1. Block diagram and description

1.1 Pipelined design

In figure 1, registers are added between multipliers and adders(i_x to o_y path) to create a 11-stage pipeline circuit. To maintain the correctness of the circuit, two registers need to be inserted in the multiplier to multiplier path, in total nine registers need to be inserted in the i_valid to o_valid path. The signals before and after pipeline registers are labeled properly in blue and match the signal naming in pipelined HDL(see appendix I).

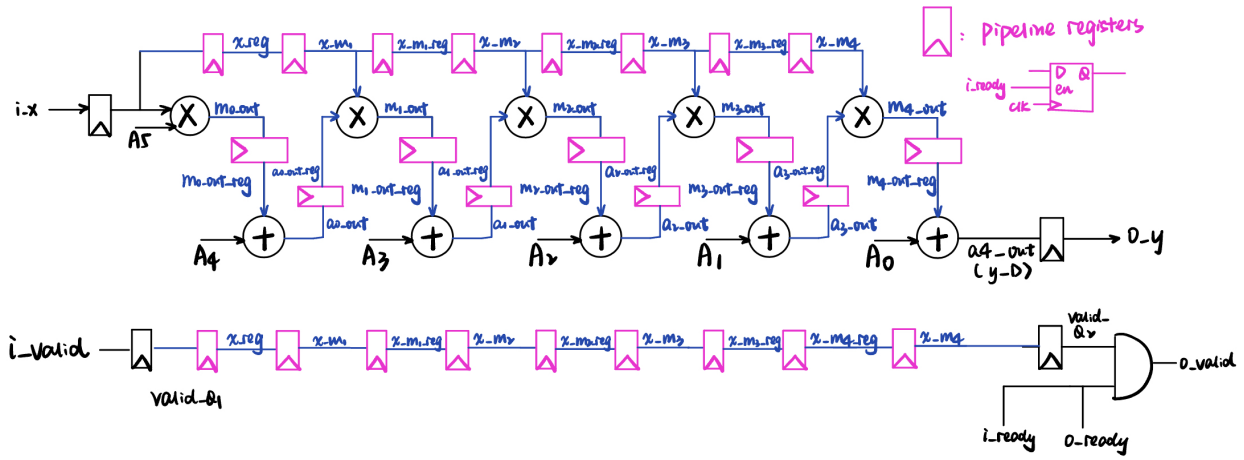

Figure 1: Block diagram of pipelined design.

1.2 SharedHW design

Figure 2 shows the data flow diagram for the shared circuit. i_x will be registered and feed as input of multiplier m_in_b once start_x signal is enabled(This is controlled by i_valid signal). The other multiplier input will use multsel_in to select between A5 and y_store(previous output from the adder). The output of multiplier a_in_a will be directly fed into adder, and the other input a_in_b will use addsel_in to select value from A0 to A4. Occupied signal is also introduced to show current multiplier and/or adder is busy at computing, which indicates o_ready value. Those control signals (register enable: startx and wb_y; sel from mux: multsel_in and addsel_in; occupied signal) are generated from a control flow of FSM. (Shown in figure 3)
The control signals and registers' output are labeled properly in blue/purple and match the signal naming in sharedHW HDL(see appendix II).

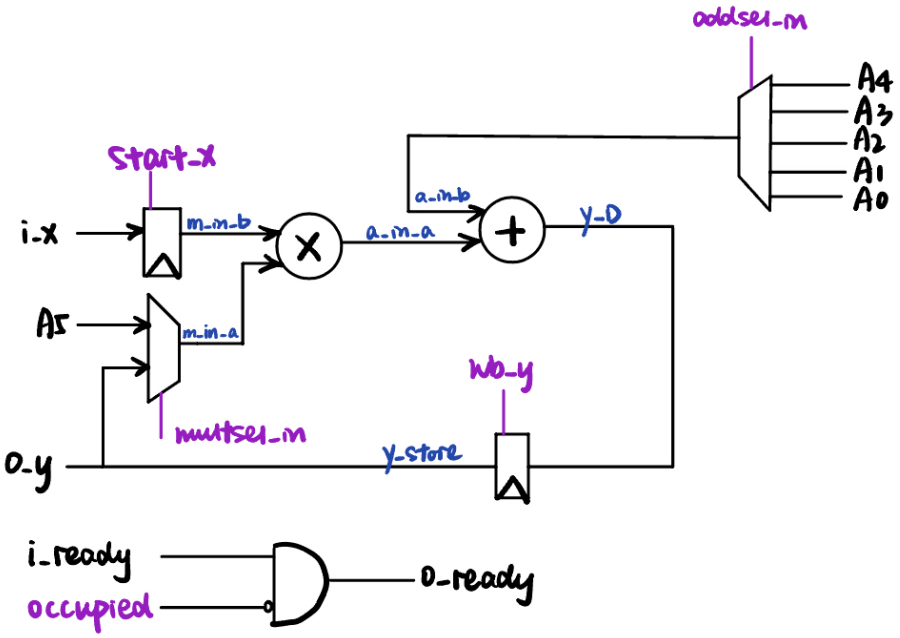

Figure 2: Block diagram of sharedHW design.

2. Simulation and testbench output

2.1 Pipelined design

2.1.1 Simulation waveform

(* To better show the signal's transition, all signal's value are presented in decimal here)

Waveform is shown in figure 4, pink arrows denote values are passed down through pipeline registers. Blue arrows denote computation for multiplier/adder changes in signal values. Finally, a valid output value is present with the valid signal.

2.1.2 Testbench output

All tests are passed as shown in figure 5.

2.2 SharedHW design

2.2.1 Simulation waveform

(* To better show the signal's transition, all signal's value are presented in decimal here)

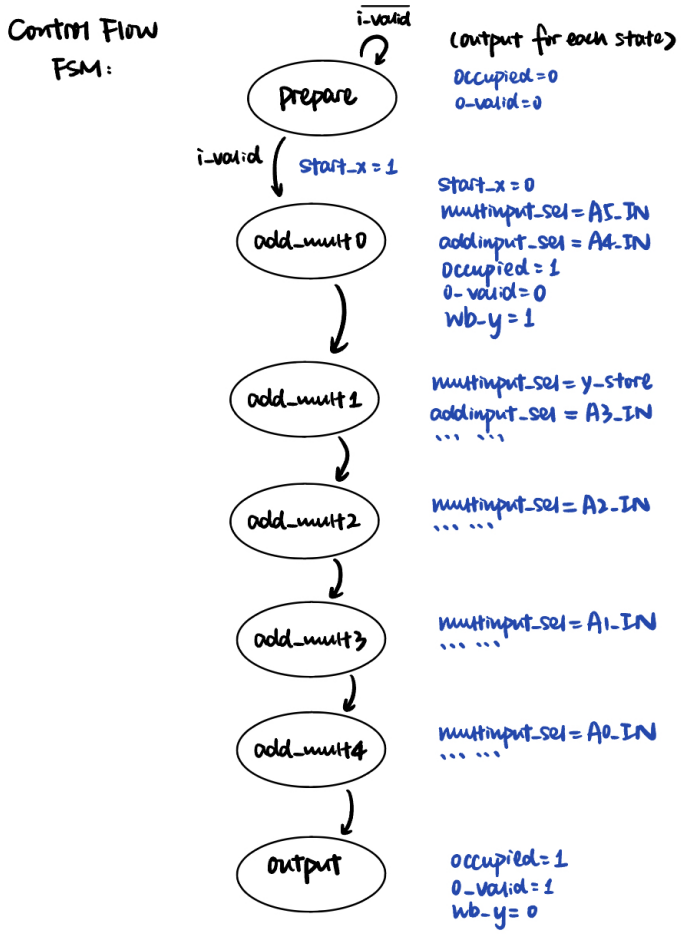

Figure 3: FSM of sharedHW design.

