Overview of Verilog

Using a Hardware Description Language

- Today most digital design of processors is done using a hardware description language.
  - it provides an abstract description of the hardware to simulate and debug the design.
- Verilog is one of the two primary hardware description language.
- Verilog can specify both a behavioral and a structural definition of a digital system.
  - A behavioral specification describes how a digital system operates functionally.
  - A structural specification describes how a digital system is organized in terms of a hierarchical connection of elements.

Verilog

- Hardware description language (HDL) to model electronic systems
- Used in design, verification and implementation of digital logic chips
- Verilog allows a designer to develop a complex hardware system, e.g., a VLSI chip containing millions of transistors, by defining it at *various levels of abstraction*
  - at the (highest) *behavioral*, or *algorithmic*, *level* the design consists of C-like procedures that express functionality without regard to implementation
  - at the *dataflow level* the design consist of specifying how data is processed and moved between registers
  - at the *gate level* the structure is defined as an interconnection of logic gates
  - at the (lowest) *switch level* the structure is an interconnection of transistors

Verilog

- A programming language similar to C
  - Variables
  - Control flow keywords
  - Procedural blocks
- But differs by including
  - The propagation of time
  - Signal dependencies

Digital System

- Entire systems can be viewed as being composed of numerous individual *module*s. A Verilog module is a *system block* with well defined
  - 1. input signals, say x1,x2,...,xn
  - 2. output signals, say y1,y2,...,ym and
  - 3. internal structure and connections or behavior.

Structural Hardware Modeling

$$z = a \cdot b + a \cdot c + \overline{a} \cdot d + \overline{e} \cdot d.$$

```
or    or1(z,x1,x2,x3,x4);

and   and1(x1,a,b);

and   and2(x2,a,c);

not   not1(na,a);

not   not2(ne,e);

and   and3(x3,na,d);

and   and4(x4,ne,d);
```

Modules

```
module Functionz(z,a,b,c,d,e);

  output z;         /* Output wire for Functionz */

  input  a,b,c,d,e; /* Input ports */

  wire x1,x2,x3,x4,na,ne; /* Wires for internal connections */

   or   or1(z,x1,x2,x3,x4); /* Instantiation of OR gate `or1' */

  and  and1(x1,a,b); /* Instantiation of AND gate `and1' */

  and  and2(x2,a,c); /* etc. */

  not  not1(na,a);

  not  not2(ne,e);

  and  and3(x3,na,d);

  and  and4(x4,ne,d);

endmodule
```

Each module defines a set of inputs and outputs connections, called ports. Verilog ports are connection points with other modules in the system.

Module components are interconnected by wires. Wires represent the actual conductors connecting the output of a gate to the input of another.

