Hardware Modeling

Verilog is a descriptive language that describes the *relationship between signals* in a circuit, and is *not* a computational program.

Verilog also has a semantic of *time* associated with signals, because it needs to model their temporal relationships and evolution.

Module Declaration:

module_declaration ::= *module_keyword module_id* [list_of_ports]
{module_item}
endmodule

module_keyword ::= **module** | **macromodule**

module_item ::=
module_item_declaration
| parameter_override
| continuous_assign
| gate_instantiation
| upd_instantiation
| module_instantiation
| specify_block
| initial_construct
| always_construct

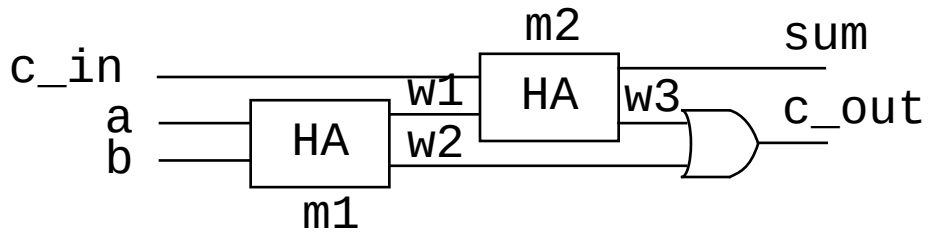
Verilog Module Instantiation

Module ports can be scalar or vector objects (one dimensional).

Keywords to classify ports include **input**, **output** and **inout**

Modules can be instantiated in parents.

```
module half_adder (sum, c_out, a, b);
 input a, b;
 output sum,c_out;
 wire c_out_bar;
 xor (sum, a, b);
 nand (c_out_bar, a, b);
 not (c_out, c_out_bar);
endmodule
```




```
module full_adder (sum, c_out, a, b, c_in);
 input a, b, c_in;
 output sum, c_out;
 wire w1, w2, w3;
 half_adder m1(w1, w2, a, b);
 half_adder m2(sum, w3, w1, c_in);
 or (c_out, w2, w3);
endmodule
```

Verilog Primitives

The predefined primitives implement the behavior of a combinational logic function or transistor level switch.

- Combinational Logic
  `and, nand, or, nor, xor, xnor, buf, not`
- Three state
  `bufif0, bufif1, notif0, notif1`
- MOS gates
  `nmos, pmos, rnmos, rpmos`
- CMOS gates
  `cmos, rcmos`
- Bi-directional gates
  `tran, tranif0, tranif1, rtran, rtranif0, rtranif1`
- Pull Gates
  `pulldown, pullup`

Note that the first port in any primitive is the *output* port.

Verilog Primitives

The primitives allow for *gate level* and *switch level* modeling.

They are idealized models because they ignore the time delays exhibited in real gates by default.

However, delays can be assigned when they are instantiated.

```
module AOI_4_unit (y_out, x_in1, x_in2, x_in3, x_in4);
  input x_in1, x_in2, x_in3, x_in4;
  output y_out;
  wire y1, y2;
  and #1 (y1, x_in1, x_in2);
  and #1 (y2, x_in3, x_in4);
  nor #1 (y_out, y1, y2);
endmodule
```

Note that these delays are ONLY used during simulation, i.e., they have no effect during synthesis.

Verilog Primitives

Timing is a very important property of a circuit and Verilog allows accurate timing information to be incorporated (if it is available).

For example, propagation delay may be different for rising and falling edges and the designer may want to simulate *worst*, *typical* and *best* case corners.

Process variations make each chip a little bit different.

```
module nanf201 (O, A1, B1);
  input A1, B1;
  output O;
  nand (O, A1, B1);
  specify
   specparam
     Tpd_0_1 = 1.13:3.09:7.75
     Tpd_1_0 = 0.93:2.50:7.34

     (A1 => O) = (Tpd_0_1, Tpd_1_0);
     (B1 => O) = (Tpd_0_1, Tpd_1_0);
   endspecify
endmodule
```

instantiation of the *nand*

min delay

typical delay

max delay

rising delay

rising delay

The individual path delays are given for each input-to-output path.

Verilog Primitives

Verilog primitives are "smart" because the same primitive, e.g., **nand**, can be used for a gate with any number of inputs.

For example, **nand**(O, A1, A2, A3) instantiates a *3-input* version of the Verilog **nand** gate primitive.

Explicit and Implicit Structural Description

As indicated, *explicit* structural descriptions are analogous to placing and wiring components on a schematic diagram.

We've seen examples of this style previously.

Implicit structural descriptions make use of Verilog built-in operators within the **continuous assignment statement.**

```
module nand2_RTL (y, x1, x2);
  input x1, x2;
  output y;

  assign y = x1 ~& x2;     Bitwise-nand

endmodule
```

Explicit and Implicit Structural Description

Here, the operators are not bound directly to physical gates.

The keyword **assign** declares a Verilog *continuous assignment*.

Continuous assignment corresponds to combinational logic, without requiring explicit instantiation of gates.

Think of these as "event scheduling rules".

