Getting Started with Verilog

- Verilog is used to describe **hardware**, not software
- Verilog code is composed of a number of modules
- Modules are like functions or methods: they describe the interface of a component and its implementation
- Components modeled by a module can be primitive or complex objects
- Models can be implemented behaviorally or structurally
A Verilog Module

module xor_gate (A, B, F);
    input A, B;
    output F;
    assign F = (A & B) | (~A & ~B);
endmodule
Many data types in Verilog, but we will use a small subset:

```verilog
integer i; // used for loop indices

reg r1;
reg [31:0] r2;

wire sigA, sigB;
wire [31:0] sigC;
tri busD;
tri [31:0] busE;
```
Literals

- General form is `[size][radix]value`

0
10
'b10
'h10
4'b100
4'bx
8'b0100_1001
8'o170
Declarations

module mymod (inbus, outbus);

  parameter bus_width = 32;
  parameter rise_delay = 20;

  input [31:0] inbus;
  output [31:0] outbus;

  reg [31:0] regA;

  wire somesig;

endmodule
Operators

- **Logical**
  - A&&B  A| |B  !A  ~A  A&B  A|B  A~^B

- **Reduction**
  - &A  ~&A  |A  ~|A  ^~A  ~^A

- **Relational**

- **Shift**
  - A<<B  A>>B

- **Arithmetic**
  - A+B  A-B  A*B  A/B  A%B

- **Conditional, concatenate, replicate**
  - A?B:C  {A, B}  {A{B}}
Continuous Signal Assignment

- Consists entirely of boolean equations
- References only wire signals
- Constantly re-evaluates whenever inputs change

assign f = (a & ~b) OR (~a & b);
Structural Models

- Defines a module using components
- Defines internal signals to connect components
- All basic gates are available

```verilog
wire s1, s2;

nor(s1, s2, f);
and(a, b, s1);
nor(a, b, s2);
```
Example Structural Model

module component1 (in1, in2, out);
    input in1, in2;
    output out;
    assign out = (in1 & in2) | (~in1 & in2);
endmodule

module mainmod (a, b, c, d, e);
    input a, b, c, d;
    output e;
    wire w1, w2;
    and (a, b, w1);
    or (c, d, w2);
    component1 inst1 (w1, w2, e);
endmodule
always Statements

- Behavioral models

```plaintext
always @( <sensitivity-list> )
begin
  <procedural-statments>
  <if-else>
  <case>
  <while, for, repeat>
end
```
Sensitivity List

- Sensitivity list tells simulator when to evaluate the always block's code
- When any signal in the list changes its value, the always block is executed
- Special constructs posedge and negedge allow edge-triggered (flip-flop) models
always Example

//
// an 8-bit 2-to-1 mux
//
wire control;
reg [7:0]a, [7:0]b, [7:0]c;

always @(control or a or b)
begin
  if (control == 1)
    c <= b;
  else
    c <= a;
end
Example using case statement

//
// an 8-bit 4-to-1 mux
//
wire [1:0]control;
reg [7:0]a, [7:0]b, [7:0]c, [7:0]d, [7:0]e;
always @(control or a or b or c or d)
begin
  case (control)
    0: e <= a;
    1: e <= b;
    2: e <= c;
    3: e <= d;
  endcase
end
initial Statements

☐ Simulation models - testbenches

initial
begin
    procedural-statments
    if-else
    case
    while, for, repeat
end
Statements

- All statements end with a semi-colon ;
- Compound statements bracketed with 
  
  \begin{verbatim}
  begin ... end
  \end{verbatim}

  \begin{verbatim}
  begin
    a = b + c;
    b = b & d;
  end
  \end{verbatim}
Control Flow Statements

if (<condition>)
    <statement>
else
    <statement>

case (<expression>)
    <value> : <statement>
    <value> : <statement>
    default : <statement>
endcase

for (<initial>; <condition>; <increment>)
    <statement>
Combinational Logic

- Use continuous assignments or ...
- always block with inputs in the sensitivity list

```verbatim
module some_logic (a, b, c, d, e);
  input a, b, c, d;
  output e;
  reg s1;
  assign e = s1 | d;
  always @(a or b or c)
    if (a & ~b)
      s1 = c;
    else
      s1 = ~a;
endmodule
```
Assignment Statements

