18-240: Structure and Design of Digital Systems

Electrical & Computer ENGINEERING

HW6 [6 problems, 64 points]

Covers lectures L12 – L14 *Due: 30 October 2023*

Homework sets are due at 5:00PM on the due date. Upload your answers, to Gradescope by then. No late homework will be accepted. Remember, we let you drop two homework assignments over the semester.

By now, you know how to use **`handin240`** to create the files for the Gradescope upload. Refer to the course wiki or to previous homework if you need refresher instructions.

Discussions about homework in small groups are encouraged — think of this as giving hints, not solutions, to each other. However, homework must be written up individually (no copying is allowed). If you discussed your homework solutions with someone else, either as the giver or receiver of information, your write-up must explicitly identify the individuals and the manner information was shared.

You must show details of your work. There is no credit for just writing down an answer.

Drill Problems [28 points]

1. [12 points, Lecture 12] You've now learned the **parameter** keyword in SystemVerilog, which is used to specify signal sizes at the time a module is instantiated. Use this knowledge to collect and update a bunch of SystemVerilog modules for commonly used datapath components. We are also including the sequential components and other often-used building blocks.

   In a file named **library.sv**, build the modules shown on the following page. You may update the **library.sv** file you turned in for previous homework exercises. Again, you are free to copy code you have found in lecture slides, homework problems or the textbook.

   For each module, be careful to use the exact names from the diagrams, including case. Thick lines with a slash are multi-bit signals, whose size is related to the parameterization values given. The **Decoder** and **Multiplexer** have signals with a question mark for the width, as those signal sizes are a function of the WIDTH parameter.

   The **MagComp** and **Adder** are combinational components. I hope you are learning to eschew complexity as much as I, so make all your control signals active high. Those outputs on the comparator specify if A is less than B (**AltB**), A is equal to B (**AeqB**) or if A is greater than B (**AgtB**). A and B are unsigned.

   The **Multiplexer** is a combinational circuit that chooses a single bit from the multi-bit input, the width of which is parameterized (because **S** is a function of the input width, you might find the system function **$clog2** to be of use here. It will return the ceiling of the log (base 2) of its sole parameter). Note that we are following the 240 Coding Standards by not naming this thing **mux**. That name messes with similarly named modules in the Quartus library.

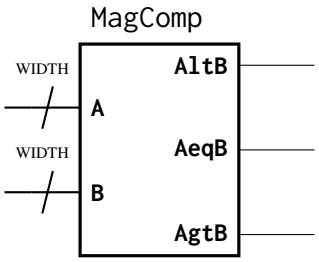
   The **Mux2to1** is similar to the **Multiplexer**, but allows for selection between two multi-bit inputs.

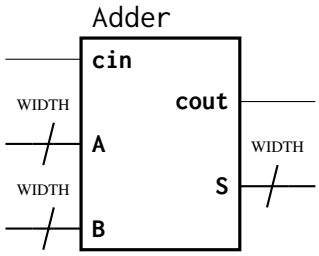
   The **Decoder** is another combinational circuit that converts from binary to one-hot if enabled. Use the WIDTH parameter to specify the size of **D**, not **I**.

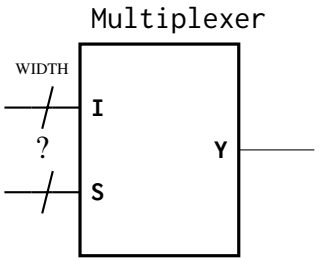
   A **DFlipFlop** stores a single bit. Despite what I just said about complexity and active high signals, this component traditionally has some active low ones. The **preset_L** and **reset_L** inputs are used to asynchronously set or clear the stored value. All signals on this component are single bit wide, so you will not need to parameterize anything.

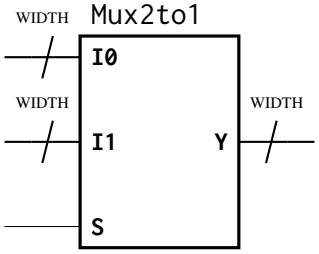
   The **Register** is made of many **DFlipFlop**s, though your code probably won't be written that way. If enabled, on a positive edge of the input named **clock**, it will store the **D** input. The **clear** is … well, I'm not going to tell you if it is synchronous or asynchronous, you should know from the name. The **enable** input has priority over **clear**.

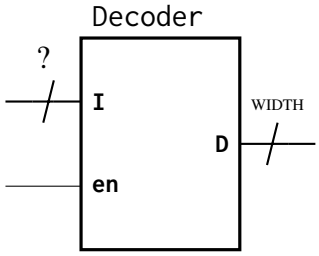
   The **Counter** has control inputs to enable counting, clearing, loading a value from the **D** inputs and to determine the direction of count (up or down). The **clear** input takes priority over **load**, which takes priority over counting.

