LSU EE 4755 Homework 5 Solution Due: 22 November 2022

For instructions visit `https://www.ece.lsu.edu/koppel/v/proc.html`. *For the complete Verilog for this assignment without visiting the lab follow* `https://www.ece.lsu.edu/koppel/v/2022/hw05.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw05.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Assignment Background

As we should know the synthesis program, given a Verilog description of a module, writes a design file with an optimized version of the module mapped to the chosen technology. For this assignment the chosen technology is the same Oklahoma University ASIC process we've been using throughout the semester.

An important skill for those writing Verilog descriptions is to estimate the cost and performance of those synthesized modules. In this assignment we'll look at how well the synthesis program handles the different modules we considered for computing the floating-point expression $v_0^2 + v_0 v_1 + v_1^2$. We will consider the combinational, sequential, and pipelined modules covered in class.

A synthesis script will be used to synthesize these modules, plus three arithmetic unit modules, plus additional modules created for the solution to this problem. To complete the assignment the output of the script must be understood and the synthesis script must be modified. The output of the synthesis script is similar to the output of the scripts used in prior assignments, so it should be familiar. Modifying the script will be something new, and might be a challenge for some of you. It is okay to seek help modifying the script from classmates and others, though the solutions to the problems themselves must be completed individually.

Modules

This assignment includes modules for the combinational, sequential, and pipelined implementations of the multi-step computation. They are named `ms_comb`, `ms_seq`, and `ms_pipe`. For comparison the assignment also includes modules containing a single floating-point unit, they are named `try_mult`, `try_add`, and `try_sq` (square).

Four additional modules are provided for experimentation, `m1_func`, `m1_comb`, `m1_seq`, and `m1_pipe`. These modules initially perform the computation $v_0 + v_0 v_1 + v_1^2$, but they can be modified to perform other computations. Module `m1_func` is used by the testbench to obtain a correct value, so modify it first so that it computes the desired computation. Then modify the others that you want to synthesize. (The synthesis program does not care whether a module passes the testbench, but no conclusion can be drawn from the area and delay of module that does not work correctly.)

All of these modules have the same parameters and ports, though not every module uses every port. For example, only `ms_seq` and `ms_pipe` are sequential so that the `clk` and `reset` ports on the others serve no function. These unused ports will be eliminated during optimization so they won't affect cost or timing.

Module Parameters and Floating Point Format

The modules used in this assignment all have the same parameters, these parameters specify the floating-point number format to be used. The first parameter, `wsig`, specifies the number of bits in the significand (fractional part) of the floating point number. The default value is 23, which is the same as an IEEE 754 single (C `float`). The second parameter, `wexp`, is the number of bits in the exponent. The default value is 8, which matches an IEEE single. The third parameter, `ieee`, specifies whether the IEEE floating-point format should be strictly followed. The default value

is 1, which means yes; a 0 means that special cases do not have to be handled correctly. These include `NaN` (not a number) and subnormal values. The size of the floating point number using these parameters is `1+wexp+wsig`, the extra `1` is for the sign bit.

For this assignment all modules are instantiated with `ieee=0`. This is done to explore the fuller range of optimization possibilities and also to reduce the time needed for synthesis.

The sample synthesis runs consider two formats, IEEE single in which `wsig=23` and `wexp=8`, and the ML-friendly BF16 (informally known as brain float) in which `wsig=7` and `wexp=8`. The advantage of BF16 for machine learning is that it is half the size of a single, and with a 7-bit significand, requires half the energy for multiplication than the older 16-bit FP16 format. For us the big advantage is that it takes less time to synthesize than a single.

Testbench

The testbench exercises the six modules, `ms_comb`, `ms_seq`, `ms_pipe`, `m1_comb`, `m1_seq`, and `m1_pipe` instantiated with a significand size of 7 and 23. They should all initially pass. As with other testbenchs in this class, a line will be printed for the first few module errors, and a tally will be provided for each module and size. The testbench uses `ms_func` to determine the correct output of the `ms` modules and `m1_func` to determine the correct output of the `m1` modules. When modifying the `m1` modules be sure to also modify `m1_func` so that the testbench can show you whether your modified modules do what you think they are doing.

