Lichen Liu
1001721498

Assignment 1

ECE 1756H
October 5, 2022

1 Block Diagrams

1.1 Pipelined

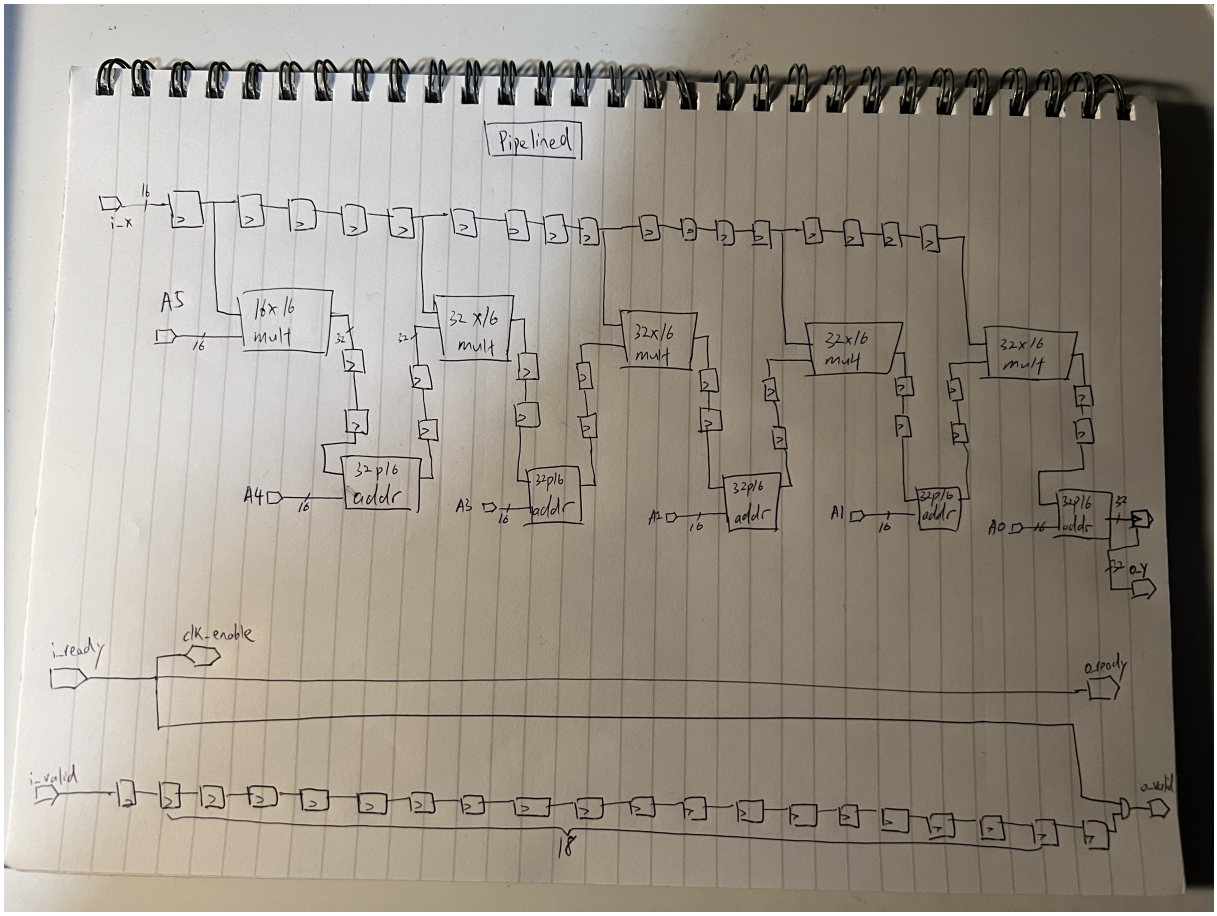

Figure 1: The 18 stages pipelined design, with Fmax of 425.17 MHz, throughput 1/cycle

In the pipelined version, a total of 18 stages of pipeline registers are added on top of the baseline version. In the computation path, pipeline stages are added to both the input and the output of the mult or addr module, except for the first mult and the last addr, where the pipeline stage is only added to their output and input respectfully. As a result, this includes a total of 18 pipeline stages to the design, on top of the few existing interface level registers. To make the rest of the circuit functional, 16 pipeline stages are added to the path of logic **i_x**, and 18 pipeline stages are added to the path of logic **i_valid**.

It is worth noting that 2 pipeline stages are added between the mult and the addr module (or vice versa), rather than 1 pipeline stage. This is to fully allow Quartus to absorb registers into the DSP, while also serving to hide the latency between the DSP and its fanouts (LABs). It is important to add a register right before the post-DSP logics, which is the critical path of

the design. On this critical path, it is interesting to see that Quartus Synthesis has actually created two separate adder chains, with the first one responsible for summing up the partial multiplication results from two DSPs in parallel, while the second one is the addr after the mult module. Unfortunately, I could not figure out how to force Quartus Synthesis to map the mult output register or the addr input register to be inserted between the two adder chains, therefore it is crucial to register this two adder chains path as tightly as possible.

