Computer-Aided VLSI System Design

Chap.2-1 Logic Design at Register-Transfer Level

Lecturer: 徐以帆

Graduate Institute of Electronics Engineering, National Taiwan University

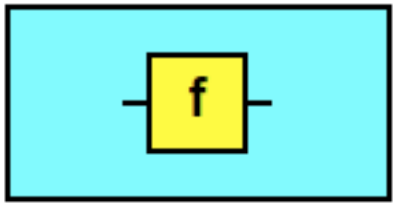
NTU GIEE

Outline

- Introduction to Register-Transfer Level (RTL)
- Continuous Assignments
- Procedural Assignments
- Modeling Flip-Flops

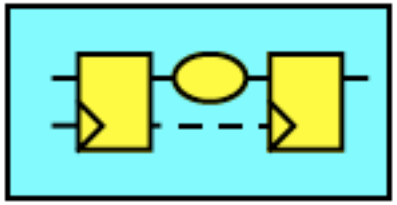
Brief Review of Modeling Levels

Behavioral Level



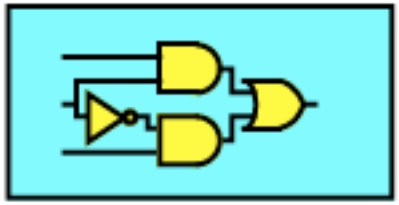
Most used modeling level, easy for designer, can be simulated and synthesized to gate-level by EDA tools

Register Transfer Level (RTL)



Common used modeling level for small sub-modules, can be simulated and synthesized to gate-level

Structural/Gate Level



Usually generated by synthesis tool by using a front-end cell library, can be simulated by EDA tools. A gate is mapped to a cell in library

Transistor/Physical Level

Usually generated by synthesis tool by using a back-end cell library, can be simulated by SPICE

What is Register Transfer Level?

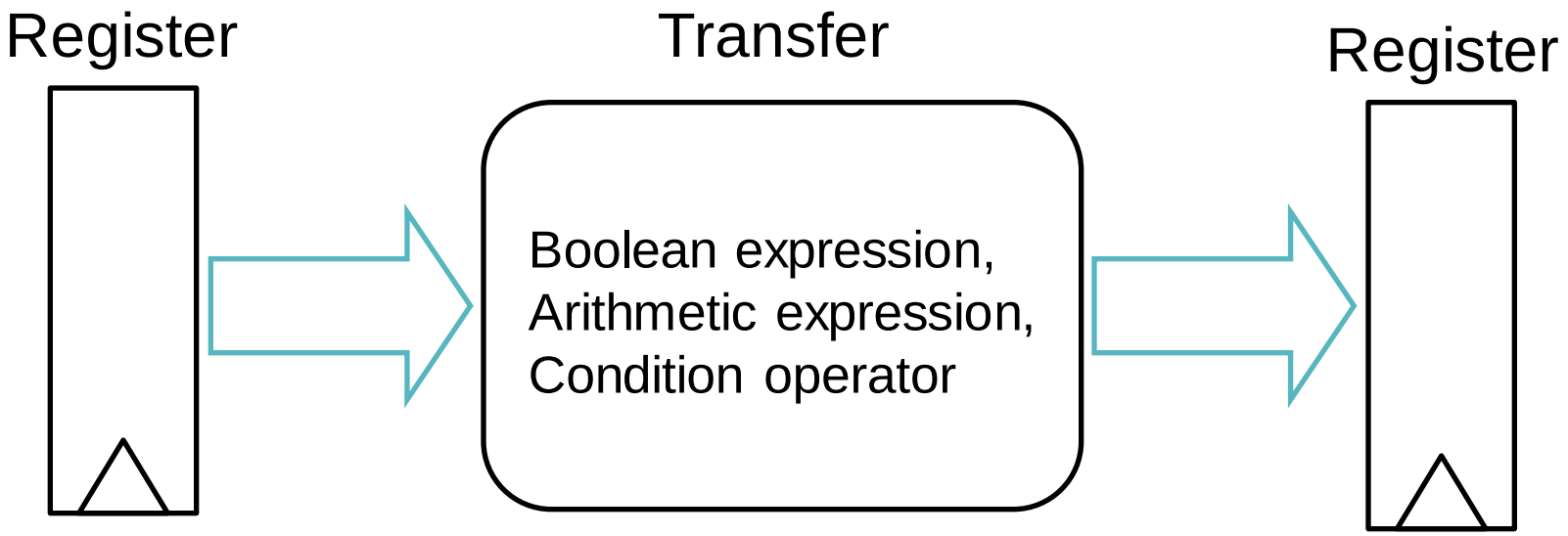
- In integrated circuit design, Register Transfer Level (RTL) description is a way of describing the operation of a synchronous digital circuit.

- In RTL design, a circuit's behavior is defined in terms of the flow of signals (or transfer of data) between synchronous registers and the logical operations performed on those signals.

- Modern RTL code: "Any code that is synthesizable is called RTL code"

Description at RT-Level

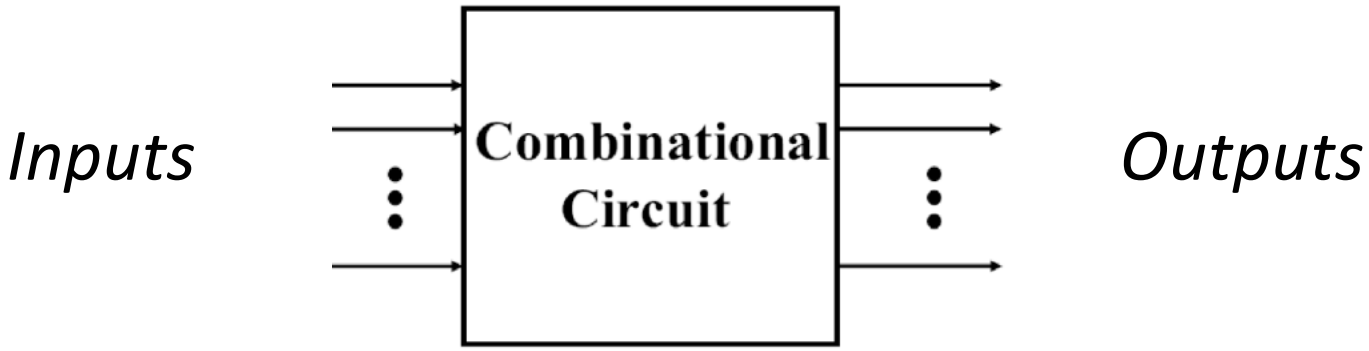
- Register (memory, usually D flip-flops, **sequential**)
- Transfer (operation, **combinational**)



