CT Verilog Series

Verilog Overview Part 3
Behavioral Modeling

黃稚存

Chih-Tsun Huang

cthuang@cs.nthu.edu.tw

國立清華大學
資訊工程學系

05

聲明

- 本課程之內容 (包括但不限於教材、影片、圖片、檔案資料等)，僅供修課學生個人合理使用，非經授課教師同意，不得以任何形式轉載、重製、散布、公開播送、出版或發行本影片內容 (例如將課程內容放置公開平台上，如 Facebook, Instagram, YouTube, Twitter, Google Drive, Dropbox 等等)。如有侵權行為，需自負法律責任。

Outline

- Behavioral Modeling
- Procedural Timing Controls
- Review of Basic Module Structure
- Test Stimulus
- Tasks and Functions

- Part 2 Recap
  - Numbers
  - Data Types
  - Operators
  - Compiler Directives
  - Dataflow Modeling

Behavioral Modeling

Behavioral Process (1/2)

- Two constructs: `always` and `initial` blocks
  - **`initial`**: one-shot activity flow
    - Starts at time 0, executes exactly once
    - Usually for defining testbenches
    - (Almost) not synthesizable
  - **`always`**: repetitive activity flow
    - Starts at time 0, executes continuously in a repeated fashion once any event in the sensitivity list is triggered
    - Can describe either combinational or sequential design

- Here, both one-shot and repetitive are how simulator behaves

Behavioral Process (2/2)

- Body of `initial`/`always` may consist of a single statement or a block statement
  - Block statement begins with **`begin`** and ends with **`end`**
- Cannot be nested
- Use high-level procedural statements inside `initial`/`always` to assign values to register variables
  - `if-else`, `case`, `for`, `repeat`, `while`, etc.

Procedural Assignment

- Procedural assignment can assign a value to reg-type, integer-type, or float-type variables
- **Blocking** procedural assignment: **=**
  - An assignment is completed before the next assignment starts

(assume a = 0)

```
a = 1;
c = a; // c = 1
```

a=Ø a=1 c=a=1

t_1

Simulated time

$\Delta t = Ø$

- **Non-blocking** procedural assignment: **<=**
  - Assignments are executed in parallel

(assume a = 0)

```
a <= 1;
c <= a; // c = 0
```

a<=1 tmp1=1 a=tmp1=1

a=Ø c<=a tmpa=a. c=tmpa=Ø

t_1

Simulated time

$\Delta t = Ø$

Blocking vs. Non-Blocking

(Initially, let `a = 1; b = 0;`)

Blocking

```
...
a = b; =0
b = a; =0
```

a = ? 0
b = ? 0

```
...
b = a; =1
a = b; =1
```

a = ? 1
b = ? 1

Non-Blocking

```
...
a <= b;
b <= a;
```

a = ? 0
b = ? 1

```
...
b <= a;
a <= b;
```

a = ? 0
b = ? 1

initial

- Execute once (and only once)

```
...
initial                 // An “initial” behavior
begin
    sig_a = 0;          // Procedural assignments
    sig_b = 1;          // execute sequentially.
    sig_c = 0;
end
...
```

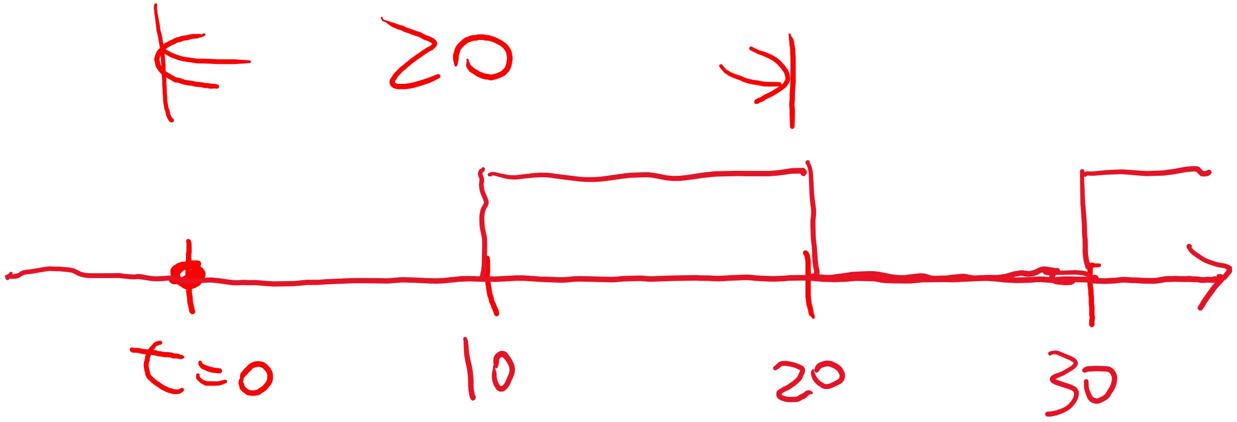
always

Sensitivity list

- Infinite loop until simulation stops

@(a, b, ci)

```
...
always @*
  {co, s} = a + b + ci;

initial
  clock = 0;

always
  #10 clock = ~clock;

initial
  #100 $finish;
```



