Design Using Verilog
Basic Verilog

- Lexical Convention
- Lexical convention are close to C++.
- Comment
  - // to the end of the line.
  - /* to */ across several lines
- Keywords are lower case letter & it is case sensitive
- VERILOG uses 4 valued logic: 0, 1, x and z
- **Comments:** // Verilog code for AND-OR-INVERT gate

```verilog
module <module_name> (<module_terminal_list>);
<module_terminal_definitions>
...
<functionality_of_module>
...
endmodule
```
Taste of Verilog

```
module Add_half ( sum, c_out, a, b );
    input a, b;
    output sum, c_out;
    wire c_out_bar;
    xor (sum, a, b);
    // xor G1(sum, a, b);
    nand (c_out_bar, a, b);
    not (c_out, c_out_bar);
endmodule
```

Verilog keywords

- module
- input
- output
- wire
- xor
- nand
- not
Lexical Convention

- Numbers are specified in the traditional form or below.
  \[ \langle \text{size} \rangle \langle \text{base format} \rangle \langle \text{number} \rangle \]
- Size: contains decimal digitals that specify the size of the constant in the number of bits.
- Base format: is the single character \( \{' \) followed by one of the following characters \( b(\text{binary}), d(\text{decimal}), o(\text{octal}), h(\text{hex}) \).
- Number: legal digital.

Example:
- 347 -- decimal number
- 4’b101 -- 4-bit \( 0101_2 \)
- 2’o12 -- 2-bit octal number
- 5’h87f7 -- 5-digit \( 87F7_{16} \)
- 2’d83 -- 2-digit decimal
- String in double quotes “this is an introduction”
Three Modeling Styles in Verilog

- **Structural modeling (Gate-level)**
  - Use predefined or user-defined primitive gates.

- **Dataflow modeling**
  - Use assignment statements (assign)

- **Behavioral modeling**
  - Use procedural assignment statements (always)
Structural Verilog Description of Two-Bit Greater-Than Circuit

```verilog
// Two-bit greater-than circuit: Verilog structural model
// See Figure 2-27 for logic diagram

module comparator_greater_than_structural(A, B, A_greater_than_B);

input [1:0] A, B;
output A_greater_than_B;
wire B0_n, B1_n, and0_out, and1_out, and2_out;

not
  inv0(B0_n, B[0]), inv1(B1_n, B[1]);

and
  and0(and0_out, A[1], B1_n),
  and1(and1_out, A[1], A[0], B0_n),
  and2(and2_out, A[0], B1_n, B0_n);

or
  or0(A_greater_than_B, and0_out, and1_out, and2_out);
endmodule
```

![Diagram of two-bit greater-than circuit]
Dissection

• **Module and Port declarations**
  • Verilog-2001 syntax
    • `module AOI (input A, B, C, D, output F);`
  • Verilog-1995 syntax
    module AOI (A, B, C, D, F);
    input A, B, C, D;
    output F;

• **Wires**: Continuous assignment to an internal signal
A Simple Dataflow Design

// Verilog code for AND-OR-INVERT gate
module AOI (input A, B, C, D, output F);
   wire F; // the default
   wire AB, CD, O; // necessary
   assign AB = A & B;
   assign CD = C & D;
   assign O = AB | CD;
   assign F = ~O;
endmodule

// end of Verilog code
// Verilog code for AND-OR-INVERT gate
module AOI (input A, B, C, D, output F);
    assign F = ~(A & B) | (C & D);
endmodule

