FreeRange
Digital Design Foundation
Modeling

Lab Activity Manual

Table of Contents

Lab Activity Report Guidelines

Lab reports provide a description of your work in the lab. There are two basic forms of information in the lab report: an objective portion and a subjective portion. The objective portion of the lab report documents what you did in the lab (designed stuff, took data points, etc.) while the subjective portion provides your interpretation of the results you obtained during the lab (namely the conclusion).

Each instructor has their own set of rules and expectations regarding lab reports. Generally speaking, lab reports are graded on how well you follow those rules and meet those expectations. The goal of the following comments is to help you generate a quality lab report while minimizing the amount of time you spend writing the report. Please do not hesitate to ask questions. Lab write-ups are not exams so feel free to ask me to look over your report before you submit it.

General Rules:

- Lab reports should be written using a word processor and submitted in hard copy form.
- Each lab group should submit one lab report. Each member of the lab group should write their own conclusion independently from other group members and submit it with the lab report.
- Reports should not have a title page. The lab activity title should be boldly centered on the first page. The course and section number, and the names of group members should also appear on the first page.
- Wording in the lab report should be brief but concise. Longer lab reports rarely correlate to higher quality lab reports. Moreover, using concise wording save you time writing the report and saves someone else time reading it.
- Wording in the lab report should use “good English” and appropriate technical style. Correct spelling, appropriate use of technical terms, and appropriate technical style allows you to create a professional document that highlights your lab activities. If you are new to technical report writing, strongly consider having someone else read over your report before you submit it. There are also many features in MS Word that can help you with the grammar.
- Lab reports should be neat and intelligently organized. Hand-written and hand-drawn items such as circuit diagrams should be neat (use appropriate drawing software if necessary).
- All figures and diagrams should contain captions and/or titles and should be referenced directly from the body of the report (do not say things like “in the figure below”). Plan on using the cross-referencing feature of your word processor (ask someone if you don’t know how to use it).
- Different sections and diagrams in the lab report should be well delineated from each other. Using extra space to in the lab report generally creates a more professional looking document.

Report Format:

Each lab report should at least contain the following clearly indicated sections. More sections are permissible but the document should remain concise and well organized.

- **Objectives** (Say why you’re bothering): This section states that point of performing the lab activity. This section is generally a rewording of the stated activities objectives in such a way as to show that understood what you’re attempting to do with in the lab activity. Don’t just copy the state lab activity’s objectives; use your own words instead.

- **Procedures** <u>(Say what you did):</u> Describe what you did during the lab activity.

    - The use of words in this section should be minimized in favor of more expressive items such as truth tables, equations, circuit diagrams, block diagrams, timing diagrams.
    - Someone should be able to read though this section and know exactly where you started, where you ended, and the steps you used to arrive at your destination. The flow of this section should match the flow of tasks during the lab activity. This section should not depend on the description of the lab. In other words, assume that the person reading your lab report does not have a copy of the lab description.
    - This section should be written using normal English sentences and paragraphs as opposed to bulleted or numbered lists of tasks and/or commands.

- **Testing** <u>(Say why you think you did it correctly):</u> Describe the testing procedures you used to verify your circuitry met the design criteria.

- **Conclusion** <u>(Sum up the experience):</u> See comments below.

- **Questions** <u>(Say what you're supposed to say):</u> Be sure to include the answers to any questions that may appear at the end of lab activity description in your lab report.

    - Answer the questions in such a way as to re-state the original question.
    - If the question requires that you perform some non-trivial calculations, include those with the lab report.

The Conclusion:

- Conclusions should contain the following information:

    - A brief description of what was done in the lab activity (2-3 sentences).
    - Wording that implies you understand the concepts presented in the lab activity.
    - Wording that describes how the lab activity relates to other lab activities and/or topics discussed in lecture or in the world in general.
    - Wording that indicates the objectives of the lab activity were met. Don't simply state that the lab activity's objectives were met; support the assertion indirectly with your wording.

- Conclusions should *not* contain the following:

    - Detailed descriptions of the procedure followed during the lab activity.
    - Detailed descriptions of the circuits designed or used during the lab activity.
    - Comments regarding whether you liked the lab activity or not.
    - Comments regarding how much you learned during the lab activity.
    - Comments that state directly that the lab activity's objectives were met.

Lab Work Submission Guidelines

This document contains the guidelines of how your submitted lab work ("lab work" is different from a lab report) documentation should appear. Adhering to these guidelines strongly indicate that you put the time and effort into understanding the lab material. Moreover, properly formatted submissions are easier to verify that your work is correct. Because the EE Department bean counters continue to overschedule the digital labs, you need to submit quality reports. You can find more formatting-related information in the appropriate HDL Style File and RAT Assembly Style File documents (CPE 233). Most importantly, if you have questions, ask your instructor before you submit sub-par work.

The Guiding Question: ***Would I submit this work to my supervisor?***

If the answer is no, then consider quality improvements before submission.

Submission Ordering:

1) Cover sheet with course names & sections, group member names, executive summary of experiment
2) Diagrams (schematics, state diagrams, etc. from experiment assignment)
3) Source code (Verilog, VHDL, assembly code, etc. from experiment assignment)
4) Answers to question set (include question with answer)
5) Design Problem solutions (if any: HW & SW problems: code and/or diagrams)

General Comments:

- Only submit one lab submission per lab group, unless directed otherwise
- Lab reports should be stapled high in the upper-upper-left corner
- Lab work submissions should be "stand alone", which means that anyone can pick up the submission and know what they're looking at. This means:
    - All problem-type questions include the problem statement in an appropriate location
    - Answers to questions should include the question
- Your writing need to be terse and concise. Save the verbosity and schmooze for other courses where lazy professors don't read your work or have student graders do their work for them.
- Lab submissions should be neat; use whitespace make submissions neat, organized, and readable
- All diagrams (timing, state, circuit diagrams, etc.) should include a title and/or caption
- Do not break tables or diagrams across pages
- Use hex notation for all signal values wider than four bits
- Use engineering notation for all numbers
- Do not put source code in the body of a report; attach it to the back of report instead
- Do not include source code for anything you did not write

Lab Activity Questions:

- Include the question you're answering before you write the answer to the question
- Use white-space to delineate questions and answers to make them more readable

Hardware (Schematic) Diagrams:

- Must be neat; can be hand drawn or drawn using drawing program
- Do not route control and status signals to and from FSMs, but clearly label both ends of signals
- Do not route clock signals, but clearly label at clock inputs and outputs
- Do not use "hump notation"

Simulator Printouts:

- Timing diagrams should be annotated (handwritten is fine, but must be neat)
- Timing diagrams should be outputs from the simulator and never cut-and-paste screen shots

- Timing diagrams should primarily include the “things of interest” by using the correct amount of magnification
- Timing diagram annotations should generally show the causality of signals changes and/or values

FSM State Diagrams:
- State diagrams can be handwritten but must be neat
- FSM state bubbles should have symbolic names that roughly indicate purpose of state
- FSM input and output signals should have symbolic names that roughly indicate their purpose
- FSM states should only list outputs that are critical to that state; do not include output signals associated with a state if they are a “don’t care” relative to that state
- State diagrams should include legends indicating inputs, output, and state name

General Source Code (HDL Assembly)
- Each separate file of code should begin on a new page
- Source code files should have header that include names of group member, lab activity number, and a brief but complete description of the file’s content
- Never use the tab key for any reason when writing source code
- Use “courier new” font for all submitted source code (VHDL, Verilog, assembly, C, etc.)
- Each file of code should have header describing what the code does and who wrote the code
- Do not allow lines of source code to wrap
- Code should be properly indented (see appropriate style files)
- Code should use white space to increase readability of code
- Code should be sufficiently commented
- Code should use self-commenting labels (labels, signal, variable, entity, architecture names, directives, constants, etc.)
- Code should make no more than one assignment per line
- Code should be printed using separate files and included with report

HDL Source Code Specific:
- Code should define only one item (signal, variable, etc.) per line
- Code should use “s_”, “v_”, or “r_” prefix for signals, variable, or registers, respectively
- Structural models use direct mapping only
- Code should declare no more than one variable per line (entities & declarations)

RAT Assembly Source Code Specific (CPE 233 only):
- All subroutines should have proper headers describing what the subroutine does which registers are permanently modified by the subroutine
- All instructions & directives, the left-most instruction operands, and comments should be aligned

Note on Any Lab Activity Submission

As indicated in the lab activity reporting guidelines, there is some stuff you must always place in your lab reports. In case you didn't really read those guidelines, a short list of this stuff includes the following. Keep in mind that you every report should also include sections of Objective, Procedures, Testing, and Conclusion.

- Boolean equations you may have used in the lab activity
- Derivations you may have needed to do in the lab activity
- Any non-trivial calculation you performed in order to complete the lab activity
- All source code (HDL and/or assembly language) generated in the lab activity: properly formatted, well commented and with a really nice header
- All pertinent timing diagrams from the lab activity
- All circuit diagrams from the lab activity
- Any state diagram used in the lab activity
- Questions and their answers that appear at the end of the lab activity
- Solutions to any design problems that are part of the lab activity

Exp 1: Half Adder (HA) & Full Adder (FA): Standard SOP Form

Objectives:

- To be exposed to the Xilinx Design Methodology
- To design, implement, and two arithmetic-based digital circuits
- To obtain an introduction to Verilog
- To get a feel for instructor expectations for each experiment procedures and write-ups

Somewhat Meaningful Comments: Designing circuits that perform arithmetic operations is one of the more common pursuits in digital design-land. The half adder (HA) and full adder (FA) are probably the most basic digital circuits that just about everyone who is anyone must design at one time or another in their digital design careers. Both of these circuits are known as 1-bit adders; they only differ in that the FA has a carry-in input while the HA does not. The outputs of both circuits are the same: the SUM output indicates the result of the addition operation while the carry-out (CO) indicates whether the operation generates a carry (CO).

This experiment is primarily an introduction to both the Xilinx Design Methodology as it pertains to the Vivado software, and Verilog. This experiment has you design and implement two arithmetic circuits on the development boards. You'll find every lab instructor has different expectations of proper lab "conduct" and subsequent lab write-ups, thus this experiment also gives you an idea of what the instructor expects in both of these areas.

Assignment: Part A: Design a half adder and implement your design on the development board. Table 1 shows the input and output assignment for the devices on the development board. Use standard SOP form for you implementation.

Input		Output	
OP_A	OP_B	SUM	CO
left-most switch	second switch from left	left-most LED	right-most LED

Table 1: Input/Output (I/O) specification for the HA

Assignment: Part B: Design a full adder and implement your design on the development board. Table 2 shows the input and output assignment for the devices on the development board. Use standard SOP form for you implementation.

Input			Output	
OP_A	OP_B	Cin	SUM	CO
right-most switch	second switch from right	third switch from right	left-most LED	right-most LED

Table 2: Input/Output (I/O) specification for the FA

Special Deliverables:

None

Questions:

1. In your own words, what is meant by the term “methodology”.
2. Briefly describe the purpose of the “constraints file”, which is another name for the file with the “.xdc” extension.
3. Briefly describe what you’re doing when you “synthesize” your HDL model.
4. There is a programmable logic device (PLD) on the development board used for this class. What particular type of PLD is on the dev board?
5. How many assignment statements did you use in your HDL model for the HA and FA?
6. Briefly describe the basic limitation of the HA (relative to the FA) in the context of a mathematical digital circuit.
7. Briefly described how you verified your circuit in this experiment was working properly.
8. Briefly describe the main purpose of a “gate” in the context of this course.

Design Problems:

1. Design a 1-bit subtractor. This circuit has two inputs (A, B) and two outputs (SUB for the subtraction result of A -B, Bo for a “borrow” where appropriate). For this problem, show the equations for the outputs in standard SOP form.

Deliverables:

This section describes the special deliverables. This is the only experiment that lists all of the deliverables; see the list in the “Lab Work Submission Guidelines”.

1. Top-level black-box models for all circuits
2. Lower-level circuit diagrams for all circuits
3. Tables and Equations associated with circuits
4. HDL source code for all circuit models

Exp 2: Half Adder (HA) & Full Adder (FA): Standard POS Form

Objectives:

- To be exposed to the Xilinx Design Methodology
- To gain more experience working with Xilinx design environment
- To demonstrate functional equivalence of SOP and POS forms

Somewhat Meaningful Comments: This experiment entails the design of two circuits you previously designed: the HA &FA. In this experiment, we'll demonstrate the notion of *functional equivalence* by designing these same two circuits using a different but equivalent form of Boolean equations, namely the standard POS form.

Assignment: Part A: Design a half adder and implement your design on the development board. Table 3 shows the input and output assignment for the devices on the development board. The output of your circuit should show both standard SOP and POS forms both the SUM and CO signals.

Input		Output			
OP_A	OP_B	SUM_SOP	CO_SOP	SUM_POS	CO_POS
left-most switch	second switch from left	left-most LEDs	second from left-most LED	second from right-most LED	right-most LED

Table 3: Input/Output (I/O) specification for the HA

Assignment: Part B: Design a full adder and implement your design on the development board. Table 4 shows the input and output assignment for the devices on the development board. The output of your circuit should show both standard SOP and POS forms both the SUM and CO signals.

Input			Output			
Cin	OP_A	OP_B	CO_SOP	SUM_SOP	CO_POS	SUM_POS
third switch from right	second switch from right	right-most switch	left-most LEDs	second from left-most LED	second from right-most LED	right-most LED

Table 4: Input/Output (I/O) specification for the FA

Special Deliverables:

- none

<u>Questions:</u>

1. Since you now know there are many ways to implement digital circuits, describe some possible parameters involved in ascertaining the “best” way.
2. Briefly comment on which form (SOP or POS) of the HA &FA was easier to design and implement. Provide a brief justification for your answer.
3. Briefly describe how the full adder is somewhat limited in doing math operations.
4. Briefly describe how you could configure the two full adders to become a two-bit adder.
5. Generate a table the compares the number of AND gates, OR gates, and inverters to implement the FA in both SOP and POS forms. Show your equations for this problem, meaning you should have a total of four equations. Draw the final circuit for these two functions also.
6. Briefly described how you verified your circuit in this experiment was working properly.
7. Write a “conclusion” for this experiment. Note that a conclusion describes what the point of the experiment was.

