Luke Pepin

3666 HW4

3/24/2023

1. State Machine

Step 1:

from myhdl import block, always\_comb, always\_seq, Signal, intbv, instances

@block

def Register(dout, din, clock, reset):

"""

A register that always saves din to dout on positive edges

"""

@always\_seq(clock.posedge, reset=reset)

def seq\_reg():

dout.next = din

return seq\_reg

@block

def Detect3x(z, b, clock, reset):

"""

Input: b

Output: z

b is the input

z is the output

"""

# 2 bits are enough to encode 3 states

# initial state is 0

state = Signal(intbv(0)[2:])

next\_state = Signal(intbv(0)[2:])

# instantiate a register for the state

# next\_state is the input and state is the output

reg\_state = Register(state, next\_state, clock, reset)

# generate next\_state, based on state and b

@always\_comb

def next\_state\_logic():

# TODO

# Two statements to set two bits in next\_state

next\_state.next[1] = (state[1] and state[0] and b) + ((not state[1]) and state[0] and (not b))

next\_state.next[0] = (state[1] and state[0] and b) + ((not state[1]) and (not state[0]) and b) + (state[1] and (not state[0]) and (not b))

# generate output

@always\_comb

def output\_logic():

# TODO

# generate z from state and b

z.next = (not next\_state[1] and (not next\_state[0]))

# monitor that prints the values at the negative edge

@instance

def monitor():

# v is only for debugging

v = 0

while True:

yield clock.negedge # wait for negative edge

v = (v << 1) | b

print("{:3} {} | {} {} {}".format(int(state), int(b), int(next\_state), int(z), v))

# return all logic

return instances()

if \_\_name\_\_ == "\_\_main\_\_":

from myhdl import delay, instance, ResetSignal, always, StopSimulation

import argparse, re

# testbench itself is a block

@block

def test\_comb(args):

# reset signal level

ACTIVE\_LOW, INACTIVE\_HIGH = 0, 1

# create reset signal

reset = ResetSignal(0, active=ACTIVE\_LOW, isasync=True)

HALF\_PERIOD = delay(10)

# create clock signal

clock = Signal(bool(1))

# driving the clock

@always(HALF\_PERIOD)

def clockGen():

clock.next = not clock

# create signals

b, z = Signal(bool(0)), Signal(bool(0))

# instantiating a block and connect to signals

tut = Detect3x(z, b, clock, reset)

@instance

def stimulus():

print("state b | ns z v")

# release reset after a short delay

delay1 = delay(1)

yield delay1

reset.next = INACTIVE\_HIGH

for s in args.bits:

b.next = s == '1'

yield clock.posedge

# set b's value, 1ns after the positive edge

yield delay1

yield clock.negedge

yield delay1 # wait for monitor to print

raise StopSimulation()

return tut, clockGen, stimulus

parser = argparse.ArgumentParser(description='A state machine detecting 3x')

parser.add\_argument('bits', nargs='?', default="1001", help='bits to be shifted in')

parser.add\_argument('--trace', action='store\_true', help='generate trace file')

args = parser.parse\_args()

if not re.match(r"[01]+$", args.bits):

print("Error: bits can only be 0 or 1")

exit(1)

tb = test\_comb(args)

tb.config\_sim(trace=args.trace)

tb.run\_sim()

Step 2:

| state[1] | state[0] | b | next\_state[1] | next\_state | z |
| --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 |

Logic expression on Lines 35-46 in code.

Step 3:

Completed in Above Code

Step 4:

Completed in Above Code

Output when 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 | 0 1 113

0 0 | 0 1 226

0 1 | 1 0 453

1 1 | 0 1 907

1. Multiplier
2. Timing
   1. 1ns
   2. 3ns
   3. 13ns
   4. 13.5ns
3. 13.5ns
4. 74mhz
5. 74mhz

3. 5-bit multiplier

| Steps | Multiplicand | Multiplier | Product |
| --- | --- | --- | --- |
| init | 11011 | 10001 | 00000 |
| 1 | 11011 | 10001 | 11011 |
| 2 | 1011 | 1000 | 00000 |
| 3 | 011 | 100 | 00000 |
| 4 | 11 | 10 | 00000 |
| 5 | 1 | 1 | 11011 |

4.RISC-V assembly code

uint2decstr:

addi sp, sp, -8

sw, ra, 4(sp)

sw s0, 0(sp)

add s0, x0, a0

addi t0, 0, 10

blt a1, t0, less\_than\_10

addi sp, sp, -8

sw s0, 0(sp)

addi a1, a1, -10

jal ra, uint2decstr

addi sp, sp, 8

Less\_than\_10:

Modi t0, a1, 10

add a1, t0, x0

addi t0, a1, 48 #t0 = r + ‘0’

sw a0, 0(t0)

sw a0, 4(x0)

lw ra, 0(sp)

add sp, sp 8

jalr x0, ra, 0

5. Consider two processors

1. ((0.1\*1) + (0.2\*2) + (0.5\*3) + (0.2\*3)) = 2.6
2. ((0.1\*1) + (0.2\*1) + (0.5\*4) + (0.2\*5)) = 3.3
3. (2\*a) + (3\*b) = (2\*2.6)/(3\*3.3) = 0.53
4. ((0.1\*1) + (0.2\*1) + (0.5\*4) + (0.2\*2)) = 2.7
5. a/e = 3.3/2.7 =1.22

6.Methods to Accelerate a program

1. 1/((0.2/100) + 0.8) = 1.25
2. 1/((0.2/10) + (0.15/6) + 0.65) = 1.44
3. 1/((0.2/100) + (0.2/10) + (0.15/6) + 0.45) = 2.01
4. c/0.45 = 2.01/0.45 = 4.47