Solutions to Midterm Exam

Question 1

The two versions of this question used the same expressions arranged in different orders. The following code computes the size and value of each expression:

```
// solutions for midterm exam question 1
module midterm ;

   logic signed [7:0] a = 8'h80 ;
   logic [3:0] b [3:0]  =
       '{ 4'bxxxx, 4'h1, 4'd2, 4'b0011 } ;
   logic [3:0] c       = '1 ;

`define ans(expr) \
   $display("%16s: bits: %2d value: %8b", \
            `"expr`",$bits(expr),expr) ;

   initial begin
      `ans( a+1) ;
      `ans( b[0] ==? 4'bxxx1 ) ;
      `ans( a[0] ? 0 : 2 ) ;
      `ans( {2{a[7:4]}} ) ;
      `ans( &c ) ;
      `ans( a >>> 1 ) ;
      `ans( b[1] ) ;
      `ans( c && a ) ;
      `ans( b[2]==c ) ;
   end

endmodule
```

and the output is:

```
             a+1: bits: 32 value: 1...10000001
b[0] ==? 4'bxxx1: bits:  1 value: 00000001
    a[0] ? 0 : 2: bits: 32 value: 00000010
     {2{a[7:4]}}: bits:  8 value: 10001000
              &c: bits:  1 value: 00000001
         a >>> 1: bits:  8 value: 11000000
            b[1]: bits:  4 value: 00000010
          c && a: bits:  1 value: 00000001
         b[2]==c: bits:  1 value: 00000000
```

Question 2

(a) There were two versions of the question. In both the clock signal is initialized to 0 and the simulation runs for four delays of 1 μs (4 μs total). There are two rising clock edges (at 1 and 3 μs).

(b) In the first version of the question x is initialized to 1 and is incremented by 2 on each rising clock edge; in the second it is initialized to 0 and incremented by 1. So x takes on three values – the initial value and one after each clock edge – and the values of x are 1, 3, 5 (first version) or 0, 1, 2 (second version).

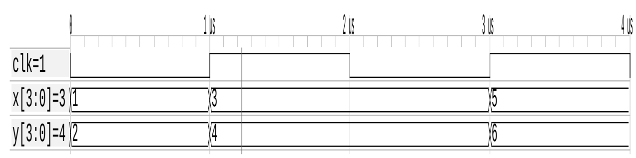
(c) The `$display()` function in the `always_ff` bock executes immediately after a non-blocking assignment so the value of `x` will not have been updated yet. So the first version prints:

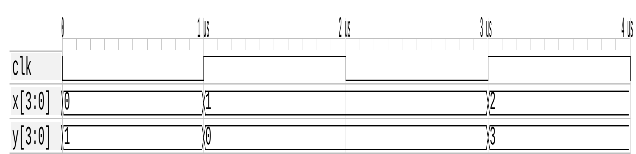
```
#  1
#  3
```

while the second version prints:

```
#  0
#  1
```

(d) The `always_comb` block executes and updates `y` whenever `x` changes. The value of `y` is set to `x+1` if the LS bit of `x` is set (`x` is odd) else to zero. So in the first version `y` is set to (2,4,6); in the second it is set to (1,0,3) as shown below:





Question 3

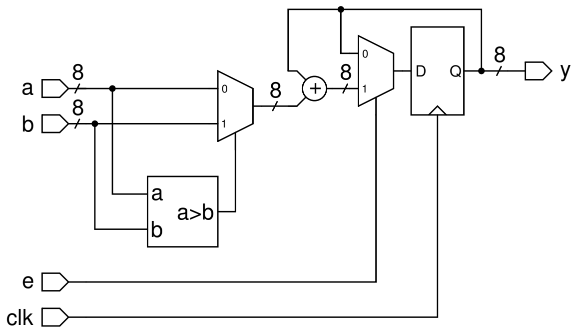
To convert a schematic to HDL, we:

- list each input or output in a module statement
- assign to each register or flip-flop output using a non-blocking assignment in an `always_ff` block

- model each multiplexer as an if/else in an `always_comb` block, using local signal names as necessary
- model combination logic blocks as operators

It's often possible to simplify the solution using continuous `assign` statements or the ternary operator.

