Overview

- Last lecture
  - Latches
  - Flip-flops
  - Edge-triggered D
  - Master-slave
  - Timing diagrams
- Today
  - Sequential Verilog

Variables

- wire
  - Connects components together
- reg
  - Saves a value
  - Part of a behavioral description
  - Does NOT necessarily become a register when you synthesize
  - May become a wire
- The rule
  - Declare a variable as reg if it is a target of an assignment statement
  - Continuous assign doesn’t count

Sequential Verilog

- Sequential circuits: Registers & combinational logic
  - Use positive edge-triggered registers
  - Avoid latches and negative edge-triggered registers
- Register is triggered by "posedge clk"

```verilog
module register(Q, D, clock);
  input D, clock;
  output Q;
  reg Q;
  always @(posedge clock) begin
    Q = D;
  end
endmodule
```

always block

- A procedure that describes a circuit’s function
  - Can contain multiple statements
  - Can contain if, for, while, case
  - Triggers at the specified conditions
  - begin/end groups statements within always block

```verilog
module register(Q, D, clock);
  input D, clock;
  output Q;
  reg Q;
  always @(posedge clock) begin
    Q = D;
  end
endmodule
```

always example

```verilog
module and_gate(out, in1, in2);
  input in1, in2;
  output out;
  reg out;
  always @(in1 or in2) begin
    out = in1 & in2;
  end
endmodule
```

Incomplete trigger or incomplete assignment

- What if you omit an input trigger (e.g. in2)
  - Compiler will insert a register to hold the state
  - Becomes a sequential circuit — NOT what you want

```verilog
module and_gate(out, in1, in2);
  input in1, in2;
  output out;
  reg out;
  always @(in1) begin
    out = in1 & in2;
  end
endmodule
```

2 rules:
1) Include all inputs in the trigger list
2) Use complete assignments
   ⇒ Every path must lead to an assignment for out
   ⇒ Otherwise out needs a state element
Assignments

- Be careful with `always` assignments
  - Which of these statements generate a latch?

```verilog
always @(c or x) begin
  if (c) begin
    value = x;
  end
  y = value;
end
always @(c or x) begin
  value = x;
  if (c) begin
    value = 0;
  end
  y = value;
end
always @(c or x) begin
  if (c) value = 0;
  else if (x) value = 1;
end
```

Another way: Use functions

- Functions for combinational logic
  - Functions can't have state

```verilog
module and_gate (out, in1, in2);
  input in1, in2;
  output out;
  assign out = myfunction(in1, in2);
  function myfunction;
    input in1, in2;
    begin
      myfunction = in1 & in2;
      end
  endfunction
endmodule
```
Simple binary encoder (input is 1-hot)

```verilog
module encode (A, Y);
  input [7:0] A; // 8-bit input vector
  output [2:0] Y; // 3-bit encoded output
  reg [2:0] Y; // target of assignment
  always @(A)
    case (A)
      8'b00000001: Y = 0;
      8'b00000010: Y = 1;
      8'b00000100: Y = 2;
      8'b00001000: Y = 3;
      8'b00010000: Y = 4;
      8'b00100000: Y = 5;
      8'b01000000: Y = 6;
      8'b10000000: Y = 7;
    default: Y = 3'bx; // Don't care about other cases
    endcase
endmodule
```

default case

```
module encode (A, Y);
  input [7:0] A; // 8-bit input vector
  output [2:0] Y; // 3-bit encoded output
  reg [2:0] Y; // target of assignment
  always @(A)
    case (A)
      8'b00000001: Y = 0;
      8'b00000010: Y = 1;
      8'b00000100: Y = 2;
      8'b00001000: Y = 3;
      8'b00010000: Y = 4;
      8'b00100000: Y = 5;
      8'b01000000: Y = 6;
      8'b10000000: Y = 7;
    default: Y = 3'bx; // Don't care about other cases
    endcase
endmodule
```

Priority encoder

```
module encode (A, Y);
  input [7:0] A; // 8-bit input vector
  output [2:0] Y; // 3-bit encoded output
  reg [2:0] Y; // target of assignment
  always @(A)
    case (1'b1)
      A[0]: Y = 0;
      A[1]: Y = 1;
      A[2]: Y = 2;
      A[3]: Y = 3;
      A[4]: Y = 4;
      A[5]: Y = 5;
      A[6]: Y = 6;
      A[7]: Y = 7;
    default: Y = 3'bx; // Don't care when input is all 0's
    endcase
endmodule
```

Verilog while/ repeat/ forever

- **while (expression) statement**
  - Execute statement while expression is true
- **repeat (expression) statement**
  - Execute statement a fixed number of times
- **forever statement**
  - Execute statement forever

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

Blocking and non-blocking assignments

- **Blocking assignments (Q = A)**
  - Variable is assigned immediately
  - New value is used by subsequent statements
- **Non-blocking assignments (Q <= A)**
  - Variable is assigned after all scheduled statements are executed
  - Value to be assigned is computed but saved for later

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

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

```
always @(posedge clk)
  begin
    A <= B;
    B <= A;
    A = temp;
  end
```
Swap

- The following code executes incorrectly
  - One block executes first
  - Loses previous value of variable
    ```verilog
    always @(posedge CLK) begin
        A = B;
    end
    always @(posedge CLK) begin
        B = A;
    end
    ```
- Non-blocking assignment fixes this
  - Both blocks are scheduled by posedge CLK
    ```verilog
    always @(posedge CLK) begin
        A <= B;
    end
    always @(posedge CLK) begin
        B <= A;
    end
    ```

Parallel versus serial execution

- **assign** statements are implicitly parallel
  - `:=` means continuous assignment
  - Example:
    ```verilog
    assign E = A & D;
    assign A = D & C;
    ```
  - `A and K change if B changes`
- **always** blocks execute in parallel
  - `always @(posedge clock)`
  - Procedural block internals not necessarily parallel
    - `:=` is a blocking assignment (sequential)
    - `<=` is a nonblocking assignment (parallel)
  - Examples of procedures: **always**, **function**, etc.

Synthesis examples

```verilog
wire [3:0] x, y, a, b, c, d;
assign apr = ^a;
assign y = a & ~b;
assign x = (a == b) ? (a + c) : (d + a);
```