DIGITAL DESIGN WITH HDL

Chapter 5: Behavioral model

Behavioral model

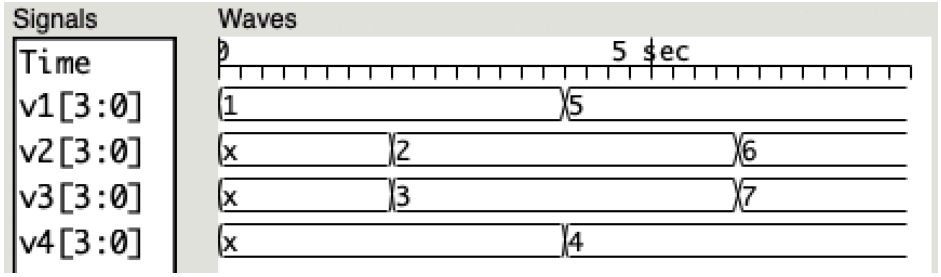
- Use procedural blocks: initial, always
- These blocks contain series of statements
  - Abstract – works *somewhat* like software
  - Be careful to still remember it's hardware!
- Parallel operation **across** blocks
  - All blocks in a module operate simultaneously
- **Sequential** or **parallel** operation within blocks
  - Depends on the way the block is written
  - Discuss this in a later lecture
- LHS of assignments are variables (reg)

Procedural blocks

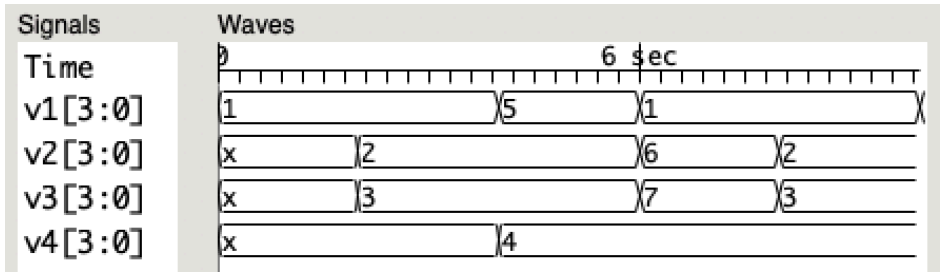
- initial
  - Behavioral block operates **ONCE**
  - Starts at time 0 (beginning of operation)
  - Useful for test benches
  - Can sometimes provide initialization of memories/FFs
    - Often better to use “reset” signal
  - Inappropriate for combinational logic
  - Usually **cannot be synthesized**
- always
  - Behavioral block operates **CONTINUOUSLY**
  - Can use a trigger list to control operation; @(a, b, c)

initial vs. always

```
reg [3:0] v1, v2, v3, v4;
initial begin
        v1 = 1;
    #2 v2 = v1 + 1;
        v3 = v2 + 1;
    #2 v4 = v3 + 1;
        v1 = v4 + 1;
    #2 v2 = v1 + 1;
        v3 = v2 + 1;
end
```




```
reg [3:0] v1, v2, v3, v4;
always begin
        v1 = 1;
    #2 v2 = v1 + 1;
        v3 = v2 + 1;
    #2 v4 = v3 + 1;
        v1 = v4 + 1;
    #2 v2 = v1 + 1;
        v3 = v2 + 1;
end
```



always blocks

- General syntax:

```
always [<trigger list>] begin
        <Procedural statements>
end
```

- Operates **continuously** or on a **trigger list** (sensitive list/event control list)
- Can be used with initial blocks or other always blocks
  - Unlimited in a module
  - All are executed in parallel
- Cannot "nest" initial or always blocks

Trigger lists

- Continuously check the trigger list in an always block to trigger the block
- Start with an event control operator: @
- Contain signals, not expressions
  - Scalar or vector
  - Net or register
- Type of trigger lists
  - **Level trigger**: combinational circuits/blocks
  - **Edge trigger**: sequential circuits/blocks
  - One module can include both

Level trigger

- Syntax:
  - Old style: @(<signal 1> or <signal 2> or … or <signal n>)
  - New style: @(<signal 1>, <signal 2>,…, <signal n>)
- Operations:
  - When any signal changes, statements in the block are executed
  - Otherwise, keep the values of LHS unchanged
- **Special case**: @(*)
  - When all signals in RHS and conditional statements are in the trigger list