<u>Design Problems:</u>

1. Design a 2-bit “modulo-2” adder. This adder has two 2-bit inputs and one 2-bit output. The output of each individual bit is “0” when adding 0+0 and 1+1; otherwise the output is “1”. Show only a truth table for this design. Be sure to include a black box diagram for your solution.
2. How would you use the circuit (not necessarily a single circuit) from the previous problem to create an 8-bit modulo-2 adder. Be sure to include a black box diagram for your solution at both a high-level and the “next lower level”.

Exp 3: 5-Bit Ripple Carry Adder (RCA)

Objectives:

- To use a HDL structural model to support notion of hierarchical digital design
- To use HDL structural model to support the notion of modular design and code reuse in HDL

Somewhat Meaningful Comments: Arithmetic circuits are massively common out there in digital design-land. One of the most basic and useful circuits the Ripple Carry Adder (RCA). For the purpose of this activity, the RCA comprises of a HA for the LSB and FAs for all the other bits. Figure 1 shows a schematic diagram of a 4-bit RCA.

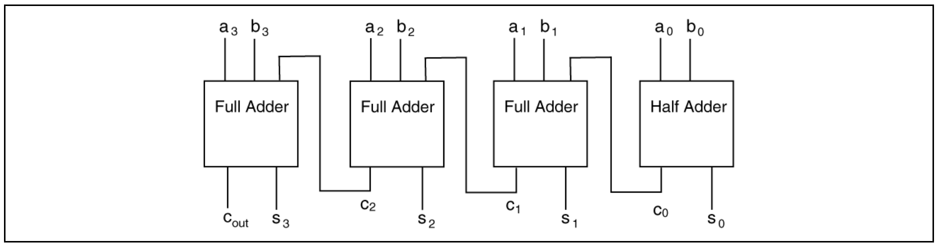


Figure 1: Schematic diagram for a 4-bit Ripple Carry Adder (RCA).

Assignment: Using your previous half adder and full adder modules, design a 5-bit RCA, similar to the 4-bit RCA shown in Figure 1. Implement your RCA on the development board using the pin assignments in Table 5. Additionally:

- Use HDL structural modeling for your implementation using both the HA and FA models you previously designed.
- Use bundle notation in your HDL code wherever possible.

Inputs		Outputs	
A	**B**	**SUM**	**Carry-out**
a_4-a_0: SW15-SW11	b_4-b_0: SW4-SW0	s_4-s_0: LD4-LD0	C_{out} = LD7

Table 5: Pin assignments for the RCA.

Deliverables:

1. HDL models for all modules you wrote in this lab activity. This experiment is a hierarchical design, so include both the higher and lower-level modules.
2. Black-box models for the circuits you modeled in this lab activity
3. Answers to the “questions”
4. Solution for design problems

Questions:

1. Briefly describe the two main attributes of modern digital design.
2. Briefly describe why is it a good idea to avoid modifying previously designed modules in your new design?
3. In your own words, briefly but completely explain why we refer to the circuit in this lab activity to as a “ripple carry adder”.
4. If you needed to extend the RCA from this lab activity to a 10-bit RCA by using a structural model with two 5-bit RCAs, what changes would you need to apply to the 5-bit RCA?
5. How many rows were there be in a truth table for a 32-bit RCA? Would it be feasible to design a 32-bit RCA using a truth table?
6. How many logic gates would it require to implement the 5-bit RCA using discrete logic? For this problem assume the logic is in standard SOP and the LSB uses a HA.
7. Write a formula in closed for that describes the number of gates in a RCA as a function of the bit-width of the RCA. Recall that the LSB of the RCA is a HA. For this question, assume the FA and HA are in reduced form (not in standard SOP form). Feel free to ask the instructor for the “reduced” form of both the HA and FA.
8. For a RCA, the result could be available immediately, or the result could be delayed. Describe a case where the result is available immediately and also describe a case where the delay is the “worst case”. State how long the worst case is in terms of “gate delays”.
9. Briefly describe the notion of concurrency in digital circuit design.
10. Do the various module instantiations in a HDL model operate in a concurrent manner? Briefly explain why or why not.
11. Consider two different HDL models that are functionally equivalent; in particular, they are mostly the same except for the notion that one used structural modeling and the other implemented a similar set of modules but did not use structural modeling. Would expect the synthesized circuit based on these models to use the equivalent amount of resources or will one approach use more or less resources. Support your answer with intelligent commentary.

Design Problems:

1. Design a circuit that adds four 10-bit unsigned binary values and outputs an unsigned binary 10-bit result. The circuit also has a VALID output that indicates when the 10-bit unsigned binary result is valid. Note that when you add two 10-bit numbers, you will generate a 11-bit result (in some cases). Show the top two levels for this solution, meaning a top-level BBD, and the next lower level which is RCAs and some gates. There is no need to show the next lower level, which would be the gate-level view of the RCAs. State how the circuit is controlled.

Exp 4: BCD-to-Seven-Segment Decoder

Objectives:

- To design the circuitry to drive an external 7-segment display.
- To design, implement, and test three digital circuits

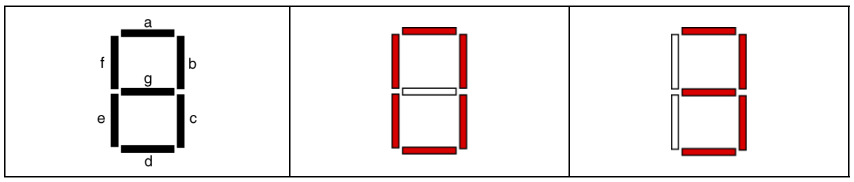
Somewhat Meaningful Comments:

The 7-segment display is one of the most common display devices in the universe; we generally use these devices to display decimal numbers. These devices as also capable of displaying the full gamut of hex numbers (although the case of the alpha characters is not consistent). The 7-segment displays on the development board consist of seven LEDs (light emitting diodes). There are actually eight LEDs, one is for a decimal point. The LEDs are the small button-like colored devices that light up occasionally during your experimentation. The LEDs in this lab activity were elongated by using small stretching racks and configured in such a manner as to easily represent decimal numbers.

The LED is a two-terminal polarity-sensitive device that turns-on when the voltage conditions across the two terminals are at the correct levels. We refer to the two LED terminals as the *anode* and the *cathode*. To make the LED emit light, the anode must be have a 0.7V higher potential than the cathode. What this means to you in digital design-land is that you must either provide the LED with a ‘1’ or a ‘0’ depending on how the LED is wired in the circuit. To find out exactly how the LED works on your dev board, read the specification of the development board (RTFM).

We represent individual decimal numbers by turning on specific sets of the segments. Referencing the seven segments is done by assigning unique letters to each of the segments. Figure 2(a) shows the most common listing of these segments is. Figure 2(b) shows a 7-segment display creating the illusion of a ‘0’ by lighting all the segments except segment ‘g’. Figure 2 (c) shows segments a, b, c, d, and g lit to simulate the number ‘3’. Once again, there are two types of 7-segment displays and two ways to connect ways to control the anodes associated with the 7-segment displays. Every development board uses different parts and wires them as required. Check the dev board’s user’s manual.

Most importantly, making an LED emit light is a two-step process on a 7-segment display. You need to both turn-on the LED and also actuate the individual 7-segment display. Both of these actuation steps involve sending a logical ‘1’ or ‘0’ to the device. Consult the reference manual for the development board for the details (and then ask a bunch of questions).



(a) **(b)** **(c)**

Figure 2: The amazing 7-segment display (a), and ‘0’ (b), and a ‘3’ (c).

Assignment: Design a BCD to 7-segment decoder using an HDL to model the circuit.

- The 4-bit input of this circuit is a BCD number. The eight outputs are the various segments of the display (including the decimal point). Use all the seven-segment displays and make them selectable using one switch for each display.
- When the input BCD value exceeds the range of decimal digits, turn off the display. Table 6 lists the input and output assignments for you development board.
- Connect the four anodes to the four left-most switches. The switches in the UP position generate a logic '1'. See Table 7 for even more details.
- Verify your design works properly visually using the pin assignments in Table 6. Choose your test vectors such that you test the major functionality of your circuit, but fit the entire simulation onto one page.

Input	Output
SW3,SW2,SW1,SW0	CA,CB,CC,CD,CE,CF,CG,CDP
four right-most switches	right-most 7-segment display

Table 6: Input/Output specification for the BCD to 7-Segment decoder.

Input	Output
SW15, SW14, SW13, SW12	AN3, AN2, AN1, AN0
four right-most switches	right-most 7-segment display

Table 7: Input/Output specification for the anodes.

Special Deliverables:

- none

Questions:

1. If you were not able to use a decoder in this experiment, who many concurrent signal assignments would you have needed to implement the segments portion of the seven segment display? Briefly explain.
2. Draw a black box diagram showing the anodes, LEDs, segments, switches, and buttons on the development board. Be sure to carefully labeled inputs and outputs on your diagram.
3. Similar to 7-segment displays, there are also 14-segment displays out there in the real world. Briefly describe the main purposed served by 14-segment displays.
4. This lab activity required that you use a generic decoder. You could have modeled this decoder using one of two possible types of procedural blocks. Provide the code for a generic decoder using one of the statements you did not use in this lab activity.
5. One of the important design approaches in modeling digital circuits is to use a LUT (decoder) whenever possible. Briefly describe why this is a good approach.
6. How can a the set of seven segment displays ever display a number such as "3948" if you can only ever display one number at a time on the set of displays?

7. A four-digit seven-segment display has 32 LEDs, which seems to indicate we needs 32 outputs to control it. But… a four-digit seven-segment display is typically driven by 12 signals. Briefly explain the main reason this is done.

Design Problems:

1. Non-standard decoders are essentially LUTs. As you know from computer programming, often times using a LUT for calculations is a great idea. For this problem, show the BBD and code for a 4-input decoder that outputs the square of the input. Consider both input and output to be unsigned binary numbers. Use as few signals for the output as possible but still be able to represent the largest possible value for the output.

Deliverables:

1. HDL models for all modules you wrote in this lab activity
2. Black-box models for the circuits you modeled in this lab activity
3. Answers to the "questions"
4. Solution for design problems

Exp 5: 8-Bit Comparator with 4-Bit Comparator Modules

Objectives:

- To design and implement one of the basic digital logic circuits
- To discover the power of HDL behavioral modeling
- To gain more experience and practice with structural modeling
- To write a HDL testbench to simulate your circuit

Somewhat Meaningful Comments: This lab activity introduces the notion of HDL behavioral modeling in the context of n-bit comparators. While you'll quickly find that a designing a simple comparator is no big deal using behavioral modeling, this lab activity becomes more meaningful by asking you to connect two 4-bit comparators in such a way as to form a single 8-bit comparator. While it would be easier to model an 8-bit comparator directly, using two 4-bit comparators builds character and provides practice with structural modeling and general digital design concepts.

Simulating your HDL models is massively important for two reasons. First, it allows you to verify your circuit is operating properly before you actually implement the circuit. Secondly, it provides a way to debug your circuit if your circuit does not work.

The notion of simulating in HDL is a matter of generating a "testbench". The testbench is simply a special form of a HDL module; it's only function is to provide "stimulus" to the circuit you design (feed the inputs with a meaningful signal). We generally refer to this stimulus as "test vectors". You the circuit designer are responsible for generating what you feel are enough test vectors to give you the warm and fuzzy notion that your circuit is operating properly. Once again, you the designer will be asked if your circuit works; you base your answer on how well you tested your circuit. In a perfect world, you would spend a lifetime testing your circuit; in reality, you're only going to have time to test a reasonable portion of your circuit, and then you need to justify why your limited test vectors prove your circuit is working. Be sure to check out the appendix for the full smear of comments on testbenches.

Assignment: Part A Design an 8-bit comparator using two 4-bit comparator modules. Figure 3 shows the two high-level black box diagrams for this lab activity. Figure 3(a) shows the 4-bit comparator module you'll use to build the 8-bit comparator module in Figure 3(b). The inputs to 8-bit comparator are both 8-bit unsigned numbers.

- Use the provided n-bit comparator model in you design
- Implement your design on the dev board using the 16 switches as inputs (A & B) and three LED as outputs (EQ, GT, LT).
- Simulate your design using a testbench. Annotate your simulation output to draw the reader's eyes to what you are testing.
- Your final black box diagram will have three modules: two 4-bit comparator modules and an extra "logic" module. Implement the logic module as a behavioral model.
- Use only two files for this design: one for the n-bit comparator module and one for the top-level module that contains the two 4-bit comparator instantiations and the behavioral model for the extra logic referenced in the previous bullet.

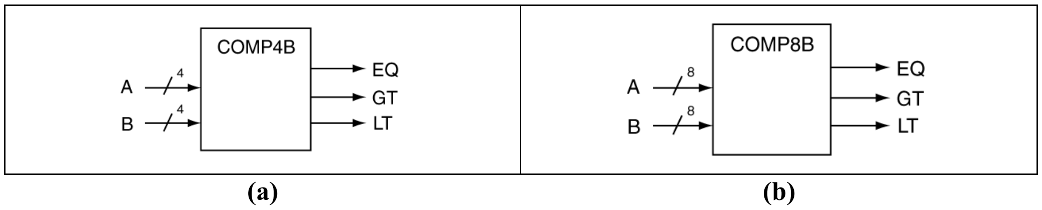


Figure 3: Black box diagrams for this lab activity: the underlying component comparator (a), and the final top-level BBD (b).

Assignment: Part B Write a testbench and simulate your comparator model using the development environment's simulator. Use the provided cheatsheet for a guide. Once you have your waveforms displayed on the screen, call over the instructor of a TA and ask them to show you the "important stuff".

- Choose your testbench test vectors such that you adequately test the major functionality of your circuit, but fit the entire simulation onto one page.
- Print out then annotate the resulting timing diagram (hand annotations is fine)
- Use hexadecimal for all bundled input and output signals.

Constraints:

- Minimize your use of hardware in your design
- Use a procedural block (an **always** block) for the "logic" portion of your design

Deliverables:

1. HDL models for all modules you wrote in this lab activity
2. Black-box models for the circuits you modeled in this lab activity
3. Answers to the "questions"
4. Solution for design problems
5. Well annotated waveforms from the simulator.