As shown in figure 6, i_valid prompts the state to 001 (add_mult0) state and starts computation. Pink arrows denote inputs are selected properly according to the state: i_x transits to m_in_b. (the second value of i_x is stalled). m_in_a selects A5 for the first computation, and then use the output from adder(y_store) as input. Yellow annotation denotes a_in_b is selecting A4 to A0. Finally, state 110 (output) raises the o_valid signal with an output directly from y_store, marked in blue in figure 6.

2.2.2 Testbench output

All tests are passed as shown in figure 7.

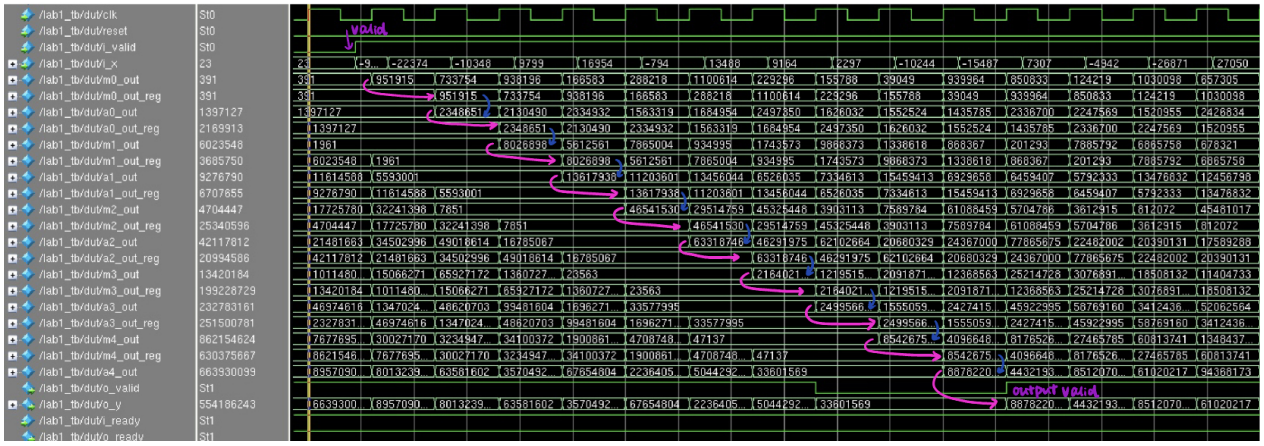

Figure 4: waveform of pipelined design.

```
VSIM 15> run -all
# ** Warning: (vsim-3116) Problem reading symbols from /lib/i386-linux-gnu/librt.so.1 : module was loaded at an absolute address.
# ** Warning: (vsim-3116) Problem reading symbols from /lib/i386-linux-gnu/libdl.so.2 : module was loaded at an absolute address.
# ** Warning: (vsim-3116) Problem reading symbols from /lib/i386-linux-gnu/libm.so.6 : module was loaded at an absolute address.
# ** Warning: (vsim-3116) Problem reading symbols from /lib/i386-linux-gnu/libpthread.so.0 : module was loaded at an absolute address.
# ** Warning: (vsim-3116) Problem reading symbols from /lib/i386-linux-gnu/libc.so.6 : module was loaded at an absolute address.
# ** Warning: (vsim-3116) Problem reading symbols from /lib/ld-linux.so.2 : module was loaded at an absolute address.
# ** Warning: (vsim-3116) Problem reading symbols from /lib/i386-linux-gnu/libnss_files.so.2 : module was loaded at an absolute address.
# at         309ns SUCCESS  X:  0.000122  Expected Y:  1.000122  Got Y:  1.000122  Error:  0.000000  <  0.045000
# at         333ns SUCCESS  X:  0.437500  Expected Y:  1.548820  Got Y:  1.548814  Error:  0.000006  <  0.045000
# at         357ns SUCCESS  X:  1.741333  Expected Y:  5.653997  Got Y:  5.652892  Error:  0.001105  <  0.045000
# at         381ns SUCCESS  X:  0.175354  Expected Y:  1.191668  Got Y:  1.191668  Error:  0.000000  <  0.045000
# at         405ns SUCCESS  X:  3.548218  Expected Y: 29.578277  Got Y: 29.552900  Error:  0.025378  <  0.045000
# at         429ns SUCCESS  X:  3.890381  Expected Y: 39.240435  Got Y: 39.201718  Error:  0.038717  <  0.045000
# at         597ns SUCCESS  X:  0.740234  Expected Y:  2.096173  Got Y:  2.096136  Error:  0.000038  <  0.045000
# at         621ns SUCCESS  X:  3.255310  Expected Y: 23.027957  Got Y: 23.010813  Error:  0.017144  <  0.045000
# at         645ns SUCCESS  X:  1.052307  Expected Y:  2.862044  Got Y:  2.861901  Error:  0.000143  <  0.045000
# at         669ns SUCCESS  X:  1.834961  Expected Y:  6.193981  Got Y:  6.192602  Error:  0.001379  <  0.045000
# at         693ns SUCCESS  X:  2.957642  Expected Y: 17.717524  Got Y: 17.706403  Error:  0.011121  <  0.045000
# at         717ns SUCCESS  X:  1.379883  Expected Y:  3.962578  Got Y:  3.962157  Error:  0.000422  <  0.045000
# at         741ns SUCCESS  X:  3.171143  Expected Y: 21.399463  Got Y: 21.384237  Error:  0.015226  <  0.045000
# at         765ns SUCCESS  X:  3.684875  Expected Y: 33.155227  Got Y: 33.125056  Error:  0.030170  <  0.045000
# at         789ns SUCCESS  X:  2.703796  Expected Y: 14.084164  Got Y: 14.076718  Error:  0.007446  <  0.045000
# at         813ns SUCCESS  X:  0.554443  Expected Y:  1.740929  Got Y:  1.740915  Error:  0.000013  <  0.045000
# at         837ns SUCCESS  X:  2.880005  Expected Y: 16.525907  Got Y: 16.516037  Error:  0.009870  <  0.045000
# at         861ns SUCCESS  X:  3.082764  Expected Y: 19.800061  Got Y: 19.786659  Error:  0.013402  <  0.045000
# at         