Pipelined Implementation

Figure 1 below shows the pipelined implementation of the exponential function. As we can see from the zoomed red box, registers are added between each multiplication block and adder block. In this way, the critical path is shortened to a single multiplication block and pipeline optimization is applied.

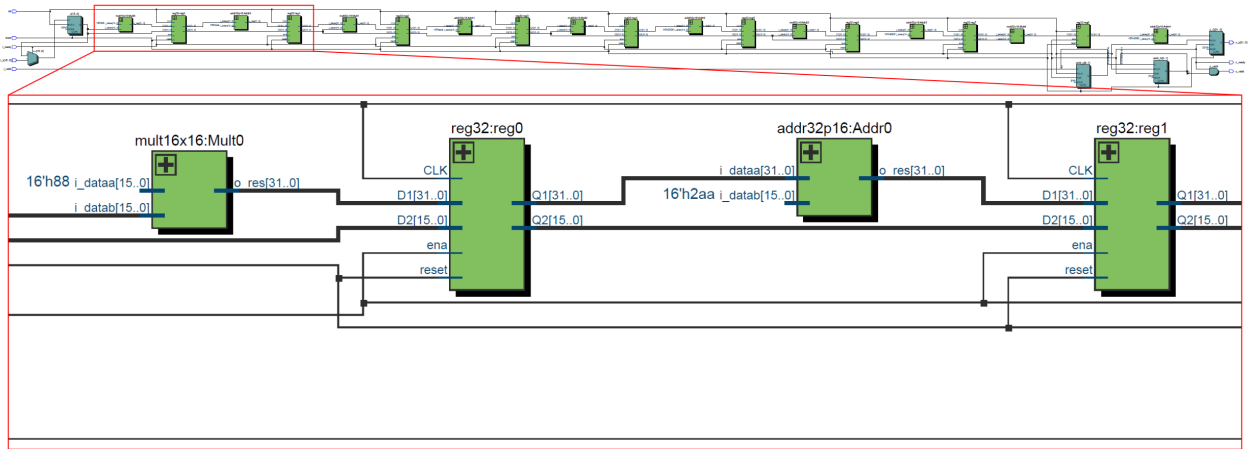

Figure 1: Block diagram of the pipelined implementation.

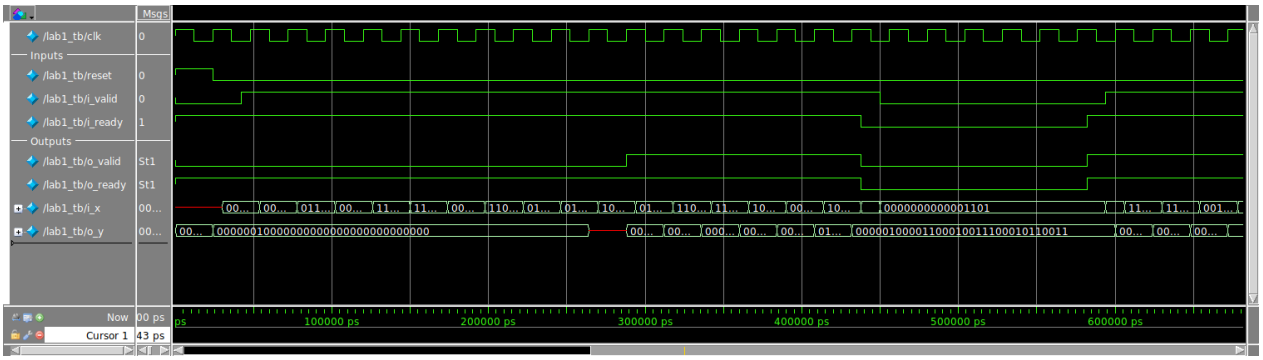

Figure 2: waveform of the pipelined implementation.

