Name Solution

Digital Design using HDLs
LSU EE 4755
Final Examination

Friday, 13 December 2019 10:00-12:00 CST

Problem 1		(30 pts)
Problem 2		(25 pts)
Problem 3		(20 pts)
Problem 4		(25 pts)
Exam Total		(100 pts)

Alias ☣ It Begins

Good Luck!

Problem 1: [30 pts] Appearing below is the solution to Homework 6, the accumulation module. The next page shows the pipelined adder and `st_occ`, which is some of the inferred hardware. Show the rest of the inferred hardware after some optimization. Leave the pipelined adder as a box.

```
module add_accum #( int w = 21, n_stages = 3 )
   ( output logic [w-1:0] sum,   output logic sum_valid,
     input uwire [w-1:0] ai,     input uwire ai_v, reset, clk );

   logic [n_stages-1:0] st_occ;
   assign sum_valid = !st_occ;
   uwire aout_v = st_occ[n_stages-1];

   uwire [w-1:0] aout;
   uwire [w-1:0] a0 = ai_v   ? ai   : sum;
   uwire [w-1:0] a1 = aout_v ? aout : sum;

   add_pipe #(w,n_stages) add_p0( aout, a0, a1, clk );

   logic sum_occupied;
   uwire [1:0] n_values = ai_v + sum_occupied + aout_v;
   uwire saa = n_values >= 2;  // Start an addition.
   uwire write_sum = !sum_occupied && n_values == 1;

   always_ff @( posedge clk ) if ( reset ) begin
      sum <= 0;
      sum_occupied <= 0;
      st_occ <= 0;
   end else begin
      if ( write_sum ) sum <= aout_v ? aout : ai;
      sum_occupied <= n_values[0];
      st_occ <= { st_occ[n_stages-1:0], saa };
   end
endmodule
```

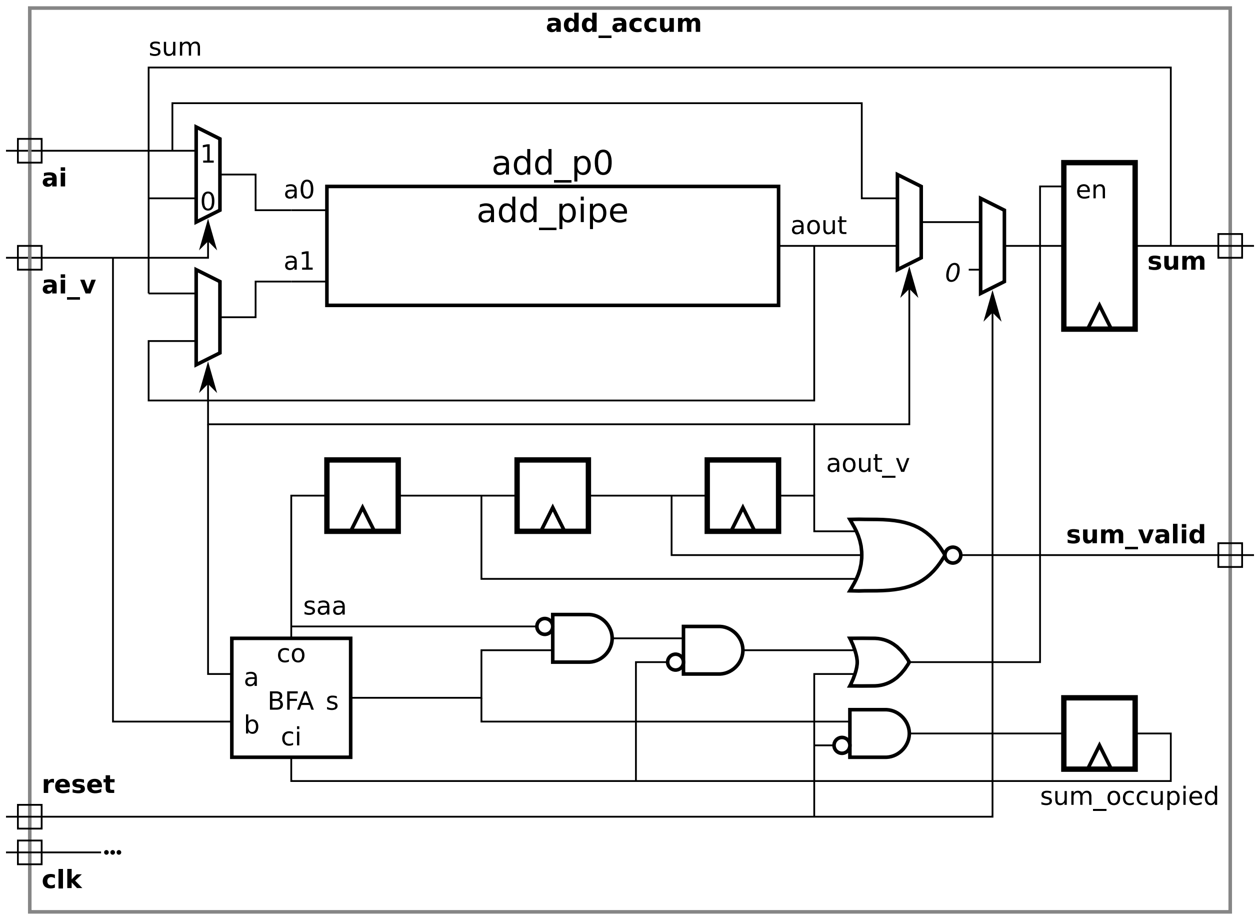
☑ Show inferred hardware after some optimization, but ☑ leave `add_pipe` as a box.