1.2 SharedHW

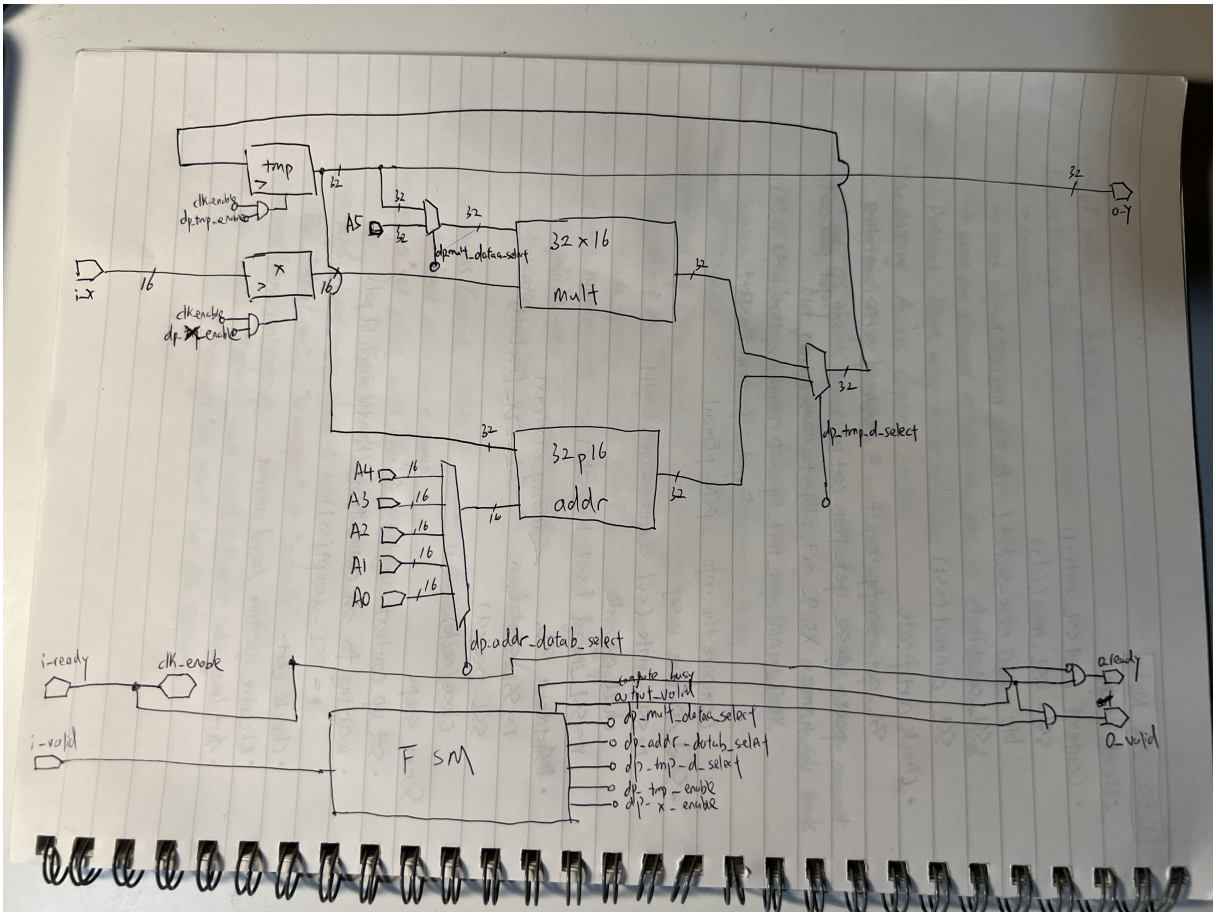

Figure 2: The Finite State Machine with Datapath (FSMD) shared hardware design, with Fmax of 179.66 MHz, throughput 0.1/cycle

In the FSMD shared hardware version, a single 32x16 multiplier and a single 32p16 adder is used. The datapath is consisted of two registers, one for storing the logic **i_x**, and the other one for storing the temporary calculation result and is used as the logic **o_y** as well. The entire datapath has five control signals driven by the controller implemented as FSM, whose

sole input is the logic **i_valid**. The FSM also computes the logic **o_valid** and **o_ready**. The entire design uses logic **i_ready** as clk enable signal.

