Computer-Aided VLSI System Design

Chap.2-1 Signing and Sizing in Verilog

Lecturer: 張惇宥

Graduate Institute of Electronics Engineering, National Taiwan University

NTU GIEE

Outline

- Signing and Sizing in Verilog
- Behavioral Level Modeling and Event-Based Simulation
- Procedural Construct and Assignment
  - **`initial`** block
  - **`always`** block
  - procedural assignment
- Data Path Modeling
  - Data Path
  - Timing Parameters
- Control Construct
- Functional Block
  - Sub-modules
  - **`function`** and **`task`**
  - **`generate`**
- Finite State Machine (FSM)
  - Moore Machine & Mealy Machine
  - Behavior Modeling of FSM

Review: RT Level Modeling

- How does Verilog determine the bit-length and the sign in expression?

```
wire [15:0] addr;
wire out, cout;

wire [2:0] a_low = a[3:0];
// net declaration assignment, a_low[2:0] = a[2:0]
assign out = i1 & i2;
// i1 and i2 are nets
assign addr[15:0] = addr1[15:0] ^ addr2[15:0];
// Continuous assign for vector nets addr is a 16-bit vector net
// addr1 and addr2 are 16-bit vector registers
assign {cout, sum[3:0]} = a[3:0] + b[3:0] + cin;
// LHS is a concatenation of a scalar net and vector net
```

Sizing and Signing

- What are the results of the following expressions

```
reg [31:0] A, B, C, D, E;

A = -12 / 3;                    // -4
B = -'d12 / 3;                  // 1431655761
C = -'sd12 / 3;                 // -4
D = -'sd12 / $signed('d3);      // -4
E = -5'sd12 / 3'sd3;            // -4
```

```
reg [7:0] o;
reg [3:0] a, b;
reg signed [3:0] c;

a = 4'd15; b = 4'd8; c = -4'd1;
o = (a + b) + c;                // 38
```

Signing (1/3)

- Signals are **"regarded as"** signed in Verilog
  - In signed operations, the bit pattern is viewed as 2's complement
  - The bit pattern itself does not change

$(1111)_2$ ⇨ signed data type ⇨ $(-1)_{10}$
$(1111)_2$ ⇨ unsigned data type ⇨$(15)_{10}$

- The number without size and base format specified is a signed decimal
- The number with the base format should be viewed as unsigned unless the ***s*** designator is included

$12 = (00\ldots001100)_2$
$-12 = (11\ldots110100)_2 = -12$ (signed)
$-'d12 = (11\ldots110100)_2 = 4291967296$ (unsigned)
$-'sd12 = (11\ldots110100)_2 = -12$ (signed)

Sign Conversion

- A built-in system functions to handle type casting
  - **`$signed()`:** Return value is signed
  - **`$unsigned()`:** Return value is unsigned

- Example

```
wire [7:0] a, b, c;
assign a = $unsigned(-4)      // a = 8'b11111100
assign b = $unsigned(-4'sd4) // b = 8'b00001100
assign c = $signed(4'b1100)  // c = 8'b11111100
```

Signed Signal (1/2)

- Signed addition and subtraction
  - Equivalent to unsigned signals
- Signed multiplication, division, and comparison
  1. Using the **`signed`** declaration of regs and wires

```
wire signed [7:0] a;
wire signed [7:0] b;
wire cmp;
assign cmp = a < b;
```

  2. Or use **`$signed()`** function to cast unsigned variables

```
wire [7:0] a, b;
wire cmp;
assign cmp = $signed(a) < $signed(b);
```

  3. (constant) using the ***`s`*** designator

```
wire [7:0] a;
wire cmp;
assign cmp = $signed(a) < -8'sd2;
```

Signed Signal (2/2)

- Some expressions will convert the signed signal to unsigned
  - Bit-select results are unsigned
  - Part-select results are unsigned, even if the part-select specifies the entire vector
  - Concatenate results are unsigned
  - Comparison results (1, 0) are unsigned

```
reg signed [7:0] a, b;
reg signed [16:0] c;

c = a[7:0]; // {{8{1'b0}}, a}
// b[7:0] is unsigned and therefore zero-extended

c = {a, b}; // {1'b0, {a, b}}
// {a, b} is unsigned and therefore zero-extended
```

Sizing and Signing Rules

- Verilog evaluating an expression by the following steps

1. **Determine the sign of the right-hand side (RHS), then coerce all RHS to the result type**
   - If all operands are signed, the result is signed
   - If any operand is unsigned, the result is unsigned
   - The LHS should not be considered
2. **Determine the expression size by choosing the largest operands size, including the size of the LHS**
3. **Resize all RHS operands to the expression size**
   - Signed number is signed extension
   - Unsigned number is zero extension
4. **Evaluate the RHS expression, producing a result of type and size found in step1-2**
5. **Assign RHS value to LHS**

Sizing and Signing Examples (1/6)

```
wire [3:0] a;
wire signed [3:0] b;
wire signed [7:0] out;
assign out = a + b;
```

```
> a = 5 (0101), b = -2 (1110)
> a = 5 (0000_0101), b = -2 (0000_1110)
> out = 19 (0001_0011)
```

- a is unsigned, b is signed ⇨ result is unsigned ⇨ **zero** extension

```
wire signed [3:0] a;
wire signed [3:0] b;
wire signed [7:0] out;
assign out = a + b;
```

```
> a = 5 (0101), b = -2 (1110)
> a = 5 (0000_0101), b = -2 (1111_1110)
> out = 3 (0000_0011)
```

- a is signed, b is signed ⇨ result is signed ⇨ **sign** extension

```
wire signed [3:0] a;
wire signed [3:0] b;
wire [7:0] out;
assign out = a + b;
```

```
> a = -3 (1101), b = -2 (1110)
> a = -3 (1111_1101), b = -2 (1111_1110)
> out = 251 (1111_1011)
> $signed(out) = -5 (1111_1011)
```

Sizing and Signing Examples (2/6)

- Relational and equality operator
- **The result of part-select is unsigned**

```
reg signed [3:0] a;
reg [3:0] b;
reg signed [2:0] c;
reg cmp;

a = -1; b = 15; c = 7;

cmp = a > 3;                    // 0, signed
cmp = a > 'd3;                  // 1, unsigned 1111 > 0011
cmp = a > 4'sd3;                // 0, signed
cmp = a[3:0] > 4'sd3;           // 1, unsigned 1111 > 0011
cmp = (a == b);                 // 1
cmp = (a == $signed(b));        // 1
cmp = (a == c);                 // 1, signed extension
```

Sizing and Signing Examples (3/6)

- Average of two number

```
reg [3:0] a, b, avg;          // a = 14, b = 10
reg [4:0] avg_good;

// will not work properly, avg = 4
avg = (a + b) >> 1;

// will work correctly, avg = 12
avg = (a + b + 0) >> 1;

// will work correctly, avg = 12
avg_good = (a + b) >> 1;
```

Sizing and Signing Examples (4/6)

- Using sign extension improperly

```
reg signed [15:0] o, o_bad;
reg signed [3:0] a, b;          // a = -4, b = 5

// will not work properly, o_bad = 1260
o_bad = { {4{a[3]}}, a } * { {4{b[3]}}, b };

// will work correctly, o = -20
o = a * b;
```

Sizing and Signing Examples (5/6)

- Signed 16-bit multiplier

```
wire [7:0] a, b;                  // a = 100, b = -100
wire signed [15:0] c, c_bad;

// will not work properly, c_bad = -16
assign c_bad = $signed( $signed(a) * $signed(b) );

// will work correctly, c = -10000
assign c = $signed(a) * $signed(b);
```

- The expression in **`$signed()`** should be evaluate first
  - The width of `$signed(a) * $signed(b)` in the c_bad expression is 8 bits, so this expression gives an incorrect result

Sizing and Signing Examples (6/6)

- Conditional operator

```
wire [7:0] a, b;                    // a = 100, b = -100
wire [15:0] c, d;                   // d = 66
wire condition;                     // condition = 1

// unsigned, c = 15600
assign c = (condition) ? $signed(a) * $signed(b) : d;

// signed, c = -10000
assign c = $signed(condition) ?
       $signed(a) * $signed(b) : $signed(d);

// signed, c = -10000
assign c = (condition) ? $signed(a) * $signed(b) : $signed(d);
```

Computer-Aided VLSI System Design

Chap.2-2 Logic Design at Behavioral Level

Graduate Institute of Electronics Engineering, National Taiwan University

NTU GIEE

Outline

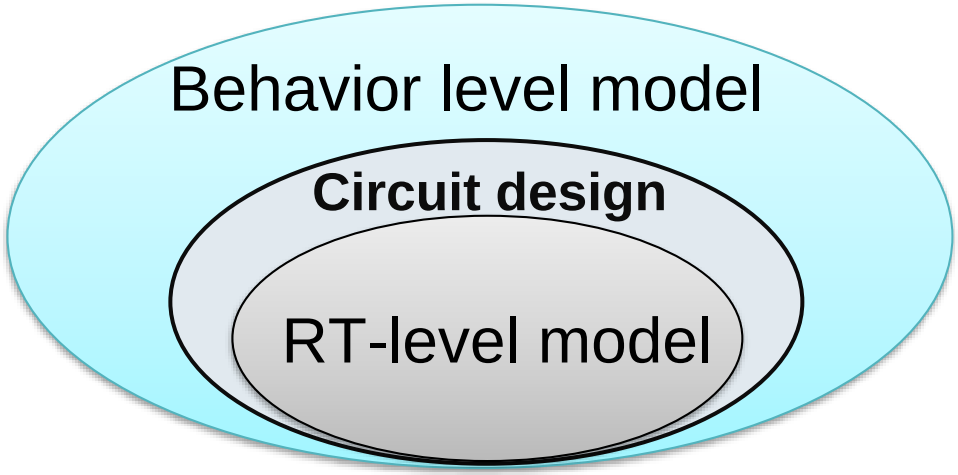
- Behavioral Level Modeling and Event-Based Simulation
- Procedural Construct and Assignment
  - **`initial`** block
  - **`always`** block
  - procedural assignment
- Data Path Modeling
  - Data Path
  - Timing Parameters
- Control Construct
- Functional Block
  - Sub-modules
  - **`function`** and **`task`**
  - **`generate`**
- Finite State Machine (FSM)
  - Moore Machine & Mealy Machine
  - Behavior Modeling of FSM

What is Behavioral Level

- Literally, modeling circuits using “behavioral” descriptions and events
- Description of an RTL model
  - Structure: constructs are separated for combinational and sequential circuits
  - Signal: continuous evaluate-update, pin accurate
  - Timing: cycle accurate
- Description of a behavioral level model
  - Structure: construct can be combinational, sequential, or hybrid circuits.
  - Signal: event-driven, timing, arithmetic (floating point or integer, … to pin accurate)
  - Timing: untimed (ordered), approximate-timed (with delay notification), cycle accurate
- **Note: Only a small part of the behavioral level syntax is used for describing circuits, others are widely used for verification (testbench)** **(Behavioral Modeling != non-synthesizable)**