☑ Show logic associated with `n_values` as basic gates and a single BFA, **do not show adders and do not show comparison units**.

☑ Clearly show all input and output ports, do not confuse parameters with ports.

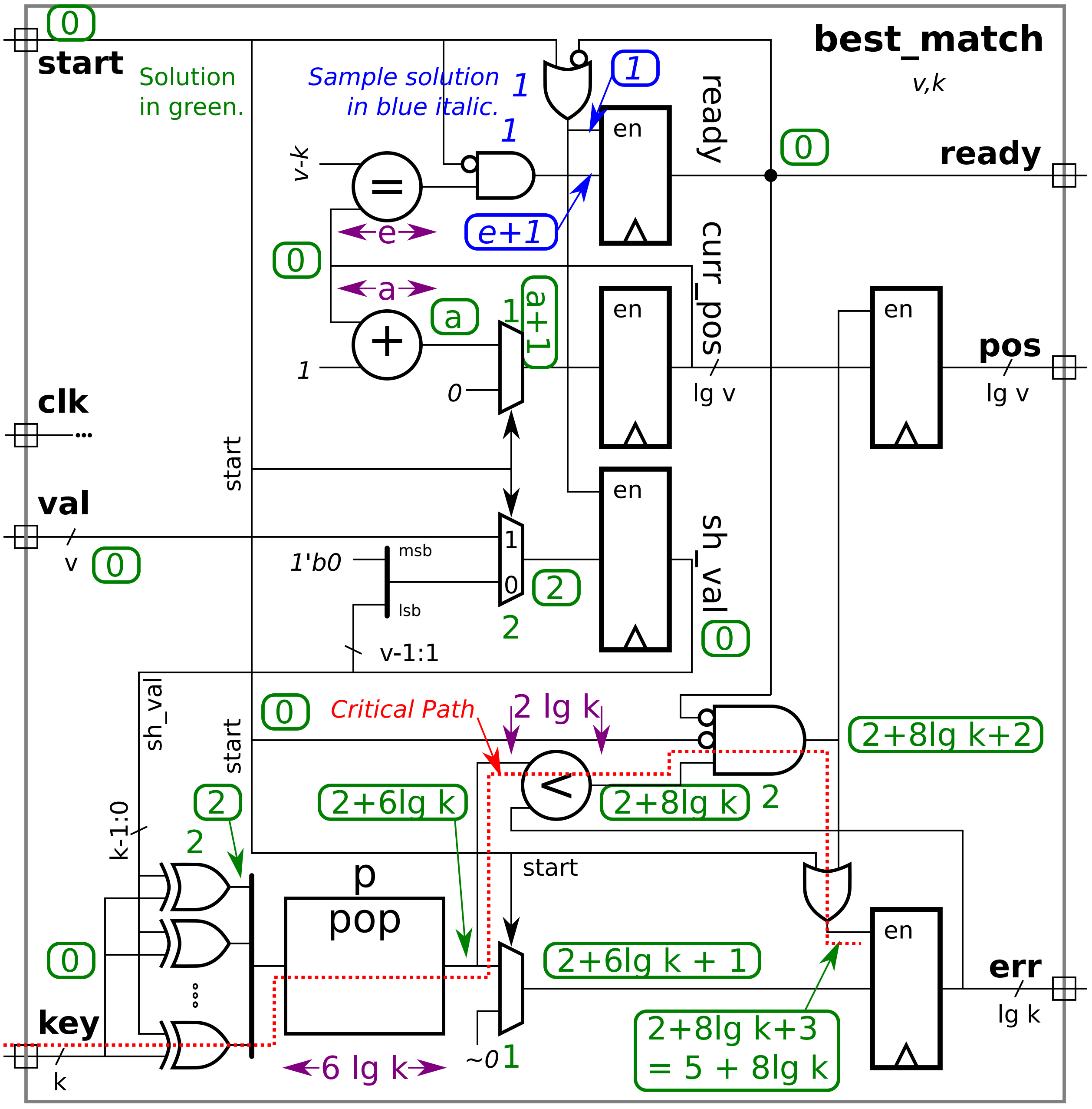
☑ Avoid effortlessly optimized hardware, such as gates with constant inputs.