Block Scope

- module/endmodule, case/endcase, always, initial, if-else, etc.
- One-line block

```
if (b == 4'b1001)  a = 0;
else  a = c + 1;
```

- Multiple-line block with begin … end

```
if (b == 4'b1001) begin
  a = 0;
  ready = 1;
end else begin
  a = c + 1;
  ready = 0;
end
```

Activity Flow Control inside `always/initial` Block Scope

- Conditional operator (`?…:`)
- `case, casex, casez`
- `if … else`
- Loops:
  `for, while, repeat, forever`

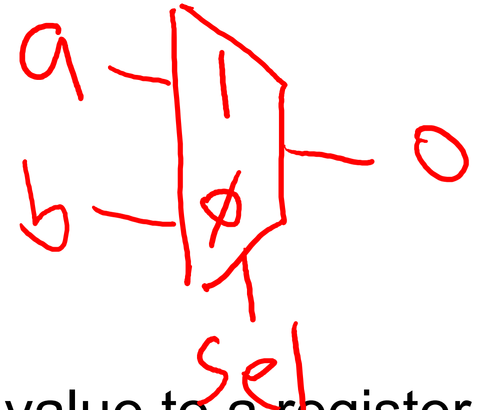
Only inside always/initial block scope

Conditional Operator ? … : …

- C-like conditional operator
- Can be in continuous assignment

```
module MUX (o, a, b, sel);
  output o;
  input  a, b, sel;
  wire   o;
  assign o = sel ? a : b;  // multiplexer
endmodule
```

2-to-1 MUX



- Can be also used in a procedural statement to assign value to a register variable

```
  reg o;
  always @*
    o = sel ? a : b;
```

Conditional Statement: `if-else`

- Procedural assignment
- Numerical value of 0, or value x is logic false, while 1 or a non-zero value is evaluated to true
- **else** is paired with the nearest **if**

2-to-1 MUX

```
reg o;
always @* begin
  if (sel == 1)  o = a;
  else  o = b;
end
```

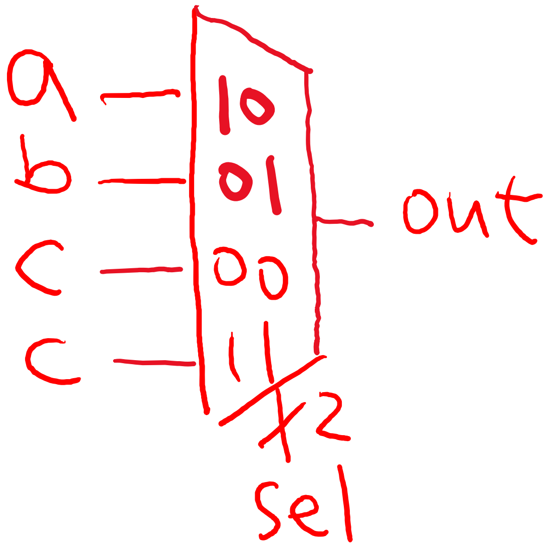
```
(a) if (a < b) c = d +1;
(b) if (a < b);
(c) if (k == 1)
       begin : A_block
         sum_out = sum_reg;
         c_out = c_reg;
       end
```

```
(d) if (a < b)
      sum = sum + 1;
    else
      sum = sum + 2;
(e) if (a == 1) sig_out = reg_a;
    else if (a == 2) sig_out = reg_b;
    else sig_out = reg_c;
```

Conditional Statement: case (1/2)

- Require complete bitwise match, so expression and case item must have the same bit length
- 2-to-1 multiplexer (MUX)

```
wire sel;
reg [7:0] out;
always @* begin
  case (sel)
    1'b0: out = b;
    1'b1: out = a;
  endcase
end
```



- case with default

```
wire [1:0] sel;
reg [7:0] out;
always @* begin
  case (sel) 1
    2'b01: out = b;
    2'b10: out = a;
    default: out = c;
  endcase
end
```

```
wire [1:0] sel;
reg [7:0] out;
always @* begin
  out = c;
  case (sel) 10
    2'b01: out = b;
    2'b10: out = a;
  endcase
end
```