Relationship between RTL and Behavioral Level

- Behavior level model
  - Hardware modeling
  - Implicit structure description for modeling
  - Flexible



- RTL level model
  - Hardware modeling
  - Explicit structure description for modeling
  - Accurate

Various Abstraction of Verilog (1/2)

- RT-Level (RTL) Verilog description of a full adder

```
module fadder(cout, sum, a, b, cin);
// port declaration
  output  cout, sum;
  input   a, b, cin;
  wire    cout, sum;

// RTL description
  assign  sum = a^b^cin;
  assign  cout = (a&b)|(b&cin)|(cin&a);

endmodule
```

Whenever a or b or c changes its logic state, evaluate *sum* and *cout* by using the equation

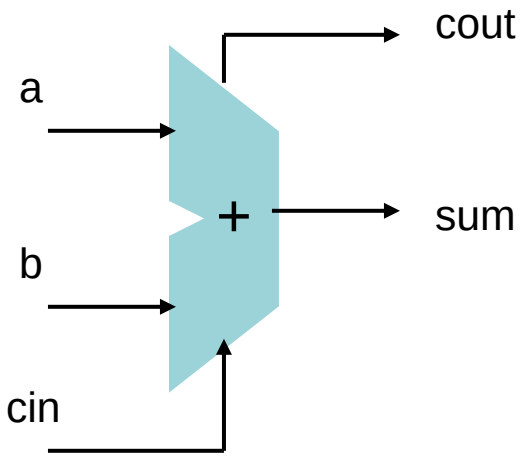
$sum = a \oplus b \oplus ci$

$cout = ab + bc + ca$

Various Abstraction of Verilog (2/2)

- Behavioral level Verilog description of a full adder

```
module fadder(cout, sum, a, b, cin);
// port declaration
  output  cout, sum;
  input   a, b, cin;
  reg     cout, sum;
// behavior description
  always @(a or b or cin)
    begin
        {cout,sum} = a + b + cin;
    end
endmodule
```

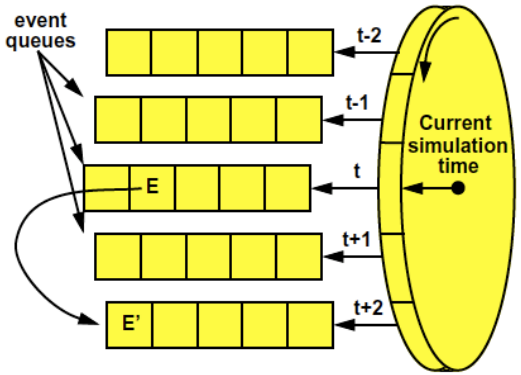


Event

- A data type has state and timing description
  - state: level change (1→0, 0→1, 1→X, 0→Z, etc.), edge (defined logic-level transition: e.g. 0→1)
  - timing: virtual time in simulator
  - example: signal s changed from 0 to 1 at 3ns

Event-Based Simulation

- Execute statement (evaluate expression and update variable) when defined event occurs
  - input transition cause an event on circuit
  - simulation is on the OR-ed occurrence of sensitive events
- Benefits
  - Accelerate simulation speed: Only evaluate expression when variables on RHS change
  - Allow high-level description (behavior) of a constructor



A current event scheduling another event

Outline

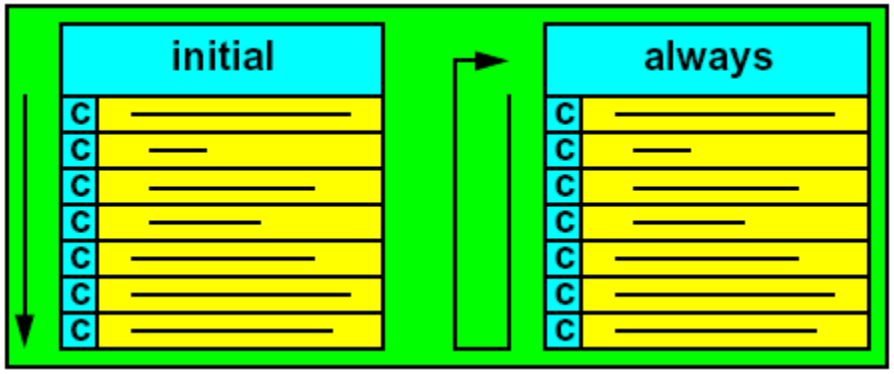
- Behavioral Level Modeling and Event-Based Simulation
- Procedural Construct and Assignment
  - **`initial`** block
  - **`always`** block
  - procedural assignment
- Data Path Modeling
  - Data Path
  - Timing Parameters
- Control Construct
- Functional Block
  - Sub-modules
  - **`function`** and **`task`**
  - **`generate`**
- Finite State Machine (FSM)
  - Moore Machine & Mealy Machine
  - Behavior Modeling of FSM

Procedural Blocks (1/3)

- There are two kinds of procedural block
  - **`initial`**: execute only once for initialization or waveform generation (**NON**-SYNTHESIZABLE)
  - **`always`**: execute in a loop

There's no circuit can work only once then disappear

- The ***initial*** and ***always*** procedural blocks are enabled at the beginning of a simulation
- All procedural blocks execute **concurrently**



Procedural Blocks (2/3)

- Procedural blocks contain the following components
- **Procedural assignment**
  - Describe the dataflow within the block
- **High-level constructs**
  - Describe the functional operation of the block
- **Timing controls**
  - To trigger the block and control the execution of the statements in the blocks

```
always @(*) begin
if (a > 0)
        c = #10 a – b;
else
        c = #5 a;
end
```

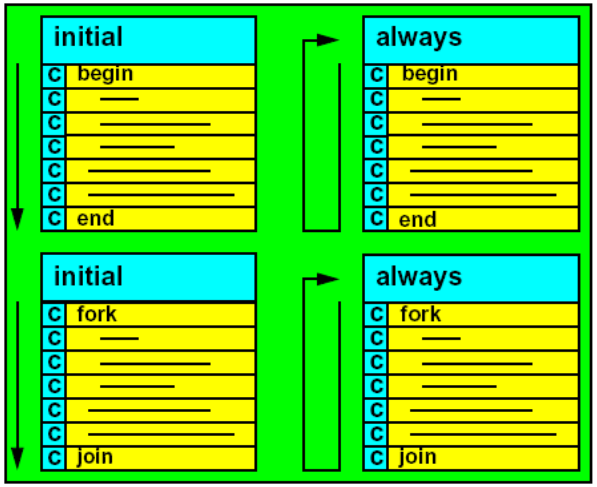
Timing control

High-level construct

Procedural assignment

Procedural Blocks (3/3)

- Sequential block: **`begin`** and **`end`** group sequential procedural statements
- Concurrent block: **`fork`** and **`join`** group concurrent procedural statements (**NON**-SYNTHESIZABLE, not for modeling circuits)
- If the procedural block contains only one assignment, then both **`begin-end`** and **`for-join`** can be omitted
- Two types of blocks differ only if there's timing control inside



Two blocks do the same operations!

```
begin          fork
  #5 a = 1;      #5  a = 1;
  #5 a = 2;      #15 a = 3;
  #5 a = 3;      #10 a = 2;
end            join
```

Example

```
module assignment_test;
  reg [3:0] r1,r2;
  reg [4:0] sum2;
  reg [4:0] sum1;

  always @(r1 or r2)
    sum1 = r1 + r2;

  initial
  begin
      r1 = 4'b0010; r2=4'b1001;
      sum2 = r1+r2;
      $display(“ r1   r2   sum1   sum2”);
      $monitorb(r1, r2, sum1, sum2);

      #10 r1 = 4'b0011;
  end

endmodule
```

Result

r1	r2	sum1	sum2
0010	1001	01011	01011
0011	1001	01100	01011

Timing Control

- Procedural timing controls can be achieved by the following three methods

1. A delay-based timing control (**NON**-SYNTHESIZABLE)
   - `# (delay)`
2. An event-based timing control
   - `@ (sensitivity list)`
3. A level sensitive timing control (**NON**-SYNTHESIZABLE)
   - `wait(<expression>)`

Delay-Based

- Using simple delay to delay stimulus in testbench or to approximate real-world delays in behavioral models
- Simple delay (# delay) is **NON**-SYNTHESIZABLE
  - because it is difficult to design a circuit that has constant delay under any environment (temperature, transistor… variant)
- Example: clock generation in the testbench

```
reg clk;
parameter CYCLE = 10;

initial clk = 0;
always # (CLCLE/2) clk = ~clk;
```

Event-Based (1/2)

```
always@ ( a or b or cin)
  begin
    {cout, sum} = a + b + cin;
  end
```

- Edge-sensitive control (**@**)
  - The sensitivity list is described after "**always @**"
  - This means if any signals inside change (have an edge in waveform), the **always** block is triggered.
  - Keywords **or** are used to separate multiple signals
  - Keywords **posedge** & **negedge** is used when the **always** block should be triggered by positive or negative edge transition of the signal
  - Missing signals in sensitivity list may lead to wrong results!

WRONG!

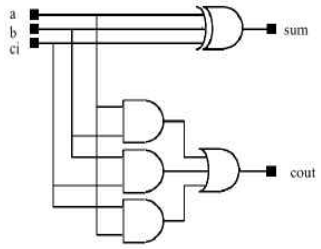
```
// initial a=0, b=0, x=0,  y=1
always@(a or b) begin  // 1. b changes to 1
    y = ~x;            // 2. y remains 1
    x = a | b;         // 3. x changes to 1
end
```

CORRECT!

```
// initial a=0, b=0, x=0,  y=1
always@(a or b or x) begin // 1. b changes to 1
    y = ~x;                // 2. y remains 1
                           // 4. y changes to 0
    x = a | b;             // 3. x changes to 1,
                                  trigger again!
                           // 5. x remains 1
end
```

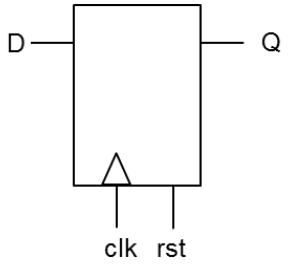
Event-Based (2/2)

- Easy way to use sensitivity list
  - For combinational circuit
    - Pure logic circuit
    - Use always @ (*)
  - For sequential circuit
    - D-FF circuit
    - Edge trigger of clock or reset signal




```
always @(*) begin
    {cout, sum} = ci + a + b;
end
```

combinational circuit




```
always @(posedge clk or posedge rst)
begin
    if (rst) Q <= 0;
    else     Q <= D;
end
```

sequential circuit

Level Sensitive Timing Control

- Using ***wait*** for level-sensitive timing control

  wait(<expression>)

- Wait until the expression is asserted
- ***wait*** is **NON**-SYNTHESIZABLE
- Example

```
module D_latch (D, Q, enable);
input D, enable;
output Q;
always @ (D) begin
    wait(enable)
    Q = D;
// change of Q will be ignored
when it is still waiting
end
endmodule
```

```
module testbench;
initial # (TIME_OUT) $finish;

initial begin
    wait(o_finish)
    // check output data here
    #10
    $finish;
end
endmodule
```

Procedural Assignment

- Assignments inside procedural blocks are called procedural assignment
- The left-hand side (LHS) of a procedural assignment
  - register data type (e.g. ***reg***)
- The data type of RHS signals is not restricted
- The variable without declared is default as ***wire***, which may cause errors in the procedural block
- Two types of procedural assignment statements
  - **Blocking assignment (=)**
  - **Non-blocking assignment (<=)**

Blocking & Non-blocking Assignments

- There are two types of procedural assignment statements: blocking (=) and non-blocking (<=)
- Blocking assignment **(=)**
  - Evaluate the RHS and pass to the LHS
  - Suitable for model or design the **combinational** circuit
  - **Difficult to model the concurrency**
- Non-blocking assignment **(<=)**
  - Evaluate the RHS, but schedule the LHS
  - Update LHS only after evaluate all RHS
  - Greatly simplify modeling concurrency

Blocking or Non-Blocking?

