ECE 351 Test #1 Study Guide

Your test is scheduled for Thursday, 25 Apr. It is a one-hour open book/ open note, multiple choice test.

The test will cover the following topics:

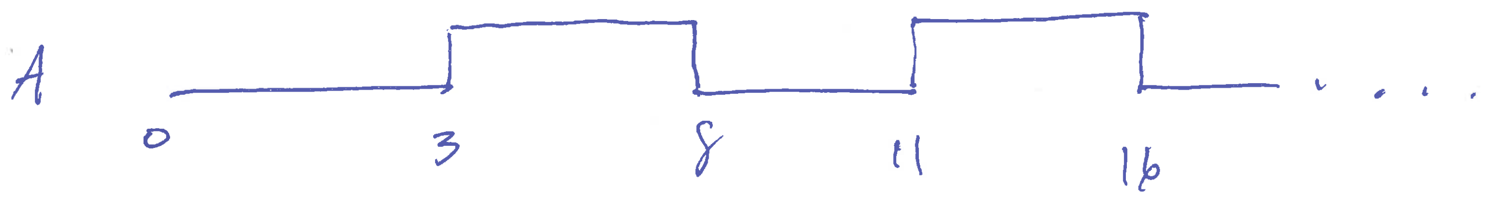
- Verilog HDL syntax/semantics. In particular you should be able to answer questions about Verilog programs that appear on the exam
- Simulator/synthesizer concepts (i.e., concepts covered in the ASIC application note)
- Basic structures of Verilog modules
- Verilog levels of abstraction (nothing on behavioral)
- module templates and instantiation

NOTES:

- You must bring a SCANTRON 882E to class to record your answers. These can be purchased at the PSU bookstore.
- You may have any electronic device capable of reading PDFs
- There will be a short lecture prior to the test.

Timing controls

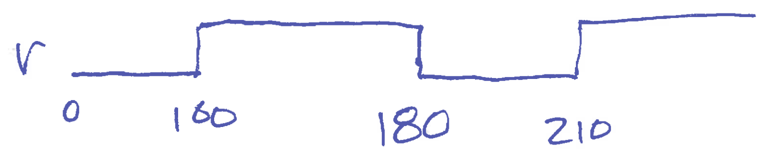
```
parameter on-delay = 3, off_delay = 5;
reg A;

   initial
      A = 0;

   always
     begin
        # on-delay  A = 1;
     end # off_delay A = 0;
```



```
a = #2  b;        or   #2 a = b;

                       hold = b;
                        #2;
                          a = hold;
```

Case 1

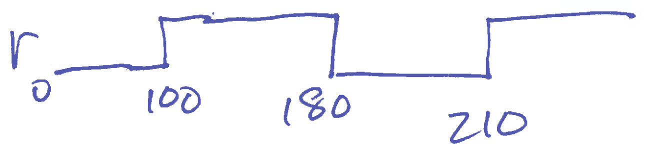
```
   initial
     begin
       r = 0;
#100   r = 1;
#80    r = 0;
#30    r = 1;
     end
```



case 2

```
   initial
     begin
       r=0;
       r= #100 1;
       r= #80 0;
       r= #30 1;
     end
```

additive



In Both cases, the delays are relative

Writing testbenches

syntax:

```
module module_name; // no port list
Local reg/net variable declarations
instantiate the modules to be tested
initial/alway statement(s) to generate
        the test patterns
endmodule
```

Question: Blocking or non-blocking assignments?

