Lab 6: High-Level State Machines

EE 316: Digital Logic Design

Overview

This lab is the final lab of the semester and requires you to draw on concepts you learned throughout the semester to make high-level state machines. By the end of this lab, you should be able to:

- Design a high-level state machine in Verilog.
- Describe the functionality of a shift-add multiplier and a multiply-accumulate circuit.
- Synthesize and implement information in previous labs to solve a problem.

This lab is worth 50% more than previous labs. While this lab is indeed difficult, we believe that, with the content you have learned through lecture and previous labs, you are extremely well-prepared to complete this lab.

Background

Shift-Add Multipliers

Multiplication in binary is similar to multiplication in decimal: multiply the top number by each bit of the bottom number to generate partial product, shift them over based on their positional weight, and sum the partial products. This can be seen in Figure 1.

```
          1 1 0 1 (13)
          1 0 1 1 (11)
          -------
          1 1 0 1
        1 1 0 1
      ---------
    1 0 0 1 1 1
    0 0 0 0
    ---------
    1 0 0 1 1 1
  1 1 0 1
---------------
1 0 0 0 1 1 1 1 (143)
```

Figure 1. Binary Multiplication

Binary multiplication requires two *n*-bit inputs, a *2n*-bit output, and an *n*-bit adder with carryout to account for overflow. In Figure 1, for 4-bit multiplication, we would need two 4-bit inputs, a 8-bit output, and a 4-bit adder with carryout. One approach to designing a binary multiplier is to compute all the partial products first, and then add them together. This is extremely taxing on resources and would require a lot more logic. Instead, we use the product register as an

accumulator so that we can keep track of the sum of partial products as we calculate each new partial product. Therefore, a shift-add multiplier has three steps:

1. Calculate the *i*th partial product.
2. Add it to the accumulator.
3. Move to the *i+1*th partial product.

Note that in binary multiplication, the *i*th partial product is always either 0 or the multiplicand based on the multiplier. The multiplicand and multiplier can be arbitrarily chosen because multiplication is commutative. For example, in Figure 1, 13 = 1101 is the multiplicand, and 11 = 1011 is the multiplier. When we multiply 13 x 11, the partial products are either 1101 (when the *i*th bit position in 11 is 1) or 0000 (when the *i*th bit position in 11 is 0). Based on this property, we can simply check each of the bits of the multiplier (11 = 1011) to determine what our partial products should be – now, the question is how.

To accomplish this in the most efficient way possible, we can initialize the product register (also called an *accumulator* or an *accumulation register*) with the multiplier in the LSB position and set one of the inputs of the adder to the other multiplicand, as seen in Figure 2. We initialize the unused bits in the accumulator to 0, as they will store the product.