- Blocking assignment
  - Evaluation and assignment are immediate

```
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 assignment deferred until all right-hand sides have been evaluated (end of the virtual timestamp)

```
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                   4. Assign x, y, and z with their new values
```

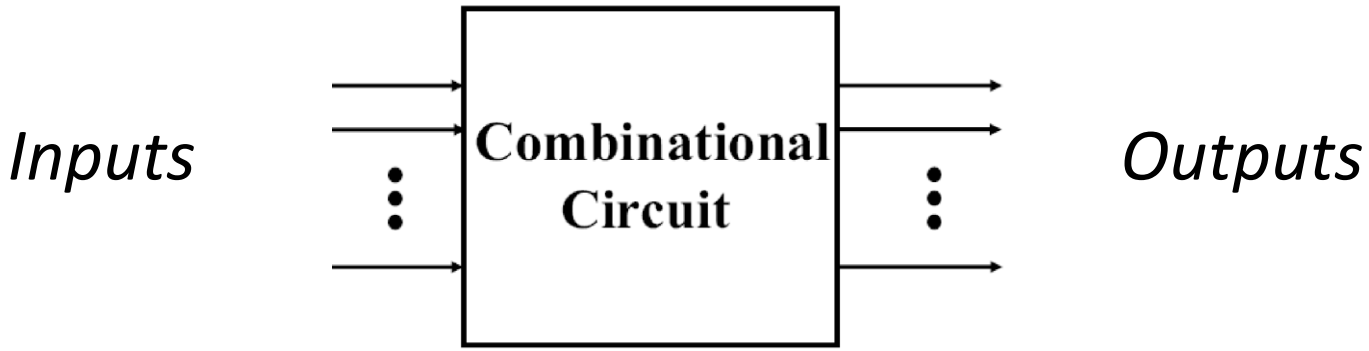
Outline

- Behavioral Level Modeling and Event-Based Simulation
- Procedural Construct and Assignment
  - **`initial`** block
  - **`always`** block
  - procedural assignment
- Data Path Modeling
  - Data Path
  - Timing Parameters
- Control Construct
- Functional Block
  - Sub-modules
  - **`function`** and **`task`**
  - **`generate`**
- Finite State Machine (FSM)
  - Moore Machine & Mealy Machine
  - Behavior Modeling of FSM

Combinational Data Path

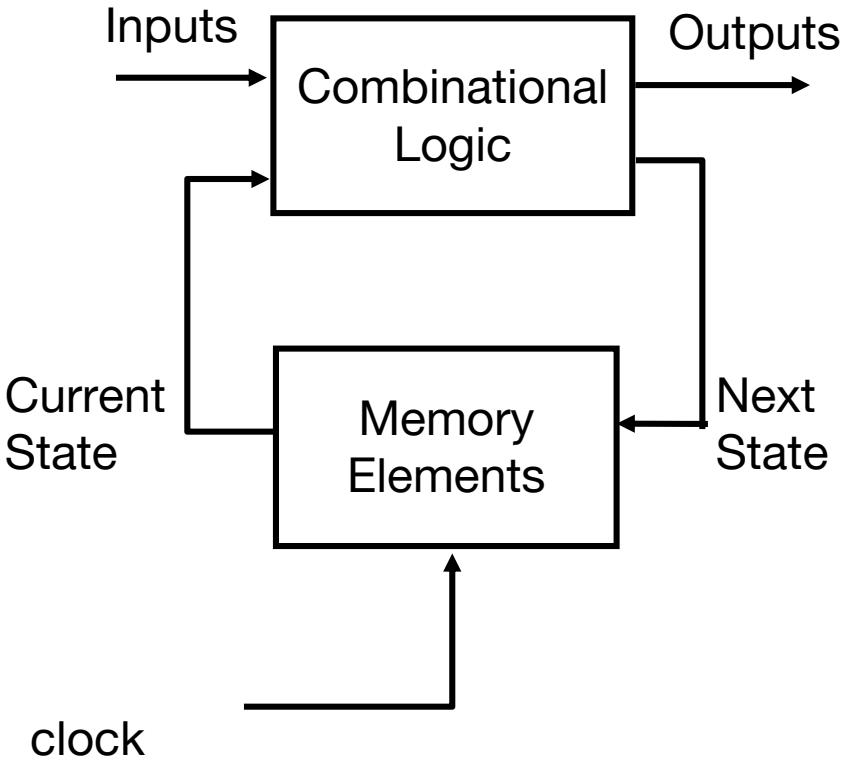
- Combinational logic circuits are **memoryless**
  - Any transition in inputs affect the whole circuit right away
  - Feedback path is not allowed

- Output can have logical transitions before settling to a stable value



Sequential Data Path

- Sequential circuits have memory (i.e. remember the past)
- The output is
  - depend on inputs
  - depend on current state
- In synchronous system
  - clock orchestrates the sequence of events
- Fundamental components
  - Combinational circuits
  - Memory elements



Modeling of Flip-Flops (1/2)

- The use of **posedge** and **negedge** makes an **always** block sequential (edge-triggered)
- Unlike combinational always block, the sensitivity list does determine the behavior of synthesis

D Flip-flop with synchronous clear

```
module dff_sync_clear(d, clearb,
clock, q);
input d, clearb, clock;
output q;
reg q;
always @ (posedge clock)
begin
  if (!clearb) q <= 1'b0;
  else q <= d;
end
endmodule
```

`always` block entered only at each positive clock edge

D Flip-flop with asynchronous clear

```
module dff_async_clear(d, clearb, clock, q);
input d, clearb, clock;
output q;
reg q;

always @ (negedge clearb or posedge clock)
begin
  if (!clearb) q <= 1'b0;
  else q <= d;
end
endmodule
```

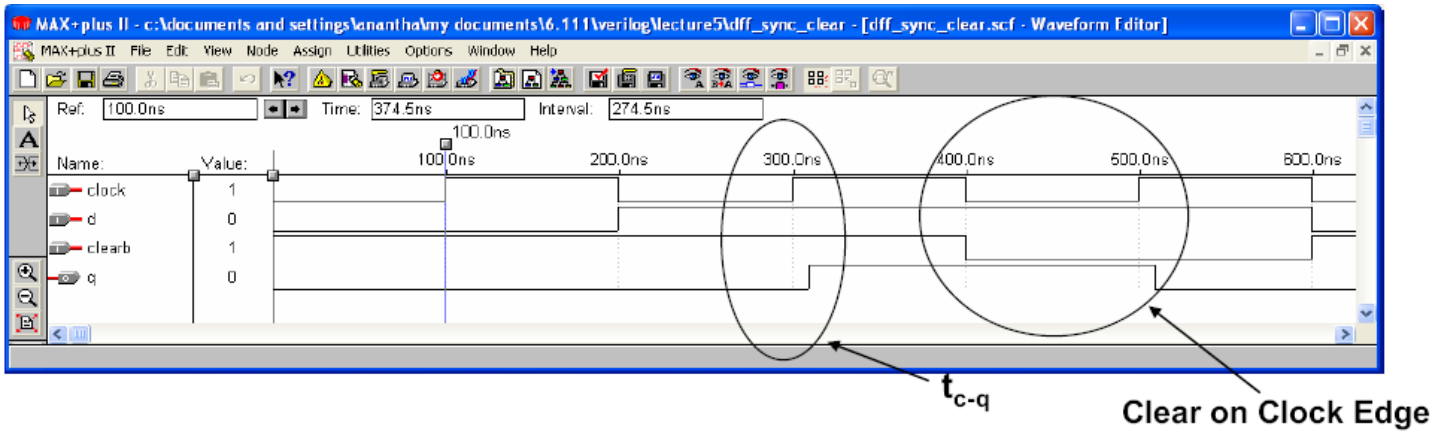
`always` block entered immediately when (active-low) clearb is asserted

Note: The following is **incorrect** syntax: `always @ (clear or negedge clock)`

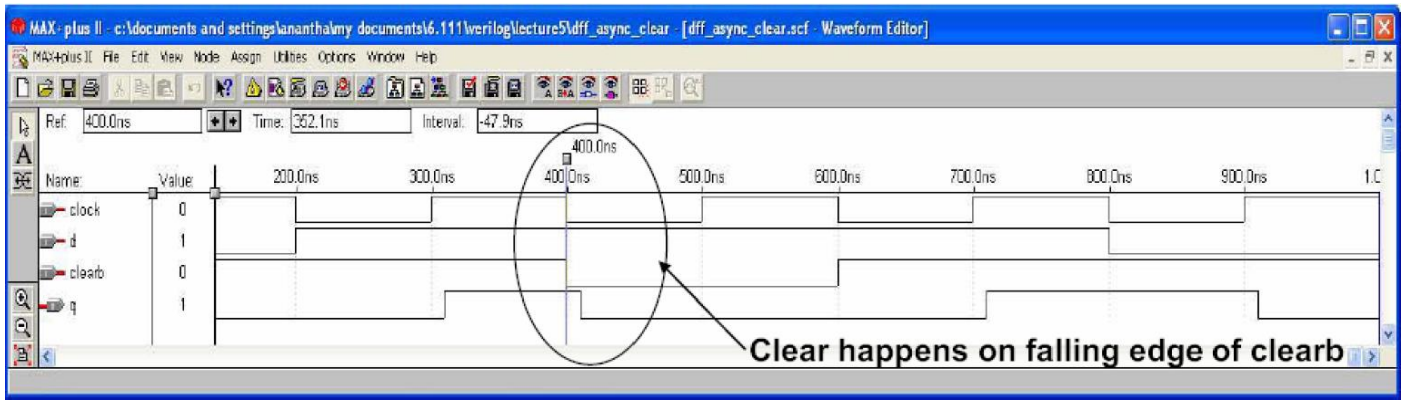
If one signal in the sensitivity list uses posedge/negedge, then all signals must.

Modeling of Flip-Flops (2/2)

- Synchronous Reset



- Asynchronous Reset



Outline

- Behavioral Level Modeling and Event-Based Simulation
- Procedural Construct and Assignment
  - **`initial`** block
  - **`always`** block
  - procedural assignment
- Data Path Modeling
  - Data Path
  - Timing Parameters
- Control Construct
- Functional Block
  - Sub-modules
  - **`function`** and **`task`**
  - **`generate`**
- Finite State Machine (FSM)
  - Moore Machine & Mealy Machine
  - Behavior Modeling of FSM

Conditional Statements (1/2)

- If and If-else statements

```
if (expression)
   statement
else
   statement
```

```
if (expression)
   statement
else if (expression)
   statement
else
   statement
```

```
if (sel == 0)
begin
    a_w = data;
    b_w = b_r;
end
else begin
    b_w = data;
    a_w = a_r;
end
```

```
a_w = a_r;
b_w = b_r;

if (sel == 0)
    a_w = data;
else
    b_w = data;
```

```
if (sel == 0)
    a <= data;
else
    b <= data;
```

- Restrictions compared with C
  - LHS in all cases should be the same!
    - Avoid latch
  - Conditions should be full-case, if must be followed by else!
    - Avoid latch
  - In short, think about MUX!