//continuous assignment
assign a = (b | c) & d;

always @(b or c or d)
begin
    //procedural blocking assignment
    a = (b | c) & d;

    //procedural non-blocking assignment
    a <<= (b | c) & d;
end
Non-Blocking Assignments

- Evaluate at the time of execution
- Complete (and update the variable) after all events at the current time
- Used only inside an always, and only to assign a reg or integer
- Only assign a given variable once
- Used in implementing sequential logic
Sequential Circuits - Simple FF

Use an always to control the inputs

```verilog
module flip-flop (clk, d, q);
    input clk, d;
    output q;

    always @(clk or d)
        if (clk)
            q <= d;

endmodule
```
Edge-Triggered FF

- Use posedge to control the clock only

```verilog
module flip-flop (clk, d, q);
    input clk, d;
    output q;

    always @(posedge clk)
        q <= d;

endmodule
```
Asynchronous Reset or Preset

- Use posedge to control the clock and reset

```verilog
module flip-flop (clk, rst, d, q);
    input clk, rst, d;
    output q;

    always @(posedge clk or posedge rst)
        if (rst == 1)
            q <= 0;
        else
            q <= d;

endmodule
```
A Simple Example
Example Circuit
module example (clk, rst, x, y, z);
  input clk, rst, x, y;
  output z;
  wire da, db;
  reg qa, qb;
  assign da = y ^ qb;
  assign db = x ^ qa;
  assign z = qb;
  always @(posedge clk orposedge rst)
    if (rst == 1)
      qa <= 0;
    else
      qa <= da;
  always @(posedge clk orposedge rst)
    if (rst == 1)
      qb <= 0;
    else
      qb <= db;
endmodule
module example (clk, rst, x, y, z);
  input clk, rst, x, y;
  output z;
  reg qa, qb;
  assign z = qb;
  always @(posedge clk or posedge rst)
    if (rst == 1)
      qa <= 0;
    else
      qa <= y ^ qb;
  always @(posedge clk or posedge rst)
    if (rst == 1)
      qb <= 0;
    else
      qb <= x ^ qa;
endmodule
module example (clk, rst, x, y, z);
    input clk, rst, x, y;
    output z;
    reg qa, qb;
    assign z = qb;
    always @(posedge clk or posedge rst)
        if (rst == 1)
            begin
                qa <= 0;
                qb <= 0;
            end
        else
            begin
                qa <= y ^ qb;
                qb <= x ^ qa;
            end
endmodule
Registers

- Registers are exactly like edge-triggered flip-flops but ...
  - They usually have more than one bit
  - They have more complex logic in the body

```plaintext
always @(posedge clk or negedge rst)
if (rst == 0)
    register <= 0;
else
    begin
        <register logic goes here>
    end
```
// data register
always @(posedge clk or negedge rst)
if (rst == 0)
    dregister <= 0;
else
    if (dload == 1)
        dregister <= dreg_in;


// shift register
always @(posedge clk or negedge rst)
if (rst == 0)
    sregister <= 0;
else
    case (scontrol)
        1 : sregister <= sregister >> 1;
        2 : sregister <= sregister << 1;
        3 : sregister <= sreg_in;
    endcase
// counter
always @(posedge clk or negedge rst)
if (rst == 0)
    counter <= 0;
else
    if (cload == 1)
        counter <= cnt_in;
    else if (count == 1)
        counter <= counter + 1;

// dreg with mux on input
always @(posedge clk or negedge rst)
if (rst == 0)
    dregister <= 0;
else
    case (mcontrol)
        0 : dregister <= abus;
        1 : dregister <= ~abus;
        2 : dregister <= abus & bbus;
        3 : dregister <= abux + bbus;
    endcase
Testbenches

- Used to test a design
- Generates all external stimulus, including clock and reset
- Uses an initial block to sequence the test
- May use many language constructs NOT used in a hardware model
Testbench Template

