Name Solution

Digital Design using HDLs
EE 4755
Final Examination

Monday, 8 December 2014 10:00-12:00 CST

Problem 1		(20 pts)
Problem 2		(20 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Problem 5		(20 pts)
Exam Total		(100 pts)

Alias Not Synthesizable

Good Luck!

Problem 1: [20 pts] The `encode` module below, based on Homework 4, is used to convert a decimal value to binary one ASCII digit at a time. Input `val_prev` is the binary value so far, and output `val_next` is the binary value after using ASCII character `ascii_char`. If `ascii_char` isn't a numeric digit `non_digit` is set to 1 and `val_next` is set to zero. There is also an `overflow` output.

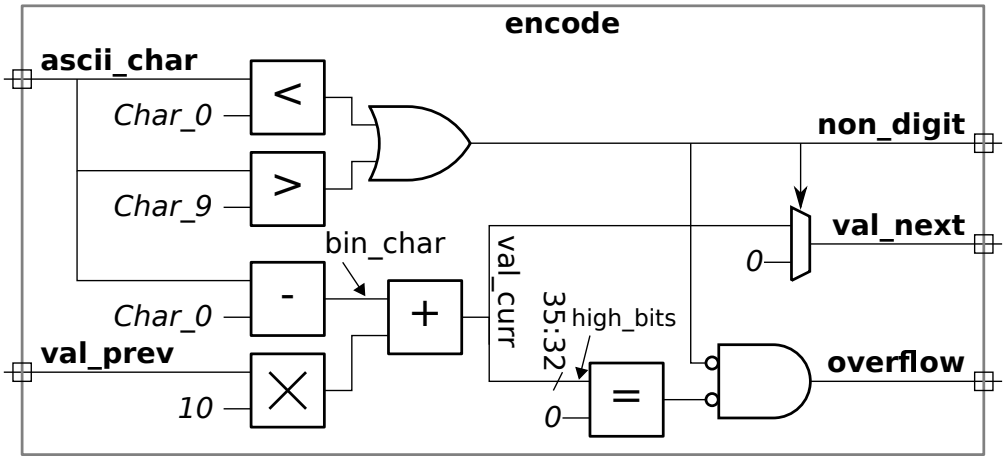
```
module encode
  #( int width = 32 )
   (output logic [width-1:0] val_next,
    output logic overflow,                output uwire non_digit,
    input uwire [7:0] ascii_char,         input uwire [width-1:0] val_prev);

   logic [width+3:0] val_curr;            logic [3:0] high_bits, bin_char;

   assign non_digit = ascii_char < Char_0 || ascii_char > Char_9;

   always_comb begin
      bin_char = ascii_char - Char_0;
      val_curr = 10 * val_prev + bin_char;
      high_bits = val_curr >> width;
      if ( non_digit ) begin    overflow = 0; val_next = 0;       end
      else             begin
        overflow = high_bits != 0;
        val_next = val_curr;
      end
   end
endmodule
```

(*a*) Show the hardware that will be synthesized for this module. Take into account optimizations (see the next subproblem).

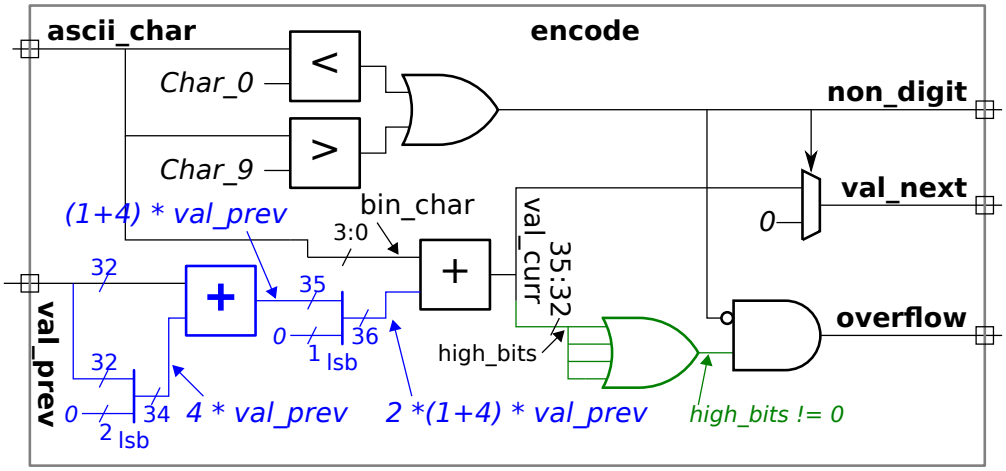
☑ Synthesized hardware.

