Hardware Description Languages: Verilog

- Verilog
  - Structural Models
  - (Combinational) Behavioral Models
  - Syntax
  - Examples

- Everything runs at the same time (not as typical SW - python, C++..)
Typical Hardware
Design Methodology

Structure and Function (Behavior) of a Design

HDL Specification

Simulation
  Verification: Design Behave as Required?
  Functional: I/O Behavior

Synthesis
  Generation: Map Specification to Implementation
Verilog

- Supports structural and behavioral descriptions
- **Structural**
  - Explicit structure of the circuit
  - How a module is composed as an interconnection of more primitive modules/components
  - E.g., each logic gate instantiated and connected to others
- **Behavioral**
  - Program describes input/output behavior of circuit
  - Many structural implementations could have same behavior
  - E.g., different implementations of one Boolean function
Verilog Introduction

The module describes a component in the circuit

Two ways to describe:

- Structural Verilog
  - List of components and how they are connected
  - Just like schematics, but using text
  - Hard to write, hard to decode
  - Useful if you don’t have integrated design tools (today everybody have integrated design tools)

- Behavioral Verilog
  - Describe what a component does, not how it does it
  - Synthesized into a circuit that has this behavior
Structural Model

- Composition of primitive gates to form more complex module
- Note use of wire declaration!

module xor_gate (out, a, b);
    input     a, b;
    output    out;
    wire      abar, bbar, t1, t2;

    inverter invA (abar, a);
    inverter invB (bbar, b);
    and_gate and1 (t1, a, bbar);
    and_gate and2 (t2, b, abar);
    or_gate  or1 (out, t1, t2);
endmodule

By default, identifiers are wires
**Structural & Behavioral Model**

**Example of full-adder**

```verilog
module full_addr (A, B, Cin, S, Cout);
    input  A, B;
    input        Cin;
    output S;
    output       Cout;

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

```verilog
module full_adder (A, B, Cin, S, Cout);
    input A, B;
    input        Cin;
    output S;
    output       Cout;
    wire         temp1,temp2,temp3;

    and_gate (temp1, A, B);
    and_gate (temp2, A, Cin);
    and_gate (temp3, B, Cin);
    or_gate (Cout, temp1, temp2, temp3);
    xor_gate (S, A, B, Cin);
endmodule
```

Behavior

Structural

![Fig. 3.17 Implementation of full-adder](image-url)
Structural & Behavioral Model

Example of 4-bit full-adder

```verilog
module full_addr4 (A, B, Cin, S, Cout);
    input [3:0] A, B;
    input Cin;
    output [3:0] S;
    output Cout;

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

```verilog
module full_adder4 (A, B, Cin, S, Cout);
    input [3:0] A, B;
    input Cin;
    output [3:0] S;
    output Cout;
    wire C1, C2, C3;

    full_addr fa0 (A[0], B[0], Cin, S[0], C1);
    full_addr fa1 (A[1], B[1], C1, S[1], C2);
    full_addr fa2 (A[2], B[2], C2, S[2], C3);
    full_addr fa3 (A[3], B[3], C3, S[3], Cout);
endmodule
```
Simple Behavioral Model

- **Combinational logic**
  - Describe output as a function of inputs
  - Note use of `assign` keyword: *continuous assignment*

```verilog
module and_gate (out, in1, in2);
input         in1, in2;
output        out;
wire out ;//default
assign out = in1 & in2;
endmodule
```
Verilog Data Types and Values

- **Bits** - value on a wire
  - 0, 1
  - x - don't know/ driven by more than one gate to different value
  - z - undriven (like circuit powered down)

- **Vectors of bits**
  - Treated as an unsigned integer value
    - e.g., A < 0 ?
  - Concatenating bits/vectors into a vector
    - e.g., sign extend
  - Style: Use a[7:3] = b[7:3] + c;
Verilog Numbers

- 14 - ordinary decimal number
- -14 - 2's complement representation
- 12'b0000_0100_0110 - binary number with 12 bits (_ is ignored)
- 12'h046 - hexadecimal number with 12 bits

Verilog values are unsigned
- if A = 0110 (6) and B = 1010(10 not 6- )
  C = 10000 not 00000
# Verilog Operators

