LSU EE 4755 — Digital Design Using Verilog
Problem Set — Sequential Code

This is a collection of EE 4755 solved problems related to sequential logic. The problems have been selected to cover important areas and are ordered from simple to more elaborate.

Each problem appears twice, first without a solution, and in the second half of this file, with a solution.

2016 Midterm Exam Problem 4

The subjects of this problem are two very simple sequential modules implementing counters. An important difference between the two modules is the logic connected to the output port. That difference, and its implications are covered in part b of the problem.

Appearing in this problem are several variations on a counter.

(*a*) Show the hardware inferred for each counter below.

```
module ctr_a( output uwire [9:0] count, input uwire clk );

   logic [9:0] last_count;
   assign count = last_count + 1;
   always_ff @( posedge clk ) last_count <= count;

endmodule

module ctr_b( output logic [9:0] count, input uwire clk );

   uwire [9:0] next_count = count + 1;
   always_ff @( posedge clk ) count <= next_count;

endmodule
```

☐ Inferred hardware for ☐ `ctr_a` and ☐ `ctr_b`.

(*b*) There is a big difference in the timing of the outputs of `ctr_a` and `ctr_b`. Explain the difference and illustrate with a timing diagram.

☐ Difference between two modules. ☐ Timing Diagram.

2015 Final Exam Problem 2

This problem starts with simple Verilog code describing a memory module, `smemory`*. Module* `smemory` *uses a Verilog array for the memory itself and provides ports to reading and writing the memory. Part (a) asks for the inferred hardware (assuming the synthesis program doesn't just map it to a memory module in the FPGA or ASIC library). In part (b) a timing diagram is to be completed, and in part (c) the module is to be modified based on a given timing diagram.*

Part (a) requires understanding of how hardware is inferred for the array index operator, for example, `a[i]`*, when used as an expression on the right-hand side,* `x = a[i];` *and as an lvalue (on the left-hand side),* `a[i] = y;`*.*

Parts (b) and (c) check for understanding of the timing of sequential circuits, part (c) also exercises design skills.

The module below implements a simple memory module.

```
module smemory #( int size_lg = 4, int dbits = 8, int size = 1 << size_lg )
   ( output uwire [dbits-1:0] rd_data,
     input uwire [size_lg-1:0] wr_idx,  input uwire [dbits-1:0] wr_data,  input uwire write,
     input uwire [size_lg-1:0] rd_idx,  input uwire clk );

   logic [dbits-1:0] storage [size-1:0];

   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;

   assign rd_data = storage[rd_idx];

endmodule
```

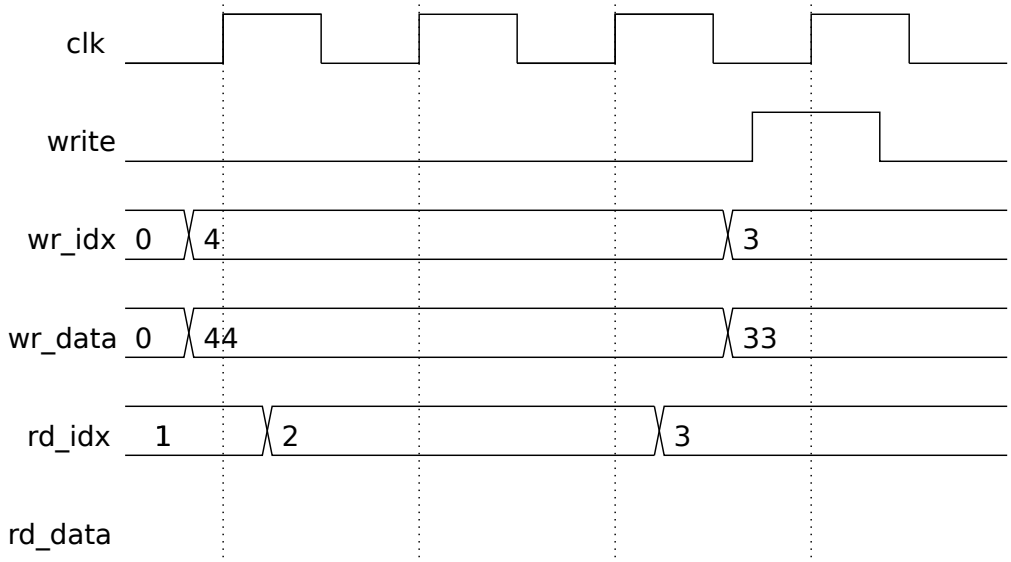
(*a*) Show the hardware that will be synthesized for this module when elaborated with `size_lg = 2`. Use registers, multiplexors, decoders, and basic gates. **Do not** use a memory module.

☐ Show synthesized hardware, including hardware for ☐ reading and ☐ writing.

Appearing below is the module from the previous page.

```
module smemory #( int size_lg = 4, int dbits = 8, int size = 1 << size_lg )
   ( output uwire [dbits-1:0] rd_data,
     input uwire [size_lg-1:0] wr_idx,  input uwire [dbits-1:0] wr_data,  input uwire write,
     input uwire [size_lg-1:0] rd_idx,  input uwire clk );

   logic [dbits-1:0] storage [size-1:0];

   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;

   assign rd_data = storage[rd_idx];

endmodule
```

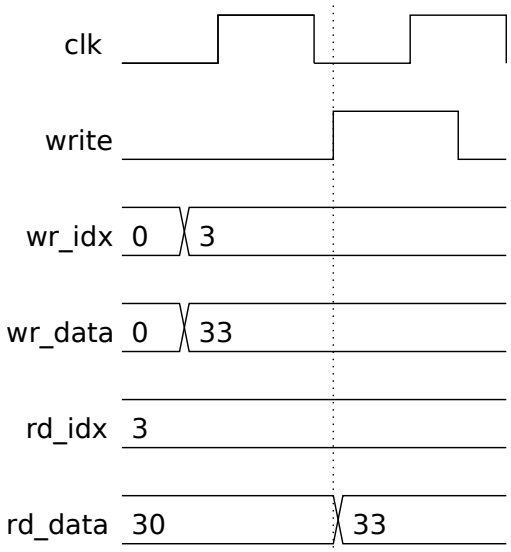
(*b*) Assume that initially location 1 (`storage[1]`) holds a 10, location 2 holds a 20, location 3 holds a 30, and so on. Complete the timing diagram below, consistent with this module.



☐ Complete `rd_data` row of timing diagram.

(*c*) Modify the module below (same as one on previous page) so that its behavior is consistent with the timing diagram to the right. That is, if the location being written is the same as the one being read the `rd_data` output shows the data on `wr_data`. If the locations don't match or nothing is being written the behavior is unchanged.