Combinational Data Path

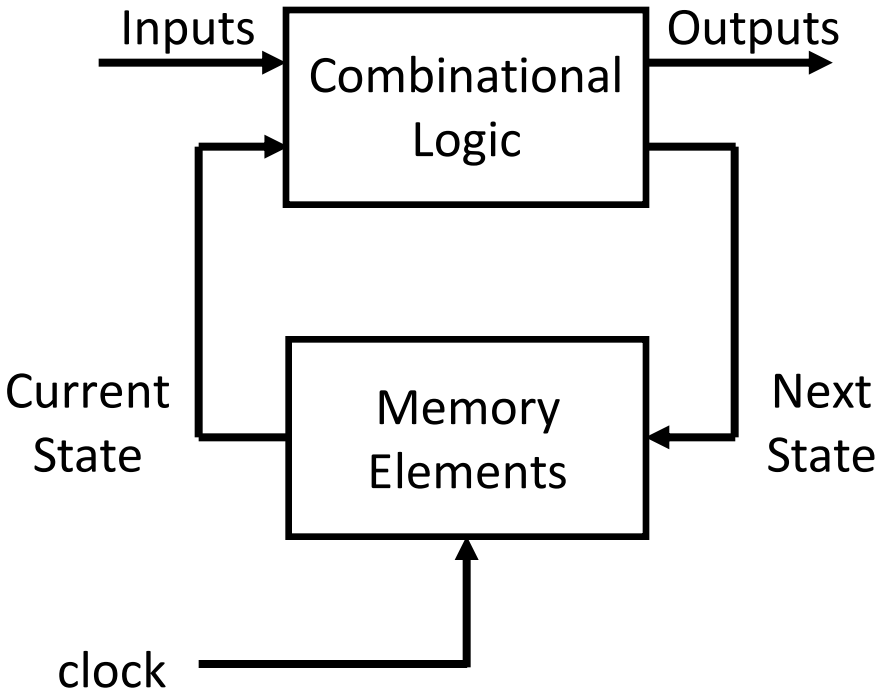
- Combinational logic circuits are **memoryless**
  - Any transition in inputs affect the whole circuit right away
  - Feedback path is not allowed

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



Sequential Data Path

- Sequential circuits have memory (i.e. remember the past)
- The output depends on
  - Current state
  - Inputs (optional)
- Fundamental components
  - Combinational circuits
  - Memory elements
- Synchronous system
  - Updates are triggered by a global clock



Steps to Design at RT-Level

- Design partition
  - Combinational circuit (transfer part)
  - Sequential circuit (register part)
- Net declaration
  - I/O ports
  - Net variable
- Transfer Circuit design
  - Logical operator
  - Arithmetical operator
  - Conditional operator

Outline

- Introduction to Register-Transfer Level (RTL)
- Continuous Assignments
- Procedural Assignments
- Modeling Flip-Flops

Assignments (1/2)

- Assignment: Drive value onto nets and variables
- Two basic forms of assignment
  - **Continuous assignment** assigns values to nets
  - **Procedural assignment** assigns values to variables
- Basic form

```
<left-hand side> = <right-hand side>
```

Statement type	Left-hand side (LHS)
Continuous assignment	Net* `wire, tri`
Procedural assignment	Variable* `reg, integer, real`

* Include scalar, vector, concatenation, and bit/part-select

Assignments (2/2)

- Continuous assignment

```
module holiday_1(sat, sun, weekend);
    input sat, sun; output weekend;
    assign weekend = sat | sun;
endmodule
```

- Procedural assignment

```
module holiday_2(sat, sun, weekend);
    input sat, sun; output weekend; reg weekend;
    always @(*) weekend = sat | sun;
endmodule
```

```
module assignments;
    // continuous assignments go here
    always begin
        // procedural assignments go here
    end
endmodule
```

Continuous Assignments (1/3)

- Syntax of continuous assignment

```
assign [#<delay>] <net name> = <expression>;
```

- The continuous assignment is **always active**
  - Any changes in the RHS of the continuous assignment are evaluated and the LHS is updated
- The LHS shall be
  - **Scalar or vector net**
  - Bit/part-select of vector net
  - Concatenation of any of the above type

```
assign {cout, sum[3:0]} = a[3:0] + b[3:0] + cin;
```

- The RHS is not restricted

Continuous Assignments (2/3)

- Syntax of continuous assignment

```
assign [#<delay>] <net name> = <expression>;
```

- The LHS is updated at any change in the RHS expression after a specified delay
  - The delay is optional
  - The continuous assignment **with delay is non-synthesizable**
  - Continuous assignments can only contain simple, left-hand side delay (i.e., limited to a **# delay**)
  - **@** timing control is unnecessary because of their continuous nature
- Example

```
assign #10 o = in1 + in2
```

Continuous Assignments (3/3)

- The continuous assignment is **always active**

```
module assignment_1();
  wire pwr_good, pwr_on, pwr_stable; reg Ok, Fire;

  assign pwr_stable = Ok & (!Fire);
  assign pwr_on = 1;
  assign pwr_good = pwr_on & pwr_stable;

  initial begin Ok = 0; Fire = 0; #1 Ok = 1; #5 Fire = 1; end
  initial begin
    $monitor("TIME=%0d",$time," ON=",pwr_on," STABLE=",pwr_stable,
             " OK=",Ok," FIRE=",Fire," GOOD=",pwr_good);
    #10 $finish;
  end
endmodule
```

```
> TIME=0 ON=1 STABLE=0 OK=0 FIRE=0 GOOD=0
> TIME=1 ON=1 STABLE=1 OK=1 FIRE=0 GOOD=1
> TIME=6 ON=1 STABLE=0 OK=1 FIRE=1 GOOD=0
```

