Hardware Description Languages and Sequential Logic

- Flip-flops
  - representation of clocks - timing of state changes
  - asynchronous vs. synchronous
- Shift registers
- Simple counters

Flip-flop in Verilog

- Use always block's sensitivity list to wait for clock edge

```verilog
module dff (clk, d, q);
  input clk, d;
  output q;
  reg q;
  always @(posedge clk)
    q = d;
endmodule
```
More Flip-flops

- Synchronous/asynchronous reset/set
  - single thread that waits for the clock
  - three parallel threads – only one of which waits for the clock

### Synchronous

```verilog
module dff (clk, s, r, d, q);
  input  clk, s, r, d;
  output q;
  reg    q;
  always @(posedge clk)
    if (r)      q = 1'b0;
    else if (s) q = 1'b1;
    else        q = d;
endmodule
```

### Asynchronous

```verilog
module dff (clk, s, r, d, q);
  input  clk, s, r, d;
  output q;
  reg    q;
  always @(posedge clk)
    q = 1'b0;
  always @(posedge s)
    q = 1'b1;
  always @(posedge clk)
    q = d;
endmodule
```

Incorrect Flip-flop in Verilog

- Use always block's sensitivity list to wait for clock to change

```verilog
module dff (clk, d, q);
  input  clk, d;
  output q;
  reg    q;
  always @(clk)
    q = d;
endmodule
```

Not correct! Q will change whenever the clock changes (both edges), not just on one edge.
Blocking and Non-Blocking Assignments

- **Blocking assignments (X=A)**
  - completes the assignment before continuing on to next statement
- **Non-blocking assignments (X<=A)**
  - completes in zero time and doesn't change the value of the target until a blocking point (delay/wait) is encountered
- **Example: swap**

```verilog
always @(posedge CLK) begin
    temp = B;
    B = A;
    A = temp;
end

always @(posedge CLK) begin
    A <= B;
    B <= A;
end

always @(posedge CLK) begin
    A = A ^ B;
    B = A ^ B;
    A = A ^ B;
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 to execute by posedge CLK

```verilog
always @(posedge CLK) begin
    A <= B;
end

always @(posedge CLK) begin
    B <= A;
end
```
Register-transfer-level (RTL) Assignment

- Non-blocking assignment is also known as an RTL assignment
  - if used in an always block triggered by a clock edge
  - all flip-flops change together

```verilog
// B, C, D all get the value of A
always @(posedge clk)
begin
  B = A;
  C = B;
  D = C;
end

// implements a shift register
always @(posedge clk)
begin
  B <= A;
  C <= B;
  D <= C;
end
```

Shift register in Verilog

```verilog
module shift_register (clk, in, out);

  input clk;
  input in;
  output [0:3] out;
  reg [0:3] out;

  initial begin
    out = 0; // out[0:3] = {0, 0, 0, 0};
  end

  always @(posedge clk) begin
    out = {in, out [0:2]};
  end

endmodule
```
Activity

```verilog
initial
begin
A = 1'b0; B = 1'b0; C = 1'b0; D = 1'b0;
end

always @(posedge clk)
begin

end
```

Binary Counter in Verilog

```verilog
module binary_counter (clk, c8, c4, c2, c1);
input clk;
output c8, c4, c2, c1;
reg [3:0] count;
initial begin
count = 0;
end
always @(posedge clk) begin
count = count + 4'b0001;
end
assign c8 = count[3];
assign c4 = count[2];
assign c2 = count[1];
assign c1 = count[0];
endmodule
```

```verilog
module binary_counter (clk, c8, c4, c2, c1, rco);
input clk;
output c8, c4, c2, c1, rco;
reg [3:0] count;
reg rco;
initial begin
end
always @(posedge clk) begin
end
assign c8 = count[3];
assign c4 = count[2];
assign c2 = count[1];
assign c1 = count[0];
assign rco = (count == 4b'1111);
endmodule
```

add RCO
8-bit register of Lab 5

```verilog
module register_8_bit (  
    input [7:0] D,  
    input clear,  
    input store,  
    output reg [7:0] Q );