```
if (sel == 0)
    a_w = data;
else
    b_w = data;
```

```
if (sel == 0)
    a_w = data;
```

Conditional Statements (2/2)

- Case statement

```
case (state_r)
    2'b00: state_w = 2'b01;
    2'b01: begin
        state_w = 2'b10;
        o_valid = 1'b1;
    end
    default: begin
        state_w = state_r;
        o_valid = 1'b0;
    end
endcase
```

Multiway Branching

- 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

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

Alternative way:

```
always @(*) begin
    y = a;

    case(sel)
        3: y = d;
        2: y = c;
        1: y = b;
    endcase
end
```

NOTICE:
Not full case would cause latch

if-else statement

```
if (sel===3)
  y = d;
else
  if (sel===2)
    y = c;
  else
    if (sel===1)
      y = b;
    else
      y = a;
```

case statement

```
case (sel)
      3: y = d;
      2: y = c;
      1: y = b;
default:y = a;
endcase
```

Looping Statements

- **The for loop (conditionally synthesizable)**
- The while loop (X)
- The repeat loop (X)
- The forever loop (X)

For Loop

- The keyword **for** is used to specify this loop. The **for** loop contain 3 parts:
  - An initial condition
  - A check to see if the terminating condition is true
  - A procedural assignment to change value of the control variable
- **Synthesizable only if the expanded form is synthesizable**
  - 沒有時間相依性
  - 不能和參數有關係

```
array[i+1] <= array[i] - 1;
array[i] =  2*i;
```

```
module bitwise_and(a, b, out);
  parameter size=2;
  input  [size-1:0] a, b;
  output [size-1:0] out;
  reg    [size-1:0] out;
  integer i;
  always @(a or b)
  begin
    for (i = 0;i < size; i = i + 1)
      out[i] = a[i] & b[i];
    end
endmodule
```

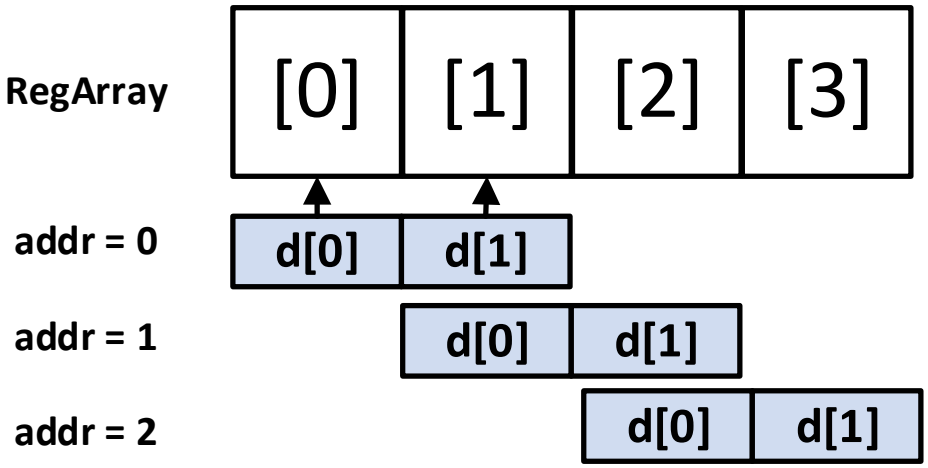
Example: RegFile

- The for-loop can express the register file
- There are no significant differences in area if the n-to-1 decoder is written explicitly

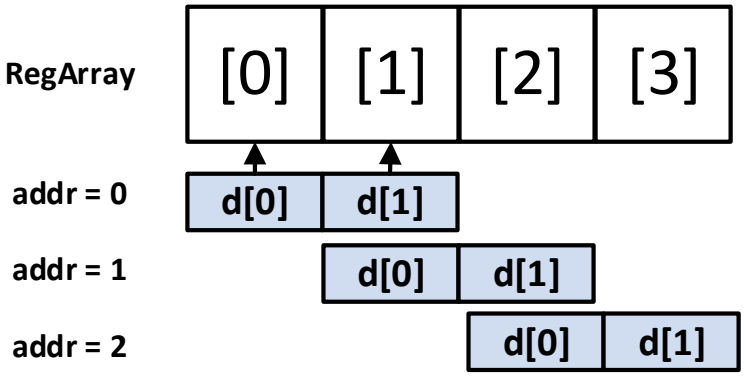
```
integer i;
always @(*) begin
  for (i=0; i<32; i=i+1) begin
    if ((wr_addr == i) && wr_enable) regfile_w[i] = wr_data;
    else                             regfile_w[i] = regfile_r[i];
  end
end
```

Example: "Asymmetry" RegArray

- In some cases, not every entry in the register array requires a n-to-1 MUX
- For example, it is impossible for d[1] assign to the entry [0] of register array
- Unrolling the for-loop can reduce the area in this case



Example: "Asymmetry" RegArray



Coding style	Comb. Area
for-loop	9731
case (Unrolled)	6178

Experiment on size(regArray) = 32

```
integer i;
always @(*) begin
  regArray_w = regArray_r;
  for (i=0; i<3; i=i+1) begin
    if (addr==i) begin
      regArray_w[i+:2] = d;
    end
  end
end
```

Using for-loop

```
always @(*) begin
  regArray_w = regArray_r;
  case (addr)
    0: begin
      regArray[0] = d[0];
      regArray[1] = d[1];
    end
    1: ...
  endcase
end
```

Using case (Unrolled)

While Loop

- The **while** loop executes until the **while**-expression becomes false
- **NON**-SYNTHESIZABLE

```
initial
  begin
    reg [7:0] tempreg;
    count = 0;
    tempreg = reg;
    while (tempreg)
      begin
        if (tempreg[0]) count = count + 1;
        tempreg = tempreg >> 1;
      end
```

rega = 101;

tempreg	count
101	1
010	1
001	2

Repeat Loop

- The keyword **repeat** is used for this loop. The **repeat** construct executes the loop a **fixed** number of times.
- **NON**-SYNTHESIZABLE

```
module multiplier(result, op_a, op_b);
  ...
  reg shift_opa, shift_opb;
  parameter size = 8;
  initial begin
    result = 0;  shift_opa = op_a;  shift_opb = op_b;
    repeat (size)
      begin
        #10  if (shift_opb[1])
                result = result + shift_opa;
        shift_opa = shift_opa << 1;
        shift_opb = shift_opb >> 1;
      end
    end
endmodule
```

default repeat 8 times

Forever Loop

- The keyword **forever** is used to express the loop. The loop does not contain any expression and executes forever until the **$finish** task is encountered
- **NON**-SYNTHESIZABLE

```
//Clock generation
//Clock with period of20 units)
reg clk;

initial
  begin
    clk=1'b0;
    forever #10 clk=~clk;
  end
```

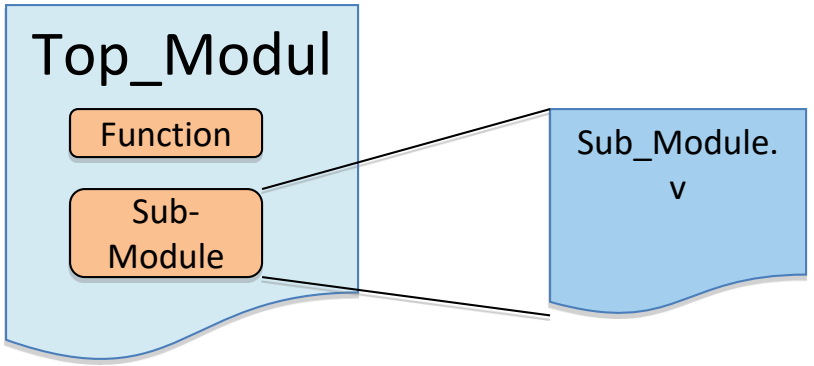
```
//Synchronize 2 register values
//at every positive edge of clock
reg clk;
reg x,y;

initial
    forever @(posedge clk) x=y;
```

Outline

- Behavioral Level Modeling and Event-Based Simulation
- Procedural Construct and Assignment
  - **`initial`** block
  - **`always`** block
  - procedural assignment
- Data Path Modeling
  - Data Path
  - Timing Parameters
- Control Construct
- Functional Block
  - Sub-modules
  - **`function`** and **`task`**
  - **`generate`**
- Finite State Machine (FSM)
  - Moore Machine & Mealy Machine
  - Behavior Modeling of FSM

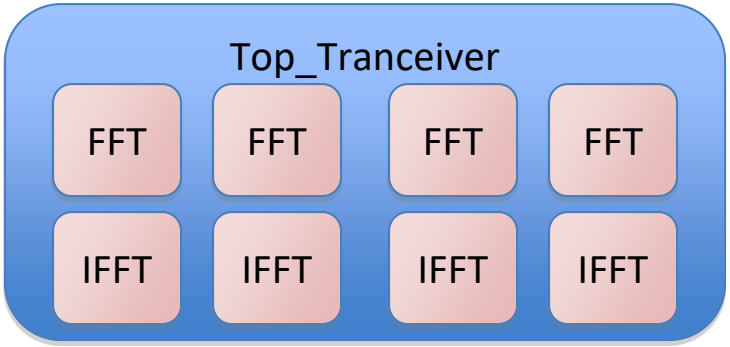
Functional Partition

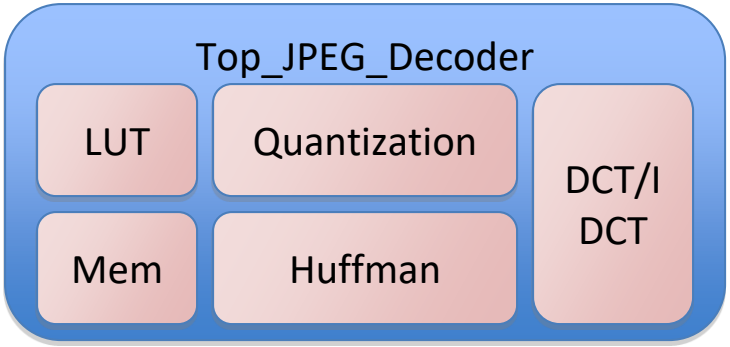
- A digital system may consist of many function blocks
  - A lumped Verilog module makes debugging & editing a great disaster
- Break the whole system into several function blocks
  - Sub-module
  - Function/Task



Sub-Module (1/2)

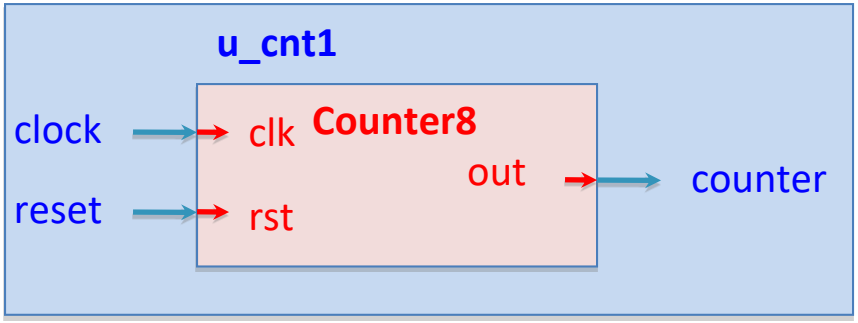
- When using a sub-module
  - Function block with many duplications
  - Function block containing specific computation





Sub-Module (2/2)

- Instantiate a sub-module



Top-module