Conditional Statement: case (2/2)

1000
0100
0010
0001

- 4-to-1 multiplexer (MUX)

```
reg [1:0] select;
always @* begin
  case (select)
    0: y = a;
    1: y = b;
    2: y = c;
    3: y = d;
    default: y = 0; // redundant
  endcase
end
```

- Another case usage

```
reg [3:0] onehot;
always @* begin
  case (1'b1)
    onehot[3]: y = a;
    onehot[2]: y = b;
    onehot[1]: y = c;
    onehot[0]: y = d;
    default: y = 0;
  endcase
end
```

priority encoder

- What if not one-hot?
  - The assignments have a priority

Other Conditional Statements: **casex** and **casez**

- **casex** treats bit positions that have **x** or **z** as don’t care
- **casez** treats bit positions that have **z** as don’t care

for Loop

- Example:

```
integer count;
initial
  for (count = 0; count < 128; count = count + 1)
    $display("Count = %d", count);
```

~~Count~~ ++

- Example:

```
parameter datawidth = 32;
reg [datawidth - 1:0] state;
integer i;
initial begin
  for (i = 0; i < datawidth; i = i + 2)
    state[i] = 0;
end
```

repeat Loops

```
count = 0;
repeat (128) begin
  $display("Count = %d", count);
  count = count + 1;
end
```

```
word_address = 0;
repeat (memory_size) begin
  memory[word_address] = 0;
  word_address = word_address + 1;
end
```

while Loops

```
count = 0;
while (count < 128)  begin
  $display("Count = %d", count);
  count = count + 1;
end
```

```
  reg [7:0] temp;
  counter = 0;
  temp = a;

  while (temp) begin
    if (temp[0])
      counter = counter + 1;
// if statement can be replaced by
// counter = counter + temp[0];
    temp = temp >> 1;
  end
```

forever Loops

```
initial
  begin: clock_loop
    clock = 0;
    forever
      #10 clock = ~clock;
  end
```

Procedural Timing Controls

Procedural Timing Controls

- Mechanisms
  - Delay control operator (**`#`**)
  - Event control operator (**`@`**)
  - Event **`or`**
  - (*we don't discuss it here) **`wait`** construct
- Delay control is not synthesizable!
  - Synthesis tool will ignore the delay control

```
data <= #10 a * b;
```

Delay Control Operator:

- Suspend the activity flow at the location of the operator
- Not synthesizable!!
- Example

```
…
always begin
   #0 clock = 0;
   #50 clock = 1;
   #50;
end
…
```

```
parameter clock_period = 100;
reg clock;
initial clock = 0;
always begin
   #(clock_period/2);
   clock = ~clock;
end
```

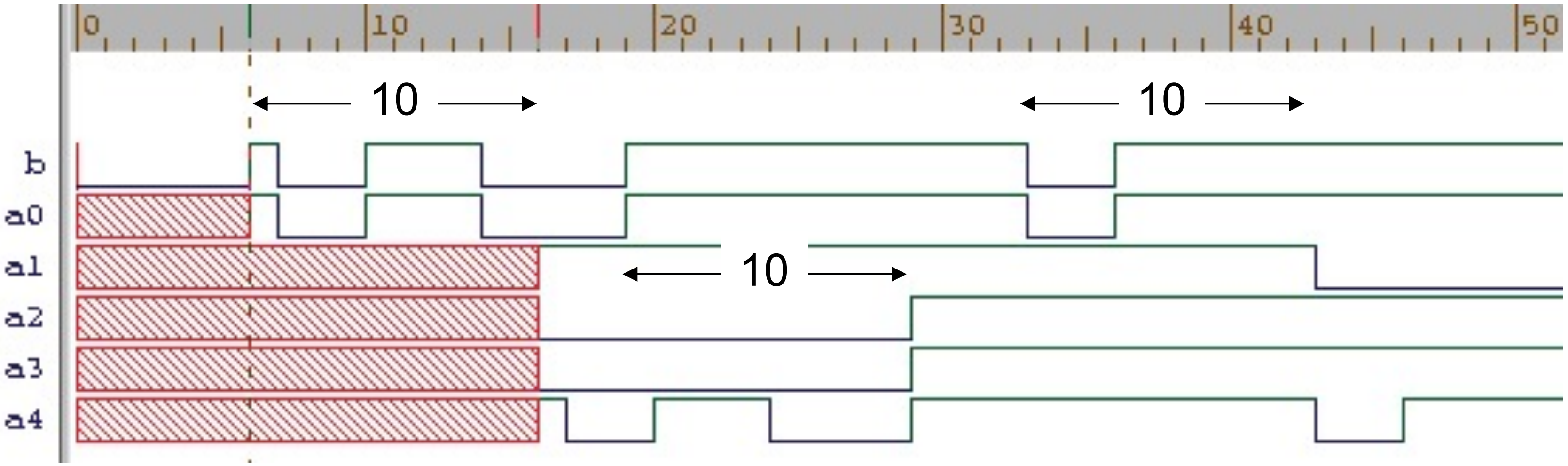
(Optional) Different Delay Controls

(Hint: inertial delay)