Questions:

1. Briefly describe whether you could have used a decoder for the "logic box" portion of this lab? Also, provide a justification as to why you would not and/or did not use a decoder for this part of the circuit.
2. In your own words, briefly describe the advantage(s) of using a behavioral model as opposed to a gate-level implementation of the comparator.
3. Based on the previous question, you probably see that modeling comparators of any size is no big deal using an HDL. Why then did this lab activity ask that you model a larger comparator using two smaller comparators?
4. We all know that the well-known hallmark of a comparator is that it uses EXOR-type gates in its implementation. Does the development board you're using actually use EXOR-type gates in your design? Briefly but completely explain.

5. The **always** block is one of Verilog's concurrent statements. But, the **always** block contains only sequential statements in the body of the always block. Describe how this seemingly oxymoron is actually possible.
6. The **always** block contains a "sensitivity list". Briefly describe what this is. Also, state what you should generally place in the sensitivity list.
7. In your own words, what is a test vector and what entity generates the vectors for your testbench in this experiment?
8. Briefly explain what it would mean if the output of your simulation did not match the outputs on hardware for the same inputs?
9. Humans primarily use decimal while computer hardware uses binary. Briefly explain what the point of having hexadecimal in the fray also.
10. We often use hexadecimal notation in digital-land. Can you use hex notation to represent signed binary numbers in RC format? Briefly explain.

Design Problems:

1. Design a circuit that has four 10-bit unsigned binary inputs A, B, C, D. The output of the circuit has the same three outputs as a normal comparator. If the sum of A+C is valid (10-bit result) and the sum of B+D is value (10-bit result), then the output of the circuit reflects the result the comparison of A+C & B+D. If either addition operation generates a carry, all of the three circuit outputs (EQ, LT, & GT) should be zero. Don't use any MUXes in your solution. Draw the BBD for this circuit as well as the lower-level schematic. Minimize your use of hardware in your design.

Exp 6: Five-Bit Magnitude Comparator

Objectives:

- To gain more skills using standard digital devices, block-level design, and structural modeling
- To gain yet more experience designing and using arithmetic circuits

Somewhat Meaningful Comments: Arithmetic circuits are massively important in digital design. Understanding the meaning of binary number representations is one of the keys to being able to design and implement arithmetic circuits that deal with binary numbers. This lab activity provides you with more experience in several of the main areas associated the design and implementation of digital circuits. This circuit is somewhat complicated; you'll need to start out with a block diagram to map out your design before you start writing HDL models.

Assignment: Design and implement a 5-bit magnitude comparator on the development board. This circuit inputs two 5-bit RC values (RC format) and displays information about these two values according the description that follows.

- Use the provided univ_sseg.v module to drive the 7-segment display with the required output values.
- Use the left-most and right-most switches to represent the two 5-bit signed binary inputs, where the sign-bit should always be the left-most bit of the input.
- If the magnitudes are not equivalent, have the 7-segment display show all dashes (be sure to read the header of the univ_sseg.v module).
- If the magnitudes are equivalent and the numbers are equivalent, display the input value; otherwise, display the magnitude.

Constraints:

- Minimize your use of hardware in your design.

Hints:

- Use the provided RCA in this design; instantiate appropriately for this experiment
- You don't need to implement this entire design using structural modeling.
- This is your first experiment that includes a clock input. This means that your top-level module contains a clock input. You then connect this input to the clk input on the univ_sseg module. Keep in mind that the development board provides the clock signal to the FPGA. The clk is a true input, which means you must uncomment the clock oriented lines in the constraints file.

The univ_sseg.v Module:

The module is provided for you; you can find the instructions for using the device by reading the instructional notes in the module's header. Figure 4 show the black box diagram for the univ_sseg.v module.

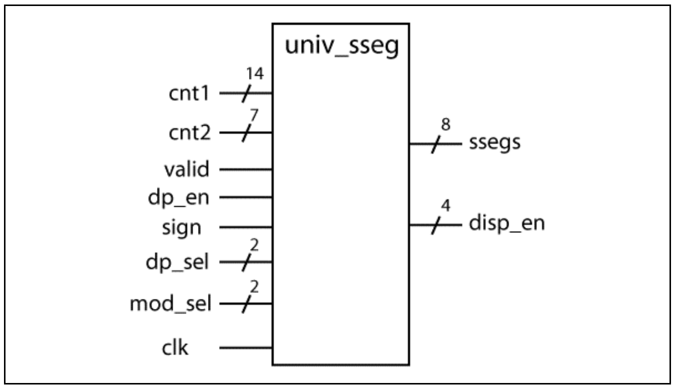


Figure 4: univ_sseg.v black box diagram.

Signal Name	Mode	Description
cnt1	**in**	This is a 14-bit unsigned binary input that operates differently for different modes. • mod_sel = 00: lower 8-bits used for [0,255] • mod_sel = 01: lower 7-bits used for [0,99] • mod_sel = 10: lower 14-bits used for [0,9999]
cnt2	**in**	This is a 7-bit input used in **mod_sel** = 01 [0,99]
valid	**in**	If this input is a '0', all the displays show a '-'; otherwise the display acts according to other selections.
sign	**in**	If this input is a '1', all the displays show a '-'(minus sign) in mod_sel = 00.
dp_en	**in**	This is input turns on a decimal point according to the **dp_sel** input.
dp_sel	**in**	This input controls decimal point display; the **dp_en** input must be on for this input to work. Only one decimal point can be active at any time. • 00: dp displayed on right-most 7-seg display • 01: dp displayed on middle-right 7-seg display • 10: dp displayed on middle-left 7-seg display • 11: dp displayed on left-most 7-seg display
mod_sel	**In**	This input selects the display mode: • 00: displays one count [0,255] with optional '-' sign • 01: displays two counts [0,99] (no sign) • 10: displays one count [0,9999] (no sign) • 11: displays average academic administrator IQ
ssegs	**out**	These are the a→dp outputs for the 7-segment displays. These are officially the cathodes.
disp_en	**out**	These are the anodes for the 7-segment display.

Table 8: Brief description of univ_sseg.vhd module inputs & outputs.

HDL Notes: The RCA in this experiment is provided for you. With external modules, you don't always need to use all the inputs and outputs of the module in your designs. For HDL to be happy, you must do the following for inputs and outputs:

INPUTS: All inputs require values, so you must assign them a '0' or '1'. The value you assign is typically the "don't do anything value" for that input as you're not going to use it and you don't want it to affect the module. The common vernacular is that you "hard-code" the input values.

OUTPUTS: If you're not using an output, you still include the output in the instantiation listing, but you don't associate it with anything. This approach serves as a note that you intentionally are not using that particular output. If this approach generates a warning, you can ignore it.

One key item in Verilog that you use all the time is the append operator ({,}). For example, this experiment works with 5-bit numbers, which you'll need to expand to for the univ_sseg.v module. In words, you'll want to bit stuff it with zero. Here's the Verilog code for the operation:

```
Signal_8bit = {3'b000, signal_5_bit}; -- convert signal from 5 to 8 bits
```

Deliverables:

1. HDL models for all modules you wrote in this lab activity
2. Black-box models for the circuits you modeled in this lab activity
3. Answers to the "questions"
4. Solution for design problems

Questions:

1. Show a closed form formula that relates the data width the number range for an unsigned binary number. Make sure you use an accepted format for specifying number ranges.
2. Show a closed form formula that relates the data width the number range for a signed binary number in RC format. Make sure you use an accepted format for specifying number ranges.
3. For a given bit-width, does the number of unique numbers in an unsigned binary and signed binary number in RC format differ? Briefly explain.
4. Can an 8-bit binary number in unsigned format ever be an odd number but have even parity? Briefly explain.
5. Can signed binary numbers in RC format have the notion of parity associated with them? Briefly explain your answer.
6. In computer programming, briefly describe why it is the best idea to use an unsigned integer type when it is known that the value will never be negative.
7. In computer programming, briefly describe what happens when a mathematical operation exceeds the ranges for the data types associated with that operation? Does the programmer typically know the range has been exceeded?
8. Often we refer to a design as a "flat" design. In terms of HDL modeling, briefly but fully describe what that term refers to.
9. In general, does the number of levels of a particular design affect the resources required to implement that design using an HDL and implementing that circuit on an FPGA? Briefly but fully explain.

<u>Design Problem:</u>

1. Design a circuit that four 10-bit inputs; two of the inputs are in RC format while the other two inputs are in unsigned binary format. The circuit has two outputs; one output indicates when all four inputs have equivalent decimal equivalents. The other output indicates when all four values have equivalent magnitudes. Use foundation modules when possible; minimize your use of hardware. State how the circuit is controlled.

Exp 7: Signed Binary 5-Bit Adder/Subtractor with Validity Detection

Objectives:

- To design a basic arithmetic circuit with output validity feature
- To gain more practice in block-level modeling and HDL structural modeling
- To gain more experience designing and using arithmetic circuits

Somewhat Meaningful Comments: This activity is similar to a previous lab activity in many ways. First, it should be designed on a block diagram level. Secondly, it should use the univ_sseg.v you used in a previous design. In other words, a good starting point for this circuit would be the final design in the previous lab activity. For this activity, you'll be designing circuit that adds two 5-bit signed binary numbers (four bits plus the sign bit) in RC format. As you know, the results of such operations are sometimes not valid so this circuit must clearly indicate this condition by using the VALID input of the univ_sseg.v module.

Assignment: Design 5-bit signed binary number adder/subtractor circuit.

- If the button is pressed, the circuit displays the results of A + B; otherwise, the circuit displays the results of A – B.
- Use the switches on the development board for the two 5-bit inputs (RC format) by considering the left five switches to be the A operand and the right five switches to be the B operand.
- Because the result of the mathematical operation may overflow the given range, you must use the VALID input on the univ_sseg.v module to indicate if the result of the operation is valid or not.
- Keep in mind that this circuit is required to perform indirect subtraction by addition. Be sure to use the provided RCA module (modify as you need to) for your implementation.
- Simulate your final design; make sure you include enough test vectors to adequately test your design.
- You don't need to check for that "special case" of validity in your design.

Simulation Notes: The required simulation for this experiment has some issues. The univ_sseg.vhd module is responsible for multiplexing the four 7-segment displays.

- Part of the univ_sseg.v module includes a clock divider, which the module requires to properly multiplex the displays. Because of this delay, your simulation output will rarely see a change in the univ_sseg.v's segment output (it changes about once every 2^{13} input clock cycles). This then requires some cleverness in testing.

  There are several ways to get around this, but there is one fairly simple approach for this experiment. When you run the simulation, the simulator automatically provides you with inputs and outputs on the highest level of the device under test. These provided values will not prove that your circuit is working for a long simulation time. The solution is to include the cnt1, sign, and valid output on the timing diagram. These signals are not directly involved with the univ_sseg.v module's output, so they are not subject to the delays associated with the multiplexing operation of the module. This means that you can see the desired changes in the cnt1, sign, and valid signals that prove your test vectors generate the correct outputs of the univ_sseg.v, but you won't see the segment and anode outputs of the module change. Be sure to annotate the cnt1, sign, and valid signal in your submitted simulation output.

Constraints:

- Don't use mathematical operators in this experiment other than provided modules.
- Minimize your use of hardware in your design.

Deliverables:

1. HDL models for all modules you wrote in this lab activity
2. Black-box models for the circuits you modeled in this lab activity
3. Answers to the "questions"
4. Solution for design problems
5. Annotated timing diagram output for your design; don't include the HDL testbench file.

Questions:

1. What is a really good reason that I don't put in the effort to design a subtractor in addition to my RCA?
2. The univ_sseg.v module you used in this lab activity seemed to do a great job of simultaneously displaying more than one number at a time on the development board's seven-segment displays. But you were previously told that the displays could only show no more than one number at a time. Briefly explain what is going on.
3. Briefly describe the steps necessary in order to extend this design to 8-bit signed binary numbers (once again, assume your development board and provided modules are not limiting factors).
4. The output of this circuit arbitrarily showed dashes when the number was not valid. Speculate on why the designer of the module chose that particular output.
5. Using the development board under the conditions stated in this lab activity, would it have been possible to design an adder/subtractor unit based on 9-bit signed binary numbers (RC format)? The problem here is that you run out of switches to support 18-bit number. Consider all possibilities and fully explain your answer.
6. Quite often in digital design, there are boundary condition issues you need to deal with. What this means to me is that 98% of the errors I make in a design are with a boundary condition. This lab activity also has a boundary condition that essentially renders the result invalid even though it passes our simple validity checker. Briefly describe this boundary condition. HINT: the notion that it is a boundary condition roughly means that is has something to do with the far end of the given number range for the RC numbers.
7. Describe the modifications you would need to make to the circuit in this lab activity is you needed a 2*A or A-B circuit instead of the A+B or A-B. Do not use a shift register in your solution.
8. Briefly state why you feel the number of test vectors you used in testing this design was sufficient to justify stating your design was working properly.

<u>Design Problems:</u>

1. Design a circuit that converts a three-digit decimal number to an 8-bit unsigned binary number. This circuit has three BCD inputs, which means four bits for the 100's, 10's, and 1's digit. The 8-bit output will always be sufficient to encode the three digital input value. State how the circuit is controlled.
2. Show a modification to your design in this experiment to support that also supports that "special case of validity. You only need to modify the BBD for this problem.

Exp 8: Basic FSM Design: Multi Output 4-Bit Up/Down Counter

Objectives:

- To design a basic finite state machine from its basic underlying modules
- To design a basic counter that includes multiple controllable outputs

Somewhat Meaningful Comments: We typically model finite state machines (FSMs) as having three distinct underlying modules: the Next State Decoder, the State Registers, and the Output Decoder. The Output Decoder is special for several reasons. First, a FSM may not have an Output Decoder. Second, the Output Decoder can have one of two types of outputs. If the output is strictly a function of the FSM's state, the output is a Moore-type output. If the output is a function of both state and an external input, it is a Mealy-type output. This experiment requires an Output Decoder that contains both types of outputs.