```
reg clock, reset;
wire [2:0] counter;

Counter8 u_cnt1(
    .clk(clock),
    .rst(reset),
    .out(counter)
);
```

Sub-module

```
module Counter8(
    clk,
    rst,
    out
);
…
endmodule
```

Function (1/3)

- Functions are declared inside a module with the keywords **function** and **endfunction**.
- A function may be called from within procedural and continuous assignment statements
- It is typically used to perform a computation, conversions, or to represent combinational logic
  - No timing control
  - Only input arguments and exactly one return signal

```
function [ WIDTH-1: 0] my_function;
<I/O ports declaration> // only input ports
<local variables declaration> // register type
begin
    <function_statements>
end
endfunction
```

Function (2/3)

- Function name is regarded as the **`output reg`** of function
  - When a function is declared, a register with name (name of function) is declared implicitly inside.
  - Thus, the function cannot have more than 1 output.
- Functions cannot invoke other tasks. They can only invoke other functions

Function (3/3)

- Function example

```
// function definition
function [7:0] abs;          //unsigned
    input [7:0] number_in;   //signed
    begin
        abs = (number_in[7])?
                  (~number_in+1'b1): number_in;
    end
endfunction

// call function
reg [7:0] var1, abs_var1;
always@(var1) begin
    abs_var1 = abs(var1);
end
```

Task (1/6)

- Task are declared inside a module with the keyword **task** and **endtask**
- Task takes the control of the block, and return the control back to the block when the task is finished or disabled
- It can't be called from a **continuous assignment**
- It is typically used to perform debugging operations
  - Could contain timing control

NON-SYNTHESIZABLE

```
task my_task;
<I/O ports declaration> // input, output, inout ports
<local variables declaration> // register type
begin
    <task_statements> // performs the work of the task
end
endtask
```

Task (2/6)

- Task arguments shall be passed follow the order of their declaration
- The output shall be register type

```
task my_task;
input a, b;
output c;
begin
    # 10
    c = a & b;
end
endtask

reg a, b;
reg c;                  // wire c; => error
initial begin
    my_task(a, b, c);   // follow the declaration order
end
```

Task (3/6)

- Task arguments are evaluated when the task is invoked (pass by value)
- Signals used in timing controls (such as clk) must not be inputs to the task, because input values are passed into the task only **once**

```
// task definition
task light;
output color;
input [31:0] tics;
begin
    repeat (tics) @ (negedge clk);
    color = off;
end
endtask
// call task
always begin
    green = on;
    light(green, green_tics);
end
```

Task (4/6)

- Task example

```
module sequence;
…
reg clock;
…
initial
  init_sequence;
…
always begin
  asymmetric_sequence;
end
…
```

Directly copy & paste to where you use it

```
task init_sequence;
begin
 clock = 1'b0;
end
endtask

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

Task (5/6)

```
reg [7:0] a, b;

initial begin
    a = 1; b = 2;
    data_monitor(a, b);
    #10
    a = 10; b = 20;
    #30
    $display("@%d, simulation finish!", $time);
    $finish;
end

task data_monitor;
input [7:0] a, b;
begin
    #20
    $display("@%d, a = %d, b = %d", $time, a, b);
    #10
    $display("@%d, task finish!", $time);
end
endtask
```

```
// with input declaration
> @ 20, a =   1, b =   2
> @ 30, task finish!
> @ 70, simulation finish!
```

Task (5/6)

```
reg [7:0] a, b;

initial begin
    a = 1; b = 2;
    #10
    a = 10; b = 20;
    #30
    $display("@%d, simulation finish!", $time);
    $finish;
end

initial data_monitor(a, b);

task data_monitor;
input [7:0] a, b;
begin
    #20
    $display("@%d, a = %d, b = %d", $time, a, b);
    #10
    $display("@%d, task finish!", $time);
end
endtask
```

```
// with input declaration
// concurrent
> @ 20, a =   1, b =   2
> @ 30, task finish!
> @ 40, simulation finish!
```

Task (5/6)

```
reg [7:0] a, b;

initial begin
    a = 1; b = 2;
    #10
    a = 10; b = 20;
    #30
    $display("@%d, simulation finish!", $time);
    $finish;
end

initial data_monitor;

task data_monitor;
input [7:0] a, b;
begin
    #20
    $display("@%d, a = %d, b = %d", $time, a, b);
    #10
    $display("@%d, task finish!", $time);
end
endtask
```

```
// with input declaration
// concurrent
> @ 20, a =   10, b =   20
> @ 30, task finish!
> @ 40, simulation finish!
```

Task (6/6)

- Use **disable** to cancel the task

```
task errmon;
    forever@(posedge data_ready) begin
        if (golden!==data)
           $display("ERR:data=%b,expected=%b",data,golden);
           $finish;
    end
endtask
initial begin
    fork
        errmon;
        begin
            runtest;
            disable errmon;
        end
    join
end
```

Comparison: Function & Task

Function	Task
1. Is typically used to perform a computation, or to represent combinational logic 2. Called in procedural statement or continuous assignment	1. Is typically used to perform debugging operations, or to behaviorally describe hardware 2. Called in procedural statement
Can enable other functions	Can enable other tasks and functions
must **not contain any delay**, event, or timing control statements (**execute in 0 simulation time**)	may contain delay, event, or timing control statements
must **have at least one input argument**.	may have **zero or more arguments** of type input, output or inout
Always return a single value. They **cannot have output or inout** arguments	Do not return with a value but can pass multiple values through output and inout arguments

Generate

- Using ***generate*** to either conditionally or multiply instantiate generate blocks
- Can't contain the following things in generate block
  - port declarations
  - parameter declarations
- **Loop generate constructs**
  - Generate blocks multiple times
- **Conditional generate constructs**
  - if-generate constructs
  - case-generate constructs

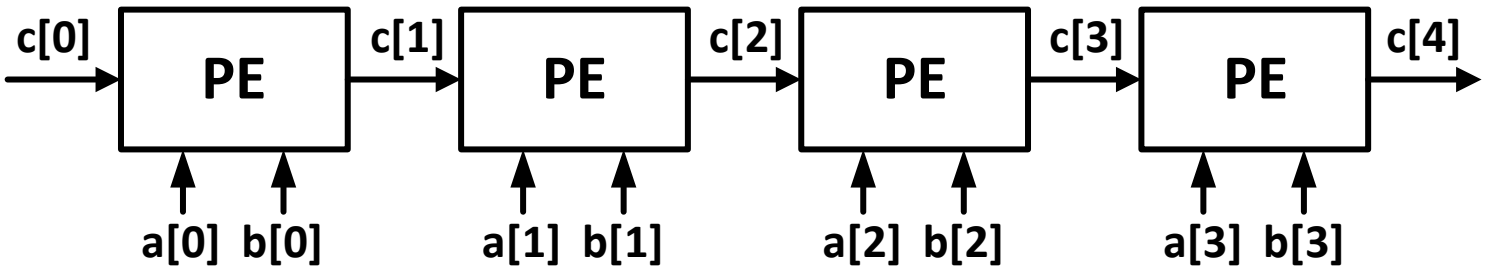
Loop Generate Constructs

- Loop generate can be used to describe the architecture that repeats many times
- Example: array architecture

```
module PE (clk, a, b, ci, co)
input clk;
input [7:0] a, b, ci;
output [7:0] co;
reg [7:0] co_r, co_w;
assign co = co_r;

always @(*) begin
    co_w = ci + a*b;
end
always @(posedge clk) begin
    co_r <= co_w;
end
endmodule
```

Loop Generate Constructs (cont.)

- Example: array architecture




```
parameter LEN = 4;
genvar i;
reg [7:0] a [0:LEN-1];
reg [7:0] b [0:LEN-1];
reg [7:0] c [0:LEN];
generate
for (i = 0; i < LEN; i = i+1) begin : array
    PE u1(i_clk, a[i], b[i], c[i], c[i+1]); // scope array[i].u1
end
endgenerate
```

Conditional Generate Constructs

- The following code can choose a suitable circuit based on the bit-width

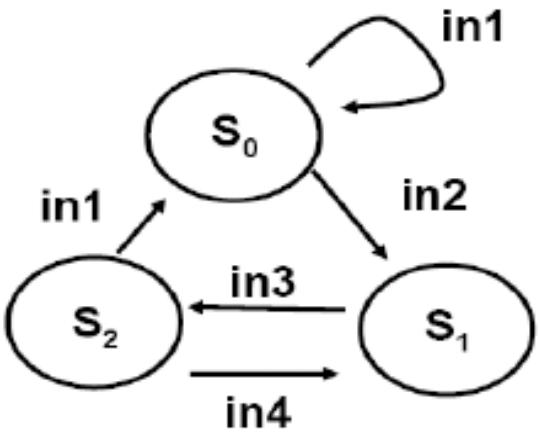
```
generate
  if((a_width < 8) || (b_width < 8)) begin: mult
    Low_Power_Module #(a_width,b_width) u1(a, b, out);
  end
  else begin: mult
    High_Performance_Module #(a_width,b_width) u1(a, b, out);
  end
endgenerate
```

Outline

- Behavioral Level Modeling and Event-Based Simulation
- Procedural Construct and Assignment
  - **`initial`** block
  - **`always`** block
  - procedural assignment
- Data Path Modeling
  - Data Path
  - Timing Parameters
- Control Construct
- Functional Block
  - Sub-modules
  - **`function`** and **`task`**
  - **`generate`**
- Finite State Machine (FSM)
  - Moore Machine & Mealy Machine
  - Behavior Modeling of FSM

Finite State Machine (FSM)

- Model of computation consisting of
  - A set (of finite number) of states
  - An initial state
  - Input symbols
  - Transition function that maps input symbols and current states to a next state.