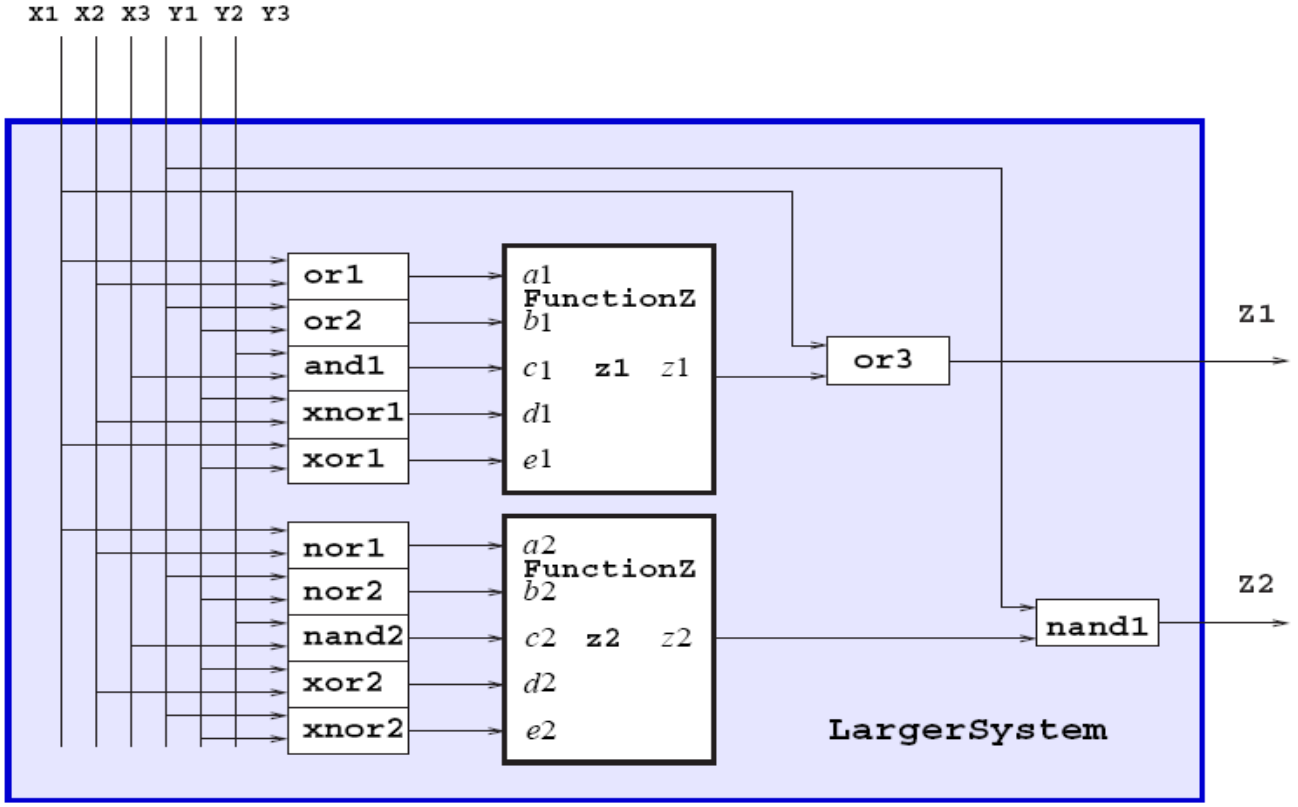
Module components can be simple logic gates, as here, or they can be instances of other modules.

Module Hierarchy

```
module LargerSystem (Z1, Z2, X1, X2, X3, Y1, Y2, Y3 );
        output Z1, Z2;
        input X1, X2, X3, Y1, Y2, Y3; /* Input ``ports'' */
        wire a1, b1, c1, d1, e1, z1, a2, b2, c2, d2, e2, z2;

        Functionz fz1(z1, a1, b1, c1, d1, e1); /* 1st instantiation of Functionz */
        Functionz fz2(z2, a2, b2, c2, d2, e2); /* 2nd instantiation of Functionz */

        or    or3  (Z1, z1, X1);
        nand  nand1(Z2, z2, Y1);
        or    or1  (a1, X1, X2);
        or    or2  (b1, Y1, Y2);
        and   and1 (c1, Y3, X3);
        xnor  xnor1(d1, Y2, X2);
        xor   xor1 (e1, Y1, Y2);
        nor   nor1 (a2, X1, X2);
        nor   nor2 (b2, Y1, Y2);
        nand  nand2(c2, Y3, X3);
        xor   xor2 (d2, Y2, X2);
        xnor  xnor2(e2, Y1, Y2);
endmodule
```



simpleGate.v

```
module aOrNotbOrc(d, a, b, c);
   output d;
   input a, b, c;
   wire p, q;

   not(q, b);
   or(p, a, q);
   or(d, p, c);
endmodule
```

A *gate-level* module does not contain procedures.

Statements in a gate-level module *can be re-ordered* without affecting the program as they simply describe a *set of connections* rather than a *sequence of actions* as in behavioral code. A gate-level module is equivalent to a *combinational circuit*.