The Synthesis Script

As with past assignments, the modules in the assignment file should be synthesized using the script `syn.tcl`. Unlike other assignments, this script will have to be modified.

The synthesis script itself is written in TCL (Tool Control Language, the abbreviation is pronounced tickle) a scripting language chosen by Cadence for scripting their EDA software. (Nowadays Python would be used. If it were up to me it would be Perl. But it's TCL.) Documentation for TCL can be found at `https://tmml.sourceforge.net/doc/tcl/`. This describes TCL, not the functionality needed to run Genus or other tools. For Genus-specific commands see the synthesis documentation linked to `https://www.ece.lsu.edu/koppel/v/ref.html`.

For this assignment it should not be necessary to use new Genus commands, just to change which modules are synthesized and which parameters to instantiate with. For that, one needs only a rudimentary knowledge of TCL, perhaps what can be learned just by looking at `syn.tcl`.

The synthesis script starts by setting some script variables, using the TCL `set` command, and by setting Genus attributes, using the Genus `set_db` command:

```
set verilog_source hw05.v
set syn_level "high"
set spew_file "spew.log"
set report_file "syn-report.log"
set_db syn_global_effort $syn_level
set rpt_chan [open $report_file w]
puts "Synthesizing at effort level \"$syn_level\"\n"
```

As one might guess `syn_level` is the amount of effort used for synthesis. Possible values are `none`, `low`, `medium`, and `high`. These initial lines are followed by the definition of a TCL procedure `syn_mod`, which emits the commands needed to synthesize a module, followed by commands to retrieve the area and delay of the synthesized module. A line of text is written showing the area and delay. It should not be necessary to modify `syn_mod` for this assignment.

Module `syn_mod` is called in a loop nest near the end of the file:

```
# List of combinational modules.
```

```
set mods_comb { ms_comb try_mult try_add try_sq }
set delay_targets { 100 0.1 }
set mods { try_mult try_add try_sq }
set mods { ms_comb ms_seq ms_pipe try_mult try_add try_sq }
set wsigs { 7 14 23 }

foreach delay_target $delay_targets {
    foreach ws $wsigs {
        foreach mod $mods {
            syn_mod $mod $delay_target " $ws 8 0 "
        }
    }
}
```

The loop nest above synthesizes each of the modules listed in `mods` (that's the inner loop). Each of these six modules is synthesized for each significand size found in `wsigs`. These modules are synthesized with each delay constraint in `delay_target`. For the code above there would be a total of $2 \times 6 \times 3$ synthesis runs. That would probably take hours.

The first `set` line writes variable `mods_comb` with a list of combinational modules. This variable must be updated with any new combinational modules that you use. Variable `mods` is set twice, first to a list of the arithmetic modules, then those are replaced with a list of the arithmetic modules and our multi-step modules. (Because of the second assignment the first assignment has no effect.) If one wanted to only synthesize the arithmetic modules one would comment out the second `mods` line. There is no need to use a loop nest. It is possible to write a `syn_mod` call for each synthesis, for example:

```
set delay_targets { 100 }
set wsigs { 7 14 23 }

syn_mod try_mult 5 "7 8 0"
syn_mod try_mult 5 "7 6 0"

# Exit before the loop nest.
close $rpt_chan
quit
foreach delay_target $delay_targets {}
```

The example above does two synthesis runs. The `5` is the delay target and the quoted part are the parameters. (The parameters must be quoted so that they are read as a single argument to `syn_mod`.) In the example above, `try_mult` is synthesized with two exponent sizes, 8 bits and 6 bits, both are synthesized with a delay target of 5 ns.

To synthesize a new module (for example, one you wrote) add the name to one of the `mod` lists, or just use the name on a direct call to `syn_mod` as in the example above. **Iff the module is combinational** add the module to `mods_comb`. Not adding a combinational module to `mods_comb` will result in an error. Adding a sequential module to `mods_comb` will result in incorrect timing.