In this example, the continuous assignment defines how output, *y*, depends on events that occur for signals *x1* and *x2*.

This style is also called a **data flow** or **RTL** description.

Continuous assignment can be made in 2 ways.

The first, as we have seen, uses the keyword **assign**:

cont_assign ::= **assign** [drive_strength][delay3] list_of_net_assignments;

Explicit and Implicit Structural Description

The second method creates a *continuous assignment* **implicitly** within the declaration.

```
module bit_or8_gate1 (y, a, b);
  input [7:0] a, b;
  output [7:0] y;
  wire [7:0] y = a|b;

endmodule
```

Port Connection Syntax

A connection to a port of a Verilog module can be made in 2 ways.

The first is by position, as we have seen, and the second is by naming.

```
module parent_mod;
  wire [3:0] g;
  child_mod ( .c(g[3],
     .d(g[2]), .b(g[0]),
     .a(g[1]));
endmodule
```

```
module child_mod (a, b, c, d);
  input a, b;
  output c, d;
        // Other stuff
endmodule
```

Behavioral Descriptions

Consist of procedural statements that define input-output signal relationships *without* reference to hardware or structure.

There are two basic styles of *behavioral description*.

- Register transfer level (**RTL**), defines input-output relationships in terms of dataflow operations on signals and register values.
- Abstract, algorithmic description of operations, that need not conform to a dataflow and which may *not* be synthesizable.

RTL/Data Flow Descriptions

Verilog language operators are used to define the flow of data.

For combinational logic, the *continuous assignment* statement is used (for *implicit structural model*).

```
module and4_rtl (y_out, x1, x2, x3, x4);
  input x1, x2, x3, x4;
  output y_out;
  assign y_out = x1 & x2 & x3 & x4;
endmodule
```

RTL/Data Flow Descriptions

A sequential example:

```
module FF (q, dat_in, clk, set, rst);
  input data_in, clk, set, rst;
  output q;
  reg q
  always @ (posedge clk)
   begin
   if ( rst == 0 ) q = 0;
   else
     if ( set == 0 ) q = 1;
     else
        q = data_in;
   end
endmodule
```

Here, the FF output, *q*, is updated **synchronously**, and retains its value between clk edges.

reg types retain an assigned value until another assignment is made to them (similar to variables in C).

They can be assigned a value ONLY by a procedural statement.

Algorithm-Based Descriptions

An algorithmic description of behavior assigns value to a register storage (**reg**) by executing *procedural statements*.

The procedural statements are those common in high-level languages.

```
module and4_algo (y_out, x_in);
 input [3:0] x_in;
 output y_out;
 reg y_out;
 integer k;
 always @ (x_in)
  begin: and_loop
   y_out = 1;
   for (k = 0; k <= 3; k = k + 1)
    if (x_in[k] == 0 )
     begin
      y_out = 0;
      disable and_loop;
     end
  end
endmodule
```

Data objects of type **reg** and **integer** can only be changed by a procedural stmt.

Algorithm-Based Descriptions

The **always** statement waits for an event on its event expression, i.e. for *x_in* to change value.

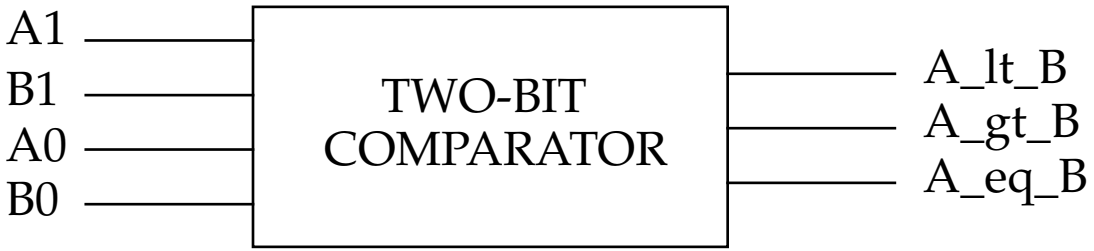
The @ operator informs the simulator to monitor changes on the event expression (in this case, if any of the bits with *x_in* change).

Verilog's **initial** statement is very similar to the **always** statement, except it only executes exactly once (does not wait and repeat like **always**).

Verilog for Synthesis

HDLs play a significant role in design fl ows that synthesize behavioral descriptions to gate level netlists.

A given functionality can be synthesized from a variety of descriptions.



