Advanced Verilog Coding

Part 1

Behavioral Modeling
Structured Procedures

2 structured procedure

- **initial**
- **always**

Basic statements in behavioral modeling

Verilog is a concurrent programming language.

Activity flows in Verilog run in parallel.

- **initial**
- **always**

Starts at simulation time 0

C Language

1. `a = 3;
2. `b = 2;
3. `c = a + b;

Verilog Language

- `initial c = a + b;
- `initial a = 3;
- `initial b = 2;

Verilog HDL

© Verilog HDL
Design & Simulation

YANG S.W.
yangsw@thu.edu.tw

---

**initial Statement**

- `initial begin
  - `<statement 1>...
  - `<statement 2>...
  - `<statement 3>...
  - `end

Starts at time 0

Executes exactly once during a simulation

<table>
<thead>
<tr>
<th>initial block</th>
</tr>
</thead>
</table>
| `initial a = 3;
| `initial b = 1;
| `initial begin
  - `c = 2;
  - `#5 c = a + b;
  - `end

YANG S.W.
yangsw@thu.edu.tw

---

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>5</th>
<th>10</th>
<th>....</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
module stimulus;
reg x,y, a,b, m;

initial
m = 1'b0;

initial
begin
#5   a = 1'b1;
#25 b = 1'b0;
end

initial
begin
#10 x = 1'b0;
#25 y = 1'b1;
end

initial
#50 $finish;
endmodule

//The clock variable is defined first
reg clock;
//The value of clock is set to 0
initial clock = 0;

//Instead of the above method, clock variable
//can be initialized at the time of declaration
//This is allowed only for variables declared
//at module level.
reg clock = 0;
always Statement

always begin
  ...<statement 1>...
  ...<statement 2>...
  ...<statement 3>...
end

starts at time 0

always block

executes exactly once during a simulation (initial)
difference

This statement is used to model a block of activity that is repeated continuously in a digital circuit

module clock_gen (output reg clock);
//Initialize clock at time zero
initial
  clock = 1'b0;
//Toggle clock every half-cycle (time period = 20)
always

  #10 clock = ~clock;
initial
  #1000 $finish;
endmodule

ANSI C Style

ANSI C style

module clock_gen (output reg clock);
//Initialize clock at time zero
initial
  clock = 1'b0;
//Toggle clock every half-cycle (time period = 20)
always

  #10 clock = ~clock;
initial
  #1000 $finish;
endmodule

General Style

module clock_gen (clock);
output clock;
reg clock;
//Initialize clock at time zero
initial
  clock = 1'b0;
//Toggle clock every half-cycle (time period = 20)
always

  #10 clock = ~clock;
initial
  #1000 $finish;
endmodule

ANSI C style General Style
Procedural Assignments

Procedural assignments update values of `reg`, `integer`, `real`, or `time` variables.

The value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value.

\[
\text{variable_value = [ delay_or_event_control ] Expression}
\]

\[
#3 \ldots \text{(delay)} \quad @(x) \ldots \text{(event)}
\]

- A reg, integer, real, or time register variable or a memory element
- A bit select of these variables (e.g., `addr[0]`)
- A part select of these variables (e.g., `addr[31:16]`)
- A concatenation of any of the above `{ }`

Procedural Assignments

**blocking** statement

The `=` operator is used to specify blocking assignments.

Blocking assignment statements are executed in the order they are specified in a sequential block.

**non-blocking** statement

Nonblocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block. (parallel execute)

A `<=` operator is used to specify non-blocking assignments.
Blocking Assignments

module blockassign;
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
initial begin
  x = 0; y = 1; z = 1;
count = 0;
  reg_a = 16'b0; reg_b = reg_a;
#15 reg_a[2] = 1'b1;
#10 reg_b[15:13] = {x, y, z};
count = count + 1;
end
endmodule

Non-Blocking Assignments

module nonblockassign;
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
initial begin
  x = 0; y = 1; z = 1;
count = 0;
  reg_a = 16'b0; reg_b = reg_a;
  reg_a[2] <= #15 1'b1;
  reg_b[15:13] <= #10 {x, y, z};
count <= count + 1;
end
endmodule

It is recommended that blocking and nonblocking assignments not be mixed in the same always block.
Non-Blocking Assignments

- **nonblocking** assignment statements are executed last in the time step
  - after all the blocking assignments in that time step are executed.
- They are used as a method to model **several concurrent data transfers** that take place after a common event.

```verilog
clock __________ __________ __________ __________
always @(posedge clock)
begin
  reg1 <= #1 in1;
  reg2 <= @(negedge clock) in2 ^ in3;
  reg3 <= #1 reg1; //The old value of reg1
end
```

Nonblocking Statements to Eliminate Race Conditions

Two concurrent always blocks

```verilog
always @(posedge clock)
a = b;
always @(posedge clock)
b = a;
```

values of registers a and b will not be swapped. both registers will get **the same value**

```verilog
clock __________ __________ __________ __________
a 3 5 3 5 3 b 5 3 5 3 5
```

```verilog
always @(posedge clock)
a <= b;
always @(posedge clock)
b <= a;
```

registers a and b are **swapped** correctly
Behavior of Non-Blocking

//Emulate the behavior of nonblocking assignments by
//using temporary variables and blocking assignments

always @(posedge clock)
begin
  // Read operation
  // store values of right-hand-side expressions in temporary variables
  temp_a = a;
  temp_b = b;
  // Write operation
  // Assign values of temporary variables to left-hand-side variables
  a = temp_b;
  b = temp_a;
end