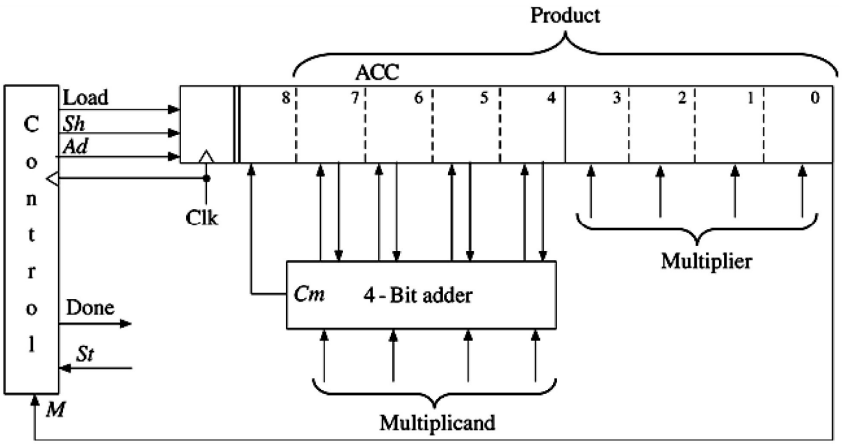


Figure 2. Accumulation Register

Once we initialize the product register, we then begin computing partial products and start adding to the product register (Figure 3). First, we check the LSB of the product register (the "M" bit), which corresponds to the LSB of the multiplicand. If it is 1, we add the multiplicand to the product register in the unused spaces. If it is 0, we do nothing (since adding 0 to anything keeps the same value). Next, we shift the product register to the right by 1 bit. This has two effects: 1. the previous LSB (bit 0 of the accumulation register in Figure 2) used to compute the first partial product is "discarded" so that the new LSB points to the second bit in the multiplier (bit 1 in Figure 2), used to calculate the second partial product, and 2. the product bits (originally bits [8:4]) are shifted right by one bit, which causes the adder to be adding to the next four bits of the product (originally bits [9:5], which moved right), in effect, left-shifting the adder. Iterating this process *n* times, where *n* corresponds to the number of bits in the multiplier, will result in the final product. Figure 3 shows this process visually.

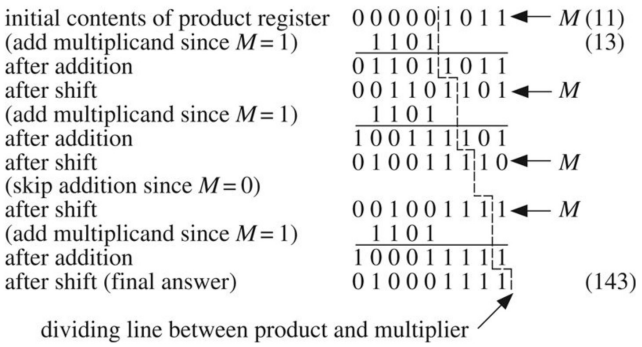


Figure 3. Shift-add Multiplication Process

Because of the sequential nature of shifting and adding, this implementation of binary multiplication can therefore be easily captured as an HLSM that uses datapath components to accomplish the shift and add operations, as seen in Figure 4.

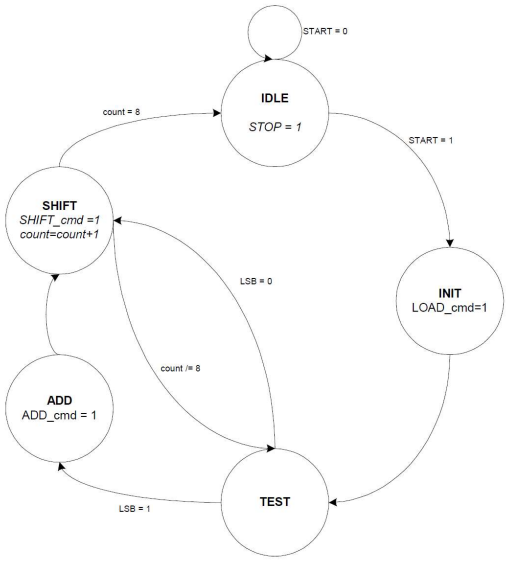


Figure 4. 8-bit Shift-Add Multiplier HLSM

The initial state of the HLSM is the “idle” state, where the HLSM waits for a start signal. When the start signal is asserted, the HLSM transitions into the load state “init,” where it initializes the product register with the zeroed-out product and the multiplier. The “test” state checks the LSB of the product register to determine if an add is required, and if it is, it transitions to the “add” state, which does the add. If the add is not required or the add is complete, the HLSM does the shift and increments a counter that counts the number of shifts completed. If the number of shifts is not equal to (read: less than) the number of multiplier bits, the multiplication is not complete yet, and the HLSM returns to the “test” state to check the next LSB of the product register. If the number of shifts is equal to the number of multiplier bits, the multiplication is complete, and the multiplier returns to the “idle” state and asserts a stop signal, indicating that the multiplication is complete.

Multiply-Accumulate Circuits (MACs)

Multiply-accumulate circuits (MACs) are extremely useful in a wide variety of applications, from machine learning and artificial intelligence to digital signal processing and matrix multiplication. Multiply-accumulate circuits have the advantage of computing sums and products significantly faster than using separate multiply and add modules.

A multiply-accumulate circuit works according to the following equation:

$$a \leftarrow a + b * c$$

where a updates by adding to the product of b and c. In hardware, a acts as an accumulation register to which the product of b and c is added. This circuit therefore requires several datapath components: adder, a multiplier, and a load register.