- Example

```
reg b, a0, a1, a2, a3, a4;
always @(b) begin
   a0 = b;
end
```

```
a1 = #10 b;
#10 a2 = b;
#10 a3 <= b;
a4 <= #10 b;
```



Event Control Operator: @

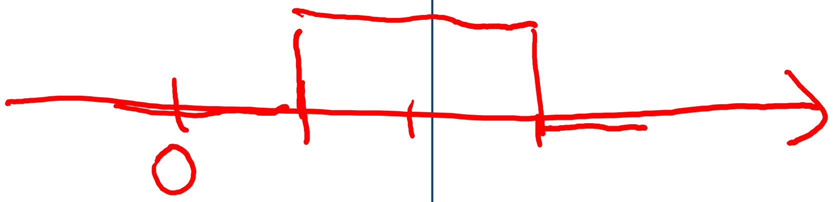
- Synchronize execution to an event

```
...
@(event_a) begin
   ...
   @(event_b) begin
   ...
   end
end
...
```

- Event:
  - Level-triggered (level-sensitive): Value change of a variable
  - Edge-triggered: when a specific positive transition of a signal or negative transition happens

Delay and Event Control

```
parameter cycle = 10;
...
initial #(cycle*10000) $finish;
initial begin
  #0 reset = 0;
  #(cycle) reset = 1;
  #(cycle*2) reset = 0;

  @(enable or done);
  ...
```



Level-triggered event

posedge (positive edge-triggered) and **negedge** (negative edge-triggered)

- **posedge**
  - √ 0→1, x or z
  - √ x→1
  - √ z→1

Positive Transition
Positive Edge
Rising Edge

- **negedge**
  - 1→0, x or z
  - x→0
  - z→0

Negative Transition
Negative Edge
Falling Edge

- Example:

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

Signal Transition with Unknown X

	X ⟶ 0	X ⟶ 1
`always @(a)`	√	√
`always @(posedge a)`	X	√

Event **or**

- Allow a complex event expression to be formed as the disjunction (logical “or”) of other variables
- DO NOT mix level-triggered event with edge-triggered event in the same sensitivity list!
- Example:

```
...
always @(a or b or ci or en) begin
   if (en == 1'b1)
      {co, s} = a + b + ci;
   else
   ...
```

```
always @(a, b, ci, en) begin
```

Sensitivity List

Review of Basic Module Structure

Basic Module Structure

ANSI-C

```
module module_name ( ports );
  output [3:0] ports ;
  input ports ;          (Signals)
  reg reg_type_variables ;
  wire wire_type_variables ;
  reference instance_name ( pins );
  assign wire_type_variable = expression ;
  always @( sensitivity_list ) begin
    ...
    reg_type_variable = expression ;
  end
endmodule
```

Data Type: reg, wire

initial ↔ always

```
module module_name (
  reg output [3:0] port,
  reg output port,
  wire input port,
  wire input port
);
```

```
if
for
while
case
...
```