Blocking & Non-Blocking

always @(posedge clock)
begin
  RegA = in;
  RegB = RegA;
  RegC = RegB;
end

always @(posedge clock)
begin
  RegA <= in;
  RegB <= RegA;
  RegC <= RegB;
end

always @(posedge clock)
begin
  RegC = RegB;
  RegB = RegA;
  RegA = in;
end

<table>
<thead>
<tr>
<th>case_1</th>
<th>case_2</th>
<th>case_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>regA = 3</td>
<td>regA = 3</td>
<td>regA = 3</td>
</tr>
<tr>
<td>regB = 3</td>
<td>regB = 1</td>
<td>regB = 1</td>
</tr>
<tr>
<td>regC = 3</td>
<td>regC = 7</td>
<td>regC = 7</td>
</tr>
</tbody>
</table>

convenient
Timing Controls

• Various behavioral timing control constructs are available in Verilog.
• In Verilog, if there are no timing control statements, the simulation time does not advance.
• Timing controls provide a way to specify the simulation time at which procedural statements will execute.
• There are three methods of timing control:
  – delay-based timing control,
  – event-based timing control,
  – level-sensitive timing control.

Delay-Based Timing Control

• Delays are specified by the symbol #.
• Delay-based timing control can be specified by
  – a number
  – identifier
  – a mintypmax_expression
• There are three types of delay control for procedural assignments:
  – regular delay control
  – intra-assignment delay control
  – zero delay control

# delay_value  #5 ← number
#xyz ← identifier
#(2:3:4) ← mintypmax_expression
Regular Delay Control

- Regular delay control is used when a non-zero delay is specified to the left of a procedural assignment.

```verilog
//define parameters
parameter latency = 20;
parameter delta = 2;
//define register variables
reg x, y, z, p, q;
initial
begin
    x = 0;       // no delay control
    #10 y = 1;   // delay control with a number. Delay execution of
                  // y = 1 by 10 units
    #latency z = 0; // Delay control with identifier. Delay of 20 units
    #(latency + delta) p = 1; // Delay control with expression
    #y x = x + 1;                      // Delay control with identifier. Take value of y.
    #(4:5:6) q = 0;                   // Minimum, typical and maximum delay values.
    //Discussed in gate-level modeling chapter.
end
```

Intra-assignment Delay Control

- Instead of specifying delay control to the left of the assignment,
- it is possible to assign a delay to the right of the assignment operator.
- Such delay specification alters the flow of activity in a different manner.

```verilog
reg x, y, z;
initial
begin
    x = 0;
    z = 0;
    y = #5 x + z;
end
```

```verilog
reg x, y, z;
reg temp_xz;
initial
begin
    x = 0;
    z = 0;
    temp_xz = x + z;
    #5 y = temp_xz;
end
```
Difference between Regular & Intra-assignment

- Regular delays defer the execution of the entire assignment.
- Intra-assignment delays compute the right-hand-side expression at the current time and defer the assignment of the computed value to the left-hand-side variable.
- Intra-assignment delays are like using regular delays with a temporary variable to store the current value of a right-hand-side expression.

\[
\text{initial}\begin{align*}
\text{begin} & \quad \#5 \ x = a + b; \\
\text{end} &
\end{align*}
\]

\[
\text{initial}\begin{align*}
\text{begin} & \quad y = \#5 \ a + b; \\
\text{end} &
\end{align*}
\]

Event-Based Timing Control

- An event is the change in the value on a register or a net.
- Events can be utilized to trigger execution of a statement or a block of statements.
- There are four types of event-based timing control
  - regular event control
  - named event control
  - event OR control
  - level-sensitive timing control

\[
\begin{align*}
\text{reg} & \quad \text{change} \quad \text{event} \\
\text{wire} & \quad \text{trigger} \\
\end{align*}
\]

\[
\text{begin}\begin{align*}
\text{begin} & \quad \ldots \\
\text{end} &
\end{align*}
\]

\[
\text{begin}\begin{align*}
\text{begin} & \quad \ldots \\
\text{end} &
\end{align*}
\]

\[
\text{begin}\begin{align*}
\text{begin} & \quad \ldots \\
\text{end} &
\end{align*}
\]

\[
\text{begin}\begin{align*}
\text{begin} & \quad \ldots \\
\text{end} &
\end{align*}
\]

\[
\text{begin}\begin{align*}
\text{begin} & \quad \ldots \\
\text{end} &
\end{align*}
\]
Regular Event Control

- The @ symbol is used to specify an event control.
- Statements can be executed on
  - changes in signal value
  - positive transition of signal value: posedge
  - negative transition of signal value: negedge

```verilog
@(clock) q = d;  // change in signal value
@(posedge clock) q = d;  // positive transition
@(negedge clock) q = d;  // negative transition
q = @(posedge clock) d;
```

Named Event Control

- Verilog provides the capability to declare an event and then trigger and recognize the occurrence of that event.
- A named event is declared by the keyword event.
- An event is triggered by the symbol “→”
- The triggering of the event is recognized by the symbol @.

```verilog
event received_data;

always @(posedge clock) begin
  if(last_data_packet) 
    -> received_data;
end

always @(received_data)
  data_buf = {data_pkt[0], data_pkt[1], data_pkt[2], data_pkt[3]};
```
Event-OR Control

