

**ECSE 222: Lab 3 Report**  
**Finite State Machines**

Group #48

Section 007

Dafne Culha (260785524)

Cheng Lin (260787697)

## **Table of Contents**

I	Introduction
II	Bi-Directional Counter Finite State Machine
	II.I    FSM design
	II.II   FSM simulation
III	Multi-Mode Counter
	III.I   Counter design
	III.II  Counter simulation
IV	FPGA Resource Utilization
V	RTL Schematic of multi-mode counter
VI	Conclusion

## I Introduction

This laboratory experiment gave us a better understanding of finite state machines (FSMs), Intel's Quartus Prime software, and the ModelSim software through the programming of a bi-directional counter and a multi-mode counter. We used the VHDL language, Quartus Prime's Test Bench Template Writer, ModelSim, and an Altera board to produce the circuits required. The final product was a programmed Altera DE1-SoC board that counts a certain sequence of numbers in either direction and can be controlled with start, stop, and reset pushbuttons.

## II Bi-Directional Counter Finite State Machine

The first circuit we designed was a bi-directional counter FSM that counts up/down in the following sequence:

$1 \leftrightarrow 2 \leftrightarrow 4 \leftrightarrow 8 \leftrightarrow 3 \leftrightarrow 6 \leftrightarrow 12 \leftrightarrow 11 \leftrightarrow 5 \leftrightarrow 10 \leftrightarrow 7 \leftrightarrow 14 \leftrightarrow 15 \leftrightarrow 13 \leftrightarrow 9 (\leftrightarrow 1 \leftrightarrow 2 \leftrightarrow 4 \dots)$

Furthermore, the counter will be controlled by an active high enable input, a direction input, and an active low reset input. When the reset input becomes active, the clock will become 1 if it is counting up, and will become 9 if it is counting down (this is indicated by the direction signal).

### II.I FSM design

Prior to designing the counter in Quartus Prime, we drew the state diagram for the circuit (Figure 1). Because the counter increments through a loop of 15 values, we designed the FSM to have 15 states labelled A through O. Each of these states corresponds to a unique output  $z$ , the 4-bit binary representation of the number the state represents. The edges between states are dictated by the value of the direction signal  $w$ . Lastly, as indicated in the diagram, the state the FSM adopts when reset is pressed is also dependent on the value of  $w$ .