885ns SUCCESS  X:  3.428040  Expected Y: 26.715896  Got Y: 26.694210  Error:  0.021686  <  0.045000
# at         909ns SUCCESS  X:  3.298218  Expected Y: 23.899509  Got Y: 23.881316  Error:  0.018193  <  0.045000
# at         933ns SUCCESS  X:  0.639221  Expected Y:  1.894901  Got Y:  1.894879  Error:  0.000022  <  0.045000
# at         957ns SUCCESS  X:  2.401550  Expected Y: 10.645304  Got Y: 10.640895  Error:  0.004409  <  0.045000
# at         981ns SUCCESS  X:  0.701355  Expected Y:  2.016301  Got Y:  2.016270  Error:  0.000031  <  0.045000
# at        1005ns SUCCESS  X:  1.910767  Expected Y:  6.666646  Got Y:  6.665008  Error:  0.001638  <  0.045000
# at        1029ns SUCCESS  X:  2.775940  Expected Y: 15.041535  Got Y: 15.033163  Error:  0.008372  <  0.045000
# at        1125ns SUCCESS  X:  3.417664  Expected Y: 26.480547  Got Y: 26.459158  Error:  0.021389  <  0.045000
# at        1149ns SUCCESS  X:  2.634399  Expected Y: 13.215598  Got Y: 13.208965  Error:  0.006633  <  0.045000
# at        1173ns SUCCESS  X:  3.368408  Expected Y: 25.387968  Got Y: 25.367948  Error:  0.020021  <  0.045000
# at        1197ns SUCCESS  X:  0.598083  Expected Y:  1.818561  Got Y:  1.818544  Error:  0.000017  <  0.045000
# at        1221ns SUCCESS  X:  1.034790  Expected Y:  2.812525  Got Y:  2.812391  Error:  0.000134  <  0.045000
# at        1245ns SUCCESS  X:  3.951538  Expected Y: 41.228160  Got Y: 41.186558  Error:  0.041602  <  0.045000
# at        1269ns SUCCESS  X:  0.823242  Expected Y:  2.277386  Got Y:  2.277330  Error:  0.000056  <  0.045000
# at        1293ns SUCCESS  X:  0.559326  Expected Y:  1.749448  Got Y:  1.749434  Error:  0.000014  <  0.045000
# at        1317ns SUCCESS  X:  0.140198  Expected Y:  1.150501  Got Y:  1.150501  Error:  0.000000  <  0.045000
# at        1341ns SUCCESS  X:  3.374756  Expected Y: 25.526511  Got Y: 25.506319  Error:  0.020193  <  0.045000
# at        1365ns SUCCESS  X:  3.054749  Expected Y: 19.315723  Got Y: 19.302862  Error:  0.012861  <  0.045000
# at        1389ns SUCCESS  X:  0.445984  Expected Y:  1.562015  Got Y:  1.562009  Error:  0.000006  <  0.045000
# at        1413ns SUCCESS  X:  3.698364  Expected Y: 33.527879  Got Y: 33.497199  Error:  0.030680  <  0.045000
# at        1437ns SUCCESS  X:  2.359924  Expected Y: 10.237271  Got Y: 10.233188  Error:  0.004083  <  0.045000
# at        1461ns SUCCESS  X:  1.651001  Expected Y:  5.175761  Got Y:  5.174877  Error:  0.000884  <  0.045000
# at        1485ns SUCCESS  X:  1.010254  Expected Y:  2.744582  Got Y:  2.744460  Error:  0.000122  <  0.045000
# at        1509ns SUCCESS  X:  1.281128  Expected Y:  3.593229  Got Y:  3.592917  Error:  0.000312  <  0.045000
# at        1533ns SUCCESS  X:  2.267822  Expected Y:  9.385154  Got Y:  9.381723  Error:  0.003431  <  0.045000
# at        1557ns SUCCESS  X:  0.239319  Expected Y:  1.270383  Got Y:  1.270383  Error:  0.000001  <  0.045000
# at        1581ns SUCCESS  X:  1.531799  Expected Y:  4.603725  Got Y:  4.603078  Error:  0.000647  <  0.045000
# at        1605ns SUCCESS  X:  3.903503  Expected Y: 39.659827  Got Y: 39.620505  Error:  0.039322  <  0.045000
# at        1629ns SUCCESS  X:  3.112427  Expected Y: 20.324626  Got Y: 20.310633  Error:  0.013993  <  0.045000
# at        1653ns SUCCESS  X:  0.248901  Expected Y:  1.282615  Got Y:  1.282614  Error:  0.000001  <  0.045000
# at        1677ns SUCCESS  X:  3.393311  Expected Y: 25.935299  Got Y: 25.914596  Error:  0.020703  <  0.045000
# at        1701ns SUCCESS  X:  0.620972  Expected Y:  1.860649  Got Y:  1.860629  Error:  0.000020  <  0.045000
#  ALL TESTS PASSED
```