Examples

```
always @(x1 or x2 or x3)
  z1 = #5 (x1 & x2 & x3);
```

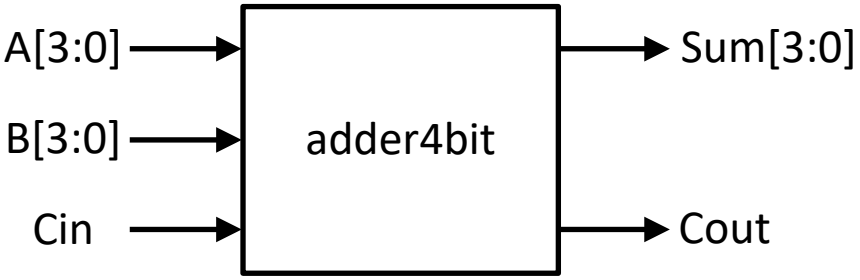
```
always @(in1, in0, sel) begin
  if (sel == 1'b0) out = in0;
  else out = in1;
end
```

```
always @ (a or b or cin) begin
  sum = a + b + cin;
  cout = (a[3] & b[3]) |
         ((a[3] | b[3]) & (a[2] & b[2])) |
         ((a[3] | b[3]) & (a[2] | b[2]) & (a[1] & b[1])) |
         ((a[3] | b[3]) & (a[2] | b[2]) & (a[1] | b[1])
          & (a[0] & b[0])) |
         ((a[3] | b[3]) & (a[2] | b[2]) & (a[1] | b[1])
          & (a[0] | b[0]) & cin);
end
```

```
always @(a, b, c) begin
  a1 = a & b;
  a2 = b & c;
  a3 = a & c;
  carry = a1 | a2 | a3;
end
```

Example

- Design and implement an adder-4bit




```
module  adder4b (Sum, Cout, A, B, Cin);
  input    [3:0] A, B;
  input    Cin;
  output reg [3:0] Sum;
  output reg Cout;
  always @(A, B, Cin)
       {Cout,Sum} = A + B + Cin;
endmodule
```

Edge trigger

- Used to detect edges of signals
  - Rising edge: posedge
    - $0, x, z \rightarrow 1$
  - Falling edge: negedge
    - $1, x, z \rightarrow 0$
- Syntax:
  - Old style: @(posedge/negedge <signal 1> or posedge/negedge <signal 2> or … or posedge/negedge <signal n>)
  - New style: @(posedge/negedge <signal 1>, posedge/negedge <signal 2>, …, posedge/negedge <signal n>)
- Operations:
  - When **any signal in the list changes accordingly to the edge detect operator**, statements in the block are executed
  - Otherwise, all LHS variables are kept unchanged

Examples

```
always @(posedge clk)
   q <= d;
```

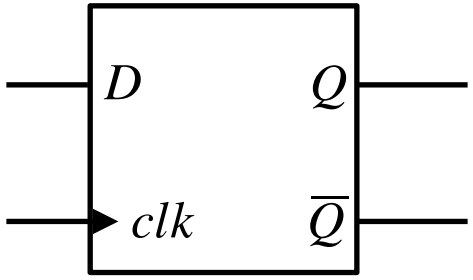
```
always @(posedge clk or negedge rst_n) begin
  if (rst_n == 0)
     count <= 4'b0000;
  else
     count <= (count + 1) % 16;
end
```

```
always @(posedge clk, posedge rst_n) begin
  if (rst_n == 1)
     y <= 4'b0000;
  else
     y <= d;
end
```

```
always @(posedge clk) begin
  if (rst_n == 0)
     q <= 1'b0;
  else
     q <= d;
end
```

Example

- Design and implement a simple D flip flop



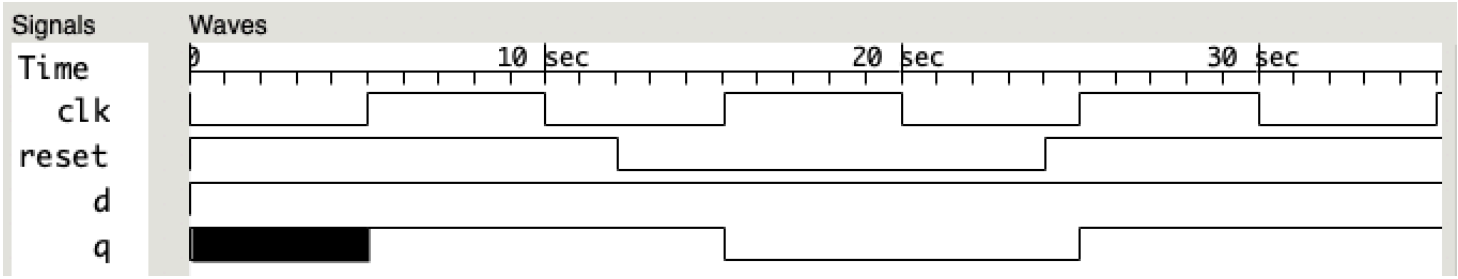

```
module dff(D, clk, Q, Qbar);
    input D, clk;
    output reg Q, Qbar;
    always @(posedge clk) begin
        Q <= D;
        Qbar <= ~D;
    end
```