// end of Verilog code

‘&’ for AND, ‘|’ for OR, ‘^’ for XOR ‘^~’ for XNOR, ‘&~’ for NAND
Dataflow Verilog Description of Two-Bit Greater-Than Comparator

```verilog
// Two-bit greater-than circuit: Dataflow model
// See Figure 2-27 for logic diagram
module comparator_greater_than_dataflow(A, B, A_greater_than_B);
    input [1:0] A, B;
    output A_greater_than_B;
    wire B1_n, B0_n, and0_out, and1_out, and2_out;
    assign B1_n = ~B[1];
    assign B0_n = ~B[0];
    assign and0_out = A[1] & B1_n;
    assign and1_out = A[1] & A[0] & B0_n;
    assign and2_out = A[0] & B1_n & B0_n;
    assign A_greater_than_B = and0_out | and1_out | and2_out;
endmodule
```
// Two-bit greater-than circuit: Conditional model  // 1
// See Figure 2-27 for logic diagram  // 2
module comparator_greater_than_conditional2(A, B, A_greater_than_B);  // 3
  input [1:0] A, B;  // 4
  output A_greater_than_B;  // 5
  assign A_greater_than_B = (A > B)? 1'b1 :  // 6
    1'b0;  // 7
endmodule  // 8

Copyright ©2016 Pearson Education, All Rights Reserved
Verilog Description of Two-Bit Greater-Than Circuit

// Two-bit greater-than circuit: Behavioral model // 1
// See Figure 2-27 for logic diagram // 2
module comparator_greater_than_behavioral(A, B, A_greater_than_B); // 3
    input [1:0] A, B; // 4
    output A_greater_than_B; // 5
    assign A_greater_than_B = A > B; // 6
endmodule // 7

Copyright ©2016 Pearson Education, All Rights Reserved
A Design Hierarchy

- **Module Instances**
  - MUX_2 module contains references to each of the lower level modules

```
// Verilog code for 2-input multiplexer
module MUX2 (input SEL, A, B, output F);
// 2:1 multiplexer
// wires SELB and FB are implicit
// Module instances...
INV G1 (SEL, SELB);
AOI G2 (SELB, A, SEL, B, FB);
INV G3 (.A(FB), .F(F));  // Named mapping
endmodule
// end of Verilog code
```

1. Invert SEL and get SELB
2. Use AOI and get F’
3. Invert F’ and get F
Another Example

module decoder (A,B, D0,D1,D2,D3);
input A,B;
output D0,D1,D2,D3;
assign D0 = \sim A\&\sim B;
assign D1 = \sim A\&B;
assign D2 = A\&\sim B;
assign D3 = A\&B;
endmodule

Figure 6. Logic diagram of 2-to-4 decoder
module fulladder (A, B, CIN, S, COUT);

input A, B, CIN;

output S, COUT;

assign S = A ^ B ^ CIN;

assign COUT = (A & B) | (A & CIN) | (B & CIN);

endmodule
module four_bit_adder (CIN, X3, X2, X1, X0, Y3, Y2, Y1, Y0, S3, S2, S1, S0, COUT);
input  CIN, X3, X2, X1, X0, Y3, Y2, Y1, Y0;
output S3, S2, S1, S0, COUT;
wire   C1, C2, C3;
fulladder FA0 (X0, Y0, CIN, S0, C1);
fulladder FA1 (X1, Y1, C1, S1, C2);
fulladder FA2 (X2, Y2, C2, S2, C3);
fulladder FA3 (X3, Y3, C3, S3, COUT);
endmodule

Figure 9. Four bit Full Adder
module adder_4 (A, B, CIN, S, COUT);
input [3:0] A,B;
input CIN;
output [3:0] S;
output COUT;
wire [4:0] C;
full_adder FA0 (B(0), A(0), C(0), S(0), C(1));
full_adder FA1 (B(1), A(1), C(1), S(1), C(2));
full_adder FA2 (B(2), A(2), C(2), S(2), C(3));
full_adder FA3 (B(3), A(3), C(3), S(3), C(4));
assign C(0) = CIN;
assign COUT = C(4);
endmodule

Figure 9. Four bit Full Adder
Verilog Statements

Verilog has two basic types of statements

1. Concurrent statements (combinational)
   (things are happening concurrently, ordering does not matter)
   - Gate instantiations
     - \texttt{and} (z, x, y), \texttt{or} (c, a, b), \texttt{xor} (S, x, y), etc.
   - Continuous assignments
     - \texttt{assign} \( Z = x \& y; c = a \mid b; S = x \^ y \)