Figure 5: testbench of pipelined design.

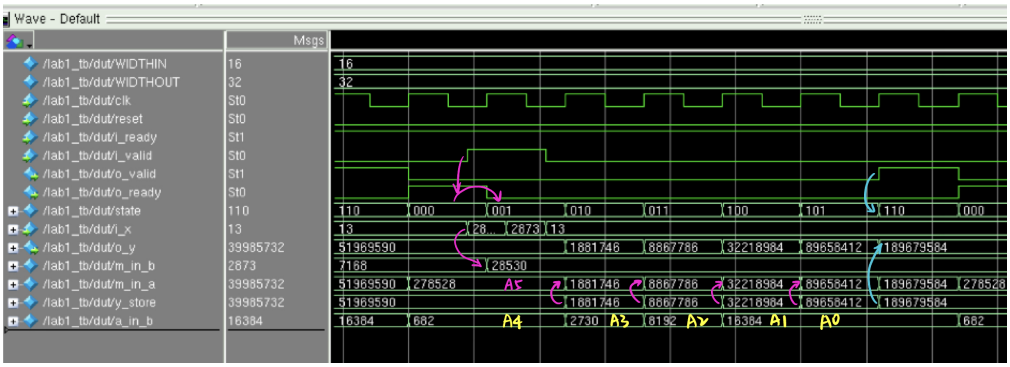

Figure 6: waveform of sharedHW design.

```
# at         189ns SUCCESS  X:  0.000122  Expected Y:  1.000122  Got Y:  1.000122  Error:  0.000000  <  0.045000
# at         357ns SUCCESS  X:  0.437500  Expected Y:  1.548820  Got Y:  1.548814  Error:  0.000006  <  0.045000
# at         525ns SUCCESS  X:  1.741333  Expected Y:  5.653997  Got Y:  5.652892  Error:  0.001105  <  0.045000
# at         693ns SUCCESS  X:  0.175354  Expected Y:  1.191668  Got Y:  1.191668  Error:  0.000000  <  0.045000
# at         861ns SUCCESS  X:  3.548218  Expected Y: 29.578277  Got Y: 29.552900  Error:  0.025378  <  0.045000
# at        1029ns SUCCESS  X:  3.890381  Expected Y: 39.240435  Got Y: 39.201718  Error:  0.038717  <  0.045000
# at        1341ns SUCCESS  X:  0.740234  Expected Y:  2.096173  Got Y:  2.096136  Error:  0.000038  <  0.045000
# at        1509ns SUCCESS  X:  3.255310  Expected Y: 23.027957  Got Y: 23.010813  Error:  0.017144  <  0.045000
# at        1677ns SUCCESS  X:  1.052307  Expected Y:  2.862044  Got Y:  2.861901  Error:  0.000143  <  0.045000
# at        1845ns SUCCESS  X:  1.834961  Expected Y:  6.193981  Got Y:  6.192602  Error:  0.001379  <  0.045000
# at        2013ns SUCCESS  X:  2.957642  Expected Y: 17.717524  Got Y: 17.706403  Error:  0.011121  <  0.045000
# at        2181ns SUCCESS  X:  1.379883  Expected Y:  3.962578  Got Y:  3.962157  Error:  0.000422  <  0.045000
# at        2349ns SUCCESS  X:  3.171143  Expected Y: 21.399463  Got Y: 21.384237  Error:  0.015226  <  0.045000
# at        2517ns SUCCESS  X:  3.684875  Expected Y: 33.155227  Got Y: 33.125056  Error:  0.030170  <  0.045000
# at        2685ns SUCCESS  X:  2.703796  Expected Y: 14.084164  Got Y: 14.076718  Error:  0.007446  <  0.045000
# at        2853ns SUCCESS  X:  0.554443  Expected Y:  1.740929  Got Y:  1.740915  Error:  0.000013  <  0.045000
# at        3021ns SUCCESS  X:  2.880005  Expected Y: 16.525907  Got Y: 16.516037  Error:  0.009870  <  0.045000
# at        3189ns SUCCESS  X:  3.082764  Expected Y: 19.800061  Got Y: 19.786659  Error:  0.013402  <  0.045000
# at        3357ns SUCCESS  X:  3.428040  Expected Y: 26.715896  Got Y: 26.694210  Error:  0.021686  <  0.045000
# at        3525ns SUCCESS  X:  3.298218  Expected Y: 23.899509  Got Y: 23.881316  Error:  0.018193  <  0.045000
# at        3693ns SUCCESS  X:  0.639221  Expected Y:  1.894901  Got Y:  1.894879  Error:  0.000022  <  0.045000
# at        3861ns SUCCESS  X:  2.401550  Expected Y: 10.645304  Got Y: 10.640895  Error:  0.004409  <  0.045000
# at        4029ns SUCCESS  X:  0.701355  Expected Y:  2.016301  Got Y:  2.016270  Error:  0.000031  <  0.045000
# at        4197ns SUCCESS  X:  1.910767  Expected Y:  6.666646  Got Y:  6.665008  Error:  0.001638  <  0.045000
# at        4365ns SUCCESS  X:  2.775940  Expected Y: 15.041535  Got Y: 15.033163  Error:  0.008372  <  0.045000
# at        4605ns SUCCESS  X:  3.417664  Expected Y: 26.480547  Got Y: 26.459158  Error:  0.021389  <  0.045000
# at        4773ns SUCCESS  X:  2.634399  Expected Y: 13.215598  Got Y: 13.208965  Error:  0.006633  <  0.045000
# at        4941ns SUCCESS  X:  3.368408  Expected Y: 25.387968  Got Y: 25.367948  Error:  0.020021  <  0.045000
# at        5109ns SUCCESS  X:  0.598083  Expected Y:  1.818561  Got Y:  1.818544  Error:  0.000017  <  0.045000
# at        5277ns SUCCESS  X:  1.034790  Expected Y:  2.812525  Got Y:  2.812391  Error:  0.000134  <  0.045000
# at        5445ns SUCCESS  X:  3.951538  Expected Y: 41.228160  Got Y: 41.186558  Error:  0.041602  <  0.045000
# at        5613ns SUCCESS  X:  0.823242  Expected Y:  2.277386  Got Y:  2.277330  Error:  0.000056  <  0.045000
# at        5781ns SUCCESS  X:  0.559326  Expected Y:  1.749448  Got Y:  1.749434  Error:  0.000014  <  0.045000
# at        5949ns SUCCESS  X:  0.140198  Expected Y:  1.150501  Got Y:  1.150501  Error:  0.000000  <  0.045000
# at        6117ns SUCCESS  X:  3.374756  Expected Y: 25.526511  Got Y: 25.506319  Error:  0.020193  <  0.045000
# at        6285ns SUCCESS  X:  3.054749  Expected Y: 19.315723  Got Y: 19.302862  Error:  0.012861  <  0.045000
# at        6453ns SUCCESS  X:  0.445984  Expected Y:  1.562015  Got Y:  1.562009  Error:  0.000006  <  0.045000
# at        6621ns SUCCESS  X:  3.698364  Expected Y: 33.527879  Got Y: 33.497199  Error:  0.030680  <  0.045000
# at        6789ns SUCCESS  X:  2.359924  Expected Y: 10.237271  Got Y: 10.233188  Error:  0.004083  <  0.045000
# at        6957ns SUCCESS  X:  1.651001  Expected Y:  5.175761  Got Y:  5.174877  Error:  0.000884  <  0.045000
# at        7125ns SUCCESS  X:  1.010254  Expected Y:  2.744582  Got Y:  2.744460  Error:  0.000122  <  0.045000
# at        7293ns SUCCESS  X:  1.281128  Expected Y:  3.593229  Got Y:  3.592917  Error:  0.000312  <  0.045000
# at        7461ns SUCCESS  X:  2.267822  Expected Y:  9.385154  Got Y:  9.381723  Error:  0.003431  <  0.045000
# at        7629ns SUCCESS  X:  0.239319  Expected Y:  1.270383  Got Y:  1.270383  Error:  0.000001  <  0.045000
# at        7797ns SUCCESS  X:  1.531799  Expected Y:  4.603725  Got Y:  4.603078  Error:  0.000647  <  0.045000
# at        7965ns SUCCESS  X:  3.903503  Expected Y: 39.659827  Got Y: 39.620505  Error:  0.039322  <  0.045000
# at        8133ns SUCCESS  X:  3.112427  Expected Y: 20.324626  Got Y: 20.310633  Error:  0.013993  <  0.045000
# at        8301ns SUCCESS  X:  0.248901  Expected Y:  1.282615  Got Y:  1.282614  Error:  0.000001  <  0.045000
# at        8469ns SUCCESS  X:  3.393311  Expected Y: 25.935299  Got Y: 25.914596  Error:  0.020703  <  0.045000
# at        8637ns SUCCESS  X:  0.620972  Expected Y:  1.860649  Got Y:  1.860629  Error:  0.000020  <  0.045000
#  ALL TESTS PASSED
```

Figure 7: testbench of pipelined design.

3. Table and description

	Baseline Circuit	Pipelined Circuit	Shared HW Circuit
Resources for one circuit	21 ALMs + 9 DSPs	98 ALMs + 9 DSPs	29 ALMs + 2 DSPs
Operating frequency	44.8MHz	249.5MHz	162.79MHz
Critical path	DSP mult-add + 2x DSP mult + LE-based adder + 6x DSP mult + LE-based adder	LE-based mult + LE-based adder + 1x DSP mult	LE-based for mux logic + 2x DSP mult + LE-based adder
Cycles per valid output	1	1	6
Max. # of copies / device	168	168	759
Max. Throughput for a full device (computations/s)	7.53×10^9 (@ 44.8 MHz)	4.19×10^{10} (@ 249.5 MHz)	2.06×10^{10} (@ 162.79 MHz)
Dynamic power of one circuit @ 42 MHz	1.96mW	2.03mW	0.7mW
Max. throughput/Watt for a full device	3.47×10^9	3.45×10^9	2.38×10^9

Table 1: Implementation results

3.1 Resources for one circuit

$$\#\ \text{ALMs} = \text{total ALMs} - \text{unavailable \# ALMs due to virtual I/Os} \tag{1}$$

ALMs and DSPs used in each circuit are shown in figure 8-11.

