

ECE 111 Winter 2022

HW6

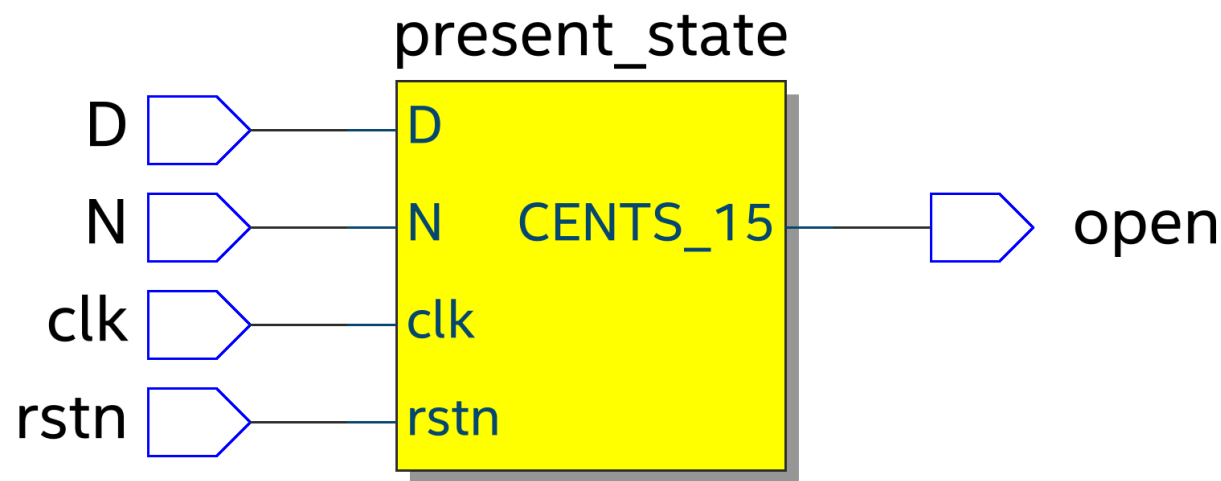
Hao Le A15547504

## Moore Vending Machine

Code

```
1 // Vending Machine RTL Code
2 module vending_machine_moore(
3     input logic clk, rstn,
4     input logic N, D,
5     output logic open);
6
7 // state variables and state encoding parameters
8 parameter [1:0] CENTS_0=2'b00, CENTS_5=2'b01, CENTS_10=2'b10, CENTS_15=2'b11;
9 logic [1:0] present_state, next_state;
10
11 // Sequential Logic for present state
12 always_ff @(posedge clk) begin
13     if (!rstn) begin
14         present_state <= CENTS_0;
15     end
16     else begin
17         present_state <= next_state;
18     end
19 end
20
21 // Combination Logic for Next State and Output
22 always_comb begin
23     case(present_state)
24         CENTS_0:begin
25             open = 0;
26             if (D==1) begin
27                 next_state = CENTS_10;
28             end
29             else if (N==1) begin
30                 next_state = CENTS_5;
31             end
32             else begin
33                 next_state = CENTS_0;
34             end
35         end
36         CENTS_5:begin
37             open = 0;
38             if (D==1) begin
39                 next_state = CENTS_15;
40             end
41             else if (N==1) begin
42                 next_state = CENTS_10;
43             end
44             else begin
45                 next_state = CENTS_5;
46             end
47         end
48         CENTS_10:begin
49             open = 0;
50             if (D==1) begin
51                 next_state = CENTS_15;
52             end
53             else if (N==1) begin
54                 next_state = CENTS_15;
55             end
56             else begin
57                 next_state = CENTS_10;
58             end
59         end
60         CENTS_15:begin
61             open = 1;
62             if (rstn) begin
63                 next_state = CENTS_15;
64             end
65             else begin
66                 next_state = CENTS_0;
67             end
68         end
69         default:begin
70             next_state = CENTS_0;
71             open = 0;
72         end
73     endcase
74 end
75 endmodule: vending_machine_moore
```

