Name Solution

Digital Design using HDLs

EE 4755

Final Examination

Thursday, 8 December 2016 12:30-14:30 CST

Problem 1		(30 pts)
Problem 2		(20 pts)
Problem 3		(15 pts)
Problem 4		(15 pts)
Problem 5		(10 pts)
Problem 6		(10 pts)
Exam Total		(100 pts)

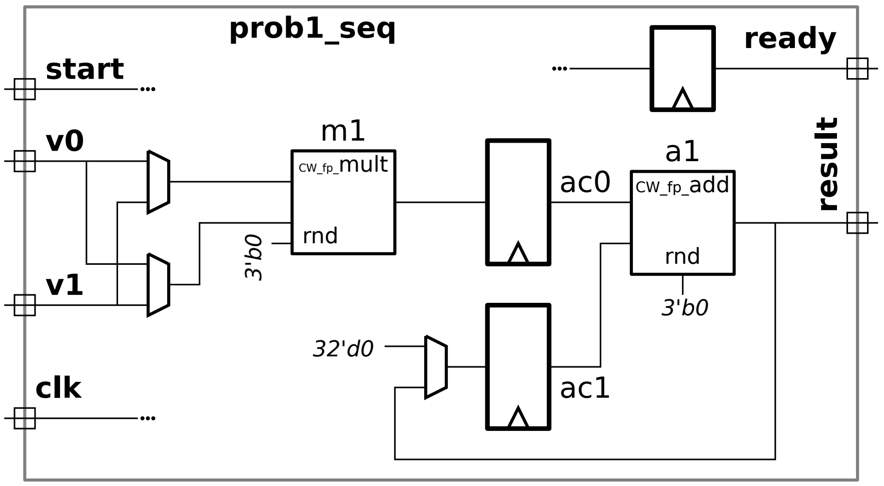
Alias The Hottest Place in Hell

Good Luck!

Problem 1: [30 pts] The diagram and Verilog code below show incomplete versions of module `prob1_seq`. This module is to operate something like `mag_seq` from Homework 6. When `start` is `1` at a positive clock edge the module will set `ready` to `0` and start computing `v0*v0 + v0*v1 + v1*v1`, where `v0` and `v1` are each IEEE 754 FP single values. The module will set `ready` to `1` at the first positive edge after the result is ready.

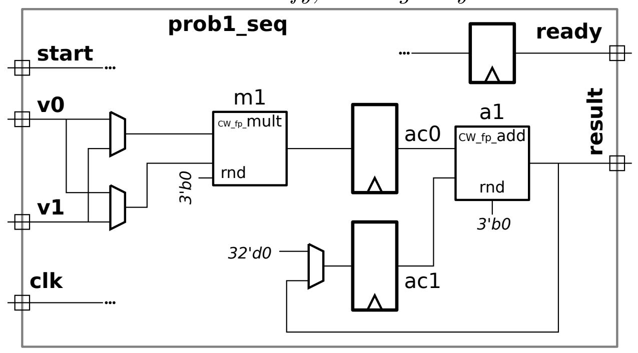
Complete the Verilog code so that the module works as indicated and is consistent with the diagram. It is okay to change declarations from, say, `logic` to `uwire`. But the synthesized hardware cannot change what is already on the diagram, for example, don't remove a register such as `ac0` and don't insert any new registers in existing wires, such as those between the multiplier inputs and the multiplexors.

Don't modify this diagram, write Verilog code.



Don't modify this diagram, write Verilog code.

Don't modify, Verilog only.