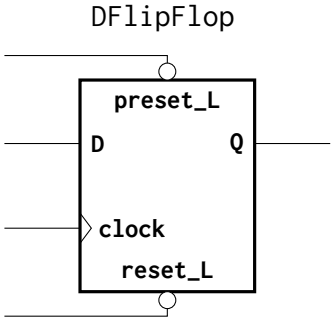

MagComp
WIDTH
A
WIDTH
B
AltB
AeqB
AgtB

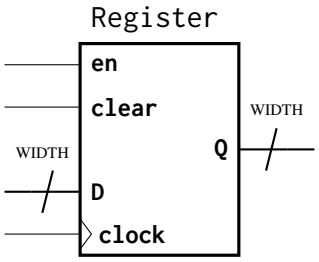

Adder
cin
WIDTH
A
WIDTH
B
cout
WIDTH
S

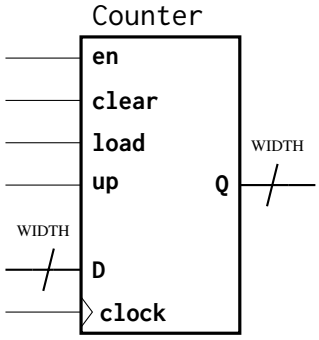

Multiplexer
WIDTH
I
?
S
Y

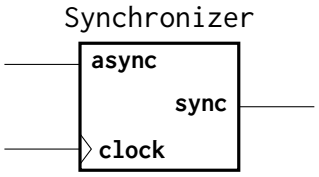

WIDTH
Mux2to1
I0
WIDTH
I1
S
Y
WIDTH

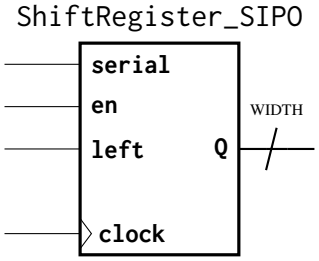

Decoder
?
I
en
D
WIDTH

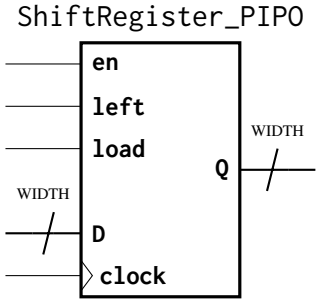

DFlipFlop
preset_L
D
Q
clock
reset_L

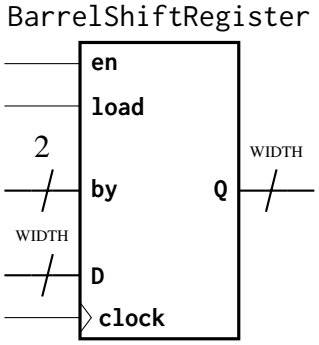

Register
en
clear
WIDTH
D
clock
Q
WIDTH


Counter
en
clear
load
up
WIDTH
D
clock
Q
WIDTH


Synchronizer
async
clock
sync


ShiftRegister_SIPO
serial
en
left
clock
Q
WIDTH


ShiftRegister_PIPO
en
left
load
WIDTH
D
clock
Q
WIDTH

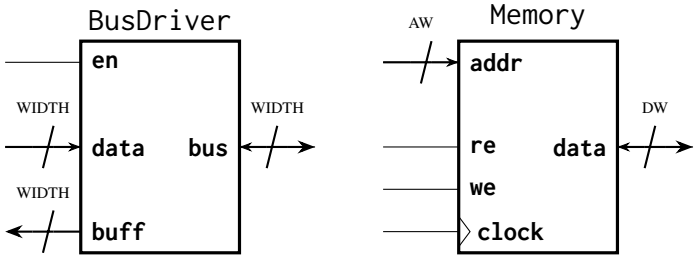

BarrelShiftRegister
en
load
2
WIDTH
D
clock
by
Q
WIDTH

The **Synchronizer** is the circuitry that protects an FSM or hardware thread from an asynchronous input signal, which could be asserted at any time and thus cause setup/hold violations or metastable situations. The result is a signal synchronized to the local clock (the **clock** input).

The **ShiftRegister_PIPO** is a PIPO register that shifts either left or right depending on the **left** control input. It only shifts when enabled and **load** is not active.

The **ShiftRegister_SIPO** is a SIPO register that shifts either left or right. It will consume the bit on the **serial** input and place it in either the MSB or LSB position of the output. When not enabled, nothing will change.

The **BarrelShiftRegister** is a PIPO register that shifts left. It shifts left either 0, 1, 2 or 3 positions based on the 2-bit **by** input (short for "shift by this amount"). It only shifts when enabled, of course. Load has priority over shifting.