Two versions of the solution appear below. In the first only basic optimizations are shown. The optimizations shown are for the overflow logic and for the computation of `high_bits`.

In the Verilog `high_bits`, which is four bits, is the result of an expression using a right shift operator. All this does is assign bits 35 to 32 of `val_curr` to `high_bits`, so in the diagram below that is all that is shown. The other basic optimization is logic for `overflow`. Since `overflow` is one bit it makes more sense to use simple gates rather than a multiplexor, and that is what is shown.



The solution below shows further optimizations: the subtractor to compute `bin_char` is eliminated, the times-ten multiplier has been replaced by an adder (that's shown in blue), and the `!=0` operation on `high_bits` is now shown as a four-input OR gate (in green). The replacement for the multiplier uses two constant shifters, they are shown by heavy vertical lines. (The heavy vertical lines indicate, in this case, the grouping together of bits. In this case putting one or two 0's in the LSB position to form a new quantity.)



(*b*) Indicate how many units such as adders, multipliers, shifters, and multiplexors will actually be present in the optimized hardware. The count should be based on the units that are present after optimization, not on the hardware first inferred from the Verilog.

☑ Number of adders. ☑ Number of multipliers. ☑ Number of shifters. ☑ Number of multiplexors.

Adders, 2; multipliers, 0; shifters, 0, number of multiplexors, 1. See the solution to the previous part.

Problem 2: [20 pts] Appearing below is another encode module, this one has a new input `radix`, which indicates the radix (base) of the number to be converted. When completed the module should function like the module from the previous problem, except that the digits form a radix-`radix` number. For example, if `radix` were 10 it would operate like the previous module. If `radix` were 8 the digits would be octal, etc.

(*a*) Modify the module so that it takes into account the radix. Assume that `radix` can be any value from 2 to 16. Note that for a radix of 16 the valid digits are 0-9 and A-F (only consider upper case).

☑ Modify the module to generate the correct `non_digit` output.

☑ Modify the module to update `val_next` correctly given the radix.

Solution appears below. An `is_af` signal is added to detect legitimate hexadecimal digits. A `digit_val` value (value of the current digit) is computed which is correct for radix 2 to 16 (and higher). To detect if the current digit is valid (see `digit_in_range`), the hardware checks if it's in the range 0-9 or A-F. If so, it then looks at the value to make sure that it's less than `radix`.

Grading Notes: In many solutions incorrectly rejected digits in the range 0-9 when `radix` *was greater than 10. A surprisingly large number of solutions used case statements to compute* `non_digit` *with a case for each radix value.*

```
typedef enum {Char_0 = 48, Char_9 = 57, Char_A = 65, Char_F = 70} Chars;
module encode_radix #( int width = 32 )
   (output logic [width-1:0] val_next,
    output logic overflow,                    output uwire non_digit,
    input uwire [7:0] ascii_char,             input uwire [width-1:0] val_prev,
    input uwire [4:0] radix);

   logic [width+3:0] val_curr;

   logic [3:0] high_bits;  // SOLUTION: Remove bin_char, not used.

   // SOLUTION
   uwire   is_digit = ascii_char >= Char_0 && ascii_char <= Char_9;
   uwire   is_af    = ascii_char >= Char_A && ascii_char <= Char_F;
   uwire [3:0] digit_val = ascii_char - ( is_digit ? Char_0 : Char_A - 10 );
   uwire digit_in_range = ( is_digit || is_af ) && digit_val < radix;
   assign non_digit = !digit_in_range;
   always @* begin

      val_curr = radix * val_prev + digit_val; // SOLUTION: Multiply by radix.
      // SOLUTION ends here, text below is unchanged.

      high_bits = val_curr >> width;
      if ( non_digit ) begin
         overflow = 0;
         val_next = 0;
      end else begin
         overflow = high_bits != 0;
         val_next = val_curr;
      end
   end
endmodule
```