High-level Behavioral Statements

Test Stimulus

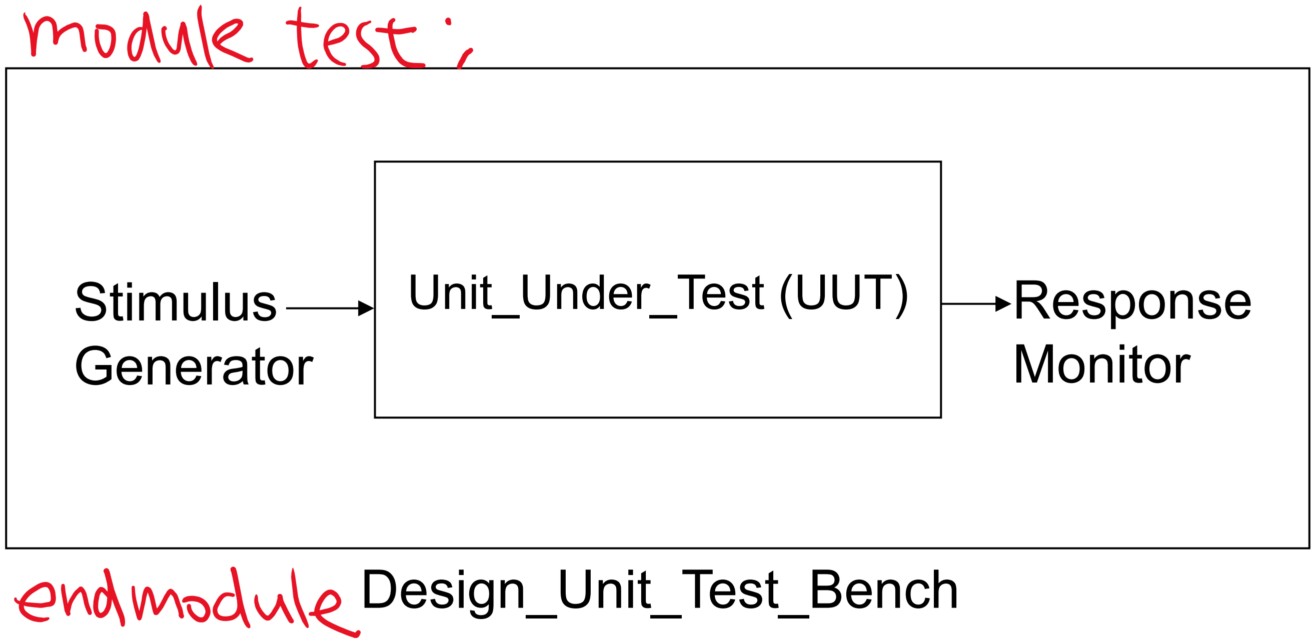
» Or testbench

Event-Driven Simulation

- Verilog variable/signal values
  - `{0, 1, x, z}`
  - x: unknown, ambiguous
  - z: high impedance, open circuit
- An event occurs when a signal changes in value
- Simulation is event-driven

Testbench (Test Stimulus)

- Use Verilog module to produce testing environment including stimulus generation and response monitoring