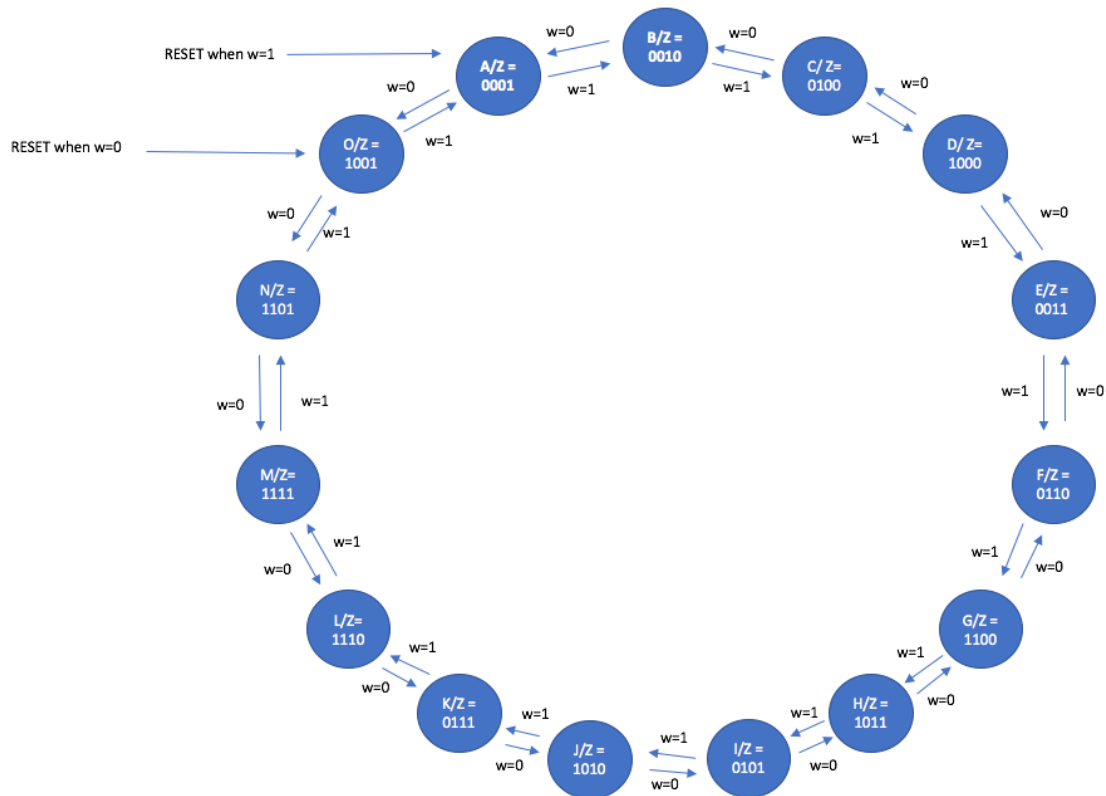


Fig. 1: State diagram for FSM.

Once we drew the state diagram, we began creating the circuit in VHDL. To begin, we created the entity declaration for the FSM, specifying the four inputs and single output of the circuit.

```

16 entity g48_FSM is
17   Port (enable : in std_logic;
18         direction : in std_logic;
19         reset : in std_logic;
20         clk : in std_logic;
21         count : out std_logic_vector(3 downto 0));
22
23 end g48_FSM;
24

```

Next, in the architecture of the counter, we declare a “state\_type” signal type and create a variable count\_temp that is of that type. Count\_temp can therefore take on values A through O, where each letter represents a state in the FSM.

```

26 --declare architecture for FSM
27 architecture behaviour of g48_FSM is
28     --create a state_type signal type to hold each of the counter's state
29     --each consecutive letter corresponds to the following number in the sequence
30     --A->1, B->2, C->4, D->8, etc.
31     type state_type is (A,B,C,D,E,F,G,H,I,J,K,L,M,N,O);
32
33     --create signal to hold state of count
34     signal count_temp : state_type := A;
35

```

Finally, in the second part of the architecture (below), we create a process that includes the clock and reset inputs as variables on its sensitivity list. This means that the process will execute when there is a change in either of those values. The clock is on the sensitivity list because the counter should increment up/down every clock cycle; the reset value is on the list because it is an asynchronous signal.

Within the process, we first check if the reset input is active (equals 0) and restart the count\_temp accordingly (depending on the value of the direction signal).

```

36 begin
37
38     --declare a process block since this is a sequential circuit
39     --define clk and reset in sensitivity list as variables we keep track of
40     --all other variables are synchronized with clk
41
42     Process(clk, reset) begin
43         --check the value of reset first because it is an asynchronous signal
44         --reset is active low
45         if(reset = '0') then
46             --if direction is 1, we are counting up
47             --reset counter to state A
48             if (direction = '1') then
49                 count_temp <= A;
50             --otherwise we are counting down, reset counter to state 0
51             else
52                 count_temp <= 0;
53             end if;
54

```

Otherwise, if the clock has a rising edge and our enable input is equal to 1, we increment count\_temp to the next state in the sequence (i.e.  $A \rightarrow B$ ,  $B \rightarrow C$ , etc., or  $O \rightarrow N$ ,  $N \rightarrow M$ , etc.). This next state is a function of the direction signal, which tells us if we are counting up or down.

We use a case statement to update the value of count\_temp. The logic for incrementing count\_temp when it equals A in the current cycle is shown below (lines 64-69); the control flow for all other state values are very similar.

```

54 |
55 | --check for rising edge of clk
56 | elsif(rising_edge(clk)) then
57 |
58 |     --check if enable is on (enable is active high)
59 |     if(enable = '1') then
60 |         --check for counting direction
61 |         --direction = 1 means count up, direction = 0 means count down)
62 |         case count_temp is
63 |             --when count_temp = A, we either increment to B or decrement to 0
64 |             when A =>
65 |                 if (direction = '1') then
66 |                     count_temp <= B;
67 |                 else
68 |                     count_temp <= 0;
69 |                 end if;
70 |             when B =>
71 |                 if (direction = '1') then
72 |                     count_temp <= C;