Synthesis Script Output

The synthesis script `syn.tcl` is run using the command `genus -files syn.tcl`. The run starts with a substantial amount of header output, including warnings, copyright information, and system information. Some is shown below:

```
[cyc.ece.lsu.edu] % genus -files syn.tcl
```

```
2022/11/13 16:52:05 WARNING This OS does not appear to be a Cadence supported Linux configuration.
2022/11/13 16:52:05 For more info, please run CheckSysConf in <cdsRoot/tools.lnx86/bin/checkSysConf <productId>
TMPDIR is being set to /tmp/genus_temp_566634_cyc.ece.lsu.edu_koppel_nvftYI
Cadence Genus(TM) Synthesis Solution.
Copyright 2022 Cadence Design Systems, Inc. All rights reserved worldwide.
Cadence and the Cadence logo are registered trademarks and Genus is a trademark
of Cadence Design Systems, Inc. in the United States and other countries.

[16:52:12.338826] Configured Lic search path (21.01-s002): /apps/linux/cadence/share/license/license.dat:/opt/pgi/license.dat
```

The output of the script proper (as opposed to Genus, the synthesis program) starts with an announcement of the synthesis effort level followed by a table of synthesis results:

```
Synthesizing at effort level "high"

Module Name                              Area   Delay   Delay     Synth
                                              Actual  Target     Time
ms_comb_wsig7_wexp8_ieee0              600190  12.219     0.1 ns    423 s
ms_seq_wsig7_wexp8_ieee0               445400   5.754     0.1 ns    236 s
ms_pipe_wsig7_wexp8_ieee0              797327   5.678     0.1 ns    309 s
ms_comb_wsig14_wexp8_ieee0            1363980  14.391     0.1 ns    707 s
```

Each line of the table shows the result of one synthesis run. The `Module Name` column shows the name of the module followed by the parameter values used in its instantiation. In the sample above three different modules are synthesized, `ms_comb`, `ms_seq`, and `ms_pipe`. Module `ms_comb` is synthesized once with significand of 7 bits and once with a significand of 14 bits.

The `Area` *column* shows the area given by the Genus `report area` command. The units are relative to the OSU technology. *The* `Delay Actual` *column* shows the length of critical path through the module in units of nanoseconds. *The* `Delay Target` *column* shows the delay constraint that the synthesis program was set to meet. In the example above the constraint is 0.1 ns, which means the critical path can be no longer than 0.1 ns. This constraint was intentionally set to an impossibly low value, to determine the minimum delay that the synthesis program could achieve. Normally the delay constraint is set to something achievable, perhaps 4 ns in the example above, and the synthesis program would generate the least expensive design that meets the delay constraint. *The* `Synth Time` *column* shows the wall-clock (elapsed) time needed to perform the synthesis. The wall-clock time is shown to help plan the synthesis runs, it does not directly affect or describe the design itself.

Problem 1: In class we considered three ways of implementing `multi_step`, the modules that computed $v_0^2 + v_0 v_1 + v_1^2$: A combinational version, a sequential version, and a pipelined version. Appearing below are the results from synthesizing these three modules, named `ms_comb`, `ms_seq`, and `ms_pipe`, followed by results of synthesizing modules consisting only of the Chipware floating-point multiplier, adder, and a multiplier with the same value used for both operands. These are synthesized with a large delay constraint, meaning that the cost has been minimized.

Module Name	Area	Delay Actual	Delay Target	Synth Time
ms_comb_wsig23_wexp8_ieee0	1597692	75.142	100.0 ns	229 s
ms_seq_wsig23_wexp8_ieee0	945919	29.324	100.0 ns	111 s
ms_pipe_wsig23_wexp8_ieee0	1866509	28.273	100.0 ns	205 s
try_mult_wsig23_wexp8_ieee0	525991	28.231	100.0 ns	62 s
try_add_wsig23_wexp8_ieee0	339036	27.396	100.0 ns	53 s
try_sq_wsig23_wexp8_ieee0	297753	25.504	100.0 ns	38 s
ms_comb_wsig7_wexp8_ieee0	375767	34.708	100.0 ns	75 s
ms_seq_wsig7_wexp8_ieee0	275858	15.305	100.0 ns	34 s
ms_pipe_wsig7_wexp8_ieee0	526000	14.466	100.0 ns	62 s
try_mult_wsig7_wexp8_ieee0	94274	9.346	100.0 ns	13 s
try_add_wsig7_wexp8_ieee0	140221	14.196	100.0 ns	21 s
try_sq_wsig7_wexp8_ieee0	57802	6.085	100.0 ns	8 s

(*a*) Based on the data above, show the latency and throughput of each module for the 23-bit significand. It might be necessary to look at the module descriptions (Verilog code) to answer this question.

