3666 hw 4

Casey Provitera

March 2024

1

We build a state machine in this problem. The machine has four states, 1-bit input, and 1-bit output. The initial state is 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,each row specifies what the state machine should do in a cycle, for each state and input combination. b is the input and z is the output.

Implement the state machine in MyHDL. The skeleton code is in q1.py. We

State	b	NextState	z
S0	0	S0	1
S0	1	S1	0
S1	0	S2	0
S1	1	S0	1
S2	0	S1	0
S2	1	S2	0
S3	0	S0	1
S3	1	S1	0

complete the design in 4 steps. Steps 2, 3, and 4 are combinational circuit design.

Since we have four states, we will use a signal of two bits to keep track of the state. In the skeleton code, the signal is state. It has two bits. The underlying data type of the signal is a 2-bit vector. It is the output of a register. Its value indicates the current state. 0 means S0, 1 means S1 and so on. We can access each bit in state with state[0] or state[1].

Step 1. Instantiate a register to keep the state. This step is already done. Signal next_stateis the input of the register and state is the output. Study the code and learn how to instantiate a block and a register.

Step 2. We start by turning the state table (above) into a truth table like the following. Then we write a logic expression for each next_state[0], next_state[1], and z. The logic expressions will be used in later steps.

Step 3. Complete the next_state_logic() function in q1.py. The function generates next_state[1] and next_state[0], based on the logic expressions in Step 2. Note that next_state is the input to the register and determines the state in the next cycle. See the additional requirements in the comments.

Step 4. Complete the output_logic() function. The function uses the logic expression in Step 2 to generate the output signal z from the current state and the current input. See the additional requirements in the comments.

Testing
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	0	1
1	0	2	0	2
2	0	1	0	4
1	1	0	1	9
0	1	1	0	19
1	1	0	1	39

In the submitted PDF file, include the following for this problem.

1. Step 2. Truth table and logic expressions.
   Truth Table

S[1]	S[0]	B	Z	Next State S[1]	S[0]
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	0	0	1
1	0	1	0	1	0
1	1	0	1	0	0
1	1	1	0	0	1

   Logical expressions:

z = $(s[\hat{1}] * s[\hat{0}] * \hat{b}) + (s[\hat{1}] * s[0] * b) + (s[1] * s[0] * \hat{b})$
next state:
s[1] = $(s[\hat{1}] * s[0] * \hat{b}) + (s[1] * s[\hat{0}] * b)$
s[0] = $(s[1] * s[0] * b) + (s[1] * s[\hat{0}] * \hat{b}) + (s[\hat{1}] * s[\hat{0}] * b)$

2. Step 3. Code for function next_state_logic.
```
def next_state_logic():
# Two statements to set two bits in next_state
# Use only logic operators 'not', 'and', and 'or'.
s1, s0 = state[1], state[0]
next_state.next[1] = (not s1 and s0 and not b) or (s1 and not s0 and b)
next_state.next[0] = (s1 and s0 and b) or (s1 and not s0 and not b) or
(not s1 and not s0 and b)
```

3. Step 4. Code for function output_logic.
```
def output_logic():
# generate z from state and b
# Use only logic operators 'not', 'and', and 'or'.
s1, s0 = state[1], state[0]
z.next = (not s1 and not s0 and not b) or (not s1 and s0 and b) or (s1
and s0 and not b)
```

4. The output of the program when the bit string is 111000101.
Command:
```
(venv) PS C:/Users/casey/OneDrive/Desktop/Spring 2024/CSE 3666/hw/hw4>
python q1.py 111000101
```

state	b	ns	z	v
0	1	1	0	1
1	1	0	1	3
0	1	1	0	7
1	0	2	0	14
2	0	1	0	28
1	0	2	0	56
2	1	2	0	113
2	0	1	0	226
1	1	0	1	453
0	1	1	0	907

2

Consider the multiplier we have studied. Inside the control module, there is also a register that counts the steps and a combinational circuit. Note that the register in the control module is triggered by the same clock and the output of the control module depends on the register. Assume the following.

- The propagation delay of the registers is 2 ns.
- The delay of the adder is 10 ns,
- The delay of the combinational circuit in the control module is 1 ns.
- Do not consider the propagation delays on wire.

Answer the following questions. Round answers to the nearest tenth if necessary.