Solution appears below.



Problem 2: [25 pts] Appearing below is hardware from the solution to Homework 5, Problem 2. The parameter names have been shortened, such as changing `wv` to `v` and using `lg v` for `wvb`. The diagram shows the delay through some of the modules, including the `pop` module. Treat `e` and `a` (delays for $\boxed{=}$ and $\boxed{+}$) as given constants for the first part.

(*a*) Based on the provided delays and using the simple model for others, compute the arrival time (delay) of signals at each register input. That's two inputs for each of five registers. The solution for `ready` is shown in blue, so only four registers remain. Also, highlight the/a critical path to the `err` register.



☑ Show the arrival time of the enable and data signal at each register input and ☑ Highlight a critical path to `err` with a squiggly line.

☑ Take into account constant inputs when computing delays.

Solution to part a: Arrival times at register inputs, as well as the delay at other points, shown in green. The critical path appears as a red dashed (not squiggly) line.

Note that the delay of a mux with a constant data input is 1, which applies to two of the multiplexors in the diagram.

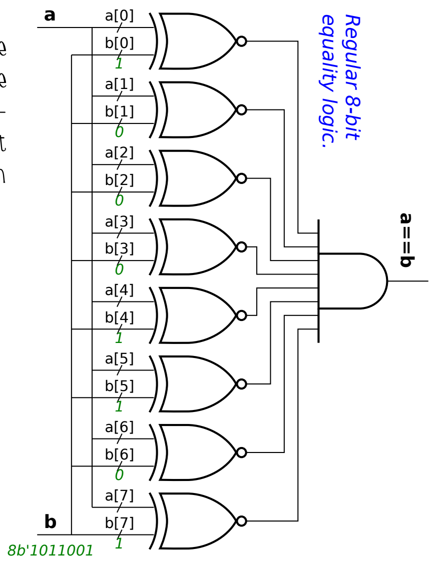
The critical path in the solution starts at `key`. It would also be correct to start the critical path at `sh_val` (and passing through the XOR gates).

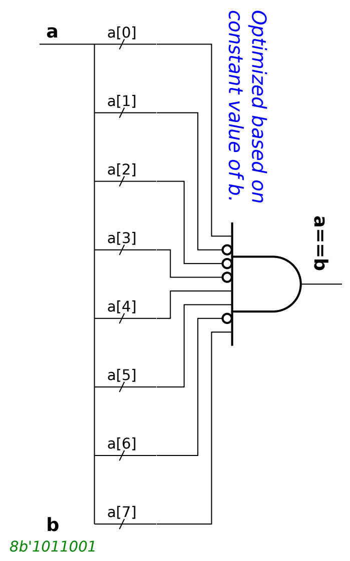
A common mistake was to show the critical path passing *through* a register. Paths start at register outputs and end at register inputs.

(*b*) The equality module is shown with a delay of `e`. Show the hardware for that module and compute the cost and delay using the simple model. Take into account the width of the inputs and the fact that one input is a constant.