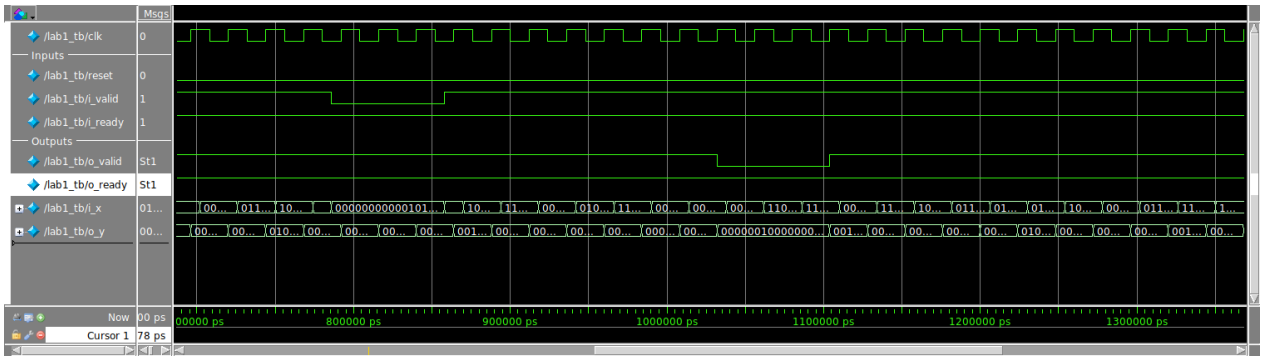

Figure 3: waveform of the pipelined implementation.

```
# 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
```

Testbench 1: Pipelined implementation.

Shared Operator Implementation

The shared operator RTL diagram is shown in figure 4. This design used one 32x16 multiplier (shown in the middle green block) and one 32+16 adder (shown in the right green box) to reduce the DSP block utilization. In addition to the baseline version, a Finite-state-machine (shown in the left green box) is added to control the multiplexers and output signals. Two multiplexers are introduced. One is 2to1 multiplexer with inputs from A5 parameter and init_x register, where the init_x register keeps the initial i_x value. The second multiplexer is a 5to1 mux, it contains the parameters from A4 to A0 and feed the correct parameter to the adder during computation.

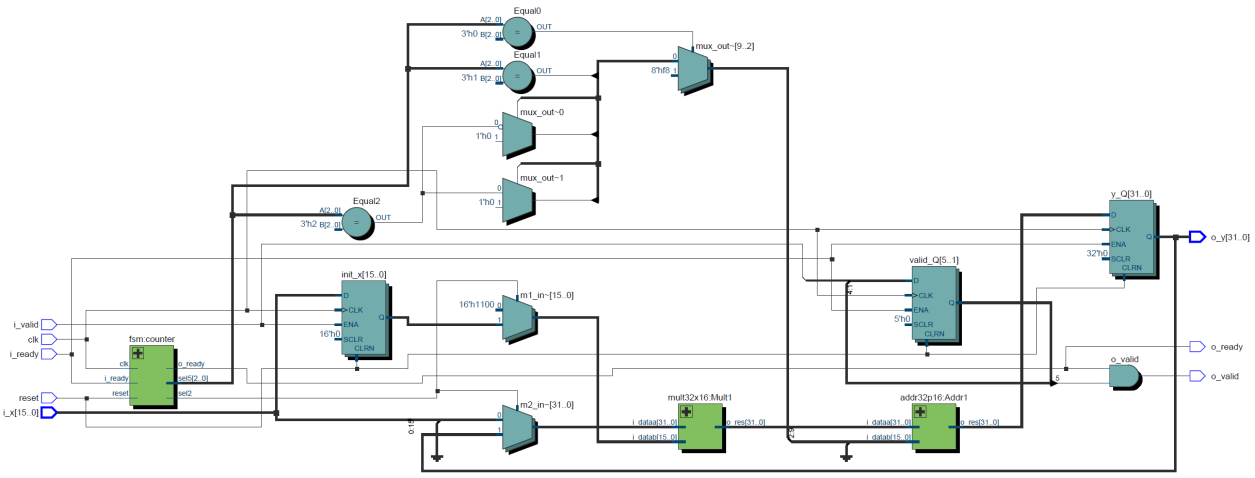

Figure 4: Block diagram of the shared operator implementation.

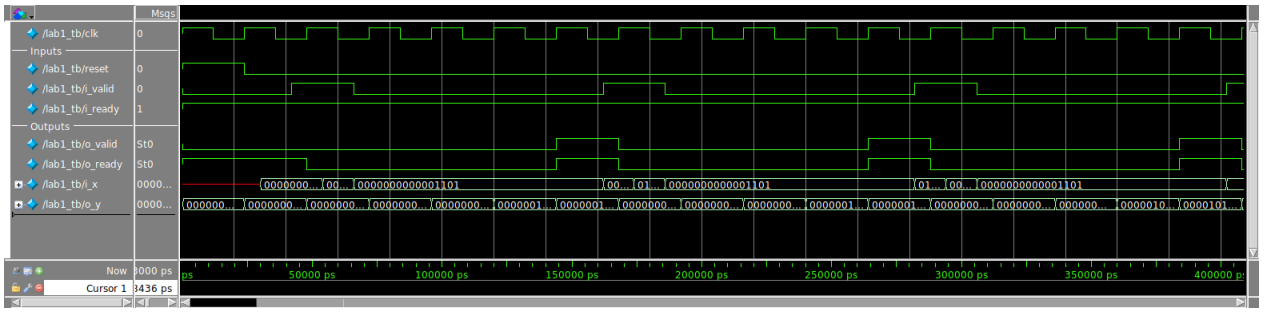

Figure 5: Waveform of the shared operator implementation.