- A transition on any one of multiple signals or events can trigger the execution of a statement or a block of statements.
- This is expressed as an OR of events or signals.
- The list of events or signals expressed as an OR is also known as a sensitivity list.
- The keyword “or” is used to specify multiple triggers.

```
always @(reset or clock or d)
begin
  if (reset)
    q = 1'b0;
  else if(clock)
    q = d;
end
```

Event-OR Control

- A transition on any one of multiple signals or events can trigger the execution of a statement or a block of statements.
- This is expressed as an OR of events or signals.
- The list of events or signals expressed as an OR is also known as a sensitivity list.
- The keyword “or” is used to specify multiple triggers.

```
always @(a or b or c or d or e or f or g or h or p or m)
begin
  out1 = a ? b+c : d+e;
  out2 = f ? g+h : p+m;
end
```

If an input variable is missed from the sensitivity list, asynchronous reset is not triggered.

```
always @(reset, clock, d)
begin
  if (reset)
    q = 1'b0;
  else if(clock)
    q = d;
end
```

A positive edge triggered D flip-flop with asynchronous falling reset.

```
always @(posedge clk, negedge reset)
if(!reset)
  q <= 0;
else
  q <= d;
```

If an input variable is missed from the sensitivity list, asynchronous reset is not triggered.
### Synchronous & Asynchronous

#### Synchronous

```verilog
always @(posedge clk)
    if(!reset)
        syn <= 0;
    else
        syn <= d;
```

#### Asynchronous

```verilog
always @(posedge clk, negedge reset)
    if(!reset)
        asyn <= 0;
    else
        asyn <= d;
```

---

### D Flip-Flop

#### Synchronous Reset

```verilog
module DFF(clk, rst_n, d, q);
    input clk, rst_n;
    input d;
    output q;
    always @(posedge clk)
        begin
            if(!rst)
                q <= 1'b0;
            else
                q <= d;
        end
endmodule
```

#### Asynchronous Reset

```verilog
module DFF(clk, rst_n, d, q);
    input clk, rst_n;
    input d;
    output q;
    always @(posedge clk or negedge rst)
        begin
            if(!rst)
                q <= 1'b0;
            else
                q <= d;
        end
endmodule
```
Level-Sensitive Timing Control

- Verilog also allows level-sensitive timing control
  - the ability to `wait` for a certain condition to be true before a statement or a block of statements is executed.
- The keyword “`wait`” is used for level-sensitive constructs.

```verilog
always
  wait (count_enable) #20 count = count + 1;
```

<table>
<thead>
<tr>
<th>count_enable</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Conditional Statement (if-else)

- Conditional statements are used for making decisions based upon certain conditions.
- These conditions are used to decide whether or not a statement should be executed.
- Keywords: `if` and `else`
- There are three types of conditional statements

```
if <expression>
  statement 1;
else
  statement 2;
```
1st Type Conditional Statement

// Type 1 conditional statement. No else statement.
// Statement executes or does not execute.

if (<expression>) true_statement;

// Type 1 statements
if(!lock) buffer = data;
if(enable) out = in;

2nd Type Conditional Statement

// Type 2 conditional statement. One else statement
// Either true_statement or false_statement is evaluated

if (<expression>) true_statement;
else false_statement;

// Type 2 statements
if (number_queued < MAX_Q_DEPTH) begin
  data_queue = data;
  number_queued = number_queued + 1;
end
else
  $display("Queue Full. Try again");
3rd Type Conditional Statement

// Type 3 conditional statement. Nested if-else-if.
// Choice of multiple statements. **Only one is executed.**
if (<expression1>) true_statement1;
else if (<expression2>) true_statement2;
else if (<expression3>) true_statement3;
else default_statement;

Assume: x = 7, z = 2

<table>
<thead>
<tr>
<th>alu_control</th>
<th>y</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x + z</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>x - z</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>x * z</td>
<td>14</td>
</tr>
</tbody>
</table>

Invalid ALU control signal

Multi-way Branching (case)

- In type 3 conditional statement, there were many alternatives, from which one was chosen.
- The nested if-else-if can become unwieldy if there are too many alternatives.
- A shortcut to achieve the same result is to use the **case** statement.
- Keyword: **case**, **endcase**, and **default**

```verilog
case (expression)
  alternative1: statement1;
  alternative2: statement2;
  alternative3: statement3;
  ...
  ...
  default: default_statement;
endcase
```

Rightarrow match

expression

alternative1

alternative2

alternative3

...
Multi-way Branching

```verilog
if (alu_control == 0)    
   y = x + z;
else if(alu_control == 1)     
   y = x - z;
else if(alu_control == 2)    
   y = x * z;
else
   $display("Invalid ALU control signal");
endcase
```

```verilog
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;

always @(s1 or s0 or i0 or i1 or i2 or i3)
  case ({s1, s0})
    2'd0 : out = i0;
    2'd1 : out = i1;
    2'd2 : out = i2;
    2'd3 : out = i3;
     default : $display("Invalid control signals");
    endcase
endmodule
```
Multi-way Branching

module demultiplexer1_to_4 (out0, out1, out2, out3, in, s1, s0);

output out0, out1, out2, out3;
reg out0, out1, out2, out3;
input in;
input s1, s0;