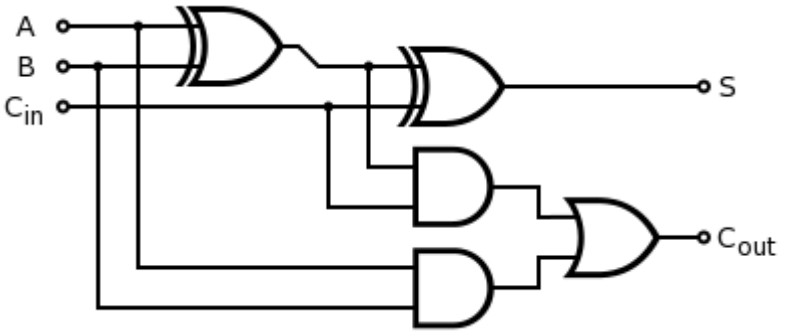
Wire data type can have one of four values: 0, 1, x, z. Wires *cannot store* values – they are continuously *driven*.

Primitive gates. Verilog provides several such, e.g., *and*, *or*, *nand*, *nor*, *not*, *buf*, etc.

1-bit adder

```
module adder1 (s, cout, a, b, cin);
    output s, cout;
    input a, b, cin;

    xor (t1, a, b);
    xor (s, t1, cin);
    and (t2, t1, cin),
        (t3, a, b);
    or  (cout, t2, t3);
endmodule
```




```
module adder4 (sum, carry, inA, inB);
    output [3:0] sum;
    output carry;
    input [3:0] inA, inB;

    adder1 a0 (sum[0], c0,    inA[0], inB[0], 1'b0);
    adder1 a1 (sum[1], c1,    inA[1], inB[1], c0);
    adder1 a2 (sum[2], c2,    inA[2], inB[2], c1);
    adder1 a3 (sum[3], carry, inA[3], inB[3], c2);
endmodule
```

Numeric Constants

- <Width in bits>'<base letter><number>
- 549 // decimal number
- 12'h123 - Hexadecimal 123 (using 12 bits)
- 4'b11 // 4-bit binary number 11 (0011)
- 3'b10x // 3-bit binary number with least significant bit unknown/don't care
- 12'o1345 // 12-bit octal number
- 5'd3 // 5-bit decimal number 3 (00011)
- -4'b11 // 4-bit two's complement negation of binary number 11 (1101)
- -4'd3 // 4-bit value decimal number -3 in two's complement (1101)

Numeric Constants (cont.)

- Values can also be concatenated by placing them within **{ }** separated by commas.
- The notation **{x {bit field}}** replicates bit field x times.
  - **{16{2'b01}}** creates a 32-bit value with the pattern 0101 . . . 01.
  - **{A[31:16],B[15:0]}** creates a value whose upper 16 bits come from A and whose lower 16 bits come from B.

Check Yourself

Which of the following define exactly the same value?

1. 8’b11110000
2. 8’hF0
3. 8’d240
4. {{4{1’b1}},{4{1’b0}}}
5. {4’b1,4’b0}

numbers.v

```
module numbers;
  integer i, j;
  reg[3:0] x, y;

  initial
    begin
    i = 'b1101;
    $display( "decimal i = %d, binary i = %b", i, i );
    $display( "octal i = %o, hex i = %h", i, i );

    j = -1;
    $display( "decimal j = %d, binary j = %b", j, j );
    $display( "octal j = %o, hex j = %h", j, j );

    x = 4'b1011;
    $display( "decimal x = %d, binary x = %b", x, x );
    $display( "octal x = %o, hex x = %h", x, x );

    y = 4'd7;
    $display( "decimal y = %d, binary y = %b", y, y );
    $display( "octal y = %o, hex y = %h", y, y );

    $finish;
    end
endmodule
```

Register array.