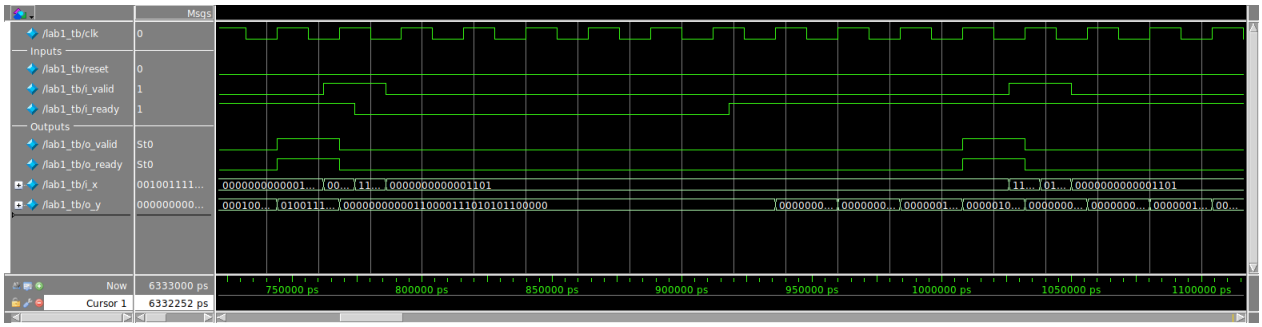

Figure 6: Waveform of the shared operator implementation.

```
# at  165ns SUCCESS X: 0.000122 Expected Y: 1.000122 Got Y: 1.000122 Error: 0.000000 <
    0.045000
# at  285ns SUCCESS X: 0.437500 Expected Y: 1.548820 Got Y: 1.548814 Error: 0.000006 <
    0.045000
# at  405ns SUCCESS X: 1.741333 Expected Y: 5.653997 Got Y: 5.652892 Error: 0.001105 <
    0.045000
# at  525ns SUCCESS X: 0.175354 Expected Y: 1.191668 Got Y: 1.191668 Error: 0.000000 <
    0.045000
# at  645ns SUCCESS X: 3.548218 Expected Y: 29.578277 Got Y: 29.552900 Error: 0.025378 <
    0.045000
# at  765ns SUCCESS X: 3.890381 Expected Y: 39.240435 Got Y: 39.201718 Error: 0.038717 <
    0.045000
# at 1029ns SUCCESS X: 0.740234 Expected Y: 2.096173 Got Y: 2.096136 Error: 0.000038 <
    0.045000
# at 1149ns SUCCESS X: 3.255310 Expected Y: 23.027957 Got Y: 23.010813 Error: 0.017144 <
    0.045000
# at 1269ns SUCCESS X: 1.052307 Expected Y: 2.862044 Got Y: 2.861901 Error: 0.000143 <
    0.045000
# at 1389ns SUCCESS X: 1.834961 Expected Y: 6.193981 Got Y: 6.192602 Error: 0.001379 <
    0.045000
# at 1509ns SUCCESS X: 2.957642 Expected Y: 17.717524 Got Y: 17.706403 Error: 0.011121 <
    0.045000
# at 1629ns SUCCESS X: 1.379883 Expected Y: 3.962578 Got Y: 3.962157 Error: 0.000422 <
    0.045000
# at 1749ns SUCCESS X: 3.171143 Expected Y: 21.399463 Got Y: 21.384237 Error: 0.015226 <
    0.045000
# at 1869ns SUCCESS X: 3.684875 Expected Y: 33.155227 Got Y: 33.125056 Error: 0.030170 <
    0.045000
# at 1989ns SUCCESS X: 2.703796 Expected Y: 14.084164 Got Y: 14.076718 Error: 0.007446 <
    0.045000
# at 2109ns SUCCESS X: 0.554443 Expected Y: 1.740929 Got Y: 1.740915 Error: 0.000013 <
    0.045000
# at 2229ns SUCCESS X: 2.880005 Expected Y: 16.525907 Got Y: 16.516037 Error: 0.009870 <
    0.045000
# at 2349ns SUCCESS X: 3.082764 Expected Y: 19.800061 Got Y: 19.786659 Error: 0.013402 <
    0.045000
# at 2469ns SUCCESS X: 3.428040 Expected Y: 26.715896 Got Y: 26.694210 Error: 0.021686 <
    0.045000
# at 2589ns SUCCESS X: 3.298218 Expected Y: 23.899509 Got Y: 23.881316 Error: 0.018193 <
    0.045000
# at 2709ns SUCCESS X: 0.639221 Expected Y: 1.894901 Got Y: 1.894879 Error: 0.000022 <
    0.045000
# at 2829ns SUCCESS X: 2.401550 Expected Y: 10.645304 Got Y: 10.640895 Error: 0.004409 <
    0.045000
# at 2949ns SUCCESS X: 0.701355 Expected Y: 2.016301 Got Y: 2.016270 Error: 0.000031 <
    0.045000
```

```
# at 3069ns SUCCESS X: 1.910767 Expected Y: 6.666646 Got Y: 6.665008 Error: 0.001638 <
    0.045000
# at 3189ns SUCCESS X: 2.775940 Expected Y: 15.041535 Got Y: 15.033163 Error: 0.008372 <
    0.045000