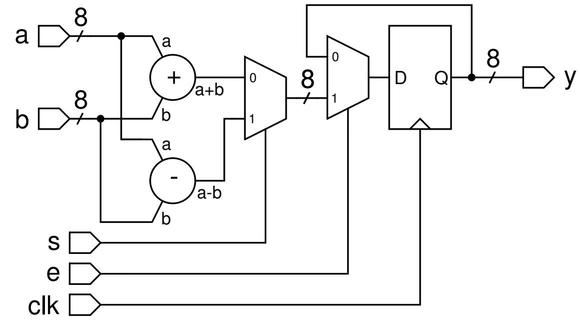
For the first schematic:



examples of concise and more verbose System Verilog solutions are:

```
module midterm3a
  ( input  logic [7:0] a, b,
    output logic [7:0] y,
    input  logic e, clk ) ;

`ifdef simple

   always_ff@(posedge clk)
     y <= e ? y + ( a>b? b : a ) : y ;

`else

   logic [7:0] y_next, y_sum ;

   always_ff@(posedge clk)
     y <= y_next ;

   always_comb
     if ( e )
       y_next = y_sum ;
     else
       y_next = y ;

   always_comb
     if ( a > b )
       y_sum = b ;
     else
       y_sum = a ;

`endif

endmodule
```

and for the second schematic:



examples of the System Verilog code are:

```
module midterm3b
  ( input logic [7:0] a, b,
    output logic [7:0] y,
    input  logic s, e, clk );

`ifdef simple

   always_ff@(posedge clk)
     y <= e ? ( s ? a-b : a+b ) : y ;

`else

   logic [7:0] y_next, y_sumdiff ;

   always_ff@(posedge clk)
     y <= y_next ;

   always_comb
     if ( e )
       y_next = y_sumdiff ;
     else
       y_next = y ;

   always_comb
     if ( s )
       y_sumdiff = a - b ;
     else
       y_sumdiff = a + b ;
`endif

endmodule
```

Question 4

There were two System Verilog modules. Drawing the schematic involves drawing:

- an I/O connector for each input or output in the module statement
- a register or flip-flop for each `always_ff` block

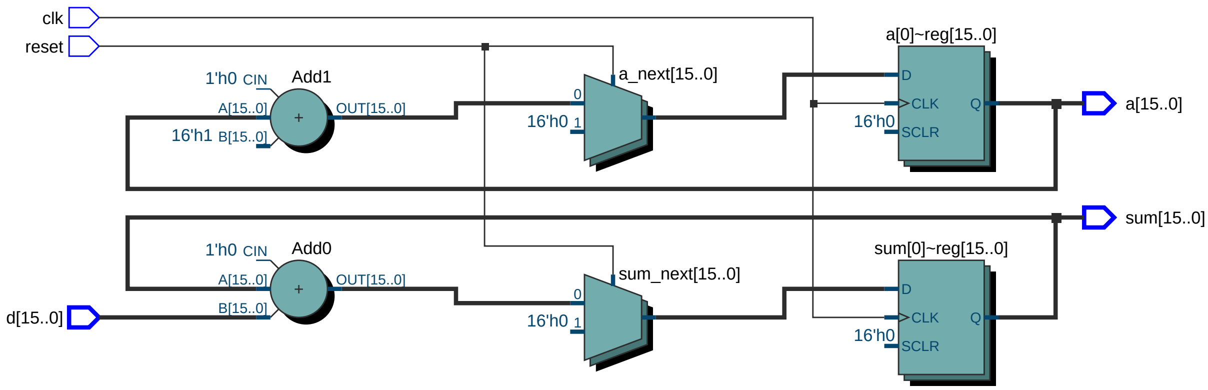


Figure 1: `mksum` schematic.

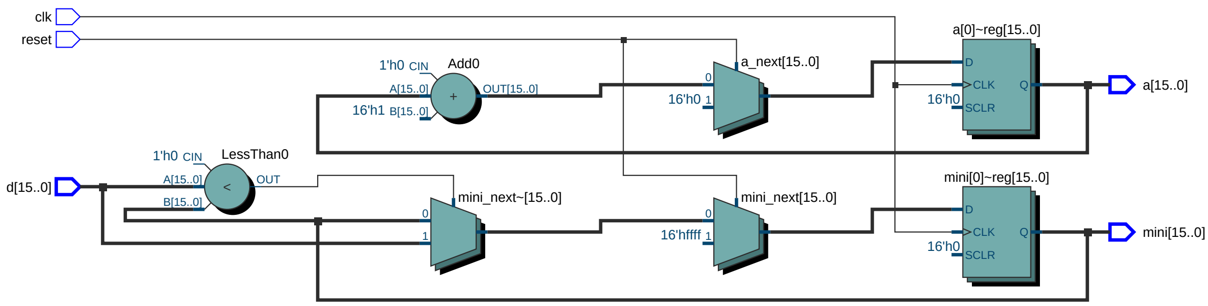


Figure 2: `mkmini` schematic.

- a multiplexer for each if/else or ternary operator
- a combinational logic block for each operator in an expression

Figure 1 shows the Quartus-generated schematic for the `mksum` module and Figure 2 shows the Quartus-generated schematic for the `mkmini` module.