```

Lastly, outside of the process block, we associate a value to the output signal, count, for every state that count\_temp may take. This section of the code corresponds to the combinational circuit for the output logic of the FSM.

```

163 | --set count output for a given count_temp state
164 | count <= "0001" when count_temp = A else
165 |          "0010" when count_temp = B else
166 |          "0100" when count_temp = C else
167 |          "1000" when count_temp = D else
168 |          "0011" when count_temp = E else
169 |          "0110" when count_temp = F else
170 |          "1100" when count_temp = G else
171 |          "1011" when count_temp = H else
172 |          "0101" when count_temp = I else
173 |          "1010" when count_temp = J else
174 |          "0111" when count_temp = K else
175 |          "1110" when count_temp = L else
176 |          "1111" when count_temp = M else
177 |          "1101" when count_temp = N else
178 |          "1001" when count_temp = 0;
179 |
180 | end behaviour;

```

## II.II FSM simulation

We used ModelSim to test our FSM and simulate the values of inputs and outputs over time. We compiled our .vhd file on Quartus, added the same file to ModelSim, then used the Quartus Testbench Template Writer to create an empty testbench file. Lastly, we added simulation processes to the .vht file and simulated the circuit on ModelSim.

Firstly, we use the init process to loop over clk values.

```

60  --init process to loop clock values
61  init : PROCESS
62  BEGIN
63      --FPGA board's clock has 50 MHz frequency
64      --this means one period every 20 ns (so change clk every 10 ns)
65
66      clk <= '1';
67      WAIT FOR 1ns;
68      clk <= '0';
69      WAIT FOR 1ns;
70  END PROCESS init;

```

Then, we use the always process to test reset, enable and counting up / down functionalities of FSM.

```

76  BEGIN
77
78      --set initial values
79      reset <= '1';
80      enable <= '1';
81      direction <='1';
82
83      --test reset
84      WAIT FOR 45ns;
85      reset<='0';
86      WAIT FOR 10ns;
87      reset <= '1';
88
89      --test enable
90      WAIT FOR 34ns;
91      enable <= '0';
92      WAIT FOR 25ns;
93      enable <= '1';
94
95      --test direction
96      WAIT FOR 34ns;
97      direction <='0';
98      WAIT FOR 41ns;
99      reset<='0';
100     WAIT FOR 2ns;
101     reset <='1';
102
103     WAIT;
104     END PROCESS always;

```

Firstly, we set the initial values of reset, enable and direction as '1'.

```

76 BEGIN
77
78         --set initial values
79         reset <= '1';
80         enable <= '1';
81         direction <='1';
82

```

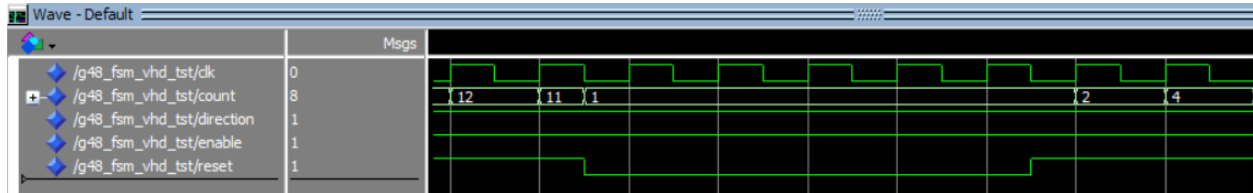
To test reset for both directions, we change its value from '1' to '0'.

```

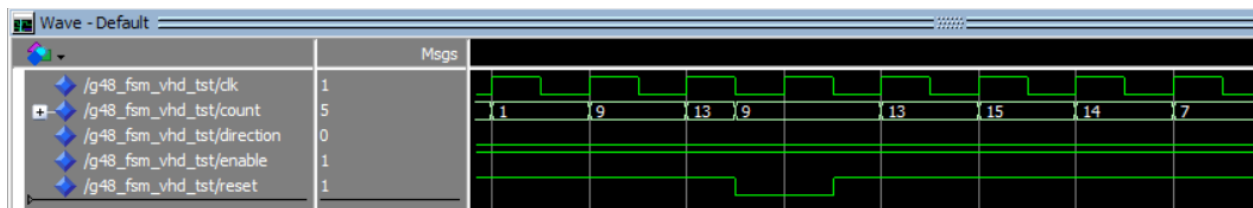
83         --test reset
84         WAIT FOR 45ns;
85         reset<='0';
86         WAIT FOR 10ns;
87         reset <= '1';

```

To test the reset function while counting upwards, we set the value of reset to '0' while direction has a value of '0'. As seen from the waveform below, when reset becomes active, the count returns to 1, as expected.



To test the reset function while counting downward, we set the value of reset as '0' while direction has a value of '1'. As can be seen from the waveform below, when reset becomes active, the count returns to 9.