    // buttons are active low so look for  
    // negedges and test for !clear  

    always @(negedge clear, negedge store)  
        if (!clear) Q <= 8'b0000_0000;  
        else        Q <= D;

endmodule
```

Parallel versus serial execution

- **assign** statements are implicitly parallel
  - "=" means continuous assignment
  - Example
    ```verilog
    assign E = A & D;  
    assign A = B & C;
    ```
  - A and E change if B changes

- **always** blocks execute in parallel
  - **always @(posedge clock)**

- Always block internals not necessarily parallel
  - "=" is a blocking assignment (sequential)
  - "<=" is a non-blocking assignment (parallel)
Sequential logic summary

- Fundamental building blocks of circuits with state
  - latch and flip-flop
  - R-S latch, R-S master/slave, D master/slave, edge-triggered D flip-flop
- Timing methodologies
  - use of clocks
  - cascaded FFs work because $T_{prop} > T_{hold}$
  - beware of clock skew
  - period > $T_{propFF} + T_{propCL} + T_{setup}$
- Basic registers
  - shift registers
  - counters
- Hardware description languages and sequential logic
  - always (@ posedge clk)
  - blocking and non-blocking assignments

Verilog review/style
Variables in Verilog

- **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

**Important rule**
- Declare a variable as reg if it is a target of an assignment statement inside an always block
  - Continuous assign doesn’t count

Always block

- A construct that describes a circuit’s behavior
  - begin/end groups multiple statements within an always block
  - Can contain if, for, while, case
  - Triggers at the specified conditions in sensitivity list: @(...)
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
```

Example: a D flip-flop

- Register: in this case, holds value of `Q` between clock edges
  - We want this register to be SYNTHESIZED

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

Always example

- Holds assignment in always block – but we do NOT want a SYNTHESIZED register
- The compiler will not synthesize this code to a register, because `out` changes whenever `in1` or `in2` change. Could simply write:
  ```verilog
  wire [0:1] out, in1, in2;
  and (out, in1, in2);
  ```

- Specifies when block is executed i.e. triggered by changes in `in1` or `in2`
Incomplete sensitivity list or incomplete assignment

- What if you omit an input trigger (e.g. in2)
  - Compiler will insert a latch 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

Real state! Holds out because in2 isn't specified in always sensitivity list — a register is synthesized that we DO NOT want

Assignments

- Be careful with always assignments
  - Which of these statements generate state?

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

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

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

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

- Same as Java/C if statement

```verilog
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel; // 2-bit control signal
input A, B, C, D;
output Y;
reg Y; // target of assignment

always @(sel or A or B or C or D)
  if (sel[0] == 0)
    if (sel[1] == 0)  Y = A;
    else              Y = B;
  else
    if (sel[1] == 0)  Y = C;
    else              Y = D;
endmodule
```

⇒ Single if statements synthesize to multiplexers
⇒ Nested if/else statements usually synthesize to logic

if (another way)

```verilog
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
input [1:0] sel; // 2-bit control signal
input A, B, C, D;
output Y;
reg Y; // target of assignment

always @(sel or A or B or C or D)
  if (sel[0] == 0)
    if (sel[1] == 0)  Y = A;
    else
      Y = B;
  else
    if (sel[1] == 0)  Y = C;
    else
      Y = D;
endmodule
```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
    input [1:0] sel;  // 2-bit control signal
    input A, B, C, D;
    output Y;
    reg Y;          // target of assignment

    always @(sel or A or B or C or D)
        case (sel)
            2'b00: Y = A;
            2'b01: Y = B;
            2'b10: Y = C;
            2'b11: Y = D;
        endcase
endmodule

case executes sequentially
⇒ First match executes
⇒ Don’t need to break out of case

// Simple binary encoder (input is 1-hot) - comb. logic
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

If you omit the default, the compiler will create a latch for Y – not good
// 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

for statements synthesize as cascaded combinational logic
⇒ Verilog unrolls the loop
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

Some simple synthesis examples

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