Control vs. Clock signals

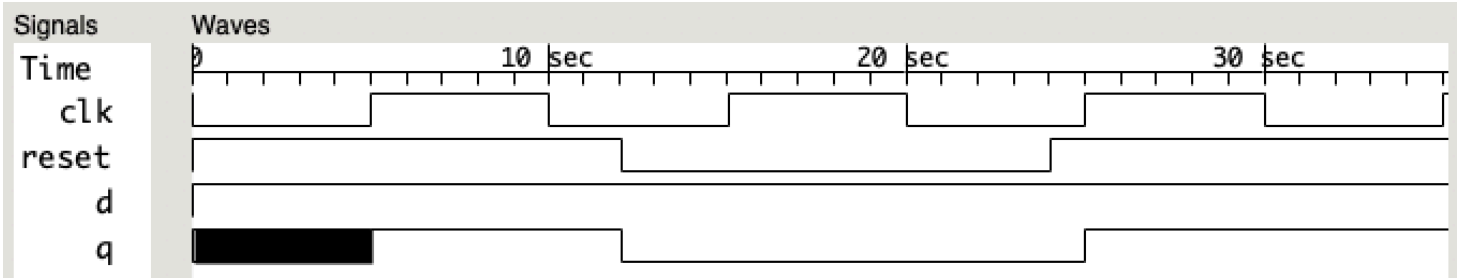
- Clock signal: **synchronize** behaviors of a module
  - Always included in trigger lists
  - No need to check inside the block
  - Rising or falling edges
- Control signal: **handle** operations of a module
  - E.g., reset/clear or set/preset signals
  - Sometimes included in trigger lists; sometimes not
  - Always checked inside the block
  - Active low or active high

Control signals

- Synchronous control signals
  - Need to **wait for clock edges** to handle the module



- Asynchronous control signals
  - Handle the module **right after active**



Control signals in Verilog

- Synchronous control signals
  - **Do not include** in trigger lists
  - Active time should be at least **1 cycle**
- Asynchronous control signals
  - **Must include** in trigger lists, need posedge/negedge
    - Active low: negedge
    - Active high: posedge
  - Active time can be arbitrary

```
always @(posedge clk)
    if (reset == 0) q <= 0;
    else q <= d;
```

```
always @(posedge clk, negedge reset)
    if (reset == 0) q <= 0;
    else q <= d;
```

What happen when there is no "negedge" is associated with the reset signal?

Procedural assignments

- A statement that **assigns values to register variables** is called a procedural assignment
  - LHS: register, bit-select, part-select, or concatenation
    - To assign to output: `output reg [3:0] a;`
  - Take place within an `initial` block or an `always` block
  - RHS is evaluated and **scheduled to be assigned** to LHS at a time specified by an optional timing control