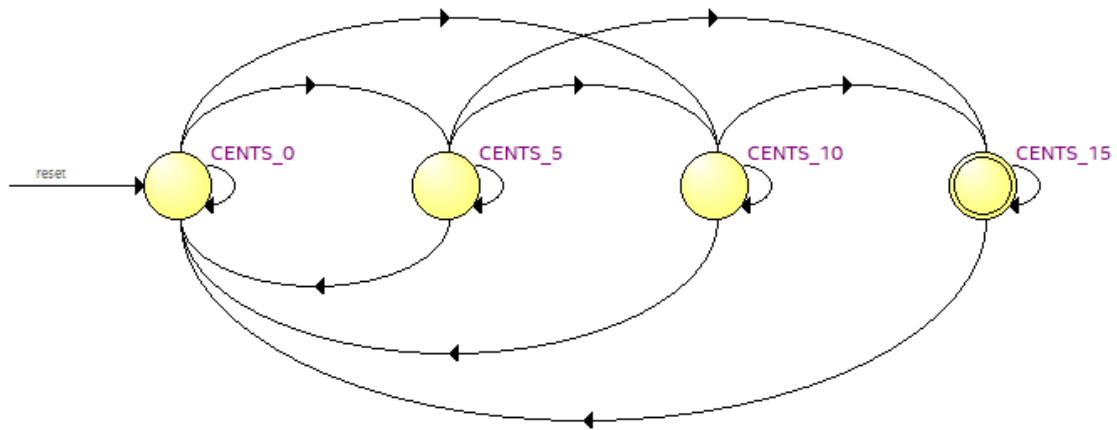
RTL netlist



Resource usage

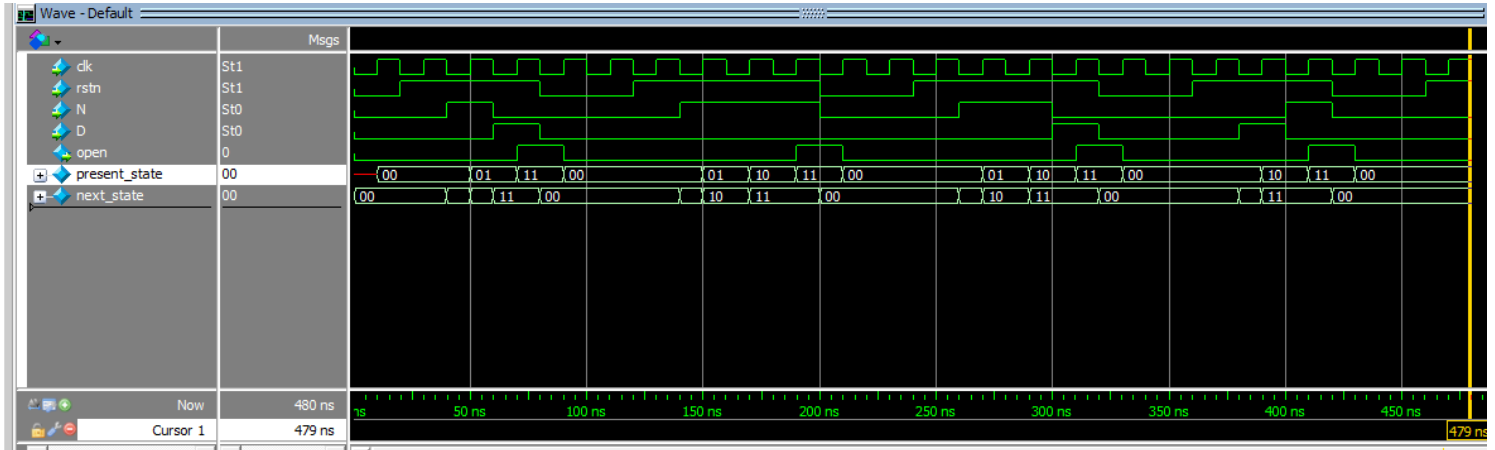
	resource	usage
1	Estimated ALUTs Used	4
1	-- Combinational ALUTs	4
2	-- Memory ALUTs	0
3	-- LUT_REGS	0
2	Dedicated logic registers	4
3		
4	Estimated ALUTs Unavailable	2
1	-- Due to unpartnered combinational logic	2
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	4
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	0
2	-- 6 input functions	2
3	-- 5 input functions	1
4	-- 4 input functions	1
5	-- <=3 input functions	0
8		
9	Combinational ALUTs by mode	
1	-- normal mode	4
2	-- extended LUT mode	0
3	-- arithmetic mode	0
4	-- shared arithmetic mode	0
10		
11	Estimated ALUT/register pairs used	6
12		
13	Total registers	4
1	-- Dedicated logic registers	4
2	-- I/O registers	0
3	-- LUT_REGS	0
14		
15		
16	I/O pins	5
17		
18	DSP block 18-bit elements	0
19		

## State diagram



1	CENTS_0	CENTS_0	(!rstn) + (rstn).(!N).(!D)
2	CENTS_0	CENTS_10	(D).(rstn)
3	CENTS_0	CENTS_5	(N).(!D).(rstn)
4	CENTS_5	CENTS_0	(!rstn)
5	CENTS_5	CENTS_10	(N).(!D).(rstn)
6	CENTS_5	CENTS_15	(rstn).(D)
7	CENTS_5	CENTS_5	(!N).(!D).(rstn)
8	CENTS_10	CENTS_0	(!rstn)
9	CENTS_10	CENTS_10	(!N).(!D).(rstn)
10	CENTS_10	CENTS_15	(rstn).(!N).(D) + (rstn).(N)
11	CENTS_15	CENTS_0	(!rstn)
12	CENTS_15	CENTS_15	(rstn)

## Testbench simulation waveform



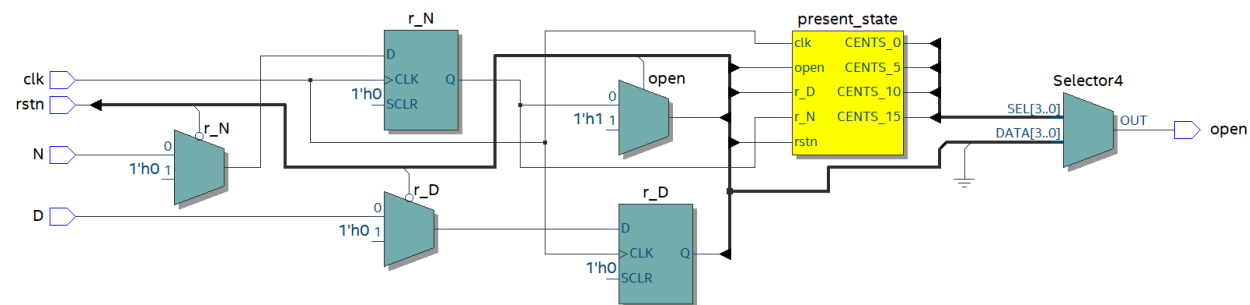
- A Moore state machine generates an output based on the current state, regardless of the inputs
- We see this behavior in the waveform; for example, in the first operation starting at around 30ns, a nickel is inserted, but since it was inserted at the falling edge of the clock, the state does not change from 00 to 01 (5 cents) until the next rising clock edge. We see this for the proceeding dime, where upon the rising edge, the state changes from 5 cents to 15 cents, thus bringing the output high
- This is distinct from a Mealy state machine in that the output did not go high immediately after the dime was inserted
- Lastly, once the 15 Cents (11) state has been reached, rstn goes low, but only at the rising clock edge does the present state switch back to 0 Cents, and the output goes low
- Described above is the correct behavior