☐ Modify the module.




```
module smemory_bp #( int size_lg = 4, int dbits = 8, int size = 1 << size_lg )
 ( output uwire [dbits-1:0] rd_data,
   input uwire [size_lg-1:0] wr_idx,  input uwire [dbits-1:0] wr_data,  input uwire write,
   input uwire [size_lg-1:0] rd_idx,  input uwire clk );

   logic [dbits-1:0] storage [size-1:0];









   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;









   assign rd_data = storage[rd_idx];









endmodule
```

2018 Midterm Exam Problem 5

This problem checks for understanding of how sequential logic is inferred from a Verilog description. An important element is understanding which objects will be synthesized into registers. This particular module performs an iterative calculation, meaning that the results computed in one cycle are used in a subsequent cycle.

Show the hardware that will be inferred for the Verilog code below.

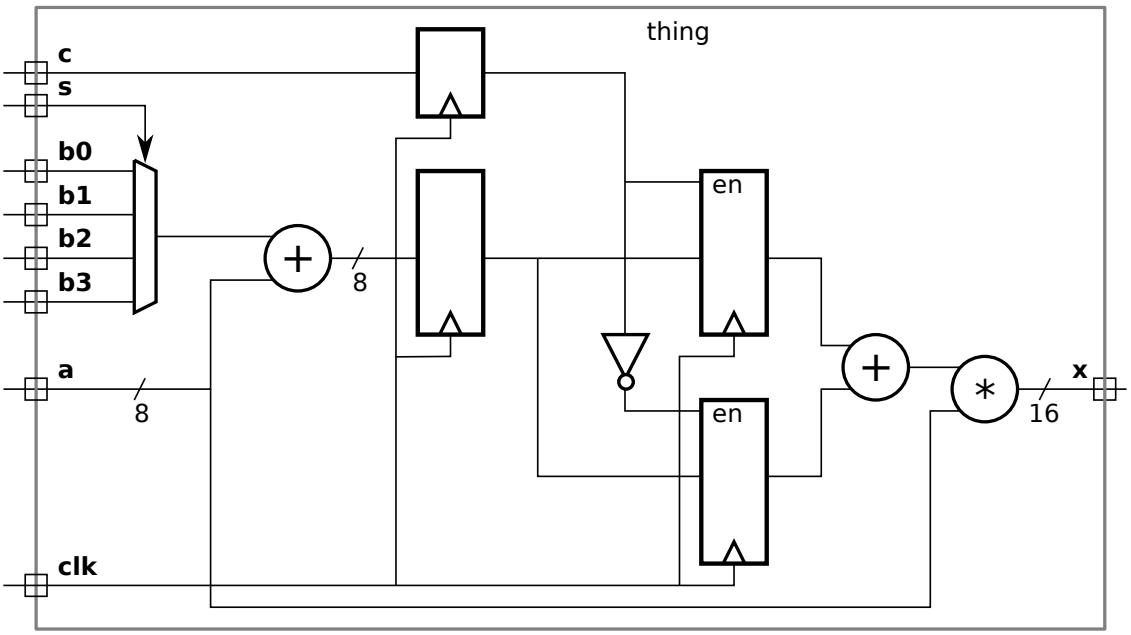
- ☐ Clearly show module ports.
- ☐ Show inferred hardware. Don't optimize.
- ☐ Pay close attention to what is and is not inferred as a register.

```
module regs #( int w = 10, int k1 = 20, int k2 = 30 )
   ( output logic [w-1:0] y,
     input logic [w-1:0] b, c,
     input uwire clk );

   logic [w-1:0] a, x, z;

   always_ff @( posedge clk ) begin

      a = b + c;
      if ( a > k1 ) x = b + 10;
      if ( a > k2 ) z = b + x; else z = c - x;
      y = x + z;

   end

endmodule
```

2015 Final Exam Problem 1

In this problem a Verilog description is to be written based on a given diagram of a sequential circuit. An important concept to understand is the advancement of data through the module with each clock edge. This must be understood before writing Verilog, and one must code the Verilog description so data is written into registers at the right time.

Write a Verilog description of the hardware illustrated below.



☐ Verilog description of hardware including ☐ port declarations and ☐ port and other sizes.

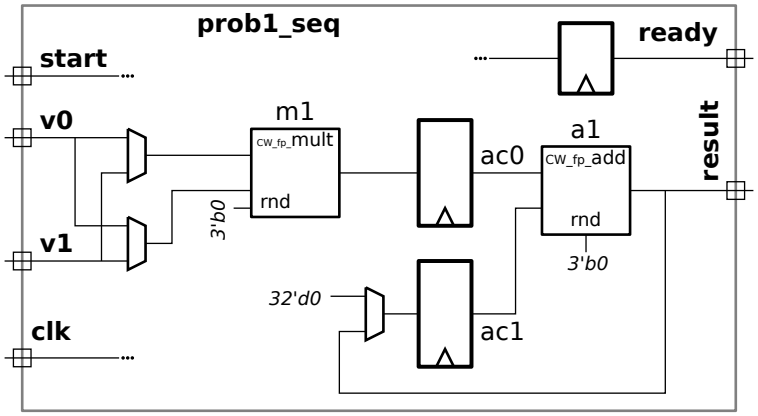
2016 Final Exam Problem 1

This problem starts with both incomplete Verilog and an incomplete hardware diagram. They show two presumably costly floating-point calculation modules, an adder and a multiplier. The goal of the problem is to use these modules over multiple cycles to perform a calculation. To solve the problem one must decide what the adder and multiplier will do each cycle based on what is needed in the calculation and what connections are available based on the diagram. Also, one must know how to write Verilog to choreograph all of this.

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.

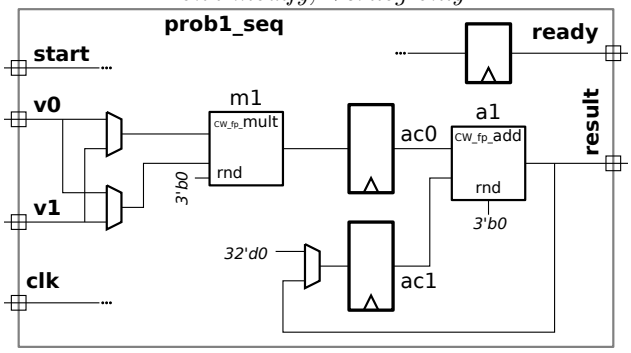


Don't modify this diagram, write Verilog code.

```
module prob1_seq( output uwire [31:0] result,   output uwire 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, add_a, add_b, prod, sum;

   logic [31:0] ac0, ac1;       logic [2:0]  step;

   localparam   int last_step = 1;
   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(3'd0), .z( prod ), .status(mul_s) );
   CW_fp_add  a1( .a( add_a ), .b( add_b ), .rnd(3'd0), .z( sum ),  .status(add_s) );

   assign       ready = step == last_step; /// THIS MUST BE CHANGED.

   /// USE NEXT PAGE FOR SOLUTION!
endmodule
```