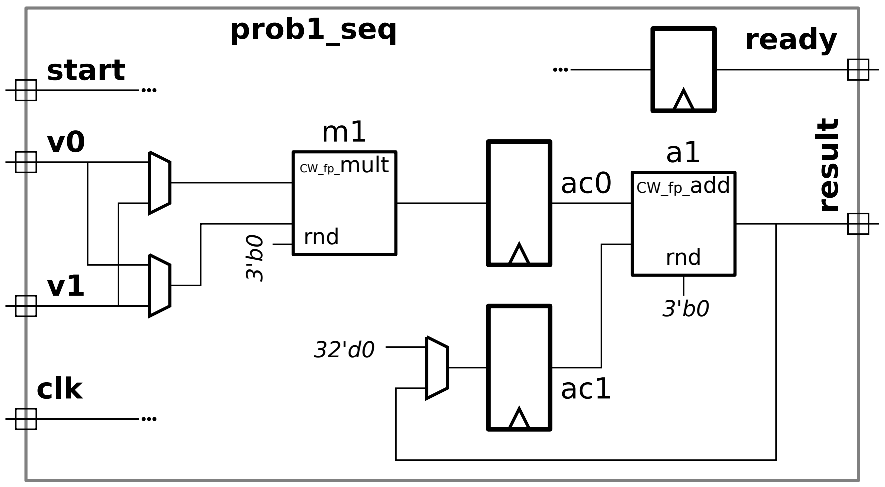


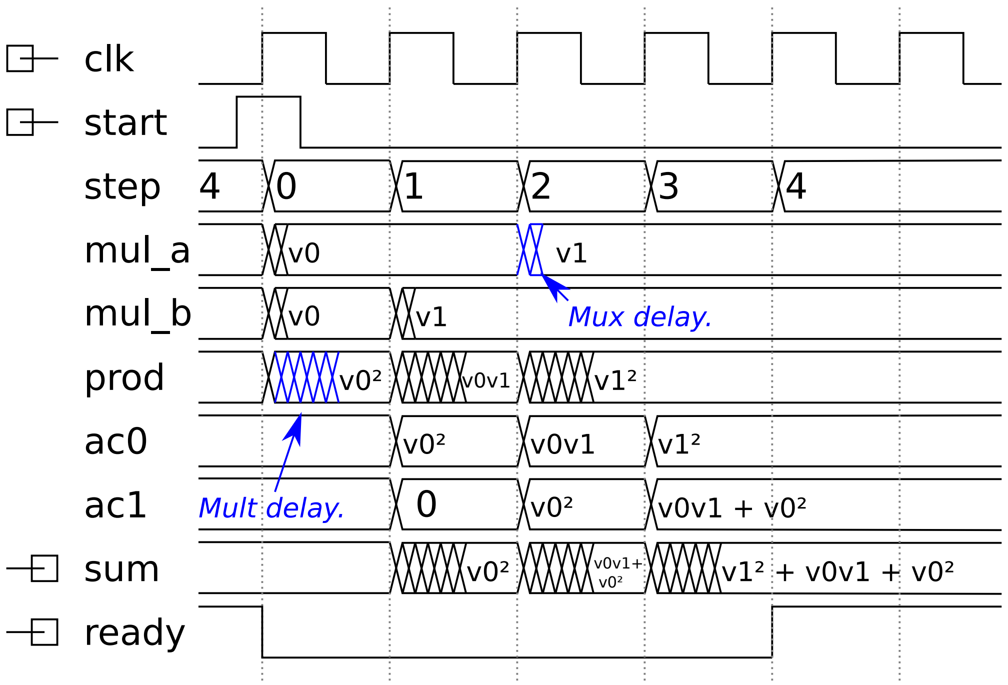
Problem 1, continued: Solution on this page.

- ✓ Complete Verilog so that module computes `v0*v0 + v0*v1 + v1*v1`.
- ✓ Synthesized hardware must be consistent with diagram, ✓ especially synthesized registers.
- ✓ Note that `ready` must come from a register.
- ✓ Don't skip the easy part: connections to adder.

```
module prob1_seq( output uwire [31:0] result,   output logic ready,
                  input uwire [31:0] v0, v1,    input uwire start, clk);
   uwire [7:0] mul_s, add_s;
   uwire [31:0] mul_a, mul_b;     uwire [31:0] add_a, add_b;    uwire [31:0]  prod, sum;
   logic [31:0] ac0, ac1;         logic [2:0]  step;

   localparam   int last_step = 4;  // ⟵  SOLUTION.

   always_ff @( posedge clk ) if      ( start )             step <= 0;
                              else if ( step < last_step ) step <= step + 1;

   CW_fp_mult m1( .a(mul_a), .b(mul_b), .rnd(rnd), .z(prod), .status(mul_s));
   CW_fp_add  a1( .a(add_a), .b(add_b), .rnd(rnd), .z(sum),  .status(add_s));

   // assign ready = step == last_step; // SOLUTION: Remove this line.

   // SOLUTION (remainder of module is solution)

   assign mul_a = step < 2  ? v0 : v1;   // Connect FP multiplier ports ..
   assign mul_b = step == 0 ? v0 : v1;   // .. to appropriate values.
   assign add_a = ac0,   add_b = ac1;    // Connect FP adder input ports.

   always_ff @( posedge clk ) begin      // Assign registers ac0, ac1, and ready.
        ac0 <= prod;                     // Always write ac0.

        case ( step )                    // Set ac1 based on the step value ..
           0: ac1 <= 0;                  // .. *before* the positive clk edge.
           1, 2: ac1 <= sum;
        endcase

        if ( start )         ready <= 0;  // Reset ready *before* step 0 ..
        else if ( step == last_step-1 ) ready <= 1;  // .. and set ready when will be done.
     end

   assign result = sum;                  // Connect FP adder output to this module's output.

endmodule
```

To understand how the solution works refer to the timing diagram below. Note that the value of `step` in the second `always_ff` is *before* it is incremented.





Problem 2: [20 pts] Analyze the timing of the two similar modules on the next page using the timing model used in class, as requested in the subproblems. Assume that **all adders are synthesized as a ripple connection of binary full adders** and that the **comparison units are also based on ripple hardware**.

(*a*) Before analyzing the modules, show the delay of each of the components listed below using the simple model given in class. For this part assume that all inputs are available at $t = 0$.

In the simple timing model the delay of an n-input AND and OR gate is $\lceil \lg n \rceil$ units, which works out to 1 for 2-input gates. For larger gates the delay is what would be obtained by constructing a reduction tree of 2-input gates. NOT gates have a delay of zero. Other combinational logic is based on the delay of an implementation using AND, OR, and NOT gates. For example, the delay of a 2-input XOR gate is 2.

✓ Delay for BFA is:

The delay (of `bfa-unopt`) is 4 units for the `sum` and 3 units for `co`.

When the `a` and `b` inputs arrive at least two cycles before `ci`, the delay of `bfa-fast` is 2 units from input `ci` to the `sum` and `co` outputs. If `a` and `b` arrive at the same time as `ci` then the delay of `bfa-fast` is the same as `bfa-unopt`: 4 time units.

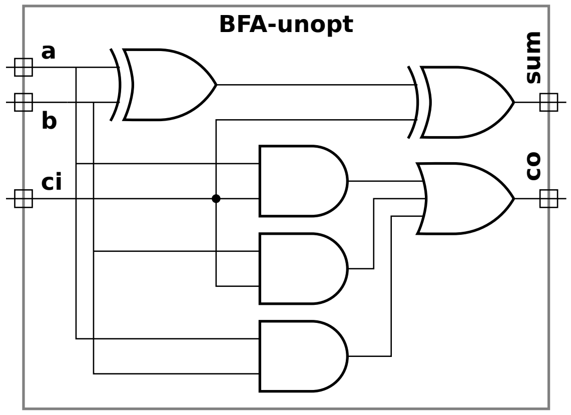
✓ Explain or show diagram.

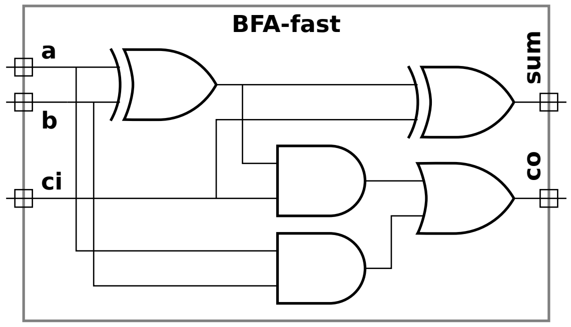
Short Answer: The delay is based on the `bfa-unopt` BFA implementation below. The lower diagram shows the timing analysis.