Problem 2, continued:

(*b*) Suppose that module `encode_radix` (from the previous part) were to be used in a larger design in which the values of `radix` could only be 2, 8, 10, and 16. Also suppose that the synthesis program can't figure out that `radix` is limited to these values. Why would the cost be higher than necessary, and how could `encode_radix` be modified to get the lower cost hardware?

☑ Explain why the cost will be higher than is necessary.

Short answer: The synthesis program will generate a regular multiplier when all that's really needed are some shifts and an add.

Longer explanation: The Verilog code uses a multiply operator in the expression assigning `val_curr`. For the decimal version of the hardware (from Problem 1 and the Homework assignment) one operand of the multiply is the constant 10, and so the multiplication operator will be synthesized as an adder (computing the sum `val_prev[width-1:1] + val_prev[width-1:3]` which is equivalent to `2 * val_prev[width-1:0] + 8 * val_prev[width-1:0]`). For part a of this problem where `radix` could take on any value a true multiplier had to be synthesized (albeit one in which one input was only four bits).

But in this part we are limiting the radices that are available, so we don't really need a full multiplier. In fact, other than radix 10, all we need to do is shift by a constant amount. The synthesis program could generate a much lower cost design IF it were aware of the limited range of `radix` values, but according to the problem it's not (meaning the synthesis program expects a full range of radix values).

☑ Show the changes to `encode_radix` so that the synthesis program *will* generate the lower cost design. The port definitions **cannot** be changed.

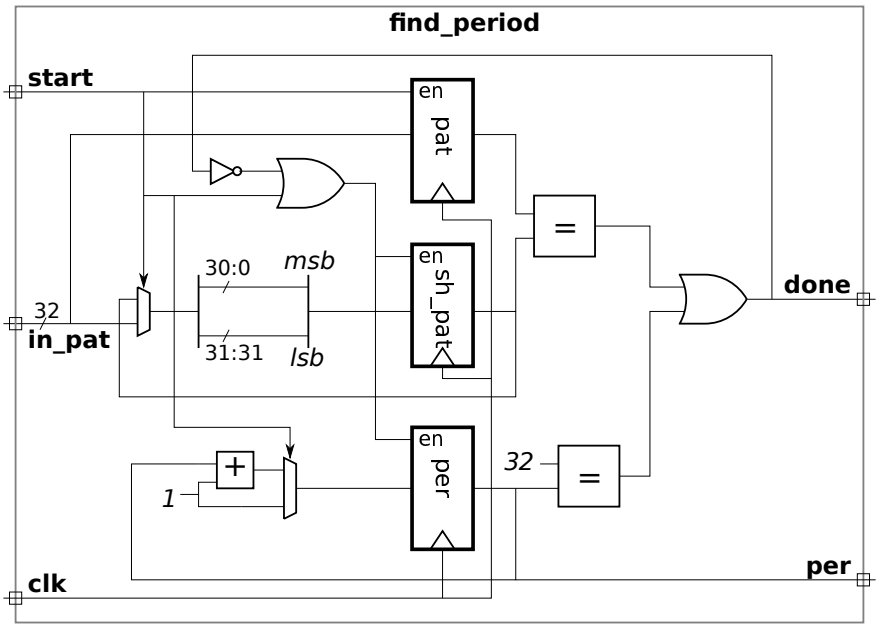
Since the problem is that the synthesis program will generate a real multiplier when we use `radix` with the multiply operator, we won't use radix with the multiplier operator. Instead we'll use a case statement, which is shown below. Notice that a `default` case is included, that's to make sure that a latch is not synthesized for `val_scaled`. Also note that the default case matches one of the other cases, that's to make sure that unused logic is not synthesized.

```
always_comb begin
   case ( radix )
     2: begin
        val_curr = 2 * val_prev + digit_val;
        high_bits = val_curr >> 2;
     end
     8: begin
        val_curr = 8 * val_prev + digit_val;
        high_bits = val_curr >> 8;
     end
     10: begin
        val_curr = 10 * val_prev + digit_val;
        high_bits = val_curr >> 10;
     end
     16: begin
        val_curr = 16 * val_prev + digit_val;
        high_bits = val_curr >> 16;
     end
     default: begin
        val_curr = 10 * val_prev + digit_val;
        high_bits = val_curr >> 10;
     end
   endcase
```