3.2 Operating frequency

$$\text{operating frequency} = \frac{1}{1 + \text{worst-case setup slack(900mV, 0C)}} \tag{2}$$

Worst-case setup slack in each circuit are shown in figure 12-13.

3.3 Critical path

Critical paths are annotated and shown in figure 14 - 15.

Fitter Resource Usage Summary

<<Filter>>

	Resource	Usage	%
1	Logic utilization (ALMs needed / total ALMs on device)	125 / 427,200	< 1 %
2	ALMs needed [=A-B+C]	125	
1	[A] ALMs used in final placement [=a+b+c+d]	172 / 427,200	< 1 %
1	[a] ALMs used for LUT logic and registers	44	
2	[b] ALMs used for LUT logic	43	
3	[c] ALMs used for registers	85	
4	[d] ALMs used for memory (up to half of total ALMs)	0	
2	[B] Estimate of ALMs recoverable by dense packing	74 / 427,200	< 1 %
3	[C] Estimate of ALMs unavailable [=a+b+c+d]	27 / 427,200	< 1 %
1	[a] Due to location constrained logic	0	
2	[b] Due to LAB-wide signal conflicts	0	
3	[c] Due to LAB input limits	0	
4	[d] Due to virtual I/Os	27	
3			

Figure 8: ALM used in pipelined design.

Fitter Resource Usage Summary

<<Filter>>

	Resource	Usage	%
1	Logic utilization (ALMs needed / total ALMs on device)	56 / 427,200	< 1 %
2	ALMs needed [=A-B+C]	56	
1	[A] ALMs used in final placement [=a+b+c+d]	33 / 427,200	< 1 %
1	[a] ALMs used for LUT logic and registers	15	
2	[b] ALMs used for LUT logic	14	
3	[c] ALMs used for registers	4	
4	[d] ALMs used for memory (up to half of total ALMs)	0	
2	[B] Estimate of ALMs recoverable by dense packing	4 / 427,200	< 1 %
3	[C] Estimate of ALMs unavailable [=a+b+c+d]	27 / 427,200	< 1 %
1	[a] Due to location constrained logic	0	
2	[b] Due to LAB-wide signal conflicts	0	
3	[c] Due to LAB input limits	0	
4	[d] Due to virtual I/Os	27	
3			

Figure 9: ALM used in sharedHW design.

Fitter Resource Utilization by Entity

<<Filter>>

ALMs used for memory	Combinational ALUTs	Dedicated Logic Registers	I/O Registers	Block Memory Bits	M20Ks	DSP Blocks
0.0 (0.0)	173 (1)	299 (155)	0 (0)	0	0	9

Figure 10: DSP used in pipelined design.

3.4 Cycles per valid output

After saturation, pipelined design could generate 1 valid output per cycle. (see waveform figure 4) Due to the FSM used in sharedHW design, 6 cycles are needed for 1 valid output. (see waveform figure 6)

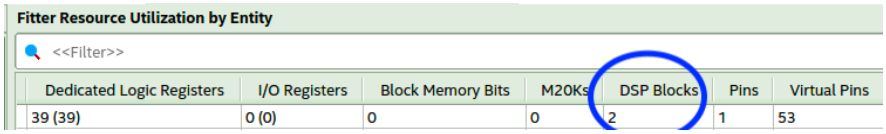

Figure 11: DSP used in sharedHW design.

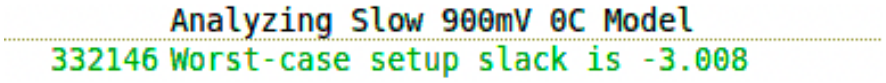

Figure 12: worst-case setup slack(900mV, 0C) in pipelined design.

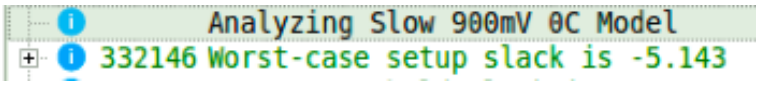

Figure 13: worst-case setup slack(900mV, 0C) in sharedHW design.

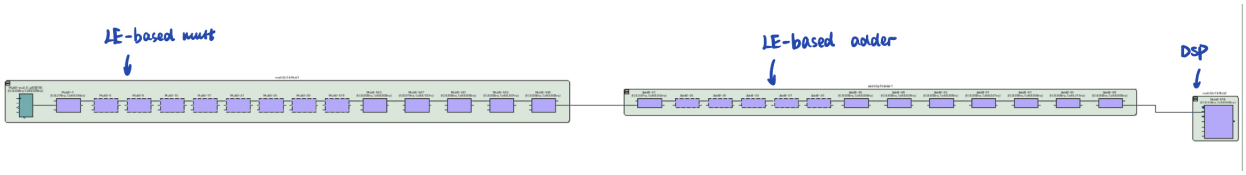

Figure 14: critical path for pipelined design.

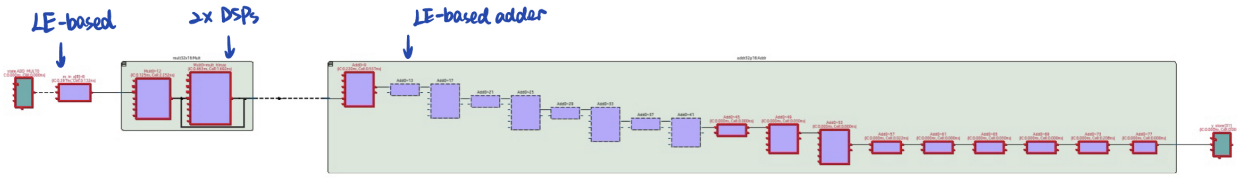

Figure 15: critical path for sharedHW design.

3.5 Max. # of copies / device

$$\text{Max. \# of copies / device} = 1518(\text{total \# of DSPs}) \div (\text{\# of DSPs used in each design}) \quad (3)$$

3.6 Max. Throughput for a full device

$$\text{Max. Throughput for a full device} = \text{operating frequency in (2)} \div \text{cycles per valid output} \times \text{Max. \# of copies per device in (3)} \quad (4)$$

3.7 Dynamic power of one circuit

$$\text{Dynamic power of one circuit @ 42 MHz} = \text{Thermal Power of (DSP + Combinational cell + Register cell} \tag{5}$$

Thermal power by block type for each circuit is denoted in figure 16 - 17.

Thermal Power Dissipation by Block Type

<<Filter>>

	Block Type	Total Thermal Power by Block Type
1	DSP	0.99 mW
2	Combinational cell	0.33 mW
3	Register cell	0.71 mW

Figure 16: thermal power for pipelined design.

Thermal Power Dissipation by Block Type

<<Filter>>

	Block Type	Total Thermal Power by Block Type
1	DSP	0.47 mW
2	Combinational cell	0.11 mW
3	Register cell	0.12 mW

Figure 17: thermal power for sharedHW design.

3.8 Max. throughput/Watt for a full device

$$\text{total power @ 42 MHz} = \text{dynamic power in (5)} \times \text{copies in (3)} + \text{IO thermal power} + \text{device static thermal power} \tag{6}$$

$$\text{total power @ operating frequency} = \frac{\text{total power (in (6))} \times \text{operating frequency in (2)}}{42\,\text{MHz}} \tag{7}$$

$$\text{throughput/Watt per device} = \frac{\text{Max.Throughput in (4)}}{\text{total power @ operating frequency in (7)}} \tag{8}$$

4. Discussion

(a) 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 the error?