# Mealy Vending Machine

## Code

```
1 // Vending Machine RTL Code
2 module vending_machine_mealy(
3     input logic clk, rstn,
4     input logic N, D,
5     output logic open);
6
7 // State encoding and state variables
8 parameter [1:0] CENTS_0=2'b00, CENTS_5=2'b01, CENTS_10=2'b10, CENTS_15=2'b11;
9 logic [1:0] present_state, next_state;
10
11 // Local variables for registering inputs N and D
12 logic r_N, r_D;
13
14 // Note : output open is not registered (i.e. no flipflop at output port open
15 // remember we learnt in class that mealy reacts immediately to change in inp
16 // Add flipflop for each input 'N' and 'D'
17 // Sequential Logic for present state
18 always_ff@(posedge clk) begin
19     if (!rstn) begin
20         present_state <= CENTS_0;
21         r_N <= 0;
22         r_D <= 0;
23     end
24     else begin
25         r_N <= N;
26         r_D <= D;
27         present_state <= next_state;
28     end
29 end
30
31 // Combination Logic for Next State and Output
32 always_comb begin
33     case(present_state)
34
35         CENTS_0:begin
36             if (r_D==1) begin
37                 next_state = CENTS_10;
38                 open = 0;
39             end
40             else if (r_N==1) begin
41                 next_state = CENTS_5;
42                 open = 0;
43             end
44             else begin
45                 next_state = CENTS_0;
46                 open = 0;
47             end
48         end
49
50         CENTS_5:begin
51             if (r_D==1) begin
52                 next_state = CENTS_15;
53                 open = 1;
54             end
55             else if (r_N==1) begin
56                 next_state = CENTS_10;
57                 open = 0;
58             end
59             else begin
60                 next_state = CENTS_5;
61                 open = 0;
62             end
63         end
64
65         CENTS_10:begin
66             if (r_D==1) begin
67                 next_state = CENTS_15;
68                 open = 1;
69             end
70             else if (r_N==1) begin
71                 next_state = CENTS_15;
72                 open = 1;
73             end
74             else begin
75                 next_state = CENTS_10;
76                 open = 0;
77             end
78         end
79
80         CENTS_15:begin
81             if (rstn) begin
82                 next_state = CENTS_15;
83                 open = 1;
84             end
85             else begin
86                 next_state = CENTS_0;
87                 open = 0;
88             end
89         end
90
91         default:begin
92             next_state = CENTS_0;
93             open = 0;
94         end
95     endcase
96 end
97 endmodule: vending_machine_mealy
```

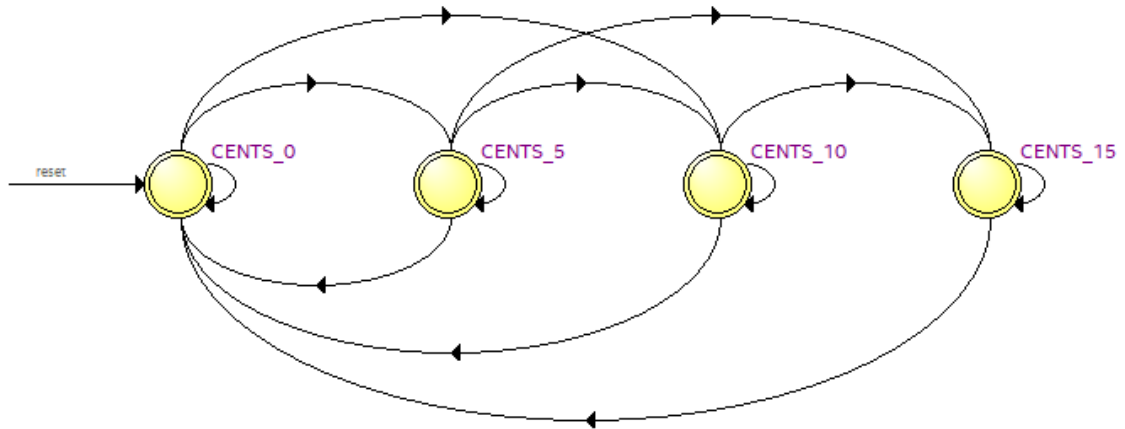
RTL netlist



Resource usage

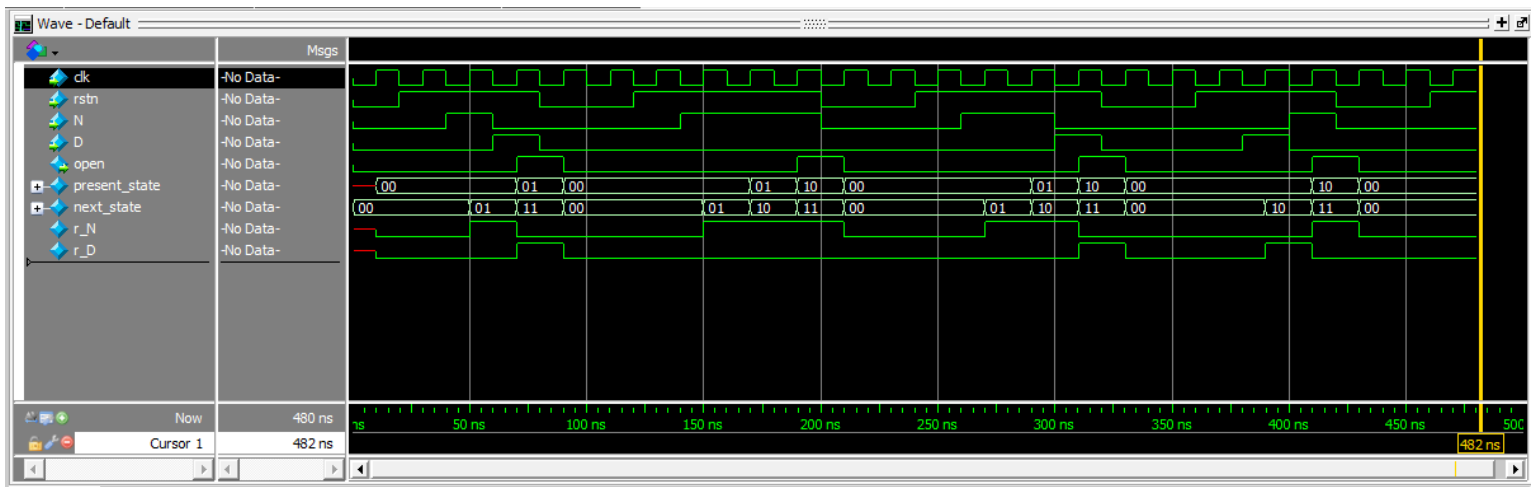
	Resource	Usage
1	Estimated ALUTs Used	7
1	-- Combinational ALUTs	7
2	-- Memory ALUTs	0
3	-- LUT_REGS	0
2	Dedicated logic registers	6
3		
4	Estimated ALUTs Unavailable	3
1	-- Due to unpartnered combinational logic	3
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	7
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	0
2	-- 6 input functions	3
3	-- 5 input functions	1
4	-- 4 input functions	1
5	-- <=3 input functions	2
8		
9	Combinational ALUTs by mode	
1	-- normal mode	7
2	-- extended LUT mode	0
3	-- arithmetic mode	0
4	-- shared arithmetic mode	0
10		
11	Estimated ALUT/register pairs used	10
12		
13	Total registers	6
1	-- Dedicated logic registers	6
2	-- I/O registers	0
3	-- LUT_REGS	0
14		
15		
16	I/O pins	5
17		
18	DSP block 18-bit elements	0