Example of Continuous Assignment

- Example of continuous assignment

```
wire [15:0] addr;  wire [3:0] sum;  wire out, cout;

// net declaration assignment, a_low[2:0] = a[2:0]
wire [2:0] a_low = a[3:0];

// i1 and i2 are nets
assign out = i1 & i2;

// Continuous assign for vector nets, addr is a 16-b
// vector net, addr1 and addr2 are 16-b vector variables
assign addr[15:0] = addr1[15:0] ^ addr2[15:0];

// LHS is a concatenation of a scalar net and vector net
assign {cout, sum[3:0]} = a[3:0] + b[3:0] + cin;
```

Continuous Assignment Delays

- Regular assignment delay

```
wire out;
assign #10 out = in1 & in2;
```

- Implicit continuous assignment delay

```
wire #10 out = in1 & in2;
```

- Net declaration delay

```
wire #10 out; // net delays
assign out = in1 & in2;
```

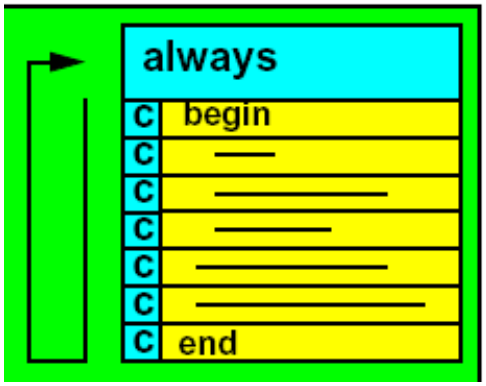
Non-synthesizable! For Testbench/Behavior model usage only.

Outline

- Introduction to Register-Transfer Level (RTL)
- Continuous Assignments
- Procedural Assignments
- Modeling Flip-Flops

Preview: Procedures

- Procedures are structures mainly used for behavioral modeling, and also sequential circuit modeling
  - **`initial`**, **`always`**, **`task`**, **`function`**
- All procedures execute **concurrently**
- Only variables (**`reg`**) can be LHS in an `always` construct
  - RHS is not restricted
- **`begin`** and **`end`** are used to frame a procedural block



Preview: Always Construct

- **always** blocks are essentially infinite loops
- Creates a simulation deadlock if no timing control
- Sensitivity list is used to control when the always block will be executed

```
always @(<events>) begin … end
```

- For combinational circuits, list all signals read in the **always** block, or just a wildcard character **@(*)**

```
always @(in1 or in2 …)
```

(recommended)

```
always @(*)
```

- For sequential circuits, only include the clock edge event

```
always @(posedge clock)
```

```
always @(negedge clock)
```

Procedural Assignment (1/2)

- Syntax of procedural assignment

```
<start of procedural block>
    [#<delay>] <variable name> = <expression>;
<end of procedural block>
```

- The LHS shall be
  - **Scalar or vector variable (e.g. reg)**
  - Bit/part-select of vector variable
  - Concatenation of any of the above type

```
always @(*) begin
    {cout, sum[3:0]} = a[3:0] + b[3:0] + cin;
end
```

- The RHS is not restricted

Procedural Assignment (2/2)

- Syntax of procedural assignment

```
<start of procedural block>
    [#<delay>] <variable name> = <expression>;
<end of procedural block>
```

- Signals without specification 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

- Blocking assignment (=)
  - Evaluate the RHS, and update the LHS immediately
  - Suitable for modeling **combinational** circuits
  - **Difficult to model the concurrency**

- Non-blocking assignment (<=)
  - Evaluate the RHS, but schedule the update of LHS
  - Update LHS only after evaluate all RHS
  - Suitable for modeling **sequential** circuits (concurrency)

Blocking or Non-Blocking?

- Blocking assignment
  - Evaluation and assignment are immediate

```
// cntr = 2 -> 3,  x = 7,  y = 5
always @(cntr) begin
    x = cntr + 1;        // Evaluate to 3 + 1, assign x = 4
    y = x + 1;           // Evaluate to 4 + 1, assign y = 5
end
```

- Non-blocking assignment
  - All assignments deferred until all RHS have been evaluated (end of the virtual timestep)

```
// cntr = 2 -> 3,  x = 7,  y = 5
always @(cntr) begin
    x <= cntr + 1;       // Evaluate to 3 + 1, defer assignment
    y <= x + 1;          // Evaluate to 7 + 1, defer assignment
end                      // assign x = 4,  y = 8
```

Example of Procedural Assignment

- Example of procedural assignment

```
reg [15:0] addr;  reg [3:0] sum;  reg out, cout;

// net declaration assignment, a_low[2:0] = a[2:0]
reg [2:0] a_low = a[3:0];

always @(*) begin
    out = i1 & i2;    // i1 and i2 are nets

    // Procedural assign for vector reg, addr is a 16-b vector
    // reg, addr1 and addr2 are 16-b vector reg
    addr[15:0] = addr1[15:0] ^ addr2[15:0];

    // LHS is a concatenation of a scalar reg and vector reg
    {cout, sum[3:0]} = a[3:0] + b[3:0] + cin;
end
```

Procedural Assignment Delays

- Regular assignment delay
  - Delay both the evaluation and assignment of the statement

```
reg out;
always @(*) begin
  #10 out = in1 & in2;
end
```

- Intra-assignment delay
  - Evaluate RHS immediately, delay the assignment of LHS

```
reg out;
always @(*) begin
  out = #10 in1 & in2;
end
```

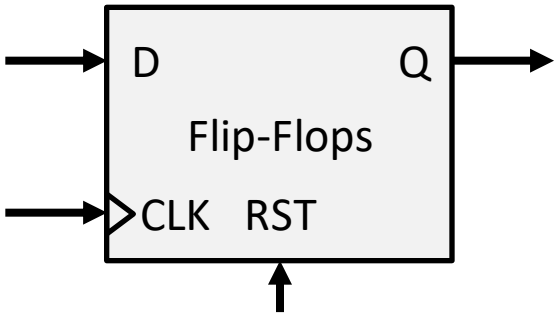
Non-synthesizable! For Testbench/Behavior model usage only.