Procedure

Part A: Up-down Counter

In this section, you will use the up-down counter example located on Canvas under Files > labs > Verilog Examples to understand how to design an HLSM in Verilog. Download the example code, simulate it in Vivado, and complete the following tasks:

1. Read the file labeled readme.pdf, and answer the following questions in your PDF:
    a. How many bits are needed to store the current state of the HLSM?
    b. How long does the HLSM spend in each state before transitioning?
    c. Why is the wait state necessary for the up-down counter? (In other words, why not just use the initial state as the wait state?)
2. Now read through the Verilog source files and answer the following questions in your PDF. Try to limit your responses to two to four medium-length sentences.
    a. What is the XNOR Boolean operator in Verilog?
    b. What is the max value of the counter (in signed decimal), and what happens when the counter reaches its max value?
    c. What is the minimum value of the counter (in signed decimal) and what happens when the counter reaches its minimum value?
    d. What happens if the up and down buttons are pressed at the same time?
    e. What is an inferred latch? When does it occur? Why is it bad? How do you prevent inferred latches from occurring in your code?
    f. When and why is a load register structure necessary for some variables in your HLSM?
    g. How would you implement an asynchronous reset?
3. Take a screenshot of the simulation waveform showing the up button transitioning from low to high. Answer these questions:
    a. How long does it take for the output to reflect onto the seven-seg?
    b. What explains the delay between the button press and the seven-seg update?
    c. In no more than 3-4 sentences, briefly describe how you would use the up-down counter example code and the onoff example code to design a stopwatch.

Part B: Shift-Add Multiplier

Using the information provided in the Background section, design a 4-bit shift-add multiplier. Your top module (`mult.v`) should look like this:

```
module mult(
    input clk,                  // Use input clock - no need for clock divider
    input start,                // Center button
    input [7:0] sw,             // Multiplicands - sw[7:4], sw[3:0]
    output stop,                // Stop signal - LED[15]
    output reg [7:0] product    // Product - LED[7:0]
    );
```

Do not modify the module declaration. You should not need any other design sources besides `mult.v` for this part. In your testbench (`tb_mult.v`), test the cases given in Table 1. Be sure to change the radix of the waveform values to "unsigned decimal" for readability. Screenshot each testcase separately; you should have a total of 7 screenshots.

Table 1. Part B Testcases (in unsigned decimal)

Multiplicand 1	Multiplicand 2
0	0
1	1
1	10
2	5
7	9
15	10
15	15

Figure 5 is an example waveform of an 8-bit multiplier showing one testcase (you don't need the reset line). It is not necessary for you to show the intermediate values of the multiplier, but it is slightly easier to design it that way. Again, don't forget to change the radix of the waveform values.

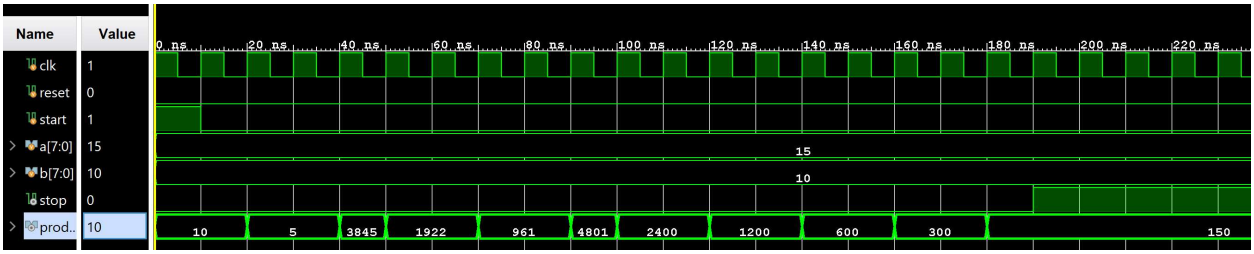


Figure 5. Example Waveform for Part B

Once you get your waveforms and testbench working, add a constraints file (`mult_constrs.xdc`) and generate the bitstream to make sure it is successful.