2 Simulation Waveforms and Testbench Output

2.1 Pipelined

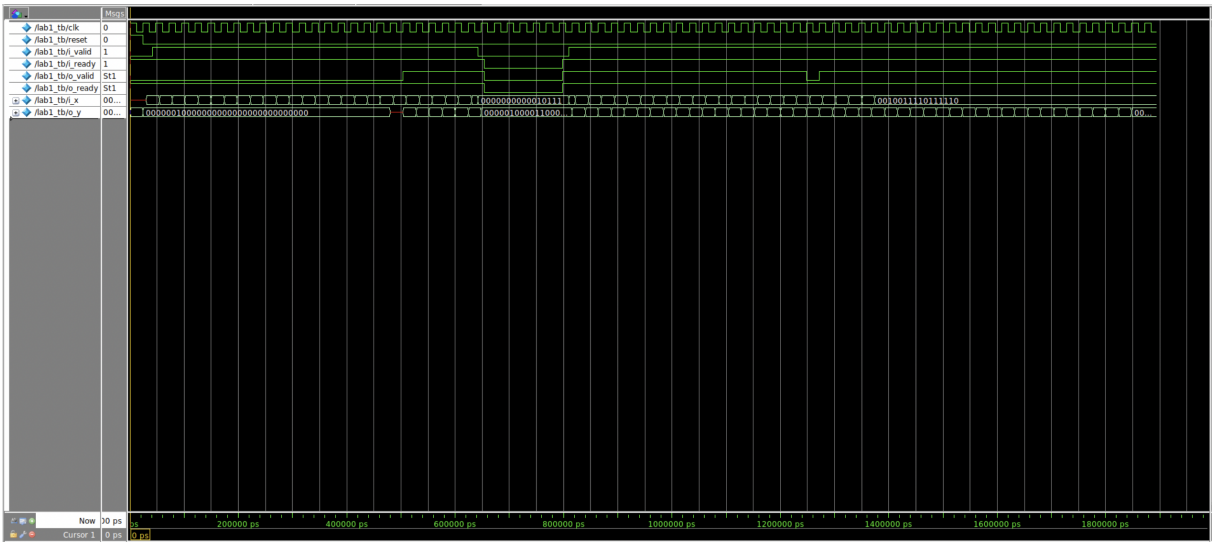

Figure 3: ModelSim waveforms for the pipelined design as expected

Assignment 1

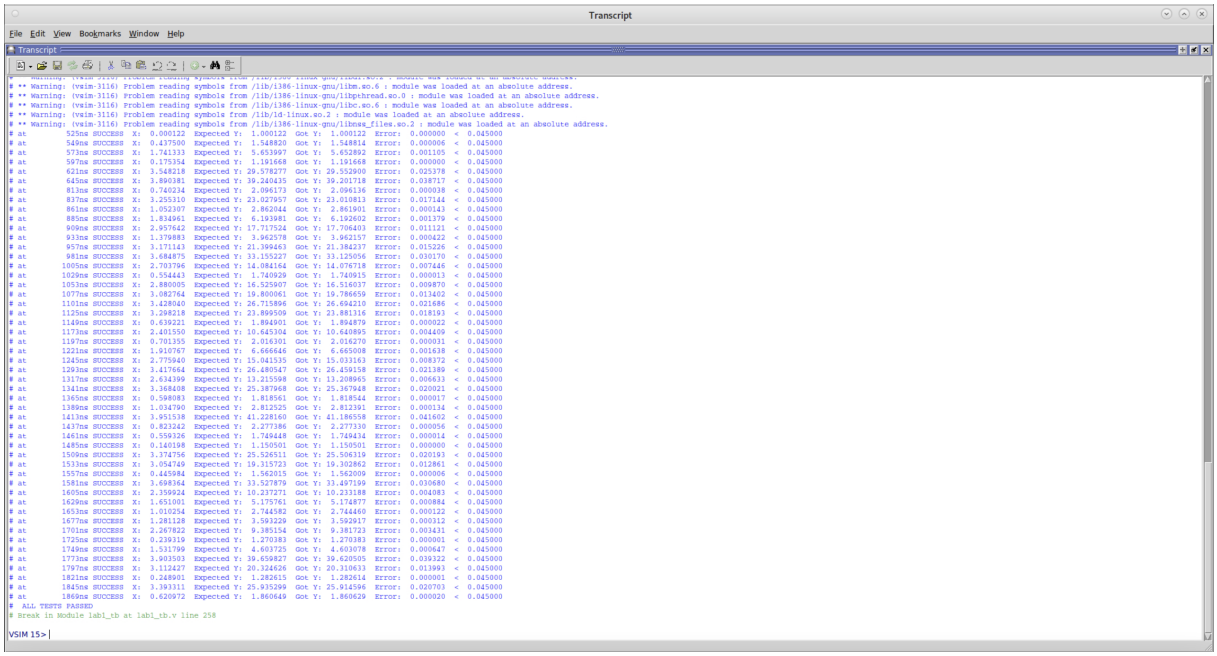

Figure 4: ModelSim simulation log for the pipelined design passing all test cases

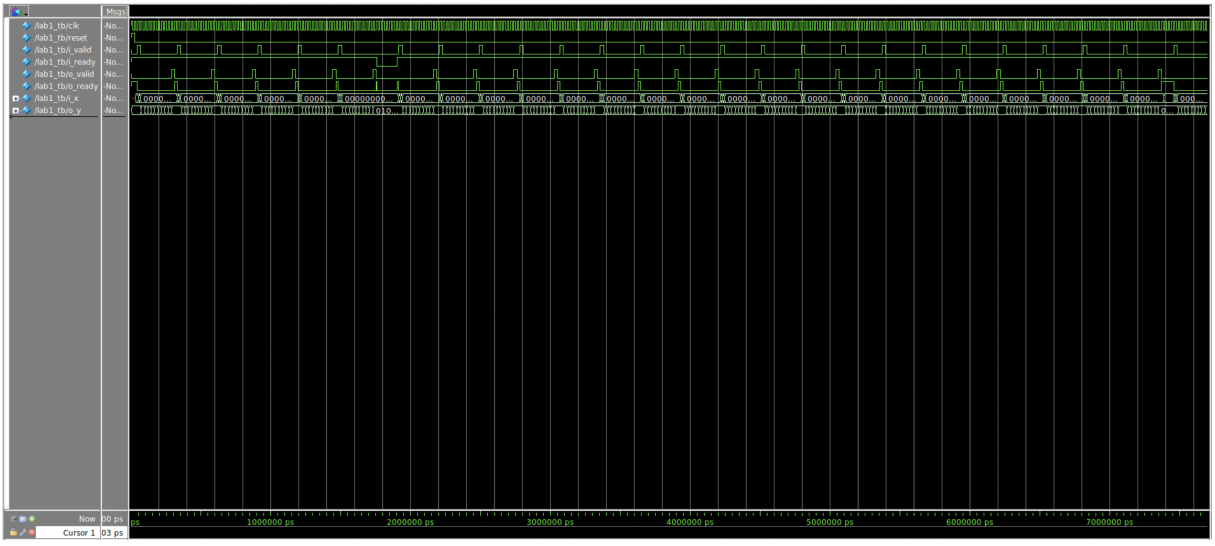

Figure 5: ModelSim waveforms part 1 of 2 for the shared hardware design as expected

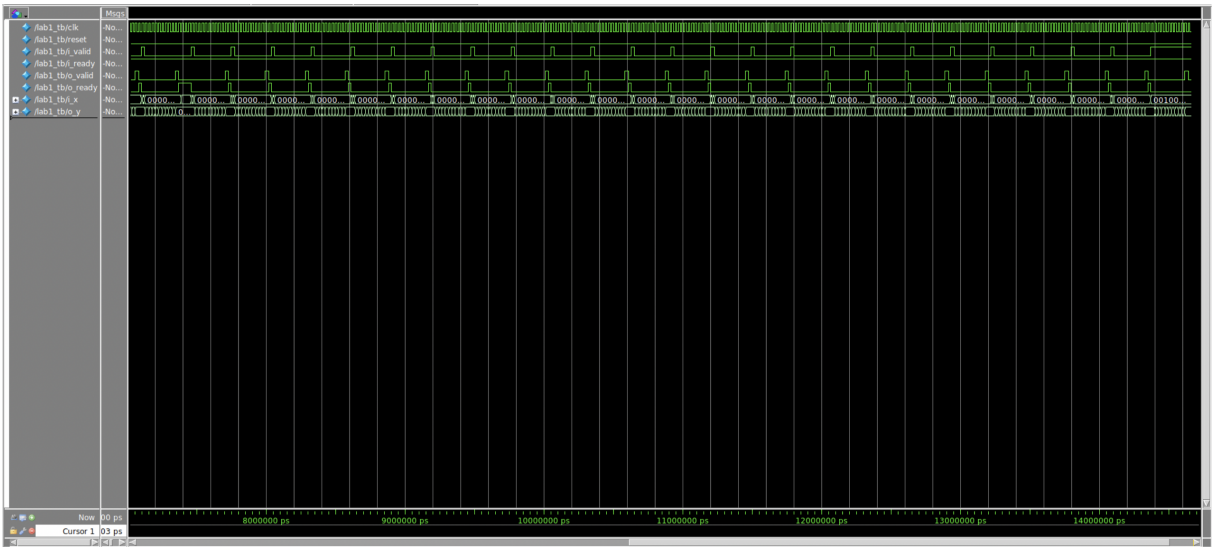

Figure 6: ModelSim waveforms part 2 of 2 for the shared hardware design as expected