always @(s1 or s0 or in)
  case ({s1, s0}) //Switch based on control signals
    2'b00 : begin out0 = in; out1 = 1'bz; out2 = 1'bz; out3 = 1'bz; end
    2'b01 : begin out0 = 1'bz; out1 = in; out2 = 1'bz; out3 = 1'bz; end
    2'b10 : begin out0 = 1'bz; out1 = 1'bz; out2 = in; out3 = 1'bz; end
    2'b11 : begin out0 = 1'bz; out1 = 1'bz; out2 = 1'bz; out3 = in; end
    //Account for unknown signals on select. If any select signal is x
    //then outputs are x. If any select signal is z, outputs are z.
    //If one is x and the other is z, x gets higher priority.
    2'bx0, 2'bx1, 2'bxz, 2'bxx, 2'b0x, 2'b1x, 2'bzx :
      begin out0 = 1'bx; out1 = 1'bx; out2 = 1'bx; out3 = 1'bx; end
    2'b0z, 2'bz1, 2'bzz, 2'b0z, 2'b1z :
      begin out0 = 1'bz; out1 = 1'bz; out2 = 1'bz; out3 = 1'bz; end
    default: $display("Unspecified control signals");
  endcase
endmodule

if-else & case-endcase

if(condition)
  statement1;
else
  statement2;

case(variable)
  match1: statement1;
  match2: statement2;
  match3: statement3;
  ...
  default: statementX;
endcase
Generate Blocks

- Generate statements allow Verilog code to be generated dynamically at elaboration time before the simulation begins.
- This facilitates the creation of parametrized models.
- Generate statements are particularly convenient
  - when the same operation or module instance is repeated for multiple bits of a vector
  - when certain Verilog code is conditionally included based on parameter definitions.
- Generate statements allow control over the declaration of variables, functions, and tasks, as well as control over instantiations.
- All generate instantiations are coded with a module scope and require the keywords `generate - endgenerate`.

Generate Blocks

- There are three methods to create generate statements:
  - Generate `loop`
  - Generate `conditional`
  - Generate `case`
module bitwise_xor (out, i0, i1);
    parameter N = 32;
    output [N-1:0] out;
    input [N-1:0] i0, i1;

genvar j;  // temporary loop variable

generate
    for (j=0; j<N; j=j+1)
    begin: xor_loop
        xor g1 (out[j], i0[j], i1[j]);
    end
endgenerate
endmodule

module ripple_adder(co, sum, a0, a1, ci);
    parameter N = 4;
    output [N-1:0] sum;
    output co;
    input [N-1:0] a0, a1;
    input ci;
    wire [N-1:0] carry;
    assign carry[0] = ci;

genvar i;

generate
    for (i=0; i<N; i=i+1)
    begin: r_loop
        wire t1, t2, t3;
        xor g1 (t1, a0[i], a1[i]);
        xor g2 (sum[i], t1, carry[i]);
        and g3 (t2, a0[i], a1[i]);
        and g4 (t3, t1, carry[i]);
        or g5 (carry[i+1], t2, t3);
    end
endgenerate
assign co = carry[N];
endmodule
Generate Conditional

```verilog
multiplier #(18,16) mx(Prod, A, B)

module multiplier (product, a0, a1);
    parameter a0_width = 8; // 8-bit bus by default
    parameter a1_width = 8; // 8-bit bus by default
    localparam product_width = a0_width + a1_width;
    output [product_width-1:0] product;
    input [a0_width-1:0] a0;
    input [a1_width-1:0] a1;

generate
    if (a0_width < 8) || (a1_width < 8)
        cla_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
    else
        tree_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
endgenerate
endmodule
```

Generate Case

```verilog
generate an N-bit adder

module adder(co, sum, a0, a1, ci);
    parameter N = 4;
    output [N-1:0] sum;
    output co;
    input [N-1:0] a0, a1;
    input ci;

    // Instantiate the appropriate adder based on the width of the bus.
    generate
        case (N) //Special cases for 1 and 2 bit adders
            1: adder_1bit adder1(c0, sum, a0, a1, ci); //1-bit implementation
            2: adder_2bit adder2(c0, sum, a0, a1, ci); //2-bit implementation
            default: adder_cla #(N) adder3(c0, sum, a0, a1, ci);
        endcase
    endgenerate
endmodule
```
Traffic Signal Controller

To design a traffic signal controller, using a finite state machine approach.

\[ X = 1 \rightarrow \text{car on country road} \]
Traffic Signal Controller – Design

```verilog
module sig_control (hwy, cntry, X, clock, clear);
output [1:0] hwy, cntry;
reg [1:0] hwy, cntry;
input X;
input clock, clear;

parameter RED = 2'd0,
YELLOW = 2'd1,
GREEN = 2'd2;

parameter S0 = 3'd0,   // GREEN      RED
S1 = 3'd1,   // YELLOW   RED
S2 = 3'd2,   // RED           RED
S3 = 3'd3,   // RED           GREEN
S4 = 3'd4;   // RED           YELLOW

delay = 3

delay = 2

reg [2:0] state;
reg [2:0] next_state;

always @(posedge clock)
if (clear) state <= S0;
else state <= next_state;

always @(state)
begin
 hwy = GREEN;
cntry = RED;
end

always @(state)
begin
 hwy = YELLOW;
end

always @(state)
begin
 hwy = RED;
cntry = GREEN;
end

always @(state)
begin
 hwy = RED;
cntry = YELLOW;
end

always @(state or X)
begin
case (state)
S0: if(X) next_state = S1;
   else next_state = S0;
S1: begin
    repeat(`Y2RDELAY) @(posedge clock) ;
    next_state = S2;
end
S2: begin
    repeat(`R2GDELAY) @(posedge clock);
    next_state = S3;
end
S3: if(X) next_state = S3;
   else next_state = S4;
S4: begin
    repeat(`Y2RDELAY) @(posedge clock) ;
    next_state = S0;
end
default: next_state = S0;
endcase
end

delay = 3

delay = 2

Traffic Signal Controller – Design
```