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 uwire 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,  add_a, add_b,  prod, sum;
   logic [31:0] ac0, ac1;
   logic [2:0]  step;

   localparam   int last_step = 1;                          //  ☐  MUST BE CHANGED.
   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(3'd0), .z( prod ), .status(mul_s) );
   CW_fp_add  a1( .a( add_a ), .b( add_b ), .rnd(3'd0), .z( sum ),  .status(add_s) );

   assign       ready = step == last_step;                  //  ☐  MUST BE CHANGED.
```

```
endmodule
```

2018 Homework 8 Problem 3

This is a good moderate-difficulty problem on inference of hardware and of computing cost and delay. The subject of this problem is Verilog code for a degree-m sequential multiplier, a multiplier that computes m partial products per clock cycle. Unlike those other degree-m sequential multipliers this one skips over multiplicand digits that are equal to zero! In part a the inferred hardware for the Verilog is to be found. This requires a good understanding not just of which objects are inferred into registers, but also of if statements and iteration. In part b the cost and delay is to be found. A good part c would be a comparison with conventional (those other) degree-m multipliers, but that's not part of the problem.

Appearing below is a solution to 2018 Homework 7, Problem 2, the streamlined degree-m multiplier with handshaking. The complete solution is at `https://www.ece.lsu.edu/koppel/v/2018/hw07-sol.v.html`. For this problem assume that w and m are both powers of 2.

```
module mult_seq_d_prob_2 #( int w = 16, int m = 2 )
   ( output logic [2*w-1:0] prod,   output logic out_avail,
     input uwire clk, in_valid,     input uwire [w-1:0] plier, cand );

   localparam int iterations = ( w + m - 1 ) / m;
   localparam int iter_lg = $clog2(iterations);

   uwire [iterations-1:0][m-1:0] cand_2d = cand;

   bit [iter_lg-1:0] iter;
   logic [2*w-1:0] accum;

   always_ff @( posedge clk ) begin

      logic [iter_lg-1:0] next_iter;

      if ( in_valid ) begin
         iter = 0;
         accum = 0;
         out_avail = 0;
      end else if ( !out_avail && iter == 0 ) begin
         prod = accum;
         out_avail = 1;
      end

      accum += plier * cand_2d[iter] << ( iter * m );

      next_iter = 0;
      for ( int i=iterations-1;  i>0;  i-- )
        if ( i>iter && cand_2d[i] ) next_iter = i;
      iter = next_iter;
   end

endmodule
```

(*a*) Show the hardware that will be inferred for this module.

(*b*) Compute the cost and delays for this module using the simple model. Show these in terms of w and m. Clearly show the critical path on your diagram.

Solutions

2016 Midterm Exam Problem 4 — Solution

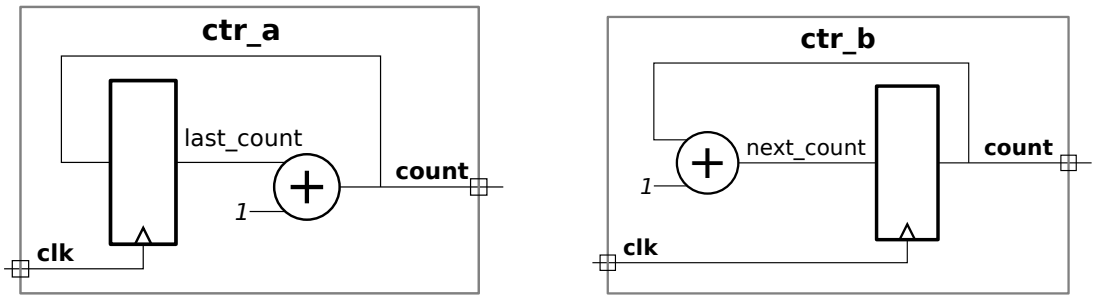
The subjects of this problem are two very simple sequential modules implementing counters. An important difference between the two modules is the logic connected to the output port. That difference, and its implications are covered in part b of the problem.

Appearing in this problem are several variations on a counter.

(*a*) Show the hardware inferred for each counter below.

```
module ctr_a( output uwire [9:0] count, input uwire clk );

   logic [9:0] last_count;
   assign count = last_count + 1;
   always_ff @( posedge clk ) last_count <= count;

endmodule

module ctr_b( output logic [9:0] count, input uwire clk );

   uwire [9:0] next_count = count + 1;
   always_ff @( posedge clk ) count <= next_count;

endmodule
```

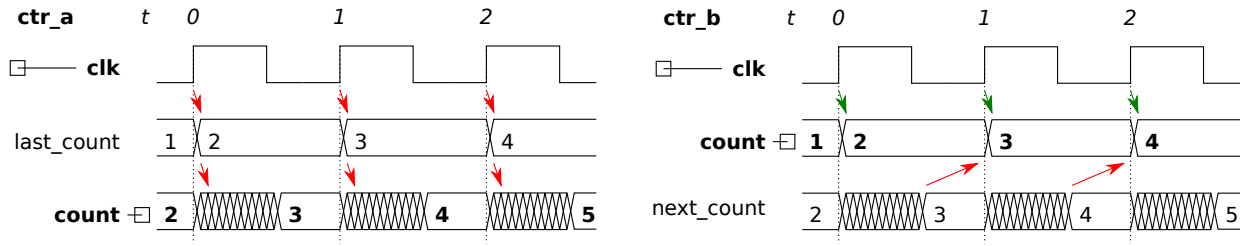
☑ Inferred hardware for ☑ `ctr_a` and ☑ `ctr_b`.



(*b*) There is a big difference in the timing of the outputs of `ctr_a` and `ctr_b`. Explain the difference and illustrate with a timing diagram.

☑ Difference between two modules. ☑ Timing Diagram.

In `ctr_a` the module output, `count`, is connected to the output of an adder. That means the value at the output will not be stable until later in the clock cycle. See the left-side timing diagram below. External hardware could not do anything with the value other than clocking it into a register for use in the next clock cycle. In contrast, the `ctr_b` module output, `count`, is connected to a register output, and so it is available for use at the beginning of the clock cycle.



2015 Final Exam Problem 2 — Solution

This problem starts with simple Verilog code describing a memory module, `smemory`*. Module* `smemory` *uses a Verilog array for the memory itself and provides ports to reading and writing the memory. Part (a) asks for the inferred hardware (assuming the synthesis program doesn't just map it to a memory module in the FPGA or ASIC library). In part (b) a timing diagram is to be completed, and in part (c) the module is to be modified based on a given timing diagram.*

Part (a) requires understanding of how hardware is inferred for the array index operator, for example, `a[i]`*, when used as an expression on the right-hand side,* `x = a[i];` *and as an lvalue (on the left-hand side),* `a[i] = y;`*.*

Parts (b) and (c) check for understanding of the timing of sequential circuits, part (c) also exercises design skills.

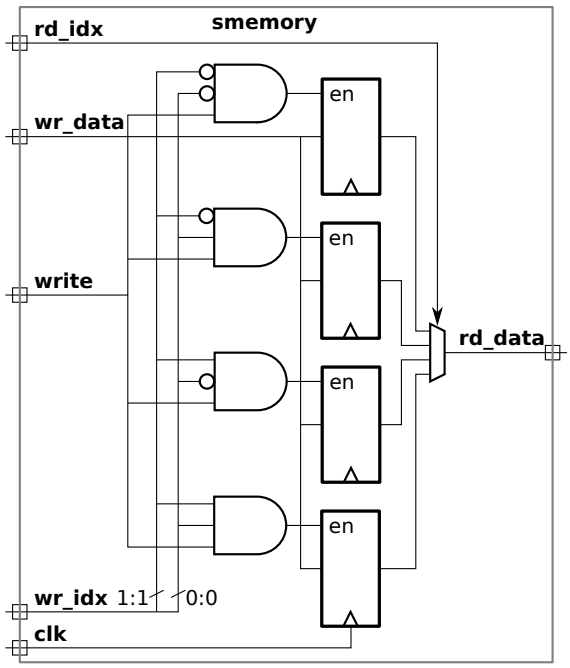
The module below implements a simple memory module.