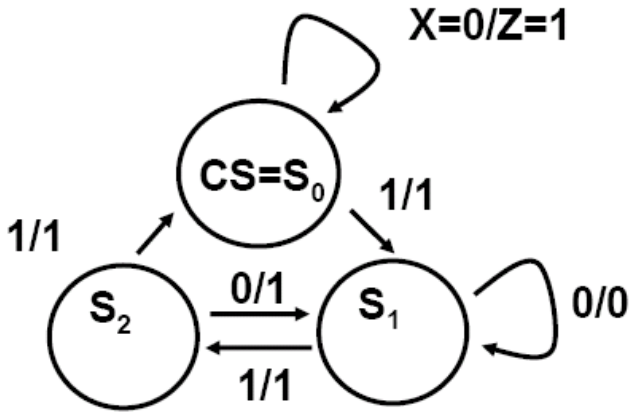
State transition diagram

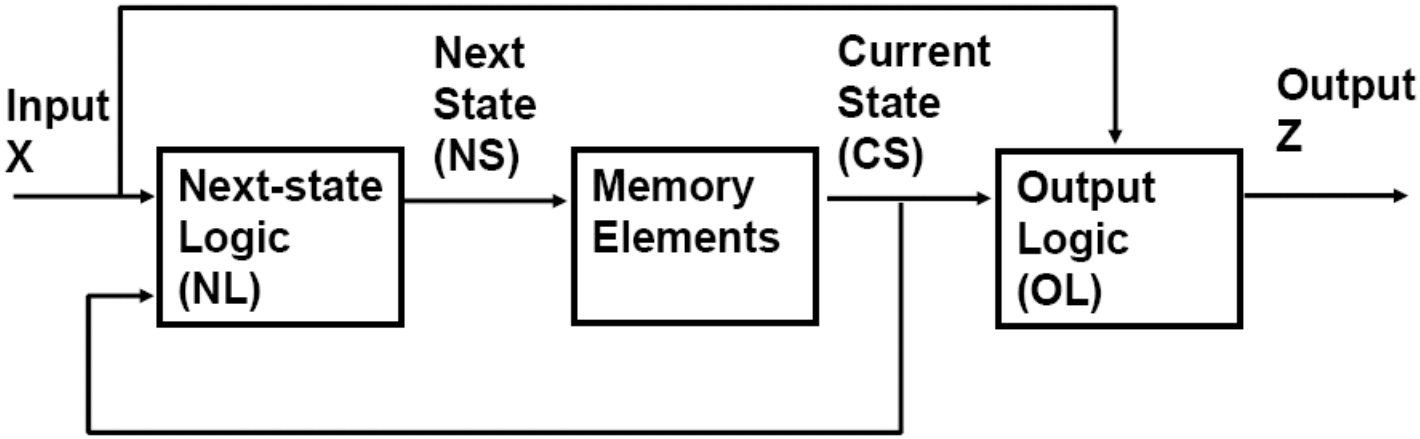
Elements of FSM

- Memory Elements
  - Memorize Current States (CS)
  - Usually consist of FF or latch
  - N-bit FF have 2^n possible states
- Next-state Logic (NL)
  - Combinational Logic
  - Produce next state
  - Based on current state (CS) and input (X)
- Output Logic (OL)
  - Combinational Logic
  - Produce outputs (Z)
    - Based on current state
    - Based on current state and input

Mealy Machine

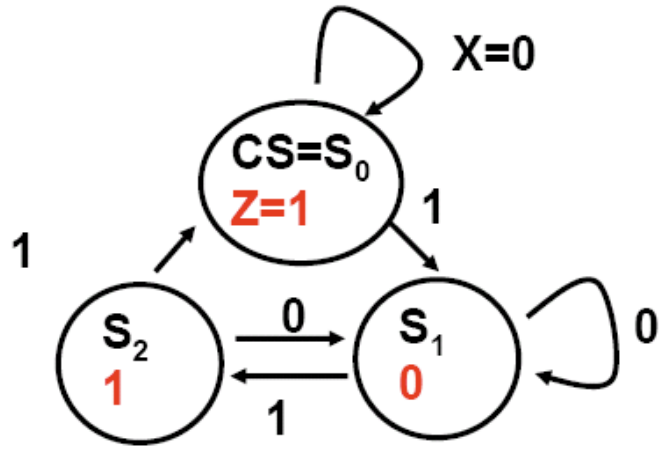
- Output is a function of
  - **both current state & input**

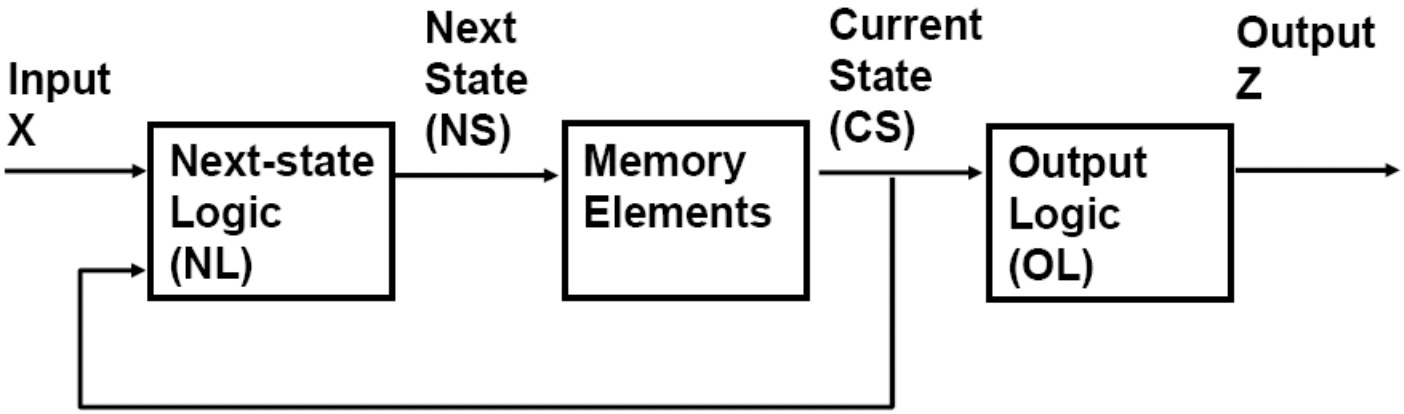




Moore Machine

- Output is a function of
  - **only current state**





Behavior Modeling of FSM

- Combinational Part
  - Next-state logic (NL)
  - Output logic (OL)
- Sequential Part
  - Current state (CS) stored in flip-flops
- 3 Coding Style
  - Separate CS, OL and NL
  - Combine NL+ OL, separate CS
  - Combine CS + NL, separate OL

Coding Style 1: Separate CS, OL and NL

- CS

```
always @ (posedge clk)
        current_state <= next_state;
```

- NL

```
always @ (current_state or In)
  case (current_state)
    S0: case (In)
              In0:  next_state = S1;
              In1:  next_state = S0;
              . . .
         endcase //In
    S1: . . .
    S2: . . .
 endcase //current_state
```

- OL

```
// if Moore
always @ (current_state)
        Z = output_value;
```

```
// if Mealy
always @ (current_state or In)
        Z = output_value;
```

Coding Style 2: Combine NL+ OL, Separate CS

- CS

```
always @ (posedge clk)
        current_state <= next_state;
```

- NL+OL

```
always @ (current_state or In)
  case (current_state)
    S0: beging
          case (In)
            In0:  begin
                    next_state = S1;
                    Z =values; // Mealy
                  end
            In1:  . . .
          endcase // In
          Z =values; // Moore
        end //S0
    S1: . . .
 endcase // current_state
```

Coding Style 3: Combine CS+NL, Separate OL

- CS+NL

```
always @ (posedge clk)
  begin
    case (state)
      S0: case (In)
            In0:   state<= S1;
            In1:   state<= S0;
            . . .
          endcase //In
      S1: . . .
    endcase //state
  end
```

Do not mix comb and seq in one always block

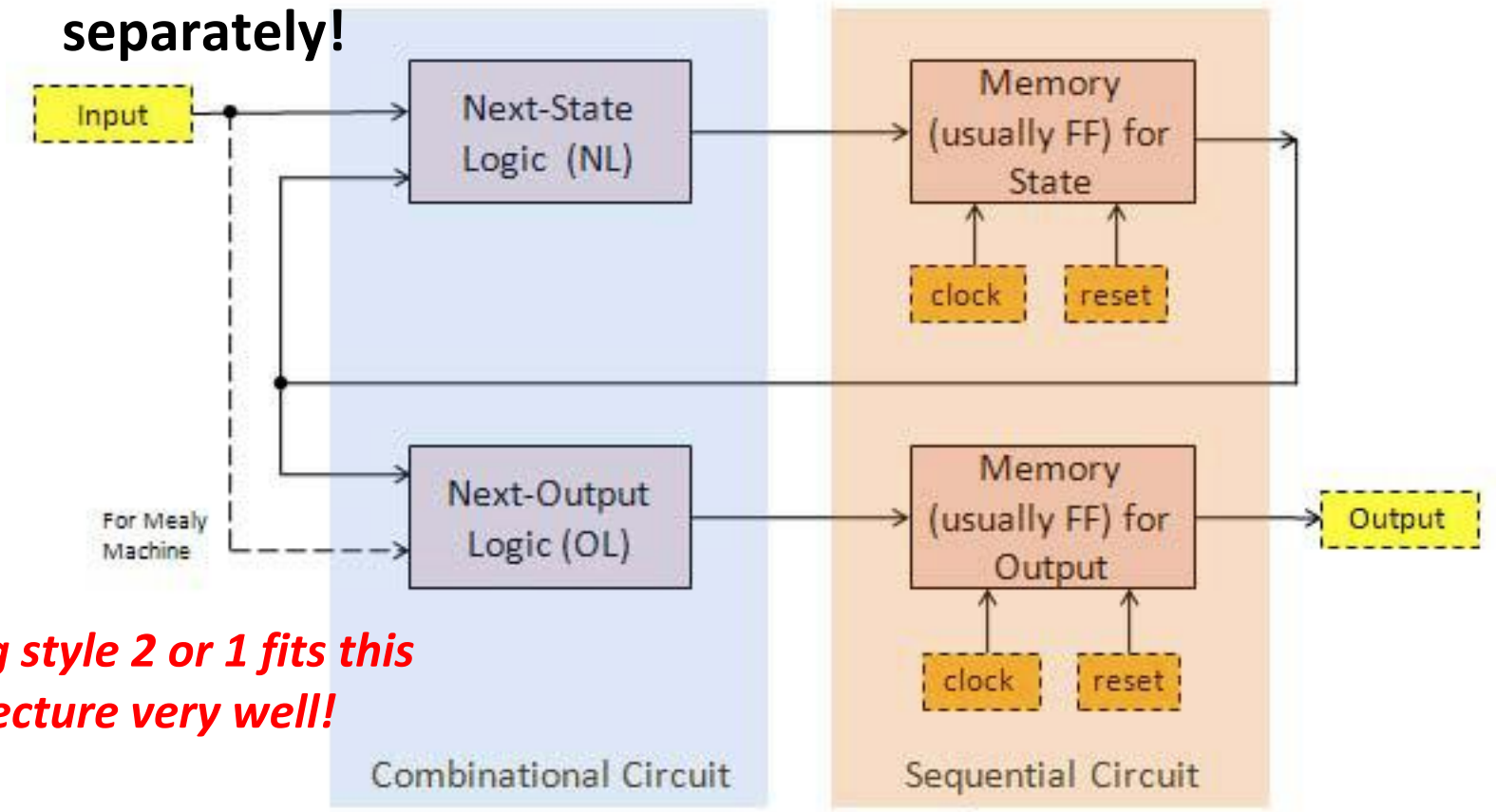
- OL

```
// if Moore
always @ (state)
      Z = output_value;
```

```
// if Mealy
always @ (state or In)
      Z = output_value;
```