# at 3429ns SUCCESS X: 3.417664 Expected Y: 26.480547 Got Y: 26.459158 Error: 0.021389 <
    0.045000
# at 3549ns SUCCESS X: 2.634399 Expected Y: 13.215598 Got Y: 13.208965 Error: 0.006633 <
    0.045000
# at 3669ns SUCCESS X: 3.368408 Expected Y: 25.387968 Got Y: 25.367948 Error: 0.020021 <
    0.045000
# at 3789ns SUCCESS X: 0.598083 Expected Y: 1.818561 Got Y: 1.818544 Error: 0.000017 <
    0.045000
# at 3909ns SUCCESS X: 1.034790 Expected Y: 2.812525 Got Y: 2.812391 Error: 0.000134 <
    0.045000
# at 4029ns SUCCESS X: 3.951538 Expected Y: 41.228160 Got Y: 41.186558 Error: 0.041602 <
    0.045000
# at 4149ns SUCCESS X: 0.823242 Expected Y: 2.277386 Got Y: 2.277330 Error: 0.000056 <
    0.045000
# at 4269ns SUCCESS X: 0.559326 Expected Y: 1.749448 Got Y: 1.749434 Error: 0.000014 <
    0.045000
# at 4389ns SUCCESS X: 0.140198 Expected Y: 1.150501 Got Y: 1.150501 Error: 0.000000 <
    0.045000
# at 4509ns SUCCESS X: 3.374756 Expected Y: 25.526511 Got Y: 25.506319 Error: 0.020193 <
    0.045000
# at 4629ns SUCCESS X: 3.054749 Expected Y: 19.315723 Got Y: 19.302862 Error: 0.012861 <
    0.045000
# at 4749ns SUCCESS X: 0.445984 Expected Y: 1.562015 Got Y: 1.562009 Error: 0.000006 <
    0.045000
# at 4869ns SUCCESS X: 3.698364 Expected Y: 33.527879 Got Y: 33.497199 Error: 0.030680 <
    0.045000
# at 4989ns SUCCESS X: 2.359924 Expected Y: 10.237271 Got Y: 10.233188 Error: 0.004083 <
    0.045000
# at 5109ns SUCCESS X: 1.651001 Expected Y: 5.175761 Got Y: 5.174877 Error: 0.000884 <
    0.045000
# at 5229ns SUCCESS X: 1.010254 Expected Y: 2.744582 Got Y: 2.744460 Error: 0.000122 <
    0.045000
# at 5349ns SUCCESS X: 1.281128 Expected Y: 3.593229 Got Y: 3.592917 Error: 0.000312 <
    0.045000
# at 5469ns SUCCESS X: 2.267822 Expected Y: 9.385154 Got Y: 9.381723 Error: 0.003431 <
    0.045000
# at 5589ns SUCCESS X: 0.239319 Expected Y: 1.270383 Got Y: 1.270383 Error: 0.000001 <
    0.045000
# at 5709ns SUCCESS X: 1.531799 Expected Y: 4.603725 Got Y: 4.603078 Error: 0.000647 <
    0.045000
# at 5829ns SUCCESS X: 3.903503 Expected Y: 39.659827 Got Y: 39.620505 Error: 0.039322 <
    0.045000
# at 5949ns SUCCESS X: 3.112427 Expected Y: 20.324626 Got Y: 20.310633 Error: 0.013993 <
    0.045000
# at 6069ns SUCCESS X: 0.248901 Expected Y: 1.282615 Got Y: 1.282614 Error: 0.000001 <
    0.045000
# at 6189ns SUCCESS X: 3.393311 Expected Y: 25.935299 Got Y: 25.914596 Error: 0.020703 <
    0.045000
# at 6309ns SUCCESS X: 0.620972 Expected Y: 1.860649 Got Y: 1.860629 Error: 0.000020 <
    0.045000
# ALL TESTS PASSED
```

Testbench 2: Shared operator implementation.