## State diagram



	Source State	Destination State	Condition
1	CENTS_0	CENTS_10	(r_D).(rstn)
2	CENTS_0	CENTS_5	(r_N).(!r_D).(rstn)
3	CENTS_0	CENTS_0	(!rstn) + (rstn).(!r_N).(!r_D)
4	CENTS_5	CENTS_15	(rstn).(r_D)
5	CENTS_5	CENTS_10	(r_N).(!r_D).(rstn)
6	CENTS_5	CENTS_5	(!r_N).(!r_D).(rstn)
7	CENTS_5	CENTS_0	(!rstn)
8	CENTS_10	CENTS_15	(rstn).(open)
9	CENTS_10	CENTS_10	(!r_N).(!r_D).(rstn)
10	CENTS_10	CENTS_0	(!rstn)
11	CENTS_15	CENTS_15	(rstn)
12	CENTS_15	CENTS_0	(!rstn)

Testbench simulation waveform



- A Mealy state machine generates an output based on the present state and the inputs; this differs from a Moore in that the output reacts immediately to a changing input
- In this implementation, flip flops were added to the inputs to delay them by half a clock cycle; this is to remedy the issue of double counting, especially during the third operation of the vending machine at around 260ns. As a side effect, the actual operation of this state machine is exactly like a Moore
- Without the flip flops, the inserted nickel would be counted three times, thus triggering the output before the dime even gets inserted which is incorrect behavior
- The “ideal” inputs would be the outputs of the registers
- Focusing on r\_N and r\_D as the inputs now, we see that at 50ns, the first nickel gets inserted, and the state is changed to 01 (5 Cents) at the positive clock edge
- Then, a dime is inserted, and the state changes to 11 (15 Cents) at the next positive clock edge
- We also observe that the output goes high as soon as the dime is inserted even though at that point, the present state is still 01 (5 Cents)
- This is because the Mealy machine knows the next state will be 15 Cents, so it triggers the output beforehand, saving a clock cycle.
- Described above is the correct behavior, assuming the inputs are timed correctly