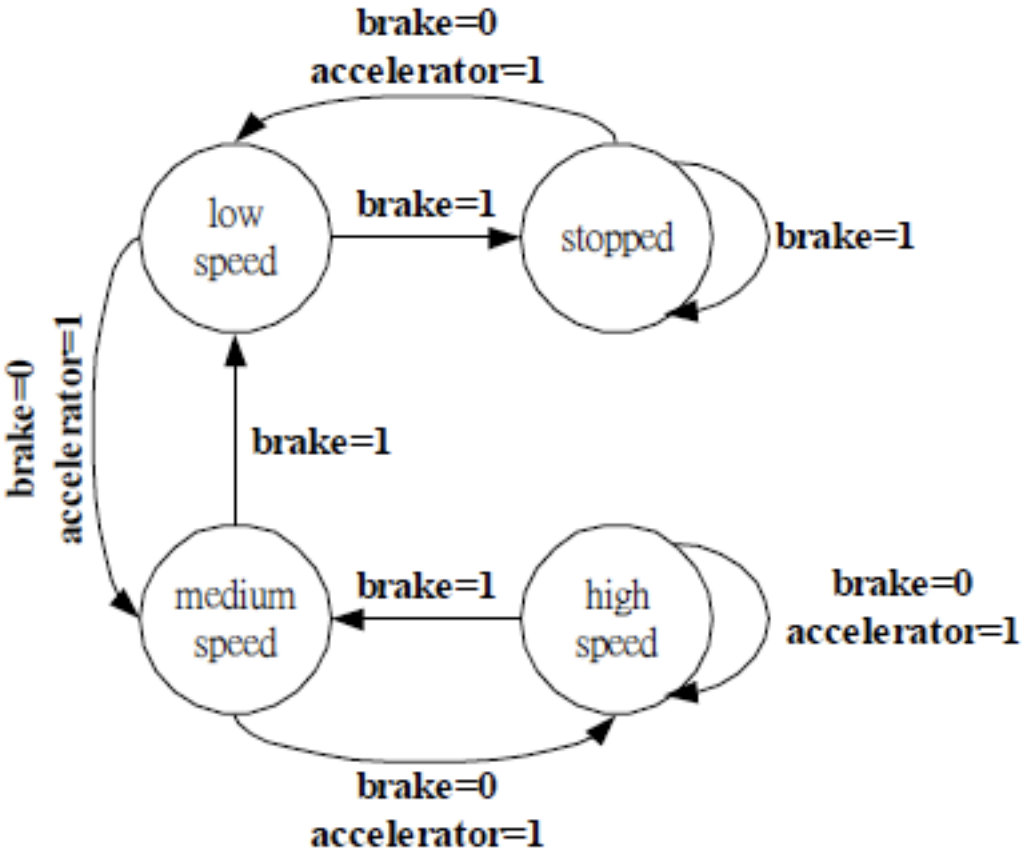
Architecture of FSM

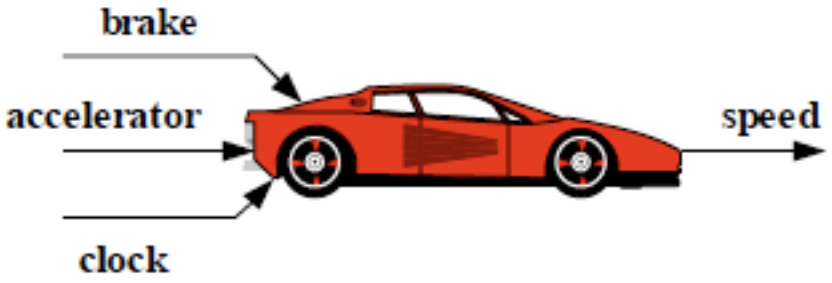
Build combinational and sequential parts separately!



Coding style 2 or 1 fits this architecture very well!

FSM Example: Speed Machine





FSM Example: Reference Code Using Coding Style 1

```
1  module speed_machine (
2      clock,     // system clock
3      reset,     // high-active asynchronous reset
4      accelerator,      // input: accelarator signal
5      brake,            // input: brake signal
6      speed             // output: current speed
7  );
8
9  //==== PARAMETER DEFINITION ====================
10     // using sequential code for state encoding
11     parameter stopped  = 2'b00;
12     parameter s_low    = 2'b01;
13     parameter s_medium = 2'b10;
14     parameter s_high   = 2'b11;
15
16 //==== IN/OUT DECLARATION ======================
17     input        clock, reset;
18     input        accelerator, brake;
19     output [1:0] speed;
20
21 //==== REG/WIRE DECLARATION ====================
22     //--- wires ---
23     reg  [1:0] next_state;
24     wire [1:0] next_speed;
25
26     //--- flip-flops ---
27     reg [1:0] state;     // memory for current state
28     reg [1:0] speed;     // memory for current output
29
30 //==== COMBINATIONAL CIRCUIT ====================
31     //--- next-output logic (OL) ---
32     assign next_speed = state;
33
34     //--- next-state logic (NL) ---
35     always@( state or accelerator or brake ) begin
36         if( brake ) begin
37             case( state )
38                 stopped: next_state = stopped;
39                 s_low:   next_state = stopped;
40                 s_medium:next_state = s_low;
41                 s_high:  next_state = s_medium;
42                 default: next_state = stopped;
43             endcase
44         end
45         else if( accelerator ) begin
46             case( state )
47                 stopped: next_state = s_low;
48                 s_low:   next_state = s_medium;
49                 s_medium:next_state = s_high;
50                 s_high:  next_state = s_high;
51                 default: next_state = stopped;
52             endcase
53         end
54         else next_state = state;
55     end
56
57 //==== SEQUENTIAL CIRCUIT ======================
58     //--- memory elements ---
59     always@( posedge clock or posedge reset ) begin
60         if( reset ) begin
61            state <= 2'd0;
62            speed <= 2'd0;
63         end
64         else begin
65            state <= next_state;
66            speed <= next_speed;
67         end
68     end
69 endmodule
```

FSM Design Notice

- Partition FSM and non-FSM logic
- Partition combinational part and sequential part
  - Coding style 1, 2 are preferred
  - For beginner, Do not use Coding style 3
- Use parameter to define names of the state vector
- Assign a default (reset) state

Computer-Aided VLSI System Design

Chap.2-3 Simulation & Verification

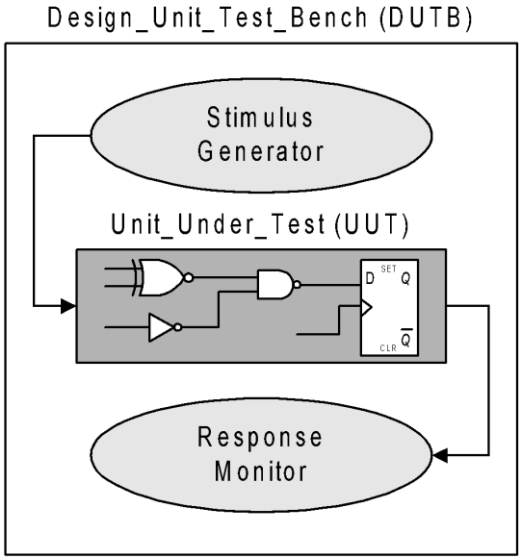
Lecturer: 張惇宥

Graduate Institute of Electronics Engineering, National Taiwan University

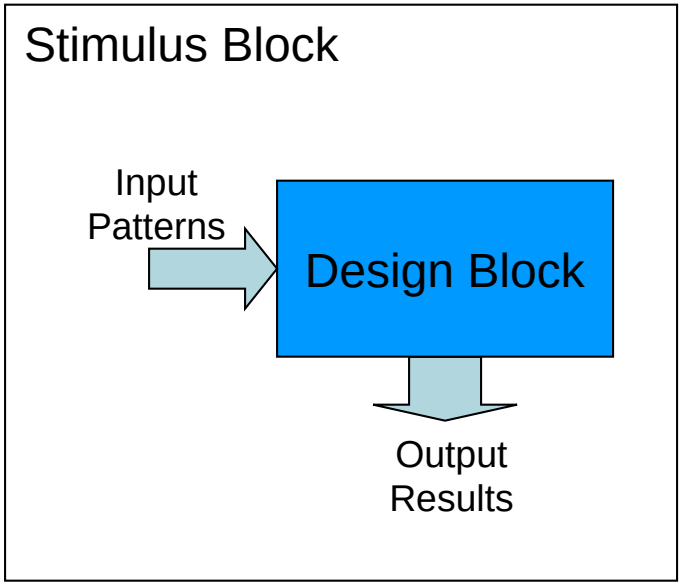
NTU GIEE

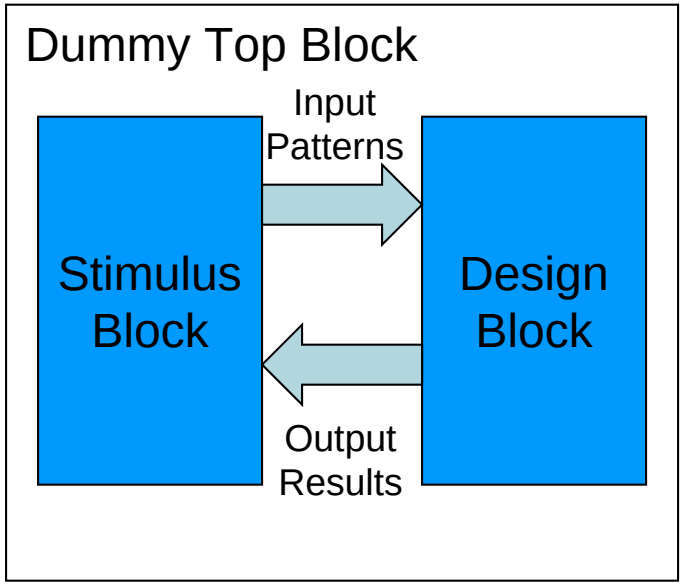
Verification Methodology

- Task: systematically verify the functionality of a model.
- Approaches: Simulation and/or formal verification
- Simulation:
  (1) detect syntax violations in source code
  (2) simulate behavior
  (3) monitor results