The basic module you'll design in this experiment is a counter, which is a sequential circuit that outputs a repeatable sequence of values (the "count"). Counters typically synchronize their count output with the active edge of a periodic input such as a clock signal. The counter in this experiment is a 4-bit counter, which inherently means it counts with binary count sequence spanning [0,15], or ["0000", "1111"]. The counter in this experiment can count both up (0→15) or down (15→0).

Assignment: Design 4-bit up/down counter to the following specification.

- The counter is synchronous, which means changes in the count outputs are synchronized with the rising edge of the clock input. The counter counts up when the single button input (left-most button) is pressed; otherwise the counter counts down with the button is not pressed.
- The counter has two outputs as follows:
    - ❖ 15-bit stoneage unary output, which you display on the 15 LEDs.
    - ❖ Decimal output, which is displayed on the seven-segment displays (use the univ_sseg.v device).

Supporting Information:

Your counter will need a clock divider module (provided for you) in order to slow down the system clock from 100MHz to something us humans can see on the dev board. Configure the clock divider so that the count frequency is about 2Hz. Keep in mind that the univ_sseg.v module requires a fast clock such as the dev board's clock. Figure 5 shows a diagram showing the basic FSM model; not that this experiment requires only Moore-type outputs.

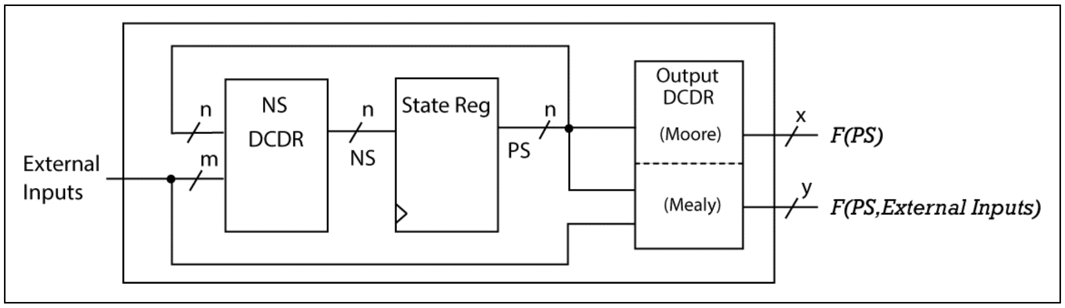


Figure 5: A diagram showing basic sub-modules of a FSM.

Constraints:

- Your design must include the three basic parts of a FSM. These must be modeled using discrete devices. That means decoders for the input and output, and registers for the state registers. I apologize in advance for the large decoders.

Deliverables:

1. HDL models for all modules you wrote in this lab activity
2. Black-box models for the circuits you modeled in this lab activity
3. State diagram; you only need to include states associated with {0, 1, 2… 13, 14, 15} (the idea is the full state diagram is repetitious (the thus tedious), so figure out a clean way of showing part of the state diagram while not wasting your time drawing the entire state diagram).
4. PS/NS table (no need to include the Output Decoder's output)
5. Answers to the "questions"
6. Solution for design problem(s)

Questions:

1. Briefly describe the difference between a flip-flop and a latch.
2. Sequential circuits are referred to as having "state". In your own words, briefly describe what exactly that means.
3. Briefly describe why state diagrams generally do not include any notion of a clock signal.
4. This experiment only asked you to draw a state diagram using six states. Briefly describe how many states the FSM in this experiment actually contains.
5. Briefly but completely describe the three basic modules of a FSM. Make sure your description includes which of the two classes of digital circuits the modules represent.
6. The state registers in a FSM are considered a synchronous circuit. Briefly describe what this means in context of the FSM.
7. This experiment used a clock divider to slow down the dev board's clock to something around 2Hz. Show how exactly you slowed down the clock, including any associated calculations.
8. Sequential circuits are known to contain memory, but it is not clear from the word "sequential" where the memory comes from. For this problem, briefly describe the relation between the word "sequential" and memory.
9. The FSM you designed in this experiment has status inputs and control outputs. Make a table showing the FSM's inputs and output; briefly describe the function of each input and output in terms of status and control signals.

<u>Design Problems:</u>

1. Design a 4-bit synchronous down counter. This output of this counter should reflect a binary output that represents RC numbers, which effectively means the counter counts as listed below. This counter should follow the standard FSM model that includes three basic modules, though you may not need all three modules for this problem. Provide a BBD for the top two levels of your design and a state diagram that models your FSM (feel free to abbreviate the state diagram, but make it clear).

{… 6, 5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5, -6,-7,-8, 7, 6, 5…}

Exp 9: 4-Bit Up/Down Counter w/ Multiplexed 7-Segment Display

Objectives:

- To implement a FSM with both Moore and Mealy-type outputs
- To implement a simple 2-digit 7-segment display multiplexing circuit
- To reinforce the notion of decoders when all else is in doubt

Somewhat Meaningful Comments: Yet another rite of passage in digital design is designing a circuit that drives a multiplexed display. The multiplexed display takes advantage of one of the less amazing aspects of the human visual system. Seven-segment display devices are common out there in digital land, and designing a multiplexed driver for such a device is not a big deal.

A four digit multiplexed display (7-segment devices) consists of 32 separate LEDs (including the radix point). This requires a relatively large amount of outputs to drive the circuit. To reduce the number of required outputs, we connect the display in a manner and take advantage of the human visual system (HVS) to make it "appear" to be doing what we want.

In a typical 7-segment display device, all the digits in the display share a single signal for each of the digits seven segments (and generally the decimal point as well). This means that at any given time, you can't drive more than one different number to the 7-segment display. In order to make it appear like there is more than one number on the display, we turn one number on for a short period of time, and then turn the next number on for a short period of time, etc. If we do this fast enough, it appears to your HVS that there is actually more than one number on the display at the same time. We refer to this as display multiplexing.

Assignment: Start with the circuit from the previous experiment, but do the following two modifications.

1) Remove the univ_sseg.v device and replace it with display multiplexing circuitry that you design. This multiplexing circuit must include a clock divider to slow the clock down to a reasonable frequency. This clock divider is in addition to the clock divider you already have in the circuit (recall that clock divider slow the clock down to a frequency that allowed humans to see the counting action on the 7-segment displays). The counter is a 4-bit counter that counts from 0-15, so you only need to design a two-digit display multiplexor.

2) Modify the stoneage unary output to allow it to show the count in either normal form or inverted form. Use the input that allows the counter to count up or down for this control. So when the counter is counting up, the LED display shows a normal stoneage unary count (LEDs on for each "count"); when the counter is counting down, show an inverted stoneage unary count (LEDs off for each "count").

Constraints:

- Minimize your use of hardware in your design
- Your display multiplexor hardware must be purely combinatorial

Deliverables:

1. HDL models for all modules you wrote in this lab activity
2. Black-box models for the circuits you modeled in this lab activity; make sure to show the appropriate low-level details of the display multiplexor.
3. Answers to the "questions"
4. Solution for design problems

Questions:

1. Briefly describe the FSM output-type classification of the stoneage unary output you used in the previous experiment and then in this experiment.
2. Briefly describe the FSM output-type classification of the decimal output you used in this experiment.
3. Briefly describe the particular attribute of a digital circuit (not of an HDL model) that gives the circuit the ability to store data.
4. Briefly describe the relation, if any, between a sequential circuit in digital design and a sequential statement in Verilog?
5. Briefly describe what the term “sequential” refers to in the term “sequential circuit”.
6. Briefly describe why an AND gate is not functionally complete.
7. Explicitly show how you can obtain an OR function from an AND gate. In your own writing, show both the derivation and the resulting circuit element.
8. Explicitly show how you can obtain an AND function from an OR gates. In your own writing, show both the derivation and the resulting circuit element.
9. An EXNOR gate is often referred to as an “equivalence gate”. Briefly explain how it would get such a name.
10. Briefly explain what characteristic of a circuit makes it a “mixed logic” circuit?

Design Problems:

Design a 3-bit synchronous up counter. Include a HOLD input that when asserted, prevents the circuit from counting. This output of this counter should be a 3-bit value in binary [0-7] format when the HOLD is asserted; otherwise a compliment of that 3-bit value when the HOLD is not asserted. This counter should contain all three basic FSM modules. Be sure to include a state diagram with your solution. Include a state diagram and a PS/NS table with your solution. State how the circuit is controlled.

Exp 10: Full-Feature 3-Bit Up/Down Counter

Objectives:

- To gain experience designing Finite State Machines configured as counters
- To gain experience using the HDL behavioral method for implementing FSMs

Somewhat Meaningful Comments: Using FSM design techniques for counters is common digital design pursuit. The counter in this lab activity is a counter that has many potentially useful features. As you will see from the specification below, this FSM is not completely specified. You the designer thus have the choice of how to handle cases that are not completely specified. Your mission is to make it work as specified and to fill in the gaps in an intelligent manner. This is typical of a real-world design experience where someone who know a lot less than you do (namely, the management) has specified the design.

Assignment: Design a FSM that implements a synchronous 3-bit up/down counter with the attributes listed in Table 9. Visually test your counter using the circuit shown in Figure 6; choose about 2Hz for the FSM clock. The possible count values for this counter are: 0, 1, 2, 3, 4, 5, 6, 7; this counter naturally rolls over (7→0) and rolls under (0→7).

- Use the provided FSM template as a starting point for the FSM
- Make the **RESET** signal asynchronous. Put asynchronous control signals in the code that models the FSM's registers; do not place it in the combinatorial portion of the circuit.
- If neither **EVEN** nor **ODD** are asserted, the counter counts in a normal count sequence in the direction determined by the **UP** control input.
- Your state diagram must have eight states
- You can display the output using a single bit or the univ_sseg.v module

Control Input	Comment	Attribute	I/O Map
RESET	Clears counter (count = 0)	asynchronous	left-most Button
UP	direction of count sequence (UP = '1' up; UP= '0' down)	synchronous	SW15
EVEN	counts in an even sequence according to UP direction when asserted	synchronous	SW0
ODD	counts in an odd sequence according to UP direction when asserted	synchronous	SW1
HOLD	prevents count from changing	synchronous	right-most button

Table 9: Attributes for the up/down counter.

Suggestions: Here are a few items that will expedite your path to success:

- When drawing your state diagram, use the labels **E**, **O**, and **H** for signals **EVEN**, **ODD**, and **HOLD**. This state diagram becomes rather busy; using these labels will help it readable.
- Plan on using an entire sheet of paper for your state diagram. The state diagram becomes busy; once again, your mission is to keep it readable.

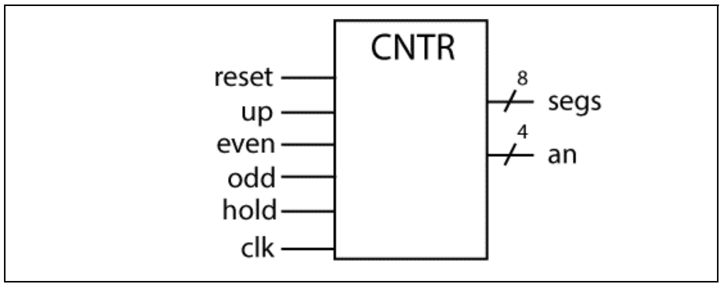


Figure 6: The top-level circuit for testing the up/down counter.

Constraints:

- Don't use math operators in your FSM
- Your state diagram modeling the FSM must have eight states
- You must model your FSM using the behavioral modeling template. This model has two procedural blocks; one block is combinatorial and the other block is sequential.

Deliverables:

1. HDL models for all modules you wrote in this lab activity
2. Black-box models for the circuits you modeled in this lab activity
3. State diagram for FSM
4. Answers to the "questions"
5. Solution for design problems

Questions:

1. From your state diagram, what would happen if both the ODD and EVEN input were simultaneously asserted?
2. From your HDL model, what would happen if any other input is asserted at the same time as the RESET input?
3. Briefly describe the symmetry present in both the state diagram and HDL model of this FSM.
4. Self-correcting hardware is a great feature in digital design. Briefly describe what the term "illegal state recovery" means and the specific condition that it attempts to avoid.
5. Did the FSM model you used for the circuit in this lab activity use some form of illegal state recovery? Briefly explain.
6. Briefly state why it is that state diagrams never includes clock signals.
7. If someone told you they encoded their FSM using eight flip-flops, would you know how many unique states were in their state diagram? Briefly explain.
8. Write a closed form formula that relates the maximum number of unique states in a sequential circuit to the number of 1-bit storage elements in the sequential circuit.
9. Write a closed form formula that shows how many storage elements (n) would need to implement a circuit requiring Q unique states. This formula should include a floor or ceiling function.

Design Problems:

1. Design a counter that continuously counts in the following sequence: {2, 5, 7, 1, 4}.

   - The counter has a hold input that prevents the counter from counting. The counter's output shows the sequence as given when the hold input is not asserted (positive logic), or negative values of the sequence when the hold input is asserted. In other words, the hold input pauses the counter and causes the output to display a negative version (RC format) of the current count.
   - Make the design self-correcting by sending any unused states to state associated with count 7. The counter's output for unused states is zero.

   Provide a high-level BBD, a PS/NS table, and a state diagram for this problem. Minimize your use of hardware.

Exp 11: Sequence Detector FSMs

Objectives:

- To design and implement a FSM that you can use for something relatively useful
- To gain more experience using the HDL behavioral method for implementing FSMs

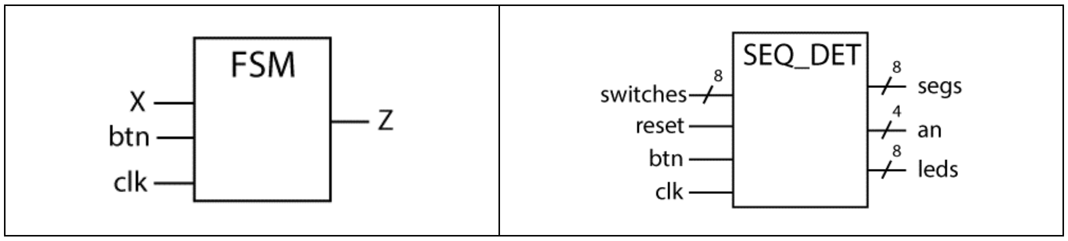
Somewhat Meaningful Comments: The design of sequence detectors using FSMs is one of the more basic uses for a FSM. The designs are not overly complicated yet they provided immediate gratification to FSM designers. In order to convince you that FSMs really work and they're actually on the cool side, you'll be implementing a FSM in this lab activity that acts as a sequence detector.