Outline

- Introduction to Register-Transfer Level (RTL)
- Continuous Assignments
- Procedural Assignments
- Modeling Flip-Flops

Flip-Flops

- Flips-flops are the memory elements in digital circuits



- Trigging edge
  - Positive: sample data at rising clock edge
  - Negative: sample data at falling clock edge
- Reset scheme
  - Synchronous: reset at trigging clock edge
  - Asynchronous: reset at RST edge

Modeling Flip-Flops (1/2)

- Use of **posedge** and **negedge** makes an **always** block sequential (edge-triggered)
- Content in the sensitivity list determines whether the flip-flop is synchronous or asynchronous

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
```

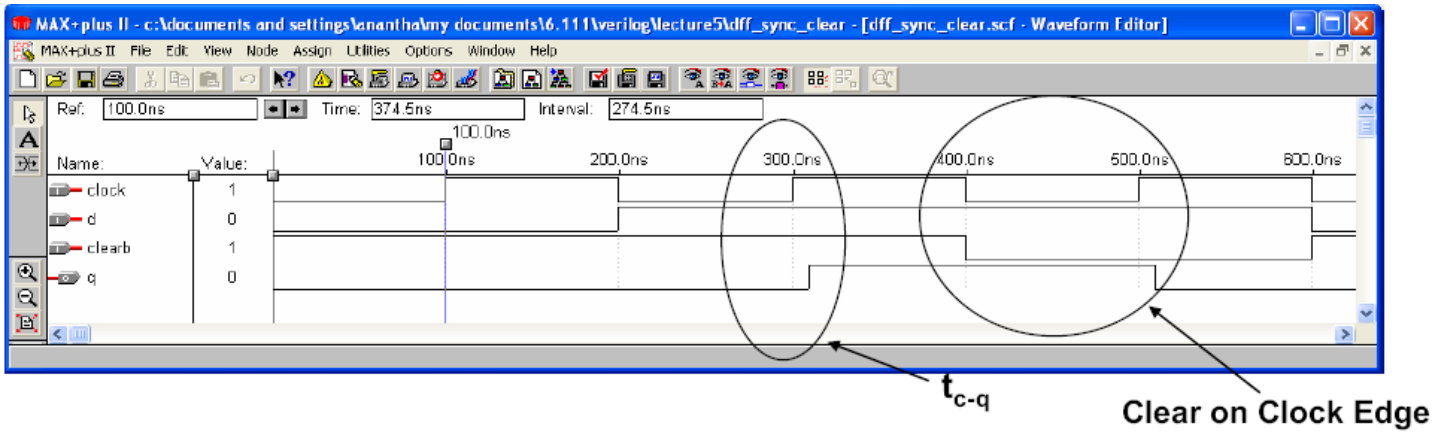
always block entered immediately when (active-low) clearb is asserted

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

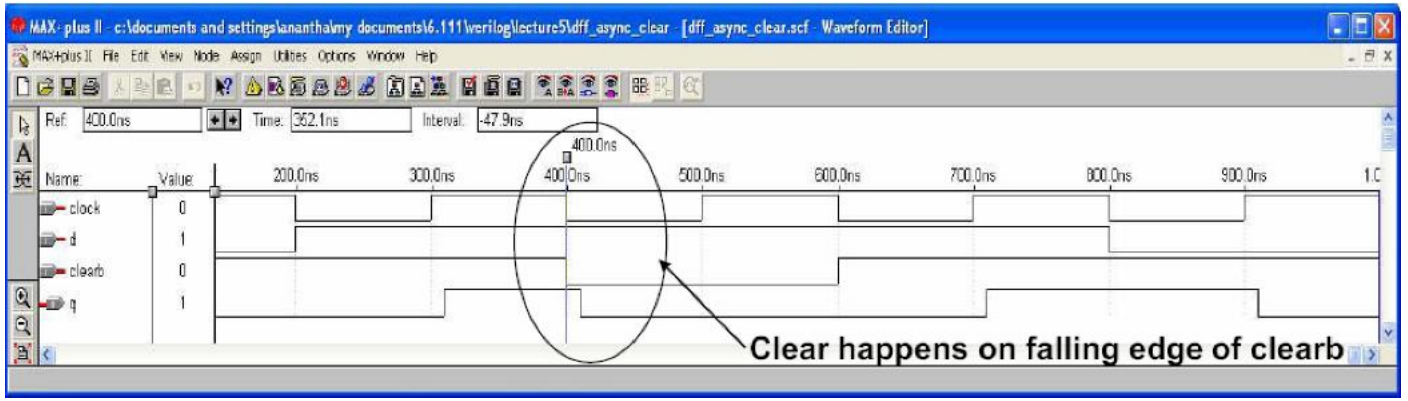
If one signal in the sensitivity list uses posedge/negedge, then all signals must. (for synthesis)

Modeling Flip-Flops (2/2)

- Synchronous Reset



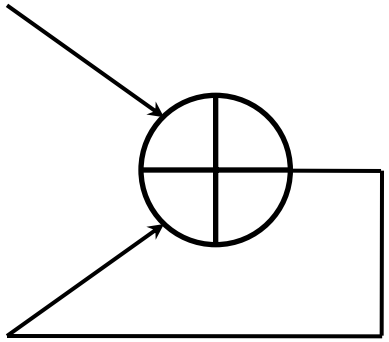
- Asynchronous Reset



Avoiding Combinational Loops

- Avoid combinational loops (or logic loops)
  - Synthesis tools will automatically break asynchronous combinational loops
  - Without disabling the combinational feedback loop, the static timing analyzer can't resolve
  - Example

```
wire [3:0] a;
wire [3:0] b;

assign a = b + a;
```



  - Simple solution: Add a layer of flip-flops

Example Design in RT-Level

```
module accumulator (
  input  clk,          input  rst,
  input  [7:0] idata,  output [7:0] odata
);
  // Declaration
  reg [7:0] odata_r, odata_w;

  // Transfer
  assign odata = odata_r;                  // continuous assign.
  always @(*) odata_w = odata_r + idata;   // procedural assign.