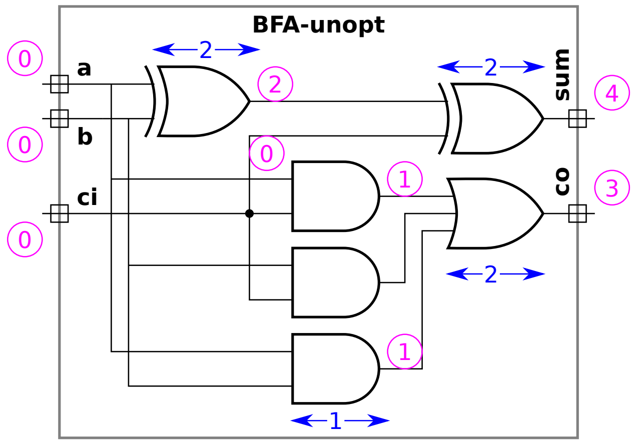
Long Answer: Appearing below are two implementations of a binary full adder, shown with and without a timing analysis. In the first, `bfa-unopt`, separate logic is used to generate the sum and carry out signals. In the second, `bfa-fast`, an XOR gate is shared by the sum and carry out logic, both reducing cost and reducing the critical path in a ripple adder.

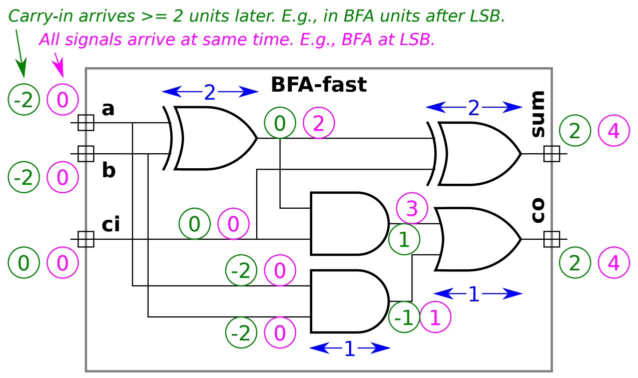
The blue labels show the gate delays, circled numbers show the time that a signal is available. The purple signals show the timing under the assumption that all signals arrive at module inputs $t = 0$ and the green signals show the timing under the assumption that the carry in signal arrives at $t = 0$ but that the `a` and `b` signals arrive at $t = -2$. The rationale for the green signals assumption is that when a BFA is used as part of a ripple adder the carry in signal for all but the least-significant bit BFA will arrive later than the `a` and `b` inputs. Note that even with the `a` and `b` signals arriving early the delay for `co` in `BFA-unopt` would still be 3.

In summary: For `BFA-unopt`, sum at $t = 4$ and carry out at $t = 3$. For `BFA-fast` with all signals arriving at $t = 0$, the sum is available at $t = 4$ and carry out at $t = 4$. With early arrival, the carry out is available at $t = 2$.







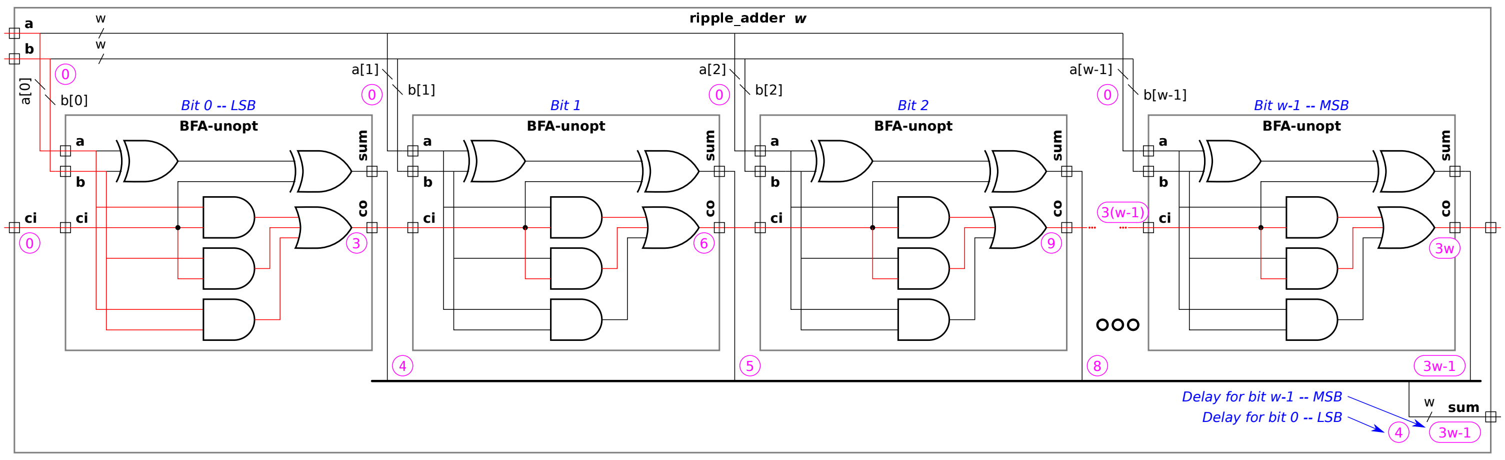

Carry-in arrives >= 2 units later. E.g., in BFA units after LSB.
All signals arrive at same time. E.g., BFA at LSB.
BFA-fast
a
b
ci
sum
co
2
1

