Homework #4
Assigned: February 24, 2020
Due: March 1, 2019, 11:59pm
This assignment is to be completed individually. Submit all files in a single tar-file.

This homework is intended to provide experience with logic design and synthesis. Read and follow the instructions very carefully and include adequate comments in your model file as well as testbench.

Problem 1: FSM synthesis (20 points)

In this problem, you are asked to synthesize the Traffic Light Controller you designed in HW#3 using the gscl45nm standard cell library. Please refer to the script provided on the Blackboard to constrain the sequential circuit. As you are going to synthesize your own RTL design, read through the Design Compiler output carefully to ensure that no unwanted latches or flip-flops are interpreted by Design Compiler from your RTL code. Answer the following question:

1. Check the timing report. Did the synthesis result meet the timing constraints? What is the critical path and slack?
2. Please compare the area requirement of different state encoding schemes in the Traffic Light Controller. Specifically, you need to synthesize two controller designs: one with binary encoding and another with 1-hot encoding of the states. Report the areas of the two designs with the same timing constraint that is specified in the script.

Table I – Example of Binary vs. 1-Hot State Encoding for a 8-state FSM

State	Binary code	1-hot code
S0	000	00000001
S1	001	00000010
S2	010	00000100
S3	011	00001000
S4	100	00010000
S5	101	00100000
S6	110	01000000
S7	111	10000000

3. Run post-synthesis simulation to verify the logic function of your design using the same benchmark you have used in HW#3. Does the gate-level netlist pass the test? If it fails, can you identify the reason? Please make sure your design functions correctly before submitting (at least when a slow clock is used for simulation).

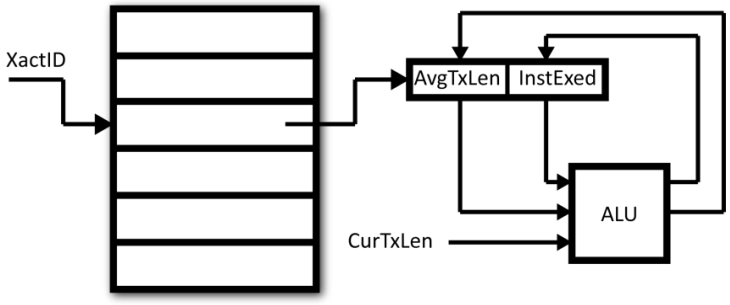
Submission:

1. RTL Verilog code: traffic_control_binary.syn.v, traffic_control_1hot.syn.v, tb_traffic_control.v
2. Synthesis reports: traffic_control.timing, traffic_control.area, traffic_control.check_design, traffic_control.lec (LEC transcript)

Problem 2: Pipelined ALU Design Exploration (30 points)

As the computing paradigm has fundamentally shifted to multi/many-core chip multiprocessors, such multiprocessors enjoy an increasing popularity in a wide range of computing platforms from mobile devices to high performance servers. However, one of the grand challenges is to write parallel programs to harness the large number of cores. Transactional Memory (TM) has been proposed as an alternative to locks to simplify parallel programming. Essentially, a *transaction* is an atomic code block that either executes completely or has no effect. Programmers only need to specify the boundaries of transactions. Then, either the software runtime or hardware ensures that concurrent accesses from transactions to the shared memory are synchronized correctly and deadlock-free. In particular, Hardware Transactional Memory (HTM) provides hardware mechanisms in support of high performance transactional execution. The length of a transaction in terms of cycle counts is a characteristic of great importance to guide the transaction execution. A hardware structure has been proposed to monitor the transaction length during the execution. An architectural overview of the structure is given as below.

Figure 1: Architectural overview of the hardware structure to track transaction length.



A transaction will be executed many times (~100 to ~100,000) in an application. The storage structure tracks the average transaction length (`AvgTxLen`) and the number of times that particular transaction has committed (`InstExed`). The length of the most recent execution of that transaction is given in `CurTxLen`. The average transaction length is updated using the following formula.

$$^{t}AvgTxLen_{new} = \frac{AvgTxLen_{old} \times InstExed + CurTxLen}{InstExed+1}$$

The architectural design team asks you to implement and characterize a gate-level ALU design to perform the computation specified by the given formula. As the architecture l design team wants the design to achieve high throughput, the ALU needs to be pipelined. They also provide you with a pipeline design for your reference (see Figure 2).