Enable controls whether the count value increments or not each clock cycle. To test enable, we change its value from '0' to '1'.

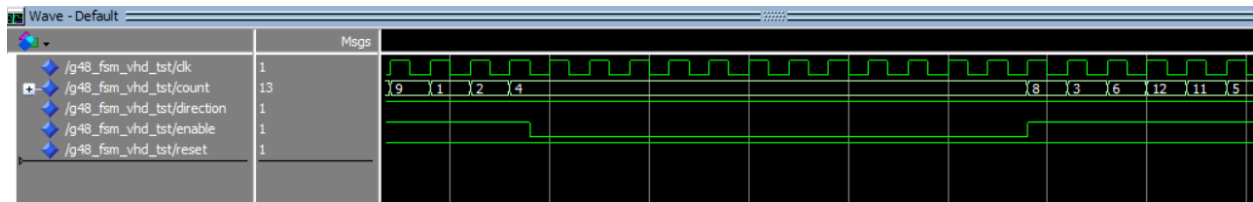


```

89          --test enable
90          WAIT FOR 34ns;
91          enable <= '0';
92          WAIT FOR 25ns;
93          enable <= '1';

```

As can be seen from the waveform below, the count value is no longer updated at the positive edge of the clock when enable is set to '0'. Furthermore, when enable is set back to '1', the count variable begins changing again.



To test if counting up/down functionalities of FSM works overall, we set the direction to '0' and '1' and wait long enough to see if it follows the correct patterns and if it loops:

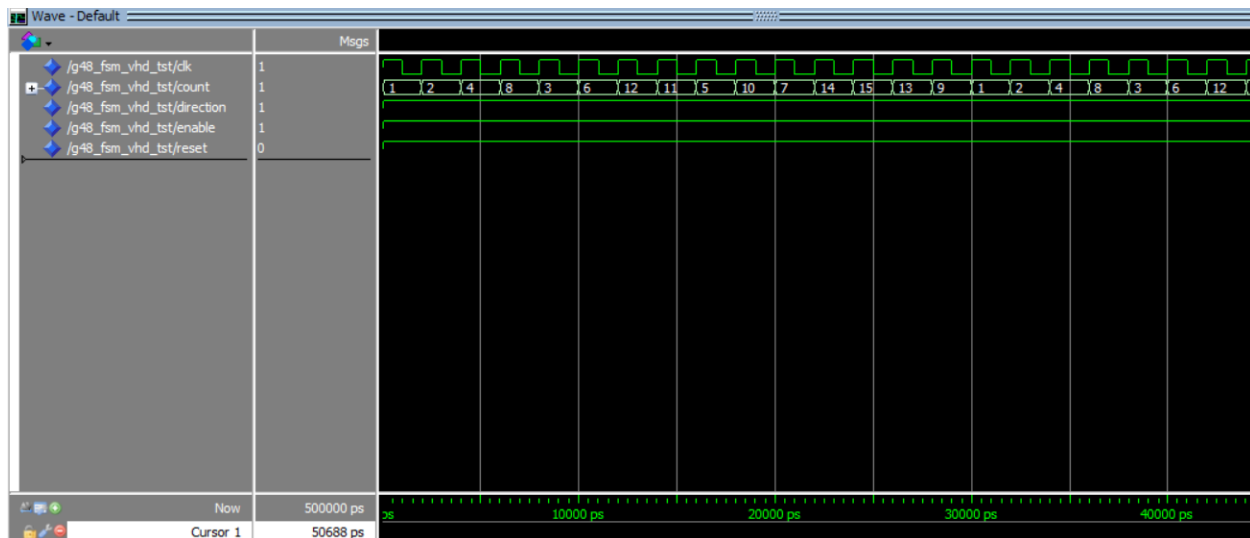
1↔2↔4↔8↔3↔6↔12↔11↔5↔10↔7↔14↔15↔13↔9 (↔1↔2↔4...)

```

95          --test direction
96          WAIT FOR 34ns;
97          direction <='0';
98          WAIT FOR 41ns;
99          reset<='0';
100         WAIT FOR 2ns;
101         reset <='1';

```

For example, when direction is set to '1', the count variable increments and loops through the cycle correctly.



A similar result is seen when we set the direction variable to '0'. Therefore, by running the FSM on ModelSim, we have confirmed that it works as expected.