Part C: Multiply-Accumulate Circuit

In this part, you will create a multiply-accumulate circuit. Name your top module `mac.v`. You are free to use and modify any modules from previous labs that you may find helpful. You may also need to modify modules (e.g. the adders) that you made in this lab; duplicate the source file

so you have a copy of the original before you modify it. For example, you may need a 16-bit adder; you can create it by modifying `rca.v`. In addition, we have provided a working `sevenseg.v` module that interfaces with the seven-segment display for you (see Canvas). The module is a solution to Part C of Lab 4 that you should feel free (and likely need) to modify. **You cannot use arithmetic operators + and *,** but you can use / and %**.**

There are several inputs to your circuit:

- SW[15:8] – Initial value of a, the accumulation register
- SW[7:4] – Value of b, multiplicand
- SW[3:0] – Value of c, multiplicand
- btnC – Load button. When pressed, the MAC performs the operation $a \leftarrow a + b * c$, and the display is updated.
- btnU – Reset button. When pressed, the MAC resets the accumulation register to the value represented by SW[15:8].
- Both the load and reset should be synchronous (i.e. only execute on a rising clock edge). If load and reset are both pressed or unpressed, do nothing.

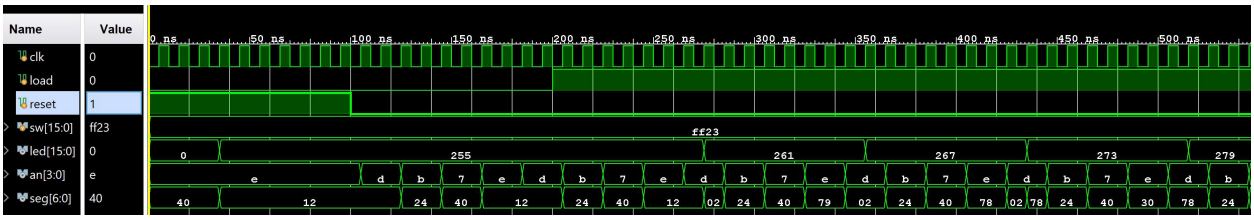
You should display the output as a decimal number from 0000 to 9999 on the seven-segment display and the bit pattern represented by the current value of the accumulation register on LED[14:0] (not all LEDs will be used). If the decimal number has less than four digits, the display should display `0' on the anodes that are unused (e.g. to display the number "54", display "0054"). If the number is larger than 9999 and therefore cannot be represented on the seven-seg, display four dashes "- - - -" and turn LED[15] on to indicate an overflow error. LED[15] should be off when there is no overflow error. You can assume that the largest number that will be contained in the MAC before it resets is 2^{15}-1. The seven-segment display should refresh at 12.5 MHz; that is, each digit in the seven-segment display should be on for 20 ns. (Remember that your clock signal is 100 MHz and should therefore be set to toggle every 5 ns in simulation. The refresh rate requires you to use a clock divider, which you should have from Lab 4.) You should also implement a done signal (on dp) to indicate when your MAC has finished updating.

Based on the requirements above, your top module should look like this (do not change the module to make it more usable – the goal of the lab is still to be able to have your designs function on the board):

```
module mac(
        input clk, load, reset,
        input [15:0] sw,
        output done, // dp
        output [15:0] led,
        output [3:0] an,
        output [6:0] seg
    );
endmodule
```

Below is a snippet of a waveform we expect from your testbench (`tb_mac.v`) Note that the waveform begins with reset asserted; it is good practice to reset your system before using it

because when the board turns on, there is no way to know what values the register contains. The waveform also does not contain a done signal – you will need to show that as well.



Make sure to capture at least four waveforms, with one that shows the system running normally and one that shows the overflow case. Note that you will need at least two clock dividers in this system because your display refresh rate must be at least 4x faster than the rate at which you load new values into the MAC. Also note that your waveforms should display led in decimal, not hex. You may choose any testcases you wish, but here are a few suggestions:

- sw = 16'h02FF, load = 1 until it overflows
- sw = 16'h0000, load = 1 as an edge case
- sw = 16'h0234, load = 1 to make sure the calculations are correct (running normally)

Once you get your waveforms and testbench working, add a constraints file (`mac_constrs.xdc`) and generate the bitstream.

(Bonus) Part D: Memes

We know that this semester has posed a large number of unique challenges for many of you, and many of you have had to rapidly adapt to significant shifts in learning content this semester. We would like to provide you with an opportunity to express yourself and your thoughts on the course through an untraditional format.