'<base>: base can be d, b, o, h

Default base: d

Array of register arrays simulate memory. Example memory declaration with 1K 32-bit words:
reg[31:0] smallMem[0:1023];

Negative numbers are stored in two's complement form.

Typical format:
<size>'<base><number>
size is a *decimal* value that specifies the size of the number in *bits*.

4-value logic

- Each bit can take on one of four possible values:
  - 0: logic 0
  - 1: logic 1
  - z: high impedance (for tri-state driver output)
  - x: unknown or undefined

Time Delays

- #i time delay expression

```
begin

...

#5 D = B; /* Time delay for 5 units */

...

end
```

Initial and always

- Two separate ways of declaring a Verilog process
- Initial
  - Begins at simulator time 0, executes once
- Always
  - Begins at simulator time 0, executes infinitely

Event Lists

The expression @(...) is called a sensitivity list. Statements preceded by a sensitivity list suspend execution until the event specifier's condition is satisfied.

any change in the value of variables
A or B or C

```
always @(A or B or C or negedge CLK or posedge Reset)
begin

...

#5 D = B; /* Time delay for 5 units */

...

end
```

the value of the CLK signal
makes the high-to-low transition (falling edge)
or the value of the Reset signal
takes the low-to-high transition (rising edge)

```
initial
  begin
    a = 1; // Assign a value to reg a at time 0
    #1; // Wait 1 time unit
    b = a; // Assign the value of reg a to reg b
  end
```

```
always @(a or b) // Any time a or b CHANGE, run the process
begin
  if (a)
    c = b;
  else
    d = ~b;
end // Done with this block, now return to the top (i.e. the @ event-control)
```

```
always @(posedge a)// Run whenever reg a has a low to high change
  a <= b;
```

Assignment

- "=" operator
  - Blocking assignment
  - The target variable is updated immediately
- "<=" operator
  - Non-blocking assignment
  - Its action doesn't register until the next clock cycle
- Assign keyword
  - Continuous assignment
  - Whenever the value of a variable on the right-hand side changes, the expression is re-evaluated and the value of the left-hand side is updated

Blocking Assignments

- "=" operator
- Are executed in spesific order

```
// blocking assignments
initial begin
   x = #5 1'b0;    // at time 5
   y = #3 1'b1;    // at time 8
   z = #6 1'b0;    // at time 14
end
```

Non-blocking Assignments

- "<=" operator
- Used to model several concurrent data transfers.

```
reg  x, y, z;
// nonblocking assignments
initial begin
   x <= #5 1'b0;  // at time 5
   y <= #3 1'b1;  // at time 3
   z <= #6 1'b0;  // at time 6
end
```

Example: Blocking and Non-Blocking

Assignments		
A = 3 ; B = 4 ; A <= 3 + 4 ; C = A ;	A = 3 ; B = 4 ; A = 3 + 4 ; C = A ;	A=2; A <= 3 ; B <= 4 ; C <= A ;
Results		
A = 7 ; B = 4 ; C = 3 ;	A = 7 ; B = 4 ; C = 7 ;	A = 3 ; B = 4 ; C = 2 ;

Non-Blocking → Parallel
Blocking → Sequential

blockingVSnba1.v

```
module blockingVSnba1;
  integer i, j, k, l;

  initial
    begin
    #1 i = 3;
    #1 i = i + 1;
    j = i +1;
    #1 $display( "i = %d, j = %d", i, j );

    #1 i = 3;
    #1 i <= i + 1;
    j <= i + 1;
    #1 $display( "i = %d, j = %d", i, j );

    $finish;
    end
endmodule
```

Blocking (procedural) assignment: the whole statement must execute before control is released, as in traditional programming languages.

Non-blocking (procedural) assignment: *all* the RHSs for the current time instant are evaluated (and stored transparently in temporaries) first and, subsequently, the LHSs are updated at the end of the time instant.