### III Multi-Mode Counter

The second circuit we built was a multi-mode counter: a sequential circuit that accepts start, stop, and reset inputs to control the FSM that was designed in the previous section. This circuit will accept the Altera board's 50 MHz PLL clock and use a clock divider to assert a change in the FSM every second. Furthermore, the multi-mode counter will use two 7-segment decoders to show the output of the FSM as two-digit decimal numbers on 7-segment LEDs.

#### III.I Counter design

To begin, we declare the counter as an entity that accepts a start, stop, direction, reset, and clock input and returns two 7-bit vector outputs.

```

12  --declare an entity with five boolean inputs (start, stop, direction, reset, clk)
13  --and two 7-bit vector outputs for two 7-seg displays
14
15  entity g48_multimode_counter is
16  port
17      start : in std_logic;
18      stop  : in std_logic;
19      direction : in std_logic;
20      reset : in std_logic;
21      clk   : in std_logic;
22      HEX0  : out std_logic_vector(6 downto 0);
23      HEX1  : out std_logic_vector(6 downto 0);
24  end g48_multimode_counter;

```

In the architecture of the counter, we begin by importing the FSM component, the clock divider component, and the 7-segment decoder component in the workspace. It should be noted that the

clock divider component in this circuit has a generic T constant of 50000000 and will assert an en\_out of 1 every second given a 50 MHz clock input.

```

26  --declare architecture for FSM
27  architecture behaviour of g48_multimode_counter is
28  |
29  |      --import the FSM component
30  |      component g48_FSM is
31  |      |      Port (enable : in std_logic;
32  |      |            direction : in std_logic;
33  |      |            reset : in std_logic;
34  |      |            clk : in std_logic;
35  |      |            count : out std_logic_vector(3 downto 0));
36  |      end component g48_FSM;
37  |
38  |      --import the clock divider component
39  |      component g48_clock_divider is
40  |      |      Port(enable : in std_logic;
41  |      |            reset : in std_logic;
42  |      |            clk : in std_logic;
43  |      |            en_out : out std_logic);
44  |      end component g48_clock_divider;
45  |
46  |      --import 7 seg decoder component
47  |      component g48_7_segment_decoder is
48  |      |      Port(code: in std_logic_vector(3 downto 0);
49  |      |            segments: out std_logic_vector(6 downto 0));
50  |      end component g48_7_segment_decoder;
51  |

```

Then, we instantiate five signals: one signal to store the output of the clock divider, one signal to store the temporary enable input for the sequential circuit components, one signal to store the output of the FSM, and two signals to store the BCD digits that will be inputted into the 7-segment decoders.

```

52  --declare signal to hold clock from divider
53  signal divided_clk : std_logic;
54  --declare signal to hold temporary enable, instantiate as 0
55  signal enable_temp : std_logic := '0';
56
57  --declare signal to hold temporary count
58  signal count_temp : std_logic_vector(3 downto 0);
59  --declare signals to hold BCD digits of count_temp
60  signal digit0 : std_logic_vector(3 downto 0);
61  signal digit1 : std_logic_vector(3 downto 0);
62

```

Next, we instantiate a clock divider and FSM component in the architecture and specify their port maps. Note that both the clock\_divider and the FSM share the same enable and reset input. However, the output of the clock divider acts as the *clock* to the FSM component. This is because the FSM should see a rising edge once every second, the exact frequency in which the clock divider asserts an output of 1. Lastly, the output of the FSM goes to count\_temp, which will be used to obtain the digit0 and digit1 vector inputs later on.

```

63 begin
64
65 --create clock divider
66 --clock divider is controlled by reset, clk, and enable_temp variables
67 --it outputs divided_clk which acts as the clock for to FSM
68 clock_divider : g48_clock_divider PORT MAP(enable => enable_temp,
69                                             reset => reset,
70                                             clk => clk,
71                                             en_out => divided_clk);
72
73 --create an FSM component
74 FSM : g48_FSM PORT MAP(enable => enable_temp,
75                         direction => direction,
76                         reset => reset,
77                         clk => divided_clk,
78                         count => count_temp);
79

```

We now create a process that is sensitive to the start, stop, and count\_temp inputs. These variables are included on the sensitivity list because the counter needs to respond to any changes in these three variables. The start/stop variables appear because the enable\_temp signal is updated each time they change; the count\_temp variable is included because digit0 and digit1 need to be updated whenever count\_temp increments.