Table Analysis

Filled version of implementation results table is shown in table 1. I will use the pipelined implementation as an example on how to get the values. Baseline and shared HW will follow the same methodology.

Resources for one circuit is obtained by checking the Resource Section in Quartus. The circuit uses 125 ALMs and 9 DSPs, where 27 of 125 ALMs are used by virtual pins. Therefore, the circuit uses 125 ALMs and 9 DSPs.

Operating frequency is obtained by checking the worst-case setup slack of the slow 900mV 0C model, which is -2.835. This means the circuit can run at a maximum rate of 3.835 ns, which is 260.76 MHz.

Critical path is obtained by examining the Data Arrival Path in Timing Analyzer.

Cycles per valid output is confirmed with the waveform from ModelSim.

Max.# of copies/device is calculated by checking the summary report from Fitter. Given the FPGA has a total of 427,200 ALMs and 1518 DSPs. Maximum number of copies is limited by the DSP numbers, which is 1518/9 = 168 copies.

Max. throughput for a full device is calculated by multiplying number of copies and frequency, which is 168 x 260.76 MHz = 43.81 Giga computations / s.

Dynamic power of one circuit @ 42 MHz is calculated by summing the thermal power of DSP, combinational cell and register cell, which are 1.02 mW, 0.33 mW and 0.74 mW respectively.

Max. throughput / Watt for a full device is computed by first finding the dynamic power at max frequency, which is 2.09 mW x 260.76/42 = 12.98 mW. Then, the total dynamic power consumption for max copies is 168 x 12.98 mW = 2179.95 mW. Besides dynamic power, we also need to include power consumptions due to device and IO, which is 1704.5 mW and 0.17 mW. In the end, the total power consumption is estimated to be 2179.95 + 1704.5 + 0.17 = 3884.62 mW. Divide this number by the max. throughput, we can get the Max. throughput / Watt is 11.28 Giga. Computations / s Watt.

Discussion

a) What are the different sources of error? What changes could you make to reduces the error?

Error in limited Taylor terms. Taylor expansion of the exponential function produces infinite number of terms. However, our hardware circuit only implements the first 6 terms for simplicity. The error comes from the difference between infinite terms and the first 6 terms.

Quantization error. In the design, data are stored as fixed-point representation. The lim-

	Baseline Circuit	Pipelined Circuit	Shared HW Circuit
Resources for one circuit	21 ALMs + 9 DSPs	98 ALMs + 9 DSPs	38 ALMs + 2 DSPs
Operating frequency	44.8 MHz	260.76 MHz	158 MHz
Critical path	DSP mult-add + 2x DSP mult + LE-based adder + 6x DSP mult + LE-based adder	DSP mult + LE-based adder + DSP mult	2x DSP mult + LE-based adder
Cycles per valid output	1	1	5
Max. # of copies/device	168	168	759
Max. throughput for a full device (computations/s)	7.53 G.Comput/s	43.81 G.Comput/s	23.98 G.Comput/s
Dynamic power of one circuit @ 42 MHz	1.96 mW	2.09 mW	1.13 mW
Max. throughput /Watt for a full device	3.66 G.Comput/(sWatt)	11.28 G.Comput/(sWatt)	4.86 G.Comput/(sWatt)

Table 1: Implementation results of 3 different circuits

ited number of bits can not represent fraction numbers exactly. For example, the parameter A5 should be $\frac{1}{120}$, but the fixed point representation is 16'b00_00000010001000, which is $\frac{17}{2048}$.
Truncation error. The multiplication modules in the code multiply Q2.14 with Q2.14, or Q2.14 with Q7.25. The results would be Q4.28 and Q9.39. Both formats are not allowed in the design, some most-significant and least-significant bits are truncated to match the Q7.25 format. This action causes precision loss and introduces truncation error.