  // Register
  always @(posedge clk or posedge rst) begin  // async reset
    odata_r <= reset ? 8'b0 : odata_w;
  end
endmodule
```

Computer-Aided VLSI System Design

Chap.2-2 Logic Design at Behavioral Level

Graduate Institute of Electronics Engineering, National Taiwan University

NTU GIEE

Outline

- Introduction to Behavioral Level
- Procedural Construct
- Timing Control
- Procedural Statements
- Functional Partition
- Finite State Machine (FSM)

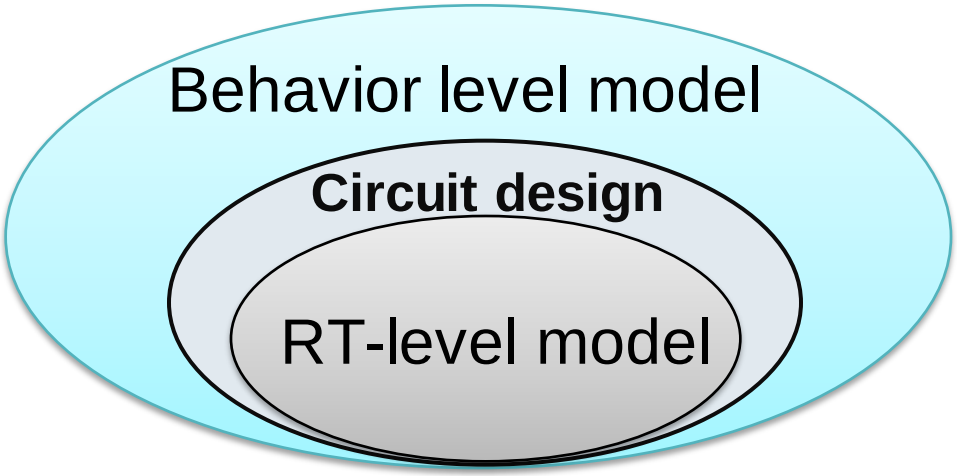
What is Behavioral Level

- Model circuits using "behavioral" descriptions and events
- Description of an RTL model
  - Structure: separated for combinational and sequential circuits
  - Signal: continuous evaluate-update, pin accurate
  - Timing: cycle accurate
- Description of a behavioral level model
  - Structure: can be combinational, sequential, or hybrid circuits
  - Signal: event-driven, timing, arithmetic (floating point, 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

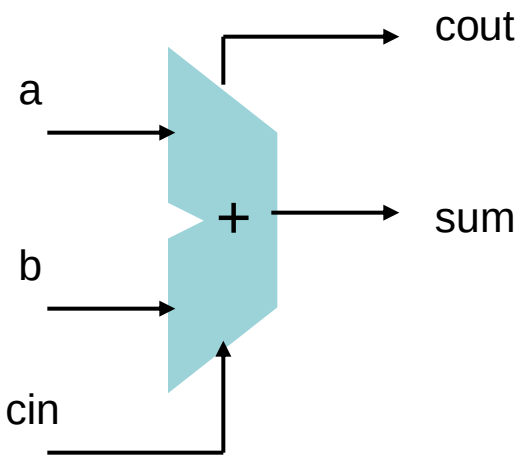
$sum = a \oplus b \oplus ci$

$cout = ab + bc + ca$

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
```



Outline

- Introduction to Behavioral Level
- Procedural Construct
  - **`initial`** Block
  - **`always`** Block
- Timing Control
- Procedural Statements
- Functional Partition
- Finite State Machine (FSM)

Procedures

- Procedures are structures mainly used for behavioral modeling
  - **`initial`** construct
  - **`always`** construct
  - **`task`**
  - **`function`**
- All procedures execute **concurrently**

- Procedural statements wrapped inside a **`begin-end`** block or **`fork-join`** block act as one single statement
  - Similar to { } in C/C++

Syntax Rules of Procedural Constructs

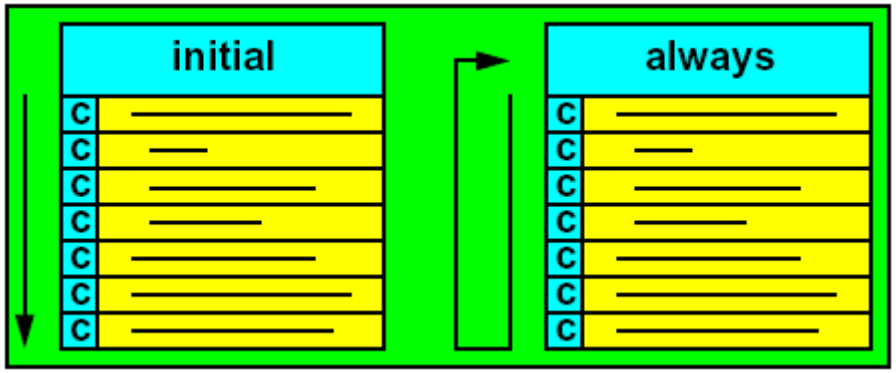
- Two types of constructs
  - **`initial`** construct
  - **`always`** construct
- Only event-driven variables (**`reg`**) can be LHS
  - It's syntax. Not relevant to whether it's a flip-flop or a metal wire!
  - RHS is not restricted
- Event-driven variables are updated by *procedural assignment*
  - **Blocking assignment (=)**
  - **Non-blocking assignment (<=)**

Procedural Blocks (1/3)

- Two kinds of procedural block
  - **`initial`**: execute only once for initialization or waveform generation (**NON**-SYNTHESIZABLE)
  - **`always`**: infinite loop of execution