Problem 3: [20 pts] Appearing to the right is hardware and a corresponding Verilog module. The module is incomplete, finish it. *Hint: The hardware includes an end-around shift, that's the part with the msb/lsb labels.*

☑ Add sizes and other information to port declarations.

☑ Finish the Verilog code.



Solution appears below. The size for `sh_pat` is 32 bits since it's connected to the 32-bit end-around shift unit. The size of `pat` is 32 bits since it's connected to a 32-bit input port. The size of `per` is set to six bits based on the comparison with 32 in the diagram. A size of less than six bits could not hold a 32, and anything larger than six bits would not be needed because the value of `per` stops incrementing when it reaches a value of 32.

From the diagram we find three edge-triggered registers, `pat`, `sh_pat`, and `per`. Edge triggered registers are specified in Verilog using `always @ ( posedge clk )` constructs, in this case using a separate `always` block for each register is cleanest.

The `done` output was realized using a continuous assignment. Because all of the values needed for `done` are register outputs, the code for `done` could have been put in an `always` block, but only if there was a single `always` block for all three registers, which is not the case with the solution below.

```
module find_period
   (output logic [5:0]   per,       output uwire  done,
    input uwire [31:0]   in_pat,
    input uwire          start,     input uwire   clk);

   logic [31:0]          pat, sh_pat;

   always_ff @( posedge clk ) if ( start ) pat <= in_pat;

   uwire [31:0] sh_in = start ? in_pat : sh_pat;
   always_ff @( posedge clk )
     if ( start || !done ) sh_pat <= { sh_in[30:0], sh_in[31] };

   always_ff @( posedge clk )
     if ( start )      per <= 1;
     else if ( !done ) per <= per + 1;

   assign done = pat == sh_pat || per == 32;

endmodule
```

