CSE3666

Homework 4

Due Date: By the end of Friday, 10/21/2022.

Total points: 100

Submit your work in a single PDF file in HuskyCT.

1. We build a state machine that detects if a binary number of an arbitrary length is divisible by 3. The state machine has three states S0, S1, and S2. The bits in the number are fed into the machine from left to right, i.e., from the most significant bit (the left-most bit) to the least significant bit, one bit per clock cycle. The state machine starts from state S0. Depending on the current state and the input bit, the state machine transits from one state to another, as shown in the following state table. In the table, `b` is the bit sent to the state machine and `z` is the output, which is 1 if and only if the state machine is in S0, indicating the bits the state machine has seen so far is divisible by 3. Note that the output `z` depends only on the current state.

State	b	NextState	z
S0	0	S0	1
S0	1	S1	1
S1	0	S2	0
S1	1	S0	0
S2	0	S1	0
S2	1	S2	0

Implement the state machine in MyHDL. The skeleton code is in `q1.py`. The diagram of state machine is the same as the one on Slide 18 (in lecture 2.2), but has different signal names. We can complete the design in 3 steps. Steps 2 and 3 are combinational circuit.

Step 1. Instantiate a register to keep the state. This step is already done. Study the code and learn how to instantiate a block and a register.
Note that `state` is a signal that has two bits. Bits stored in the state register indicate the current state. 0 means S0, 1 means S1 and 2 means S2. We can access each bit with `state[0]` or `state[1]`.

Step 2. Complete the `next_state_logic()` function, which generates the next state to be saved in the state register in the next cycle. We start by turning the state table (above) into a

truth table like the following. Then we write a logic expression for each bit to be saved in the state register in the next cycle.

State[1]	State[0]	b	NextState[1]	NextState[0]	z
0	0	0	0	0	1
0	0	1	0	1	1
…					

Step 3. Complete the `output_logic()` function, which generates the output signal `z`. We write a logic expression for `z` from the truth table constructed in Step 2.

When testing the circuit, we can specify bits on the command line. Here is the output of the program where the bit string is `10011`.

```
python q1.py 10011
state b | ns z v
  0   1 | 1  1 0   # starting from S0. MSB is 1. Next state is S1
  1   0 | 2  0 1   # MSB is accepted. Not divisible by 3
  2   0 | 1  0 2   # Got 10. Next bit is 0
  1   1 | 0  0 4   # Got 100. Next bit is 1
  0   1 | 1  1 9   # Got 1001.
  1   1 | 0  0 19  # Got 10011. b is not changed
```

In the PDF file, include the following:

a) Step 2. Truth table and the code for function `next_stage_logic`.

b) Step 3. Code for function `output_logic`.

c) The output of the program when the bit string is 111000101.

2. Consider the multiplier we have studied. Inside the control module, there is also a register that counts the steps and a combinational circuit. Assume the following.
   - The delay of the adder is 10 ns,
   - The delay of the combinational circuit in the control module is 2 ns.
   - The setup time, the hold time, and the propagation delay of the registers is 0.5 ns, 0.2 ns, and 1 ns, respectively.
   - Do not consider the delays on wire.

   Answer the following questions. Round answers to the nearest tenth if necessary. For example, enter 3 for 3, 0.3 for 1/3, and 0.7 for 2/3.

   a. In a figure like below, show the timing of the following events.

      a1. Reg-Ready. The output of registers is available.

      a2. Control-Ready. The output of control module is available.

      a3. Adder-Ready. The output of adder is available.

      a4. Latest. The input to registers must be ready. This is the example in the figure.

a4. Latest

   b. What is the minimum cycle time for this multiplier to work properly?
   c. What is the highest clock rate in MHz that this multiplier can run at?
   d. If we build sequential circuit with the same kind of registers, what is the highest clock rate in MHz we can achieve?

3. Assume we have built a 5-bit multiplier, based on the design we have discussed, and use it to calculate 27 * 17. Fill out the following table with bits stored in registers after each step. Verify with decimal arithmetic that 1) the product register has the correct answer if bits are considered as unsigned, and 2) the lower half the product register has the correct bits if bits in 27 and 17 are considered as signed.

Steps	Multiplicand	Multiplier	Product
init			
1			
2			
…			

4. Translate the following C function to RISC-V assembly code. The function converts an unsigned number into a string representing the number in decimal. For example, after the following function call, the string placed in buffer is “3666”.

   ```
   uint2decstr(buffer, 3666);
   ```

   Assume the caller has allocated enough space for the string. Skeleton code is in `q4.s`, where the function is empty. The assembly code should follow RISC-V calling convention. Clearly mark in comments how each statement is translated into instructions. Only include the instructions (and comments) in the function in the PDF file.

```
// char * means the address of a character
char * uint2decstr(char s[], unsigned int v)
{
    unsigned int r;

    if (v >= 10) {
        s = uint2decstr(s, v / 10);
    }
    r = v % 10;       // remainder
    s[0] = '0' + r;
    s[1] = 0;
    return &s[1];     // return the address of s[1]
}
```