To set the enable\_temp value in the process block, we begin by checking whether start = 0 or stop = 0 (both are active low). Depending on whether start or stop are active, enable\_temp (an active high signal) is set to 1 or 0 respectively. The enable\_temp signal controls whether or not the clock divider and the FSM increment each clock cycle.

```

83 Process(start, stop, count_temp) begin
84
85 --check value of start (active low)
86 if(start = '0') then
87     enable_temp <= '1'; --if start = 0, enable stopwatch
88
89 --check value of stop (active low)
90 elsif(stop = '0') then
91     enable_temp <= '0'; --if stop = 0, turn enable off
92
93 end if;

```

Alternatively, if start and stop are both 1 and the process block is activated by a change in count\_temp, the circuit will update the values of digit0 and digit1. Because the outputs of HEX0 and HEX1 must correspond to a 7-segment display's decimal representation of count\_temp, digit0 and digit1 are encoded in BCD format.

To convert count\_temp into two decimal digits, we have to first check if count\_temp is greater than  $1001_2$  (or  $9_{10}$ ). If count\_temp is greater than 9, digit0 is corrected by adding  $0110_2$  (or  $6_{10}$ ) to it, and digit1 becomes  $0001_2$ . Alternatively, if count\_temp is not greater than 9, digit0 becomes count\_temp and digit1 becomes 0. These if-statements generate the correct inputs for the two 7-segment decoders used in the next code snippet.

```

94 |
95 |
96 |
97 |
98 |
99 |
100 |
101 |
102 |
103 |
104 |
105 |
106 |
107 |
108 |
109 |

```

```

--convert count_temp to BCD digits stored in digit0 and digit1
--start by checking if count_temp is greater than 9
if(count_temp > "1001")then
--if count_temp greater than 9, convert to BCD
  digit0 <= std_logic_vector(unsigned(count_temp) + "0110"); --conver to BCD by adding 6
  digit1 <= "0001";
else
--if count_temp is less than 9, set digit0 = count_temp and digit1 = 0
  digit0 <= count_temp;
  digit1 <= "0000";
end if;
end Process;

```

Finally, at the end of the architecture, we create two 7-segment decoder entities. The inputs to these decoders are digit0 and digit1, the two 4-bit BCD variables whose values are set in the process block. The outputs to these decoders are HEX0 and HEX1, which are fed into the 7-segment LED displays on the FPGA.

```

110 |
111 |
112 |
113 |
114 |
115 |
116 |
117 |

```

```

--decoder for counter 0
decoder0: g48_7_segment_decoder PORT MAP(code => digit0,
      segments => HEX0);
--decoder for counter 1
decoder1: g48_7_segment_decoder PORT MAP(code => digit1,
      segments => HEX1);
end behaviour;

```

### III.II Counter testing

We tested the inputs with a programmed Altera FPGA and verified the outputs displayed by the board. We pushed the start button and waited to make sure the values displayed are correct and the circuit loops in both directions. We checked if the buttons are working correctly and if our circuit started, stopped, and reset as expected. Lastly, we verified the values of the clock reset for both directions.