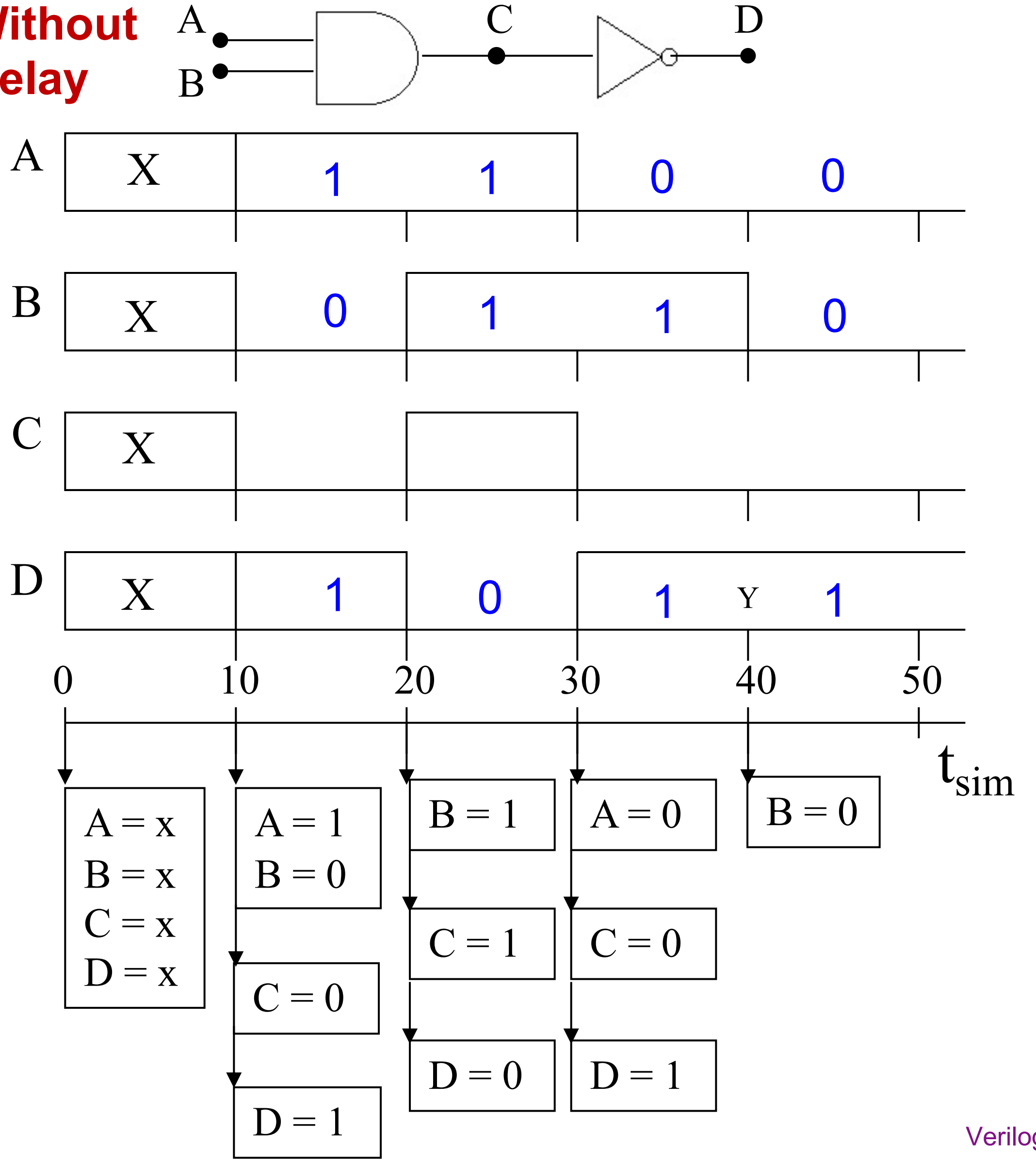


`initial` and Common System Tasks

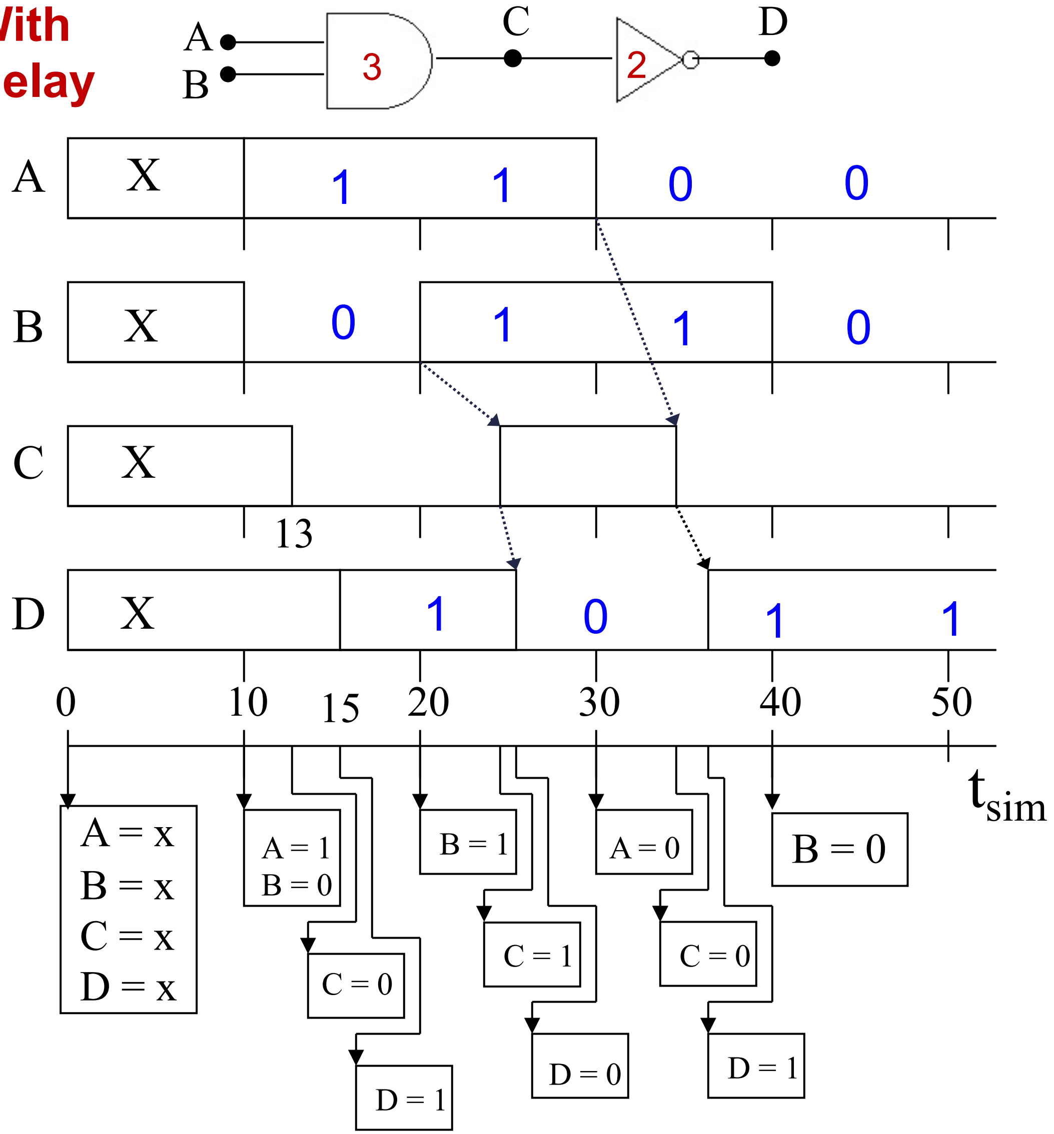
- **`initial`** declares one-shot behaviors
- **`$monitor`** is used to observe events
- **`$display`** is used to print out the values
- **`$time`** returns simulation time
- **`$finish`** returns control to operating system