```
module smemory #( int size_lg = 4, int dbits = 8, int size = 1 << size_lg )
   ( output uwire [dbits-1:0] rd_data,
     input uwire [size_lg-1:0] wr_idx,  input uwire [dbits-1:0] wr_data,  input uwire write,
     input uwire [size_lg-1:0] rd_idx,  input uwire clk );

   logic [dbits-1:0] storage [size-1:0];

   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;

   assign rd_data = storage[rd_idx];

endmodule
```

(*a*) Show the hardware that will be synthesized for this module when elaborated with `size_lg = 2`. Use registers, multiplexors, decoders, and basic gates. **Do not** use a memory module.

☑ Show synthesized hardware, including hardware for ☑ reading and ☑ writing.

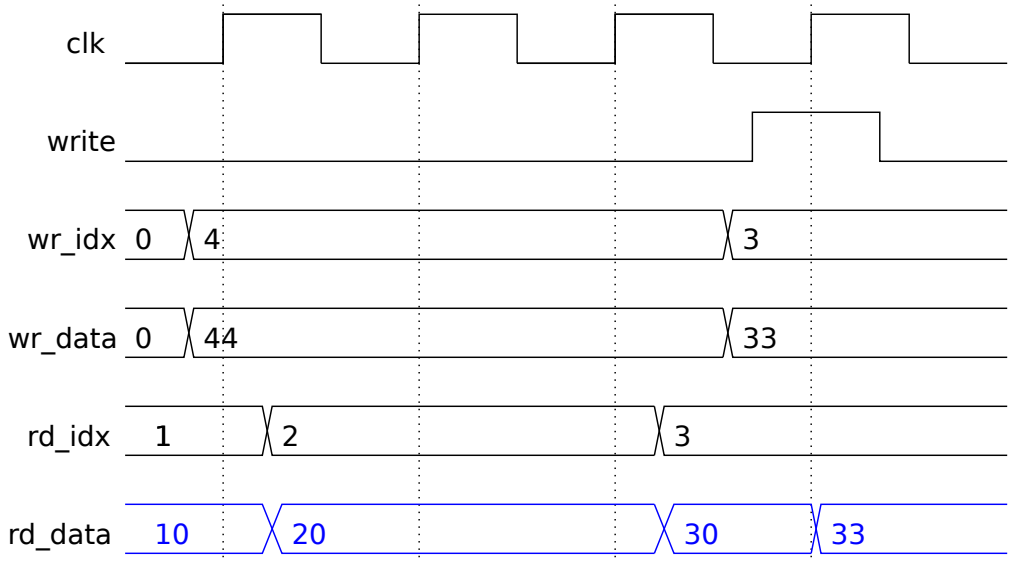
Solution appears to the right.



Appearing below is the module from the previous page.

```
module smemory #( int size_lg = 4, int dbits = 8, int size = 1 << size_lg )
   ( output uwire [dbits-1:0] rd_data,
     input uwire [size_lg-1:0] wr_idx,  input uwire [dbits-1:0] wr_data,  input uwire write,
     input uwire [size_lg-1:0] rd_idx,  input uwire clk );

   logic [dbits-1:0] storage [size-1:0];

   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;

   assign rd_data = storage[rd_idx];

endmodule
```

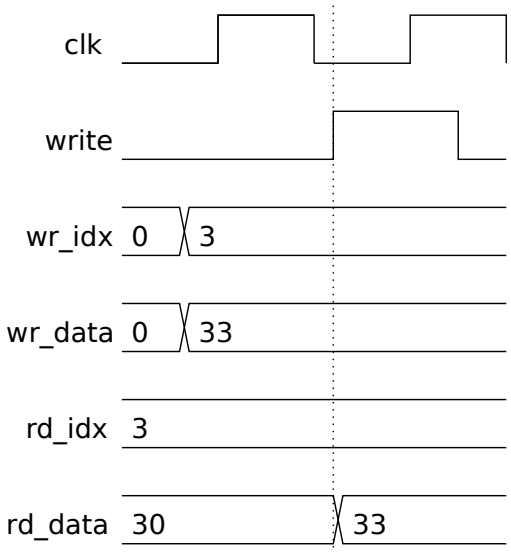
(*b*) Assume that initially location 1 (`storage[1]`) holds a 10, location 2 holds a 20, location 3 holds a 30, and so on. Complete the timing diagram below, consistent with this module.



☑ Complete `rd_data` row of timing diagram.

Solution appears above in blue.

(*c*) Modify the module below (same as one on previous page) so that its behavior is consistent with the timing diagram to the right. That is, if the location being written is the same as the one being read the `rd_data` output shows the data on `wr_data`. If the locations don't match or nothing is being written the behavior is unchanged.



☑ Modify the module.

Solution appears below. The original line is commented out for reference. Otherwise, cluttering your code with commented out lines is bad style. Instead, learn how to diff your working copy with the latest committed version and be able to do so in < 500 ms.