☑ Sketch hardware for equality module for $\lg v = 8$ and $v - k = 1011\,0001_2$, and ☑ taking into account the constant input.

Because of the constant input each XNOR gate is optimized to either a NOT gate (where the constant bit is 0) or just wire (where the constant bit is 1). So the equality module is just a $\lceil \lg v \rceil$-input AND gate. See the illustration to the right.





☑ Show the cost of the hardware for the equality module above based on the simple model in terms of $\lg v$. ☑ Don't forget to take the constant input into account.

The hardware consists of a single $\lg v$-input AND gate. Its cost is $[\lg v - 1]\,\mathrm{u_c}$.

☑ Show the delay of the hardware based on the simple model in terms of $\lg v$. ☑ Don't forget to take the constant input into account.

The delay of an $\lg v$-input AND gate is $\lceil \lg \lceil \lg v \rceil \rceil\,\mathrm{u_t}$.

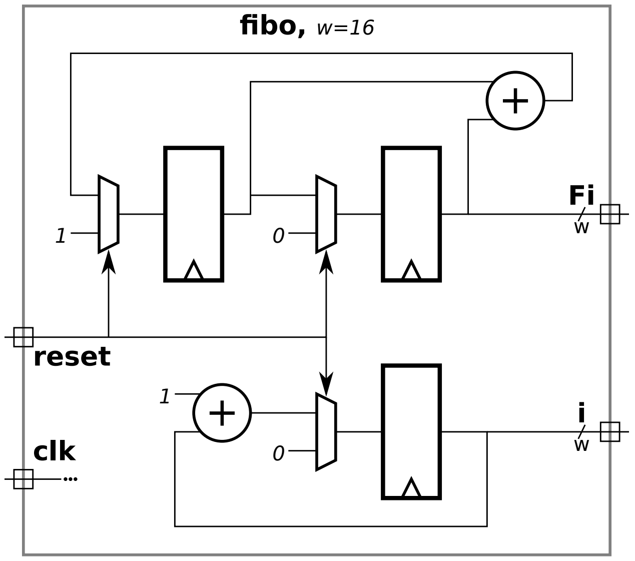
Problem 3: [20 pts] The hardware illustrated to the right emits a famous integer sequence. Write a synthesizable Verilog description of the hardware.

✓ Complete the module, ✓ be sure that it is synthesizable.

✓ Use non-blocking assignments carefully.

✓ Be sure to include all ✓ input and output ports and ✓ parameters.

✓ Make sure that all objects have the appropriate widths.



Solution appears below. The warning about non-blocking assignments needed to be heeded in the solution below so that the value of `Fi` used when updating `Fi_next` would be based on the old value of `Fi`.

```
// SOLUTION

module fibo
  #( int w = 16 )
   ( output logic [w-1:0] Fi, i,
     input uwire reset, clk );

   logic [w-1:0] Fi_next;

   always_ff @( posedge clk ) if ( reset ) begin

      Fi <= 0;
      Fi_next <= 1;
      i <= 0;

   end else begin

      Fi <= Fi_next;
      // Note: The non-blocking assignment above insures that the Fi +
      // Fi_next expression below is computed using the old value of Fi.
      Fi_next <= Fi + Fi_next;
      i <= i + 1;

   end

endmodule
```

Problem 4: [25 pts] Answer each question below.

(a) Appearing below are synthesis script results for the pipelined integer adder from Homework 6. That adder computes a w-bit integer sum using an n-stage pipeline in which each stage computes $\lceil w/n \rceil$ bits of the sum, starting with the $\lceil w/n \rceil$ least-significant bits in the first stage.

All syntheses are of a $w = 24$-bit adder, versions with $n = 1$, 2, 3, 4, and 6 stages are synthesized. The delay target is set to an easy 90 ns.

```
Module Name                               Area   Delay   Delay
                                                Actual  Target
add_pipe_w24_n_stages1                   29928  10.174  90.000 ns
add_pipe_w24_n_stages2                   47043   5.428  90.000 ns
add_pipe_w24_n_stages3                   64159   3.701  90.000 ns
add_pipe_w24_n_stages4                   81275   2.837  90.000 ns
add_pipe_w24_n_stages6                  115506   1.973  90.000 ns
```

☑ Based on this data provide the ☑ latency and ☑ throughput for the three-stage adder. Be sure to ☑ use appropriate units for the throughput.