Blocking assignments

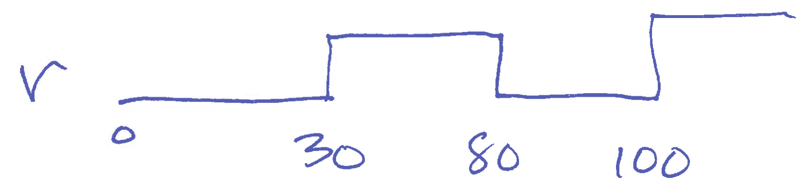
```
initial
    clock=0;

always ~~block~~
    #10 clock = ~clock;
```

what about

```
initial
    clock=0;
always
    clock <= #10 ~clock;
```

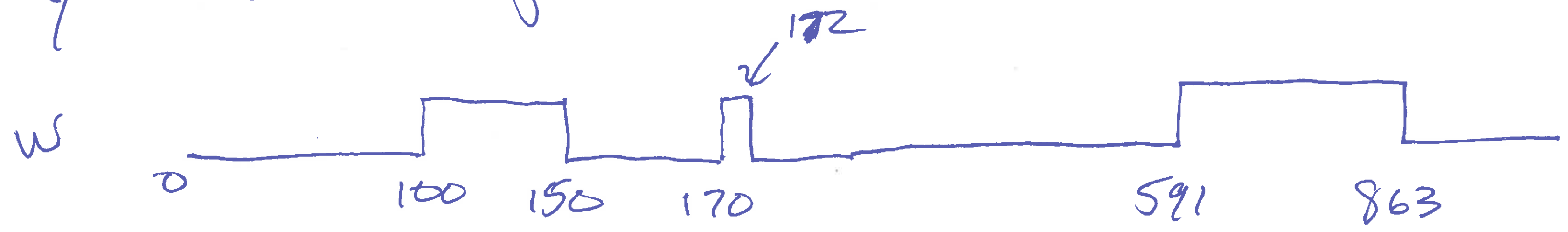
Case 3

```
Initial
 begin
      r <= 0;
      r <= #100 1;
      r <= #80  0;
      r <= #30  1;
 end
```



delays are absolute

you want to generate



Note: <= with RHS delays is preferred for non-blocking assignments, does order matter?

In Summary

- blocking assignments

  #delay LHS = RHS;

  or

  LHS = #delay RHS;

  #delays are relative to each other.

- non-blocking assignments

  LHS <= #delay RHS;

  #delays are absolute times

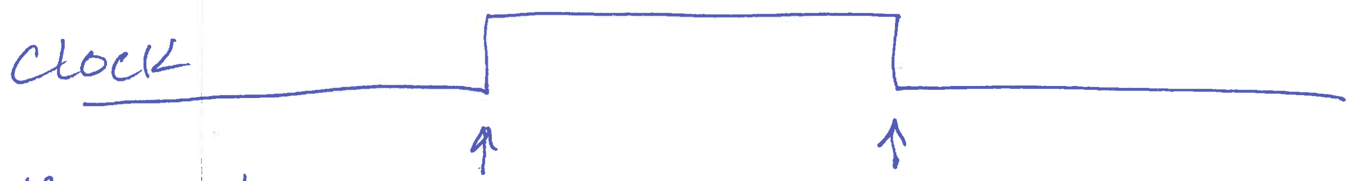
<u>event control</u>

"@" denotes event control

Two types of control

- level
- edge triggered

edge <u>triggered</u> events



Keyword:	posedge	negedge
	$\Downarrow$	
	$0 \to 1$	$1 \to 0$
	$0 \to x$	$1 \to z$
	$0 \to z$	$1 \to x$
	$x \to 1$	$x \to 0$
	$z \to 1$	$z \to 0$

Example 7-13 *Regular Event Control*

```
@(clock) q = d; //q = d is executed whenever signal clock changes value
@(posedge clock) q = d; //q = d is executed whenever signal clock does
                        //a positive transition ( 0 to 1,x or z,
                        // x to 1, z to 1 )
@(negedge clock) q = d; //q = d is executed whenever signal clock does
                        //a negative transition ( 1 to 0,x or z,
                        //x to 0, z to 0)
q = @(posedge clock) d; //d is evaluated immediately and assigned
                        //to q at the positive edge of clock
```

you can 'OR' events

<u>e.g.</u>

keyword

```
@ (posedge A or negedge B)
```

Example 7-15 *Event OR Control (Sensitivity List)*

```
//A level-sensitive latch with asynchronous reset
always @( reset or clock or d)
                                //Wait for reset or clock or d to change
begin
        if (reset)               //if reset signal is high, set q to 0.
                q = 1'b0;
        else    if(clock)       //if clock is high, latch input
                q = d;
end
```