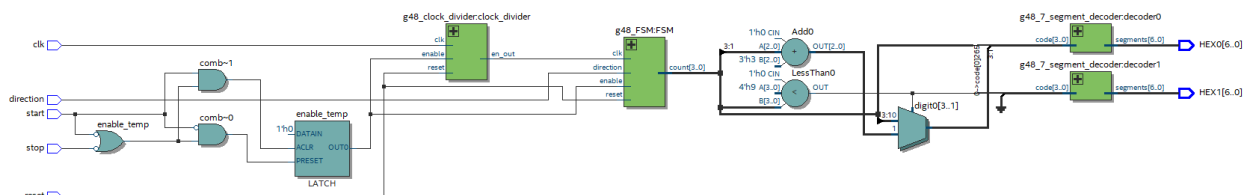
## IV FPGA Resource Utilization

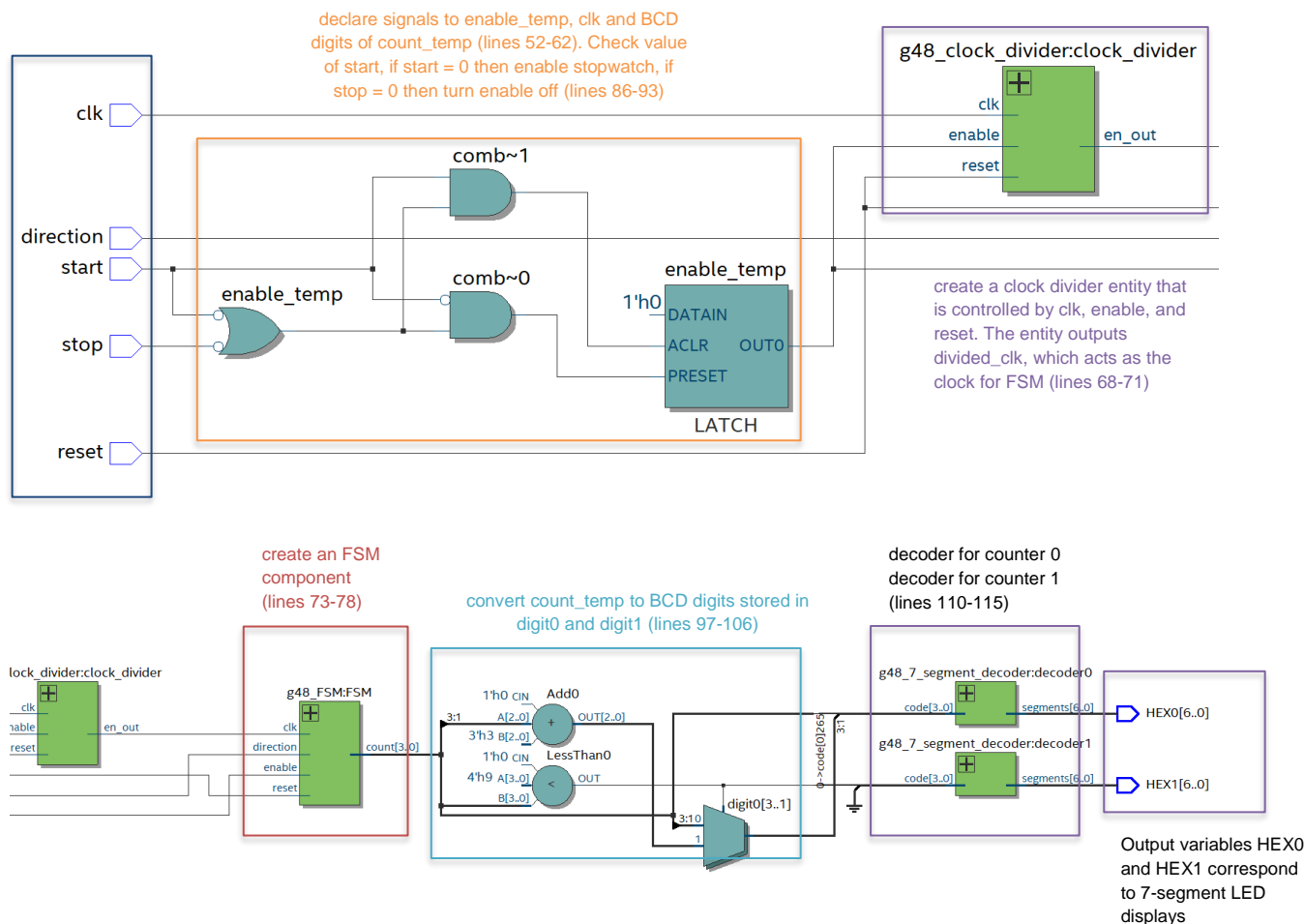
Flow Summary	
<<Filter>>	
Flow Status	Successful - Tue Apr 02 23:38:32 2019
Quartus Prime Version	18.0.0 Build 614 04/24/2018 SJ Lite Edition
Revision Name	g48_lab3
Top-level Entity Name	g48_multimode_counter
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	55 / 32,070 ( < 1 % )
Total registers	62
Total pins	19 / 457 ( 4 % )
Total virtual pins	0
Total block memory bits	0 / 4,065,280 ( 0 % )
Total DSP Blocks	0 / 87 ( 0 % )
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 ( 0 % )
Total DLLs	0 / 4 ( 0 % )

Logic utilization measures how much of the device is used to implement our circuit. As seen above, our compilation flow summary indicates that our logic utilization is less than 1%. This means that we have used very little of the board's total resources to implement our counter.

## V RTL Schematic of Multi-mode counter

The following is the RTL schematic of the multi-mode counter. For clarity, the schematic will be split up into two sections and each component's function (and corresponding lines of code) will be labelled on the visuals below. In summary, each green component represents a circuit entity (clock divider, FSM, or decoders). As well, the blue square is a register to store the enable input, and the other circuit components either check for the start/stop values or convert the temporary count signal to BCD format.