Components of a Simulation

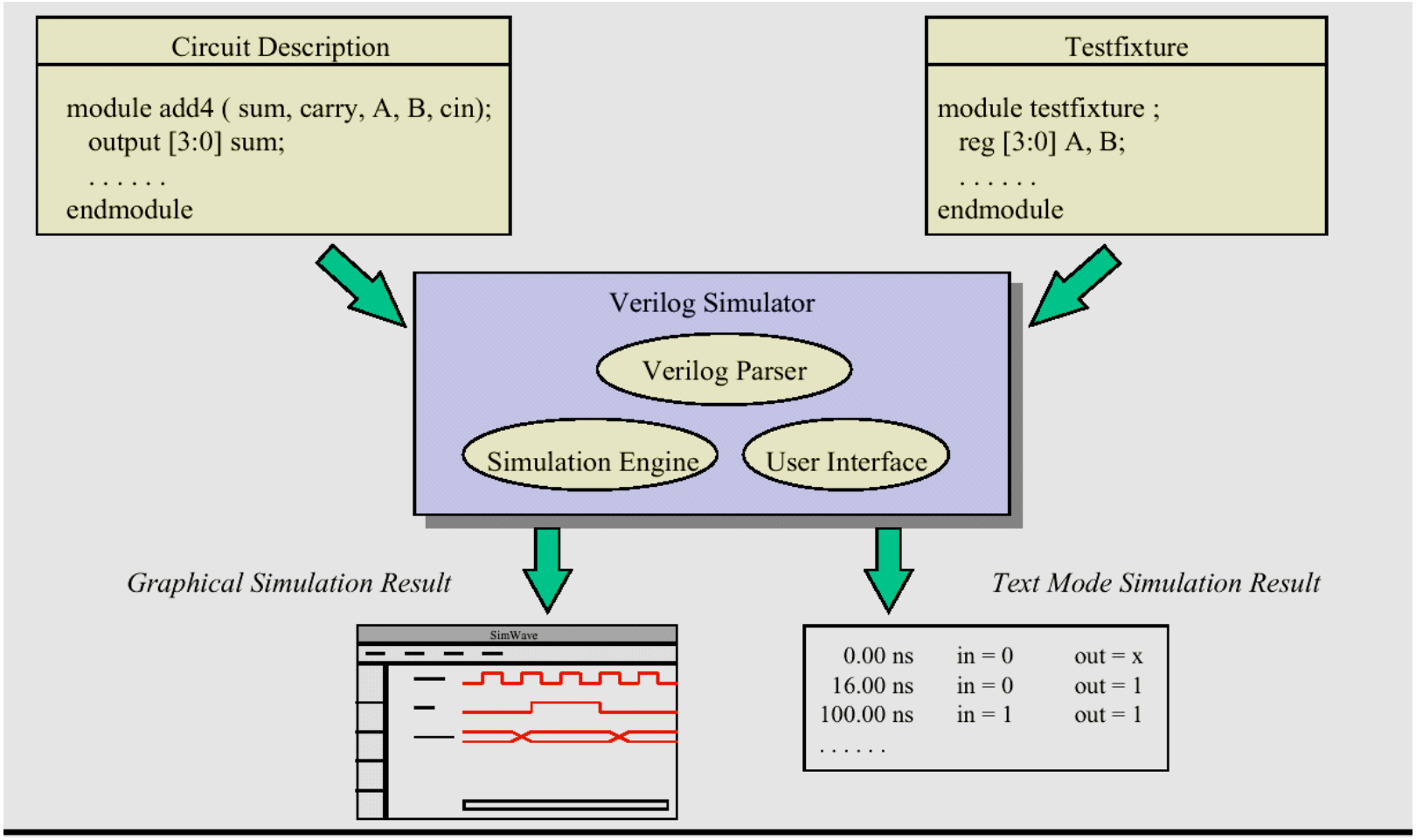




The output results are verified by console/waveform viewer

The output results are verified by testbench or stimulus block

Verilog Simulator



Testbench Template

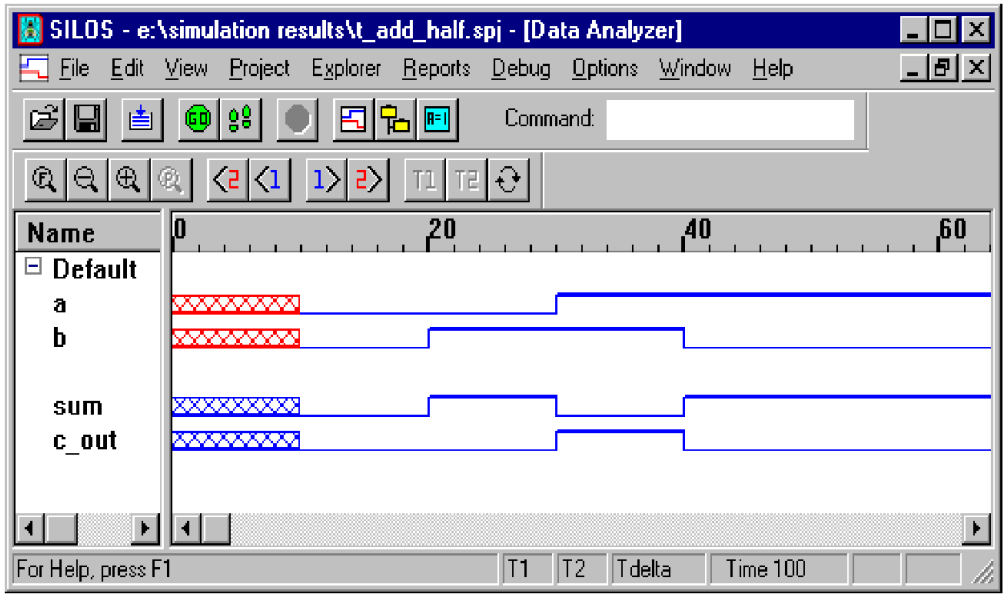
- Consider the following template as a guide for simple testbenches:

```
module t_DUTB_name ();      // substitute the name of the UUT
  reg …;                    // Declaration of register variables for primary inputs of the UUT
  wire …;                   // Declaration of primary outputs of the UUT
  parameter      time_out = // Provide a value

  UUT_name M1_instance_name ( UUT ports go here);

  initial $monitor (  );    // Specification of signals to be monitored and displayed as text

  initial #time_out $stop; // (Also $finish) Stopwatch to assure termination of simulation

  initial                   // Develop one or more behaviors for pattern generation and/or
                            // error detection
    begin
                            // Behavioral statements generating waveforms
                            // to the input ports, and comments documenting the test.
                            // Use the full repertoire of behavioral
                            // constructs for loops and conditionals.
    end
endmodule
```

Example: Testbench

```
`timescale 1ns/10ps

module t_Add_half();
  wire          sum, c_out;
  reg           a, b;                       // Variable for waveforms

  Add_half_0_delay M1 (sum, c_out, a, b); // UUT

  initial begin                             // Time Out
    #100 $finish;                           // Stopwatch
  end

  initial begin                             // Stimulus patterns
    #10 a = 0; b = 0;                       // Statements execute in sequence
    #10 b = 1;
    #10 a = 1;
    #10 b = 0;
  end
endmodule
```

Simulation Results



MODELING TIP

A Verilog simulator assigns an *initial* value of **x** to all variables.

System Tasks and Function: $ (1/2)

- Displaying information
  - *$display*("ID of the port is %b", port_id);
    - ID of the port is 00101
- Monitoring information
  - *$monitor*($time, "Value of signals clk = %b rst = %b", clk, rst);

```
0 Value of signals clk = 0 rst = 1
5 Value of signals clk = 1 rst = 1
10 Value of signals clk = 0 rst = 0
```

- Stopping and finishing in a simulation
  - *$stop*; // provided to stop during a simulation
  - *$finish*; // terminates the simulator

System Tasks and Function: $ (2/2)

- Math functions
  - *$clog2*(<arg>);
    - the ceiling of the log base 2 of the argument

```
module ram_model (address, write, chip_select, data);
parameter ram_depth = 256;
localparam addr_width = $clog2(ram_depth);
input [addr_width - 1:0] address;
input write, chip_select;
```

- Probabilistic distribution functions
  - *$random*;
    - 32-bit random signed integer
- Conversion functions
  - *$rtoi*(<real_value>); // convert real value to integer
  - *$itor*(<integer>); // convert integer to real value

Compiler Directives: `

- ***`define***
  - `define RAM_SIZE 16
  - Defining a name and gives a constant value to it.
  - the identifier `RAM_SIZE will be replaced by 16
- ***`include***
  - `include adder.v
  - Including the entire contents of another Verilog source file.
- ***`timescale***
  - `timescale 1ns/10ps
  - `timescale *<reference_time_unit>* / *<time_precision>*
  - Setting the reference time unit and time precision of your simulation.

Example: Error Calculation in TB

- Calculate the percentage error of an L-dim 16-bit array

```
`define diff_abs(a, b) (a-b)>0 ? a-b : b-a
module test;
parameter array_len = L;
integer i;
real total_error = 0;
real error_percentage;

reg [15:0] mem [0:array_len-1];
real golden [0:array_len-1];

initial begin
    for (i=0; i<array_len; i=i+1) begin
        total_error = total_error
           + (`diff_abs($itor(mem[i]), golden[i]))/golden[i];
    end
    error_percentage = total_error*100/$itor(array_len);
    $display("error percentage: %f %%", error_percentage );
end
endmodule
```

Simulation Schemes

- There are 3 categories of simulation schemes
  - Time-based: Simulation on real time scale, used by SPICE simulators
  - **Event-based**: Simulation on events of signal transition, used by Verilog simulators. Note each event must occurs on discrete time specified by the testbench.
  - Cycle-based: Used by system/platform level verification, less used in cell-based IC designing.

Case Study: Bubble Sort

- The software style RTL code here is poor writing
  1. The number in termination condition should be constant
  2. The default condition should be added
  3. It is better to use more cycles

```
reg [15:0] mem [0:15];
reg [15:0] tmp;

integer i, j;
always @(*) begin
    for (i=0; i<16; i=i+1) begin
        for (j=0; j<16-i-1; j=j+1) begin
            if (mem[j] > mem[j+1]) begin
                tmp = mem[j];
                mem[j] = mem[j+1];
                mem[j+1] = tmp;
            end
        end
    end
end
```

References

- TSRI Verilog 上課講義
- IEEE Standard for Verilog Hardware Description Language
- Verilog 教學網站