Datatypes in Verilog

- There are two primary datatypes in Verilog.
  - A wire specifies a combinational signal.
  - A reg (register) holds a value, which can vary with time.
  - An array of registers is used for a structure like a register file or memory.
  - Thus, the declaration

```
reg [31:0] registerfile[0:31]
```

    specifies a variable registerfile that is equivalent to a MIPS register file.

Data Type - wire

- wire (combinational logic)
  - used to connect input and output ports of a module instantiation
  - used as inputs and outputs within an actual module declaration
  - cannot store a value without being driven
  - cannot be used as the left-hand side of an **=** or **<=** sign in an **always**@ block
  - only legal type on the left-hand side of an **assign** statement.

Legal use of the wire element

```
1 wire        A, B, C, D, E; // simple 1-bit wide wires
2 wire [8:0] Wide;     // a 9-bit wide wire
3 reg I;
4
5 assign A = B & C;    // using a wire with an assign statement
6
7 always @(B or C) begin
8     I = B | C;       // using wires on the right-hand side of an always@
9                      // assignment
10 end
11
12 mymodule MyModule(.In (D), // using a wire as the input of a module
13                    .Out(E)); // using a wire as the output of a module
```

Data Type - reg (register)

- reg (combinational and sequential logic)
  - can be connected to the input port of a module instantiation
  - cannot be connected to the output port of a module instantiation
  - can be used as outputs within an actual module declaration
  - cannot be used as inputs within an actual module declaration
  - only legal type on the left-hand side of an **always**@ block **=** or **<=** sign
  - only legal type on the left-hand side of an **initial** block **=** sign
  - cannot be used on the left-hand side of an **assign** statement

Legal use of the reg element

```
1 wire      A, B;
2 reg       I, J, K;    // simple 1-bit wide reg elements
3 reg[8:0] Wide;        // a 9-bit wide reg element
4
5 always @(A or B) begin
6     I = A | B;        // using a reg as the left-hand side of an always@
7                       // assignment
8 end
9
10 initial begin        // using a reg in an initial block
11     J = 1'b1;
12     #1
13     J = 1'b0;
14 end
15
16 always @(posedge Clock) begin
17     K <= I;          // using a reg to create a positive-edge-triggered register
18 end
```

When wire and reg Elements are Interchangeable

- wire and reg elements can be used interchangeably in certain situations:

1. Both can appear on the right-hand side of **assign** statements and **always**@ block **=** or **<=** signs.

2. Both can be connected to the input ports of module instantiations.

Mux Example

1.

```
// The first example uses continuous assignment
wire out;
assign out = sel ? a : b;
```

2.

```
// the second example uses a procedure to accomplish the same thing.
reg out;
always @(a or b or sel)
  begin
    case(sel)
      1'b0: out = b;
      1'b1: out = a;
    endcase
  end
```

3.

```
// Finally - you can use if/else in a procedural structure.
reg out;
always @(a or b or sel)
  if (sel)
    out = a;
  else
    out = b;
```