```
module smemory_bp #( int size_lg = 4, int dbits = 8, int size = 1 << size_lg )
 ( output uwire [dbits-1:0] rd_data,
   input uwire [size_lg-1:0] wr_idx,  input uwire [dbits-1:0] wr_data,  input uwire write,
   input uwire [size_lg-1:0] rd_idx,  input uwire clk );

   logic [dbits-1:0] storage [size-1:0];


   always_ff @( posedge clk ) if ( write ) storage[wr_idx] = wr_data;



   // assign rd_data = storage[rd_idx];
   // SOLUTION
   assign rd_data = write && rd_idx == wr_idx  ?  wr_data  :  storage[rd_idx];


endmodule
```

2018 Midterm Exam Problem 5 — Solution

This problem checks for understanding of how sequential logic is inferred from a Verilog description. An important element is understanding which objects will be synthesized into registers. This particular module performs an iterative calculation, meaning that the results computed in one cycle are used in a subsequent cycle.

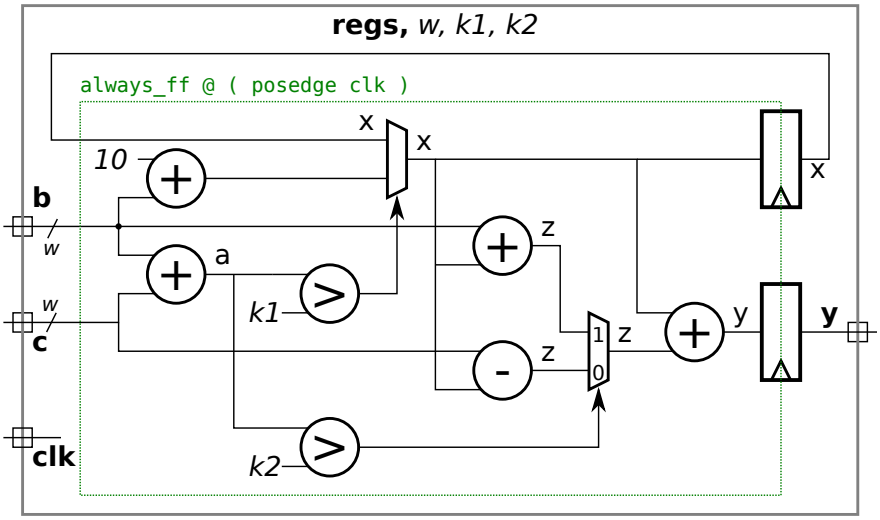
Show the hardware that will be inferred for the Verilog code below.

☑ Clearly show module ports.

☑ Show inferred hardware. Don't optimize.

☑ Pay close attention to what is and is not inferred as a register.

```
module regs #( int w = 10, int k1 = 20, int k2 = 30 )
   ( output logic [w-1:0] y,
     input logic [w-1:0] b, c,
     input uwire clk );

   logic [w-1:0] a, x, z;

   always_ff @( posedge clk ) begin

      a = b + c;
      if ( a > k1 ) x = b + 10;
      if ( a > k2 ) z = b + x; else z = c - x;
      y = x + z;

   end

endmodule
```



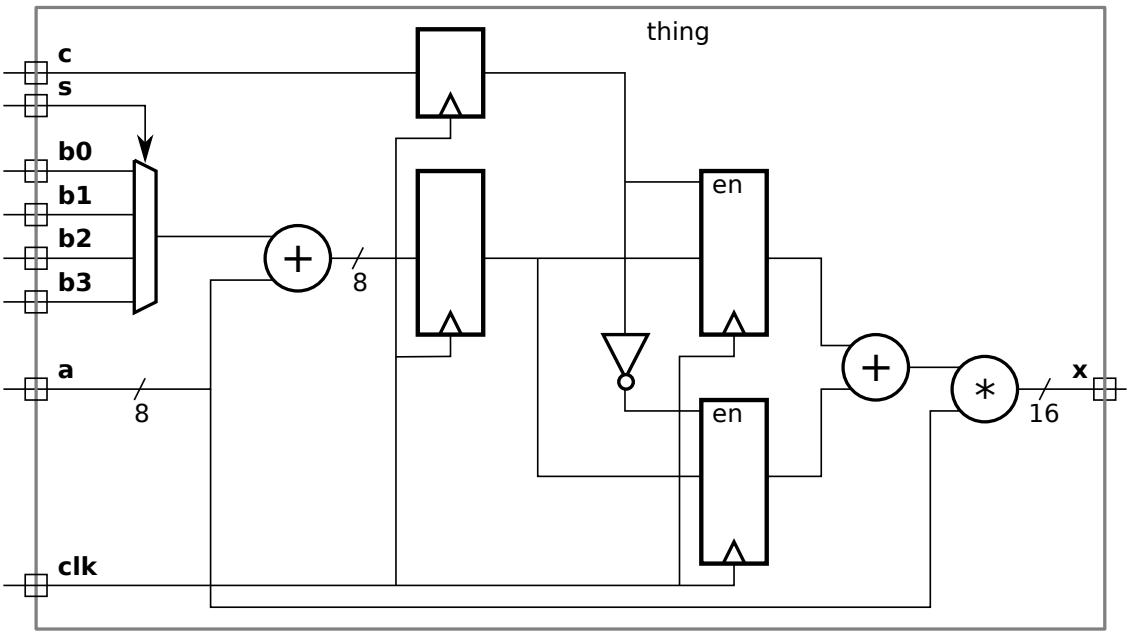
Solution appears above

Explanation: The area corresponding to the `always_ff` block is outlined in a green dashed line. Registers are shown on the right-hand boundary because the value that gets clocked into a register is the value present when control reaches the end of the block (the `end` statement above). Four values are assigned within the block, `a`, `x`, `z`, and `y`. Registers are inferred only for those variables that are a *live out* object of the block. That is true for `y` since it's also a module output and so its value is needed outside the block. In contrast, the value of `a` that is computed in the block is not used again after the `end` is reached. (When the block is re-entered a new value of `a` will be computed.) The same is true for `z`. But the value of `x` may be used after `end` is reached. That happens when the block is re-entered and $a \leq k_1$, in which case `x` is set to the previous value of `x` (the one in the register) rather than `b+10`.

2015 Final Exam Problem 1 — Solution

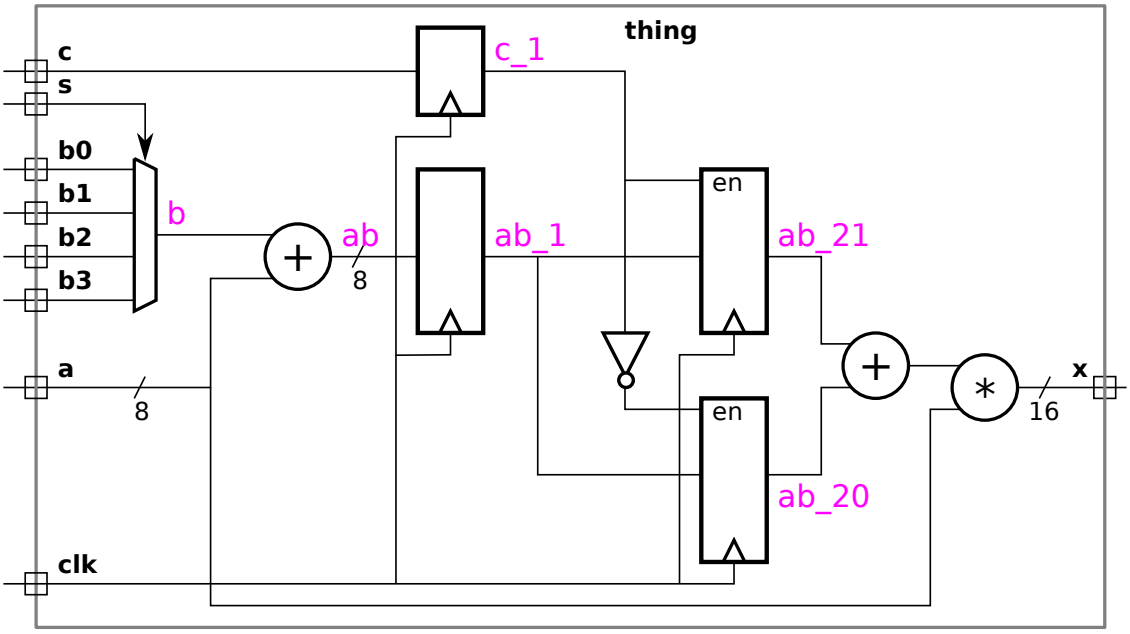
In this problem a Verilog description is to be written based on a given diagram of a sequential circuit. An important concept to understand is the advancement of data through the module with each clock edge. This must be understood before writing Verilog, and one must code the Verilog description so data is written into registers at the right time.

Write a Verilog description of the hardware illustrated below.



SOLUTION ON NEXT PAGE

☑ Verilog description of hardware including ☑ port declarations and ☑ port and other sizes.



The solution appears below. Names for wires that were unlabeled in the problem appear in purple. (That is, the purple labels are part of the solution.) Note the use of `case`/`endcase` for the mux. Though using an `if`/`else` chain or the conditional operator, `?:`, would be correct, they are more tedious and prone to error and so it's worth taking the trouble to remember to use `case`.