The following is another copy of our VHDL code for line references in the RTL diagram:

```

1  --ECSE 222 Lab 3
2  --Group 48
3  --Dafne Culha (260785524), Cheng Lin (260787697)
4  --01 Apr 2019
5
6
7  library IEEE;
8  use IEEE.STD_LOGIC_1164.ALL;
9  use IEEE.NUMERIC_STD.ALL;
10 use work.ALL;
11
12 --declare an entity with five boolean inputs (start, stop, direction, reset, clk)
13 --and two 7-bit vector outputs for two 7-seg displays
14
15 entity g48_multimode_counter is
16     Port (start : in std_logic;
17           stop : in std_logic;
18           direction : in std_logic;
19           reset : in std_logic;
20           clk : in std_logic;
21           HEX0 : out std_logic_vector(6 downto 0);
22           HEX1 : out std_logic_vector(6 downto 0));
23
24 end g48_multimode_counter;

```

```

25
26 --declare architecture for FSM
27 architecture behaviour of g48_multimode_counter is
28
29     --import the FSM component
30     component g48_FSM is
31         Port (enable : in std_logic;
32               direction : in std_logic;
33               reset : in std_logic;
34               clk : in std_logic;
35               count : out std_logic_vector(3 downto 0));
36     end component g48_FSM;
37
38     --import the clock divider component
39     component g48_clock_divider is
40         Port(enable : in std_logic;
41               reset : in std_logic;
42               clk : in std_logic;
43               en_out : out std_logic);
44     end component g48_clock_divider;
45
46     --import 7 seg decoder component
47     component g48_7_segment_decoder is
48         Port(code: in std_logic_vector(3 downto 0);
49               segments: out std_logic_vector(6 downto 0));
50     end component g48_7_segment_decoder;
51
52     --declare signal to hold clock from divider
53     signal divided_clk : std_logic;
54     --declare signal to hold temporary enable, instantiate as 0
55     signal enable_temp : std_logic := '0';
56
57     --declare signal to hold temporary count
58     signal count_temp : std_logic_vector(3 downto 0);
59     --declare signals to hold BCD digits of count_temp
60     signal digit0 : std_logic_vector(3 downto 0);
61     signal digit1 : std_logic_vector(3 downto 0);
62
63 begin
64
65     --create clock divider
66     --clock divider is controlled by reset, clk, and enable_temp variables
67     --it outputs divided_clk which acts as the clock for to FSM
68     clock_divider : g48_clock_divider PORT MAP(enable => enable_temp,
69                                                reset => reset,
70                                                clk => clk,
71                                                en_out => divided_clk);
72
73     --create an FSM component
74     FSM : g48_FSM PORT MAP(enable => enable_temp,
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```



```

80      --declare a process block since this is a sequential circuit with memory
81      --define start and stop in sensitivity list as variables we keep track of
82
83      Process(start, stop, count_temp) begin
84
85          --check value of start (active low)
86          if(start = '0') then
87              enable_temp <= '1'; --if start = 0, enable stopwatch
88
89          --check value of stop (active low)
90          elsif(stop = '0') then
91              enable_temp <= '0'; --if stop = 0, turn enable off
92
93          end if;
94
95          --convert count_temp to BCD digits stored in digit0 and digit1
96          --start by checking if count_temp is greater than 9
97          if(count_temp > "1001")then
98              --if count_temp greater than 9, convert to BCD
99              digit0 <= std_logic_vector(unsigned(count_temp) + "0110"); --conver to BCD by adding 6
100             digit1 <= "0001";
101
102          else
103              --if count_temp is less than 9, set digit0 = count_temp and digit1 = 0
104              digit0 <= count_temp;
105              digit1 <= "0000";
106          end if;
107
108      end Process;
109
110      --decoder for counter 0
111      decoder0: g48_7_segment_decoder PORT MAP(code => digit0,
112                                                  segments => HEX0);
113
114      --decoder for counter 1
115      decoder1: g48_7_segment_decoder PORT MAP(code => digit1,
116                                                  segments => HEX1);
117  end behaviour;

```

---

## VII Conclusion

For this laboratory experiment, we designed a finite state machine and a multi-mode counter on an Altera board which can count a certain sequence of numbers in either direction and can be controlled by start, stop, reset variables. Overall, this lab experiment gave us a better understanding of finite state machines (FSMs), Intel's Quartus Prime software, and the ModelSim software.