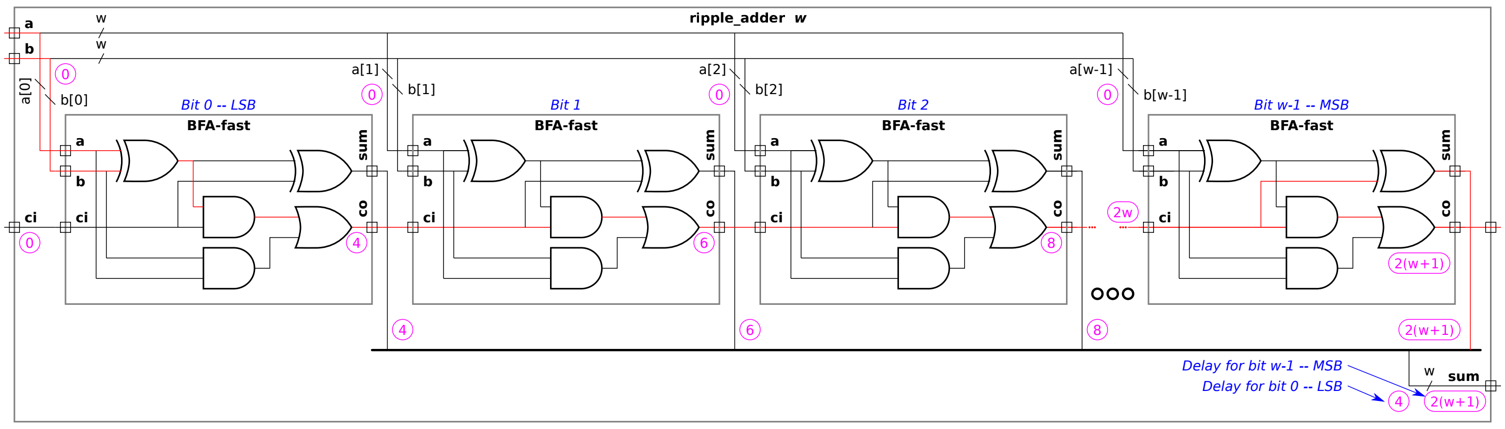
☑ Delay for a w-bit adder is:

Using `BFA-unopt` the delay for $w > 1$ is $3w$ units for the carry out and $3w - 1$ units for the MSB of the sum.

Using `BFA-fast` the delay for $w > 1$ is $2(w + 1)$ units for the carry out and the MSB of the sum. The LSB (bit 0) of the sum arrives at time 4, bit 1 arrives at 6, and bit i of the sum arrives at time $4 + 2i$.

☑ Explain or show diagram.





Short Answer: As shown in red in the diagram above, the critical path passes from `ci` to `co` of the linearly-connected BFAs, so the total delay is $3w$ using `BFA-unopt` or $2w + 2$ using `BFA-fast`.

When `BFA-unopt` is used the `co` signal of the BFA for bit i is available at $3(i + 1)$, where $i = 0$ is the least-significant bit. If the w-bit adder itself has a carry-out signal, the delay is $3w$ bits, based on the availability of the carry out at bit $w - 1$. If there is no carry out then the delay of the MSB is two units after the arrival of the carry in, so the total delay is $3(w - 1) + 2 = 3w - 1$ units. Bit $i = 0$, the LSB, of the sum is ready at $t = 4$, bit $i \geq 1$ of the sum is ready at $3i + 2$.

When `BFA-fast` is used the `co` signal is available at time 4 for the LSB, for bit i it is available at $4 + 2i$. The sum is available 2 cycles after the `ci` arrival, so overall timing is $4 + 2(w - 1) = 2w + 2$ whether or not a carry out is used.

☑ Delay for a w-bit $<$ (less than) comparison unit is:

Using subtraction, $3w$ units based on `BFA-unopt` or $2(w+1)$ units using `BFA-fast`.

☑ Explain or show diagram.

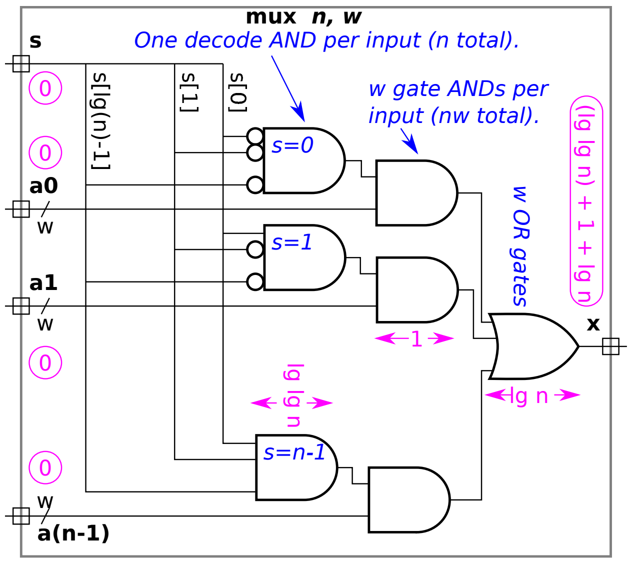
To compute $a < b$ use a w-bit adder to compute $a - b$, where a and b are w-bit unsigned numbers. (To perform subtraction the `b` inputs are inverted and the adder carry in is set to 1. This doesn't affect cost or delay under the simple model since NOT gates are free and zero-delay.) If the carry out is zero then the difference is negative and so $a < b$ is true. Note that logic that is only used for computing `sum` is omitted.

Using `BFA-unopt` the delay is $3w$, using `BFA-fast` the delay is $2w$. (In both cases the cost is $5w$. In `BFA-unopt` both XOR gates are eliminated. In `BFA-fast` one XOR gate is eliminated and the other is replaced with an OR gate.)

☑ Delay for a w-bit, n-input multiplexor is:

The delay is $\lceil \lg \lceil \lg n \rceil \rceil + 1 + \lceil \lg n \rceil$ units.

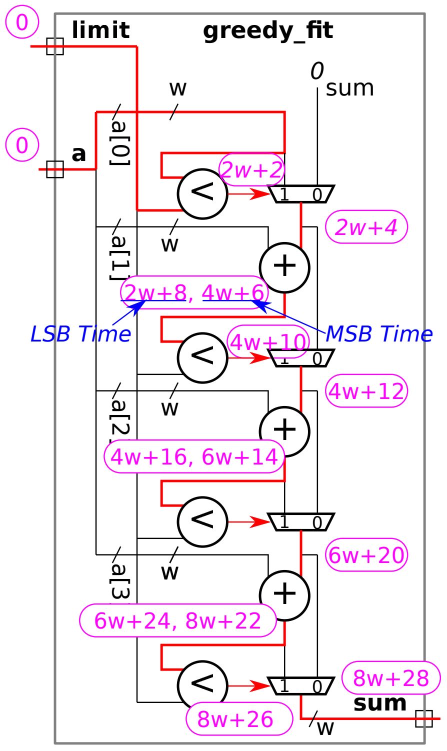
☑ Explain or show diagram.