1. In a figure like below, show the timing of the following events, relative to the beginning of a cycle (i.e., when after the rising edge does each of the following events happen?)
   (a) aReg-Ready. The output of registers is available.
   After the rising edge but before the falling edge.
   (b) Control-Ready. The output of control module is available.
   After the falling edge but before the rising edge that starts a new cycle.
   (c) Adder-Ready. The output of adder is available.
   Right before the rising edge that starts a new cycle.

2. What is the minimum cycle time for this multiplier to work properly?
   The minimum cycle time is going to be the total sum from all the propagation delays.
   2 ns for updating registers, 10 ns for adder adder + write to product register
   = ns(10 + 2)
   = 12 ns

3. What is the highest clock rate in MHz that this multiplier can run at?
   Clock rate = 1/ cycle time
   = 1 / 12 ns
   = about 83 MHz

4. If we build sequential circuit with the same kind of registers, what is the highest clock rate in MHz we can achieve?
   If we mortify the circuit to be sequential instead of combinational we can make the clock rate much higher since we no longer need the adder block and instead can simply conjoin everything at the end. This works because we had split the circuit into many mini circuit all of which work on a single bit. The only thing we need to worry about now is register delay.
   This means the cycle time is only 2 ns making the clock rate 500 MHz.

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
   If the bit are considered unsigned then when we shift right values are padded with 0s. This can be shown to produce the proper output to the product register.
   This can be seen in the table below, where we pad with 0. We know this gives the correct answer because if we convert 0111001011 back to decimal it is 459 which is the value of 27*17.

Steps	Multiplicand	Multiplier	Product
init	0000011011	10001	0
1	0000110110	01000	11011
2	0001101100	00100	11011
3	0011011000	00010	11011
4	0110110000	00001	11011
5	1101100000	00000	0111001011

2. The lower half the product register has the correct bits if bits in 27 and 17 are considered as signed.
   Now if the bits in 27 and 17 are considered signed bits then they are padded with the sign bit which in the case for each number since we are only using a 5 bit multiplier is 1. So differently from when the bits were considered unsigned now when we shift right we pad with 1s.
   In the table below, we pad with the sign bit, we can see that the final 5 bits in the last entry of the product column match the last entry in the product column in the table above. So although we know the final value

is not correct while padding with the signed bit we have found that the lower half of the bits are correct.

Steps	Multiplicand	Multiplier	Product
init	1111111011	10001	0
1	1111110110	11000	1111111011
2	1111101100	11100	1111111011
3	1111011000	11110	1111111011
4	1110110000	11111	1111111011
5	1101100000	11111	11110101011

4

Translate the following C function to RISC-V assembly code. We can use M extension in this question. 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

```
1. char * uint2decstr(char s[], unsigned int v) {
2.     unsigned int r;
3.     if (v >= 10) {
4.         s = uint2decstr(s, v / 10);}
5.     r = v % 10; # remainder
6.     s[0] = '0' + r;
7.     s[1] = 0;
8.     return &s[1]; # return the address of s[1]}
```

```
1. uint2decstr:
   # key; s0=10, s1=r, s2=s[0]
   # setting s0 to 10 since the number 10 is used multiple times
2.      addi s0, x0, 10
   # we are putting things in memory since there is a function call within
   the function, storing outside of if block call just incase the if block is not
   executed
3.      addi sp, sp, -4
4.      sw ra, 0(sp)
   # we should not be changing v forever when we divide by 10 only changing
   it in the call, when we get past the call we should be using the original v
5.      addi sp, sp, -4
6.      sw a1, 0(sp)
   # if statement seeing is v >= 10
7.      bge a1, s0, recall
8.      beq x0, x0, skip
9.      recall:
   # recall uint2decstr with s and v/10
10.         divu a1, a1, s0
11.         jal ra, uint2decstr
12.     skip:
   # getting v back from the stack
13.     lw a1, 0(sp)
14.     addi sp, sp, 4
   # modulo v and 10 = r
15.     remu s1, a1, s0
   # s[0] = r + '0'
16.     addi s2, s1, '0'
   # storing statements to put the values back in the proper spot of the char
   array
17.     sw s2, 0(a0)
18.     sw x0, 4(a0)
   # loading return address back from stack
19.     lw ra, 0(sp)
```

```
20.     addi sp, sp, 4
    # a0 is return value we need to set it to the address of a[1]
21.     add a0, x0, a1
22.     jalr ra, x0, 0
```