In this problem, you need to submit a gate-level ALU design to meet the architecture l team's requirement. First of all, you need to design the ALU at RTL-level. Then, you will synthesize the design using the NCSU 45nm library to produce a gate-level netlist. Finally you will perform **static timing analysis** to see if the design meets the timing constraints. Here is the module to be implemented:

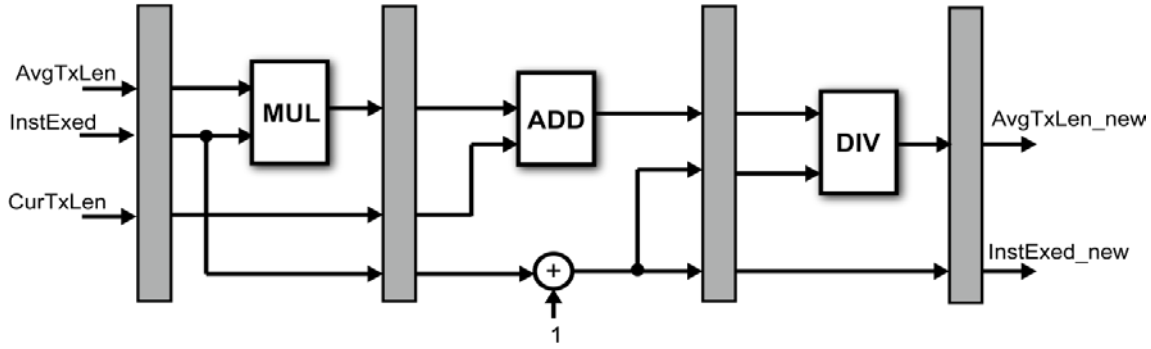


Figure 2: Pipelined ALU reference design.

```
module TM_ALU ( clk, reset,
AvgTxLen,                  // 8-bit input
InstExed,                  // 8-bit input
CurTxLen,                  // 8-bit input
AvgTxLen_new,              // 8-bit output
InstExed_new,              // 8-bit output );
```

Tips:

1. Aim for maximum clock frequency.
2. Test your design with reasonable test cases.
3. You can either use your own design of the adder, multiplier and divider or use the Synopsys Design Ware.
4. Please use the referenced pipeline design in Figure 2.

Please answer the following question.

Can you come up with an alternative pipeline design of the ALU which could provide either performance or area benefit? Provide the schematic of the proposed design along with one paragraph to justify your design choice.

Submission:

1. Behavioral RTL code, synthesized netlist, and testbench: **TM_ALU.v**, **TM_ALU.syn.v** and **tb_TM_ALU.v**.
2. Synthesis report: **TM_ALU.area, TM_ALU.timing, TM_ALU.check_design, TM_ALU.lec (LEC transcipt)**.
3. Please report the maximum clock frequency your design can achieve along with the estimated

power consumption (use report_power in the synthesis script, report sum of dynamic and leakage power). Put the two numbers into a txt file with following format. The txt file name convention: **<last_name>_<first_name>_hw4p2.txt**. Also include the results in the pdf file with answers to other problems.

4. **FREQ: xxx MHz POWER: x.xxx W**

Problem 3: Implementing SEC-DED for the memory system (40 points)

Soft-errors due to neutron and alpha particle strikes are becoming a major concern in microprocessor design. As the technology keeps scaling, transistor count will increase exponentially and the charge stored in a memory cell reduces greatly. Thus microprocessors are increasingly susceptible to single and multiple bit soft-errors. In particular, the on-chip cache is critical to a microprocessor's reliability. Therefore, it is extremely important to detect and recover from those errors to avoid program failure.

To combat soft errors, modern microprocessors protect their on-chip cache using an Error Correction Code (ECC) scheme. In particular, the Hamming code, which was invented by Richard Hamming in 1950, is widely used due to its minimum redundancy. In this problem, you are asked to implement the encoder and decoder modules for a given *extended Hamming code* using behavior RTL Verilog HDL, design a testbench to test your design, synthesize the encoder and decoder modules, and conduct post-synthesis simulation to verify the gate-level design.

The Hamming (7, 4) is a linear error-correcting code that encodes 4 bits of data into 7 bits by adding 3 parity bits. The Hamming codes can be computed using linear algebra through matrix multiplication. Two Hamming matrices can be defined: the code generator matrix G and the parity check matrix H. The two matrices are given below.

Generator matrix of Hamming (7, 4):

$$\mathbf{G} := \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Parity-check matrix of Hamming (7, 4):

$$\mathbf{H} := \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

The encoder takes a 4-bit data (P) as input and generates a 7-bit code word (X):

$$X = \begin{pmatrix} x0 \\ x1 \\ x2 \\ x3 \\ x4 \\ x5 \\ x6 \end{pmatrix} = G \times \begin{pmatrix} p0 \\ p1 \\ p2 \\ p3 \end{pmatrix}$$

The decoder takes the 7-bit code word (X) as input and generates a 3-bit *syndrome vector* (Z) which indicates whether an error has occurred, and if so, which code word bit is corrupted. A zero syndrome vector indicates error-free.

$$Z = \begin{pmatrix} s0 \\ s1 \\ s2 \end{pmatrix} = H \times \begin{pmatrix} x0 \\ x1 \\ x2 \\ x3 \\ x4 \\ x5 \\ x6 \end{pmatrix}$$

Here is an example of Hamming (7, 4) code. The encoder takes the product of G and P to get the code word X:

$$X = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

The code word is stored in the cache. Suppose one bit is flipped while the code word is residing in the cache due to a soft-error. So the code word becomes: (1, 1, 1, 0, 0, 1, 1). Later, when the code word is read from the cache, the decoder takes the code word to generate the syndrome vector Z:

The syndrome vector gives a 3-bit binary value 1, which indicates X0 is corrupted.

$$Z = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

For many applications, the Hamming (7, 4), which is a single error correcting code, would be considered unsatisfactory. A SEC-DED (Single Error Correction Double Error Detection) code seems safer, and it is the level of correction and detection most often used in computer memories. The Hamming code can be converted to a SEC-DED code by adding one check bit, which is a parity bit (let us assume even parity) on *all the bits* in the SEC code word. This code is called an *extended Hamming code*.