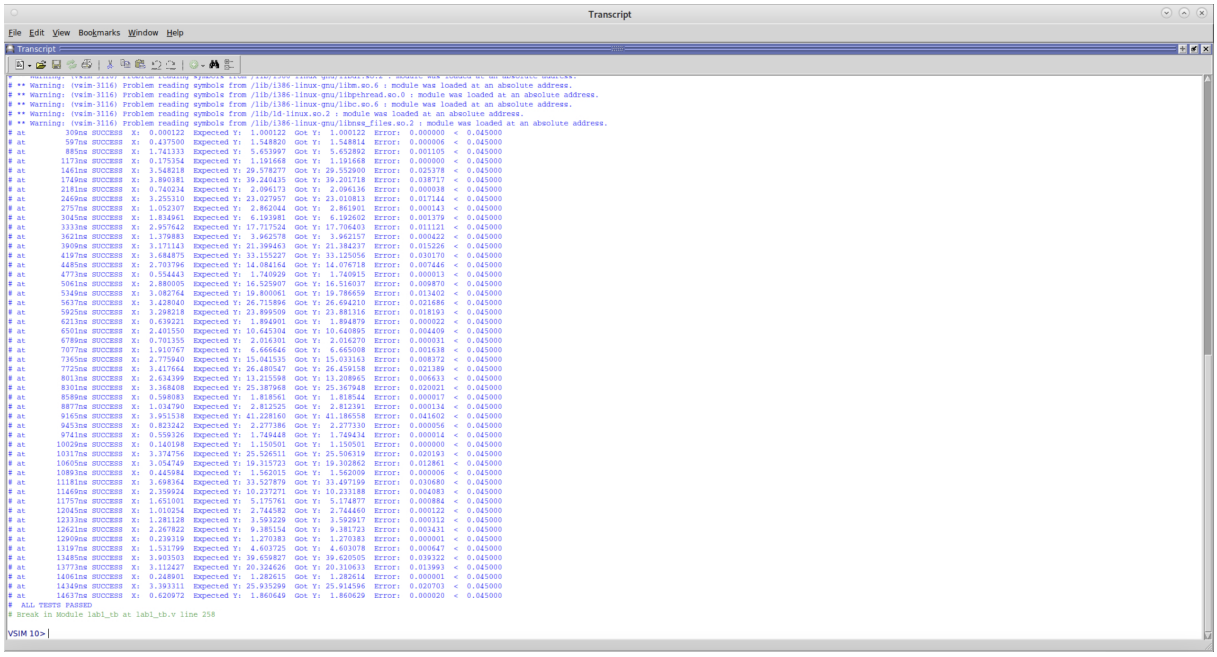

Figure 7: ModelSim simulation log for the shared hardware design passing all test cases

2.2 SharedHW

3 QoR Table

	Baseline Circuit	**Pipelined Circuit**	**Shared HW Circuit**
Resources for one circuit	21 ALMs + 9 DSPs	154 ALMs + 9 DSPs	42 ALMs + 2 DSPs
Operating frequency	44.8 MHz	425.17 MHz	179.66 MHz
Critical path	DSP mult-add + 2x DSP mult + LE-based adder + 6x DSP mult + LE-based adder	LE-based mult adder + LE-based addr adder	LE-based multiplexer + 2x DSP mult
Cycles per valid output	1	1	10
Max. # of copies/device	168	168	759
Max. Throughput for a full device (computations/s)	$7.5264 \cdot 10^9$	$71.42856 \cdot 10^9$	$13.63619 \cdot 10^9$
Dynamic power of one circuit @ 42 MHz	1.96 mW	1.77 mW	0.6 mW
Max. throughput/Watt for a full device	$21.42857 \cdot 10^9/W$	$23.72881 \cdot 10^9/W$	$7.0 \cdot 10^9/W$

Calculations

max # of copies/device: $round\_down(dsp\_resource \div dsp\_used)$

max throughput for a full device: $Fmax \div cycles\_per\_output \times max\_num\_copies$

max throughput/watt for a full device: $max\_throughput\_full\_device \div (power\_at\_42MHz \div 42MHz \times Fmax \times max\_num\_copies)$

4 Discussions

1. *What are the different sources of error (i.e. difference between exp(x) and Hardware Output in the graph you plotted for the testbench output)? What changes could you make to the circuit to reduce this error?*

   There are mainly two sources of errors, which are fixed-point precision and taylor series error. To improve fixed-point precision related error, we could use wider fixed point

representations for input number and taylor coefficients used in the calculation. To reduce the taylor series error, we could include more terms (i.e., higher order terms) from the taylor series. However, as the coefficient for the exp function gets smaller, the fixed-point precision starts to matter more.

2. *Which of the 3 hardware circuits (baseline, pipelined, and shared) achieves the highest throughput/device? Explain the reasons for the efficiency differences between them.*

   The pipelined design achieves the highest throughput/device of $71.42856 \cdot 10^9$ computations per second across the 3 hardware circuits. Throughput is determined by both the maximum operating frequency of the circuit and the number of cycles per valid output. The maximum operating frequency is the most obvious factor, the highly optimized pipelined design is able to achieve an Fmax of 425.17 MHz, by balancing the different delays on the timing paths via the injection of pipeline registers. Secondly, the baseline and pipelined circuits are all using the streaming dataflow, which means valid output can be produced at every cycle; whereas the shared hardware design uses an FSM to control the datapath, which means the input data has to wait for all FSM states to complete before it can be processed. The pipelined version is again in the leading position. Lastly, on this FPGA chip, DSP block is the bottleneck that determines the maximum number of circuit copies that can fit, as the 3 hardware circuits all have DSP utilization higher than that of the other resource types. Putting altogether, although the shared hardware design is able to have 759 copies fit on the chip, the pipelined design has a dominant lead in Fmax and cycles per valid output, therefore achieves the highest throughput/device.

3. *Look at the average toggle rates of the blocks for the 3 circuits (this information is in the PowerAnalyzer report). Explain why some circuit styles lead to higher toggle rates for the DSP blocks and combinational/registered logic cells than others. Comment on the relative efficiency of the 3 circuits in terms of computations/J, and explain why each style is more or less efficient in computations/J than the others.*

   The average toggle rates for the baseline, pipelined and shared hardware versions are 12.819, 10.933 and 10.557 millions of transitions per second respectively, and are all analyzed under the same clock frequency of 42 MHz. Shared hardware has the lowest toggle rates because it has the lowest resource utilization. Although toggle rates are measured under various stages of glitch-filtering, glitches might still be present in the design under power analysis. The unintended toggling occurs when a merged logic path has unbalanced delays on its input ends, which is more obvious in the baseline design, as there are no pipeline stages in the middle of the computation path. With the very fine-grained pipeline register insertion in the pipelined design, the computation paths are now more balanced in terms of delays, and thus results in much lower toggle rates than the baseline.

   In terms of computation-power efficiency, pipelined version gives the highest of $23.72881 \cdot 10^9 computations/J$, baseline version is slightly lower which is $21.42857 \cdot 10^9 computations/J$,

and the shared hardware version is the lowest which is $7.0 \cdot 10^9 computations/J$. This calculation can be derived directly from the **Dynamic power of one circuit @42 MHz** and **Cycles per valid output**. Although the shared hardware version has the lowest dynamic power (about 3-4x less than the other two designs), it requires 10x more cycles for a valid output, thus giving it the lowest efficiency. On the other hand, the pipelined version has lower dynamic power than the baseline version, thus making it the most efficient design.