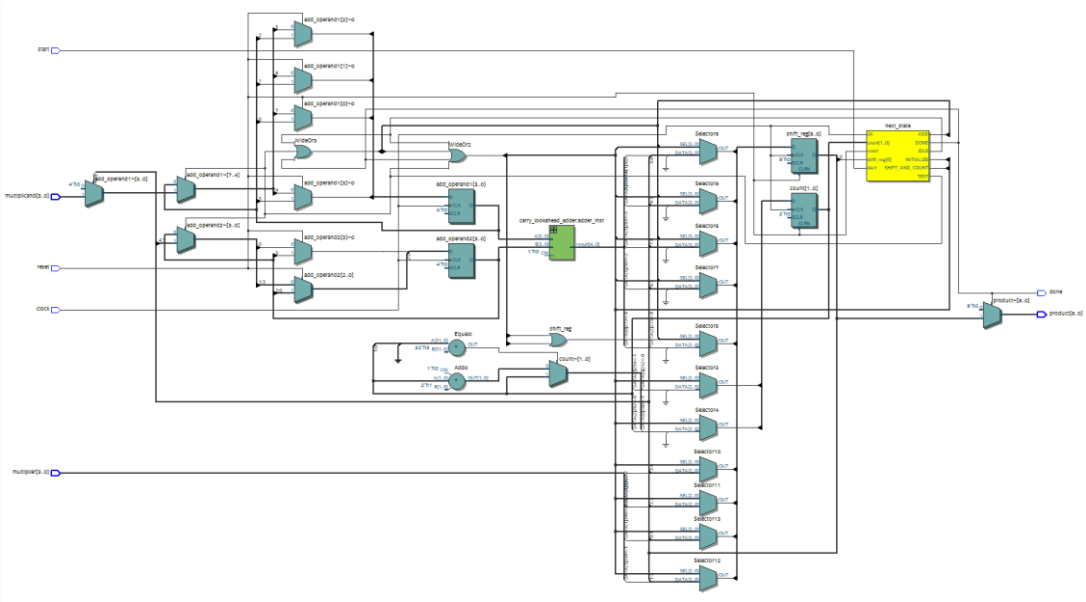


# Integer Multiplier

## Code

```
1 timescale 1ns/1ps
2 // include "carry_lookahead_adder.sv"
3
4 module integer_multiplier // Module start declaration
5 # (parameter N=4) // Parameter declaration
6
7 input clock, reset, start,
8 input logic[N-1:0] multiplicand, multiplier,
9 output logic[(2*N):0] product,
10 output logic done
11
12
13 // Count variable for ADD/SHIFT stages
14 logic [$clog2(N)-1:0] count;
15
16 // Register to load multiplicand value
17 logic[N-1:0] load_reg;
18
19 // Register to store Adder sum and multiplier
20 logic[(2*N):0] shift_reg;
21
22 // wires to connect with carry lookahead adder
23 logic[N-1:0] add_operand1, add_operand2;
24 logic[N:0] sum;
25
26 // next_state encoding and next_state variable
27 enum logic[2:0] {
28     IDLE = 3'b000,
29     INITIALIZE = 3'b001,
30     TEST = 3'b010,
31     ADD = 3'b011,
32     SHIFT_AND_COUNT = 3'b100,
33     DONE = 3'b101
34 } next_state;
35
36 // Instantiate N-bit carry lookahead adder
37 // Pass add_operand1, add_operand2 and sum
38 // Tie CIN to '0'
39
40 carry_lookahead_adder #(N(N)) adder_inst (
41     .A(add_operand1),
42     .B(add_operand2),
43     .CIN(1'b0),
44     .result(sum)
45 );
46
47 // Control FSM for Integer Multiplier
48 // use single always block FSM approach
49 // use "only" non-blocking assignment statements within always block
50 always_ff @(posedge clock, posedge reset) begin
51     if (reset) begin
52         count <= 0;
53         next_state <= IDLE;
54         load_reg <= 0;
55         shift_reg <= 0;
56     end
57     else begin
58         case (next_state)
59         IDLE: begin
60             if (!start) begin
61                 next_state <= IDLE;
62             end
63             else begin
64                 next_state <= INITIALIZE;
65             end
66         INITIALIZE: begin
67             count <= 0;
68             shift_reg <= {1'b0, {N{1'b0}}, multiplier};
69             next_state <= TEST;
70         TEST: begin
71             if (shift_reg[0] == 1'b1) begin
72                 add_operand1 <= multiplicand;
73                 add_operand2 <= shift_reg[(2*N)-1:N];
74                 next_state <= ADD;
75             end
76             else begin
77                 add_operand1 <= 0;
78                 add_operand2 <= shift_reg[(2*N)-1:N];
79                 next_state <= SHIFT_AND_COUNT;
80             end
81         ADD: begin
82             shift_reg <= {sum, shift_reg[N-1:0]};
83             next_state <= SHIFT_AND_COUNT;
84         SHIFT_AND_COUNT: begin
85             if (count == N-1) begin
86                 shift_reg <= shift_reg >> 1;
87                 next_state <= DONE;
88             end
89             else begin
90                 shift_reg <= shift_reg >> 1;
91                 count <= count + 1;
92                 next_state <= TEST;
93             end
94         DONE: begin
95             next_state <= IDLE;
96         end
97     end
98 end
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 // Generate done=1 when FSM reaches DONE state
133 assign done = (next_state == DONE) ? 1 : 0;
134
135 // Generate Product in DONE state by loading shift_reg value to it
136 assign product = (next_state == DONE) ? shift_reg : 0;
137
138 endmodule: integer_multiplier
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
```

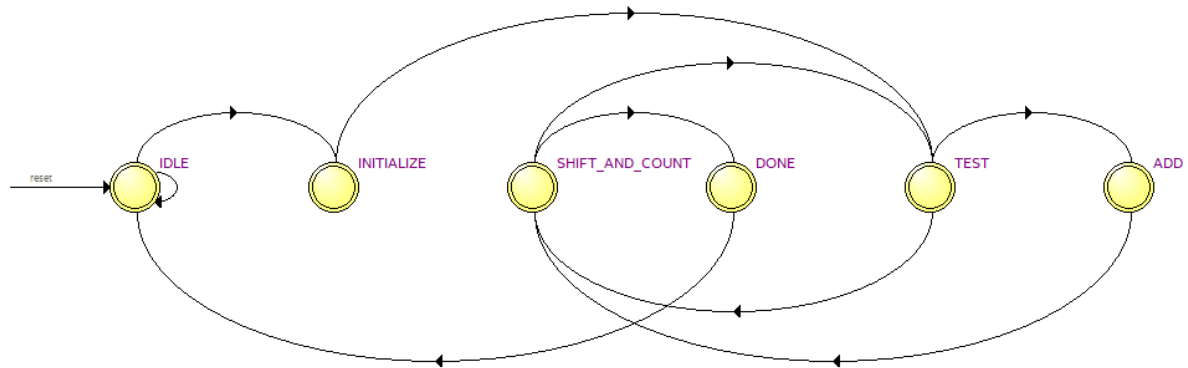
RTL netlist



Resource usage

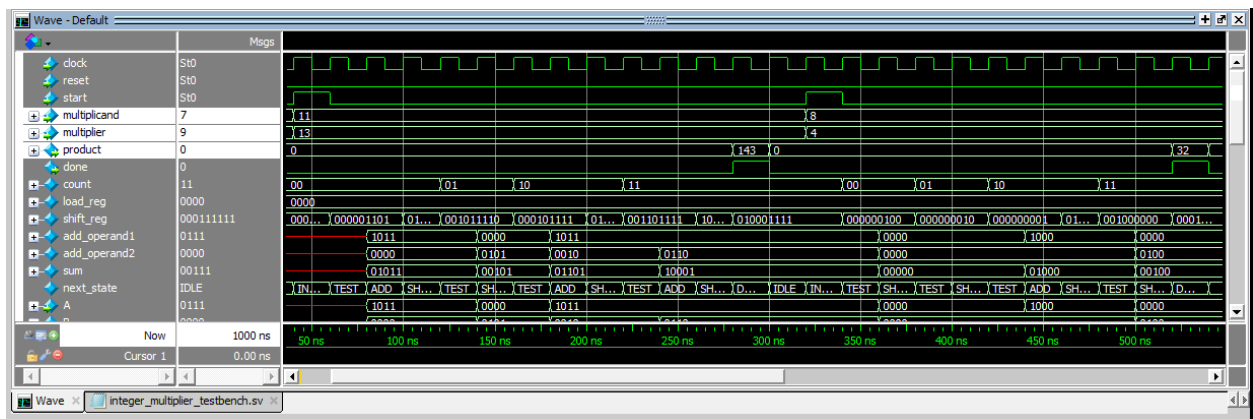
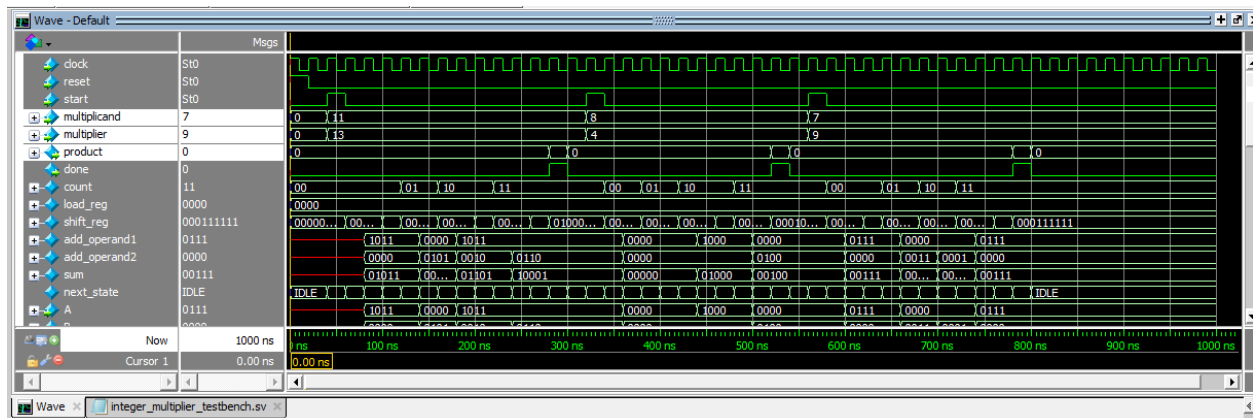
	Resource	Usage
1	Estimated ALUTs Used	38
1	-- Combinational ALUTs	38
2	-- Memory ALUTs	0
3	-- LUT_REGS	0
2	Dedicated logic registers	25
3		
4	Estimated ALUTs Unavailable	1
1	-- Due to unpartnered combinational logic	1
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	38
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	1
2	-- 6 input functions	2
3	-- 5 input functions	2
4	-- 4 input functions	7
5	-- <=3 input functions	26
8		
9	Combinational ALUTs by mode	
1	-- normal mode	37
2	-- extended LUT mode	1
3	-- arithmetic mode	0
4	-- shared arithmetic mode	0
10		
11	Estimated ALUT/register pairs used	42
12		
13	Total registers	25
1	-- Dedicated logic registers	25
2	-- I/O registers	0
3	-- LUT_REGS	0
14		
15		
16	I/O pins	21
17		
18	DSP block 18-bit elements	0
19		