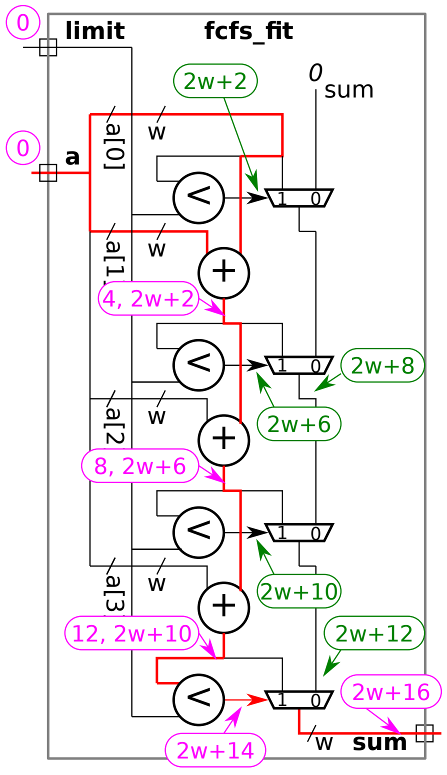


As shown in the illustration above each mux input has an AND gate decoding the select signal (with some inputs inverted based on the mux input number). The AND gate has $\lceil \lg n \rceil$ inputs (which is the number of bits in the select signal) and so by the simple model has delay of $\lceil \lg \lceil \lg n \rceil \rceil$ units. (For brevity the diagram omits the ceiling function.) The decoder and input connect to another AND gate, adding 1 to the delay. Finally, there is an n-input OR gate, contributing another $\lceil \lg n \rceil$ units of delay.

Problem 2, continued:

(*b*) Find the length of critical path in the two modules below using the timings above. **Where applicable** make the reasonable assumption that a ripple adder can start when its lower bits arrive, not when all bits of its input are stable.





☑ Length of critical path for `greedy_fit` **in terms of** w. ☑ Show work for partial credit.

Note: Around 9 October 2019 the solution was changed from a circuit using the unoptimized BFA to one using the fast BFA.

Short Answer: Using `bfa-fast` the critical path length is $8w + 28$ units, see the diagram above in which the critical path is shown in red and is labeled with timing along the critical path.

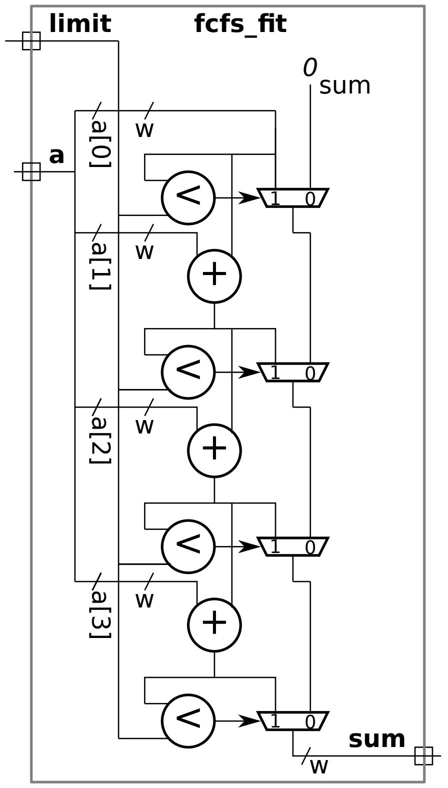
Long Answer: The time for the comparison unit is $2w + 2$ units. The time for an adder is 4 units to produce the LSB of the sum and $2w + 2$ for the complete sum, including the carry out. Note that because the comparison unit is implemented using the carry path of an adder, it can start on the LSB as soon as it arrives (meaning before more significant bits arrive), and it can keep up with the pace of a new input bit being ready every 2 time units after the 2nd bit. Signals arrive at the inputs to the `a[1]` adder at time $2w + 4$ and so the comparison gets its LSB at time $2w + 8$, the next bits at $2w + 10$, $2w + 12$, $2w + 14$, ... The comparison unit's output is ready at $2w + 8 + 2w + 2 = 4w + 10$. Unfortunately for the `a[2]` adder one of its inputs is the output of a multiplexor, meaning that all bits arrive at the same time and so it cannot start early.

☑ Length of critical path for `fcfs_fit` **in terms of** w. ☑ Show work for partial credit.

Short Answer: Using `bfa-fast` the critical path length is $2w + 16$ units, see the diagram above in which the critical path is shown in red and is labeled with timing along the critical path. Green shows timing of non-critical signals.

Long Answer: Unlike `greedy_fit` the inputs to the adders in `fcfc_fit` are either a module input or the output of another adder. For that reason the second adder can start working on the LSB after only 4 units, and the third starts after 8 units. For this reason the total delay through the 3 adders is $2 \times 4 + 2w + 2 = 2w + 10$. The remainder of the critical path passes through a comparison and a mux. The path through the first two multiplexors is almost critical, it just has two units of slack.

Problem 3: [15 pts] Complete the Verilog code so that it corresponds to the module shown.

