Exp#5 & 6

Introduction to Verilog

COE203
Digital Logic Laboratory

Dr. Ahmad Almulhem
KFUPM
Spring 2009
What is Verilog?

- Verilog is a hardware description language (HDL)
- Can describe everything from single gate to a complete computer system
- Allows to design at Behavior Level, Register Transfer Level (RTL), Gate level and at switch level
- Syntax is similar to C language (case-sensitize, similar keywords)
- Parallel not serial (Not like C language)
- Technology-independent, easy to design and debug
- Most popular HDLs are VHDL and Verilog
Why use HDL?

- Digital systems are highly complex; millions of transistors.
- For large digital systems, gate-level design is unmanageable.
- HDL offers the mechanism to describe, test and synthesize large designs.
- Computer-aided design tools aid in the design process.

- © Intel P4 Processor
- Introduced in 2000
- 40 Million Transistors
- 1.5GHz Clock
Levels of description in Verilog

• **Switch Level**
  - Layout of the wires, resistors and transistors on an Integrated Circuit (IC) chip

• **Gate (structural) Level**
  - Logical gates, flip flops and their interconnection

• **RTL (dataflow) Level**
  - The registers and the transfers of vectors of information between registers

• **Behavioral (algorithmic) Level**
  - Highest level of abstraction
  - Description of algorithm without hardware implementation details (like C programming)
In Verilog, a digital system is described as a set of **modules**.

Each module has a **port list (interface)** to other modules.

Modules can represent pieces of hardware ranging from simple gates to complete systems, e.g., a microprocessor.
Gate Level (Structural) Modeling

- Text description of the circuit structure (netlist)
  - built-in primitives gates

```verilog
// my gate
module my_gate(OUT1, IN1, IN2);
  output OUT1;
  input IN1, IN2;
  wire X;
  and (X, IN1, IN2);
  not (OUT1, X);
endmodule
```

Any internal net must be defined as `wire`
Verilog Primitives

Basic logic gates only

- and
- or
- not
- buf
- xor
- nand
- nor
- xnor
- bufif1, bufif0
-notif1, notif0
Primitive Pins Are Expandable

\[ \text{nand} (y, \text{in1}, \text{in2}) ; \]

\[ \text{nand} (y, \text{in1}, \text{in2}, \text{in3}) ; \]

\[ \text{nand} (y, \text{in1}, \text{in2}, \text{in3}, \text{in4}) ; \]
// Half Adder
module hadd (S, C, X, Y);
    input X, Y;
    output S, C;
    xor (S, X, Y);
    and (C, X, Y);
endmodule
// Full Adder
module fadd (S, C, X, Y, Z);
    input X, Y, Z;
    output S, C;
    wire w1, w2, w3;

    hadd M1 (w1, w2, X, Y);
    hadd M2 (S, w3, w1, Z);
    or (C, w2, w3);
endmodule

- Here we instantiate hadd twice. i.e., placing two hadd circuits and connecting them.

- This full adder is built from two half adders and an OR gate.
module add3 (s, cout, ci, a, b);
    input [2:0] a, b; // port declarations
    input ci;
    output [2:0] s;  // vector
    output cout;
    wire [1:0] co;

    fadd a0 (co[0], s[0], a[0], b[0], ci);
    fadd a1 (co[1], s[1], a[1], b[1], co[0]);
    fadd a2 (co[2], s[2], a[2], b[2], co[1]);
endmodule
Continuous Assignments

- Describe combinational logic
- Operands + operators
- Drive values to a net
  - `assign out = a&b ; // and gate`
  - `assign eq = (a==b) ; // comparator`
- Changes to right hand side are evaluated and the result propagated to the left hand side
- Avoid logic loops
  - `assign a = b + a ;`
### Simple XOR Gate

```verilog
// simple xor gate using RTL
module my_xor( C, A, B );

  output C;
  input A, B;

  assign C = (A ^ B);
endmodule
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>Bitwise NOT</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
</tr>
</tbody>
</table>
Can we change the order of the assign statements?
Behavioral Description of an Adder

```verilog
module adder3 (S, Cout, A, B, Cin);
  input [2:0]  A, B;
  input  Cin;
  output [2:0]  S;
  output  Cout;
  assign  { Cout, S } = A + B + Cin;
  // note: Verilog treats wires
  // as 'unsigned' numbers
endmodule
```

3-bit operands, 4-bit result

{ Cout, S } is a 5 bit bus:

Behavioral Description of an Adder

```verilog
module adder (S, Cout, A, B, Cin);
  parameter width = 2;
  input Cin;
  input [width:0] A, B;
  output [width:0] S;
  output Cout;