<table>
<thead>
<tr>
<th>Verilog Operator</th>
<th>Name</th>
<th>Functional Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>bit-select or part-select</td>
<td></td>
</tr>
<tr>
<td>( )</td>
<td>parenthesis</td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>logical negation</td>
<td>Logical</td>
</tr>
<tr>
<td>~</td>
<td>negation</td>
<td>Bit-wise</td>
</tr>
<tr>
<td>&amp;</td>
<td>reduction AND</td>
<td>Reduction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reduction OR</td>
</tr>
<tr>
<td>~&amp;</td>
<td>reduction NAND</td>
<td>Reduction</td>
</tr>
<tr>
<td>^</td>
<td>reduction NOR</td>
<td>Reduction</td>
</tr>
<tr>
<td>^ or ^~</td>
<td>reduction XOR</td>
<td>Reduction</td>
</tr>
<tr>
<td>^</td>
<td>reduction XNOR</td>
<td>Reduction</td>
</tr>
<tr>
<td>+</td>
<td>unary (sign) plus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>-</td>
<td>unary (sign) minus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>()</td>
<td>concatenation</td>
<td>Concatenation</td>
</tr>
<tr>
<td>{{}}</td>
<td>replication</td>
<td>Replication</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>+</td>
<td>binary plus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>-</td>
<td>binary minus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
<td>Shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
<td>Shift</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>Relational</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>Relational</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>Relational</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>Relational</td>
</tr>
<tr>
<td>==</td>
<td>logical equality</td>
<td>Equality</td>
</tr>
<tr>
<td>!=</td>
<td>logical inequality</td>
<td>Equality</td>
</tr>
<tr>
<td>===</td>
<td>case equality</td>
<td>Equality</td>
</tr>
<tr>
<td>!==</td>
<td>case inequality</td>
<td>Equality</td>
</tr>
<tr>
<td>&amp;</td>
<td>bit-wise AND</td>
<td>Bit-wise</td>
</tr>
<tr>
<td>^</td>
<td>bit-wise XOR</td>
<td>Bit-wise</td>
</tr>
<tr>
<td>~ or ~^</td>
<td>bit-wise XNOR</td>
<td>Bit-wise</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit-wise OR</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
<td>Logical</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>conditional</td>
<td>Conditional</td>
</tr>
</tbody>
</table>
Verilog “Variables”

- **wire**
  - Variable used simply to connect components together

- **reg**
  - Variable that saves a value as part of a behavioral description
  - Usually corresponds to a wire in the circuit
  - Is *NOT* necessarily a register in the circuit
Verilog Module

★ Corresponds to a circuit component
  ★ “Parameter list” is the list of external connections, aka “ports”
  ★ Ports are declared “input”, “output” or “inout”
    ★ inout ports used on tri-state buses
  ★ Port declarations imply that the variables are wires (default)

```verilog
module full_addr (A, B, Cin, S, Cout);
  input A, B, Cin;
  output S, Cout;
  assign {Cout, S} = A + B + Cin;
endmodule
```
Verilog Continuous Assignment

- Assignment is continuously evaluated
- `assign` corresponds to a connection or a simple component with the described function
- Target is NEVER a reg variable

```verilog
assign A = X | (Y & ~Z);
assign C[15:0] = 16'h00ff;
```

- Use of Boolean operators (`~` for bit-wise, `!` for logical negation)
- Variables can be n-bits wide (MSB:LSB)
- Multiple assignment (concatenation)
- Use of arithmetic operator
Comparator Example

module Compare1 (A, B, Equal, A_larger, B_larger);
    input [3:0] A, B;
    output Equal, A_larger, B_larger;

    assign Equal = A == B;
    assign A_larger = A > B;
    assign B_larger = A< B;
endmodule
Simple Behavioral Model: the always block

★ always block

▲ Always waiting for a change to a trigger signal
▲ Then executes the body

module and_gate (out1, out2, out3, clk,in1, in2);
  input in1, in2, clk;
  output out1 , out2, out3;
  reg out1 , out2;
  always @(posedge clk) begin
    out1 <= in1 & in2;
  end
  always @(in1 or in2) begin
    out2 = in1 & in2;
  end
assign out3 = in1 & in2;
endmodule
Incomplete Triggers - dangerous for the novice!

- NOT combinational logic ⇒ latch is inserted (implied memory)
- Variable keeps its old value

```verilog
module and_gate (out, in1, in2);
    input    in1, in2;
    output   out;
    reg      out;

    always @(in1) begin
        out = in1 & in2;
    end
endmodule
```

Any variable assigned in an always block should be assigned for any (and every!) execution of the block
Verilog if

- Same as C if statement

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

always @(select or A or B or C or D)
  if (select == 2'b00) Y = A;
  else if (select == 2'b01) Y = B;
  else if (select == 2'b10) Y = C;
  else if (select == 2'b11) Y = D;
endmodule
```
Verilog 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[1] == 0)
    if (sel[0] == 0) Y = A;
    else             Y = B;
  else
    if (sel[0] == 0) Y = C;
    else             Y = D;
endmodule
```
Verilog case

- **Sequential execution of cases**
- Only first case that matches is executed (implicit break)
- Default case can be used

```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)
    case (sel)
        2’b00: Y = A;
        2’b01: Y = B;
        2’b10: Y = C;
        2’b11: Y = D;
    endcase
endmodule
```

Conditions tested in top to bottom order
Verilog case

- Without the default case, this example would create a latch for Y
- Assigning X to a variable means synthesis is free to assign any value

```verilog
// Simple binary encoder (input is 1-hot)
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'bXXX; // Don’t care when input is not 1-hot
endcase
endmodule
```