One of the sources of error is caused by the limited terms hardware used in the implementation. We only include the first 6 terms of the approximation polynomial for our hardware implementation(a 5-degree polynomial, n = 5). Taylor expansion would be more accurate as the value of n increases. To reduce this type of source error, we could make the circuit to include more higher-order terms. Notice that the coefficient of each term is decreasing with the order getting higher. Thus, the enhancement in accuracy doesn't significantly increase as more terms are incorporated, while simultaneously, the inclusion of these terms also escalates the complexity of the hardware(see Figure 18). We should manage this trade-off and choose wisely a number of terms to be used in the hardware.

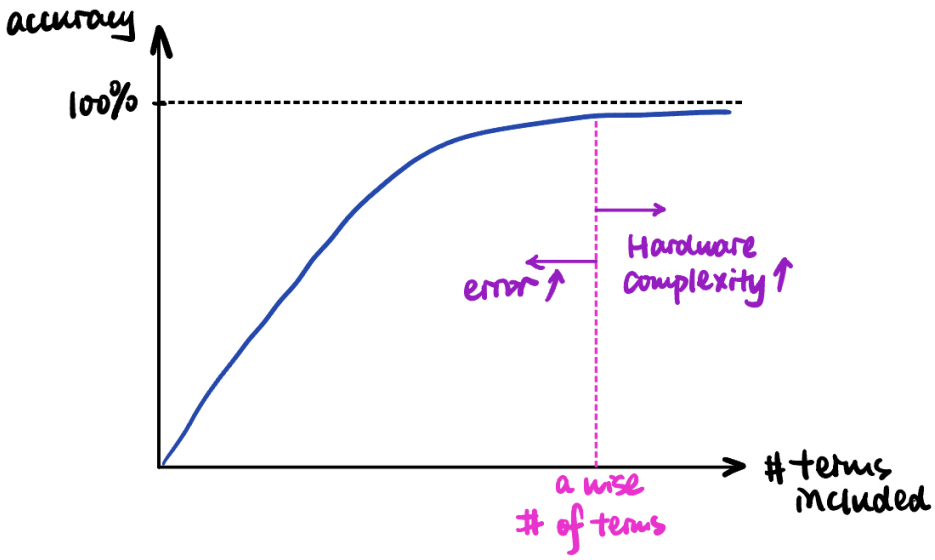

Figure 18: accuracy-complexity trade off.

The other type of the source of error is precision used in the fixed-point taylor coefficients. It will also reduce circuit's error if we could include more fractional bits.

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

From the table 1, pipelined circuit achieves the highest throughput/device, followed by shared circuit, and then followed by baseline circuit.

Pipelined vs. baseline:

Both baseline and pipelined circuits are able to provide a valid output per cycle. Since we shorten the critical path by inserting registers for the pipelined, the pipelined circuit achieves at a higher frequency. i.e. Pipelined circuit generates valid output per cycle more frequently than the baseline circuit.

Pipelined vs. shared:

In order to reuse the same LEs(for add) and DSPs(for multiplication) resources, we introduced FSM for the shared circuit. There will be in total 6 cycles needed for the shared circuit to generate a valid output, while a new input needs to stall until the previous output has been generated. Moreover, the shared circuit results in a longer critical path compared with pipelined. In conclusion, pipelined is running at a higher frequency with fewer cycles to generate a valid output. It is obvious that pipelined's throughput is greater than the shared's.

Shared vs. baseline:

Although shared circuit achieves a higher frequency, it needs 6 cycles to generate a valid output, which drags the throughput lower than baseline(in one copy). However, since LEs and DSPs are reused, shared circuit can implement 759 copies. It results in a higher throughput by spanning 759 copies in the device.

(c) Look at the average toggle rates(how often the average signal changes) for the 3 circuits. Explain why some circuit styles lead to higher toggle rates 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.

	Baseline Circuit	Pipelined Circuit	Shared HW Circuit
toggle rate (millions of transitions/sec)	12.819	11.527	13.257

Table 2: toggle rate results

From table 2, shared has the highest toggle rates, followed by baseline, and then followed by pipelined circuit. Shared circuit reuse the same logic and DSPs. Under the same frequency 42MHz, logic and DSPs for shared circuit switch output the most in order to compute different outputs during different states of FSM. While in baseline and pipelined, it will "hold" longer since the resources are not reused for the same input. Pipelined circuit achieve the lowest

toggle rates. Due to the insertion of pipeline registers, output of the each stage will be hold in the register(acts like "store"). Compared to baseline circuit, which needs propagate each output to the next resources(LEs and DSPs), it reduces toggle rate.

From table 1, baseline has the highest power-efficiency, followed by pipelined(slightly lower), and then followed by shared circuit.

pipelined vs. baseline:

Despite the fact that pipeline achieves the highest throughput, it uses more resources due to the additional registers in the circuit. The extra power consumption leads to a slightly lower power-efficiency compared to the baseline circuit.

pipelined vs. shared:

Shared circuit obtains the lowest dynamic power by restricting resources used during computation. However, it takes 6 cycles for a valid output, which slows down the computation efficiency, results in a lower power-efficiency than pipelined circuit.

shared vs. baseline:

If we convert the throughput to the same frequency @ 42MHz, we can conclude that the reduced throughput of the shared circuit ultimately contributes to a lower power efficiency compared to the baseline.