YANG S.W.
yangsw@thu.edu.tw

© Verilog HDL
Design & Simulation

Traffic Signal Controller – Design

always @((posedge clock))
if (clear) state <= S0;
else state <= next_state;

always @((state))
begin
 hwy = GREEN;
cntry = RED;
end

always @((state))
begin
 hwy = YELLOW;
end

always @((state))
begin
 hwy = RED;
cntry = GREEN;
end

always @((state))
begin
 hwy = RED;
cntry = YELLOW;
end

always @((state or X))
begin
case (state)
S0: if(X) next_state = S1;
   else next_state = S0;
S1: begin
    repeat(`Y2RDELAY) @(posedge clock) ;
    next_state = S2;
end
S2: begin
    repeat(`R2GDELAY) @(posedge clock);
    next_state = S3;
end
S3: if(X) next_state = S3;
   else next_state = S4;
S4: begin
    repeat(`Y2RDELAY) @(posedge clock) ;
    next_state = S0;
end
default: next_state = S0;
endcase
end
```

Traffic Signal Controller – Design

always @((posedge clock))
if (clear) state <= S0;
else state <= next_state;

always @((state))
begin
 hwy = GREEN;
cntry = RED;
end

always @((state))
begin
 hwy = YELLOW;
end

always @((state))
begin
 hwy = RED;
cntry = GREEN;
end

always @((state))
begin
 hwy = RED;
cntry = YELLOW;
end

always @((state or X))
begin
case (state)
S0: if(X) next_state = S1;
   else next_state = S0;
S1: begin
    repeat(`Y2RDELAY) @(posedge clock) ;
    next_state = S2;
end
S2: begin
    repeat(`R2GDELAY) @(posedge clock);
    next_state = S3;
end
S3: if(X) next_state = S3;
   else next_state = S4;
S4: begin
    repeat(`Y2RDELAY) @(posedge clock) ;
    next_state = S0;
end
default: next_state = S0;
endcase
end
```
Traffic Signal Controller – Stimulus

module stimulus;
wire [1:0] MAIN_SIG, CNTRY_SIG;
reg CAR_ON_CNTRY_RD;
reg CLOCK, CLEAR;
sig_control SC(MAIN_SIG, CNTRY_SIG,
CAR_ON_CNTRY_RD,
CLOCK, CLEAR);

initial
$monitor($time, " Main Sig = %b Country Sig = %b Car_on_cntry = %b",
MAIN_SIG, CNTRY_SIG, CAR_ON_CNTRY_RD);

initial
begin
CLOCK = `FALSE;
forever #5 CLOCK = ~CLOCK;
end

initial
begin
CLEAR = `TRUE;
repeat (5) @(negedge CLOCK);
CLEAR = `FALSE;
end

sig_control (hwy, cntry, X, clock, clear);

`define TRUE 1'b1
`define FALSE 1'b0

Traffic Signal Controller – Stimulus

initial
begin
CAR_ON_CNTRY_RD = `FALSE;
repeat(20) @(negedge CLOCK);
CAR_ON_CNTRY_RD = `TRUE;
repeat(10) @(negedge CLOCK);
CAR_ON_CNTRY_RD = `FALSE;
repeat(20) @(negedge CLOCK);
CAR_ON_CNTRY_RD = `TRUE;
repeat(10) @(negedge CLOCK);
CAR_ON_CNTRY_RD = `FALSE;
repeat(20) @(negedge CLOCK);
CAR_ON_CNTRY_RD = `TRUE;
repeat(10) @(negedge CLOCK);
CAR_ON_CNTRY_RD = `FALSE;
repeat(10) @(negedge CLOCK);
$stop;
end
endmodule
Simulation Result

0 Main Sig = x Country Sig = x Car_on_cntry = 0
5 Main Sig = 2 Country Sig = 0 Car_on_cntry = 0
200 Main Sig = 2 Country Sig = 0 Car_on_cntry = 1
205 Main Sig = 1 Country Sig = 0 Car_on_cntry = 1
245 Main Sig = 0 Country Sig = 0 Car_on_cntry = 1
275 Main Sig = 0 Country Sig = 2 Car_on_cntry = 1
300 Main Sig = 0 Country Sig = 2 Car_on_cntry = 0
305 Main Sig = 0 Country Sig = 1 Car_on_cntry = 0
345 Main Sig = 2 Country Sig = 0 Car_on_cntry = 0
500 Main Sig = 2 Country Sig = 0 Car_on_cntry = 1
505 Main Sig = 1 Country Sig = 0 Car_on_cntry = 1
545 Main Sig = 0 Country Sig = 0 Car_on_cntry = 1
575 Main Sig = 0 Country Sig = 2 Car_on_cntry = 1
600 Main Sig = 0 Country Sig = 2 Car_on_cntry = 0
605 Main Sig = 0 Country Sig = 1 Car_on_cntry = 0
645 Main Sig = 2 Country Sig = 0 Car_on_cntry = 0
800 Main Sig = 2 Country Sig = 0 Car_on_cntry = 1
805 Main Sig = 1 Country Sig = 0 Car_on_cntry = 1
845 Main Sig = 0 Country Sig = 0 Car_on_cntry = 1
875 Main Sig = 0 Country Sig = 2 Car_on_cntry = 1
900 Main Sig = 0 Country Sig = 2 Car_on_cntry = 0
905 Main Sig = 0 Country Sig = 1 Car_on_cntry = 0
945 Main Sig = 2 Country Sig = 0 Car_on_cntry = 0

Part 2

Timing and Delays
Types of Delay Models

- **Distributed Delay**
  - Delay values are assigned to individual elements in the circuit.