## State diagram



	Source State	Destination State	Condition
1	ADD	SHIFT_AND_COUNT	
2	DONE	IDLE	
3	IDLE	INITIALIZE	(start)
4	IDLE	IDLE	(!start)
5	INITIALIZE	TEST	
6	SHIFT_AND_COUNT	TEST	(!count[0]) + (count[0]).(!count[1])
7	SHIFT_AND_COUNT	DONE	(count[0]).(count[1])
8	TEST	SHIFT_AND_COUNT	(!shift_reg[0])
9	TEST	ADD	(shift_reg[0])

## Testbench simulation waveform



```

VSIM8> run -all
# time=280000 Multiplicand=11 Multiplier=13 Product=143 Done=1 Correct Result
#
# time=520000 Multiplicand= 8 Multiplier= 4 Product= 32 Done=1 Correct Result
#
# time=780000 Multiplicand= 7 Multiplier= 9 Product= 63 Done=1 Correct Result
#

```

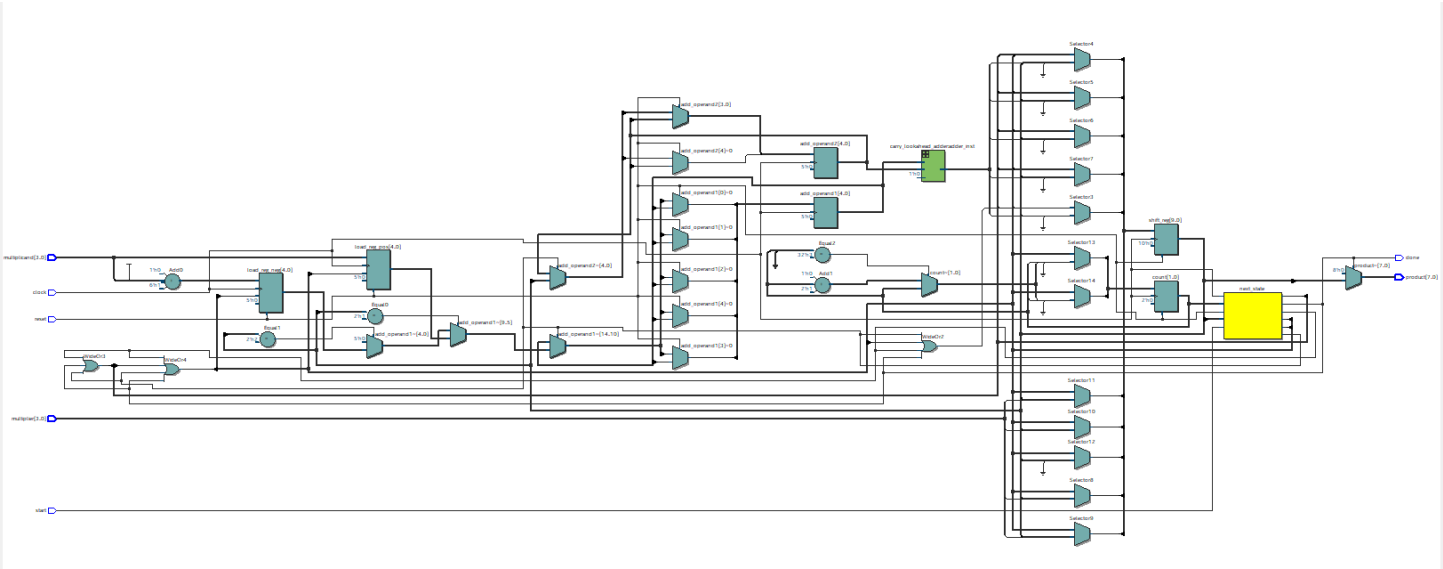
- The waveforms shown are of 3 views: the overall waveform, a zoomed in section to show the correct results from input pairs, and the transcript to demonstrate that each case yielded the correct result
- The multiplier behaves correctly because upon loading two numbers to be multiplied, and by setting start to high, this starts the internal cycle of states, and once the final product has been calculated, it is outputted with the done signal going high
- Then upon the next clock cycle, product resets to 0, and the multiplier idles until a new pair of numbers is loaded along with the pull up of the start input