Here are the specific tasks you need to perform to accomplish this problem:

1. Please specify how the decoder can get the original 4-bit data from the SEC codeword.
2. Implement the encoder and decoder modules to provide SEC-DED protection to the 4-bit data.

```
module ham_74_encoder (d, // [3:0] input data word
                       c, // [7:0] output code word );
module ham_74_decoder (c, // [7:0] input codeword
                       qc, // [7:0] corrected code word
                      qd, // [3:0] corrected data word
                      x // 1-bit double error flag );
```

3. Build your testbench as shown in the pipeline figure below. Generate a random number for the data word d[3:0] and two random numbers for the errors **e0[2:0]** and **e1[2:0]** every clock cycle using Verilog system task $random. Connect the MSB of the **4-bit** counter output to the mux which can introduce a double-bit error every 8 cycles. The counter uses the same clock and reset signal as the stage registers. Please remember to reset the circuit at the beginning of your simulation. Print the outputs corrected code word qc[7:0], corrected data word qd[3:0], original data word d[3:0] and the value of the double error flag, using Verilog system task $display, every clock after the reset. Use appropriate (synchronous or asynchronous) reset for the pipeline registers. Run the test for at least 10 cycles after reset.
4. Synthesize your encoder and decoder designs targeting the **gscl45nm** standard cell library. Apply appropriate timing constraints to synthesize your design in the script.
5. Plug in the synthesized gate-level netlist into the above testbench. Compare the results. Are there any differences between the two simulations? If so, what are the differences? And, why did they occur?
6. We will additionally perform Logical equivalence on the designed and synthesized designs. Use **Cadence Conformal** to test the logical equivalence of your synthesized design.

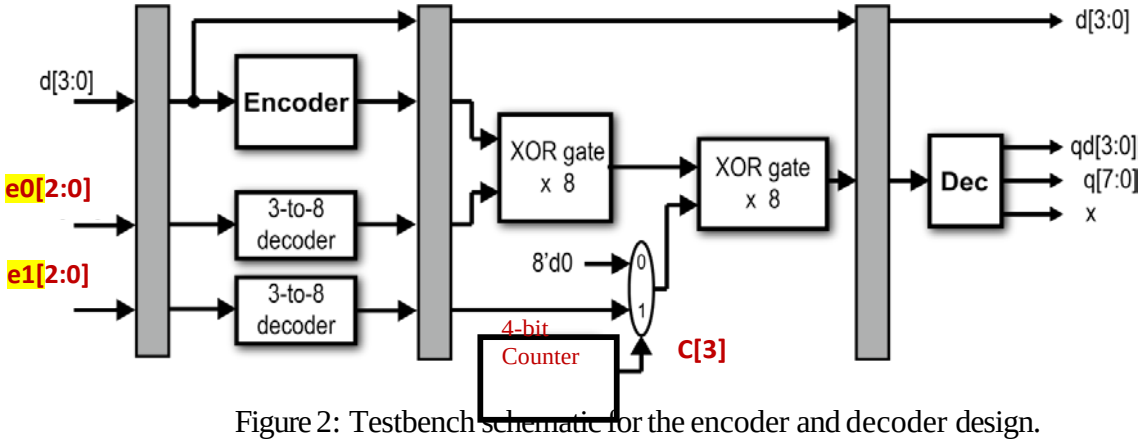


Figure 2: Testbench schematic for the encoder and decoder design.

Submission:

1. Behavioral RTL code: **ham_74_encoder.v** and **ham_74_decoder.v**
2. Testbench file: **tb_ham_74.v**
3. Gate-level netlist: **ham_74_encoder.syn.v** and **ham_74_decoder.syn.v**.
4. Synthesis reports: **ham_74_encoder.timing**, **ham_74_decoder.timing**
   ham_74_encoder.area, **ham_74_decoder.area**
   ham_74_encoder.check_design, **ham_74_decoder.check_design**
   ham_74_encoder.lec, **ham_74_decoder.lec**

Combine answers for all the questions into a single pdf file called HW4_report.pdf and submit along-with remaining files.

Submission Process

Use the provided directory structure for submission. Keep all your design files in **rtl/**, synthesized netlist in **netlist/**, all reports/log/.txt/PDF in **report/**, and all the testbench files for simulation in **sim_tb/**, and for synthesized netlist simulation in **syn_tb/.** "Tar" all the required files for HW#4 and use the upload link of HW4 in BB for electronic submission.