5 Appendix

5.1 HDL - Pipelined

```
module lab1 #
(
        parameter WIDTHIN = 16,                 // Input format is Q2.14 (2 integer bits + 14 fractional bits = 16 bits)
        parameter WIDTHOUT = 32,        // Intermediate/Output format is Q7.25 (7 integer bits + 25 fractional bits = 32 bits)
        // Taylor coefficients for the first five terms in Q2.14 format
        parameter [WIDTHIN-1:0] A0 = 16'b01_00000000000000, // a0 = 1
        parameter [WIDTHIN-1:0] A1 = 16'b01_00000000000000, // a1 = 1
        parameter [WIDTHIN-1:0] A2 = 16'b00_10000000000000, // a2 = 1/2
        parameter [WIDTHIN-1:0] A3 = 16'b00_00101010101010, // a3 = 1/6
        parameter [WIDTHIN-1:0] A4 = 16'b00_00001010101010, // a4 = 1/24
        parameter [WIDTHIN-1:0] A5 = 16'b00_00000010001000  // a5 = 1/120
)
(
        input clk,
        input reset,

        input i_valid,
        input i_ready,
        output o_valid,
        output o_ready,

        input [WIDTHIN-1:0] i_x,
        output [WIDTHOUT-1:0] o_y
);
//Output value could overflow (32-bit output, and 16-bit inputs multiplied
//together repeatedly).  Don't worry about that -- assume that only the bottom
```

```
27 //32 bits are of interest, and keep them.
28 logic [WIDTHIN-1:0] x;         // Register to hold input X
29 logic [WIDTHOUT-1:0] y_Q;        // Register to hold output Y
30 logic valid_Q1;                 // Output of register x is valid
31 logic valid_Q2;                 // Output of register y is valid
32
33 // signal for enabling sequential circuit elements
34 logic enable;
35
36 // Signals for computing the y output
37 logic [WIDTHOUT-1:0] m0_out; // A5 * x
38 logic [WIDTHOUT-1:0] a0_out; // A5 * x + A4
39 logic [WIDTHOUT-1:0] m1_out; // (A5 * x + A4) * x
40 logic [WIDTHOUT-1:0] a1_out; // (A5 * x + A4) * x + A3
41 logic [WIDTHOUT-1:0] m2_out; // ((A5 * x + A4) * x + A3) * x
42 logic [WIDTHOUT-1:0] a2_out; // ((A5 * x + A4) * x + A3) * x + A2
43 logic [WIDTHOUT-1:0] m3_out; // (((A5 * x + A4) * x + A3) * x + A2) * x
44 logic [WIDTHOUT-1:0] a3_out; // (((A5 * x + A4) * x + A3) * x + A2) * x +
    ↪  A1
45 logic [WIDTHOUT-1:0] m4_out; // ((((A5 * x + A4) * x + A3) * x + A2) * x +
    ↪  A1) * x
46 logic [WIDTHOUT-1:0] a4_out; // ((((A5 * x + A4) * x + A3) * x + A2) * x +
    ↪  A1) * x + A0
47 logic [WIDTHOUT-1:0] y_D;
48
49 // Piepeline: 2 * 9 (operator-bounded) stages in total
50 localparam PIPELINE_STAGES = 18;
51 // Pipelined signals for computing the y output
52 logic [WIDTHOUT-1:0] m0_out_reg; // A5 * x
53 logic [WIDTHOUT-1:0] m0_out_reg_reg; // A5 * x
54 logic [WIDTHOUT-1:0] a0_out_reg; // A5 * x + A4
55 logic [WIDTHOUT-1:0] a0_out_reg_reg; // A5 * x + A4
56 logic [WIDTHOUT-1:0] m1_out_reg; // (A5 * x + A4) * x
57 logic [WIDTHOUT-1:0] m1_out_reg_reg; // (A5 * x + A4) * x
58 logic [WIDTHOUT-1:0] a1_out_reg; // (A5 * x + A4) * x + A3
59 logic [WIDTHOUT-1:0] a1_out_reg_reg; // (A5 * x + A4) * x + A3
60 logic [WIDTHOUT-1:0] m2_out_reg; // ((A5 * x + A4) * x + A3) * x
61 logic [WIDTHOUT-1:0] m2_out_reg_reg; // ((A5 * x + A4) * x + A3) * x
62 logic [WIDTHOUT-1:0] a2_out_reg; // ((A5 * x + A4) * x + A3) * x + A2
63 logic [WIDTHOUT-1:0] a2_out_reg_reg; // ((A5 * x + A4) * x + A3) * x + A2
64 logic [WIDTHOUT-1:0] m3_out_reg; // (((A5 * x + A4) * x + A3) * x + A2) * x
```

```
65 logic [WIDTHOUT-1:0] m3_out_reg_reg; // (((A5 * x + A4) * x + A3) * x + A2)
   ↪  * x
66 logic [WIDTHOUT-1:0] a3_out_reg; // (((A5 * x + A4) * x + A3) * x + A2) * x
   ↪  + A1
67 logic [WIDTHOUT-1:0] a3_out_reg_reg; // (((A5 * x + A4) * x + A3) * x + A2)
   ↪  * x + A1
68 logic [WIDTHOUT-1:0] m4_out_reg; // ((((A5 * x + A4) * x + A3) * x + A2) *
   ↪  x + A1) * x
69 logic [WIDTHOUT-1:0] m4_out_reg_reg; // ((((A5 * x + A4) * x + A3) * x +
   ↪  A2) * x + A1) * x
70 // Pipelined x signal
71 logic [PIPELINE_STAGES-1:0] [WIDTHIN-1:0] x_regs;
72 // Pipelined valid_Q1 signal
73 logic [PIPELINE_STAGES-1:0] valid_Q1_regs;
74
75 // compute y value
76 mult16x16 Mult0 (.i_dataa(A5),                                  .i_datab(x),
   ↪                          .o_res(m0_out));
77
   ↪  // ^^stage 0 (m0_out_reg)
78 addr32p16 Addr0 (.i_dataa(m0_out_reg_reg),          .i_datab(A4),
   ↪                          .o_res(a0_out)); // ^^stage 1 (m0_out_reg_reg)
79
   ↪  // ^^stage 2 (a0_out_reg)
80 mult32x16 Mult1 (.i_dataa(a0_out_reg_reg),          .i_datab(x_regs[3]),
   ↪          .o_res(m1_out)); // ^^stage 3 (a0_out_reg_reg)
81
   ↪  // ^^stage 4 (m1_out_reg)
82 addr32p16 Addr1 (.i_dataa(m1_out_reg_reg),          .i_datab(A3),
   ↪                          .o_res(a1_out)); // ^^stage 5 (m1_out_reg_reg)
83
   ↪  // ^^stage 6 (a1_out_reg)
84 mult32x16 Mult2 (.i_dataa(a1_out_reg_reg),          .i_datab(x_regs[7]),
   ↪          .o_res(m2_out)); // ^^stage 7 (a1_out_reg_reg)
85
   ↪  // ^^stage 8 (m2_out_reg)
86 addr32p16 Addr2 (.i_dataa(m2_out_reg_reg),          .i_datab(A2),
   ↪                          .o_res(a2_out)); // ^^stage 9 (m2_out_reg_reg)
87
   ↪  // ^^stage 10 (a2_out_reg)
88 mult32x16 Mult3 (.i_dataa(a2_out_reg_reg),          .i_datab(x_regs[11]),
   ↪          .o_res(m3_out)); // ^^stage 11 (a2_out_reg_reg)
```

```
89
    ↪  // ^^stage 12 (m3_out_reg)
90  addr32p16 Addr3 (.i_dataa(m3_out_reg_reg),          .i_datab(A1),
    ↪                           .o_res(a3_out)); // ^^stage 13 (m3_out_reg_reg)
91
    ↪  // ^^stage 14 (a3_out_reg)