Simulation Waveform

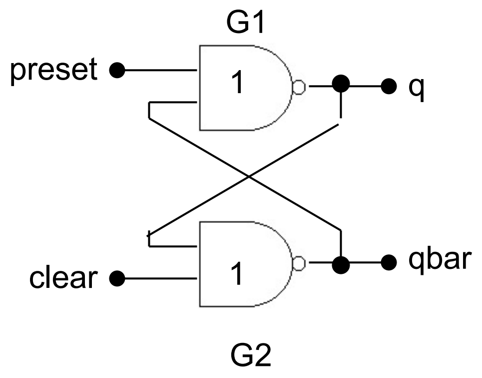
Without Delay



With Delay

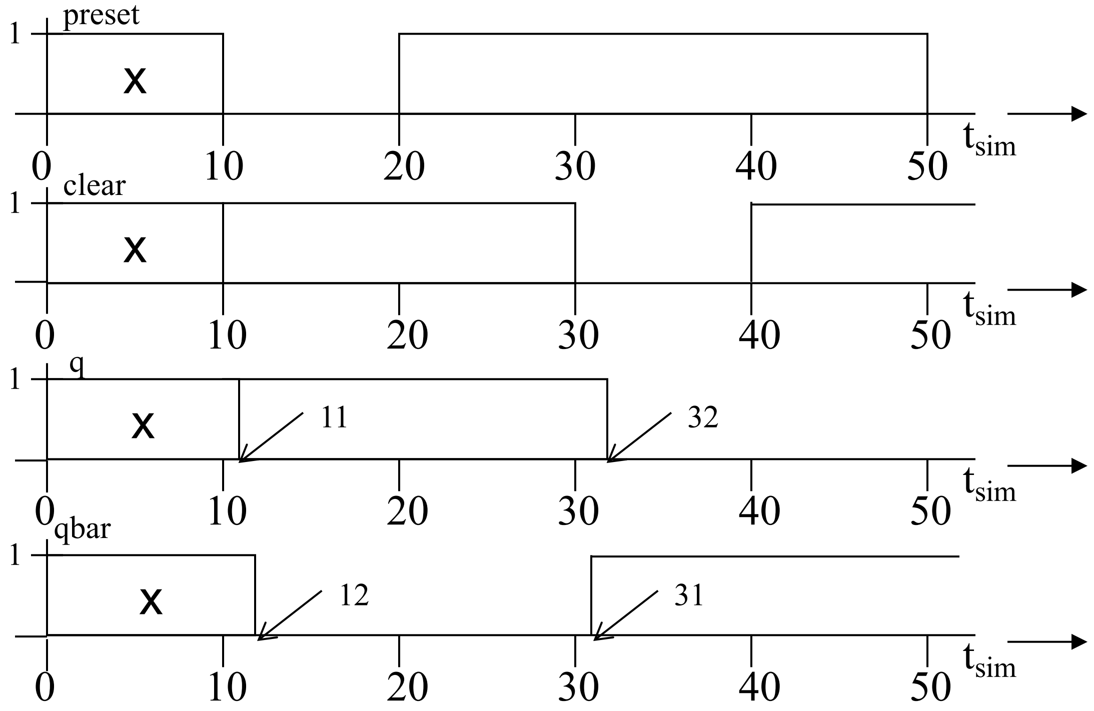


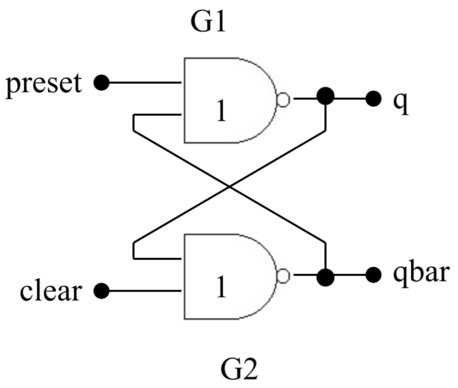
Example: NAND-Type Latch




```
module Nand_Latch_1 (q, qbar, preset, clear);
  output     q, qbar;
  input      preset, clear;

  nand #1    G1 (q, preset, qbar),
             G2 (qbar, clear, q);
endmodule

module test_Nand_Latch_1;        // Design Unit Testbench
  reg        preset, clear;
  wire       q, qbar;

  Nand_Latch_1 M1 (q, qbar, preset, clear);     // Instantiate UUT

  initial                  // Create DUTB response monitor
   begin
    $monitor ($time, "preset = %b clear = %b q = %b qbar = %b", preset, clear, q, qbar);
   end

  initial
   begin                   // Create DUTB stimulus generator
    #10     preset     =0;         clear = 1;
    #10     preset     =1;
    #10     clear      =0;
    #10     clear      =1;
    #10     preset     =0;
   end

  initial
   #60      $finish;     // Stop the simulation
endmodule
```

Simulation Results





0	preset = x	clear = x	q = x	qbar = x
10	preset = 0	clear = 1	q = x	qbar = x
11	preset = 0	clear = 1	q = 1	qbar = x
12	preset = 0	clear = 1	q = 1	qbar = 0
20	preset = 1	clear = 1	q = 1	qbar = 0
30	preset = 1	clear = 0	q = 1	qbar = 0
31	preset = 1	clear = 0	q = 1	qbar = 1
32	preset = 1	clear = 0	q = 0	qbar = 1
40	preset = 1	clear = 1	q = 0	qbar = 1

Tasks and Functions

Tasks and Functions

- High-level structural description in Verilog modules
  - Must be defined in a module
  - Local to the module
  - Can have local variables
    - Registers, time, integers, real
  - No `initial`/`always` blocks
- Tasks
  - Macro for common code segment
  - Delays, timing, event constructs
  - Multiple output arguments
- Functions
  - Conversions or calculations
  - Purely combinational
    - Zero simulation time
  - Only one output

Tasks

Tasks: Simple Example

```
module operation;
  parameter delay = 10;
  reg [15:0] A, B;
  reg [15:0] AB_AND, AB_OR, AB_XOR;
  always @* begin
    bitwise_oper(AB_AND, AB_OR, AB_XOR, A, B);
  end