Assignment: Design a sequence detector that will detect the provided sequences according to the button press information below. Make your FSM a Moore-type FSM the resets when the one of the desired sequences are found. Drop your FSM design into the circuit shown in Figure 7. Here are the sequences:

If BT0 is pressed:	1 0 0 0 1 0
If BT0 is not pressed:	1 0 0 1 1 0

For this design, the SEQ_DET module of Figure 7(b) is provided for you, but is missing the FSM. Figure 7(a) shows the BBD for the FSM you need in this experiment. The circuitry of the SEQ_DET module provides the values on the **switches** input one bit at a time; the **leds** output shows which bit is currently being input to the FSM.

Constraints: only use one FSM in your design. Be sure to properly label all aspects of the associated state diagram.



(a) **(b)**

Figure 7: The FSM BBD (a) and the top-level BBD (b) for this experiment.

Deliverables:

1. HDL models for all modules you wrote in this lab activity
2. Top-level BBD for your model
3. State diagram for FSM
4. Answers to the "questions"
5. Solution for design problem(s)

Questions:

1. Describe an application where sequence detectors could potentially be useful.
2. Given two FSMs that perform the same basic function, a Mealy-type model typically has fewer states than Moore-type models. Briefly describe why that is generally the case.
3. Mealy-type FSM are able to "react" more quickly than Moore-type FSMs. Briefly describe what characteristic of a Mealy-type FSM makes this the case.
4. In the previous question, briefly describe what exactly the term "react" refers to?
5. Slower clock speeds of any circuit are generally considered better because they save power. Briefly describe the main factor in deciding the minimum clock speed for your FSM and still have it do the job you need it to do.
6. Provide a Mealy-type state diagram for the resetting version of the FSM you designed in this lab activity.
7. Show one approach (draw a diagram) to change an XOR gate into an inverter. Include a truth table that clearly shows the foundation of this approach.
8. Briefly describe why an XOR gate is not functionally complete.
9. Show two approaches (draw a diagram) of how you can change a NAND gate into an inverter.
10. Show two approaches (draw a diagram) of how you can change a NOR gate into an inverter.

Design Problems:

Design a circuit that indicates when it finds one and only one of the two following sequences "01101" and "10110" on a single serial input. The most straight-forward approach is to use two FSMs in your; be sure to provided state diagrams for any FSM you use in your design. Assume the serial input does not change more than once per clock cycle. Minimize your use of hardware in your design. State how the circuit is controlled.

Exp 12: Four 4-Bit Number Sort

Objectives:

- To design, synthesize, and test a non-trivial circuit

Somewhat Meaningful Comments: Once again, sorting numbers makes for simple yet interesting digital design problems. This experiment involves the design of a circuit that solves a simple sort problem. There are many approaches to sorting numbers; the easiest approach is probably a bubble sort, particularly if you're going to implement the algorithm in hardware. Feel free to use any sort you want for this experiment, but keep in mind the overall limit to the number of states in your solution.

Assignment: Design a circuit that sorts four numbers in ascending order and displays the sorted values on the four 7-segment displays on the development board. The four inputs are 4-bit unsigned binary numbers; the 16 switches on the dev board provides these inputs. This circuit also has an LED that indicates when the sort algorithm is happening (LED off), and when the sort algorithm has completed (LED on).

- Use a button press to start the sort
- Check for the button release after the sort is completed, which means if the user holds the button down, the hardware will not initiate another sort until the user releases the button.
- Use any clock speed you want, but make sure you can see the LED turn on and off as the hardware implements the sort algorithm.
- Display the values as hex numbers

Constraints:

- Don't use more than 18 states in your solution's FSM
- Your FSM must never input any data value that the hardware is sorting
- Your FSM must never use mathematical or equality operators
- Use only digital design foundation modules in your design
- Don't use more than 14 total modules in your design
- Minimize your use of hardware in your design
- You must use your own display multiplexor

Deliverables:

1. HDL models for all modules you wrote in this lab activity
2. Black-box models for the circuits you modeled in this lab activity (top two levels)
3. State diagram for the FSM
4. Answers to the "questions"
5. Solution for design problems

Questions:

1. In your own words, provide a complete written description of how the circuit you designed in this experiment operates. Be sure to reference the block diagram for your circuit. This description should be more than a detailed description of the state diagram.
2. How many clock cycles did your circuit require to complete the sort? Your answer to this question should address best and worst case scenarios (if your approach had both scenarios).
3. Relating to the previous problem, if the system clock frequency was 100MHz, how much time was required to complete the complete sort operation for the best and least case scenarios. Show your calculations for this problem. Assume your circuit is not using a clock divider. Don't even think of not writing your answer using engineering notation.
4. There are many sort algorithms out there. State two reasons why you would choose one over another. For this problem, consider your task is to implement the problems in hardware.
5. Sorting of course can also be done in software. It would be hard to state whether the algorithm you implemented in hardware in this experiment would run faster than the same algorithm implemented in software. Just for kicks, name a few parameters that you would need to consider if you were to make such a comparison.
6. Briefly describe how you would need to change your hardware if your circuit used a switch to decide between sorting the switches in ascending and descending order.
7. HDL allows you to write complex **if** statements (**if** statements containing many conditions) as well as nested if statements. Make a statement about the complexity of an "if" statement and the size of the hardware generated by such statements.
8. Verilog **always** blocks contain "sensitivity lists". If you modeled two MUXes in Verilog that were identical except for the fact that one MUX included the select signals in the sensitivity list, and the other not, would the hardware generated by these two MUXes be identical? Briefly explain.
9. Briefly state why was one of the design constraints in this experiment to not input any data being sorted into the FSM.

Design Problems:

1. Design a circuit that has four 8-bit data inputs and one 8-bit output (each are unsigned binary numbers). After a button is pressed, the circuit processes then outputs the largest of the circuit's four inputs; the value remains on the circuit outputs. The output can only change if the button is pressed again. The button must be released before the circuit finds a new maximum value. The circuit ignores button presses after the initial button press until it finds the maximum value. Assume the circuit inputs do not change after the initial button press. Minimize your use of hardware in this design. Use no more than one comparator in your design. State how the circuit is controlled.

Exp 13: Serial Parity Generator

Objectives:

- To learn about one of the most common and useful digital circuit elements: the shift register
- To learn about the concept of parity in a digital circuit

Somewhat Meaningful Comments: Parity generation is quite useful out there in digital-land. But what's really cool about it is that it presents a great opportunity to create somewhat meaningful circuits using basic circuit elements without the circuit being too complex. The notion of parity is associated with a set of bits, which can be a serial stream of bits, or a parallel configuration of bits.

The shift register is a common circuit out there in digital-land simply because they are so versatile. There are several ways to generate parity, but for this experiment you need to use a shift register. Because shift registers are so useful, you generally see shift register referred to as "universal shift registers", which means that they both shift left & right, hold, rotate left & right, etc.

Assignment: Design a circuit that has two outputs. One output is the number of bits set in a 16-bit word (a stone-age binary to decimal conversion), and the other is an output indicates the parity of the 16-bit word. Use the 16 switches on the development board as the input word. Use two 7-segment display devices to display the result of the stone-age binary to decimal conversion. Use the other two unused 7-segment displays to indicate parity with an "EE" for even parity and "oo" for odd parity.

- When the user presses a button, the circuit calculates the parity and displays the results according to the previous paragraph. It then waits for another button press.
- You can have a high clock rate for this experiment (too fast to see) or have it run slow. Having a slow clock helps with debugging (in case you need to debug).

HINTS: There are many approaches for multiplexing the displays in this design. For this experiment, consider modifying your 7-segment display decoder from the previous experiment to include outputs that actuate and "E" and an "o".

Constraints:

- You must use the provided n-bit shift register (don't modify it) for this design
- Don't attempt to use the hardware version of parity generation
- You must use FSM in your design to control the shift register
- You must use your own display multiplexor
- Minimize your use of hardware in your design.
- Do not use more than five states in FSM for your design.
- Do not use mathematical operators anywhere other than provided modules

Deliverables:

1. HDL models for all modules you wrote in this lab activity
2. Black-box models for the circuits you modeled in this lab activity (top two levels)
3. State diagram for FSM
4. Answers to the "questions"
5. Solution for design problems

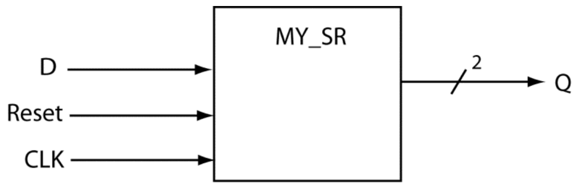
Questions:

1. Briefly speculate on why this experiment had you use the iterative version of parity detection.
2. Briefly describe why the pros and cons of having two ways to calculate parity for a given set of bits. In other words, describe why we have both a "human version" (associated with methods that count the number of set bits) and a "hardware version"?
3. Briefly describe the main purpose of a parity generation and detection.
4. Shift registers are synchronous circuits. Describe what limits how fast you can shift a simple shift register and still have the device operate properly.
5. Another standard function performed by shift register is "rotation", such as rotate right and rotate left. Briefly describe the changes you would need to do to the shift register in this lab activity if you wanted the shift register to only rotate left.
6. One of the great selling points of many computer-type devices is that they do exclusively integer-based math as opposed to using floating point math. Why would this be a good selling point for hardware? Briefly but completely support your answer.
7. If a shift register was used with signed binary numbers (RC), briefly describe what you would have to do when you right-shift a number. For this problem, assume the shifted number result is always valid.
8. A shift-right operation officially performs truncation on the value in the shift register. Briefly explain what this means in the context of a shift register that shifts right.
9. Show a diagram indicating how you would connect a shift register to obtain the following pattern on the shift register's storage elements (example shows output for a four-bit shift register). Make sure your shift register has a way of getting into the first state listed below. Don't use a FSM in your design; minimize your use of hardware in your design. Assume that the circuit starts at the 0000 value.

```
0000 – 0001 – 0011 – 0111 – 1111 – 1110 – 1100 – 1000 – 0000 – 0001 …
```

<u>Design Problems:</u>

1. Design a 16-bit parity checker using only 4-input 1-output decoders.
2. Provide a state diagram that models the operation of a special 2-bit left shifting shift register. For this design, the shift register should ***never*** have the same output for more than three clock cycles. Avoid this condition by directing potential violations of this condition to state Q="10". *Minimize the number of states in your design. Don't use a counter in this design.*
   - The D input represents the value that is synchronously shifted into the shift register
   - The Reset signal is an asynchronous input that places the input into the "11" state
   - The Q output is the value of the shift register



Exp 14: FSM-Based 5-Bit Multiplier

Objectives:

- To design an relatively useful circuit.
- To delve into the lower-level details of FSM design and implementations.

Somewhat Meaningful Comments: This lab activity shows that FSMs combined with standard digital models can be configured to produce a meaningful and useful result, all under the resource constraints of the development board. Surprisingly enough, mathematical operations are quite useful in real-world applications. Though we've done a lot of applications using some sort of adder, we have limited ourselves to either addition or subtraction operations. This lab activity is somewhat different in that we'll implement a multiply operation while still using our friendly RCA. There are many approaches to performing multiplication; this lab represents only one of them. We can model multiplication as a series of shifts and additions; this approach is effectively the long multiplication you learned in third grade.

Figure 8 shows an example of a paper-based 3-bit multiply operation, which shows that you can model this operation as three addition operations. The general algorithm is "shift then add". The item you're shifting (a left shift) is the A operand (op_A). You're always adding something; the number you're adding is based on the value of a bit in the B operand (op_B). Specifically, if the B operand bit is a '0', you add zero to your result; otherwise, you add the current shifted value to the result. You'll be "accumulating" your results as you progress through the algorithm, which means that you'll need an "accumulator" to store the intermediate addition values (as you progress through the algorithm) as well as the result (when the algorithm is complete). You'll use a FSM to control the "stepping" through the algorithm

Figure 8 shows the functionality your circuit will have: there are three basic steps to this 3-bit multiply operation, which we indicate with the circled numbers. Step 1 is based on a "0 * 111" multiplication; Steps 2 & 3 are based on "1 * 111" operations. Your mission in this lab activity is to implement this functionality with the digital modules under the control of a FSM. The algorithm is relatively simple, which is no surprise as you've been multiplying numbers using this algorithm since the third grade. The challenge in this lab activity is implementing the algorithm using a digital circuit.

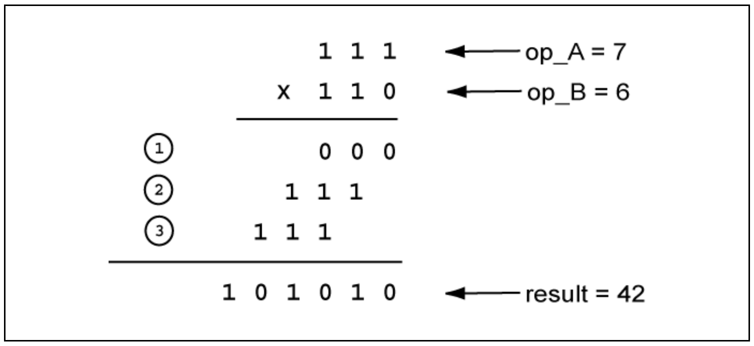


Figure 8: An example of binary multiplication of two 3-bit values.

Assignment: Design a 5-bit multiplier driven that uses successive additions in conjunction with appropriate shifting and accumulate operations to generate a 5-bit multiply operation.

- Use the switches on the development board to input two 5-bit unsigned binary values. These values are multiplied and the seven-segment displays show the result.
- The multiply operation is initiated when you press the development board's left-most button. The final result should remain on the output display until the button is pressed again.
- Ignore all button presses between the initial button press and obtaining the final result.
- Use the univ_sseg.v module to display your results and use a clock division module to slow down the circuit operation for added visual benefit.
- Strongly consider using the LEDs on the development board to indicate the various states you use in your design. Specifically, one LED should be lit for each different state in your design. This will help you debug your circuit if you're so inclined not to use the simulator.