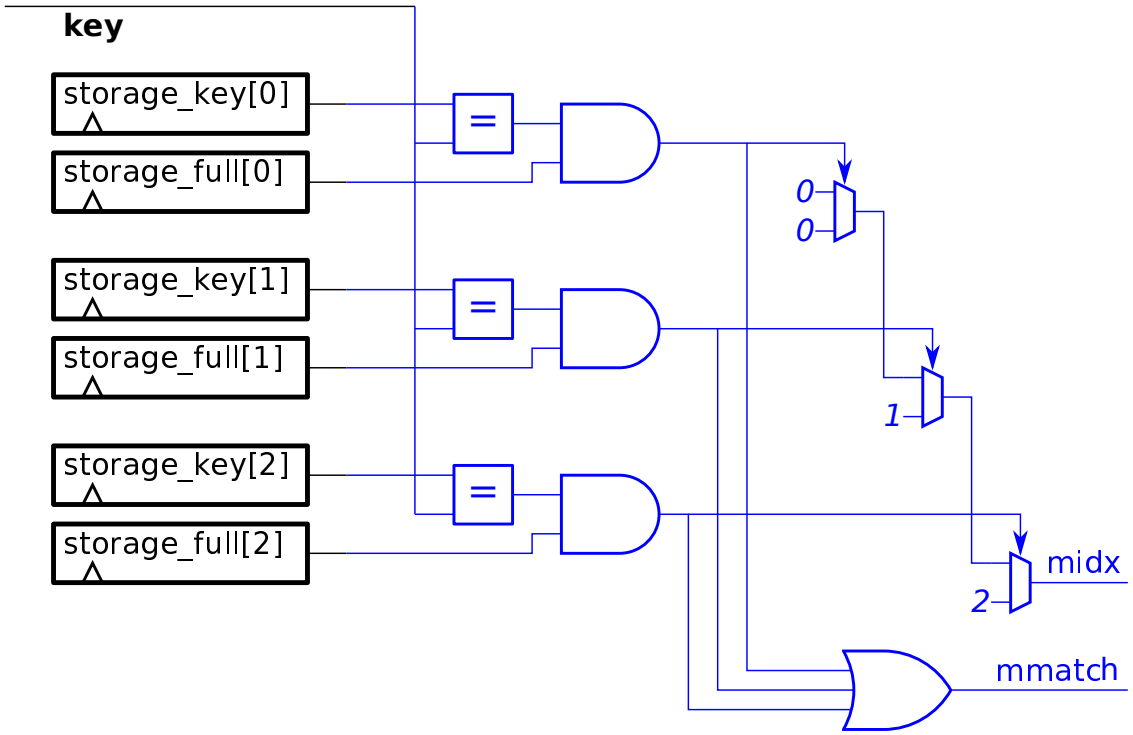
Problem 4: [20 pts] The Verilog below is the key lookup part of the simple CAM module used in class.

```
logic [dwid:1] storage_data [ssize];
logic [kwid:1] storage_key [ssize];
logic [ssize-1:0] storage_full;

always_comb begin
   mmatch = 0;   midx = 0;
   for ( int i=0; i<ssize; i++ )
     if ( storage_full[i] && storage_key[i] == key ) begin mmatch = 1; midx = i; end
end

assign out_data = storage_data[midx];
```

(*a*) Starting with the registers and `key` shown below, sketch the hardware synthesized for this code without optimization. The hardware should produce values for `mmatch` and `midx` (but not `out_data`). Do so for `ssize=3`. In class we often showed part of this as a box labeled "priority encoder" (or "pri" for short), in this problem actually show the hardware.

☑ Synthesized hardware for `ssize = 3` to generate ☑ `mmatch` and ☑ `midx`.

Solution appears below in blue. Note that the first (uppermost) multiplexor can trivially be optimized out since both of its inputs are zero. A chain of multiplexors was chosen to generate `midx`, and a similar chain could have been used for `mmatch`. However the OR gate performs the same operation and is much simpler.



(*b*) Assume that the cost of an a-bit comparison unit is a, and its delay is also a. Assume that the cost of an a-input, b-bit multiplexor is ab and the delay is 1. Compute the cost and delay of the logic used to compute `midx` in terms of `ssize` (use s in your formulas) and `kwid` (use k in your formulas). As with the previous part, do this for the unoptimized hardware. Remember to solve this for an arbitrary value of `ssize` (s), not for $s = 3$.

☑ Cost in terms of s and k:

From the diagram it's clear that there are s k-bit comparison units, they cost sk units. The multiplexors increase in size from the beginning to the end of the chain. The mux at the end of the chain must be large enough to hold the value $s-1$, which requires $\lceil \log_2 s \rceil$ bits. For simplicity assume that all multiplexors are that size, then the multiplexor cost is $2s\lceil \log_2 s \rceil$ units. The s-input OR gate can be assumed to have cost $s-1$ (by setting the cost of a 2-input OR gate at 1).

The $\boxed{\text{total cost is } sk + 2s\lceil \log_2 s \rceil + s - 1 \text{ units}}$.