Verilog for Synthesis

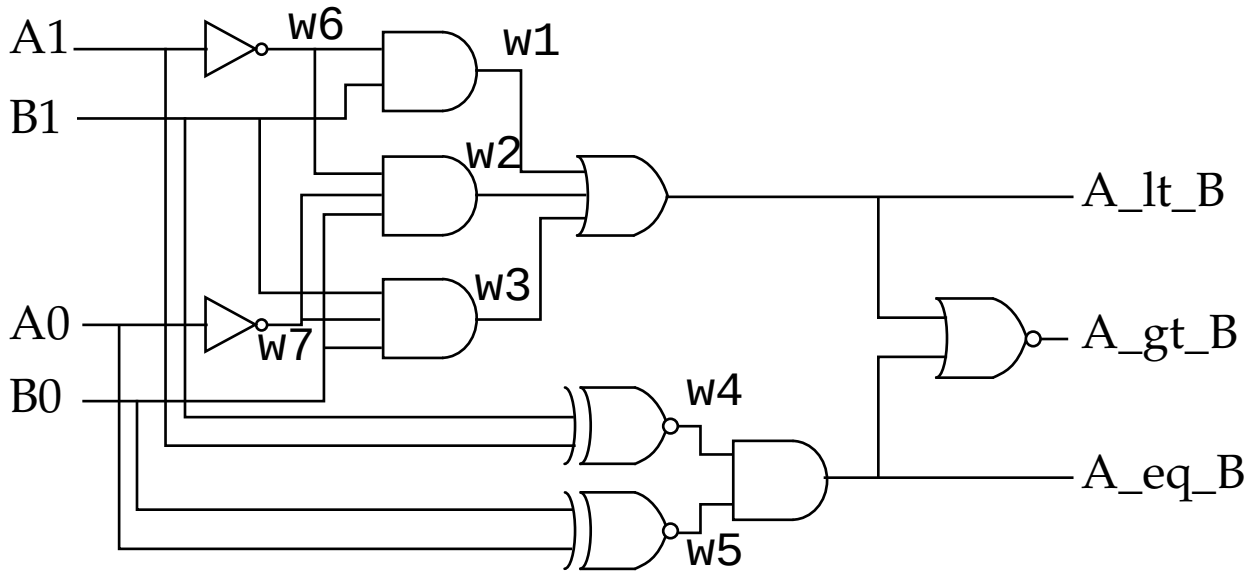
The functionality of the comparator can be represented by the following boolean expressions.

$A\_lt\_B = \overline{A1}\ B1 + \overline{A1}\ \overline{A0}\ B0 + \overline{A0}\ B1\ B0$

$A\_gt\_B = A1\ \overline{B1} + A0\ \overline{B1}\ \overline{B0} + A1\ A0\ \overline{B0}$

$A\_eq\_B = \overline{A1}\ \overline{A0}\ \overline{B1}\ \overline{B0} + \overline{A1}\ A0\ \overline{B1}\ B0 + A1\ A0\ B1\ B0 + A1\ \overline{A0}\ B1\ \overline{B0}$

Using Karnaugh maps to reduce yields



Verilog for Synthesis

The structural Verilog description can be derived directly from the schematic.

```
module compare_2_str (A_lt_B, A_gt_B, A_eq_B, A0, A1,
          B0, B1);
  input A0, A1, B0, B1;
  output A_lt_B, A_gt_B, A_eq_B;
  wire w1, w2, w3, w4, w5, w6, w7;

  or (A_lt_B, w1, w2, w3);
  nor (A_gt_B, A_lt_B, A_eq_B);
  and (A_eq_B, w4, w5);
  and (w1, w6, B1);
  and (w2, w6, w7, B0);
  and (w3, w7, B0, B1);
  not (w6, A1);
  not (w7, A0);
  xnor (w4, A1, B1);
  xnor (w5, A0, B0);
endmodule
```

Verilog for Synthesis

Alternatively, the comparator can be described by a Verilog RTL model using *continuous assignment*.

```
module compare_2a (A_lt_B, A_gt_B, A_eq_B, A0, A1,
            B0, B1);
  input A0, A1, B0, B1;
  output A_lt_B, A_gt_B, A_eq_B;

  assign A_lt_B = (~A1) & B1 | (~A1) & (~A0) & B0 |
              (~A0) & B1 & B0;
  assign A_gt_B = A1 & (~B1) | A0 & (~B1) & (~B0) |
              A1 & A0 & (~B0);
  assign A_eq_B = (~A1) & (~A0) & (~B1) & (~B0) |
              (~A1) & A0 & (~B1) & B0 |;
              A1 & (~A0) & B1 & (~B0) |;
              A1 & A0 & B1 & B0;
endmodule
```

Focus here is not on the detail but rather only on the input-output relationship.

Verilog for Synthesis

A simplier implementation that exploits Verilog operators.

```
module compare_2b (A_lt_B, A_gt_B, A_eq_B, A0, A1,
                   B0, B1);
  input A0, A1, B0, B1;
  output A_lt_B, A_gt_B, A_eq_B;

  assign A_lt_B = ({A1,A0} < {B1,B0})
  assign A_gt_B = {A1,A0} > {B1,B0})
  assign A_eq_B = {A1,A0} == {B1,B0})
endmodule
```

Verilog for Synthesis

The algorithmic model

```
module compare_2_algo (A_lt_B, A_gt_B, A_eq_B, A, B);
  input [1:0] A,B;
  output A_lt_B, A_gt_B, A_eq_B;
  reg A_lt_B, A_gt_B, A_eq_B;

  always @ (A or B)
  begin
    A_lt_B = 0; A_gt_B = 0; A_eq_B = 0;
    if (A == B) A_eq_B = 1;
    else if (A > B) A_gt_B = 1;
    else A_lt_B = 1;
  end
endmodule
```