The latency is $3 \times 3.701 = 11.103\,\mathrm{ns}$. The throughput is $\frac{\text{1 addition}}{3.701\,\mathrm{ns}} = 270.2 \times 10^6$ additions per second.

☑ Note that the area (cost) increases with the number of stages. Based on the description above what is the main contributor to the increase in cost?

The main contributor to cost are the registers. Each stage requires three registers, two for the source operands and one for the sum.

(*b*) The two modules below appear to be similar.

```
module plan_I(output logic [7:0] e, input logic [7:0] a,b);
   logic [7:0] c;
   always_comb begin
      c = a + b;
      e = c + a;
   end
endmodule

module plan_II(output logic [7:0] e, input logic [7:0] a,b);
   logic [7:0] c;
   always_comb e = c + a;
   always_comb c = a + b;
endmodule
```

☑ For which module will the simulator perform unnecessary addition? ☑ Explain.

Module `plan_II` will require extra work because when `a` changes the `e = c + a` can be executed twice, first for the change in `a` then for the change in `c` due to execution of the `c = a + b`.

☑ Is the result computed by the two modules different or the same? ☑ Explain.

The result at the end of a time step is the same. However `plan_II` can leave `e` in different value than `plan_I` during a time step (before `e = c+a` executes a second time, as described above).

(*c*) What value will `y` have at the end of the initial block?

```
module s;
   logic [15:0] a,b,y;
   initial begin
      a = 1;        // SOLUTION information in comments below.
      b = 100;      // Value of b set to 100.
      b <= 10;      // Update event b = 10 is put in NBA region. b still 100.
      y = 0;        // Value of y set to zero.
      y <= a + b;   // a+b computed: 1 + 100 = 101. Update event y=101 put in NBA region.
      y = 999;      // Value of y set to 999.

      #1;           // After #1 reached NBA events executed:
                    //   b set to 10
                    //   y set to 101. (a+b computed above using older b).
                    // The lines below have no impact on y.
      a = 2;
      b <= 20;
      #200;
      // Show value of y at this point in execution.
      // SOLUTION: y is 101.
   end
endmodule
```

☑ Value of `y` at end of block is:

Short answer: `y=101`.

Explanation: `y` is assigned three times. For the blocking assignments, `y=0` and `y=999`, the value is written when the respective statement is executed. For the non-blocking assignment, `y<=a+b`, the value `a+b` is computed when the statement is reached, but the result is not assigned until the simulator reaches the timeslot $t = 0$ NBA region. The same holds for non-blocking assignment `b<=10`. For that reason `a+b` is computed using `a=1` and `b=100`. See the comments in the code above.

(*d*) Consider the declarations below.

```
module types;
   int en;
   logic [31:0] lo;
   bit [31:0] b;
   uwire [31:0] u = 33;
   localparam int p = 22;
endmodule
```

☑ Object `u` has the same data type as one of the other objects. Which is it?

It has the same data type as $\boxed{\texttt{lo}}$. The data type is `logic`. Declarator `uwire` is an object kind, not a data type. For `uwire` kinds the default data type is `logic`.

☑ What is the difference between `lo` and `b` (`logic` and `bit`)?

Both are used to represent one bit. Type `bit` has two states, `0` and `1`, while `logic` has four states, `0`, `1`, `x`, and `z`. The `var logic` objects have value `x` until they are assigned a value. In net `logic` objects (such as something declared `wire`) the value is `x` when there is more than one driver and at least one is driving a `0` and at least another is driving a `1`. A net object with zero drivers has value `z`. It is also possible to specify these values in literals, such as `1'bz`.

☑ Notice that `u` is assigned a value. What is it about object `lo` that makes it illegal to assign a value in its declaration?

Object `lo` is a variable type, and so it can only be assigned in procedural code.

☑ Add correct code to assign value 44 to `lo`.

The solution appears below. If the goal is to assign an initial value then an `initial` block is appropriate.

An `assign lo=44;` **is wrong** because `lo` is a var kind and continuous assignments (including `assign`) should only be performed on net kinds, such as `uwire`.

```
  // SOLUTION
  initial lo = 44;
```