01_run_vcs_rtl
   ./05_nWave
2. 02_SYN/ (Synthesis):
   ./01_run_dc_shell
   ./08_check
   (Check **latch** by searching the keyword "**Latch**" in 02_SYN/**syn.log**)
   (Check the design's timing in /Report/ HF.timing)
   (Check the design's area in /Report/ HF.area)
3. 03_GATE/ (Gate-level simulation):
   ./01_run_vcs_gate
   ./05_nWave
4. 09_SUBMIT/ (submit your files):
   ./00_tar
   ./01_submit
   ./02_check

You can key in **./09_clean_up** to clear all log files and dump files in each folder

```
|-- 00_TESTBED
|   |-- PATTERN.v
|   |-- TESTBED.v
|   |-- Test_data_gen_ref.py
|   |-- filelist.f
|   |-- input.txt
|   |-- makefile
|   `-- output.txt
|-- 01_RTL
|   |-- 01_run_vcs_rtl
|   |-- 02_irun_rtl
|   |-- 03_xrun_rtl
|   |-- 04_verdi
|   |-- 05_nWave
|   |-- 08_check
|   |-- 09_clean_up
|   |-- PATTERN.v -> ../00_TESTBED/PATTERN.v
|   |-- SMC.v
|   |-- TESTBED.v -> ../00_TESTBED/TESTBED.v
|   |-- filelist.f -> ../00_TESTBED/filelist.f
|   `-- makefile -> ../00_TESTBED/makefile
|-- 02_SYN
|   |-- 01_run_dc_shell
|   |-- 02_run_design_vision
|   |-- 03_read_dv_rtl
|   |-- 04_read_ddc
|   |-- 08_check
|   |-- 09_clean_up
|   |-- Netlist
|   |-- Report
|   |-- makefile -> ../00_TESTBED/makefile
|   `-- syn.tcl
|-- 03_GATE
|   |-- 01_run_vcs_gate
|   |-- 02_irun_gate
|   |-- 03_xrun_gate
|   |-- 04_verdi
|   |-- 05_nWave
|   |-- 08_check
|   |-- 09_clean_up
|   |-- PATTERN.v -> ../00_TESTBED/PATTERN.v
|   |-- SMC_SYN.sdf -> ../02_SYN/Netlist/SMC_SYN.sdf
|   |-- SMC_SYN.v -> ../02_SYN/Netlist/SMC_SYN.v
|   |-- TESTBED.v -> ../00_TESTBED/TESTBED.v
|   |-- filelist.f -> ../00_TESTBED/filelist.f
|   `-- makefile -> ../00_TESTBED/makefile
`-- 09_SUBMIT
    |-- 00_tar -> /RAID2/COURSE/BackUp/2023_Spring/iclab/iclabta01/AutoSubmitScript/Lab01/09_SUBMIT/00_tar
    |-- 01_submit -> /RAID2/COURSE/BackUp/2023_Spring/iclab/iclabta01/AutoSubmitScript/Lab01/09_SUBMIT/01_submit
    `-- 02_check -> /RAID2/COURSE/BackUp/2023_Spring/iclab/iclabta01/AutoSubmitScript/Lab01/09_SUBMIT/02_check
```

Example Waveform

Input and output signal:



Debug file (In 00_TESTBED folder)

Input.txt debug.txt output.txt

```
1000

5 10 19 1 23

5 21 13 9 3

5 6 17 23 23

19 11 22 18 26
```

```
1, a:1001 b:101  c:11   d:1000 e:0
2, a:1101 b:0    c:10   d:111  e:1100
3, a:000  b:001  c:01   d:10   e:11
4, a:00   b:110  c:01   d:111  e:10
5, a:00   b:110  c:10   d:01   e:111
```

```
611200

852604

4387

24946

25111
```

Hint

Hint1: Try to use **behavior modeling description** instead of gate level description.

Hint2: Try to use **submodule** rather than copy and paste to simply your design. (not necessary in this lab)

```
// ------------------------------------------------
// Example for using submodule
// BBQ bbq0(.meat(meat_0), .vagetable(vagetable_0), .water(water_0),.cost(cost[0]));
// ------------------------------------------------
```

Hint3: Try to think if there is any possible **hardware** that can be **shared** with different mode operation. You can use command dc_shell-gui to examine your design.(not necessary in this lab)

Hint4: Pattern provided by TA will cover only some simple cases, you can try to write your own input / output file by yourself. Here is the format how TA will read in PATTERN:

```
//================================================================
// Hint
//================================================================
// if you want to use c++/python to generate test data, here is
// a sample format for you. You can change for your convinience.
/* input.txt format
1. [PATTERN_NUM]

repeat(PATTERN_NUM)
    1. [symbol_freq_0] [symbol_freq_1] ... [symbol_freq_4}
*/

/* output.txt format
    1. [out_encoded]
*/
```

You can check input.txt and PATTERN.v in 00_TESTBED as a reference, and choose to write either c++/python or Verilog code for generating corner cases.