☑ Delay in terms of s and k:

From the synthesized hardware it should be clear that the critical path used to compute the delay starts at the first comparison unit and continues through the multiplexor chain. The k-bit comparison takes time k, and the length-s multiplexor chain has delay s.

The $\boxed{\text{total delay is } k + s \text{ units}}$.

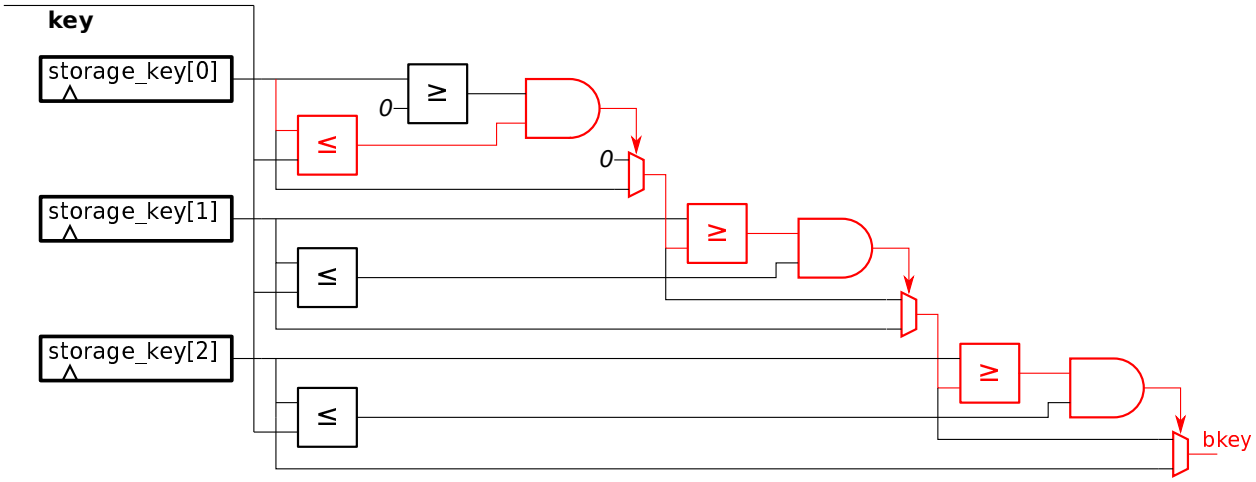
Problem 4, continued: Appearing below is a variation on the key lookup from the CAM module. Instead of finding a matching key it finds the largest stored key that is $\leq$ to the lookup key. Note that this version doesn't include `storage_full`.

```
logic [dwid:1] storage_data [ssize];
logic [kwid:1] storage_key [ssize];

always_comb begin
   midx = 0;  bkey = 0;
   for ( int i=0; i<ssize; i++ )
     if ( storage_key[i] >= bkey && storage_key[i] <= key ) // READ THIS LINE CAREFULLY
       begin   midx = i;  bkey = storage_key[i];   end
end

assign out_data = storage_data[midx];
```

(*c*) Sketch the hardware for `ssize=3`.

☑ Sketch the synthesized hardware needed to generate `bkey`.

Solution appears below, with the critical path shown in red. An important thing to notice is that the $\geq$ comparison at iteration `i` is being made with the value of `bkey` produced in iteration `i-1`. Those values of `bkey` pass through the multiplexor chain, and for that reason the delay in this circuit is significantly longer than in the version from the previous part. See the next sub-part.



(*d*) Compute the cost and performance in terms of `ssize` (use s) and the key size (use k). As before a k-bit comparison unit (equality or magnitude) costs k and has a delay of k and an a-input, b-bit mux costs ab and has a delay of 1. *Hint: There's a big difference.*

☑ Cost in terms of s and k:

There are now $2s$ comparison units, costing $2sk$ cost units. The s multiplexors now carry k-bit values, so their cost is $2sk$. Plus there are s AND gates which we'll set at cost s. The $\boxed{\text{total cost is } 4sk + s \text{ units}}$, which is significantly higher.