2. Procedural statements (sequential)
   (executed in the order written in the code)
   - \texttt{always @} - executed continuously when the event is active
   - \texttt{Initial} - executed only once (used in simulation)
   - \texttt{if then else} statements
module Add_half ( sum, c_out, a, b );

input a, b;
output sum, c_out;
reg sum, c_out;

always @ ( a or b )
begin
    sum = a ^ b;  // Exclusive or
    c_out = a & b;  // And
end
endmodule
Conditional Statement

- Conditional_expression ? true_expression : false_expression;

Example:
- Assign $A = (B < C) ? (D + 5) : (D + 2)$;
  - if $B$ is less than $C$, the value of $A$ will be $D + 5$, or else $A$ will have the value $D + 2$.

- An **if-else** statement is a procedural statement.

```verilog
// Behavioral specification
module mux2to1 (w0, w1, s, F);
input w0, w1, s;
output F;
reg F;

always @(w0, w1, s)
  if (s == 1) F = w1;
  else F = w0;
endmodule
```

```verilog
// Behavioral specification
module mux2to1 (w0, w1, s, F);
input w0, w1, s;
output F;
reg F;

always @(w0, w1, s)
  F = s ? w1 : w2;
endmodule
```
Mux 4-to-1

module mux4to1 (w0, w1, w2, w3, S, F);
input w0, w1, w2, w3, [1:0] S;
output F;
reg F;
always @ (w0, w1, w2, w3, S)
if (S==0) F = w0;
else if (S==1) F = w1;
else if (S==2) F = w2;
else F = w3;
endmodule
<table>
<thead>
<tr>
<th>Verilog Operator</th>
<th>Name</th>
<th>Functional Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;    &gt;=    &lt;    &lt;=</td>
<td>greater than greater than or equal to less than or equal to</td>
<td>relational</td>
</tr>
<tr>
<td>==   !=</td>
<td>case equality case inequality</td>
<td>equality</td>
</tr>
<tr>
<td>&amp;    ^</td>
<td>bit-wise AND bit-wise XOR bit-wise OR</td>
<td>bit-wise bit-wise</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Another Example

// Dataflow description of a 4-bit comparator.
module mag_comp (A,B,ALTB,AGTB,AEQB);
input [3:0] A,B;
output ALTB,AGTB,AEQB;
assign ALTB = (A < B),
AGTB = (A > B),
AEQB = (A == B);
endmodule
// Dataflow description of 4-bit adder
module binary_adder (A, B, Cin, SUM, Cout);
input [3:0] A, B;
input Cin;
output [3:0] SUM;
output Cout;
assign {Cout, SUM} = A + B + Cin;
endmodule
Design of an ALU using Case Statement

<table>
<thead>
<tr>
<th>S</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Clear</td>
</tr>
<tr>
<td>1</td>
<td>B - A</td>
</tr>
<tr>
<td>2</td>
<td>A - B</td>
</tr>
<tr>
<td>3</td>
<td>A + B</td>
</tr>
<tr>
<td>4</td>
<td>A XOR B</td>
</tr>
<tr>
<td>5</td>
<td>A OR B</td>
</tr>
<tr>
<td>6</td>
<td>A AND B</td>
</tr>
<tr>
<td>7</td>
<td>Set to all 1's</td>
</tr>
</tbody>
</table>

```plaintext
// 74381 ALU
module alu(s, A, B, F);
  input [2:0] s;
  input [3:0] A, B;
  output [3:0] F;
  reg [3:0] F;
  always @(s or A or B)
    case (s)
      0: F = 4'b0000;
      1: F = B - A;
      2: F = A - B;
      3: F = A + B;
      4: F = A ^ B;
      5: F = A | B;
      6: F = A & B;
      7: F = 4'b1111;
    endcase
endmodule
```
module VALU(s, A, B, F);
input [2:0] s;
input [3:0] A, B;
output [3:0] F;
reg [3:0] F;
always @(s or A or B)
case (s)
  0: F = 4'b0000;
  1: F = B - A;
  2: F = A - B;
  3: F = A + B;
  4: F = A ^ B;
  5: F = A | B;
  6: F = A & B;
  7: F = 4'b1111;
endcase
endmodule
Blocking vs. Nonblocking Assignments

- Verilog supports two types of assignments within *always* blocks, with subtly different behaviors.