In the discussion below call the value in the `Delay Actual` column of the synthesis results table the *clock period* and let t_c denote its value. For example, for `ms_comb` with the 23-bit significand the clock period is $t_c = 75.142\,\text{ns}$. Also, let L denote latency and θ denote throughput.

Combinational Module, `ms_comb`*:* Latency: $L = t_c = 75.142\,\text{ns}$ and throughput $\theta = \frac{1}{t_c} = \frac{1}{75.142\,\text{ns}}$. The combinational module computes the entire result in one cycle and so the clock period is the latency. It can compute a new result every cycle and so the throughout is the reciprocal of the latency.

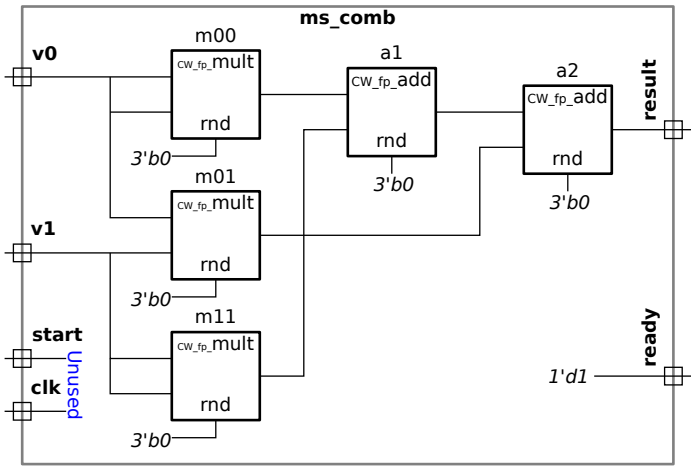
Sequential Module, `ms_seq`*:* Latency: $L = n_c t_c = 5 \times 29.324\,\text{ns} = 146.62\,\text{ns}$, where n_c is the number of cycles needed to compute a result. Throughput: $\theta = \frac{1}{n_c t_c} = \frac{1}{146.62\,\text{ns}}$. The sequential module needs five cycles ($n_c = 5$) to compute a result, so its latency is five times its clock period. Because a new computation cannot start while a computation is in progress the throughput is one over the latency.

Pipelined Module, `ms_pipe`*:* Latency $L = n_s t_c = 4 \times 28.273\,\text{ns}$, where n_s is the number of stages. Throughput $\theta = \frac{1}{t_c} = \frac{1}{28.273\,\text{ns}}$. Like the sequential circuit, the latency of the pipelined unit is the clock period times the number of cycles needed to compute a result. Unlike the sequential circuit, the pipelined circuit can start a new computation every cycle, and so the throughput is the reciprocal of the clock period.

(*b*) For each of the two significand sizes, show that the delay of the three `ms` modules are what one would expect given the delays of the three arithmetic modules.