92  mult32x16 Mult4 (.i_dataa(a3_out_reg_reg),          .i_datab(x_regs[15]),
    ↪           .o_res(m4_out)); // ^^stage 15 (a3_out_reg_reg)
93
    ↪  // ^^stage 16 (m4_out_reg)
94  addr32p16 Addr4 (.i_dataa(m4_out_reg_reg),          .i_datab(A0),
    ↪                           .o_res(a4_out)); // ^^stage 17 (m4_out_reg_reg)
95
96  assign y_D = a4_out;
97
98  // Combinational logic
99  always_comb begin
100         // signal for enable
101         enable = i_ready;
102 end
103
104 // Infer the registers
105 always_ff @(posedge clk or posedge reset) begin
106         if (reset) begin
107                 valid_Q1 <= 1'b0;
108                 valid_Q2 <= 1'b0;
109
110                 x <= 0;
111                 y_Q <= 0;
112
113                 // pipeline registers
114                 x_regs <= 0;
115                 valid_Q1_regs <= 0;
116                 m0_out_reg <= 0;
117                 m1_out_reg <= 0;
118                 m2_out_reg <= 0;
119                 m3_out_reg <= 0;
120                 m4_out_reg <= 0;
121                 m0_out_reg_reg <= 0;
122                 m1_out_reg_reg <= 0;
123                 m2_out_reg_reg <= 0;
124                 m3_out_reg_reg <= 0;
```

```
125                 m4_out_reg_reg <= 0;
126                 a0_out_reg <= 0;
127                 a1_out_reg <= 0;
128                 a2_out_reg <= 0;
129                 a3_out_reg <= 0;
130                 a0_out_reg_reg <= 0;
131                 a1_out_reg_reg <= 0;
132                 a2_out_reg_reg <= 0;
133                 a3_out_reg_reg <= 0;
134         end else if (enable) begin
135                 // propagate the valid value
136                 valid_Q1 <= i_valid;
137                 // pipeline valid_Q1
138                 {valid_Q2, valid_Q1_regs} <= {valid_Q1_regs, valid_Q1};
139
140                 // read in new x value
141                 x <= i_x;
142                 // pipeline x
143                 x_regs <= {x_regs[(PIPELINE_STAGES-1)-1:0], x};
144
145                 // output computed y value
146                 y_Q <= y_D;
147
148                 // pipeline calculation intermediate results
149                 m0_out_reg <= m0_out;
150                 m1_out_reg <= m1_out;
151                 m2_out_reg <= m2_out;
152                 m3_out_reg <= m3_out;
153                 m4_out_reg <= m4_out;
154                 m0_out_reg_reg <= m0_out_reg;
155                 m1_out_reg_reg <= m1_out_reg;
156                 m2_out_reg_reg <= m2_out_reg;
157                 m3_out_reg_reg <= m3_out_reg;
158                 m4_out_reg_reg <= m4_out_reg;
159                 a0_out_reg <= a0_out;
160                 a1_out_reg <= a1_out;
161                 a2_out_reg <= a2_out;
162                 a3_out_reg <= a3_out;
163                 a0_out_reg_reg <= a0_out_reg;
164                 a1_out_reg_reg <= a1_out_reg;
165                 a2_out_reg_reg <= a2_out_reg;
166                 a3_out_reg_reg <= a3_out_reg;
```

```
167         end
168 end
169
170 // assign outputs
171 assign o_y = y_Q;
172 // ready for inputs as long as receiver is ready for outputs */
173 assign o_ready = i_ready;
174 // the output is valid as long as the corresponding input was valid and
175 //         the receiver is ready. If the receiver isn't ready, the computed
    ↪  output
176 //         will still remain on the register outputs and the circuit will
    ↪  resume
177 //  normal operation when the receiver is ready again (i_ready is high)
178 assign o_valid = valid_Q2 & i_ready;
179
180 endmodule
181
182 /*****************************************/
183
184 // Multiplier module for the first 16x16 multiplication
185 module mult16x16 (
186         input  [15:0] i_dataa,
187         input  [15:0] i_datab,
188         output [31:0] o_res
189 );
190
191 logic [31:0] result;
192
193 always_comb begin
194         result = i_dataa * i_datab;
195 end
196
197 // The result of Q2.14 x Q2.14 is in the Q4.28 format. Therefore we need
    ↪  to change it
198 // to the Q7.25 format specified in the assignment by shifting right and
    ↪  padding with zeros.
199 assign o_res = {3'b000, result[31:3]};
200
201 endmodule
202
203 /*****************************************/
204
```

```
205 // Multiplier module for all the remaining 32x16 multiplications
206 module mult32x16 (
207         input  [31:0] i_dataa,
208         input  [15:0] i_datab,
209         output [31:0] o_res
210 );
211
212 logic [47:0] result;
213
214 always_comb begin
215         result = i_dataa * i_datab;
216 end
217
218 // The result of Q7.25 x Q2.14 is in the Q9.39 format. Therefore we need
    ↪  to change it
219 // to the Q7.25 format specified in the assignment by selecting the
    ↪  appropriate bits
220 // (i.e. dropping the most-significant 2 bits and least-significant 14
    ↪  bits).
221 assign o_res = result[45:14];
222
223 endmodule
224
225 /*********************************************/
226
227 // Adder module for all the 32b+16b addition operations
228 module addr32p16 (
229         input [31:0] i_dataa,
230         input [15:0] i_datab,
231         output [31:0] o_res
232 );
233
234 // The 16-bit Q2.14 input needs to be aligned with the 32-bit Q7.25 input
    ↪  by zero padding
235 assign o_res = i_dataa + {5'b00000, i_datab, 11'b00000000000};
236
237 endmodule
238
239 /*********************************************/
```