Appendix

Appendix I: HDL source code for pipelined circuit

```
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
//32 bits are of interest, and keep them.
logic [WIDTHIN-1:0] x;          // Register to hold input X
logic [WIDTHOUT-1:0] y_Q;          // Register to hold output Y
logic valid_Q1;                 // Output of register x is valid
logic valid_Q2;                 // Output of register y is valid

// signal for enabling sequential circuit elements
logic enable;
```

```
// Signals for computing the y output
logic [WIDTHOUT-1:0] m0_out; // A5 * x
logic [WIDTHOUT-1:0] a0_out; // A5 * x + A4
logic [WIDTHOUT-1:0] m1_out; // (A5 * x + A4) * x
logic [WIDTHOUT-1:0] a1_out; // (A5 * x + A4) * x + A3
logic [WIDTHOUT-1:0] m2_out; // ((A5 * x + A4) * x + A3) * x
logic [WIDTHOUT-1:0] a2_out; // ((A5 * x + A4) * x + A3) * x + A2
logic [WIDTHOUT-1:0] m3_out; // (((A5 * x + A4) * x + A3) * x + A2) * x
logic [WIDTHOUT-1:0] a3_out; // (((A5 * x + A4) * x + A3) * x + A2) * x +
↪  A1
logic [WIDTHOUT-1:0] m4_out; // ((((A5 * x + A4) * x + A3) * x + A2) * x +
↪  A1) * x
logic [WIDTHOUT-1:0] a4_out; // ((((A5 * x + A4) * x + A3) * x + A2) * x +
↪  A1) * x + A0
logic [WIDTHOUT-1:0] y_D;

// Signals for pipeline registers
logic [WIDTHOUT-1:0] m0_out_reg; // pipeline registers for m0_out
logic [WIDTHOUT-1:0] a0_out_reg; // pipeline registers for a0_out
logic [WIDTHOUT-1:0] m1_out_reg; // pipeline registers for m1_out
logic [WIDTHOUT-1:0] a1_out_reg; // pipeline registers for a1_out
logic [WIDTHOUT-1:0] m2_out_reg; // pipeline registers for m2_out
logic [WIDTHOUT-1:0] a2_out_reg; // pipeline registers for a2_out
logic [WIDTHOUT-1:0] m3_out_reg; // pipeline registers for m3_out
logic [WIDTHOUT-1:0] a3_out_reg; // pipeline registers for a3_out
logic [WIDTHOUT-1:0] m4_out_reg; // pipeline registers for m4_out

logic [WIDTHIN-1:0] x_reg; // pipeline registers for x
logic [WIDTHIN-1:0] x_m1; //pipeline registers for x_reg (input to Mult1)
logic [WIDTHIN-1:0] x_m1_reg; // pipeline registers for x_m1
logic [WIDTHIN-1:0] x_m2; //pipeline registers for x_m1_reg (input to
↪  Mult2)
logic [WIDTHIN-1:0] x_m2_reg; // pipeline registers for x_m2
logic [WIDTHIN-1:0] x_m3; //pipeline registers for x_m2_reg (input to
↪  Mult3)
logic [WIDTHIN-1:0] x_m3_reg; // pipeline registers for x_m3
logic [WIDTHIN-1:0] x_m4; //pipeline registers for x_m3_reg (input to
↪  Mult4)

logic valid_reg;                // Output of register valid_Q1
logic valid_m1;                       // Output of register valid_reg
logic valid_m1_reg;     // Output of register valid_m1
```

```
logic valid_m2;         // Output of register valid_m1_reg
logic valid_m2_reg;     // Output of register valid_m2
logic valid_m3;         // Output of register valid_m2_reg
logic valid_m3_reg;           // Output of register valid_m3
logic valid_m4;               // Output of register valid_m3_reg
logic valid_m4_reg;               // Output of register valid_m4

// compute y value
mult16x16 Mult0 (.i_dataa(A5),                  .i_datab(x),
 ↪          .o_res(m0_out));
addr32p16 Addr0 (.i_dataa(m0_out_reg),          .i_datab(A4),
 ↪          .o_res(a0_out));

mult32x16 Mult1 (.i_dataa(a0_out_reg),          .i_datab(x_m1),
 ↪          .o_res(m1_out));
addr32p16 Addr1 (.i_dataa(m1_out_reg),          .i_datab(A3),
 ↪          .o_res(a1_out));

mult32x16 Mult2 (.i_dataa(a1_out_reg),          .i_datab(x_m2),
 ↪          .o_res(m2_out));
addr32p16 Addr2 (.i_dataa(m2_out_reg),          .i_datab(A2),
 ↪          .o_res(a2_out));

mult32x16 Mult3 (.i_dataa(a2_out_reg),          .i_datab(x_m3),
 ↪          .o_res(m3_out));
addr32p16 Addr3 (.i_dataa(m3_out_reg),          .i_datab(A1),
 ↪          .o_res(a3_out));

mult32x16 Mult4 (.i_dataa(a3_out_reg),          .i_datab(x_m4),
 ↪          .o_res(m4_out));
addr32p16 Addr4 (.i_dataa(m4_out_reg),          .i_datab(A0),
 ↪          .o_res(a4_out));

assign y_D = a4_out;

// Combinational logic
always_comb begin
        // signal for enable
        enable = i_ready;
end

// Infer the registers
```

```
always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
                valid_Q1 <= 1'b0;
                valid_reg <= 1'b0;
                valid_m1 <= 1'b0;
                valid_m1_reg <= 1'b0;
                valid_m2 <= 1'b0;
                valid_m2_reg <= 1'b0;
                valid_m3 <= 1'b0;
                valid_m3_reg <= 1'b0;
                valid_m4 <= 1'b0;
                valid_m4_reg <= 1'b0;
                valid_Q2 <= 1'b0;

                x <= 0;
                y_Q <= 0;

                m0_out_reg <= 'd0;
                a0_out_reg <= 'd0;
                m1_out_reg <= 'd0;
                a1_out_reg <= 'd0;
                m2_out_reg <= 'd0;
                a2_out_reg <= 'd0;
                m3_out_reg <= 'd0;
                a3_out_reg <= 'd0;
                m4_out_reg <= 'd0;

                x_reg <= 'd0;
                x_m1 <= 'd0;
                x_m1_reg <= 'd0;
                x_m2 <= 'd0;
                x_m2_reg <= 'd0;
                x_m3 <= 'd0;
                x_m3_reg <= 'd0;
                x_m4 <= 'd0;

        end else if (enable) begin

                // pipeline for mult-add data path
                m0_out_reg <= m0_out;
                a0_out_reg <= a0_out;
                m1_out_reg <= m1_out;
```

```
                a1_out_reg <= a1_out;
                m2_out_reg <= m2_out;
                a2_out_reg <= a2_out;
                m3_out_reg <= m3_out;
                a3_out_reg <= a3_out;
                m4_out_reg <= m4_out;
                // output computed y value
                y_Q <= y_D;

                // pipeline for carry-over x data path
                // read in new x value
                x <= i_x;
                x_reg <= x;
                x_m1 <= x_reg;
                x_m1_reg <= x_m1;
                x_m2 <= x_m1_reg;
                x_m2_reg <= x_m2;
                x_m3 <= x_m2_reg;
                x_m3_reg <= x_m3;
                x_m4 <= x_m3_reg;

                // pipeline for i_valid to o_valid data path
                // propagate the valid value
                valid_Q1 <= i_valid;
                valid_reg <= valid_Q1;
                valid_m1 <= valid_reg;
                valid_m1_reg <= valid_m1;
                valid_m2 <= valid_m1_reg;
                valid_m2_reg <= valid_m2;
                valid_m3 <= valid_m2_reg;
                valid_m3_reg <= valid_m3;
                valid_m4 <= valid_m3_reg;
                valid_m4_reg <= valid_m4;
                valid_Q2 <= valid_m4_reg;

        end
end

// assign outputs
assign o_y = y_Q;
// ready for inputs as long as receiver is ready for outputs */
assign o_ready = i_ready;
```

```
// the output is valid as long as the corresponding input was valid and
//        the receiver is ready. If the receiver isn't ready, the computed
↪  output
//        will still remain on the register outputs and the circuit will
↪  resume
//  normal operation when the receiver is ready again (i_ready is high)
assign o_valid = valid_Q2 & i_ready;

endmodule

/*****************************************************************************

// Multiplier module for the first 16x16 multiplication
module mult16x16 (
        input  [15:0] i_dataa,
        input  [15:0] i_datab,
        output [31:0] o_res
);

logic [31:0] result;

always_comb begin
        result = i_dataa * i_datab;
end

// The result of Q2.14 x Q2.14 is in the Q4.28 format. Therefore we need
↪  to change it
// to the Q7.25 format specified in the assignment by shifting right and
↪  padding with zeros.
assign o_res = {3'b000, result[31:3]};

endmodule

/*****************************************************************************

// Multiplier module for all the remaining 32x16 multiplications
module mult32x16 (
        input  [31:0] i_dataa,
        input  [15:0] i_datab,
        output [31:0] o_res
);
```

```
logic [47:0] result;

always_comb begin
        result = i_dataa * i_datab;
end

// The result of Q7.25 x Q2.14 is in the Q9.39 format. Therefore we need
↪  to change it
// to the Q7.25 format specified in the assignment by selecting the
↪  appropriate bits
// (i.e. dropping the most-significant 2 bits and least-significant 14
↪  bits).
assign o_res = result[45:14];

endmodule

/*****************************************************************************

// Adder module for all the 32b+16b addition operations
module addr32p16 (
        input [31:0] i_dataa,
        input [15:0] i_datab,
        output [31:0] o_res
);

// The 16-bit Q2.14 input needs to be aligned with the 32-bit Q7.25 input
↪  by zero padding
assign o_res = i_dataa + {5'b00000, i_datab, 11'b00000000000};

endmodule

/*****************************************************************************
```