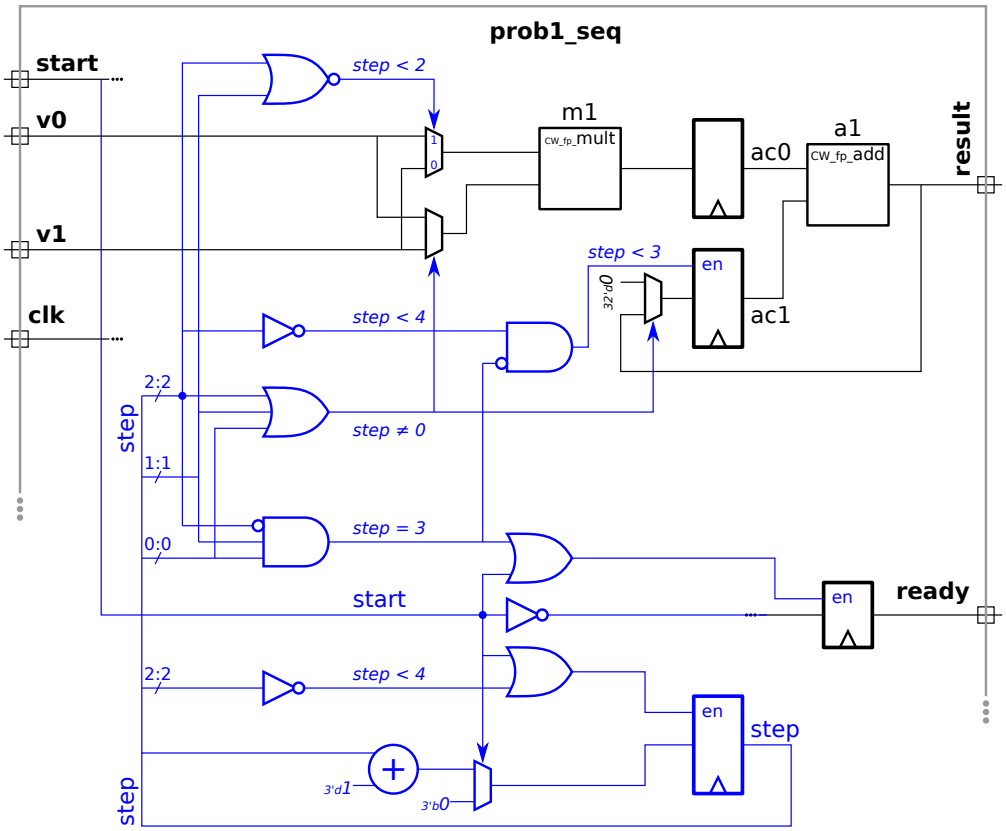
Combinational Module, `ms_comb`*:* To solve this problem one needs to find the critical path through the module. Refer to the Verilog description and the diagram of the inferred hardware below.

```
module ms_comb
  #( int wsig = 23, wexp = 8,
     int ieee = 1,
     int wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,
     output uwire ready,
     input uwire [wf-1:0] v0, v1,
     input uwire start, clk);

   localparam logic [2:0] rm = 0;
   assign ready = 1;

   uwire [7:0] mul_s1, mul_s2, mul_s3;
   uwire [7:0] a_s1, a_s2;
   uwire [wf-1:0] v00, v01, v11, s1;

   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m00( .a(v0), .b(v0), .rnd(rm), .z(v00), .status(mul_s1));
   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m01( .a(v0), .b(v1), .rnd(rm), .z(v01), .status(mul_s2));
   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m11( .a(v1), .b(v1), .rnd(rm), .z(v11), .status(mul_s3));

   CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     a1(.a(v00), .b(v11), .rnd(rm), .z(s1), .status(a_s1));
   CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     a2(.a(s1), .b(v01), .rnd(rm), .z(result), .status(a_s2));
endmodule
```



Based on the timings given in the synthesis results table the critical path goes through `m00`, `a1`, and `a2`. Both `m00` and `m11` compute the square of their inputs, and based on the data in the synthesis results table computing a square takes slightly less time than computing a product, 25.504 ns versus 28.321 ns.

Using the timings from the synthesis table, the delay (critical path) through `ms_comb` is $t_{\text{sq}} + 2t_{\text{add}} = 25.504\,\text{ns} + 2 \times 27.396\,\text{ns} = 80.296\,\text{ns}$. This is about 5 ns longer than the delay for `ms_comb` reported in the synthesis table, 75.142 ns, a difference of only about 6%.