5.2 HDL - SharedHW

```
module lab1 #
(
        parameter WIDTHIN = 16,                 // Input format is Q2.14 (2 integer bits + 14 fractional bits = 16 bits)
        parameter WIDTHOUT = 32,        // Intermediate/Output format is Q7.25 (7 integer bits + 25 fractional bits = 32 bits)
        // Taylor coefficients for the first five terms in Q2.14 format
        parameter [WIDTHIN-1:0] A0 = 16'b01_00000000000000, // a0 = 1
        parameter [WIDTHIN-1:0] A1 = 16'b01_00000000000000, // a1 = 1
        parameter [WIDTHIN-1:0] A2 = 16'b00_10000000000000, // a2 = 1/2
        parameter [WIDTHIN-1:0] A3 = 16'b00_00101010101010, // a3 = 1/6
        parameter [WIDTHIN-1:0] A4 = 16'b00_00001010101010, // a4 = 1/24
        parameter [WIDTHIN-1:0] A5 = 16'b00_00000010001000  // a5 = 1/120
)
(
        input clk,
        input reset,

        input i_valid,
        input i_ready,
        output o_valid,
        output o_ready,

        input [WIDTHIN-1:0] i_x,
        output [WIDTHOUT-1:0] o_y
);

logic enable;
assign enable = i_ready;

/*****************************************/
// Datapath
// Parameters
localparam [WIDTHOUT-1:0] A5_32b = 32'b0000000_00000010001000000000000000; // a5 = 1/120

// Register
logic [WIDTHOUT-1:0] dp_tmp; // Register to hold intermediate result
logic [WIDTHIN-1:0] dp_x; // Register to hold input X