There are many modifications can be used to reduce the error. Here, I suggest three methods, adding more terms, using larger data bit-width and limit x-input to be small. Adding more terms can significantly reduce the differential error compares to infinite terms of Taylor expansion, this has be explained in section a) error in limited Taylor terms. Method two, using larger data bit-width reduces the quantization error and truncation error. Larger bit-width can represent fraction number more precisely and smaller error in truncating least-significant bits. Previous methods both come with increased computation, area and power costs. However, we can simply just limit the inputs to be small. From the Exact vs. Approximated plot in the lab manual, we can see the error is pretty small as long as inputs are smaller than 2. This method adds no extra cost but it's up for the design requirements whether it's allowed or not.

b) Which hardware circuits achieves highest throughput/device? Explain the reasons for the efficiency differences. From the table ??, we can see the pipelined version achieves the highest throughput per device. Pipeline uses more registers to minimize the critical path, which allows it to achieve the highest frequency among three designs. Comparing to the baseline and shared operator, pipeline maintains the same number of DSP blocks as baseline and generate 1 valid output per cycle while the shared operator needs 5 cycles. Pipeline

fully utilize the device is DSP-limited instead of ALM-limited. Although it uses the highest number of ALMs, the number of copies is not affected by ALMs. The advantages in frequency, cycles per valid output and number of copies make the pipelined design to have the highest throughput per device.

c) Explain why some circuit styles lead to average higher toggle rates. Comment on the relative efficiency of 3 circuits in terms of computations / J and explain why each style is more or less efficient.

Toggle rates:

Baseline: 12.819 million transitions/sec

Pipeline: 11.527 million transitions/sec

Shared operator: 14.810 million transitions/sec

As shown above, shared operator design has the highest toggle rate. Since shared operator uses multiple cycles to compute one valid output, the signals among the multiplier and the adder need to handle different intermediate results. This leads to the highest average toggle rate among three designs. The baseline has a slight higher toggle rate than pipeline design. This is because the baseline calculates intermediate and final results within one cycle, where the pipeline calculates intermediate and final results over five cycles. Although the signal toggle the same number of times, baseline uses less time, which leads to a higher average toggle rate than pipeline.

Among the three designs, the pipeline design has the best efficiency. It has an efficiency of 11.28 Giga computations per joule, which is roughly 3 times more baseline and 2.3 times more than shared operator. In terms of power consumption, pipeline definitely consumes the most power whether it's at 42 MHz or 260MHz. But the point is the static device power is about 1700 mW, which takes a huge portion when we calculating the total power consumption. Even if we assume the a design consumes 0 dynamic power, the difference compares to the pipeline design is approximately 2X. This disadvantage can be overcame easily by the significant performance increase. By comparing the maximum throughput of three designs, we can the pipeline has 43.81 Giga computations per second, which is about 6X more than baseline and 2X more than shared operator. The 6X performance difference between baseline and pipeline overcomes the slight power disadvantage of pipeline. On the other hand, pipeline and shared operator do have a small difference in maximum throughput, but one can find that the share operator design actually consumes more power due to it's huge number of copies.