Sequential Module, `ms_seq`*:* The inferred hardware is shown above, taken from the solution to the 2020 final exam (the module name was `prob1_seq` in the exam). For `ms_seq` the critical path cannot pass through both the mutliplier and adder, it must pass through one or the other. In addition to these arithmetic units there is also one multiplexor delay and some logic gates. Assuming that the multiplexor and logic gates' delays are small compared to the arithmetic unit, the critical path will be the larger of the two delays, $28.231\,\text{ns}$ for the multiplier and $27.396\,\text{ns}$ for the adder. So the clock period would be a bit over $28.231\,\text{ns}$. This is very close to the results from the table, $29.324\,\text{ns}$.
Pipelined Module, `ms_pipe`*:* As with the sequential module, the critical path will be through the arithmetic unit that takes the most time, the multiplier. Unlike the sequential version, there are no multiplexors or logic between the arithmetic units and the pipeline latches, and so we would expect the delay to be even closer to the multiplier delay, $28.231\,\text{ns}$. The reported delay, $28.273\,\text{ns}$ is indeed very close.

(*c*) Using the cost of the arithmetic units, show that the cost of `ms_comb` is lower than expected, but the cost of `ms_seq` and `ms_pipe` are about or perhaps a little more than what one would expect.
Combinational Module, `ms_comb`*:* This consists of one multiplier, two square units and two adders. The expected cost is $525991 + 2 \times 339036 + 2 \times 297753 = 1799569$. The reported cost is 1597692, which is lower by 11.2%.
Sequential Module, `ms_seq`*:* This module has one multipler and one adder. Their costs are $525991 + 297753 = 823744$. This estimated cost ignores the cost of registers, multiplexors, and miscellaneous logic. The reported cost is 945919, which is higher, perhaps due to the ignored hardware.
Pipelined Module, `ms_pipe`*:* This module has the same arithmetic units as the combinational module, and so the estimated cost, ignoring registers, would be the same, $525991 + 2 \times 339036 + 2 \times 297753 = 1799569$. The reported cost is 1866509 which is higher. The higher cost is probably due to ignoring the cost of registers.

Problem 2: It is welcome that the cost of `ms_comb` is lower than what one would expect based on the cost of the arithmetic units. There are several possible reasons for this, for example the synthesis program may be simplifying the two adders used in computations such as $a + b + c$ or it may be sharing hardware used to process the common b operand in expressions like $a \times b$ and $b \times c$, or perhaps it may even be transforming $v_0^2 + v_0 v_1 + v_1^2$ into $(v_0 + v_1)^2 - v_0 v_1$. Or maybe the costs for the arithmetic units shown in the table are higher than they should be.

Perform a set of synthesis runs to provide evidence for a reason that `ms_comb` cost less than its constituent parts. Consider the possible reasons given above, or one of your own. These synthesis runs can operate on one of the existing modules, a slightly modified version of the modules, or something wholly different. The modules `m1_comb`, `m1_seq`, `m1_pipe` can be used for experimentation. See the Modules section above.

Describe the results of these experiments and how they convincingly support a particular reason for the lower cost. Data from a single synthesis run, or a series of very similar runs will not be considered convincing.

The Verilog file for this assignment will be collected, but submit the answers to this question on paper or by E-mail. Please E-mail PDF files. Sending word processor source files as a final product is unprofessional, even if they are TEX files.

In your writeup:

- Indicate how you believe the synthesis program is optimizing `ms_comb`.
- Describe the modules you synthesized to come to this conclusion, and the results of synthesis. Most credit will be given for this part of the assignment.
- Explain why your experiments show that the lower cost was not due to other optimizations.

Based on the experiments described below, it appears that the synthesis program can significantly reduce the cost of computations of the form $a^2 + b^2$ computed using the ChipWare FP arithmetic modules. The optimization is not applied to similar computations such as $(a + c)^2 + b^2$.

To determine why the cost of `ms_comb` was more than 11% less than the estimated cost, a number of new modules were simulated. The modules were designed to test various hypotheses, including those suggested in the problem. The modules' names all start with `m1_`, followed by an abbreviation that may suggest what it does. (In the table of synthesis results the name is appended with the parameter values used.) For example, `m1_a3` is a module that computes $a+b+c$. Each module was tested for correctness by updating `m1_functional` so that it computes the same value as the test module. A wrapper module, `m1_comb`, provides a third input for modules that take three data inputs, such as `m1_a3`. The third input value is just `v0*v1`, so `m1_functional` uses `v0*v1` in places where `v2` might go. All of the tested modules were combinational. The synthesis script output shown below (near the end of the solution) is for runs using these modules.