☑ Delay in terms of s and k:

In the diagram of the synthesized hardware the critical path appears in red. As before, the critical path passes through the multiplexor chain, but this time the $\geq$ units are also on the critical path. The critical path includes now $s \geq$ units, s muxen, and s AND gates. The $\boxed{\text{total delay is } s(k+2) \text{ units}}$, which is significantly higher than the delay of the first version used in this problem.

Problem 5: [20 pts] Answer each question below.

(*a*) The module below is supposed to count from 0 to `max` (inclusive), then return to zero. Strictly speaking it does, but there are problems, including the fact that it's not synthesizable. Fix the problems.

```
module counter #(int max = 3)(output logic [7:0] count, input uwire clk);

   always @( posedge clk ) begin

      count <= count + 1;

   end

   always @* begin

      if ( count == max ) count <= 0;

   end

endmodule
```

☑ Why isn't the module synthesizable?

It's not synthesizable because `count` is assigned in two different `always` blocks.

☑ Fix the problem.

Just combine the two blocks:

```
module counter #(int max = 3)(output logic [7:0] count, input uwire clk);

   always @( posedge clk ) count <= count == max ? 0 : count + 1;

endmodule
```

(*b*) There is a problem with the module below due to the way that `a` is declared.

```
module sa1(output uwire a, input uwire c, d);

   always_comb begin

      a = c & d;

   end

endmodule
```

The problem is that `a` is being declared as a net type (which includes `uwire`) but it is being assigned in procedural code. Anything assigned in procedural code must be a variable type.

☑ Fix the problem **by changing** the declaration of `a`.

```
module sa1(output logic a, input uwire c, d);
   // SOLUTION: Declare a as a variable type (change uwire a to logic a).

   always_comb begin
      a = c & d;
   end
endmodule
```

☑ Fix the problem **without changing** the declaration of `a`.

```
// SOLUTION
module sa1(output uwire a, input uwire c, d);
   assign a = c & d;
endmodule
```

(*c*) Describe a situation in which using `always_comb` has a benefit over using `always @*`.

☑ Situation where `always_comb` helps.

In the code below `x` is not always assigned and so it could be synthesized into a latch (level-triggered flip-flop). But the SystemVerilog-literate programmer used `always_comb` because he or she intended purely combinational logic—no latches. The fact that `x` was not always assigned was an oversight on the part of the programmer. Because `always_comb` was used well-written Verilog tools will warn the programmer about this. That's how it helps.

```
   always_comb begin
      if ( a < 10 )
        x = a + b;
      else if ( a > 1000 )
        x = a - b;
   end
```

(*d*) The module below is supposed to be computing $x^2 + y^2$.

```
module sa2(output logic [63:0] sos, input uwire [63:0] x, y);

   logic [63:0] a1, b1, a2, b2;
   uwire [63:0]  p, s;

   fpmul f1(p,a1,b1);
   fpadd f2(s,a2,b2);

   always @* begin

      // Compute x^2.
      a1 = x;  b1 = x;
      #1;
      sos = p;

      // Compute y^2.
      a1 = y;  b1 = y;
      #1;

      // Compute x^2 + y^2.
      a2 = p;  b2 = sos;
      #1;
      sos = s;

   end

endmodule
```

☑ Explain why the module is not synthesizable.

It's not synthesizable because it uses delays.

☑ Fix the problem.

The module is trying to use `fpmul` twice. Since there is no clock input, there is no way to do that. A simple solution would be to instantiate a second `fpmul` and connect it appropriately, that's the solution shown below. (A more complex solution would use a `clk` input and use the same multiplier over two cycles.)

```
// SOLUTION
module sa2sol(output uwire [63:0] sos, input uwire [63:0] x, y);

   uwire [63:0]  p, s;

   fpmul fm1(p,x,x);
   fpmul fm2(s,y,y);
   fpadd f2(sos,p,s);

endmodule
```