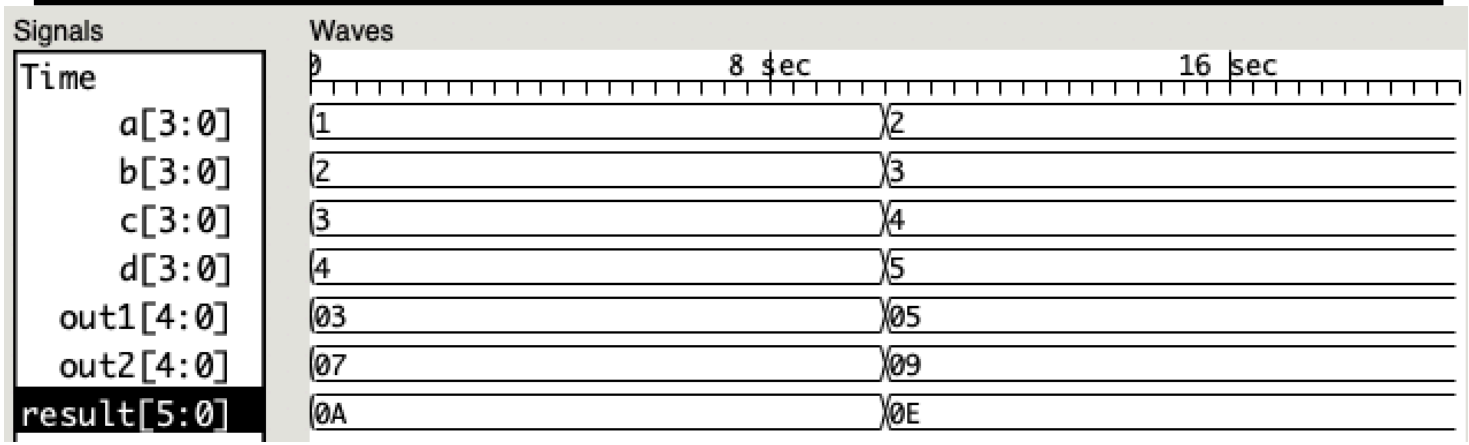
Procedural Assignment	Continuous Assignment
Occurs within an **initial** or an **always** statement	Not used in an **initial** or an **always** statement
Operates with respect to other statements in the **initial** or **always** block	Executes concurrently with other statements
Drives registers	Drives nets
Uses blocking (=) or nonblocking (<=) symbols	Uses the assignment (=) symbol

Blocking assignments

- A **blocking assignment** needs to complete execution before the next statement executes
  - "=" is used for blocking assignment
  - Assignments are performed **sequentially**
    - Order of assignment is very important
  - Used with **level-trigger** always blocks to describe **combinational circuits**
  - Used in initial blocks

Example

```
module add_seq(input [3:0] a, b, c, d, output reg [5:0] result);
  reg [4:0] out1, out2;
  always @(a, b, c, d) begin
    out1 = a + b;
    out2 = c + d;
    result = out1 + out2;
  end
endmodule
```

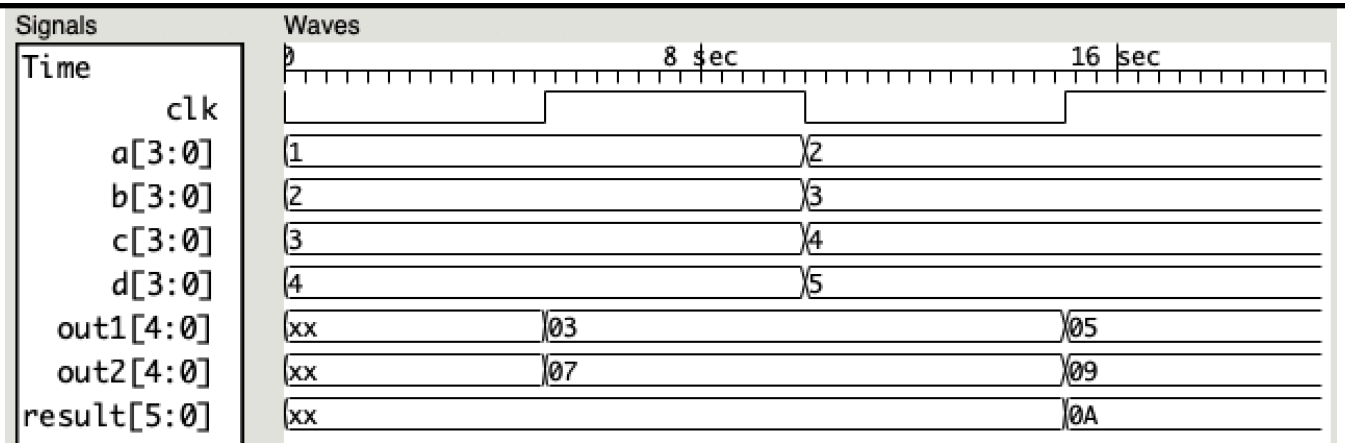


Non-blocking assignments

- **Non-blocking assignments** allow multiple assignments to be performed **without blocking** each others
  - "<=" is used for non-blocking assignments
  - Assignments are performed in **parallel**
    - Order of assignments is NOT important
  - Used with **edge trigger** always blocks to describe **sequential circuits**

Example

```
module add_par(input clk, input [3:0] a, b, c, d, output reg [5:0] result);
  reg [4:0] out1, out2;
  always @(posedge clk) begin
    out1 <= a + b;
    out2 <= c + d;
    result <= out1 + out2;
  end
endmodule
```