- **Lumped Delay**
  - Specified as a single delay on the output gate of the module.
  - The cumulative delay of all paths is lumped at one location.

- **Pin-to-Pin Delays**
  - Delays are assigned individually to paths from each input to each output.
  - Delays can be separately specified for each input/output path.

```verilog
module M (out, a, b, c, d);
  output out;
  input a, b, c, d;
  wire e, f;
  and #5 a1(e, a, b);
  and #7 a2(f, c, d);
  and #4 a3(out, e, f);
endmodule
```

```verilog
module M (out, a, b, c, d);
  output out;
  input a, b, c, d;
  wire e, f;
  assign #5 e = a & b;
  assign #7 f = c & d;
  assign #4 out = e & f;
endmodule
```
Lumped Delay

- Computed the **maximum delay** from any input to the output
  - $5 + 4 = 9$ units
  - $7 + 4 = 11$ units
- The entire delay is lumped into the output gate.

```verilog
module M (out, a, b, c, d);
output out;
input a, b, c, d;
wire e, f;
and a1(e, a, b);
and a2(f, c, d);
and #11 a3(out, e, f);
endmodule
```

Pin-to-Pin Delays

- $5 + 4 = 9$ units
- $7 + 4 = 11$ units

```verilog
path, a-e-out, delay = 9
path, b-e-out, delay = 9
path, c-f-out, delay = 11
path, d-f-out, delay = 11
```

**path delay model**
Path Delay Modeling

- A delay between a source (input or inout) pin and a destination (output or inout) pin of a module is called a module path delay.
- Path delays are assigned in Verilog within the keywords specify and endspecify.
- The statements within these keywords constitute a specify block.
  - Assign pin-to-pin timing delays across module paths
  - Set up timing checks in the circuits
  - Define specparam constants

Pin-to-Pin Delays

module M (out, a, b, c, d);
output out;
input a, b, c, d; wire e, f;
specify
  (a => out) = 9;
  (b => out) = 9;
  (c => out) = 11;
  (d => out) = 11;
endspecify

and a1(e, a, b);
and a2(f, c, d);
and a3(out, e, f);
endmodule

The specify block is a separate block in the module
Inside Specify Blocks

☆ Parallel connection
A parallel connection is specified by the symbol =>
Usage:
( <source_field> => <destination_field>) = <delay_value>;

```
(a[0] => out[0]) = 9;
(a[1] => out[1]) = 9;
(a[2] => out[2]) = 9;
(a[3] => out[3]) = 9;
```

<table>
<thead>
<tr>
<th>BW(a)</th>
<th>BW(out)</th>
<th>description</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>(a =&gt; out) = 9;</td>
<td>legal</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>(a =&gt; out) = 9;</td>
<td>legal</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>(a =&gt; out) = 9;</td>
<td>illegal</td>
</tr>
</tbody>
</table>

bit width does not match.

Inside Specify Blocks

☆ Full connection
A full connection is specified by the symbol *>
Usage:
( <source_field> *> <destination_field>) = <delay_value>;

```
module M (out, a, b, c, d);
output out;
input a, b, c, d;
wire e, f;
specify
  (a, b *> out) = 9;
  (c, d *> out) = 11;
endspecify
  and a1(e, a, b);
  and a2(f, c, d);
  and a3(out, e, f);
endmodule
```

<table>
<thead>
<tr>
<th>BW(a)</th>
<th>BW(out)</th>
<th>description</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>16</td>
<td>(a *&gt; out) = 9;</td>
<td>legal</td>
</tr>
</tbody>
</table>
Inside Specify Blocks

//Specify parameters using specparam statement
specify
//define parameters inside the specify block
   specparam d_to_q = 9;
   specparam clk_to_q = 11;
   (d => q) = d_to_q;
   (clk => q) = clk_to_q;
endspecify

Specify parameters are provided for convenience in assigning delays.

It is recommended that all pin-to-pin delay values be expressed in terms of specify parameters instead of hardcoded numbers.

If timing specifications of the circuit change, the user has to change only the values of specify parameters.

specparam statements

• Special parameters can be declared for use inside a specify block. They are declared by the keyword specparam.
• Instead of using hardcoded delay numbers to specify pin-to-pin delays, it is common to define specify parameters by using specparam and then to use those parameters inside the specify block.
• The specparam values are often used to store values for nonsimulation tools, such as delay calculators, synthesis tools, and layout estimators.
Timing Checks

- three most common timing checks tasks:
  - $setup$: check the setup time constraint
  - $hold$: check the hold time constraint
  - $width$: check the width of a pulse
- timing checks must be inside the `specify` blocks only
- optional notifier arguments used in these timing check system tasks are omitted to simplify the discussion

setup time is the minimum time the data must arrive before the active clock edge.

hold time is the minimum time the data cannot change after the active clock edge.

the width of a pulse meets the minimum width requirement
$setup task

Setup checks can be specified with the system task $setup.