# Booth Multiplier

## Code

```
1 timescale 1ns/1ps
2 //include "carry_lookahead_adder.sv"
3
4 module booth_multiplier // Module start declaration
5 #(parameter N=4) // Parameter declaration
6
7 (
8     input clock, reset, start,
9     input logic signed [N-1:0] multiplicand, multiplier,
10    output logic signed [(2*N)-1:0] product,
11    output logic done
12 );
13
14 //Variable to store 2's complement of Multiplicand
15 logic [N:0] multiplicand_neg;
16
17 // Count variable for ADD/SHIFT stages
18 logic [$clog2(N)-1:0] count;
19
20 // Register to store Adder sum and multiplier
21 logic signed [(2*N)+1:0] shift_reg;
22
23 // Register to load multiplicand value
24 logic signed [N:0] load_reg_pos;
25 logic signed [N:0] load_reg_neg;
26
27 // wires to connect with carry lookahead adder
28 logic [N:0] add_operand1, add_operand2;
29 logic c1a_carry;
30
31 // next_state encoding and next_state variable
32 enum logic [2:0] {
33     IDLE = 3'b000,
34     INITIALIZE = 3'b001,
35     TEST = 3'b010,
36     ADD = 3'b011,
37     SHIFT_AND_COUNT = 3'b100,
38     DONE = 3'b101
39 } next_state;
40
41 // Instantiate (N+1)-bit carry lookahead adder
42 //Use add_operand1, add_operand2, sum to connect carry lookahead adder
43 //Hint: Carry out from the adder is ignored in our calculations and output sum
44 //has same length as add_operand1 and add_operand2
45 // Tie CIN to '0'
46 carry_lookahead_adder #(N(N+1)) adder_inst(
47     .A(add_operand1),
48     .B(add_operand2),
49     .CIN(1'b0),
50     .result(sum)
51 );
52
53 // Create negative multiplicand value
54 assign multiplicand_neg = ~multiplicand;
55
56 always_ff@(posedge clock, posedge reset) begin
57     if(reset) begin
58         count <= 0;
59         next_state <= IDLE;
60         load_reg_pos <= 0;
61         load_reg_neg <= 0;
62         shift_reg <= 0;
63     end
64
65     else begin
66         case(next_state)
67             IDLE: begin
68                 if(!start) begin
69                     next_state <= IDLE;
70                 end
71                 else begin
72                     next_state <= INITIALIZE;
73                 end
74             end
75             INITIALIZE: begin
76                 load_reg_pos <= multiplicand;
77                 load_reg_neg <= multiplicand_neg[N:0];
78                 shift_reg <= {1'b0, {N{1'b0}}}, multiplier, 1'b0;
79                 next_state <= TEST;
80                 count <= 0;
81             end
82             TEST: begin
83                 if(shift_reg[1:0] == 2'b01) begin
84                     next_state <= ADD;
85                     add_operand1 <= load_reg_pos;
86                     add_operand2 <= shift_reg[(2*N)+1:(2*N)-3];
87                 end
88                 else if(shift_reg[1:0] == 2'b10) begin
89                     next_state <= ADD;
90                     add_operand1 <= load_reg_neg;
91                     add_operand2 <= shift_reg[(2*N)+1:(2*N)-3];
92                 end
93                 else begin
94                     next_state <= SHIFT_AND_COUNT;
95                     add_operand1 <= {sum, shift_reg[N:0]};
96                 end
97             end
98             ADD: begin
99                 next_state <= SHIFT_AND_COUNT;
100                 shift_reg <= {sum, shift_reg[N:0]};
101             end
102             SHIFT_AND_COUNT: begin
103                 shift_reg <= (shift_reg >> 1); // Right Arithmetic shift e
104                 if(count == N-1) begin // If 'N' times SHIFT operation perf
105                     next_state <= DONE;
106                 end
107                 else begin
108                     next_state <= TEST;
109                     count <= count + 1;
110                 end
111             end
112         endcase
113     end
114
115     done = (next_state == DONE) ? 1 : 0;
116     assign product = (next_state == DONE) ? {shift_reg[(2*N)], shift_reg[(2*N):1]} : 0;
117 endmodule: booth_multiplier
```