Blocking vs. Non-blocking example

- Assume initially that A=1, B=2, C=3, and D=4

```
reg [7:0] A, B, C, D;
always @(posedge clk)
begin
  A = B + C;
  B = A + D;
  C = A + B;
  D = B + D;
end
```

Correct in results but **don't use blocking in edge trigger**

```
reg [7:0] A, B, C, D;
always @(posedge clk)
begin
  A <= B + C;
  B <= A + D;
  C <= A + B;
  D <= B + D;
end
```

Incorrect in results but **use non-blocking in edge trigger**

Correcting the example

```
reg [7:0] A, B, C, D;
reg [7:0] newA, newB, newC, newD;
always @(posedge clk)
begin
  A <= newA; B <= newB;
  C <= newC; D <= newD;
end
always @(*) begin
  newA = B + C;
  newB = newA + D;
  newC = newA + newB;
  newD = newB + D;
end
```

```
reg [7:0] A, B, C, D;
always @(posedge clk)
begin
  A <= B + C;
  B <= B + C + D;
  C <= B + C + B + C + D;
  D <= B + C + D + D;
end
```

Why non-blocking in sequential?

```
always @ (posedge clk, posedge rst) // behavior1
  if (rst) y1 = 0; //reset
  else     y1 = y2;
always @ (posedge clk or posedge rst) // behavior2
  if (rst) y2 = 1; // preset
  else     y2 = y1;
```

- If behavior1 always first after reset, y1 = y2 = 1
- If behavior2 always first after reset, y2 = y1 = 0.
- Results are order dependent, **ambiguous – race condition**!
- This is why we don't use blocking assigns for flip-flops…

Correcting the example

```
always @ (posedge clk, posedge rst) // behavior1
  if (rst) y1 <= 0; //reset
  else     y1 <= y2;
always @ (posedge clk or posedge rst) // behavior2
  if (rst) y2 <= 1; // preset
  else     y2 <= y1;
```

- Assignments for y1 and y2 occur in **parallel**
- y1 = 1 and y2 = 0 after reset
- Values swap each clock cycle after the reset
- **No race condition**!

Control statements

- Behavioral Verilog looks a lot like software
  - risk! – danger and opportunity
- Provides similar control structures
  - Not all of these actually synthesized, but some non-synthesizable statements are useful in test benches

- What kinds synthesize?
  - `if`
  - `case`
  - `for` loops with **constant bounds**

- **Must be placed inside a procedural block**

if - else if - else

- Operator ? : for simple conditional assignments
- Sometimes need more complex behavior, can use if statement!
  - Does not conditionally "execute" block of "code"
  - Does not conditionally create hardware!
  - It makes a multiplexer or similar logic
- Generally:
  - Hardware for all paths is created
  - All paths produce their results in parallel
  - One path's result is selected depending on the condition

if - else if - else

- Syntax:

```
if (expression) statement1;
```

```
if (x1 & x2) z1 = 1;
```

```
if (expression) statement1;
else statement2;
```

```
if (rst_n == 0)
        ctr = 3'b000;
else ctr = next_count;
```

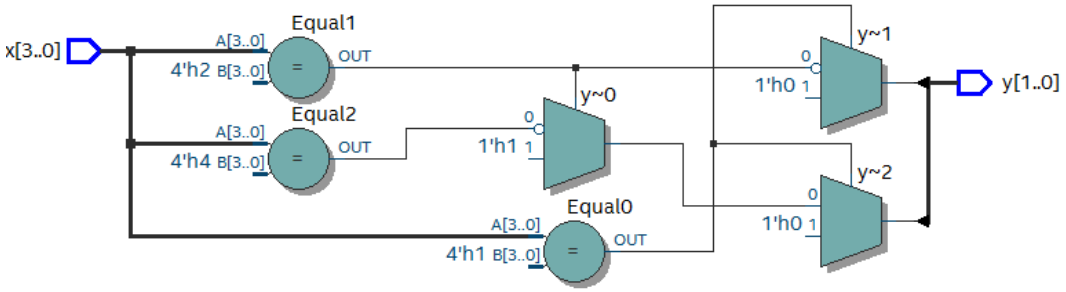
```
if (expression1) statement1;
else if (expression2) statement2;
else if (expression3) statement3;
else default statement;
```

```
if (opcode == 2'b 00)
        z1 = x1 + x2 ;
else if (opcode == 2'b01)
      z1 = x1 - x2 ;
else if (opcode == 2'b10)
      z1 = x1 * x2 ;
else
      z1 = x1 / x2 ;
```