// no arguments here
module testbench;

wire <declare a wire for each output>;
reg <declare a reg for each input>;

initial
begin
    <test sequence goes here>
end

// this is a 20s clock
always #10
    clk <= ~clk;

// here we declar an instance of the design
design uut (<input/output signals go here>)
endmodule
Controlling Time in Verilog

- Several statements can cause a process to wait
  - To wait for a specific period of time
    
    #10; // waits for 10s

    #param; // waits for the time specified by the parameter

  - To wait for a specific event to occur
    
    @(event-list); // just like the always sensitivity list
module testbench;
   reg clk, rst, x, y;
   wire z;
   always #10
      clk = ~clk;
initial
begin
   x = 0;
   y = 0;
   rst = 0;
   #5;
   rst = 1;
   #40;
   rst = 0;
   @(posedge clk);
   x = 1;
   @(posedge clk);
   x = 0;
Testbench Example

y = 1;
 @(posedge clk);
x = 1;
 @(posedge clk);
 @(posedge clk);
y = 0;
 @(posedge clk);
x = 0;
 @(posedge clk);
 @(posedge clk);
y = 1;
 @(posedge clk);
x = 1;
 @(posedge clk);
x = 0;
y = 0;
 @(posedge clk);
end

example uut (clk, rst, x, y, z);
endmodule
module testbench;
    reg clk, rst, start;
    reg [31:0] multiplier;
    reg [31:0] multiplicand;
    wire [63:0] product;
    wire busy;

initial
begin
    clk = 0;
    rst = 0;
    start = 0;
    multiplier = 0;
    multiplicand = 0;
    #10;
    rst = 1;
    #40;
    rst = 0;
    $display("reset done");
    @(posedge clk);
    $display("starting multiply");
    multiplier = 8;
    multiplicand = 22;
    start = 1;
    @(posedge busy);
    $display("busy is high");
    start = 0;
    multiplier = 0;
    multiplicand = 0;
    @(negedge busy);
    $display("busy is low");
    $finish;
end

always #10
    clk = ~clk;

multiplier uut (clk, rst, start, busy,
    multiplier, multiplicand, product);
State Machines in Verilog
module moore_machine (clk, rst, x, z);
    input clk, rst, x;
    output z;
    reg [1:0] state, next_state;
    parameter [1:0] A=2'b00, B=2'b01, C=2'b11;

    assign z = (state == C); // Moore output

    always @(posedge clk or posedge rst)
        if (rst == 1)
            state <= A;
        else
            state <= next_state;

    always @(state, x)
        case (state)
            A: if (x == 1) next_state <= B else next_state <= A;
            B: if (x == 1) next_state <= A else next_state <= C;
            C: if (x == 1) next_state <= A else next_state <= B;
        endcase
endmodule
State Machines in Verilog

A/0

B/0

C/1

reset

x

w&x

x&y

x
module moore_machine (clk, rst, w, x, y, z);
    input clk, rst, w, x, y;
    output z;
    reg [1:0] state, next_state;
    parameter [1:0] A=2'b00, B=2'b01, C=2'b11;

    assign z = (state == C); // Moore output

    always @(posedge clk or posedge rst)
        if (rst == 1)
            state <= A;
        else
            state <= next_state;

    always @(state, w, x, y)
        case (state)
            A: if (x) next_state <= B else next_state <= A;
            B: if (x & y) next_state <= A else next_state <= C;
            C: if (w & x) next_state <= A else next_state <= B;
        endcase

endmodule
module moore_machine (clk, rst, w, x, y, z);
    input clk, rst, w, x, y;
    output z;
    reg [2:0] state, next_state;
    parameter [2:0] A=3'b001, B=3'b010, C=3'b100;

assign z = state[2]; // Moore output

always @(posedge clk or posedge rst)
    if (rst == 1)
        state <= A;
    else
        state <= next_state;

always @(state, w, x, y)
    case (state)
        A: if (x) next_state <= B else next_state <= A;
        B: if (x & y) next_state <= A else next_state <= C;
        C: if (W & x) next_state <= A else next_state <= B;
    endcase
endmodule