$setup(data_event, reference_event, limit);

<table>
<thead>
<tr>
<th>data_event</th>
<th>Signal that is monitored for violations</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference_event</td>
<td>Signal that establishes a reference for monitoring the data_event signal</td>
</tr>
<tr>
<td>limit</td>
<td>Minimum time required for setup of data event</td>
</tr>
</tbody>
</table>

Violation is reported if $T_{reference\_event} - T_{data\_event} < limit$

```verilog
// Setup check is set.
// clock is the reference
// data is being checked for violations
// Violation reported if Tposedge_clk - Tdata < 3
specify
$setup(data, posedge clock, 3);
endspecify
```

$hold task

Hold checks can be specified with the system task $hold.

$hold(reference_event, data_event, limit);

<table>
<thead>
<tr>
<th>reference_event</th>
<th>Signal that establishes a reference for monitoring the data_event signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_event</td>
<td>Signal that is monitored for violations</td>
</tr>
<tr>
<td>limit</td>
<td>Minimum time required for setup of data event</td>
</tr>
</tbody>
</table>

Violation is reported if $T_{data\_event} - T_{reference\_event} < limit$

```verilog
// Hold check is set.
// clock is the reference
// data is being checked for violations
// Violation reported if Tdata - Tposedge_clk < 5
specify
$hold(posedge clock, data, 5);
endspecify
```
$width Check

system task $width is used to check that the width of a pulse meets the minimum width requirement

<table>
<thead>
<tr>
<th>$width(reference_event, limit);</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference_event</td>
</tr>
<tr>
<td>limit</td>
</tr>
<tr>
<td>data_event</td>
</tr>
</tbody>
</table>

Violation is reported if $(T_{data\_event} - T_{reference\_event}) < limit$

```verilog
// width check is set. 
// posedge of clear is the reference_event 
// the next negedge of clear is the data_event 
// Violation reported if Tdata - Tclk < 6 
specify
    $width(posedge clear, 6);
endspecify
```

Delay Back-Annotation

The concept of back-annotation of delays in a simulation.

Using delay calculator and information about the IC fabrication process

Computing from the R&C information which is extracted from factors such as geometry and IC fabrication process in the layout.

A standard format called the Standard Delay Format (SDF) is popularly used for back-annotation.
Inside Specify Blocks

☆ Edge-Sensitive Paths

An edge-sensitive path construct is used to model the timing of input to output delays, which occurs only when a specified edge occurs at the source signal.

// In this example, at the positive edge of clock, a module path extends from clock signal to out signal using a rise delay of 10 and a fall delay of 8. The data path is from in to out, and the in signal is not inverted as it propagates to the out signal.

(posedge clock => (out +: in)) = (10 : 8);

Inside Specify Blocks

☆ Conditional path delays

- the pin-to-pin delays might change, based on the states of input signals to a circuit.
- path delays to be assigned conditionally, based on the value of the signals in the circuit
Conditional Path Delay Example

//Conditional Path Delays
module M (out, a, b, c, d);
output out;
input a, b, c, d;
wire e, f;
//specify block with conditional pin-to-pin timing
specify
//different pin-to-pin timing based on state of signal a.
if (a) (a => out) = 9;
if (~a) (a => out) = 10;
//Conditional expression contains two signals b, c.
//If b & c is true, delay = 9,
//Conditional Path Delays
if (b & c) (b => out) = 9;
if (~(b & c)) (b => out) = 13;
endspecify
and a1(e, a, b);
and a2(f, c, d);
and a3(out, e, f);
endmodule
Part 3

Logic Synthesis with Verilog HDL

What Is Logic Synthesis?

- **Logic synthesis** is the process of converting a high-level description of the design into an optimized gate-level representation
  - a standard cell library
  - design constraints.
- A standard cell library can have
  - simple cells, such as basic logic gates like and, or, and nor
  - macro cells, such as adders, muxes, and special flip-flops.
Logic Synthesis inside the designer's mind

Computer-Aided logic synthesis tools

architectural trade-offs
high-level description
design constraints

- designers describe the high-level design in terms of HDLs.
- high-level descriptions become more popular

Reduced conversion time
Verilog HDL Synthesis

- For the purpose of logic synthesis, designs are currently written in an HDL at a register transfer level (RTL).

- RTL description style combines
  - data flow
  - behavioral

- Logic synthesis tools
  - take the register transfer-level HDL description
  - convert it to an optimized gate-level netlist.

- the two most popular HDLs are used at the RTL level
  - Verilog
  - VHDL

Verilog Constructs

Not all constructs can be used when writing a description for a logic synthesis tool.

any construct that is used to define a cycle-by-cycle RTL description is acceptable to the logic synthesis tool.

- the while and forever loops must be “broken” by a @ (posedge clock) or @ (negedge clock) statement to enforce cycle-by-cycle behavior and to prevent combinational feedback.

- logic synthesis ignores all timing delays specified by #<delay> construct. Therefore, pre- and post-synthesis Verilog simulation results may not match. The designer must use a description style that eliminates these mismatches.

- Also, the initial construct is not supported by logic synthesis tools. Instead, the designer must use a reset mechanism to initialize the signals in the circuit.