The Verilog code used for these experiments can be found at
`https://www.ece.lsu.edu/koppel/v/2022/hw05-sol.v.html` and the synthesis script is at
`https://www.ece.lsu.edu/koppel/v/2022/syn-sol.tcl.html`.

Appearing below (on the next page) is the `m1_a3` module, its wrapper, and `m1_functional`.

```
 /// This module is synthesized.
module m1_a3
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,       output uwire ready,
     input uwire [wf-1:0] v0, v1, v2,    input uwire start, clk);

   localparam logic [2:0] rm = 0; // Rounding Mode

   uwire [7:0] mul_s1, mul_s2, mul_s3, a_s1, a_s2;
   uwire [wf-1:0] v00, v01, v11, s1;

   CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     a1(.a(v0), .b(v1), .rnd(rm), .z(s1), .status(a_s1));
   CW_fp_add #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     a2(.a(s1), .b(v2), .rnd(rm), .z(result), .status(a_s2));

   assign ready = 1;
endmodule

 /// This module is simulated
module m1_comb
  #( int wsig = 23, wexp = 8, ieee = 1, wf = 1 + wexp + wsig )
   ( output uwire [wf-1:0] result,    output uwire ready,
     input uwire [wf-1:0] v0, v1,     input uwire start, clk);

   localparam logic [2:0] rm = 0; // Rounding Mode
   uwire [wf-1:0]  v01;
   uwire [7:0] mul_s2;

   // Generate a third input for m1_a3.
   CW_fp_mult #( .sig_width(wsig), .exp_width(wexp), .ieee_compliance(ieee) )
     m01( .a(v0), .b(v1), .rnd(rm), .z(v01), .status(mul_s2));

   m1_a3 #( .wsig(wsig), .wexp(wexp), .ieee(ieee) )
   a3( result, ready, v0, v1, v01, start, clk );
endmodule

// cadence translate_off
module m1_functional
  ( output real mag, input real v0, v1 );
   // The testbench uses this module to test the others, so set
   // the computation to match the others.
   localparam string name = "A3 Func";
   // Note: The third value is v0*v1.
   always_comb mag = v0 + v1 + v0 * v1;
endmodule
// cadence translate_on
```

The results of each experiment are described below. The value inputs to the modules are called v_0, v_1, and v_2. The original multi-step modules only had two data inputs, v_0 and v_1. The third input, v_2, is set to $v_0 v_1$ by the wrapper module, `m1_comb`, for testing purposes. The synthesis program operates on modules such as `m1_a3` and so to it the value on third input, v_2, is unrelated to the other two values.

In the discussion below let c_a, c_m, and c_s denote the cost of the adder, multiplier, and square unit. Those costs are $c_m = 525991$, $c_a = 339036$, and $c_s = 297753$ for the 23-bit significand and $c_m = 94274$, $c_a = 140221$, and $c_s = 57802$ for the 7-bit significand.

Module `m1_a3`*: Computes* $v_0 + v_1 + v_2$

To test for any benefit of computing $a + b + c$ use a module that computes this sum, `m1_a3`. The expected cost is two adders, $3c_a$, which is 280442 for 7 bits and 678072 for 23 bits. The synthesized costs are 278532 and 668518, respectively or .68% and 1.41% lower than estimated. So there is not much optimization benefit from combining two adders.

Module `m1_mad`*: Computes* $v_0 * v_1 + v_2$

To test whether adder and multiplier hardware is shared, try a module that computes $v_0 \times v_1 + v - 2$, called `m1_mad`. The expected cost is $c_m + c + a$ or 234495 for 7 bits and 865027 for 23 bits. The synthesized hardware is just 2.58% and 1.48% less costly than the estimate, not enough to explain `ms_comb`.

Module `m1_mm`*: Computes* $v_0 * v_1$ *and* $v_0 * v_2$

Perhaps two multipliers that have a common multiplier can share some hardware. Module `m1_mm` tests that by using v_0 in both multiplies. This module has two outputs, one for each product. So the estimated cost is $2c_m$: 188548 and 1051982 for the 7- and 23-bit versions. The synthesized cost is just 1.76% and .59% less than the estimates.