Appendix A: Pipeline source code

```
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

// pipeline the i_valid siganl
// Since pipeline takes 9 stages, i_valid should be the same.
logic valid_p1;
logic valid_p2;
logic valid_p3;
logic valid_p4;
```

```
logic valid_p5;
logic valid_p6;
logic valid_p7;
logic valid_p8;
logic valid_p9;

// 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 pipelined registers between each mult and add block
logic [WIDTHOUT-1:0] m0_reg;
logic [WIDTHOUT-1:0] a0_reg;
logic [WIDTHOUT-1:0] m1_reg;
logic [WIDTHOUT-1:0] a1_reg;
logic [WIDTHOUT-1:0] m2_reg;
logic [WIDTHOUT-1:0] a2_reg;
logic [WIDTHOUT-1:0] m3_reg;
logic [WIDTHOUT-1:0] a3_reg;
logic [WIDTHOUT-1:0] m4_reg;

// i_x needs to be pipelined as well
// So every stage can have a consistent X value
logic [WIDTHOUT-1:0] x_0;
logic [WIDTHOUT-1:0] x_1;
logic [WIDTHOUT-1:0] x_2;
logic [WIDTHOUT-1:0] x_3;
logic [WIDTHOUT-1:0] x_4;
logic [WIDTHOUT-1:0] x_5;
```

```
logic [WIDTHOUT-1:0] x_6;
logic [WIDTHOUT-1:0] x_7;
logic [WIDTHOUT-1:0] x_8;
logic [WIDTHOUT-1:0] x_9;

// compute y value, registers are inserted between each mult and add
mult16x16 Mult0 (.i_dataa(A5),      .i_datab(x),    .o_res(m0_out));
reg32 reg0 (.reset(reset), .CLK(clk), .ena(i_ready), .D1(m0_out), .Q1(
    m0_reg), .D2(x), .Q2(x_0));
addr32p16 Addr0 (.i_dataa(m0_reg),  .i_datab(A4),   .o_res(a0_out));

reg32 reg1 (.reset(reset), .CLK(clk), .ena(i_ready), .D1(a0_out), .Q1(
    a0_reg), .D2(x_0), .Q2(x_1));

mult32x16 Mult1 (.i_dataa(a0_reg),  .i_datab(x_1),  .o_res(m1_out));
reg32 reg2 (.reset(reset), .CLK(clk), .ena(i_ready), .D1(m1_out), .Q1(
    m1_reg), .D2(x_1), .Q2(x_2));
addr32p16 Addr1 (.i_dataa(m1_reg),  .i_datab(A3),   .o_res(a1_out));

reg32 reg3 (.reset(reset), .CLK(clk), .ena(i_ready), .D1(a1_out), .Q1(
    a1_reg), .D2(x_2), .Q2(x_3));

mult32x16 Mult2 (.i_dataa(a1_reg),  .i_datab(x_3),  .o_res(m2_out));
reg32 reg4 (.reset(reset), .CLK(clk), .ena(i_ready), .D1(m2_out), .Q1(
    m2_reg), .D2(x_3), .Q2(x_4));
addr32p16 Addr2 (.i_dataa(m2_reg),  .i_datab(A2),   .o_res(a2_out));

reg32 reg5 (.reset(reset), .CLK(clk), .ena(i_ready), .D1(a2_out), .Q1(
    a2_reg), .D2(x_4), .Q2(x_5));

mult32x16 Mult3 (.i_dataa(a2_reg),  .i_datab(x_5),  .o_res(m3_out));
reg32 reg6 (.reset(reset), .CLK(clk), .ena(i_ready), .D1(m3_out), .Q1(
    m3_reg), .D2(x_5), .Q2(x_6));
addr32p16 Addr3 (.i_dataa(m3_reg),  .i_datab(A1),   .o_res(a3_out));

reg32 reg7 (.reset(reset), .CLK(clk), .ena(i_ready), .D1(a3_out), .Q1(
    a3_reg), .D2(x_6), .Q2(x_7));

mult32x16 Mult4 (.i_dataa(a3_reg),  .i_datab(x_7),  .o_res(m4_out));
reg32 reg8 (.reset(reset), .CLK(clk), .ena(i_ready), .D1(m4_out), .Q1(
    m4_reg));
addr32p16 Addr4 (.i_dataa(m4_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

always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        x <= 0;
    end else if (enable) begin
        // read in new x value
        x <= i_x;
    end
end

// pipeline the i_valid
always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        valid_p1 <= 1'b0;
        valid_p2 <= 1'b0;
        valid_p3 <= 1'b0;
        valid_p4 <= 1'b0;
        valid_p5 <= 1'b0;
        valid_p6 <= 1'b0;
        valid_p7 <= 1'b0;
        valid_p8 <= 1'b0;
        valid_p9 <= 1'b0;

    end else if(enable) begin
        valid_p1 <= i_valid;
        valid_p2 <= valid_p1;
        valid_p3 <= valid_p2;
        valid_p4 <= valid_p3;
        valid_p5 <= valid_p4;
        valid_p6 <= valid_p5;
        valid_p7 <= valid_p6;
        valid_p8 <= valid_p7;
        valid_p9 <= valid_p8;
    end
end

// Infer the registers
always_ff @(posedge clk or posedge reset) begin
```

```
    if (reset) begin
        valid_Q1 <= 1'b0;
        valid_Q2 <= 1'b0;

        y_Q <= 0;
    end else if (enable) begin
        // propagate the valid value
        valid_Q1 <= valid_p9;
        valid_Q2 <= valid_Q1;

        // output computed y value
        y_Q <= y_D;
    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

// custom register module
// Q1 is 32-bit, used for pipeline the intermedia computed values
// Q2 is 16-bit, used for passing consistent X values
// ena, enable, allowes the register to stall
module reg32(
    input reset,
    input CLK,
    input ena,

    input [31:0] D1,
    output logic [31:0] Q1,

    input [15:0] D2,
    output logic [15:0] Q2
);

    always_ff @(posedge CLK or posedge reset) begin
        if(reset) begin
            Q1 <= 0;
            Q2 <= 0;