The **BusDriver** is used to control access to a shared wire or bus. When enabled, whatever value of **data** will be driven onto the **bus**, otherwise the bus driver will not drive anything.

The **Memory** is our memory module, which stores a number of words. It is combinational read, sequential write.

▷ **Submit all of your modules in a single file named `library.sv`.**

Testing your components is a great idea[1]! Add a testbench module for each, named after the component with **_test** appended. Be careful when testing the **Memory**, as your testbench will be sending signals onto a shared bus. You will need some special circuitry in the testbench.

You need not include a testbench for the **Synchronizer** as metastability testing is a more involved and deeper topic (and, as the component itself is pretty simple and hopefully hard to mess up). I'll also let you slide on a testbench for **BusDriver** for similar reasons.

Check out the guide on Canvas to figure out how to simulate a single module from a file (i.e. a single testbench at a time).

▷ **Submit your testbenches as a file named `library_tests.sv`.**

[1] Especially as you'll come to rely on them for Lab 4!

Note: The entire point of building a **`library.sv`** file is to use the components in this *library* whenever you need them as building blocks for bigger systems. These components should be thoroughly tested and you can trust they work properly. Therefore, use the **`library.sv`** file whenever you need a component – labs, future homework, etc. Don't copy the component into your code. Rather, include the library file on the command line with VCS or put it in your Quartus project. Unused components won't take up space or get instantiated in anything.

2. [8 points, Lecture 12] Design and implement a serial magnitude comparator. As your inspiration, use the serial adder from Lecture 12. If implemented in SystemVerilog (which I'm not asking you to do, yet), your circuit would have the following header:

```
module serialMagComp
  (input  logic A, B,           // MSB first!
   input  logic clock, reset_L,
   output logic AgtB, AltB, AeqB);
```

Your circuit will take two serial input values, most significant bit first, at the inputs **`A`** and **`B`**. The signal comparison operation outputs (**`AltB`**, **`AeqB`**, **`AgtB`**) will always be calculated based on the **`A`** and **`B`** presented so far. Therefore the circuit will work, regardless of how many bits long each input will be.

Draw a schematic for your design (Neatly!).

▷ **Submit a schematic in your PDF file.**

3. [8 points, Lecture 12] Modify the datapath from L12, slide 24 so that a series of 12 **`InputA`** values will be summed, but only if they are odd and greater than 87. In other words, it will do something like this code snippet:

```
sum = 0
for _ in range(12):
    if isOdd(InputA) and (InputA > 87):
        sum = sum + InputA
```

Do this: Redraw the circuit from slide 24, but add enough circuitry to detect if **`InputA`** is odd and larger than 87 (assume **`InputA`** is an unsigned value). Think about how will the FSM use this circuitry? Then, redraw the STD for the FSM such that it will loop 12 times instead of 2.

▷ **Submit your schematic and STD in your PDF file.**

Non-Drill Problems [36 points]

4. [8 points, Lecture 12] Write SystemVerilog code for the Serial Magnitude Comparator of problem 2. Your library file has a Flip-flop module. You should probably write test benches to ensure it works, though they won't be graded.

   ▷ **Submit your code in a file named `hw6prob4.sv`.**

5. [14 points, Lecture 13] In Lecture 13, I presented several alternative designs for the ones-count problem. The alternative on Slide 40 eliminated the loop counter and replaced it with a comparator ("And Another"). Implement this version of the entire HW thread in SystemVerilog.

   To get you started, I have placed the main code into the file **`hw6prob5_given.sv`** and attached it to this homework on Canvas. Use your own shift register, comparator, and counter modules (the ones in your **`library.sv`** file; please do *not* copy them into **`hw6prob5.sv`**.) Do not modify the **`OnesCount`** module header (ports, parameter, name) in any way!

   The timing requirements for this problem are similar to those described in class. Take a look at the STD on Slide 28 ("And a Final Arc"), and you will see that on the first clock edge where **`d_in_ready`** is asserted, the input value is captured into the DIN Register. From the same FSM, you will note that **`D_out_ready`** is asserted as soon as the comparator signals done, at which point the FSM returns to the initialization state in order to prepare for another request.

