CSE3666 — Lab 7

Mike Medved

March 30th, 2022

1 Prompt

In this lab, we implement the multiplier we have discussed in lecture. The multiplicand and multiplier are of 8 bits (although we can easily change that). The multiplicand and the product are stored in 16-bit registers. The adder adds two 16-bit numbers.

The internal signals needed in the implementation are already created in the skeleton code.

The control module has been implemented. done is generated in the module. You may use done signal if needed. However, DO NOT use the counter register directly as the implementation may change.

The tasks to be completed in this lab are as follows. Budget 10 minutes for coding.

- Instantiate registers for p, x, and y. Note that although p is the output of the multiplier, it can be connected to the output of a register directly (i.e., driven directly by the register). x and y are internal signals in the multiplier.

  Let us call the instances reg_p, reg_x, and reg_y, for p, x, and y, respectively.

- Generate input signals to reg_p in comb_regs. p_en is the only signal that needs to be set.

2 Deliverables

```
...

@block
def Mul2x(p, x_init, y_init, load, done, clock, reset):

W = len(x_init)
W2 = W + W

adder_out = Signal(modbv(0)[W2:]) # output of adder
x = Signal(modbv(0)[W2:])
y = Signal(intbv(0)[W:])

p_en = Signal(bool(1))
x_en = Signal(bool(1))
y_en = Signal(bool(1))

counter = Signal(0)
counter_in = Signal(0)
counter_en = Signal(bool(1))
reg_counter = RegisterE(counter, counter_in, counter_en, clock, reset)

p_reset = ResetSignal(bool(0), active=1, isasync=False)


'''
Step 1

Instantiate registers for p, x, and y.
Note that although p (product) is the output of the multiplier,
and that it can be connected to the output of a register directly,
while x and y are internal signals in the multiplier referring
to the multiplier and multiplicand respectively.
'''
reg_p = RegisterE(p, adder_out, p_en, clock, p_reset)
reg_x = RegisterShiftLeft(x, x_init, load, x_en, clock, reset)
reg_y = RegisterShiftRight(y, y_init, load, y_en, clock, reset)
adder = Adder(adder_out, x, p)

'''
Step 2

Generate input signals for p_en in comb_regs.
p_en refers to the enable signal for the register p,
and while true allows the register to receive a new input.
'''
@always_comb
def comb_regs():
   p_reset.next = load
   p_en.next = y[0]

...
```

3 Results

The below tables demonstrate the step-by-step work done by the multiplier circuit we discussed in class. The following operations were performed on the circuit:

- $17 \cdot 36 = 612$
- $36 \cdot 202 = 7272$

load	cnt	prod	x	y	p_en	x_en	y_en	done
1	0	0000000000000000	0000000000010001	00100100	0	1	1	0
0	1	0000000000000000	0000000000100010	00010010	0	1	1	0
0	2	0000000000000000	0000000001000100	00001001	1	1	1	0
0	3	0000000001000100	0000000010001000	00000100	0	1	1	0
0	4	0000000001000100	0000000100010000	00000010	0	1	1	0
0	5	0000000001000100	0000001000100000	00000001	1	1	1	0
0	6	0000001001100100	0000010001000000	00000000	0	1	1	0
0	7	0000001001100100	0000100010000000	00000000	0	1	1	0
0	8	0000001001100100	0001000100000000	00000000	0	1	1	1
0	8	0000001001100100	0010001000000000	00000000	0	1	1	1

Table 1: $17 \cdot 36 = 612$

load	cnt	prod	x	y	p_en	x_en	y_en	done
1	0	0000000000000000	0000000000100100	11001010	0	1	1	0
0	1	0000000000000000	0000000001001000	01100101	1	1	1	0
0	2	0000000001001000	0000000010010000	00110010	0	1	1	0
0	3	0000000001001000	0000000100100000	00011001	1	1	1	0
0	4	0000000101101000	0000001001000000	00001100	0	1	1	0
0	5	0000000101101000	0000010010000000	00000110	0	1	1	0
0	6	0000000101101000	0000100100000000	00000011	1	1	1	0
0	7	0000101001101000	0001001000000000	00000001	1	1	1	0
0	8	0001110001101000	0010010000000000	00000000	0	1	1	1
0	8	0001110001101000	0100100000000000	00000000	0	1	1	1

Table 2: $36 \cdot 202 = 7272$