Constraints:

- Use only digital design foundation modules in your design plus whatever extra logic need; don't use more than ten foundation modules in your design.
- Use a FSM to control the overall operation of the circuit.
- Minimize the number of states in your design; use no more than eight states.
- Minimize your use of hardware in your design.
- Do not use mathematical operators anywhere other than the counters or RCAs.

Hints:

- Use a register to store the results that your circuit sends to the univ_sseg.v module.
- It's generally bad practice to input data to the FSM, but feel free to do so for this circuit.

Deliverables:

1. HDL models for all modules you wrote in this lab activity
2. Black-box models for the circuits you modeled in this lab activity
3. State diagram for FSM
4. Answers to the "questions"
5. Solution for design problems

Questions:

1. How would you approach designing a circuit if one of the design constraints was to make the circuit invisible?
2. What are the four operations you can perform on a single bit?
3. Write a closed form formula that relates the maximum value a result can be from multiplying two n-bit numbers. Show your calculations for this problem.
4. Attack of the SAT questions: State diagrams are to FSMs such as "X's" are to circuits. For this question, what does "X" refer to?
5. Another attack of the SAT questions: State diagrams are to FSMs such as "X's" are to computer programming code. For this question, what does "X" refer to?
6. One way to perform multiplication is to add continually. Which approach to multiplication would be faster: the continual adding approach or the approach you used in this lab activity? Answer for the best case, worst case, and average case. Briefly but complete describe and support your answer.
7. Based on the answer to the previous problem, approximately how much slower would it be to do a multiplication using the continuous addition method as opposed to the approach you used in this lab activity? For this problem, assume you're multiplying two 8-bit values.
8. Based on you FSM implementation in this lab activity and a 100MHz system clock, how long will it take to generate the results of your 4-bit multiply operation after the button is pressed? Support your answer with actual calculations.
9. So, I ask my higher-level computer programming language to execute an unsigned integer multiplication algorithm (meaning, I use the "*" operator in an expression), and it does it for me. Is there a way to know what algorithm the computer uses to perform that operation? Briefly but completely explain.
10. List the four types of information provided by a state diagram.

Design Problems:

1. Using only one RCA, one register, and one shift register, design a circuit that could effectively multiply a given 16-bit unsigned number by 6.5. Also, provide a state diagram describing an FSM that would drive your circuit. Consider the multiply operation to start when a FSM detects a button press; after that, the algorithm continues independently of button presses until the algorithm completes. When the algorithm completes, the FSM verifies the button is released before waiting for another button press. For this problem, consider the output persistent after the algorithm completes. Minimize your use of hardware in this design. State how the circuit is controlled.

Exp 15: FSM-Based 6-Bit Divider

Objectives:

- To design yet another actual relatively useful circuit.
- To delve into the lower-level details of FSM design and implementations.

Somewhat Meaningful Comments: This lab activity shows that FSMs combined with standard digital models can be configured to produce a meaningful and useful result, and once again all under the resource constraints of the development board. This is yet another circuit that implements a standard mathematical operation using standard digital modules under control of an FSM. This lab activity is similar to a previous lab activity where we generated a multiplier using a similar algorithm. Once again, note that there are many approaches to performing integer division; this lab represents one of them.

One approach to performing division is to model it as a series of shifts and subtractions. Figure 9 shows such an algorithm for division based on binary numbers. While the algorithm seems scary at first, you should notice that it is the same algorithm you used when you did "long[1] division" in elementary school. Figure 10 shows an example of an application of the algorithm in Figure 9 using a 6-bit unsigned binary number. You should be able to tell by examining one of these figures that the algorithm is a series of comparisons, left & right shifts, and subtractions, which are operations you're intimately familiar with. When the algorithm finishes, you have both the quotient and remained in two different registers. A major difference between this algorithm and the multiplication algorithm is that we "create" the result (the quotient) using a shift register rather than an accumulator.

```
A = dividend;  A = (n + 1) bits
B = divisor
Q = quotient
R = remainder

Assumptions: B will never be zero; B will never be greater than A

Initialization:    Zero extend the dividend
                   Multiply B (divisor) by 2^n to form B'


       for (x=0; x<n+1; x++)
       {
          Step 1:
              if (A ≥ B')  then
              {
                  Q(n-x) = '1'
                  A = A – B'
              }
              else
              {
                  Q(n-x) = '0'
              }

          Step 2:  B' = B' ÷ 2
       }

Completion:         Q = quotient       A = remainder
```

Figure 9: The shift-based division algorithm.

[1] It's called "long division" because it took way too long to do and actually obtain the correct answer.

Figure 10 shows an example of an application of the division algorithm; here are some highlights:

- The algorithm is iterative (M iterations were M is the bit-width of the dividend).
- The algorithm performs a subtraction only when the current value of A is greater than or equal to B' (the modified divisor)
- The quotient is nicely implemented as a left-shifting shift register

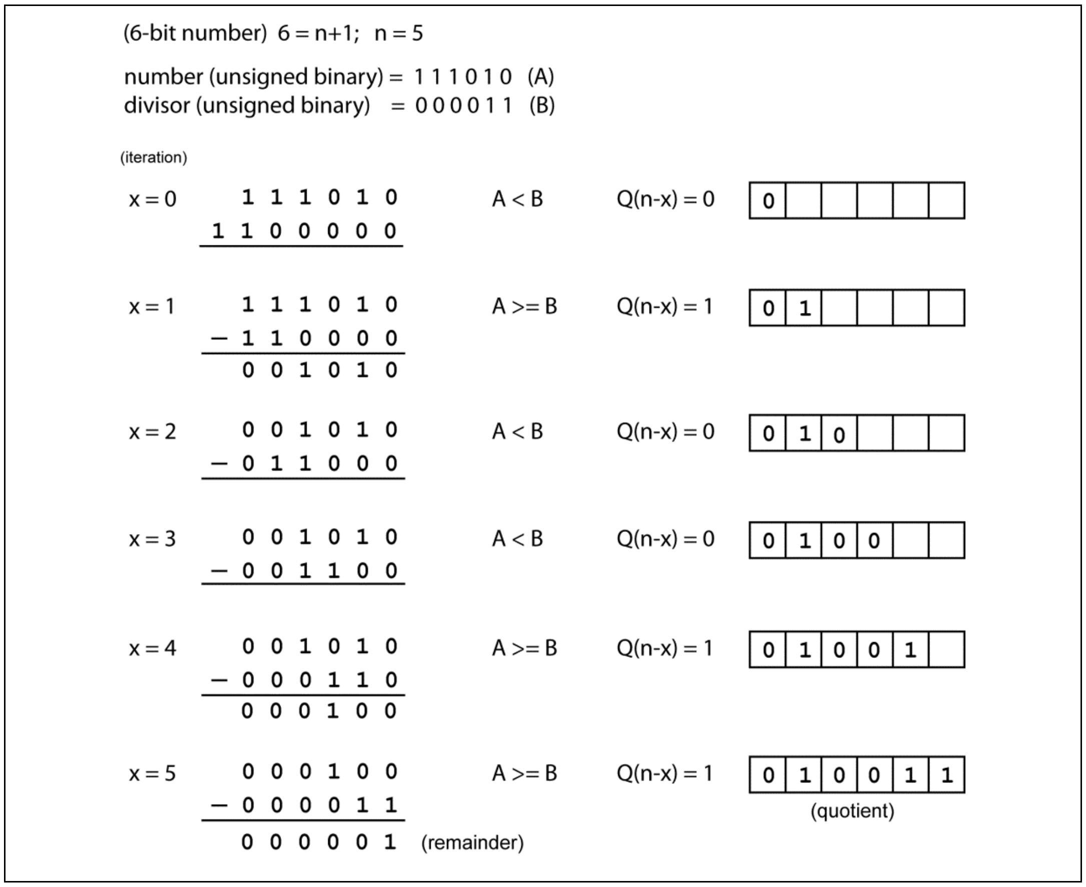

(6-bit number) 6 = n+1; n = 5
number (unsigned binary) = 1 1 1 0 1 0 (A)
divisor (unsigned binary) = 0 0 0 0 1 1 (B)
(iteration)
x = 0 1 1 1 0 1 0 / 1 1 0 0 0 0 0 A < B Q(n-x) = 0 [0]
x = 1 1 1 1 0 1 0 − 1 1 0 0 0 0 = 0 0 1 0 1 0 A >= B Q(n-x) = 1 [0 1]
x = 2 0 0 1 0 1 0 − 0 1 1 0 0 0 A < B Q(n-x) = 0 [0 1 0]
x = 3 0 0 1 0 1 0 − 0 0 1 1 0 0 A < B Q(n-x) = 0 [0 1 0 0]
x = 4 0 0 1 0 1 0 − 0 0 0 1 1 0 = 0 0 0 1 0 0 A >= B Q(n-x) = 1 [0 1 0 0 1]
x = 5 0 0 0 1 0 0 − 0 0 0 0 1 1 = 0 0 0 0 0 1 (remainder) A >= B Q(n-x) = 1 [0 1 0 0 1 1]
(quotient)

Figure 10: A shift-based division algorithm.

Assignment: Design a 6-bit divider using the algorithm in Figure 9. The circuit does this calculation upon the pressing the leftmost button on the development board. Output the quotient and remainder on the 7-segment display device.

- You can assume the divisor is never zero and always less than the dividend.
- You need to use the universal_sseg_sed module allows you to display to simultaneously display two different counts.
- You're going to need some registers, but only use universal shift registers for your register needs. The lab activity provides you with a model of a "USR", so don't spend time designing one from scratch. The provided 12-bit shift register is sufficient for this lab activity without modification.
- When you input the dividend and divisor into your circuit, do the 0-bit stuffing (the dividend) and the left-shifting (the divisor) using simple assignment statements to the register. Do not attempt to shift the values accordingly using hardware. See the algorithm for full details.

- You'll need a subtractor circuit for this lab activity; this lab activity provides a 12-bit subtractor to for you so you don't need to design it yourself.
- Design a FSM to control the overall operation of the circuit.
- Use only standard modules in your design plus whatever extra logic need; avoid using HDL to create magic circuits.
- Use the switches on the development board to input two 6-bit unsigned binary values (the dividend and the divisor). These values are multiplied and the seven-segment displays show both the quotient and remainder.
- The divide operation initiates when you press the development board's left-most button (consider this a GO signal). The final result remains on the output display until the button is pressed again.
- Ignore all button presses between the initial button press and obtaining the final result.
- Use a clock division module to slow down the circuit operation for added visual benefit and debugging help.
- Strongly consider using the LEDs on the development board to indicate the various states you use in your design. Specifically, one LED should be lit for each different state in your design. This will help you debug your circuit if you're so inclined not to use the simulator.
- Minimize the number of states in your design; keep them under 10. You're welcome to make a separate state for each phase of the algorithm (but realize you could include a counter in this circuit that counted the phases of the algorithm and thus allow it to use fewer states).
- Minimize your use of hardware in your design.

Hints:

- The initial conditions of this algorithm require you to use registers greater than 6-bits. The lab activity provides you with several 12-bit circuits, which are wide enough to complete this lab activity without making modifications to those circuits.
- This lab activity is somewhat open-ended; use this lack of implementation details as an opportunity to be creative with your implementation, but minimize your use of hardware and states in your FSM.

Deliverables:

1. HDL models for all modules you wrote in this lab activity
2. Black-box models for the circuits you modeled in this lab activity
3. State diagram for FSM
4. Answers to the "questions"
5. Solution for design problems

Questions:

1. One way to perform integer "division" is by multiple subtractions, where the quotient is the number of times you can subtract the divisor from the dividend without it underflowing its given range. Briefly but completely describe whether this algorithm or the algorithm you used in this lab activity be faster; make sure you answer includes "why" one would be faster than the other. Answer for the best case, worst case, and average case quotient and divisor.

2. Considering the previous question again. Since different division algorithms have different running issues, briefly describe at least two seriously important issues that you'll need to consider if you're in charge of choosing a particular algorithm for your company's new computer architecture.
3. If your circuit did not check for a zero divisor, briefly describe how your circuit would respond if it did receive a divisor of zero and was allowed to execute the algorithm.
4. You were allowed to use a FSM state per algorithm iteration for this circuit, which made the circuit smaller, but also less "desirable". Briefly describe how you would modify your circuit if you did not use a state per iteration. Also briefly, describe the main advantage of not using a state per iteration.
5. Briefly describe how you would modify the algorithm you implemented in this lab activity to handle signed binary numbers in RC format.
6. So, I ask my higher-level computer programming language to execute an unsigned integer division algorithm (meaning, I use the "÷" operator in an expression), and it does it for me. Is there a way to know what algorithm is being used to do that division? Briefly but completely explain.
7. I decided to use this algorithm to create a 32-bit unsigned integer division algorithm. But in order to make the algorithm more quickly calculate the result, I first divided the dividend and divisor by 128 (seven right shifts using a barrel shifter). Provide some intelligent commentary on whether my new approach would work or not, and also when this approach would be acceptable.
8. If you wanted to run the circuit you design in this lab activity at the maximum possible rate, what module do you feel would be the limiting factor in bumping up the FSM clock rate? In other words, as you increase the FSM clock rate, which module will be the first to provide unpredictable results? Briefly support your answer with intelligent commentary.
9. If the divisor in this lab activity were an integer power of two, I could do the operation much quicker using some other circuitry. Provide a block diagram and FSM showing the modifications you would need to make to your circuit in this lab activity in order to have your circuit choose the faster algorithm when possible.
10. Could you use the circuit you designed in this lab activity to heat up your cup of coffee? Briefly explain.

Design Problems:

1. Design a circuit that will sort three 8-bit unsigned binary values. The sort begins with the pressing of a button. When the sort is complete, an LED is lit; this LED is turned off as the sort happens. Use no more than three registers in this design. The first step is to latch the three values into the three registers. Provide a state diagram for this design that you could you to control the circuit. Minimize the use of hardware in the circuit; also minimize the use of states in the associated FSM. State how the circuit is controlled.