```
module thing( output uwire [15:0] x,  input uwire c,       input uwire [1:0] s,
              input uwire [7:0] b0, b1, b2, b3, a,          input wuire clk );

   logic [7:0] b, ab, ab_1, ab_20, ab_21;
   logic       c_1;

   always_comb begin
      case ( s )
        0: b = b0;
        1: b = b1;
        2: b = b2;
        3: b = b3;
      endcase
      ab = a + b;
   end

   always_ff @( posedge clk ) begin
      c_1 <= c;      // Note: Delayed assignment, so if(c_1) uses prior value.
      ab_1 <= ab;    // Delayed assignment here too.
      if ( c_1 ) ab_21 <= ab_1; else ab_20 <= ab_1;
   end

   assign       x = a * ( ab_20 + ab_21 );
endmodule
```

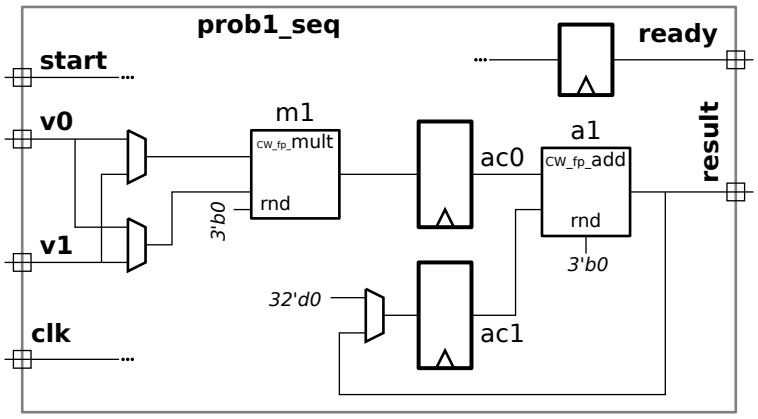
2016 Final Exam Problem 1 — Solution

This problem starts with both incomplete Verilog and an incomplete hardware diagram. They show two presumably costly floating-point calculation modules, an adder and a multiplier. The goal of the problem is to use these modules over multiple cycles to perform a calculation. To solve the problem one must decide what the adder and multiplier will do each cycle based on what is needed in the calculation and what connections are available based on the diagram. Also, one must know how to write Verilog to choreograph all of this.

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.

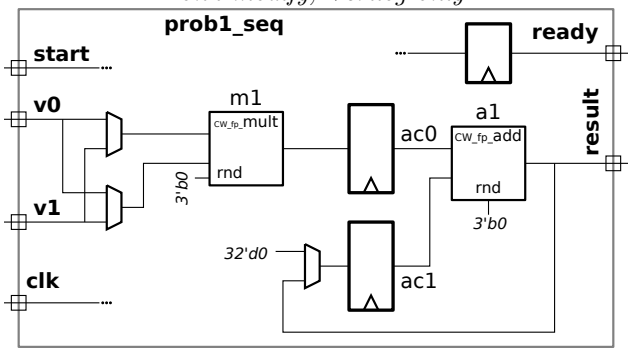
Solution on this page.

☑ Complete Verilog so that module computes `v0*v0 + v0*v1 + v1*v1`.

☑ Synthesized hardware must be consistent with diagram, ☑ especially synthesized registers.

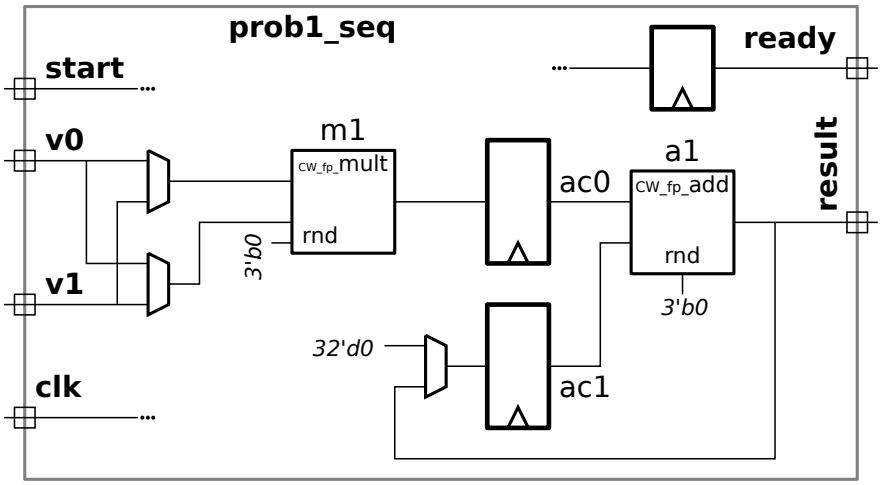
☑ Note that `ready` must come from a register.

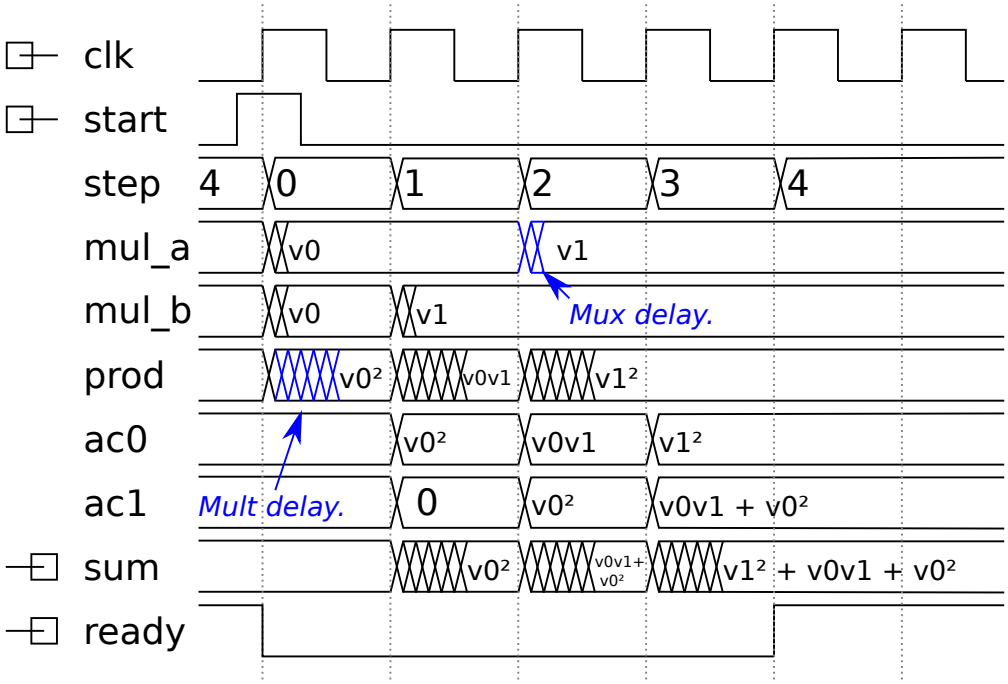
☑ Don't skip the easy part: connections to adder.