@*
@(*)

Example 7-17 *Use of @* Operator*

```
//Combination logic block using the or operator
//Cumbersome to write and it is easy to miss one input to the block
always @(a or b or c or d or e or f or g or h or p or m)

begin
out1 = a ? b+c : d+e;
out2 = f ? g+h : p+m;
end

//Instead of the above method, use @(*) symbol
//Alternately, the @* symbol can be used
//All input variables are automatically included in the
//sensitivity list.
always @(*)
begin
out1 = a ? b+c : d+e;
out2 = f ? g+h : p+m;
end
```

```
//Type 1 conditional statement. No else statement.
//Statement executes or does not execute.
if (<expression>) true_statement ;

//Type 2 conditional statement. One else statement
//Either true_statement or false_statement is evaluated
if (<expression>) true_statement ; else false_statement ;

//Type 3 conditional statement. Nested if-else-if.
//Choice of multiple statements. Only one is executed.
if (<expression1>) true_statement1 ;
else if (<expression2>) true_statement2 ;
else if (<expression3>) true_statement3 ;
else default_statement ;
```

Pg 137

Example 7-18 *Conditional Statement Examples*

```
//Type 1 statements
if(!lock) buffer = data;
if(enable) out = in;

//Type 2 statements
if (number_queued < MAX_Q_DEPTH)
begin
        data_queue = data;
        number_queued = number_queued + 1;
end
else
        $display("Queue Full. Try again");

//Type 3 statements
//Execute statements based on ALU control signal.
if (alu_control == 0)
        y = x + z;
else if(alu_control == 1)
        y = x - z;
else if(alu_control == 2)
        y = x * z;
else
        $display("Invalid ALU control signal");
```

in C

```
switch (   )
  case 1: ______
  case 2: ______
     :
     :
  default: ______
```

---

in Verilog

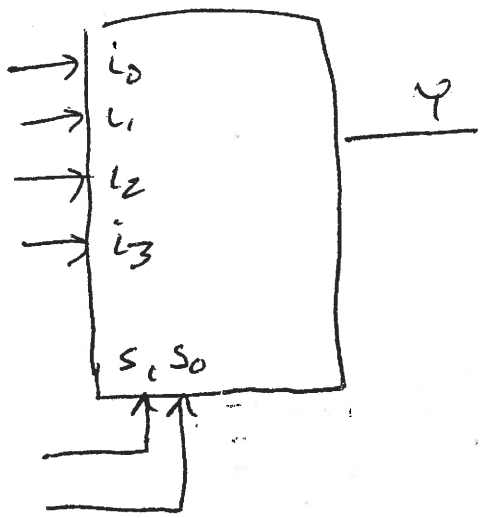
case statement

~~in C~~ Switch statements

Keywords

```
case (expression)
     alternative1: statement1;
     alternative2: statement2;
     alternative3: statement3;
         ...
         ...
     default: default_statement;
endcase
```

4 x 1 Mux



S_1 S_0	Y
0 0	i_0
0 1	i_1
1 0	i_2
1 1	i_3

Example 7-19 *4-to-1 Multiplexer with Case Statement*

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;

always @(s1 or s0 or i0 or i1 or i2 or i3)
case ({s1, s0}) //Switch based on concatenation of control signals
        2'd0 : out = i0;
        2'd1 : out = i1;
        2'd2 : out = i2;
        2'd3 : out = i3;
        default: $display("Invalid control signals");
endcase

endmodule
```

when you use

'case'	—	everything must match exactly
'casez'	—	any 'z' bits are ignored
'casex'	—	any 'z' or 'x' bits are $\underbrace{\text{ignored}}_{\text{i.e. not compared}}$

↑

all use

☆ endcase ☆

e.g. $A = 2'b11;$