4-to-1 multiplexor with 32-bit inputs

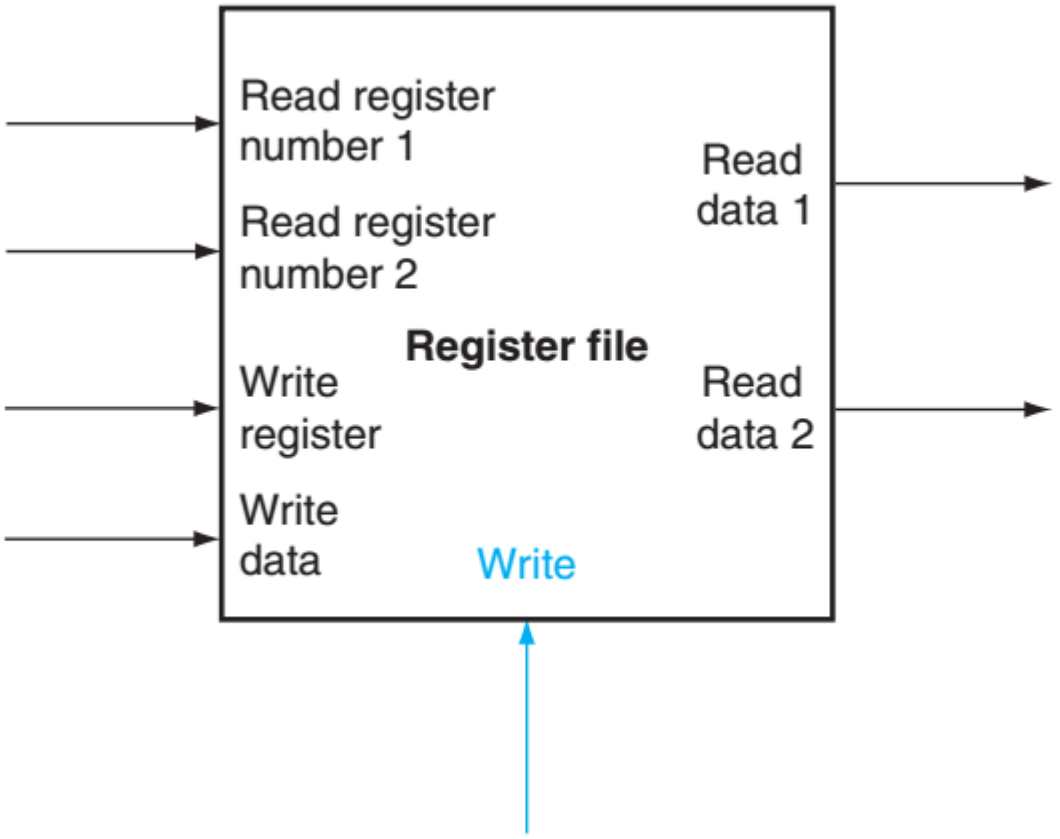
```
module Mult4to1 (In1,In2,In3,In4,Sel,Out);
   input [31:0] In1, In2, In3, In4; //four 32-bit inputs
   input [1:0] Sel; //selector signal
   output reg [31:0] Out;// 32-bit output
   always @(In1, In2, In3, In4, Sel)
   case (Sel) //a 4->1 multiplexor
      0: Out <= In1;
      1: Out <= In2;
      2: Out <= In3;
      default: Out <= In4;
   endcase
endmodule
```

Clock Generation

```
reg clock;

initial
    begin
    clock = 1'b0 ;
    forever
    # 10 clock = ~ clock
    end
```

Register File



A MIPS register file written in behavioral Verilog

```
module registerfile (Read1, Read2, WriteReg, WriteData, RegWrite,
Data1, Data2, clock);

    input[4:0] Read1, Read2, WriteReg; // the register to read or write
    input[31:0] WriteData; // data to werite
    input RegWrite, // the crite control
          clock; // the clock to trigger write

    output [31:0] Daat1, Data2; // the register values read
    reg [31:0] RF[31:0]; // 32 registers each 32 bits long

    assign Data1 = RF[Read1];
    assign Data2 = RF[Read2];

    // This register file writes on rising clock edge.
    always @(posedge clock)
    begin
    if(RegWrite)
        RF[WriteReg] <= WriteData;
    end
endmodule
```

System Tasks

- To handle I/O
  - $display - Print to screen a line followed by an automatic newline
  - $write - Write to screen a line without the newline
  - $sscanf - Read from variable a format-specified string
  - $fopen - Open a handle to a file (read or write)
  - $time - Value of current simulation time
  - ...

Verilog Simulators

- Software to emulate Verilog HDL
- Different commercial and open-source simulators
- ModelSim
  - Commercial
  - Free but limited student version

References

- http://www.cs.ait.ac.th/~guha/COA/Verilog/verilogSlides.ppt