  task bitwise_oper;
    output [15:0] ab_and, ab_or, ab_xor;
    input [15:0] a, b;
    begin
      #delay ab_and = a & b;
      ab_or = a | b;
      ab_xor = a ^ b;
    end
  endtask
endmodule
```

macro

alias

Tasks: ANSI C Style (Verilog 2001)

```
task bitwise_oper(
  output [15:0] ab_and, ab_or, ab_xor,
  input [15:0] a, b);
  begin
    #delay ab_and = a & b;
    ab_or = a | b;
    ab_xor = a ^ b;
  end
endtask
```

Tasks: Asymmetric Sequence Generator

```
reg clock;
initial
  init_sequence;
always
  asymmetric_sequence;

task init_sequence;
  begin
    clock = 1'b1;
  end
endtask

task asymmetric_sequence;
  begin
    #12 clock = 1'b0;
    #5 clock = 1'b1;
  end
endtask
```

Functions

Functions: Simple Example (Parity Calculation)

```
module parity;
  reg [31:0] addr;
  reg parity;

  always @(addr) begin
    parity = calc_parity(addr);
    $display("Parity calculated = %b",
      calc_parity(addr));
  end

  function calc_parity;
    input [31:0] address;
    begin
      calc_parity = ^address;
    end
  endfunction
endmodule
```

Functions: ANSI C Style (Verilog 2001)

```
function calc_parity (input [31:0] address);
  begin
    calc_parity = ^address;
  end
endfunction
```

Functions: Another Example (Left/Right Shifter)

```
`define LEFT_SHIFT      1'b0
`define RIGHT_SHIFT     1'b1
module shifter;
   reg [31:0] addr, left_addr, right_addr;
   reg control;
   always @(addr) begin
      left_addr = shift(addr, `LEFT_SHIFT);
      right_addr = shift(addr, `RIGHT_SHIFT);
   end

   function [31:0] shift;
     input [31:0] address;
     input control;
     begin
       shift = (control == `LEFT_SHIFT) ?
         (address << 1) : (address >> 1);
     end
   endfunction
 endmodule
```

Comparison between Tasks and Functions

Functions	Tasks
Can enable another function but not another task	Can enable another task and function
Zero simulated time	Possible non-zero simulated time
No delay, event, timing control	Can have delay, event, timing control
One or more inputs	Zero or more arguments of input/output/inout
Always return a single value	Do not return a value