   ▷ **Submit your work as a file named `hw6prob5.sv`. Include a testbench with at least four test vectors.**

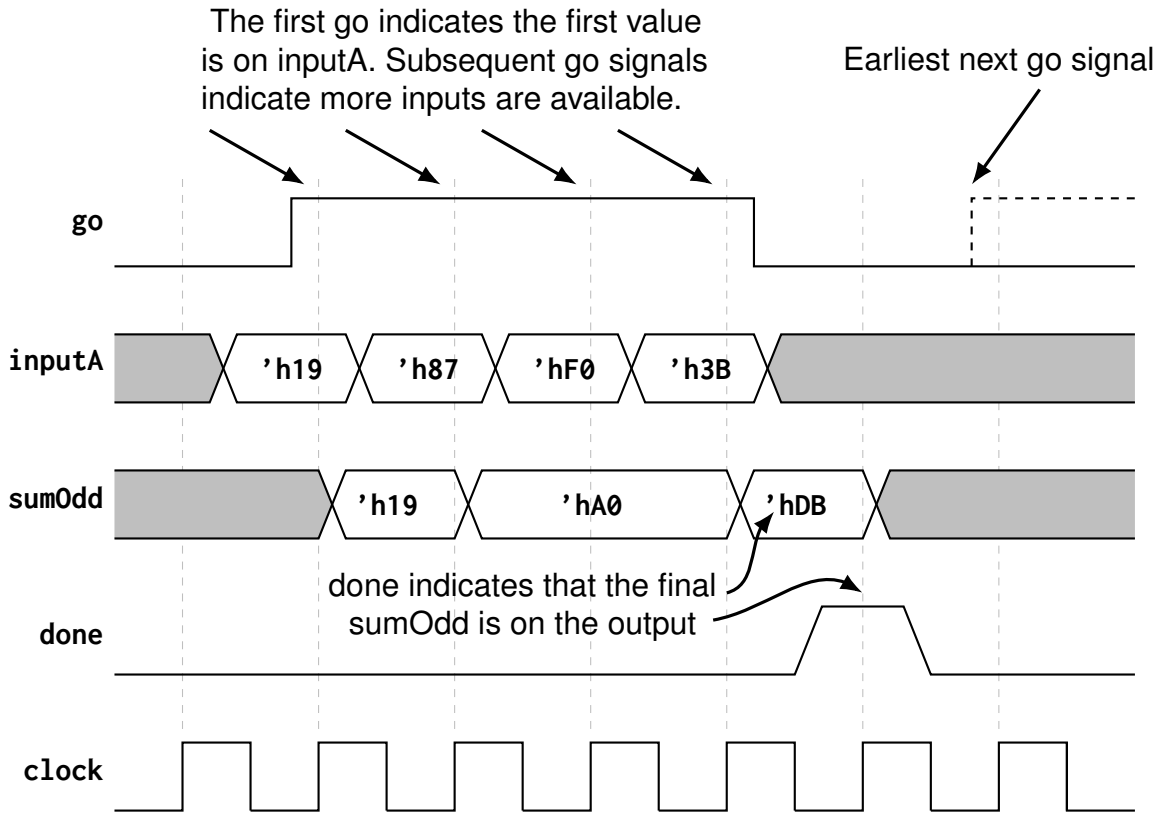
6. [14 points, Lecture 13] Design and implement a hardware thread (i.e an FSM-D) to sum the odd unsigned numbers on its input. The FSM-D port definitions are:

```
module hw6prob6
  (input  logic        go,
   input  logic [12:0] inputA,
   output logic        done,
   output logic [12:0] sumOdd,
   input  logic        clock, reset);
```

go is a control signal, telling the FSM-D when to start receiving inputs. The FSM-D will wait until the **go** signal is asserted before proceeding. When go is first asserted, the first **inputA** value is on the input. While **go** remains asserted, there will be new input values on **inputA** at each successive **clock** edge. As each of these inputs is available, **sumOdd** is updated so that it is always the sum of the prior odd inputs (over the time period since **go** began to be asserted). Something like:

```
sumOdd = (isOdd(inputA)) ? inputA + sumOdd : sumOdd;
```

When **go** becomes unasserted, **done** is asserted, indicating that the **sumOdd** output has the sum of this particular sequence of inputs; it remains asserted for one clock edge. The system then waits for the next **go** to be asserted. See timing diagram below. On **reset**, **sumOdd** should be 0 and the system is waiting for **go**. There could be more/less than four inputs in the sequence.



Do this:

(a) Determine the register transformations required by the datapath. List them using a notation like: $R1 \Leftarrow R2 \oplus R3$.

(b) Choose the datapath components for your datapath. These should be components from problem 1.

(c) Draw the datapath schematic. Label the control and status points.

(d) Draw the STD for the FSM that will control the datapath. It should use the status points to make decisions (i.e. should a transition be made to StateA or StateB). It should also output the control points that go to the datapath. Of course, the FSM may also have additional inputs and outputs.

(e) Write (and test, and debug) the SystemVerilog for your hardware thread. Do not include standard components (i.e. those in your library file) in the code for this problem. Simply include **`library.sv`** on the VCS command line instead.

▷ Submit it your code in a file named `hw6prob6.sv`. Your other design work goes in your PDF file. And yes, you should have other design work.

An encrypted testbench, called **`hw6prob6TB.svp`**, is available on Canvas. It has a top module named **`hw6prob6_test`** and it will instantiate your module as defined above.