CSE3666 — Lab 5

Mike Medved

March 7th, 2022

1 Question 1

Design a circuit that takes 4 bits as input and outputs F, which is 1 only when the 4-bit input, interpreted as an unsigned number, is positive and divisible by 3. The input signals are A, B, C, and D. D is the least significant bit.

We can start a truth table like the one below and then write a logic equation for F.
We do not simplify the logic expression in this problem.

Row No.	**A**	**B**	**C**	**D**	**F**
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	1
...					

Implement the circuit in MyHDL. The skeleton code is in q1.py. Compare the truth table generated by the script with the one constructed manually.

2 Deliverable

```
@always_comb
def comb():
   f.next = term(not a, not b, not c, not d) or \
            term(a, b, not c, not d) or \
            term(not a, b, c, not d) or \
            term(not a, not b, c, d) or \
            term(a, not b, not c, d) or \
            term(a, b, c, d)

def term(a, b, c, d):
   return a and b and c and d
```

3 Results

Both the table with a manually computed truth table, and the table with the MyHDL output are below.

As can be seen by the tables, the output is the same as the manually created truth table, since going up to 15 (the maximum 4-bit integer), there are only 6 outputs that can produce TRUE. For this range, 0, 3, 6, 9, 12, and 15 are the only nonzero positive integers divisible by three.

Table 1: MyHDL Code Output

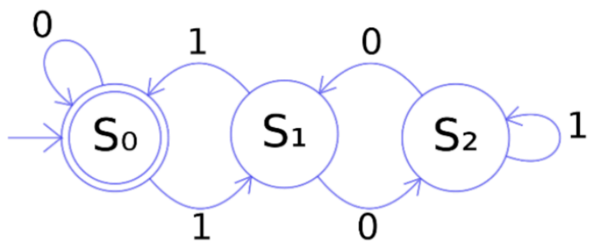
a	**b**	**c**	**d**	**f**
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Table 2: Manual Truth Table

a	**b**	**c**	**d**	**f**
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

4 Question 2

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 to the least significant bit, one bit per clock cycle.The state machine starts from S0.



Depending on the current state and the input bit, the state machine transits from one state to another, as shown in the following diagram. The numbers in the state names (S0, S1, and S2) are the remainder when we divide by 3 the bits that the machine has seen. If the state is S0, the bits are divisible by 3.

Implement the state machine in MyHDL. The skeleton code is in q2.py. We can complete the design in 3 steps. Steps 2 and 3 are the combinational circuit.

- **Step 1** Instantiate a register to keep the state. We leverage the Register block that is provided in the skeleton code. The input and output signals of the state register have already been created.
- **Step 2** Complete the *next_state_logic*() function, which generates the state to be saved in the state register in the next cycle.
- **Step 3** Complete the *z_logic*() function, which generates the output signal z, which indicates whether the number is divisible by 3. The z signal only depends on the current state.

5 Deliverable

```
@block
def Register(dout, din, clock, reset):
   @always_seq(clock.posedge, reset=reset)
   def seq_reg():
      dout.next = din

   return seq_reg

@block
def Detect3x(z, b, clock, reset):
   state = Signal(intbv(0)[2:])
   next_state = Signal(intbv(0)[2:])

   # Step 1 - Instantiating the register with the state and output
   reg = Register(state, next_state, clock, reset)

   @always_comb
   def next_state_logic():
      '''
      Step 2 - Generating the next clock cycle's state
               based on the current state, and the input bit
      '''
      next_state.next = state[1] ^ state[0] ^ b

   @always_comb
   def z_logic():
      # Step 3 - Generating the output based on the current state
      z.next = 1 if state == 0 else 0

   return instances()
```

6 Results

Below is the output of the state machine given eight randomly selected bits as input - 11101001.

Table 3: State Machine Output

b	z	v
1	0	1
1	1	3
1	0	7
0	0	14
1	1	29
0	1	58
0	1	116
1	0	233

7 Question 5

Translate the following C function to RISC-V assembly code. The function converts an unsigned number into a string that represents 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 q5.s, where the function is empty. Clearly mark in comments how each statement is translated into instructions.

```
char* uint2decstr(char* s, unsigned int v)
{
   unsigned int r;
   if (v >= 10) {
      s = uint2decstr(s, v / 10);
   }

   r = v % 10; // remainder
   s[0] =     + r;
   s[1] = 0;
   return &s[1]; // return the address of s[1]
}
```

8 Deliverable

```
uint2decstr:
   addi sp, sp, -16        # allocate 16 bytes on the stack
   sw   ra, 12(sp)         # preserve ra by putting it on the stack
   sw   s0, 8(sp)          # load s0 with the address of the string
   mv   s0, a1             # load s0 with the value of v
   addi a5, x0, 10         # load a5 with the value of 9
   bgeu a1, a5, recurse    # if v > 10, call recurse

write:
   addi a1, x0, 10         # load a1 with the value of 10
   remu s0, s0, a1         # s0 = v % 10
   addi s0, s0, 48         # s0 = s0 + 48
   sb   s0, 0(a0)          # store the result in the string
   sb   x0, 1(a0)          # store the null terminator
   addi a0, a0, 1          # increment the address of the string
   lw   ra, 12(sp)         # restore ra
   lw   s0, 8(sp)          # restore s0
   addi sp, sp, 16         # move stack pointer back
   jr   ra                 # return to caller

recurse:
   addi a1, x0, 10         # load a1 with the value of 10
   divu a1, s0, a1         # a1 = v / 10
   auipc t0, 0             # save current address in t0
   jalr ra, t0, 0xFFFFFFB8 # invoke write
   jal  write
```