

EE 421 / CS 425 Digital System Design Laboratory 5

Fall 2023
Shahid Masud

Today's Topics

- Behavior Description of Digital Systems
- Use **reg** data type
- Now we can use **always, case, switch, if then else, do while,** constructs, etc.
- **Blocking** and **Non-Blocking** Statements

Design Capture in Verilog HDL

Verilog Allows Design Capture at Various Hierarchy Levels:

1. Switch Level (NMOS and PMOS transistors)
2. Gate Level (Describing Circuit as Logic Gates)
 - a) Data Flow (RTL Sequential) Level **TO DO**
3. Dataflow (Combinational) ✓
4. Behavior Level **Today**

The Design is **Captured** in a text file or obtained from schematic diagram

always statement (infer combinational)

As the name suggests, an **always** block executes always

Unlike **initial** blocks which execute only once (at the start of simulation)

A second difference is that an **always** block should have a **sensitivity list** associated with it

The **sensitivity list** is the one which tells the **always** block when to execute the block of code

The **@** symbol after reserved word '**always**', indicates the condition in parenthesis after symbol **@**

```
reg y;  
wire a, b, sel;  
  
always @ (a or b or sel)  
begin  
    y = 0;  
    if (sel == 0) y = a;  
    else          y = b;  
end
```

The ports **a, b, sel** are of type **wire**

The output from always blocks is **y** of type **reg**

always statement (infer sequential)

There are two types of **sensitivity list**:

level-sensitive (for combinational circuits) and

edge-sensitive (for flip-flops)

The code below is the same 2×1 MUX but the output **y** is now a **flip-flop** output

```
always @ (*) // * means all the variables are in sensitivity list
// This can only be used in combinational always block
```

```
@(posedge clk)      - At the positive edge of clk
@(signal or signal) - Any change in listed signals
@*                  - Any change to any signal used as an input to the block
```

```
always @ (posedge clk) //can be posedge or negedge
begin
    if (reset == 0) y <= 0;
    else if (sel == 0) y <= a;
    else    y <= b;
end
```

y must be type reg

always statement (with delay)

This is used in **testbench**

We can have an **always** block without sensitivity list but with a **delay**

```
always  
begin  
    #5 clk = ~clk;  
end
```

Blocking assignment

Blocking assignment (=)

Evaluation and assignment are immediate

Use Blocking assignment while designing combinational circuits

```
always @ (*) /* means re-evaluate combinational on any variable change
begin
    x = a | b;      // 1. evaluate [a | b], assign result to x
    y = a ^ b ^ c;  // 2. evaluate [a ^ b ^ c], assign result to y
    z = b & ~ c;    // 3. evaluate [b & (~c)], assign result to z
end
```

Non-Blocking assignment

Non-blocking assignment (\leq)


All assignments deferred to end of simulation time step
after RHS of all expressions have been evaluated

```
always @ (a or b or c) //re-evaluate when a or b or c changes
begin
  x <= a | b;           // 1. evaluate [a | b], but defer assignment to x
  y <= a ^ b ^ c;       // 2. evaluate [a ^ b ^ c], but defer assignment to y
  z <= b & ~c;          // 3. evaluate [b & (~c)], but defer assignment to z
                        // 4. end of time step: assign new values to x, y, and z
end
```