Appendix II: HDL source code for sharedHW circuit

```
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
//32 bits are of interest, and keep them.

// signal for enabling sequential circuit elements
logic enable;

// Signals for inputs that feeds in multplier and adder operator
logic [WIDTHOUT-1:0] m_in_a;
logic [WIDTHOUT-1:0] a_in_a;
logic [WIDTHIN-1:0] m_in_b;
logic [WIDTHIN-1:0] a_in_b;
// Signals for output from adder operator
```

```
logic [WIDTHOUT-1:0] y_D;
// Register to hold output y_D
logic [WIDTHOUT-1:0] y_store;

// // Select signals for choosing inputs for operators
logic multsel_in;
logic [2:0] addsel_in;

// Enumeration for select signals' value
enum logic {A5_IN, STORE_IN} multsel_val;
enum logic [2:0] {A4_IN, A3_IN, A2_IN, A1_IN, A0_IN} addsel_val;
// Enumeration for FSM state value
enum logic [2:0] { PREPARE, ADD_MULT0, ADD_MULT1, ADD_MULT2, ADD_MULT3,
↪  ADD_MULT4, OUTPUT} state_val;
logic [2:0] state;
logic [2:0] next_state;

// Signals for output from FSM control
logic start_x;
logic occupied;
logic wb_y;
logic output_valid;

// Compute value using one multiplier and one adder
mult32x16 Mult (.i_dataa(m_in_a),         .i_datab(m_in_b),
↪          .o_res(a_in_a));
addr32p16 Addr (.i_dataa(a_in_a),         .i_datab(a_in_b),
↪          .o_res(y_D));

always_comb begin
        // signal for enable
        enable = i_ready;
end

// Combinational logic for signals
always_comb begin

        // Mux logic: choosing correct inputs
        case(multsel_in)
                A5_IN: m_in_a = {5'd0, A5, 11'd0};
                STORE_IN: m_in_a = y_store;
                default: m_in_a = '0;
```

```
        endcase
        case(addsel_in)
                A4_IN: a_in_b = A4;
                A3_IN: a_in_b = A3;
                A2_IN: a_in_b = A2;
                A1_IN: a_in_b = A1;
                A0_IN: a_in_b = A0;
                default: a_in_b = '0;
        endcase

        // fsm state transition
        case(state)
                PREPARE: if(i_valid) begin
                        next_state = ADD_MULT0;
                end else begin
                        next_state = PREPARE;
                end
                ADD_MULT0: next_state = ADD_MULT1;
                ADD_MULT1: next_state = ADD_MULT2;
                ADD_MULT2: next_state = ADD_MULT3;
                ADD_MULT3: next_state = ADD_MULT4;
                ADD_MULT4: next_state = OUTPUT;
                OUTPUT: if(enable)begin
                        next_state = PREPARE;
                end else begin
                        next_state = OUTPUT;
                end

                default: next_state = PREPARE;
        endcase
end

// Fsm control flow for signal ouputs of each state
always_comb begin
          case(state)
                 PREPARE:begin
                        occupied = 0;
                        wb_y = 0;
                        output_valid = 0;
                        if(i_valid)begin
                                start_x = 1;
                        end else begin
```

```
                start_x = 0;
        end
        multsel_in = '0;
        addsel_in = '0;
end
ADD_MULT0: begin
        start_x = 0;
        multsel_in = A5_IN;
        addsel_in = A4_IN;
        occupied = 1;
        wb_y = 1;
        output_valid = 0;
end
ADD_MULT1: begin
        start_x = 0;
        multsel_in = STORE_IN;
        addsel_in = A3_IN;
        occupied = 1;
        wb_y = 1;
        output_valid = 0;
end
ADD_MULT2: begin
        start_x = 0;
        multsel_in = STORE_IN;
        addsel_in = A2_IN;
        occupied = 1;
        wb_y = 1;
        output_valid = 0;
end
ADD_MULT3: begin
        start_x = 0;
        multsel_in = STORE_IN;
        addsel_in = A1_IN;
        occupied = 1;
        wb_y = 1;
        output_valid = 0;
end
ADD_MULT4: begin
        start_x = 0;
        multsel_in = STORE_IN;
        addsel_in = A0_IN;
        occupied = 1;
```

```
                        wb_y = 1;
                        output_valid = 0;
                end
                OUTPUT:begin
                        start_x = 0;
                        multsel_in = STORE_IN;
                        addsel_in = A0_IN;
                        occupied = 1;
                        wb_y = 0;
                        output_valid = 1;
                end
                default:begin
                        start_x = 0;
                        occupied = 1;
                        wb_y = 0;
                        output_valid = 0;
                        multsel_in = '0;
                        addsel_in = '0;
                end
          endcase
end

// Infer the registers based on FSM control signals
always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
                state <= PREPARE;
                m_in_b <= '0;
                y_store <= '0;

        end else if (enable) begin
                //move to the next state
                state <= next_state;

                // read in new x value for multiplier input
                if (start_x) begin
                        m_in_b <= i_x;
                end

                // write back the output from one round of mult+add
                if(wb_y) begin
                        y_store <= y_D;
                end
```

```
        end
end

// assign outputs
assign o_y = y_store;
assign o_valid = output_valid;
// ready for inputs when receiver is ready for outputs and no more
↪  operator being occupied
assign o_ready = i_ready & ~occupied;

endmodule

/*****************************************************************************

// Multiplier module for all the remaining 32x16 multiplications
module mult32x16 (
        input  [31:0] i_dataa,
        input  [15:0] i_datab,
        output [31:0] o_res
);

logic [47:0] result;

always_comb begin
        result = i_dataa * i_datab;
end

// The result of Q7.25 x Q2.14 is in the Q9.39 format. Therefore we need
↪  to change it
// to the Q7.25 format specified in the assignment by selecting the
↪  appropriate bits
// (i.e. dropping the most-significant 2 bits and least-significant 14
↪  bits).
assign o_res = result[45:14];

endmodule

/*****************************************************************************

// Adder module for all the 32b+16b addition operations
module addr32p16 (
        input [31:0] i_dataa,
```

```
        input [15:0] i_datab,
        output [31:0] o_res
);

// The 16-bit Q2.14 input needs to be aligned with the 32-bit Q7.25 input
↪  by zero padding
assign o_res = i_dataa + {5'b00000, i_datab, 11'b00000000000};

endmodule

/*************************************************************************************
```