        end else if(ena) begin
            Q1 <= D1;
            Q2 <= D2;
        end

    end
endmodule

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

Appendix B: Shared hardware source code

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

// signal for enabling sequential circuit elements
logic enable;
logic [2:0] select;

// output signal of 5to1 mux, contains A4 to A0 parameters
logic [WIDTHIN-1:0] mux_out;

// signal from register that keeps the inital x
logic [WIDTHIN-1:0] init_x;

//input signals for multiplier
logic [WIDTHIN-1:0] m1_in;
logic [WIDTHOUT-1:0] m2_in;
// output signal from multiplier
logic [WIDTHOUT-1:0] m_out;
// output signal from adder
logic [WIDTHOUT-1:0] a_out;
```

```
// ready signal generated by FSM
logic fsm_ready;

// signal for i_valid to wait 5 cycles
// since each valid output takes 5 cycles
logic valid_Q1;
logic valid_Q2;
logic valid_Q3;
logic valid_Q4;
logic valid_Q5;

// connect the FSM
fsm counter (.clk(clk), .reset(reset), .i_ready(i_ready), .sel2(enable)
    , .sel5(select), .o_ready(fsm_ready));

// create the 5to1 mux
assign mux_out = (select == 3'b000) ? A4 :
                 (select == 3'b001) ? A3 :
                 (select == 3'b010) ? A2 :
                 (select == 3'b011) ? A1 : A0;

// register holds the inital x value for next several cycles
always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        init_x <= 0;
    end else if (i_valid) begin
        // read in new x value
        init_x <= i_x;
    end
end

// 2to1 mux, choose A5 in the first cycle
// choose init_x for the rest four cycles
assign m1_in = enable ? init_x : A5;

// 2to1 mux, choose i_x in the first cycle
// choose y_Q for the the rest four cycles
assign m2_in = enable ? y_Q : {5'b00000, i_x, 11'b00000000000};

// one mult and one adder for shared HW design
mult32x16 Mult1 (.i_dataa(m2_in),   .i_datab(m1_in),   .o_res(m_out));
addr32p16 Addr1 (.i_dataa(m_out),   .i_datab(mux_out), .o_res(a_out));
```

```
//assign adder output to y register
assign y_D = a_out;


// Y register, holds final result when o_ready and o_valid are true
// holds intermediate value then o_ready and o_valid are false
always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        y_Q <= 0;
    end else if(i_ready) begin
        y_Q <= y_D;
    end
end

// register to propagate i_valid
always_ff @(posedge clk or posedge reset) begin
    if (reset) begin
        valid_Q1 <= 0;
        valid_Q2 <= 0;
        valid_Q3 <= 0;
        valid_Q4 <= 0;
        valid_Q5 <= 0;

    end else if (i_ready) begin
        valid_Q1 <= i_valid;
        valid_Q2 <= valid_Q1;
        valid_Q3 <= valid_Q2;
        valid_Q4 <= valid_Q3;
        valid_Q5 <= valid_Q4;
    end
end

// output is valid as long as the propagated i_valid and FSM generates
// the Y register contains the final result.
assign o_valid = valid_Q5 & fsm_ready;

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

endmodule

// FSM module contains 5 states. Each state generates control signals
```

```
    for mux
// and indicate final result is ready or not.
// FSM stalls when i_ready is false
module fsm (input clk, reset, i_ready, output logic sel2, output logic
    [2:0] sel5, output logic o_ready);
    enum logic [2:0] {S0=3'b000, S1=3'b001, S2=3'b010, S3=3'b011, S4=3'
        b100} state;

    always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
            state <= S0;
            sel2 <= 1;
            sel5 <= 3'b000;
            o_ready <= 1;
        end else if (i_ready) begin
            case (state)
            S0: begin
                state <= S1;
                sel2 <= 0;
                sel5 <= 3'b000;
                o_ready <= 1;
            end
            S1: begin
                state <= S2;
                sel2 <= 1;
                sel5 <= 3'b001;
                o_ready <= 0;
            end
            S2: begin
                state <= S3;
                sel2 <= 1;
                sel5 <= 3'b010;
                o_ready <= 0;
            end
            S3: begin
                state <= S4;
                sel2 <= 1;
                sel5 <= 3'b011;
                o_ready <= 0;
            end
            S4: begin
                state <= S0;
                sel2 <= 1;
                sel5 <= 3'b100;
```

```
                o_ready <= 0;
            end
            endcase
        end
    end
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

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