Module `m1_comb_v3`*: Computes* $v_0^2 + v_0 v_2 + v_1^2$

To rule out whether the cost reduction is due to an algebraic transformation, a version of `ms_comb` which has three value inputs was tried. The new value, v_2, replaces v_1 in the $v_0 v_1$ term. The estimated cost is $2c_s + c_m + 2c_a$, the same as the `ms_comb` estimate. The synthesized costs are 22.24% and 10.79% lower than the estimated costs, which means that the synthesis program is not doing an algebraic transformation that depends on the middle term, $v_0 v_1$, sharing a variable with the other two.

Module `m1_comb_sos`*: Computes* $v_0^2 + v_1^2$

Perhaps there's something special about a sum of squares. The estimated cost is $2c_s + c_a$, or 255825 and 934542 for the 7- and 23-bit versions. The synthesized costs are substantially lower, 38.3% and 8.61%. The fact that the benefit is larger for the smaller significand suggests that the savings is with the handling of the exponents, which are eight bits in both versions.

Module `m1_comb_sop`*: Computes* $v_0 v_2 + v_1 v_3$

Are squares special? To rule that out a module computing a sum of two products was tried. This module has four value inputs. The estimated cost is $2c_m + c_1$ or 328769 and 1391018. The synthesized costs are 1.99% and 1.58% less, suggesting that there is something special about a sum of squares.

Module `m1_comb_ssp`*: Computes* $v_0^2 + v_1 v_2$

Perhaps one square can be optimized, `m1_comb_ssp` tests that. The expected cost is $c_m + c_s + c + a$ or 292297 and 1162780. The synthesized costs are 7.48% and 2.31% less, so there is some benefit to one square, but not nearly as much as the benefit from both adder inputs being squares.

Module `m1_comb_alt`*: Computes* $(v_0^2 + v_0 v_1) + v_1^2$

Finally, just to be sure, re-do `ms_comb` so the two squares are not operands of the same adder. The expected cost is $2c_s + c_m + 2c_a$ or 441037 and 1726894. The synthesized costs are lower, 10.05% and 4.04%, suggesting that there is some benefit of using a square input to an adder, but that the benefit is substantially larger when both inputs are a square.

Synthesis Data on Next Page

Module Name	Area	Delay Actual	Delay Target	Synth Time
m1_a3_wsig7_wexp8_ieee0	278532	28.162	100.0 ns	54 s
m1_a3_wsig23_wexp8_ieee0	668518	54.005	100.0 ns	117 s
m1_mad_wsig7_wexp8_ieee0	228453	23.283	100.0 ns	36 s
m1_mad_wsig23_wexp8_ieee0	852191	53.147	100.0 ns	114 s
m1_mm_wsig7_wexp8_ieee0	185236	9.346	100.0 ns	20 s
m1_mm_wsig23_wexp8_ieee0	1045808	28.231	100.0 ns	80 s
m1_comb_v3_wsig7_wexp8_ieee0	381271	34.077	100.0 ns	79 s
m1_comb_v3_wsig23_wexp8_ieee0	1605466	74.753	100.0 ns	276 s
m1_comb_sos_wsig7_wexp8_ieee0	157807	19.544	100.0 ns	34 s
m1_comb_sos_wsig23_wexp8_ieee0	854120	48.375	100.0 ns	128 s
m1_comb_sop_wsig7_wexp8_ieee0	322223	23.763	100.0 ns	48 s
m1_comb_sop_wsig23_wexp8_ieee0	1369003	53.049	100.0 ns	169 s
m1_comb_ssp_wsig7_wexp8_ieee0	270427	23.781	100.0 ns	44 s
m1_comb_ssp_wsig23_wexp8_ieee0	1135920	53.306	100.0 ns	152 s
m1_comb_alt_wsig7_wexp8_ieee0	441037	38.112	100.0 ns	103 s
m1_comb_alt_wsig23_wexp8_ieee0	1726894	80.177	100.0 ns	281 s