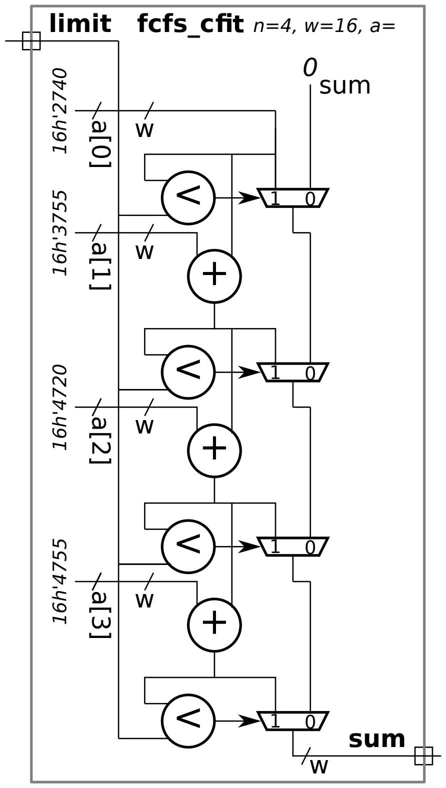


☑ Complete module.

The solution appears below. The module would be slightly simpler if the `if ( i > 0 )` were removed (making the `rsum+=a[i+1]` unconditional) and `rsum` were initialized to zero, but that would not *exactly* correspond to the illustration. Full credit would be given to either solution.

```
module fcfs_fit #( int nelts = 4, int w = 16 )
   ( output logic [w-1:0] sum,
     input uwire [w-1:0] a[nelts], limit );

   // SOLUTION

   always_comb begin

      logic [w-1:0] rsum;  // Running sum.
      rsum = a[0];
      sum = 0;

      for ( int i=0; i<nelts; i++ ) begin
         if ( i > 0 ) rsum += a[i+1];
         if ( rsum < limit ) sum = rsum;
      end

   end

endmodule
```

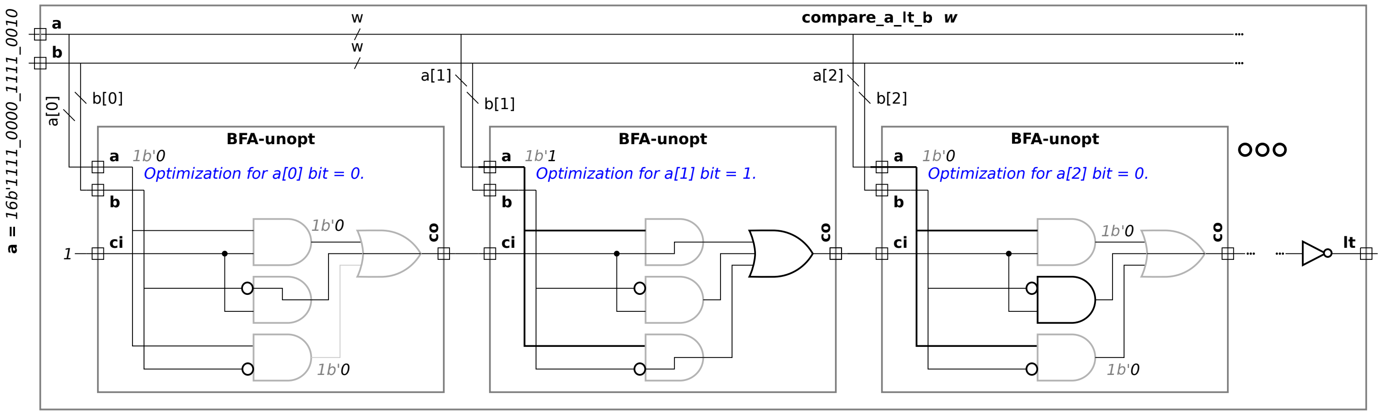
Problem 4: [15 pts] Appearing to the right is `fcfs_cfit`, a version of the `fcfs_fit` module in which the `a` input has been changed to a parameter, **meaning that `a` is an elaboration-time constant**. Compute the cost of this module using the simple model used in class and **accounting for optimization based on the constant values**. As in an earlier problem, adders and comparision units are ripple-style.



☑ Cost of the `a[0]` comparison unit.

The cost is $w - 1$ gates.

☑ Explain.



a = 16b'1111_0000_1111_0010
compare_a_lt_b w
a
b
w
w
a[0]
b[0]
a[1]
b[1]
a[2]
b[2]
BFA-unopt
Optimization for a[0] bit = 0.
Optimization for a[1] bit = 1.
Optimization for a[2] bit = 0.
1b'0
1b'1
ci
co
1
lt

Because one input is a constant the 5 gates per bit using **BFA-unopt** is reduced to 1, either an AND gate or an OR gate. (If **BFA-fast** is used then the 4 gates per bit is also reduced to 1, either an AND or OR.) See the diagram above. The least significant bit requires at most a NOT gate, which has a cost of zero. The total cost is $w - 1$ gates.

☑ Cost of the `a[1]` adder.

☑ Explain.

Since both inputs are constant the cost is zero.

☑ Cost of the `a[0]` multiplexor.

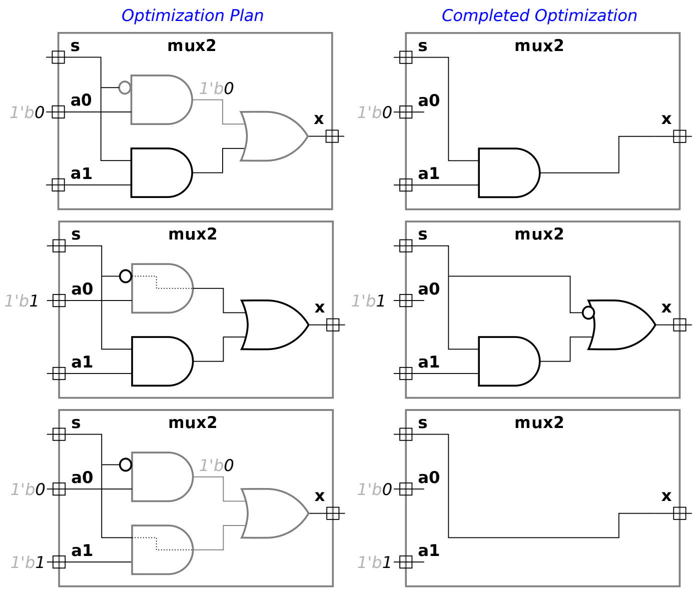
☑ Explain.