Issues with if - else if - else

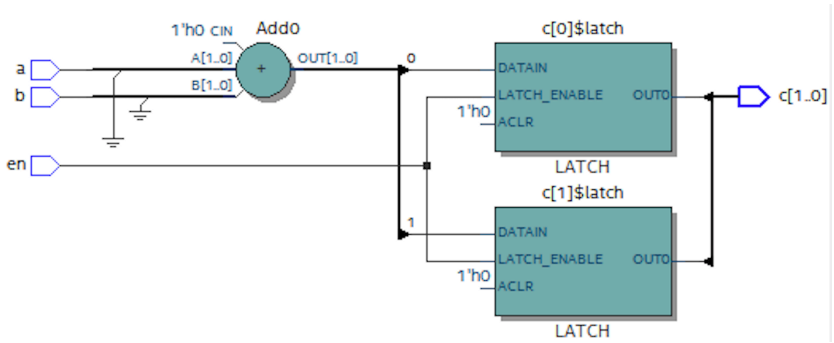
- Priority check
- Chain of multiplexers can be used: **degrade performance**

```
if (x == 4'b0001) y = 2'b00;
else if (x == 4'b0010) y = 2'b01;
else if (x == 4'b0100) y = 2'b10;
else if (x == 4'b1000) y = 2'b11;
else y = 2'bxx;
```



- **Un-wanted latch** when failing to fully specify all possible cases (combinational circuits)

```
always @(en, a, b) begin
  if (en) c = a + b;
end
```



Case statements

- Verilog has three types of case statements:
  - case, casex, and casez
- Performs bitwise match of expression and case item
  - Both must have **same bitwidth** to match!
- Syntax:

```
case (expression)
        case_item1 : procedural_statement1;
        case_item2 : procedural_statement2;
        case_item3 : procedural_statement3;
        ...
        case_itemn : procedural_statementn;
        default : default_statement;
endcase
```

Case statements

- casex
  - The values of x or z are treated as “don’t care”
- casez
  - The value of z is treated as “don’t care”
  - The value of x is a special value

```
reg [7:0] r, mask;
mask = 8'bx0x0x0x0;
casex (r ^ mask)
      8'b001100xx: stat1;
      8'b1100xx00: stat2;
      8'b00xx0011: stat3;
      8'bxx010100: stat4;
endcase
```

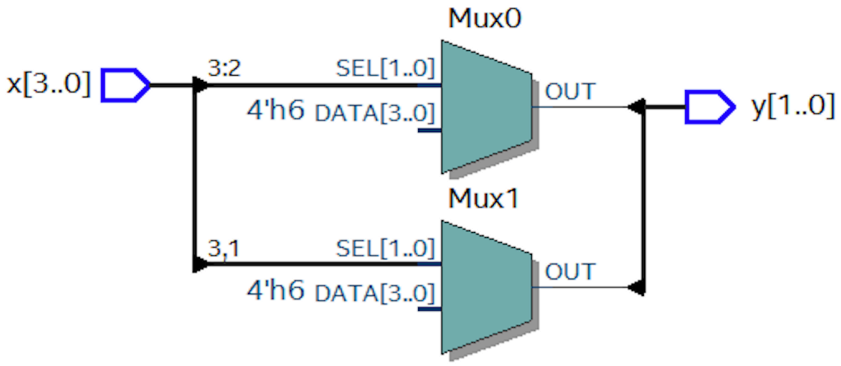
```
reg [7:0] ir;
casez (ir)
      8'b1???????: instruction1(ir);
      8'b01??????: instruction2(ir);
      8'b00010???: instruction3(ir);
      8'b000001??: instruction4(ir);
endcase
```

Case statements

- Can help in **reducing chain of multiplexers**

```
case (x)
        4'b0001: y = 2'b00;
        4'b0010: y = 2'b01;
        4'b0100: y = 2'b10;
        4'b1000: y = 2'b11;
        default: y = 2'bxx;
endcase
```

- Reduce code lines compared to if




```
module priority_encoder (output reg [1:0] Code,
                                    input [3:0] Data);
always @(Data)
  casex (Data)
      4'b1xxx   : Code = 3;
      4'b01xx   : Code = 2;
      4'b001x   : Code = 1;
      4'b0001   : Code = 0;
      default    : Code = 3'bxxx;
  endcase
endmodule
```

- May introduce **un-wanted latch** (like if)