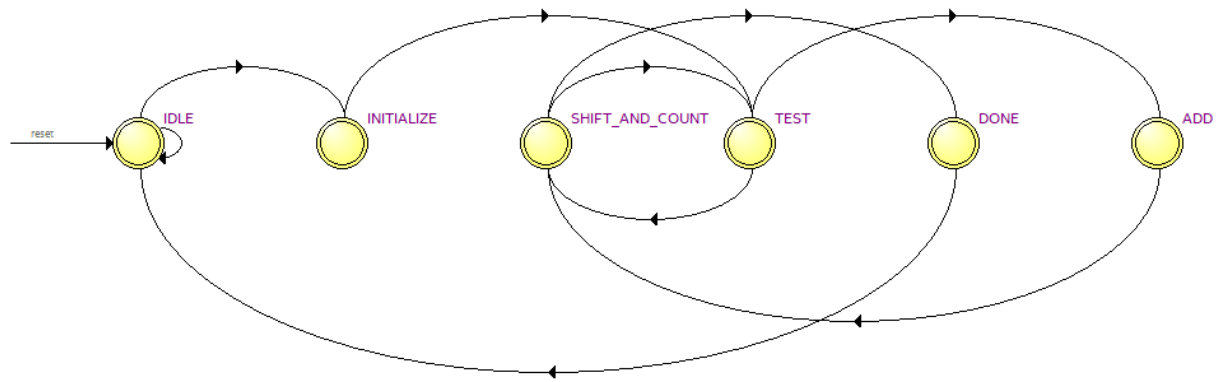
RTL netlist



Resource usage

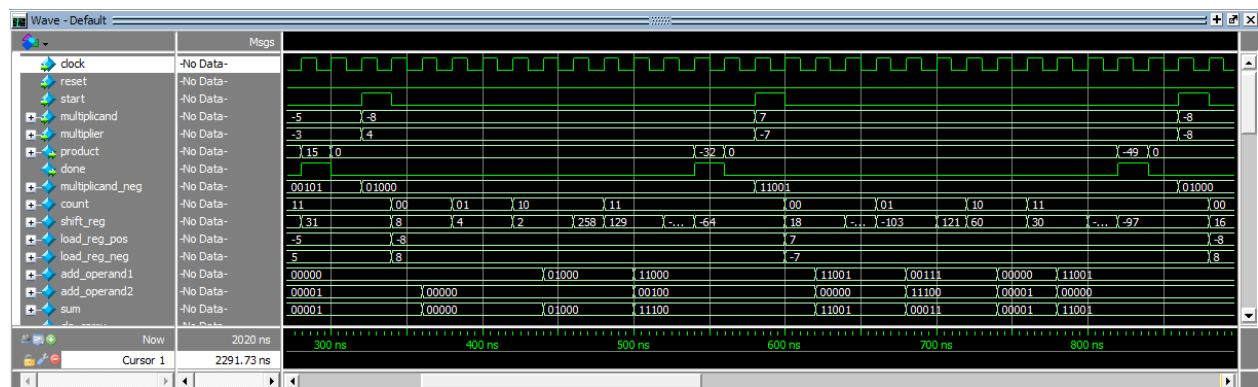
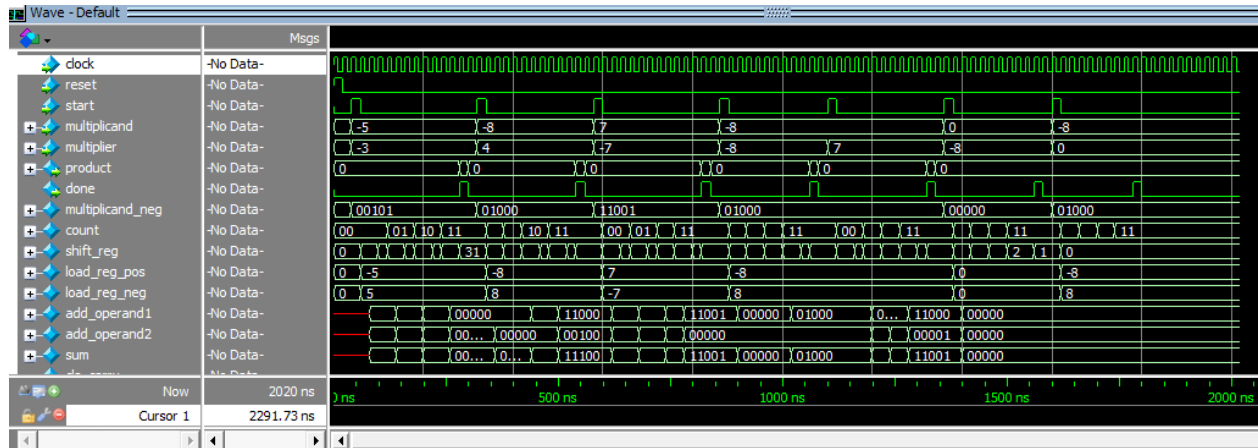
	Resource	Usage
1	Estimated ALUTs Used	43
1	-- Combinational ALUTs	43
2	-- Memory ALUTs	0
3	-- LUT_REGS	0
2	Dedicated logic registers	36
3		
4	Estimated ALUTs Unavailable	1
1	-- Due to unpartnered combinational logic	1
2	-- Due to Memory ALUTs	0
5		
6	Total combinational functions	43
7	Combinational ALUT usage by number of inputs	
1	-- 7 input functions	1
2	-- 6 input functions	3
3	-- 5 input functions	2
4	-- 4 input functions	13
5	-- <=3 input functions	24
8		
9	Combinational ALUTs by mode	
1	-- normal mode	42
2	-- extended LUT mode	1
3	-- arithmetic mode	0
4	-- shared arithmetic mode	0
10		
11	Estimated ALUT/register pairs used	49
12		
13	Total registers	36
1	-- Dedicated logic registers	36
2	-- I/O registers	0
3	-- LUT_REGS	0
14		
15		
16	I/O pins	20
17		
18	DSP block 18-bit elements	0
19		

## State diagram



	Source State	Destination State	Condition
1	ADD	SHIFT_AND_COUNT	
2	DONE	IDLE	
3	IDLE	INITIALIZE	(start)
4	IDLE	IDLE	(!start)
5	INITIALIZE	TEST	
6	SHIFT_AND_COUNT	TEST	(!count[0]) + (count[0]).(!count[1])
7	SHIFT_AND_COUNT	DONE	(count[0]).(count[1])
8	TEST	SHIFT_AND_COUNT	(!shift_reg[0]).(!shift_reg[1]) + (shift_reg[0]).(shift_reg[1])
9	TEST	ADD	(!shift_reg[0]).(shift_reg[1]) + (shift_reg[0]).(!shift_reg[1])

## Testbench simulation waveform



```

VSIM 8> run -all
# time=280000 Multiplicand= -5 Multiplier= -3 Product= 15 Done=1 Correct Result
#
# time=540000 Multiplicand= -8 Multiplier= 4 Product= -32 Done=1 Correct Result
#
# time=820000 Multiplicand= 7 Multiplier= -7 Product= -49 Done=1 Correct Result
#
# time=1060000 Multiplicand= -8 Multiplier= -8 Product= 64 Done=1 Correct Result
#
# time=1320000 Multiplicand= -8 Multiplier= 7 Product= -56 Done=1 Correct Result
#
# time=1560000 Multiplicand= 0 Multiplier= -8 Product= 0 Done=1 Correct Result
#
# time=1780000 Multiplicand= -8 Multiplier= 0 Product= 0 Done=1 Correct Result
#
A ** Note: $finish : C:/Banco/EECE 111/HW5/Tab5/booth_multiplier/booth_multiplier_testbench

```



- The waveforms shown are of 3 views: the overall waveform, the zoomed in waveform to show 3 cases and their outputs, and the transcript to demonstrate that each case yielded the correct result
- The multiplier behaves correctly because upon loading two numbers to be multiplied, and by setting start to high, this starts the internal cycle of states, and once the final product has been calculated, it is outputted with the done signal going high
- Then upon the next clock cycle, product resets to 0, and the multiplier idles until a new pair of numbers is loaded along with the pull up of the start input