  **Blocking assignment:** evaluation and assignment are immediate

  ```verilog
  always @(a or b or c)
  begin
  x = a | b;  // 1. Evaluate a | b, assign result to x
  y = a ^ b ^ c;  // 2. Evaluate a^b^c, assign result to y
  z = b & ~c;  // 3. Evaluate b&(~c), assign result to z
  end
  ```

- **Nonblocking assignment:** all assignments deferred until all right-hand sides have been evaluated (end of simulation timestep)

  ```verilog
  always @(a or b or c)
  begin
  x <= a | b;  // 1. Evaluate a | b but defer assignment of x
  y <= a ^ b ^ c;  // 2. Evaluate a^b^c but defer assignment of y
  z <= b & ~c;  // 3. Evaluate b&(~c) but defer assignment of z
  end
  ```

- Sometimes, as above, both produce the same result. Sometimes, not!
Blocking vs. Nonblocking Assignments

➢ The = token represents a blocking procedural assignment
   ✓ Evaluated and assigned in a single step
   ✓ Execution flow within the procedure is blocked until the assignment is completed

➢ The <= token represents a non-blocking assignment
   ✓ Evaluated and assigned in two steps:
     1. The right hand side is evaluated immediately
     2. The assignment to the left-hand side is postponed until other evaluations in the current time step are completed

//swap bytes in word
always @(posedge clk)
begin
  word[15:8] = word[ 7:0];
  word[ 7:0] = word[15:8];
end

//swap bytes in word
always @(posedge clk)
begin
  word[15:8] <= word[ 7:0];
  word[ 7:0] <= word[15:8];
end
Why two ways of assigning values?

Conceptual need for **two kinds** of assignment (in always blocks):

<table>
<thead>
<tr>
<th>Blocking: Evaluation and assignment are immediate</th>
<th>Non-Blocking: Assignment is postponed until all r.h.s. evaluations are done</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>a = b</strong></td>
<td><strong>x &lt;= a &amp; b</strong></td>
</tr>
<tr>
<td><strong>b = a</strong></td>
<td><strong>x &lt;= a &amp; b</strong></td>
</tr>
<tr>
<td>**y = x</td>
<td>c**</td>
</tr>
</tbody>
</table>

When to use: (only in always blocks!)

<table>
<thead>
<tr>
<th>Sequential Circuits</th>
<th>Combinational Circuits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>a &lt;= b</strong></td>
<td><strong>x &lt;= a &amp; b</strong></td>
</tr>
<tr>
<td><strong>b &lt;= a</strong></td>
<td>**y &lt;= x</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Golden Rules

- **Golden Rule 1:**
  - To synthesize combinational logic using an always block, all inputs to the design must appear in the sensitivity list.

- **Golden Rule 2:**
  - To synthesize combinational logic using an always block, all variables must be assigned under all conditions.
Golden Rules

reg f;
always @ (sel, a, b)
begin :
  if (sel == 1)
    f = a;
  else
    f = b;
end

• Proper as intended

Reg f;
always @ (sel, a, b)
begin f = a;
  if (sel == 1)
    f = a;
  end

• Setting variables to default values at the start of the always block

• OK as well!