No circuit can work only once, and then disappear

- The **`initial`** and **`always`** procedural blocks are launched 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
```

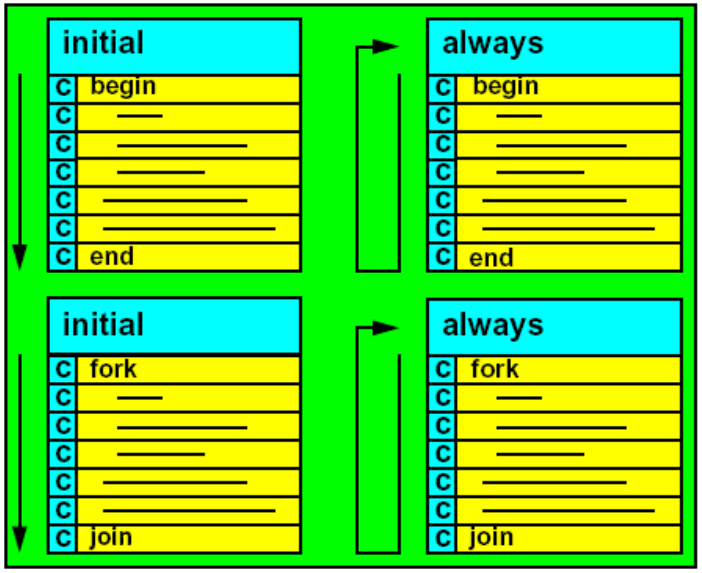
Timing control

High-level construct

Procedural assignment

Procedural Blocks (3/3)

- Sequential block (**begin-end**): execute in sequence
- Concurrent block (**fork-join**): execute in parallel
  (**NON**-SYNTHESIZABLE, not for modeling circuits)
- 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

Outline

- Introduction to Behavioral Level
- Procedural Construct
- Timing Control
  - **# (delay)**
  - **@ (event)**
  - **wait (level)**
- Procedural Statements
- Functional Partition
- Finite State Machine (FSM)

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
   - `@ (<event>)`
3. A level sensitive timing control (**NON**-SYNTHESIZABLE)
   - `wait (<expression>)`

Delay Control

- Syntax: `# (<delay>)`
- Delay the execution by a certain amount of time
- Used to delay stimulus in testbench or to approximate real-world delays in behavioral models
- Delay control 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

```
parameter CYCLE = 10;
reg clk;

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

Event Control (1/3)

- Syntax: **@ (<event>)**
- Wait until the certain level-changing event occurs
- Used to describe edge-triggered circuits (e.g. flip-flops)
- Event **or** operator is used to separate multiple events
- Keywords **posedge** and **negedge** specify the particular type of level-changing event
  - **posedge** and **negedge** only detects LSB

```
wire a;  wire [3:0] b;  reg c;  reg [3:0] d;

@ (a)          $display("detect any change of a");
@ (b or c)     $display("detect any change of b or c");
@ (posedge a) $display("detect rising edge of a");
@ (negedge d) $display("detect falling edge of d[0]");
```

Event Control (2/3)

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

- Syntax: **@ (<event>)**
- Event control following an always is called "sensitivity list"
- If any signals in the list changes, the **always** block will be triggered
- Missing signals in sensitivity list may lead to wrong results!

WRONG!

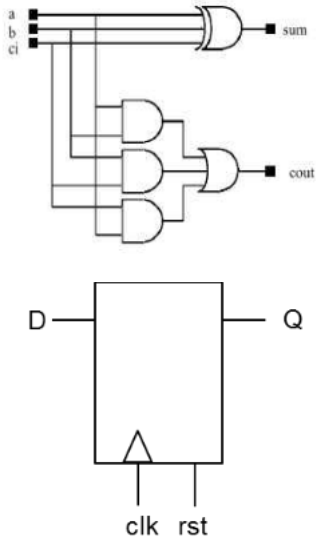
```
// initial a=0, b=0, x=0, y=1
always @(a or b) begin
                    // 1. b 0 → 1
    y = ~x;         // 2. y stay 1
    x = a | b;      // 3. x 0 → 1
end // Expect y=0 but found 1
```

CORRECT!

```
// initial a=0, b=0, x=0, y=1
always @(a or b or x) begin
                    // 1. b 0 → 1
    y = ~x;         // 2. y stay 1
                    //// 4. y 1 → 0
    x = a | b;      // 3. x 0 → 1,
                    //    trigger again!
                    //// 5. x stay 1
end // all signals as expected
```

Event Control (3/3)

- 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

- Syntax: **`wait (<expression>)`**
- Wait until the expression evaluates to ***true*** (level-sensitive)
- Used to detect simulation progress or handshake signals in testbench
- **`wait`** is **NON**-SYNTHESIZABLE
- Example

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

  initial begin
    wait(o_ready);
    // start testing sequence //
    wait(o_finish) $display("Finished");
    // check output data //
    #10 $finish;
  end
endmodule
```

Outline

- Introduction to Behavioral Level
- Procedural Construct
- Timing Control
- Procedural Statements
  - Conditional Statements
  - Loop Statements
- Functional Partition
- Finite State Machine (FSM)

If Statements (1/2)

- Syntax:

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

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

- Be careful not to create a latch (problematic for STA)
  - LHS in all cases should be the same!
  - Conditions should be full-case, must be an "else"
  - In short, think about MUX!

If Statements (2/2)

- Good example

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

(full)

```
a_w = a_r;
B_w = b_r;

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

(with default)

```
always
  @(posedge clock)
    if (sel == 0)
        a <= data;
    else
        b <= data;
```

(D-FF)

- Bad example

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

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

Case Statements (1/2)

- Syntax:

```
case (expression)
    case_item_0: <statement>
    case_item_1: <statement>
    …
    default: <statement>
endcase
```

- Test for (`expression == case_item`)
  - `case_item` can be constants, nets, variables
- Evaluate each case in sequence until a match
- To avoid latches, specify all possible value of expression or include a **`default`** case

Case Statements (2/2)

- Example

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

(not parallel)