  assign { Cout, S } = A + B + Cin;
endmodule
```
Logic Values

The underlying data representation allows for any bit to have one of four values:

- 0: for "logic 0", or a false condition
- 1: for "logic 1", or a true condition
- z: for the high-impedance state
- x: for an unknown value
Data Types

Net: wire

- variables represent physical connections between structural entities such as gates.
- A wire does not store a value.

Registers: reg

- store the last value that was procedurally assigned to them
Number representation

- Format: `<size><base format><number>`
  - `<size>`: number of bits (optional)
  - `<base format>`: 'b, 'd, 'o, 'h
  - `<number>`: digits which are legal for the `<base format>`

- Examples:
  - 549 // decimal number
  - 'h 8FF // hex number
  - 'o765 // octal number
  - 4'b11 // 4-bit binary number 0011
  - 3'b10x // 3-bit binary number with least significant bit unknown
  - 5'd3 // 5-bit decimal number
  - -4'b11 // 4-bit two's complement of 0011, or equivalently 1101
Operators

{ }    concatenation
+    -    *    /    
arithmetic
%    modulus
>    >=    <    <=    
relational
!    logical NOT
&&    logical AND
||    logical OR
==    logical equality
!=    logical inequality
? :    conditional

~    bit-wise NOT
&    bit-wise AND
|    bit-wise OR
^    bit-wise XOR
^~    ~^    bit-wise XNOR
&~    ~&    reduction AND
|~    ~|    reduction OR
~&    reduction NAND
^~    ~^    reduction XOR
~^    ^~    reduction XNOR
<<    shift left
>>    shift right
Always Block

always @(sensitivity_list)
begin [optional name]
    [optional local declaration];
    [procedural statement];
    [procedural statement];
    ........
end

- **always** blocks are procedural blocks that contain sequential statements (*order is important*).
- It executes when **sensitivity_list** changes:
  - Level change: **always @(a or b or c)**
  - Edge change: **always @(posedge clock)**
  - Edge change: **always @(negedge clock)**
- if-else and case statement are only in always block
Wire vs. Reg

There are two types of variables in Verilog:

- **Wires** *(all outputs of assign statements must be wires)*
- **Regs** *(all outputs of always blocks must be regs)*

Both variables can be used as inputs anywhere:

- Can use regs or wires as inputs (RHS) to assign statements
- assign bus = LatchOutput + ImmediateValue
  // bus must be a wire, but LatchOutput can be a reg
- Can use regs or wires as inputs (RHS) in always blocks
- always @ (in or clk)
  if (clk) out = in  // in can be a wire, out must be a reg
Always vs. Assign

```
module and3(a, b, c, y);
  input wire a, b, c;
  output reg y;
  always @ (a or b or c)
    begin
      y = a;
      y = y & b;
      y = y & c;
    end
endmodule
```

```
module and3(a, b, c, y);
  input wire a, b, c;
  output wire y;
  assign y = a;
  assign y = y & b;
  assign y = y & c;
endmodule
```
If Statements

Syntax:

```
if (boolean expression)
begin
  statements...
end
else if (boolean expression)
begin
  statements...
end
else
begin
  statements...
end
```

`begin-end` can be omitted if only one statement.

Example:

```
if (alu_func == 2'b00)
  aluout = a + b;
else if(alu_func == 2'b01)
  aluout = a - b;
else if(alu_func == 2'b10)
  aluout = a & b;
else // alu_func == 2'b11
  aluout = a | b;
```
Case Statements

Syntax

```
case (expression)
  case_choice1:
    begin
      ...statements...
    end
  case_choice2:
    begin
      ...statements...
    end
  default:
    begin
      ...statements...
    end
endcase
```

*begin-end* can be omitted if only one statement.

Example:

```
case (alu_ctr)
  2'b00: aluout = a + b;
  2'b01: aluout = a - b;
  2'b10: aluout = a & b;
  default: aluout = 1'bx; // Treated as don't cares for
                  // minimum logic generation.
```

if (m==n) 
    r = a + b + c ;
else if (m > n ) 
    r = a - b ;
else 
    r = c + 1 :

Priority routing network
sequence of 2-to-1 MUX
Case Statements

```verbatim
wire [1:0] sel;
case (sel)
  2'b00: r = a + b + c;
  2'b10: r = a - b;
  default: r = c + 1;
endcase
```

multiplexing network
sequence of n-to-1 MUX
Example: Full Adder

module fadd (A, B, Cin, S, Cout);
input A, B, Cin;
output S, Cout;
reg S, Cout;

always @(A or B or Cin)
begin
    S = (A ^ B ^ Cin);
    Cout = (A & B) | (A & Cin) | (B & Cin);
end
endmodule
Example: 2x1 Multiplexer

**Method 1**
module mux2x1 ( b, c, select, a);
  input  b, c, select;
  output a;
  assign a = (select ? b : c);
endmodule

**Method 2**
module mux2x1 ( b, c, select, a);
  input  b, c, select;
  output a; reg a;
  always@(select or b or c) begin
    if (select) a=b;
    else  a=c;
  end
endmodule

**Method 3**
module mux2x1 ( b, c, select, a);
  input  b, c, select;
  output a; reg a;
  always@(select or b or c) begin
    case (select)
      1'b1: a=b;
      1'b0: a=c;
    endcase
  end
endmodule
A 2-Bit ALU

module alu (A, B, Y, Sel);
  parameter width=2;
  input [width-1:0] A, B;
  input [2:0] Sel;
  output [width-1:0] Y;
  reg [width-1:0] Y;

  always @(A or B or Sel)
  begin
    case (Sel[2:0])
      3'b000 : Y = ... ; // A and B
      3'b001 : Y = ... ; // A or B
      3'b010 : Y = ... ; // A xor B
      3'b011 : Y = ... ; // 1's complement of A
      3'b100 : Y = ... ; // A nand B
      3'b101 : Y = ... ; // A nor B
      3'b110 : Y = ... ; // max(A,B)
      default : Y = 0;
    endcase
  end
endmodule