Reg f;
always @ (sel, a)
begin :
  if (sel == 1)
    f = a;
  end

• What if sel = 0?
  • Keep the current value
  • Undesired functionality
  • Unintended latch
  • Need to include else
<table>
<thead>
<tr>
<th>Verilog Operator</th>
<th>Name</th>
<th>Functional Group</th>
<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>
<td>+</td>
<td>binary plus</td>
<td>arithmetic</td>
</tr>
<tr>
<td></td>
<td>select</td>
<td></td>
<td>-</td>
<td>binary minus</td>
<td>arithmetic</td>
</tr>
<tr>
<td>()</td>
<td>parenthesis</td>
<td></td>
<td>&lt;&lt;</td>
<td>shift left</td>
<td>shift</td>
</tr>
<tr>
<td>!</td>
<td>logical negation</td>
<td>logical</td>
<td>&gt;&gt;</td>
<td>shift right</td>
<td>shift</td>
</tr>
<tr>
<td>~</td>
<td>negation</td>
<td>bit-wise</td>
<td>&gt;</td>
<td>greater than</td>
<td>relational</td>
</tr>
<tr>
<td>&amp;</td>
<td>reduction AND</td>
<td>reduction</td>
<td>&gt;=</td>
<td>greater than or equal</td>
<td>relational</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reduction OR</td>
<td>reduction</td>
<td>&lt;</td>
<td>to</td>
</tr>
<tr>
<td>~&amp;</td>
<td>reduction NAND</td>
<td>reduction</td>
<td>&lt;=</td>
<td>less than</td>
<td>relational</td>
</tr>
<tr>
<td>^</td>
<td>reduction XOR</td>
<td>reduction</td>
<td></td>
<td>less than or equal to</td>
<td></td>
</tr>
<tr>
<td>^ or ^~</td>
<td>reduction XNOR</td>
<td>reduction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>unary (sign) plus</td>
<td>arithmetic</td>
<td>&amp;&amp;</td>
<td>logical AND</td>
<td>logical</td>
</tr>
<tr>
<td>-</td>
<td>unary (sign) minus</td>
<td>arithmetic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{ }</td>
<td>concatenation</td>
<td>concatenation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{{ }}</td>
<td>replication</td>
<td>replication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
<td>arithmetic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>arithmetic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
<td>arithmetic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>?:</td>
<td>conditional</td>
<td>conditional</td>
</tr>
</tbody>
</table>
Arithmetic in Verilog

module Arithmetic (A, B, Y1, Y2, Y3, Y4, Y5);

    input [2:0] A, B;
    output [3:0] Y1;
    output [4:0] Y3;
    output [2:0] Y2, Y4, Y5;
    reg [3:0] Y1;
    reg [4:0] Y3;
    reg [2:0] Y2, Y4, Y5;

    always @(A or B)
    begin
        Y1 = A + B; // addition
        Y2 = A - B; // subtraction
        Y3 = A * B; // multiplication
        Y4 = A / B; // division
        Y5 = A % B; // modulus of A divided by B
    end

endmodule
module Sign (A, B, Y1, Y2, Y3);

input [2:0] A, B;
output [3:0] Y1, Y2, Y3;
reg [3:0] Y1, Y2, Y3;

always @(A or B)
begin
    Y1 = +A / -B;
    Y2 = -A + -B;
    Y3 = A * -B;
end
endmodule
Equality and inequality Operations in Verilog

```verilog
module Equality (A, B,Y1,Y2,Y3);
    input [2:0] A, B;
    output Y1,Y2;
    output [2:0] Y3;
    reg Y1,Y2;
    reg [2:0]Y3;
    always @(A or B)
    begin
        Y1=A==B; //Y1=1 if A equivalent to B
        Y2=A!=B; //Y2=1 if A not equivalent to B
        if (A==B) //parenthesis needed
            Y3=A;
        else
            Y3=B;
    end
endmodule
```
module Logical (A, B, C, D, E, F, Y);

    input [2:0] A, B, C, D, E, F;
    output Y;
    reg Y;

    always @(A or B or C or D or E or F)
    begin
        if ((A==B) && ((C>D) || !(E<F)))
            Y=1;
        else
            Y=0;
    end

endmodule
Bit-wise Operations in Verilog

module Bitwise (A, B, Y);

input [6:0] A;
input [5:0] B;
output [6:0] Y;
reg [6:0] Y;