Since both inputs are constant the cost is zero. The output for bit $0 \leq i < 16$ is either the constant zero or, where an `a` bit is `1`, is equal to the select signal. See the bottom row of the 2-input mux optimization diagram to the right.

☑ Cost of the `a[2]` multiplexor.

☑ Explain.

One input to the `a[2]` mux is constant. Where the constant input bit is 0 the logic is just an AND gate (top row of diagram) where it is 1 the logic is an AND and an OR (middle row of diagram).



☐ Total cost.

Problem 5: [10 pts] Answer each question below.

(*a*) A time slot in the Verilog event queue contains many regions, among them *active*, *inactive*, and *NBA*.

Explain how an event gets put in each region. (You can use the next subproblem for examples.)

☑ An event is put into the active region when:

Short Answer: When the active region is empty.

Explanation: All events in the first non-empty region are copied into the active region. Of the regions that have been mentioned, the inactive region in the current time slot is checked first, followed by the NBA region in the current time slot, followed by the inactive region in the next scheduled time slot, etc.

☑ An event is put into the inactive region when:

Short Answer: ... when a delay such as `#1` is encountered in procedural code and when a variable found in a sensitivity list changes.

Explanation: When a delay such as `#d` (for $d \geq 0$) is encountered in procedural code an resume event will be scheduled in the inactive region of time step $t + d$, where t is the current time slot. Note that `#0` is perfectly okay for those who understand SystemVerilog event timing. For example, in `... b=z+w; #0; c=q+r; ...` when the `#0` is reached a resume event will be put in the inactive region of the current time slot to resume execution at the `c=` statement. If the delay had been `#3` then the resume event would be put in the inactive region of time slot $t + 3$.

For those events with a sensitivity list, such as **`always_comb`**, continuous assignments, and module instantiations, events are scheduled when a variable on the sensitivity list changes. For example, consider **`always_comb begin`** `a=x+y;...`. When `x` changes an event to execute the **`always_comb`** block will be put in the inactive region.

☑ An event is put into the NBA region when:

Short Answer: ... when a non-blocking assignment is executed.

Explanation: For example, when execution reaches a statement like `a<=b+1` the left-hand side, `b+1` in the example, will immediately be computed and then an *update* `a` event will be placed in the NBA region. The update event carries the value of `b+1`. Eventually the NBA region will be copied to the active region and the *update* `a` event will be executed, causing `a` to change to the carried value.

(*b*) In the code fragment below show the order in which the statements are executed after the `posedge clk`. Identify a statement by the value that is assigned. The first two statements executed are `a` and `b`, that's shown. (Since `a` is a nonblocking assignment, the execution of `a` only means that `a+1` was computed, it doesn't mean that `a` was changed.) Complete the "Order of statements" list.

```
module regions;
   always_ff @( posedge clk ) begin
        a <= a + 1;
        b = b + 1;
    end

   always_comb s = a + b;
   always_comb ax = a + 2;
   always_comb ay = ax + 5;
   always_comb by = bx + 4;
   always_comb bx = b + 3;

endmodule
```

☑ Order of statements: a, b,

Short answer: a, b, s, bx, by, s, ax, ay.

Problem 6: [10 pts] Appearing below is the pipelined mag module from Homework 6.

(*a*) Suppose it turns out that the multiply (`CW_fp_mult`) takes twice as long as the add (`CW_fp_add`). Based on this fact, modify the pipeline to reduce cost, but without affecting clock frequency. Draw in your changes, there's no need to write Verilog. Also, comment on latency and throughput changes.

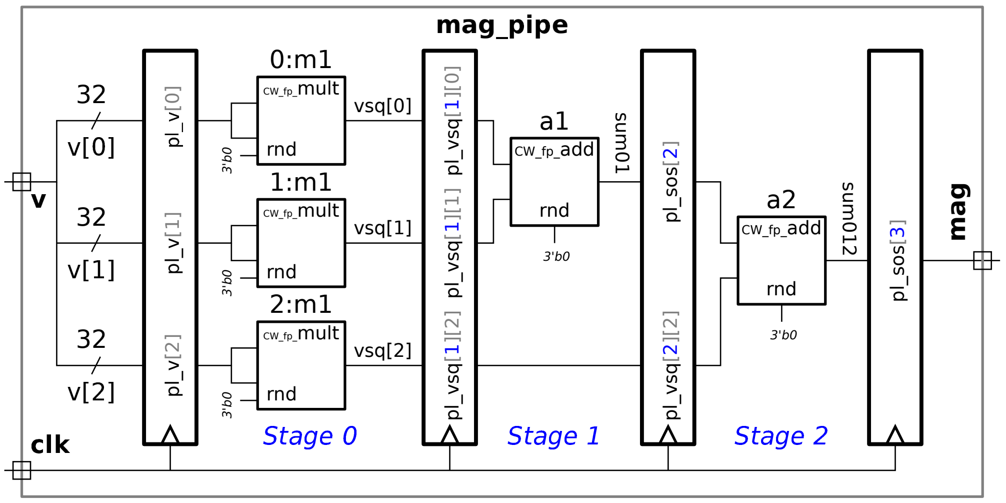
☑ Modify for lower cost based on faster adder.