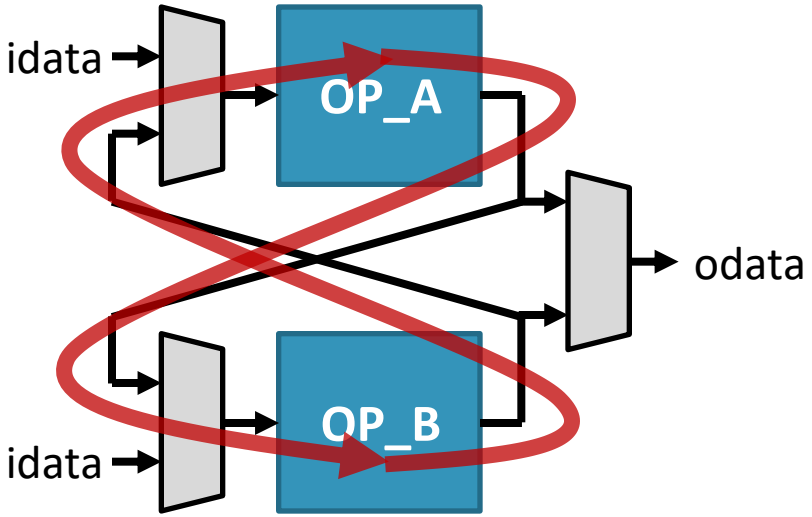
```
case (state_r)
    2'b00: begin
        state_w = 2'b01;
        o_valid = 1'b0;
    end
    2'b01: begin
        state_w = 2'b10;
        o_valid = 1'b1;
    end
    2'b10: begin
        state_w = state_r;
        o_valid = 1'b0;
    end
endcase
```

(not full)

Another Example of Combinational Loops

- Assume two modes with two types of combinational operation in different order
  - Mode 1: OP_A then OP_B
  - Mode 2: OP_B then OP_A

```
case (mode)
  MODE_ONE: begin
    in_a = idata;
    in_b = out_a;
    odata = out_b;
  end
  MODE_TWO: begin
    in_b = idata;
    in_a = out_b;
    odata = out_a;
  end
endcase
```



Looping Statements

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

For Loop (1/2)

- Syntax:

```
for (initial_assignment; expression; update_assignment)
  <statement>
```

- A **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 unrolled form is synthesizable**
  - **Constant iteration bound**
  - **No timing dependency**

For Loop (2/2)

- Example

```
parameter SIZE = 8;
reg [3:0] out [0:SIZE-1];
integer i;
always @(*) begin
    for (i = 0; i < SIZE; i=i+1)
        out[i] = a[i] + b[i] + i;
end
```

```
reg [3:0] num;  // runtime determined
reg [3:0] out [0:15];
integer i;
always @(*) begin
    for (i = 0; i < num; i=i+1)
        out[i] = 0;
end
```

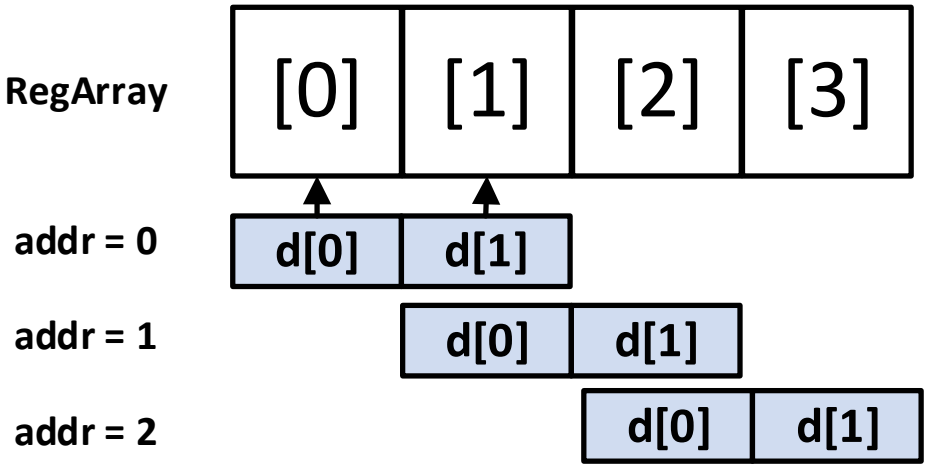
Example: RegFile

- The for-loop can express the register file

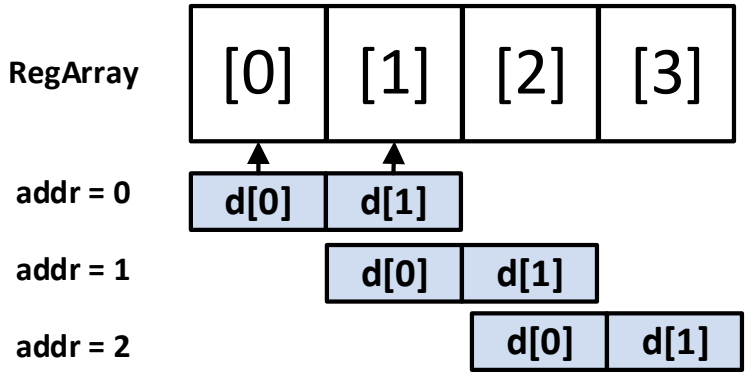
```
reg [7:0] regfile_r [0:31], regfile_w [0:31];
integer i;

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
- Manually 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

- Syntax:

```
while (expression)
  <statement>
```

- The **`while`** loop executes until the expression evaluates to ***false***
- **NON**-SYNTHESIZABLE
- Example

```
reg [7:0] tempreg;  reg [2:0] count = 0;

initial begin
    tempreg = rega;
    while (tempreg) begin
        if (tempreg[0]) count = count + 1;
        temp = temp >> 1;
    end
end
```

rega = 101;

tempreg	count
101	1
010	1
001	2

Repeat Loop

- Syntax:

```
repeat (expression)
  <statement>
```

- The **repeat** construct executes the loop a **fixed** number of times
- **NON**-SYNTHESIZABLE
- Example

```
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
```

default repeat 8 times

Forever Loop

- Syntax:

```
forever <statement>
```

- The **forever** loop does not contain any expression and executes forever until the **$finish** task is encountered
- **NON**-SYNTHESIZABLE
- Example