Exp 16:RAM/ROM Bubble Sort

Objectives:

- To design a useful circuit that requires utilization of ROM and RAM.
- To learn about the world famous "bubble sort" algorithm.

Somewhat Meaningful Comments: Although sorting problems are near and dear to the heart of every computer science person, they also make for interesting problems to complete in hardware. There are a many different sorting algorithms out there, they experiment uses a bubble sort. The bubble sort is not the fastest sort algorithm out there, but it is the most intuitive (one man's opinion), which is why we are using it in this lab activity.

We don't do a lot with ROMs and RAMs in this course, but we'll use them more in a later digital course. The good news is that they're not complicated modules. The even better news is that you don't need to "design" them for this course. We provide you with a template; you then modify the template to fit the needs of the problem you're working on solving.

Assignment: Design a circuit that, upon a button press, implements a bubble sort on the data stored in a 16x8 ROM. The circuit first loads the data to be sorted into a RAM from a ROM; the experiment provides the ROM for you. When the sort is complete, the circuit displays the sorted contents of the RAM continuously on the 7-segment display one RAM location at a time; the sort should be in ascending order. Consider the values in RAM to be unsigned 8-bit values. Figure 11 shows the top-level black box diagram for this circuit. Minimize the amount of hardware you use in your design.

- The circuit initially displays the RAM's contents upon power-up. In other words, the circuit continuously indexes through the RAM in counting order until the user presses the button. When the user presses the left-most button on the development board, the FSM proceeds to write the contents of the ROM to the RAM and then sorts the numbers in the RAM. When the sort is complete, the circuit displays the RAM contents in sorted order.
- Your FSM should include a "display" state that does nothing more than displays the contents of the RAM in a sequential manner. The FSM should return to this state after it sorts the values in the RAM.
- All synchronous circuit operations should be at 6Hz for display purposes; you may want to slow the clock to help with debugging during the development phase of this experiment.
- Display the address input of the RAM to the four right-most LEDs on the development board. In this way, you'll display every RAM access (reading or writing) on the LEDs.
- Use only one RAM in your circuit
- Demonstrate your working circuit to the instructor.

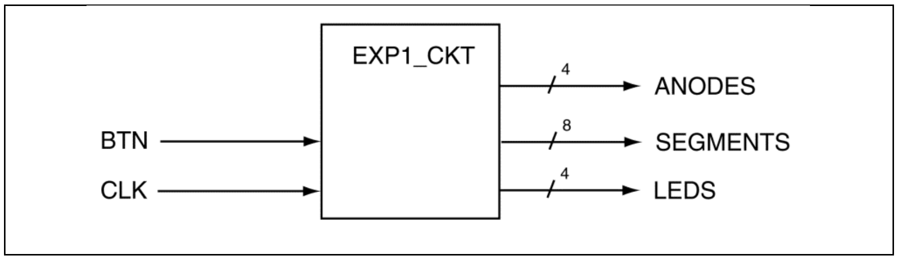


Figure 11: The top-level black box diagram for this experiment.

Constraints:

- Use the same clock frequency for both reading and writing the RAM.
- You need an FSM to control your circuit; don't use more than twelve states in your FSM.
- Only use standard digital modules in your design. Don't use HDL to model non-standard circuits. Standard circuits include registers, counters, MUXes, decoders, etc.
- Don't use more than one RAM in your design.
- Don't use HDL mathematical operators anywhere except in RCA or counter circuits
- Your FSM should be 100% independent of the length of the requested Fibonacci sequence, the size of the RAM for the bit-count circuit, or the number of items in the bubble sort circuit. This means if the size of one of these items changes, you will not need to change the FSM in your design.
- The only memory in your FSM should be the state variables
- The FSM should not respond to another button press until the assigned task has completed and the button is released

Hints:

- There are many ways to implement these designs; whatever way you choose, make it work. Think about it before choosing what you feel is the best approach. Strive to keep your circuit as simple as possible by only using standard digital modules. If you run into issues, you'll need to use the simulator to work through them.
- Ignore all debounce issues associated with the button

Assignment: A ROM contains 16 8-bit unsigned values. When a button is pressed, place those values in a 16x8 RAM in ascending order, and then continuously display the outputs of that RAM.

- The RAM and ROM modules are provided for you; the ROM does not require modification, but you'll need to fix up the dimensions of the RAM.
- Don't use more than ten states for the FSM in your design.
- Have the output display anything you want until the sort is completed. At that point, display the contents of the RAM one value per clock cycle.

Hints:

- Txxx

Deliverables:

1. HDL models for all modules you wrote in this lab activity
2. Black-box models for the circuits you modeled in this lab activity
3. Answers to the "questions"
4. Solution for design problems

Questions:

1. Oxxxxor.

Design Problems:

1. xxxx

Exp 17: Lost HDL Model Analysis and Synthesis

Objectives:

- To analyze, design, and synthesize a non-trivial digital logic circuit

Somewhat Meaningful Comments: It's been a long quarter but you've managed to obtain an impressive amount of digital analysis and digital design tools. Included in your tool chest is the ability to design some non-trivial digital circuitry. For this final lab activity, I need you to provide me with a HDL model that can be used to duplicate the characteristics of the provided .bit file. I generated the program file a long time ago and I've lost the code; please generate a HDL model that can be used to generate this circuit.

Assignment: Play around with the provided circuit. There are only three development board inputs that affect the circuit outputs: SW3, SW2, SW1, and SW0. Analyze how these inputs affect the circuit outputs. After you're sure you know how the four switches affect the circuit outputs, write a HDL model that duplicates the total functionality of the circuit. Figure 12 shows a high-level block diagram of the circuit as it should be implemented on the development board.

Constraints:

- The total number of states in all of the FSMs you use this design should not exceed 15. For this design, only consider the FSM as having state. You'll need other storage modules, but don't include those storage elements in the total state count.

- Your FSM for this lab activity should not contain any status signals (inputs) more than one-bit wide.

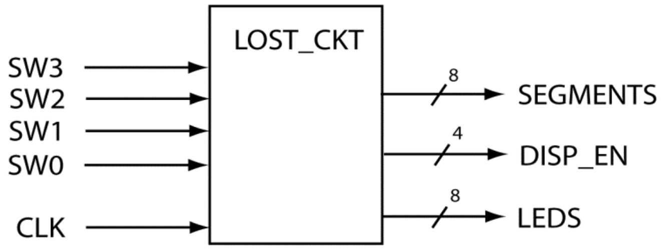


Figure 12: The top-level circuit diagram.

Apology: Yep, blinking LEDs is sort of boring. If the EE Department actually felt that the digital courses were important, we'd sure have a nice set of external models to interface with. And of course, we'd sure have benches that were as well-stocked as the analog labs.

Special Deliverables:

1. None

Questions:

1. By using just HDL behavioral modeling of FSMs, would you think it possible to model a FSM with a soul? Briefly explain.

2. Provide a complete description of your circuit and accompanying FSM. Make this a painfully compete description.

3. Briefly describe why this lab activity limited the number of states in the FSM associated with this lab activity.

4. By now you realize that modeling a counter using the counter template is more straight-forward than modeling a counter using a FSM design approach. However, modeling counters using the template approach has one huge limitation. What is it?

5. Describe a healthy relationship between generating "structured" outputs on the development board and controlling some more meaningful devices out there in the real world.

6. How would you modify the circuit in this lab activity to utilize 16 LEDs rather than 8 LEDs. For this question, assume the LEDs are displaying the "walking" LED pattern and that the other features of the circuit remain the same .

7. Counters often have RCO outputs (ripple carry out), which indicate when the counter has reached its terminal count. Show a circuit that would provide the correct RCO for a 4-bit up/down counter. The notion here is that the terminal count is different depending on whether you're counting up or down.

8. Counters are often used to count "events", such as how often a signal is asserted. Briefly but completely describe how you would configure a typical up-counter to function as an event counter that counted how many times a signal was asserted.

9. One function of a counter's RCO output is to extend the bit-range of fix-width counters. Consider two counters that follow the counter template design but have RCOs. Briefly describe how you would connect two of these "n-bit" counters to double the effective bit width.

10. Provide a Verilog model of a 4-bit decade up counter. Do this by modifying the standard counter template. Make sure your model provides an RCO.

Design Problems:

1. Design an circuit that counts on each of the four 7-segment display devices. The counter counts form 0→15 and outputs the count on the displays using hexadecimal notation. Each number in the count appears on each of the 7-segment display digits for one clock cycle, then moves to the next digit. Once it has display the same number on each of the four 7-segment displays, the count advances to the next display. Once each of the displays has shown the count, the count is incremented and starts at the first sequence again. Include an input that allows the digit displays to go from left to right or right to left. Include a state diagram to control you hardware in this design. Minimize your use of hardware in this design; also minimize the number of states in this design.

Exp 18: Static Logic Hazards (B2Spice Simulator)

Objectives:

- To understand the relationship between device models and circuit simulation
- To gain experience working with the B2 Spice Simulator
- To be able to identify and remove static-logic '1' hazards in function representations

Somewhat Meaningful Comments: Although the world is happy when you reduce an equation and implement the circuit, it is not always the best solution. If your circuit is running really slow, glitches will probably not present a problem. But they're there and let's do an exercise in removing them.

This lab activity also introduces the B2Spice simulator. There is tutorial associated with this simulator; you should definitely step through the circuit presented in that tutorial.

Assignment: When the following function is reduced using standard K-maps, it contains a static logic hazard. Use the following guidelines to both show and remove the glitch in the output circuit.

1) Using the following equation, implement reduced circuit in B2Spice.
   $F1(A,B,C) = \sum(2,3,5,7)$
2) Use the simulator to locate any glitches in the output caused by static logic hazards. Print out both the circuit schematic of the "repaired" circuit and the well-annotated timing diagram associated with the glitching circuit. Fully annotate the timing diagram and explicitly show the cause of the glitch by following the propagation of the input signal that causes the glitch from the circuit input to the circuit output. In order to analyze the entire circuit, provide an output port for every intermediate signal in your circuit.

3) Add a static logic hazard cover term to the circuit schematic in order to remove the static logic hazard. For this step, you'll need to use two 2-input OR gates to simulate a 3-input OR gate.

4) Simulate the circuit and verify that the circuit is glitch-free (don't print this timing diagram).

5) Repeat the previous four steps for a the following function: $F(A,B,C) = \prod(0,2,4,5)$. Note that while you used an SOP approach for the previous problem, you'll be using a POS approach for this problem. This problem will also have a glitch; make sure you can find it.

Special Deliverables:

1. Well-annotated circuit timing diagram clearly showing glitch in circuit outputs for both the positive and negative pulse.

2. The schematic diagrams associated with the glitching circuits and the repaired circuit.

Questions:

1. Does the fact that your circuit contains a static logic hazard guarantee there is going to be a glitch somewhere in the output? Briefly explain.

2. I decided to forego the static logic hazard problem by not reducing my functions and implementing them instead as standard SOP and POS forms. Briefly explain whether this approach will successfully avoid static logic hazards.

3. Show a K-map for a 4-variable function that contains four static logic hazards when maximally reduced.

4. In your own words, carefully explain how the inclusion of the "cover term" removed the static logic hazard. Briefly but completely explain your answer.

5. Would it be possible to have static logic hazards associated with functions that used XOR gates? Briefly but completely explain your answer.

6. Why are static logic hazards generally less of a problem if the required operating speed of your circuit is relatively low? Briefly explain.

7. What attribute of the circuits you worked with in this lab activity actually caused the glitches.

8. Based on the previous question, if you changed the switching characteristics of the devices in a circuit, would it be possible to remove glitches caused by static logic hazards without adding a cover term? Briefly explain your answer.

9. We all know K-maps are almost worthless based on their limited ability to handle more than four input variables. Does this lab activity actually put K-maps back in the "somewhat useful" realm? Briefly explain your answer.

10. One of the points of this lab activity is that there is a trade-off in implementing digital circuits. What is this trade-off. HINT: it is based on how you "fixed" your glitching issues.

Design Problems:

1. Design a 2:1 MUX that you know for sure is glitch-free. Be sure to include all the pertinent steps.

Troubleshooter's Guide to Vivado

Vivado Golden Rules

- Make sure there are no spaces in the path name to your project. If there are spaces, any one of a number of problems may arise that will prevent you from completing your project.

- Make sure you do not save project in the folder that contains the Xilinx software. You should always create a special folder for yourself away from system software areas.

XDC File Issues

- Make sure the case of signals in your XDC file match the case of the signals in the entity.

- Make sure you preserve the white space in the XDC file when you're modifying it to support your project.

- Make sure bundled signals use square brackets in the XDC file even though the HDL file uses something else.

Known Quirks and Common Errors

- Module names must not start with a numeric character (Verilog)

- Module names must not contain hyphens (Verilog)

- Module names must not contain spaces (Verilog)

- All dealings with Xilinx should be done on the local drive. Completed projects should be saved to your own personal storage device such as a USB drive. Leaving files on a lab computer is extremely problematic is you ever hope to see your work again.

- Working off a USB drive can be done without problems but it is much slower than working from the host computer's hard drive.