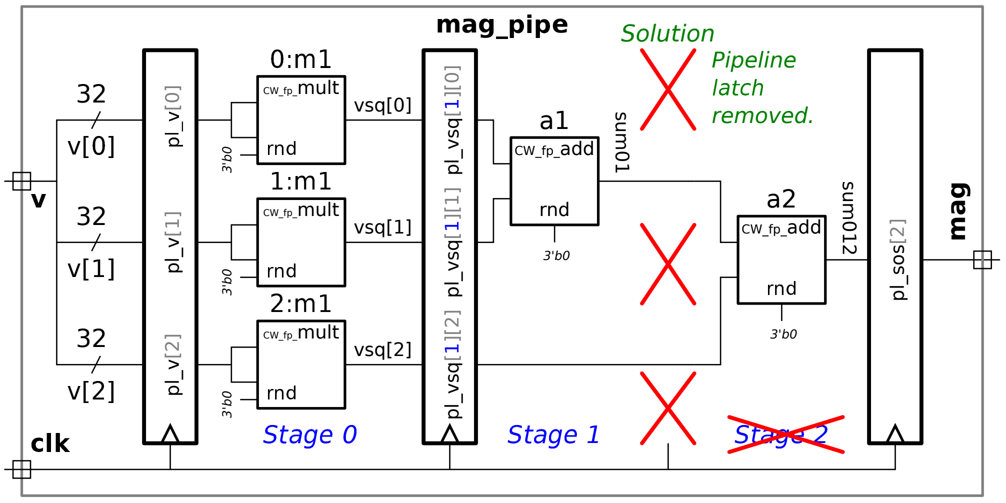
Solution appears below after the unmodified module. The pipeline latch between the two adders was removed. With this change the critical path length in the multiplier stage matches the critical path in the adder stage. (Before this change the critical path length in each of the adder stages was half the critical path in the multiplier stage.)

☑ Does the change ☑ help throughput? Does it help ☑ latency?

The change does not change throughput because the clock frequency does not change. The module can complete one calculation per cycle with or without the change.

The change reduces (helps) latency because there is one less stage.





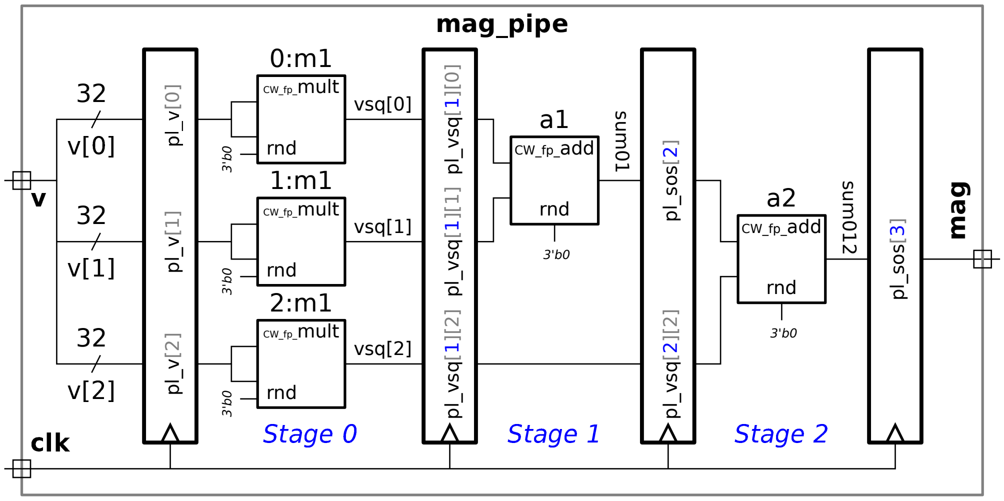
(*b*) Suppose that the `v` input arrives very early in the clock cycle. Based on this modify the pipeline to reduce cost.

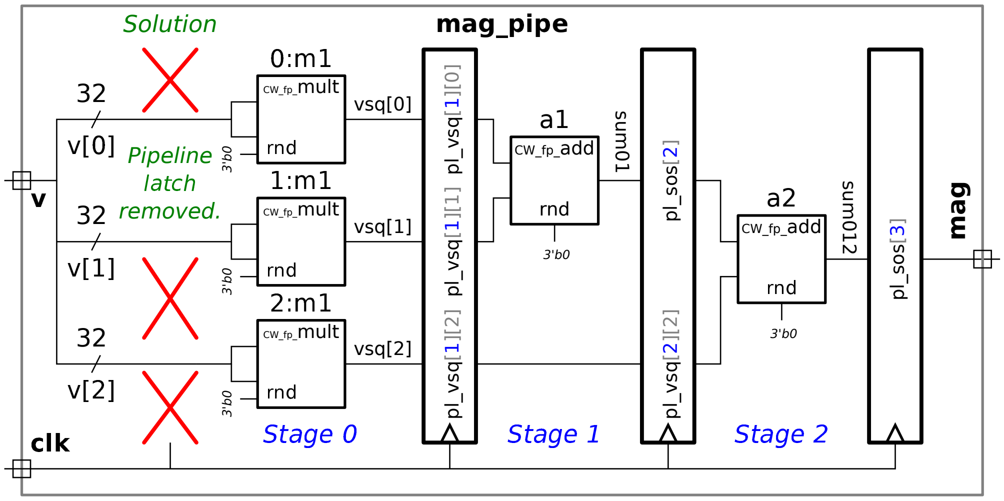
☑ Modify for early-arriving `v`.

Short Answer: Solution appears below after the unmodified module.

Explanation: In this case the pipeline latch at the inputs was removed, saving the cost of the pipeline latch. Since the inputs arrive early there should be enough time to compute the products during the clock cycle in which the inputs first arrive. The removed pipeline latch would be necessary if the inputs arrived later in a clock cycle, in that case the multiplication would not start until the next clock cycle.

This modification does not change throughput but does reduce latency.


mag_pipe
v
32
v[0]
v[1]
v[2]
clk
pl_v[0]
pl_v[1]
pl_v[2]
0:m1
1:m1
2:m1
cw_fp_mult
rnd
3'b0
vsq[0]
vsq[1]
vsq[2]
pl_vsq[1][0]
pl_vsq[1][1]
pl_vsq[1][2]
a1
cw_fp_add
sum01
pl_sos[2]
pl_vsq[2][2]
a2
sum012
pl_sos[3]
mag
Stage 0
Stage 1
Stage 2


Solution
mag_pipe
Pipeline latch removed.
v
32
v[0]
v[1]
v[2]
clk
0:m1
1:m1
2:m1
cw_fp_mult
rnd
3'b0
vsq[0]
vsq[1]
vsq[2]
pl_vsq[1][0]
pl_vsq[1][1]
pl_vsq[1][2]
a1
cw_fp_add
sum01
pl_sos[2]
pl_vsq[2][2]
a2
sum012
pl_sos[3]
mag
Stage 0
Stage 1
Stage 2