Compare Blocking and Non-Blocking assignment

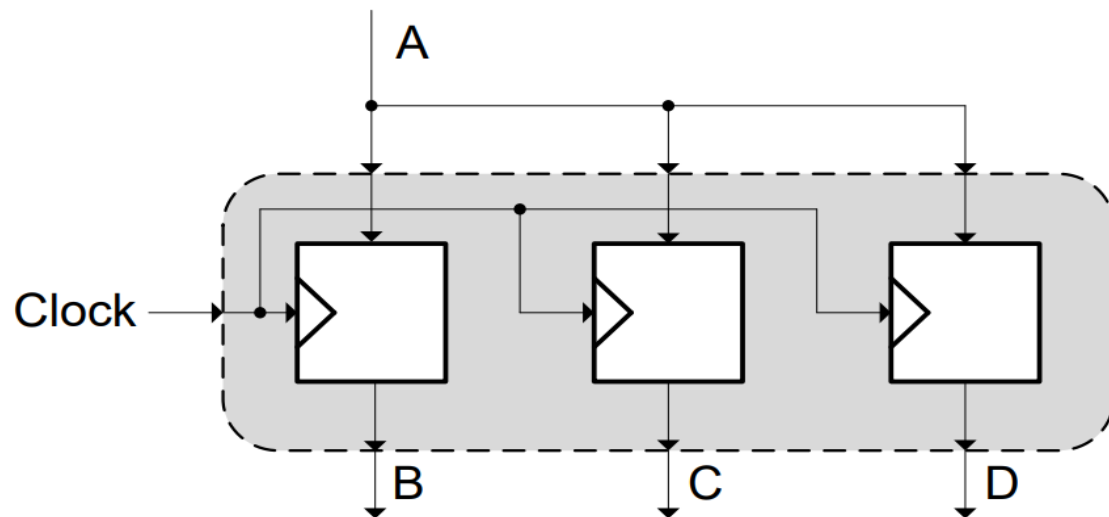
Use non-blocking statements while designing sequential circuits.

```
always @ (posedge clk)
begin
    q1 <= in;
    q2 <= q1; // uses old q1
    out <= q2; // uses old q2
end
//-----
always @ (posedge clk)
begin
    q1 = in;
    q2 = q1; // uses new q1
    out = q2; // uses new q2
end
```



What type of circuit will we get?

```
1 always @(posedge Clock) begin
2     B = A;
3     C = B;
4     D = C;
5 end
```



Three Parallel Registers

```
always @(posedge clk)
begin
    B <= A;
    C <= B;
    D <= C;
end
```

What will we get?

Hint (Shift Registers?)

Use of **case** statement

The **case** statement compares **0**, **1**, **x**, and **z** values in the expression and the alternative
The Bit-widths of signals should match for correct evaluation

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);  
output out;  
input i0, i1, i2, i3;  
input s1, s0;  
reg out; //type reg  
always @ (s1 or s0 or i0 or i1 or i2 or i3) //sensitivity list  
    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 signal");  
    endcase  
endmodule
```

Use of **while** loop statements

The keyword **while** is used to specify this loop

The **while** loop executes until the while-expression becomes false

If the loop is entered when the while-expression is false, the loop is not executed at all
multiple statements are grouped typically using keywords **begin** and **end**

```
begin
    count = 0;
    while (count < 128) // execute loop till count is 127, exits @ 128
    begin
        $display ("count = %d", count);
        count = count+1;
    end
end
```

Using **for** loop in Verilog

The keyword **for** is used to specify this loop. The **for** loop contains three parts:

- An initial condition
- A check to see if the terminating condition is true
- A procedural assignment to change value of the control variable

for loops are generally used when there is a fixed beginning and end to the loop

If the loop is simply looping on a certain condition, it is better to use the **while** loop

```
begin
  for (i = 0; i < 32; i = i + 2) // initialize all even location with 0
    state[i] = 0;
  for (i = 1; i < 32; i = i + 2) // initialize all odd location with 1
    state[i] = 1;
end
```

Using **repeat** loop

The keyword **repeat** is used for this loop

The **repeat** construct executes the loop a fixed number of times

A **repeat** construct cannot be used to loop on a general logical expression,
a **while** loop is used for that purpose

A **repeat** construct must contain a **number**, which can be a **constant**, a **variable**, or a **signal value**.

However, if the number is a **variable** or **signal value**,
it is evaluated only when the loop starts and not during the loop execution

```
repeat (128)
begin
    $display ("count = %d", count);
    count = count+1;
end
```

\$display is for simulation

Use of **forever** loop

Used typically in **testbench** to generate periodic signals

The keyword **forever** is used to express this loop

The loop does not contain any expression and executes **forever** until the **\$finish** is encountered

A **forever** loop can be exited by use of the **disable** statement

A **forever** loop is typically used in conjunction with **timing control constructs**

```
// Clock generation
// Use forever loop instead of always block
reg clock; //clock is type reg
initial
begin
    clock = 1'b0;
    forever #10 clock = ~clock; // clock with period of 20 units
end
```

Deliverables for Lab 5