always @(A or B)
begin

  Y[0] = A[0] & B[0]; // binary AND

end

endmodule
Concatenation and Replication in Verilog

- The concatenation operator "\{ , \}" combines (concatenates) the bits of two or more data objects. The objects may be scalar (single bit) or vectored (multiple bit). Multiple concatenations may be performed with a constant prefix and is known as replication.

```verilog
module Concatenation (A, B, Y);
    input [2:0] A, B;
    output [14:0] Y;
    parameter C=3'b011;
    reg [14:0] Y;
    always @(A or B)
    begin
        Y={A, B, {2{C}}, 3'b110};
    end
endmodule
```
Shift Operations in Verilog

module Shift (A,Y1,Y2);

    input [7:0] A;
    output [7:0] Y1,Y2;

    parameter B=3; reg [7:0] Y1,Y2;

    always @(A)
    begin
        Y1=A<<B; //logical shift left
        Y2=A>>B; //logical shift right
    end
endmodule
module Conditional (Time,Y);
    input [2:0] Time;
    output [2:0] Y;
    reg [2:0] Y;
    parameter Zero = 3b'000;
    parameter TimeOut = 3b'110;
    always @(Time)
    begin
        Y = (Time != TimeOut) ? Time + 1 : Zero;
    end
endmodule
module Reduction (A, Y1, Y2, Y3, Y4, Y5, Y6);

  input [3:0] A;
  output Y1, Y2, Y3, Y4, Y5, Y6;
  reg Y1, Y2, Y3, Y4, Y5, Y6;

  always @(A)
  begin
    Y1 = &A; // reduction AND
    Y2 = | A; // reduction OR
    Y3 = ~&A; // reduction NAND
    Y4 = ~| A; // reduction NOR
    Y5 = ^A; // reduction XOR
    Y6 = ~^A; // reduction XNOR
  end

endmodule
**Fig. 4-33  Stimulus and Design Modules Interaction**

Stimulus module

```
module testcircuit

reg TA, TB;
wire TC;

endmodule
```

Design module

```
module circuit (A, B, C);

input A, B;

output C;

endmodule
```
Testbench for the Structural Model of the Two-Bit Greater-Than Comparator

// Testbench for Verilog two-bit greater-than comparator
module comparator_testbench_verilog();
    reg [1:0] A, B;
    wire struct_out;
    comparator_greater_than_structural U1(A, B, struct_out);
initial
begin
    A = 2’b10;
    B = 2’b00;
    #10;
    B = 2’b01;
    #10;
    B = 2’b10;
    #10;
    B = 2’b11;
end
endmodule
Propagation Delay for an Inverter

\[ t_{pd} = \max (t_{PHL}, t_{PLH}) \]
Circuit to demonstrate an HDL (Verilog)

Module smpl_Circuit (A, B, C, D, E)
  input A, B, C;
  output D, E;
  wire w1;
  and # (30) G1 (w1, A, B);
  not #10 G2 (E, C);
  or #(20) G3 (D, w1, E);
endmodule

<table>
<thead>
<tr>
<th>Time Units (ns)</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ABC</td>
<td>E w1 D</td>
</tr>
<tr>
<td>Initial</td>
<td>0 0 0</td>
<td>1 0 1</td>
</tr>
<tr>
<td>Change</td>
<td>1 1 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>10</td>
<td>1 1 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>20</td>
<td>1 1 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>30</td>
<td>1 1 1</td>
<td>0 1 0</td>
</tr>
<tr>
<td>40</td>
<td>1 1 1</td>
<td>0 1 0</td>
</tr>
<tr>
<td>50</td>
<td>1 1 1</td>
<td>0 1 1</td>
</tr>
</tbody>
</table>
Interaction between stimulus and design modules

```verilog
module t_circuit;
    reg t_A, t_B;
    wire t_C;
    parameter stop_time = 1000;

    circuit M ( t_C, t_A, t_B);

    // Stimulus generators for
    // t_A and t_B go here
    initial # stop_time $finish;
endmodule
```

```verilog
module circuit (C, A, B);
    input A, B;
    output C;

    // Description goes here
endmodule
```