- all signal widths and variable widths be explicitly specified.
### Verilog Constructs

<table>
<thead>
<tr>
<th>Construct Type</th>
<th>Keyword or Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ports</td>
<td>input, inout, output</td>
<td></td>
</tr>
<tr>
<td>parameters</td>
<td>parameter</td>
<td></td>
</tr>
<tr>
<td>module definition</td>
<td>module</td>
<td></td>
</tr>
<tr>
<td>signals and variables</td>
<td>wire, reg, tri</td>
<td>Vectors are allowed</td>
</tr>
<tr>
<td>instantiation</td>
<td>module instances, primitive gate instances</td>
<td>E.g., mymux m1(out, i0, i1, s); E.g., nand (out, a, b);</td>
</tr>
<tr>
<td>functions and tasks</td>
<td>function, task</td>
<td>Timing constructs ignored</td>
</tr>
<tr>
<td>procedural</td>
<td>always, if, then, case, casez</td>
<td>initial is not supported</td>
</tr>
<tr>
<td>procedural blocks</td>
<td>begin, end, named blocks, disable</td>
<td>Disabling of named blocks allowed</td>
</tr>
<tr>
<td>data flow</td>
<td>assign</td>
<td>Delay information is ignored</td>
</tr>
<tr>
<td>loops</td>
<td>for, while, forever,</td>
<td>while and forever loops must contain @(posedge clk) or @(negedge clk)</td>
</tr>
</tbody>
</table>

### Verilog Operators

- Almost all operators in Verilog are allowed for logic synthesis.
- `===` and `!==` that are related to `x` and `z` are not allowed, because equality with `x` and `z` does not have much meaning in logic synthesis.
- It is recommended that you use parentheses to group logic the way you want it to appear.
## Verilog Operators

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator Symbol</th>
<th>Operation Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>*, /</td>
<td>multiply, divide</td>
</tr>
<tr>
<td></td>
<td>+, -</td>
<td>add, subtract</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>modulus</td>
</tr>
<tr>
<td></td>
<td>+, -</td>
<td>unary plus &amp; minus</td>
</tr>
<tr>
<td>Logical</td>
<td>!</td>
<td>logical negation</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp;,</td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td>&gt;, &lt;</td>
<td>greater than, less than</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>Equality</td>
<td>==</td>
<td>equality</td>
</tr>
<tr>
<td></td>
<td>!=</td>
<td>inequality</td>
</tr>
<tr>
<td>Bit-wise</td>
<td>~</td>
<td>bitwise negation</td>
</tr>
<tr>
<td></td>
<td>&amp;,</td>
<td>, ^</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator Symbol</th>
<th>Operation Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduction</td>
<td>&amp;</td>
<td>reduction and</td>
</tr>
<tr>
<td></td>
<td>~&amp;</td>
<td>reduction nand</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>~</td>
<td></td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>reduction ex-or</td>
</tr>
<tr>
<td></td>
<td>^~ or ^</td>
<td>reduction ex-nor</td>
</tr>
<tr>
<td>Shift</td>
<td>&gt;&gt;</td>
<td>right shift</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;</td>
<td>left shift</td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;&gt;</td>
<td>arithmetic right shift</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;&lt;</td>
<td>arithmetic left shift</td>
</tr>
<tr>
<td>Concatenation</td>
<td>{ }</td>
<td>concatenation</td>
</tr>
<tr>
<td>Conditional</td>
<td>?:</td>
<td>conditional</td>
</tr>
</tbody>
</table>
Synthesis Design Flow

1. **RTL Description**
   - designer spends time in functional verification
   - a simple allocation of internal resources.
   - design constraints are not considered

2. **Translation**
3. **Unoptimized Intermediate Representation**
4. **Logic Optimization**
5. **Technology Mapping and Optimization**
6. **Optimized Gate-Level Representation**

- removal of redundant logic
- boolean logic optimization
- area, timing, and power

Example of RTL-to-Gates

```verilog
module magnitude_comparator(A_gt_B, A_lt_B, A_eq_B, A, B);
output A_gt_B, A_lt_B, A_eq_B;
input [3:0] A, B;
assign A_gt_B = (A > B);
assign A_lt_B = (A < B);
assign A_eq_B = (A == B);
endmodule
```

Library cells for abc_100 technology:
- VNAND // 2-input nand gate
- VAND // 2-input and gate
- VNOR // 2-input nor gate
- VOR // 2-input or gate
- VNOT // not gate
- VBUF // buffer
- NDFF // negedge DFF
- PDFF // posedge DFF
Example of RTL-to-Gates

```
module VAND (out, in0, in1);
input in0;
input in1;
output out;
specify
  (in0 => out) = (0.260604:0.513000:0.955206, 0.255524:0.503000:0.936586);
  (in1 => out) = (0.260604:0.513000:0.955206, 0.255524:0.503000:0.936586);
endspecify
and (out, in0, in1);
endmodule
```

//All library cells will have corresponding module definitions
//in terms of Verilog primitives.

---

Example of Gate-level Simulation

```
module magnitude_comparator ( A_gt_B, A_lt_B, A_eq_B, A, B );
input [3:0] A;
input [3:0] B;
output A_gt_B, A_lt_B, A_eq_B;
wire n60, n61, n62, n50, n63, n51, n64, n52, n65,
n40, n53, n41, n54, n42, n55, n43, ..., n39;
VAND U7 ( .in0(n48), .in1(n49), .out(n38) );
VAND U8 ( .in0(n51), .in1(n52), .out(n50) );
VAND U9 ( .in0(n54), .in1(n55), .out(n53) );
VNOT U30 ( .in(A ...
```

---

```
// Example of Gate-level Simulation

Verilog Simulator

$excellence("mag_cmp.sdf", top);

```

---

**Verilog HDL Design & Simulation**

---

```
// Example of Gate-level Simulation

Verilog Simulator

```

---

```