```
case (A)
  2'bx1 : B=0;
  default : B=1;
endcase
```

$\boxed{B=1}$

$(2'b11 \neq 2'bx1)$

```
casex (A)
  2'bx1 : B=0
  default : B=1
endcase
```

$\boxed{B=0}$

(LSBs match)

Example 7-21 casex Use

```
reg [3:0] encoding;
integer state;

casex (encoding) //logic value x represents a don't care bit.
4'b1xxx : next_state = 3;
4'bx1xx : next_state = 2;
4'bxx1x : next_state = 1;
4'bxxx1 : next_state = 0;
default : next_state = 0;
endcase
```

Thus, an input encoding = 4'b10xz would cause next_state = 3 to be executed.

(first one that matches)

Example 7-20 *Case Statement with x and z*

```
module demultiplexer1_to_4 (out0, out1, out2, out3, in, s1, s0);

// Port declarations from the I/O diagram
output out0, out1, out2, out3;
reg  out0,  out1,  out2,  out3;
input in;  // data in
input s1, s0; // select lines

always @(s1 or s0 or in)
case ({s1, s0}) //Switch based on control signals
    2'b00 : begin  out0 = in;  out1 = 1'bz;  out2 = 1'bz; out3 = 1'bz; end
    2'b01 : begin  out0 = 1'bz;  out1 = in;  out2 = 1'bz; out3 = 1'bz; end
    2'b10 : begin  out0 = 1'bz;  out1 = 1'bz;  out2 = in; out3 = 1'bz; end
    2'b11 : begin  out0 = 1'bz;  out1 = 1'bz;  out2 = 1'bz; out3 = in; end

    //Account for unknown signals on select. If any select signal is x
    //then outputs are x. If any select signal is z, outputs are z.
    //If one is x and the other is z, x gets higher priority.
    2'bx0, 2'bx1, 2'bxz, 2'bxx, 2'b0x, 2'b1x, 2'bzx :
        begin
              out0 = 1'bx;  out1 = 1'bx;  out2 = 1'bx;  out3 = 1'bx;
        end
    2'bz0, 2'bz1, 2'bzz, 2'b0z, 2'b1z :
        begin
              out0 = 1'bz;  out1 = 1'bz;  out2 = 1'bz;  out3 = 1'bz;
        end
    default: $display("Unspecified control signals");
endcase

endmodule
```

Loops

4 types
- while
- for
- repeat
- forever

---

in C

```
do
{
    ≡
}while(expr);
```

in Verilog

```
while (expr)
  begin
    ≡
  end
```

Example 7-22 While Loop

```
//Illustration 1: Increment count from 0 to 127. Exit at count 128.
//Display the count variable.
integer count;

initial
begin
        count = 0;

        while (count < 128) //Execute loop till count is 127.
                            //exit at count 128
        begin
                $display("Count = %d", count);
               count = count + 1;
        end
end


//Illustration 2: Find the first bit with a value 1 in flag (vector
variable)
'define TRUE 1'b1';
'define FALSE 1'b0;
reg [15:0] flag;
integer i; //integer to keep count
reg continue;

initial
begin
  flag = 16'b 0010_0000_0000_0000;
  i = 0;
  continue = 'TRUE;

  while((i < 16) && continue ) //Multiple conditions using operators.
  begin
    if (flag[i])
    begin
      $display("Encountered a TRUE bit at element number %d", i);
      continue = 'FALSE;
    end
    i = i + 1;
  end
end
```

Example 7-23 *For Loop*

```
integer count;

initial
   for ( count=0; count < 128; count = count + 1)
         $display("Count = %d", count);
```