```
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.





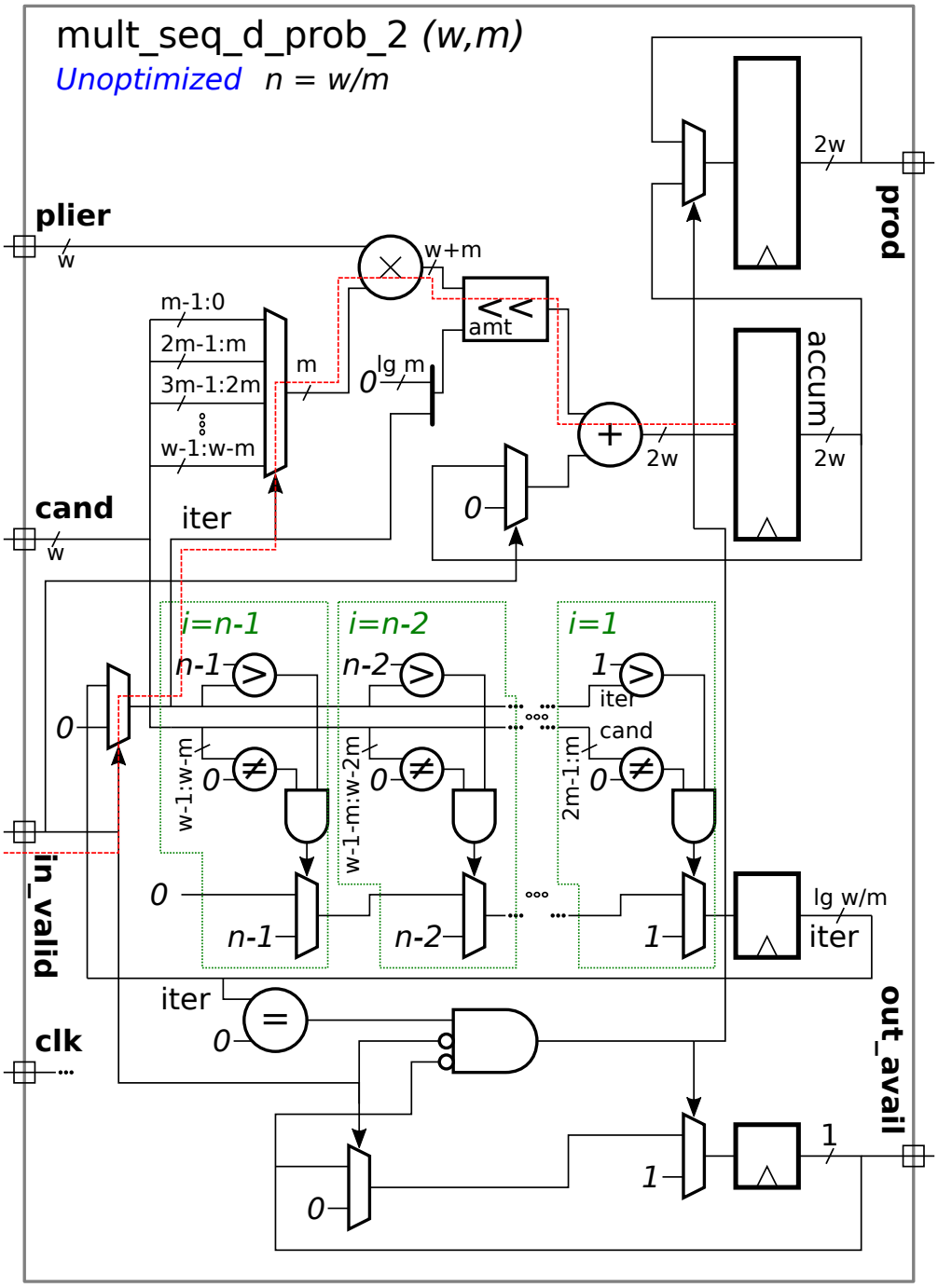
2018 Homework 8 Problem 3 — Solution

This is a good moderate-difficulty problem on inference of hardware and of computing cost and delay. The subject of this problem is Verilog code for a degree-m sequential multiplier, a multiplier that computes m partial products per clock cycle. Unlike those other degree-m sequential multipliers this one skips over multiplicand digits that are equal to zero! In part a the inferred hardware for the Verilog is to be found. This requires a good understanding not just of which objects are inferred into registers, but also of if statements and iteration. In part b the cost and delay is to be found. A good part c would be a comparison with conventional (those other) degree-m multipliers, but that's not part of the problem.