```

```
38 // Rigster control, (FSM output)
39 logic dp_tmp_enable;
40 logic dp_x_enable;
41
42 // Multiplexer control, (FSM output)
43 enum logic {FROM_A5_32B, FROM_DP_TMP} dp_mult_dataa_select;
44 enum logic [2:0] {FROM_A4, FROM_A3, FROM_A2, FROM_A1, FROM_A0}
    ↪  dp_addr_datab_select;
45 enum logic {FROM_MULT, FROM_ADDR} dp_tmp_d_select;
46
47 // Multiplexer output
48 logic [WIDTHOUT-1:0] dp_mult_dataa;
49 logic [WIDTHIN-1:0] dp_addr_datab;
50 logic [WIDTHOUT-1:0] dp_tmp_d;
51
52 // Mult and Addr output
53 logic [WIDTHOUT-1:0] dp_mult_out;
54 logic [WIDTHOUT-1:0] dp_addr_out;
55
56 // Mult and Addr
57 mult32x16 Mult0 (.i_dataa(dp_mult_dataa),         .i_datab(dp_x),
    ↪          .o_res(dp_mult_out));
58 addr32p16 Addr0 (.i_dataa(dp_tmp),         .i_datab(dp_addr_datab),
    ↪          .o_res(dp_addr_out));
59
60 // Multiplexer
61 always_comb begin
62         case (dp_mult_dataa_select)
63                 FROM_DP_TMP: dp_mult_dataa = dp_tmp;
64                 FROM_A5_32B: dp_mult_dataa = A5_32b;
65                 default: dp_mult_dataa = 0;
66         endcase
67
68         case(dp_addr_datab_select)
69                 FROM_A4: dp_addr_datab = A4;
70                 FROM_A3: dp_addr_datab = A3;
71                 FROM_A2: dp_addr_datab = A2;
72                 FROM_A1: dp_addr_datab = A1;
73                 FROM_A0: dp_addr_datab = A0;
74                 default: dp_addr_datab = 0;
75         endcase
76
```

```
77          case(dp_tmp_d_select)
78                  FROM_MULT: dp_tmp_d = dp_mult_out;
79                  FROM_ADDR: dp_tmp_d = dp_addr_out;
80                  default: dp_tmp_d = 0;
81          endcase
82  end
83
84  // Register
85  always_ff @(posedge clk or posedge reset) begin
86          if (reset) begin
87                  dp_tmp <= 0;
88                  dp_x <= 0;
89          end else if (enable) begin
90                  if (dp_tmp_enable) begin
91                          dp_tmp <= dp_tmp_d;
92                  end
93
94                  if (dp_x_enable) begin
95                          dp_x <= i_x;
96                  end
97          end
98  end
99
100 assign o_y = dp_tmp;
101
102 /*****************************************/
103 // FSM
104
105 // FSM output
106 logic output_valid;
107 logic compute_busy;
108
109 // State
110 typedef enum logic [3:0] {RECEIVE_INPUT, COMPUTE_0, COMPUTE_1, COMPUTE_2,
    ↪  COMPUTE_3, COMPUTE_4, COMPUTE_5, COMPUTE_6, COMPUTE_7, COMPUTE_8,
    ↪  COMPUTE_9, SEND_OUTPUT} fsm_state_type;
111 fsm_state_type fsm_current_state; // Register
112 fsm_state_type fsm_next_state;
113
114 // State register
115 always_ff @(posedge clk or posedge reset) begin
116         if (reset) begin
```

```
117                 fsm_current_state <= RECEIVE_INPUT;
118         end else if (enable) begin
119                 fsm_current_state <= fsm_next_state;
120         end
121 end
122
123 // State transition and control signal
124 always_comb begin
125         case(fsm_current_state)
126         RECEIVE_INPUT: begin
127                 if (i_valid) begin
128                         dp_tmp_enable = 0;
129                         dp_x_enable = 1;
130                         dp_mult_dataa_select = FROM_A5_32B;
131                         dp_addr_datab_select = FROM_A4;
132                         dp_tmp_d_select = FROM_MULT;
133                         output_valid = 0;
134                         compute_busy = 1;
135                         fsm_next_state = COMPUTE_0;
136                 end else begin
137                         dp_tmp_enable = 0;
138                         dp_x_enable = 0;
139                         dp_mult_dataa_select = FROM_A5_32B;
140                         dp_addr_datab_select = FROM_A4;
141                         dp_tmp_d_select = FROM_MULT;
142                         output_valid = 0;
143                         compute_busy = 0;
144                         fsm_next_state = RECEIVE_INPUT;
145                 end
146         end
147         COMPUTE_0: begin // tmp = A5_32b * x
148                 dp_tmp_enable = 1;
149                 dp_x_enable = 0;
150                 dp_mult_dataa_select = FROM_A5_32B;
151                 dp_addr_datab_select = FROM_A4;
152                 dp_tmp_d_select = FROM_MULT;
153                 output_valid = 0;
154                 compute_busy = 1;
155                 fsm_next_state = COMPUTE_1;
156         end
157         COMPUTE_1: begin // tmp = tmp + A4
158                 dp_tmp_enable = 1;
```

```
159                 dp_x_enable = 0;
160                 dp_mult_dataa_select = FROM_DP_TMP;
161                 dp_addr_datab_select = FROM_A4;
162                 dp_tmp_d_select = FROM_ADDR;
163                 output_valid = 0;
164                 compute_busy = 1;
165                 fsm_next_state = COMPUTE_2;
166         end
167         COMPUTE_2: begin // tmp = tmp * x
168                 dp_tmp_enable = 1;
169                 dp_x_enable = 0;
170                 dp_mult_dataa_select = FROM_DP_TMP;
171                 dp_addr_datab_select = FROM_A3;
172                 dp_tmp_d_select = FROM_MULT;
173                 output_valid = 0;
174                 compute_busy = 1;
175                 fsm_next_state = COMPUTE_3;
176         end
177         COMPUTE_3: begin // tmp = tmp + A3
178                 dp_tmp_enable = 1;
179                 dp_x_enable = 0;
180                 dp_mult_dataa_select = FROM_DP_TMP;
181                 dp_addr_datab_select = FROM_A3;
182                 dp_tmp_d_select = FROM_ADDR;
183                 output_valid = 0;
184                 compute_busy = 1;
185                 fsm_next_state = COMPUTE_4;
186         end
187         COMPUTE_4: begin // tmp = tmp * x
188                 dp_tmp_enable = 1;
189                 dp_x_enable = 0;
190                 dp_mult_dataa_select = FROM_DP_TMP;
191                 dp_addr_datab_select = FROM_A2;
192                 dp_tmp_d_select = FROM_MULT;
193                 output_valid = 0;
194                 compute_busy = 1;
195                 fsm_next_state = COMPUTE_5;
196         end
197         COMPUTE_5: begin // tmp = tmp + A2
198                 dp_tmp_enable = 1;
199                 dp_x_enable = 0;
200                 dp_mult_dataa_select = FROM_DP_TMP;
```

```
201                 dp_addr_datab_select = FROM_A2;
202                 dp_tmp_d_select = FROM_ADDR;
203                 output_valid = 0;
204                 compute_busy = 1;
205                 fsm_next_state = COMPUTE_6;
206         end
207         COMPUTE_6: begin // tmp = tmp * x
208                 dp_tmp_enable = 1;
209                 dp_x_enable = 0;
210                 dp_mult_dataa_select = FROM_DP_TMP;
211                 dp_addr_datab_select = FROM_A1;
212                 dp_tmp_d_select = FROM_MULT;
213                 output_valid = 0;
214                 compute_busy = 1;
215                 fsm_next_state = COMPUTE_7;
216         end
217         COMPUTE_7: begin // tmp = tmp + A1
218                 dp_tmp_enable = 1;
219                 dp_x_enable = 0;
220                 dp_mult_dataa_select = FROM_DP_TMP;
221                 dp_addr_datab_select = FROM_A1;
222                 dp_tmp_d_select = FROM_ADDR;
223                 output_valid = 0;
224                 compute_busy = 1;
225                 fsm_next_state = COMPUTE_8;
226         end
227         COMPUTE_8: begin // tmp = tmp * x
228                 dp_tmp_enable = 1;
229                 dp_x_enable = 0;
230                 dp_mult_dataa_select = FROM_DP_TMP;
231                 dp_addr_datab_select = FROM_A0;
232                 dp_tmp_d_select = FROM_MULT;
233                 output_valid = 0;
234                 compute_busy = 1;
235                 fsm_next_state = COMPUTE_9;
236         end
237         COMPUTE_9: begin // tmp = tmp + A0
238                 dp_tmp_enable = 1;
239                 dp_x_enable = 0;
240                 dp_mult_dataa_select = FROM_DP_TMP;
241                 dp_addr_datab_select = FROM_A0;
242                 dp_tmp_d_select = FROM_ADDR;
```

```
243                 output_valid = 0;
244                 compute_busy = 1;
245                 fsm_next_state = SEND_OUTPUT;
246         end
247         SEND_OUTPUT: begin
248                 dp_tmp_enable = 0;
249                 dp_x_enable = 0;
250                 dp_mult_dataa_select = FROM_DP_TMP;
251                 dp_addr_datab_select = FROM_A0;
252                 dp_tmp_d_select = FROM_ADDR;
253                 output_valid = 1;
254                 compute_busy = 1;
255                 fsm_next_state = RECEIVE_INPUT;
256         end
257         default: begin
258                 dp_tmp_enable = 0;
259                 dp_x_enable = 0;
260                 dp_mult_dataa_select = FROM_A5_32B;
261                 dp_addr_datab_select = FROM_A4;
262                 dp_tmp_d_select = FROM_MULT;
263                 output_valid = 0;
264                 compute_busy = 0;
265                 fsm_next_state = RECEIVE_INPUT;
266         end
267         endcase
268 end
269
270 assign o_valid = output_valid & i_ready;
271 assign o_ready = ~compute_busy & i_ready;
272 endmodule
273
274 /*********************************************/
275
276 // Multiplier module for all the remaining 32x16 multiplications
277 module mult32x16 (
278         input  [31:0] i_dataa,
279         input  [15:0] i_datab,
280         output [31:0] o_res
281 );
282
283 logic [47:0] result;
284
```

```
285 always_comb begin
286         result = i_dataa * i_datab;
287 end
288
289 // The result of Q7.25 x Q2.14 is in the Q9.39 format. Therefore we need
    ↪  to change it
290 // to the Q7.25 format specified in the assignment by selecting the
    ↪  appropriate bits
291 // (i.e. dropping the most-significant 2 bits and least-significant 14
    ↪  bits).
292 assign o_res = result[45:14];
293
294 endmodule
295
296 /*****************************************/
297
298 // Adder module for all the 32b+16b addition operations
299 module addr32p16 (
300         input [31:0] i_dataa,
301         input [15:0] i_datab,
302         output [31:0] o_res
303 );
304
305 // The 16-bit Q2.14 input needs to be aligned with the 32-bit Q7.25 input
    ↪  by zero padding
306 assign o_res = i_dataa + {5'b00000, i_datab, 11'b00000000000};
307
308 endmodule
309
310 /*****************************************/
```