Lab Grading Acronyms

√	**Check Mark** – Minimum expectations were met.
ANN	**Annotations** – A timing diagram was missing or did not have proper annotations.
BE	**Better English**: Indicates that the correct information is present, but you should strive to write better English.
BV	**Bad Signal/Variable Labels**: Be sure to use self-commenting labels for all signal and variable names. This includes both Verilog and assembly language programs.
CAP	**Caption** – A figure diagram or table did not have a proper caption or title. Each figure should have a brief stand-alone description.
G	**Good** – This indicates something was done better than minimum requirements.
IND	**Indentation Problems**: Source code (Verilog, assembly language) requires proper indentation
MC	**Missing Comments** – Source code (HDL, assembly language) should be well commented; comments are either missing or have some other problem.
MH	**Missing Header** – All source code (HDL, assembly language) should have a file header with information describing what is found in the file. This comment indicates that the header is missing or does not contain the proper information.
MP	**Missing Point** – Generally associated with conclusion but also could be associated with objectives. This comment generally means that the conclusion did not state if lab activity's objectives were met or the conclusion simply missed the point of the lab activity.
MQ	**Museum Quality** – When source code goes beyond all the rules of neatness and clarity, you get assigned this acronym. You know you've done a good job.
MS	**More Space** – Don't try to cram stuff in. Use more space between sections, figures, etc. to give the report an overall neat appearance. This also includes the insertion of white space in HDLmodels and assembly language programs.
MW	**Missing Work** – There was a calculation or derivation that you should have shown more explicitly. All non-trivial calculations and derivations should be included in your report.
NC	**Not Clear** – This can refer to anything but most often refers to descriptions of things or answers to questions. In this case, the answer is *probably* correct, but I'm not 100% sure.
NR	**Not Readable** – This can refer to anything but most often refers to hand-written stuff (try to be neat) or cut-and-pasted diagrams from other sources.
OK	**Oh Kay** – Your answer was adequate but could have been more inclusive or more clear.
RG	**Re-read Lab Report Guidelines** – If your report has too many problems, your best bet will be to go back and read the lab report guidelines.
REF	**Missing References** – All diagrams/figures in should be referenced from the body of the report.
SIG	**Signal Consistency** – Signal names on all models (timing diagrams, HDL models, circuit diagrams) should match.
TT	**Title** – Lab activity has problems with title or information in title such as course, section number, names of people in group, etc.
TSH	**Too Short** – This can apply to anything including conclusions, descriptions, objectives, testing procedures, etc.
TSM	**Too Small** – This can refer to anything. Make everything, particularly diagrams, large enough to be read by the average human.
WC	**Weak Conclusion** – This refers to the overall quality of the conclusion. The conclusion can be weak for many reasons; refer to the Lab Report document for an overview of what is expected in the conclusion.

Verilog Style File

The main goal of your Verilog source code is to model a digital circuit. This means that your only mission is to satisfy the Verilog synthesizer. But in reality, you and other humans are going to need to read and understand your Verilog source code. The single most important factor in creating readable and understandable Verilog source code is to make sure your Verilog source code follows certain appearance guidelines. The file listed below shows some more of the more important attributes and rules that all Verilog source code should adhere to. The overriding factor with your Verilog source code is to make it neat, organized, and readable; any specific items not listed in this style files should adhere to these principles.

Verilog Testbenches

<u>Learning Objectives</u>

1. Verilog modeling
    - To learn the basic of writing Verilog test benches
2. ISim
    - To learn to simulate your circuits using the simulator
    - To get a feel for some of the power and features of the simulator

<u>Introduction:</u> Testbenches are an important part of HDL modeling. The main issue is that the synthesizer has the ability to interpret your HDL models in ways you could not anticipate. One of the main themes of this text is to keep your models intended for synthesis as simple as possible in order to give the synthesizer as few options as possible (as few knobs as possible). As circuits become larger, it becomes harder to keep the HDL models simple, and you must work with the synthesizer.

Working with the synthesizer is a two-step process. First you must have a basic understanding of how the synthesizer operates. Second, you must always test the circuit the synthesizer provides for you. In other words, designing a circuit is half the battle, testing the circuit is the other half[2].

Designers know that many of the constructs in Verilog language do not synthesize into digital hardware. This may sound strange, but it underscores the fact that circuit verification is a significant part of the design process. There are many interesting constructs in Verilog designed specifically for verification; this text uses only a few of them. Circuit verification is part art and part science; it's a truly in-depth topic. The science portion of verification primarily involves writing models with as much coverage as possible for the circuit being tested; the other part involves creating generic and automatic test models that run to completion and state directly whether your circuit works as expected. In academia, the main focus of courses that use HDLs are the design and generation of circuits; the testing portion of circuit design is unfortunately highly attenuated due to time constraints.

<u>Hardware Modeling vs. Hardware Verification:</u> Any writing you find out there regarding HDLs always places a strong emphasis on the fact that using HDLs is inherently different from writing software. Moreover, such writing rightfully claims that if you use the HDL's syntax in a manner similar to writing software, your circuit has less chance of working. But here we are in the chapter presenting testbenches. It turns out that a significant amount of the constructs in any HDL are "not synthesizable". So what are they there for? They're there to help you verify the HDL code you write to model a circuit.

This text does not go deeply into verification due to time constraints associated with typical introductory digital design courses. Because of this, you won't get a good feel for the "art" of writing testbenches. Here's the funny issue… you tossed out your programming skills to become a person skilled at modeling digital circuit. If your job is to verify synthesized HDL models, you need to pick back up those computer programming skills, as writing HDL code for verification is more like writing software than it is modeling hardware. Once again, this text does not go there; but be prepared if your instructor of boss needs you to do some viable verification.

Testbenches: Test benches range from quite simple to massively complex depending on their intended purpose; the test bench can sometimes become more complicated than the actual circuit you are testing. Figure 13: The general model of an HDL testbench. shows a general model of a testbench. The test bench comprises of two main components: the *stimulus driver* and the *design under test*. The design under test, or *DUT*, is the HDL model you intend on testing. The stimulus driver is a HDL model that

[2] It's probably more than half the battle in real life; it's usually less than half the battle when first learning to model circuits with HDLs.

communicates with the DUT. In the general case, the stimulus driver provides test vectors to the DUT and also examines results. The stimulus driver can also interact with the external environment, which allows for reading test vectors from files and writing various data and status notes to files. In this text, our stimulus drives only provide information to the DUT; the DUT does not provide status back to the stimulus driver.

The stimulus driver is nothing special in terms of a HDL model. The main difference here is that instead of dealing with signals that interface with the outside world (such a switches and LEDs), we're now dealing with signals that are driving the unit we intend to test. Note that in Figure 13 there are no signals touching the dotted line, therefore the testbench itself has no inputs or outputs. The testbench generally has two sets of modules: one is the DUT, which you instantiate into your testbench; everything else in the testbench is part of the stimulus driver. This is clearer when you see it in an example.

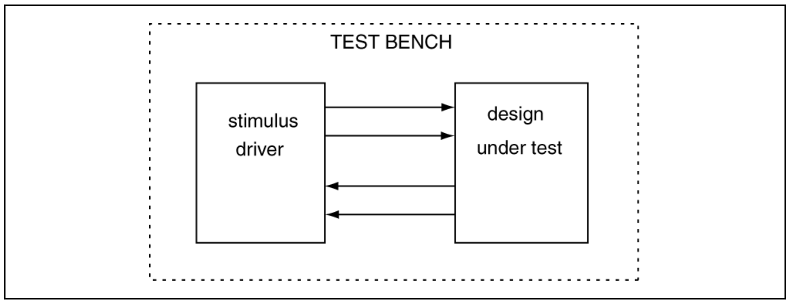


Figure 13: The general model of an HDL testbench.

Testbench Example #1

Use the Verilog model of a 4-bit comparator in Figure 13 to generate a basic testbench model.

Solution: Figure 14 shows a model of a 4-bit comparator with eq, lt, & gt outputs. Figure 15 shows the final testbench for this solution.

```
module comp_4b(a, b, eq, lt, gt);
    input  [3-1:0] a,b;
    output reg eq, lt, gt;

    always @ (a,b)
    begin
       if (a == b)
       begin
          eq = 1; lt = 0;  gt = 0;
       end
       else if (a > b)
       begin
          eq = 0; lt = 0;  gt = 1;
       end
       else if (a < b)
       begin
          eq = 0; lt = 1;  gt = 0; end
       else
       begin
          eq = 0; lt = 0;  gt = 0;
       end
    end
endmodule
```

Figure 14: The Verilog model for a 4-bit comparator

Here are the more pertinent features of the testbench model of Figure 15.

- The argument list for the testbench module declaration is empty. Note that in Figure 13 that there are no inputs or outputs to or from the testbench box.
- The testbench model then provides declarations that the stimulus driver and DUT output uses. The rule here is that we always declare outputs from the stimulus driver (inputs to the DUT) as **reg**-type and we always declare outputs from the DUT as **wire**-types. Both declarations use wire-types as needed.
- The model uses an **initial** procedural block to generate stimulus to the DUT as a function of time. This block is similar to an **always** block, but the block is only executed one time.
- We use the "#" to indicate relative time in the **initial** block. The **initial** block only specifies inputs to the DUT; the simulator automatically generates the outputs. The first data specifications do not contain time indicators, which mean the simulation start at time zero. The next time the code changes the test vectors is 20 time units later as indicated by the "#20". This code changes the values two more times. The second "#20" officially occurs 40 time units after the vectors are first assigned as this model uses relative time.
- The testbench provides all inputs values to the DUT with initial values; if we did not do this, the simulator would post unknown values on both the DUT's inputs and outputs.

```
module tb_comp_4b(   );
   //- stimulus connections to DUT
   reg [3:0] a, b;       //- stimulus outputs
   wire eq, lt, gt;      //- DUT outputs

   //- DUT instantiation
   comp_4b MY_COMP (
      .a  (a),
      .b  (b),
      .eq (eq),
      .lt (lt),
      .gt (gt)   );

   initial
      begin
         //- initial values of a & b
         a = 'hA;
         b = 'hB;

         //- a & b values 20 time units later
         #20 a = 'hB;
             b = 'hB;

         //- a & b values 20 time units later
         #20 a = 4'b1011;
             b = 4'b0001;

         //- a & b values 20 time units later
         #20 a = 4'b0001;
             b = 4'b0001;
      end
endmodule
```

Figure 15: The model of an HDL testbench for a 4-bit comparator.

Testbench Example #2

Use the Verilog model of an n-bit comparator in Figure 16: The Verilog model for an n-bit **comparator (a) and an associated testbench (b).**

(a) to generate a testbench model.

Solution**:** Figure 16: The Verilog model for an n-bit comparator (a) and **an associated testbench (b).**

shows a model of a 4-bit comparator with eq, lt, & gt outputs. Figure 16(b) shows the final testbench for this solution. This problem is similar to the previous problem; the difference being that this example uses a generic version of the comparator and the bit-width is 8 bits.

(a)	(b)

```
module comp_nb(a, b, eq, lt, gt);
    input  [n-1:0] a,b;
    output reg eq, lt, gt;

    parameter n = 8;

    always @ (a,b)
    begin
       if (a == b)
       begin
          eq = 1; lt = 0;  gt = 0;
       end
       else if (a > b)
       begin
          eq = 0; lt = 0;  gt = 1;
       end
       else if (a < b)
       begin
          eq = 0; lt = 1;  gt = 0; end
       else
       begin
          eq = 0; lt = 0;  gt = 0;
       end
    end

endmodule
```
	```
	module tb_comp_nb(   );
	   reg [3:0] a, b;
	   wire eq, lt, gt;

	   //- DUT instantiaion
	   comp_nb #(.n(4)) MY_COMP (
	      .a  (a),
	      .b  (b),
	      .eq (eq),
	      .lt (lt),
	      .gt (gt)   );

	   initial
	   begin
	      a = 'hA;
	      b = 'hB;

	      #20 a = 'hB;    //- time=20
	          b = 'hB;

	      #20 a = 4'b1011; //- time=40
	          b = 4'b0001;

	      #20 a = 4'b0001; //- time=60
	          b = 4'b0001;
	      end
	endmodule
	```

Figure 16: The Verilog model for an n-bit comparator (a) and an associated testbench (b).

Testbench Example #3: Generic Up/Down Counter

Use the Verilog model of an n-bit counter in Figure 17 to generate a basic testbench model for an 8-bit counter.

Solution: Figure 17 shows a model of an n-bit counter; Figure 18 shows the final testbench for this solution. This solution has one major difference found in most circuits, which is a clock generator.

```
module cntr_udclr_nb(clk, clr, up, ld, D, count, rco);
    input  clk, clr, up, ld;
    input  [n-1:0] D;
    output   reg [n-1:0] count;
    output   reg rco;


    //- default data-width
    parameter n = 8;

    always @(posedge clr, posedge clk)
    begin
        if (clr == 1)       // asynch reset
           count <= 0;
        else if (ld == 1)   // load new value
           count <= D;
        else if (up == 1)   // count up (increment)
           count <= count + 1;
        else if (up == 0)   // count down (decrement)
           count <= count - 1;
    end


    //- handles the RCO, which is direction dependent
    always @(count, up)
    begin
       if ( up == 1 && &count == 1'b1)
          rco = 1'b1;
       else if (up == 0 && |count == 1'b0)
          rco <= 1'b1;
       else
          rco <= 1'b0;
    end

endmodule
```

Figure 17: The Verilog model of an n-bit up/down counter.

Figure 18 shows the testbench that supports the n-bit counter. Here are a few comments of interest regarding the counter this testbench model.

- The original model defaults to an 8-bit (see parameter in n-bit counter model), so the testbench utilizes this default value. The testbench knows it needs to work with an 8-bit counter, so the testbench declares the D input and count output as an 8-bit vector. The instantiation of the DUT does not override the default value.
- The counter is a synchronous circuit, so it requires the stimulus driver to provide a clock signal. We opt to generate a periodic signal for testing as well. There are many ways to generate a clock signal in a Verilog testbench, this solution shows one of those ways. The testbench model uses a **forever** statement inside of an initial block for the clock generator. This could have been done with an **always** block, but the clk signal would need to be provided with an initial value in some other process.
- The testbench start the counter at a relatively high 8-bit value, which causes the counter to roll over after a few clock cycles. After that, the testbench changes the clock direction to count down. The up count asserts the **rco** when the counter reaches its terminal count. The **rco** asserts once again in the down direction when the count reaches zero.

```
module tb_comp_nb(   );
   reg [7:0] D;
   reg clk, clr, up, ld;
   wire [7:0] count;
   wire rco;

   cntr_udclr_nb MY_CNTR (
       .clk   (clk),
       .clr   (clr),
       .up    (up),
       .ld    (ld),
       .D     (D),
       .count (count),
       .rco   (rco)   );

   //- Generate periodic clock signal
   initial
   begin
      clk = 0;   //- init signal
      forever  #10 clk = ~clk;
   end;

   initial
      begin
         clk = 0;
         up = 1;
         ld = 0;
         D =  'hFB;
         clr = 0;

         //- send out LD pulse
         #10 ld = 1;
         #30 ld = 0;

         //- change count direction
         #200 up = 0;

      end

endmodule
```

Figure 18: The testbench for the n-bit counter.