Appearing below is a solution to 2018 Homework 7, Problem 2, the streamlined degree-m multiplier with handshaking. The complete solution is at `https://www.ece.lsu.edu/koppel/v/2018/hw07-sol.v.html`. For this problem assume that w and m are both powers of 2.

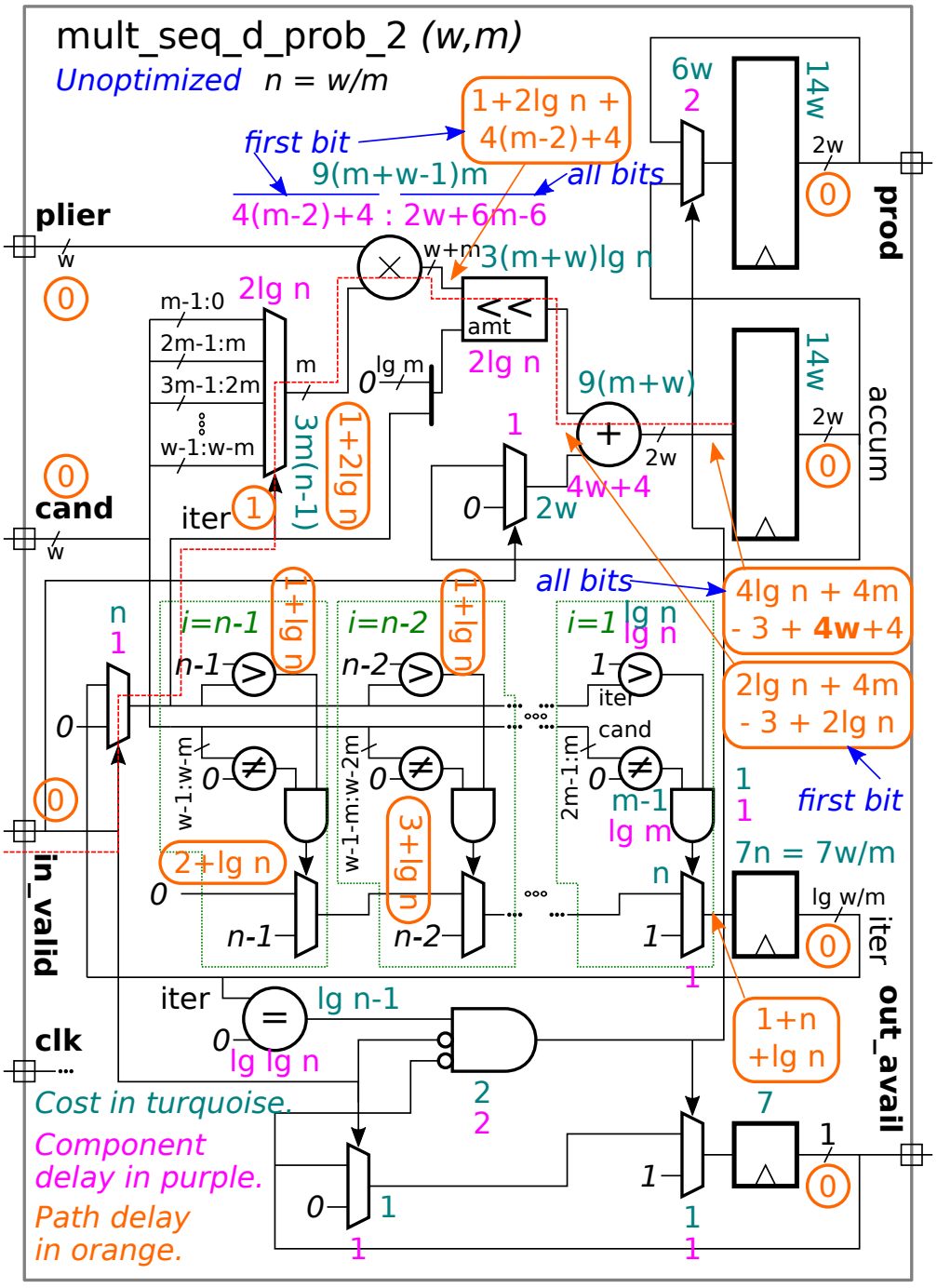
```
module mult_seq_d_prob_2 #( int w = 16, int m = 2 )
   ( output logic [2*w-1:0] prod,   output logic out_avail,
     input uwire clk, in_valid,     input uwire [w-1:0] plier, cand );

   localparam int iterations = ( w + m - 1 ) / m;
   localparam int iter_lg = $clog2(iterations);

   uwire [iterations-1:0][m-1:0] cand_2d = cand;

   bit [iter_lg-1:0] iter;
   logic [2*w-1:0] accum;

   always_ff @( posedge clk ) begin

      logic [iter_lg-1:0] next_iter;

      if ( in_valid ) begin
         iter = 0;
         accum = 0;
         out_avail = 0;
      end else if ( !out_avail && iter == 0 ) begin
         prod = accum;
         out_avail = 1;
      end

      accum += plier * cand_2d[iter] << ( iter * m );

      next_iter = 0;
      for ( int i=iterations-1;  i>0;  i-- )
        if ( i>iter && cand_2d[i] ) next_iter = i;
      iter = next_iter;
   end

endmodule
```

(*a*) Show the hardware that will be inferred for this module.



Hardware shown above with the critical path shown in red.

(*b*) Compute the cost and delays for this module using the simple model. Show these in terms of w and m. Clearly show the critical path on your diagram.



The costs and delay of each component are shown in the diagram above. The path delay for selected paths is shown in the circled orange numbers. Note that one input to all of the comparison units (for example, the zero in $\neq 0$), is a constant, reducing their costs and delays. Many of the multiplexors also have one constant data input.

The interesting thing to compare is the time needed to compute the updated `accum` value versus the time needed to find the next non-zero digit. The $i >$ iter comparison, because i is a constant, takes time $\lg w/m\,\mathrm{u_t} = \lg n\,\mathrm{u_t}$ and the $\neq 0$ takes less,

especially if $w/m > m$. The mux delay is $1\,\mathrm{u_t}$ because one data input is a constant. The time to generate the new `iter` signal is $(1 + n + \lg n)\,\mathrm{u_t}$.

The updated `accum` value consumes most of the time. Inputs arrive at the multiplier at time $1+2\lg n$. For an unoptimized m-bit by $w+m$-bit multiplier, the least significant bit takes $(4(m-2)+4)\,\mathrm{u_t}$ to compute. Since the shifter can shift by n possible amounts its delay is $2\lg n$. The least significant bit arrives at the adder at time $1+2\lg n+4(m-2)+4+2\lg n = (4\lg n+4m-3)\,\mathrm{u_t}$ (see the diagram). The adder requires $(4w+4)\,\mathrm{u_t}$ to finish and so the adder output is ready at time $(4\lg n+4m-3+4w+4)\,\mathrm{u_t}$.

The clock period would include six more cycles for the latch setup time.