For this part, make a collection of memes, Tiktoks, or other short-form content of similar nature reflecting on this semester, this course, UT Austin, or your life during the COVID-19 pandemic. At least 3 of your memes must be (o)riginal (c)ontent; you can have stolen content, but please cite where you stole it from (e.g. provide a Facebook group name, Instagram or Twitter handle, or Tiktok username). We define memes relatively loosely here – rickrolling can be considered a meme for the purposes of the assignment, but please don't submit five rickrolling memes.

We won't grade you on the quality or dankness of your memes, but do try to make them funny or at least relatable. Completing this part will give you an additional 10% on the lab – 15 extra points, 3 points per meme – provided that you complete parts A and B of the lab.

In addition, we will also start a Piazza thread for you to upload your content. You don't have to upload your content, but this will give you an opportunity to share your content, view other people's content, and hopefully relate to other people in the class. Even though it may seem like you are entirely alone in self-isolation, you are not alone, and other people do share at least some of the same experiences that you do. Seeing this in a common/shared capacity will (hopefully) make you feel less alone. Posting at least 3 of your submitted memes on Piazza by the Lab 6 due date will give you an additional 5 bonus points to your lab (note this in your submission PDF).

Submission

When you submit labs, you will need to submit two things: your zipped source code and a PDF answering questions given in the lab document. Below are the instructions for creating your zip file. Your PDF should be submitted separately, NOT included inside your zip file.

1. Create the zip file with all design files, testbenches, constraints files, and bitstreams that you used in this lab. In particular, make sure to include:
    a. Part B: `mult.v, tb_mult.v, mult_constrs.xdc, mult.bit`
    b. Part C: `mac.v, tb_mac.v, mac_constrs.xdc, mac.bit`, and any other design files, testbenches, etc. that you used for this part
2. Create your PDF. Include the following, **in order**:
    a. A cover page with your name, EID, section unique number, and professor.
    b. Answers to the questions in Part A, along with requested waveforms.
    c. Schematic for the multiplier (no need to expand any boxes).
    d. Multiplier waveforms, one per testcase in given table.
    e. Schematic for the MAC (no need to expand any boxes).
    f. Four waveforms for the MAC, with at least one testcase showing normal operation and one testcase showing overflow.
    g. Meme collection for part D. (If you have gifs or videos, add them separately to your submission and reference them with their source, if applicable, in the PDF.)
    h. Post numbers of the Piazza follow-ups with your memes if you posted them.
    i. Answers to the checkout questions below.
3. Submit your zip and PDF as one submission on Canvas.

Checkout Questions

Answer the following questions in your PDF:

1. The shift-add multiplier requires right-shifting the accumulator, which "deletes" the LSB but also "inserts" a value for the MSB. What value should be inserted into the MSB of the accumulator when it is right-shifted?
2. Assuming that the base clock is 100 MHz and the HLSM transitions immediately to the "init" state when start is asserted, write an equation that calculates the maximum time an *n*-bit multiplier would take to complete the multiplication operation.
3. Assuming the same assumptions from the previous question, write an equation for the maximum time that the multiplier would take to square an even number.
4. What is the smallest-size load register that can store all non-overflow values of the MAC at any time, in bits?
5. In 20 words or less, why must we assume that the MAC will never contain a value greater than 2^{15}-1?
6. Why must the seven-seg display module be clocked at 4x the maximum switching frequency of the load button? The maximum switching frequency is defined as the maximum frequency at which you can toggle load for the MAC to still work.

Rubric

Below is the grading rubric for this lab. Please review it before submitting your lab to ensure you meet all the requirements.

	Missing	Attempted	Half-correct	Almost correct	Fully correct
Assessed from zip file submitted on Canvas					
Multiplier	0	12	13	14	15
Multiplier testbench	0	12	13	14	15
MAC	0	12	13	14	15
MAC testbench for one testcase	0	12	13	14	15
Assessed from deliverables PDF					
Up-down counter answers	0	17	18	19	20
Multiplier schematic	0				5
Multiplier waveforms/verification	0	12	13	14	15
MAC schematic	0				5
MAC waveforms/verification	0	12	13	14	15
Checkout questions	0	10	15	18	20
On-time submission	0		5		10
Memes	0	6	9	12	15
Meme-posting	0				5