```
//Clock generation
//Clock with period of 20 units)
reg clk;

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

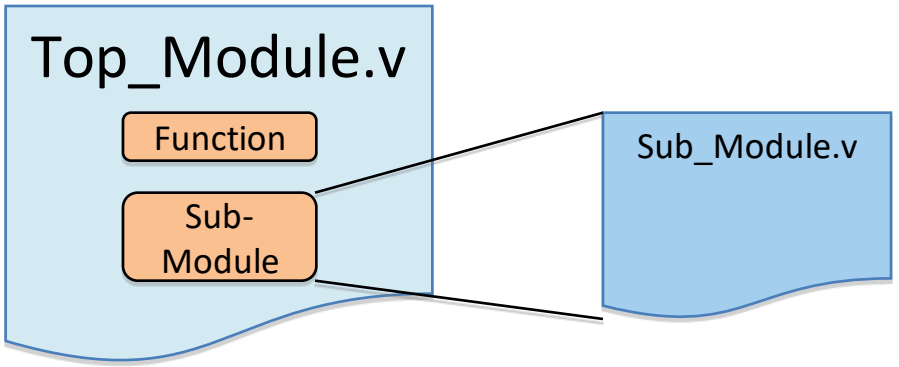
```
//Synchronize 2 register values
//at every positive edge of clock
reg clk;
reg x,y;

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

Outline

- Introduction to Behavioral Level
- Procedural Construct
- Timing Control
- Procedural Statements
- Functional Partition
  - Sub-modules
  - **`function`**
  - **`task`**
  - **`generate`**
- Finite State Machine (FSM)

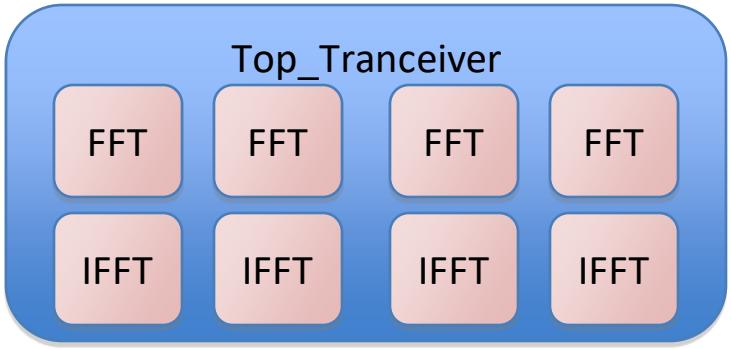
Functional Partition

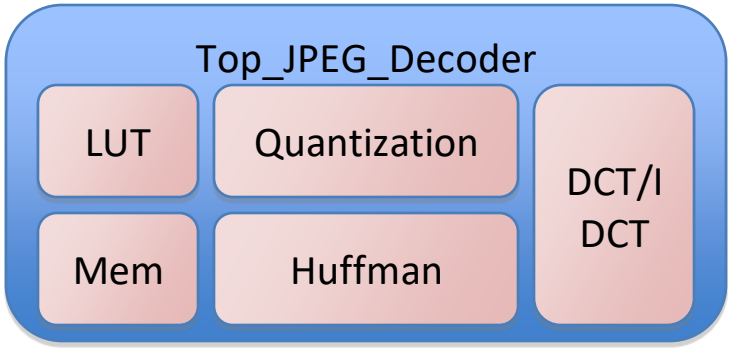
- A digital system may consist of many functional blocks
  - A lumped Verilog module makes debugging & editing a great disaster
- Break the whole system into several functional blocks
  - Sub-module
  - Function
  - Task



Sub-Module (1/2)

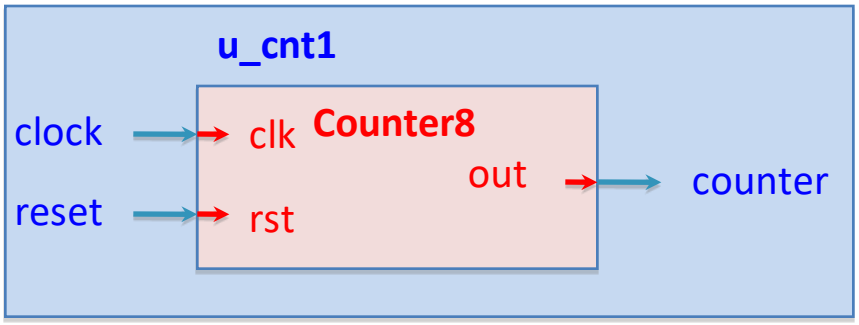
- When to use a sub-module
  - Functional block with many duplications
  - Functional block containing specific computation





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)
);
```

Must be wire

Sub-module

```
module Counter8(
    input  clk,
    input  rst,
    output 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
- Typically used to perform a common 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 the name of function is declared implicitly inside
  - Thus, the function cannot have more than 1 output
- Functions can access other signals in the module
- Functions can only invoke other functions
- Functions cannot invoke other tasks

Function (3/3)

- Function example

```
// function definition
function [7:0] abs;          // unsigned value
    input [7:0] number_in;   // signed value
    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**
- Typically used to perform debugging operations
  - May 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 in the same order of their declaration
- The output shall be connecting to a variable

```
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)
- Task can access other signals in the module
- Signals used in timing controls (such as clk) must not be inputs of 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;
  begin
      #20 $display("@%d, a = %d, b = %d", $time, a, b);
      #10 $display("@%d, task finish!", $time);
  end
endtask
```

```
// without 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 fork
    errmon;
    begin
        runtest;
        disable errmon;
    end
join
```

Comparison: Function & Task

Function	Task
1. Typically used to perform a computation, or to represent combinational logic 2. Called in procedural statement or continuous assignment	1. Typically used to perform debugging operations, or to behaviorally describe hardware 2. Called only in procedural statement
Can only 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 **Cannot have output or inout** arguments	Does not return a value but can pass multiple values through output and inout arguments

Generate

- Use **`generate`** to either conditionally or multiply instantiate 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

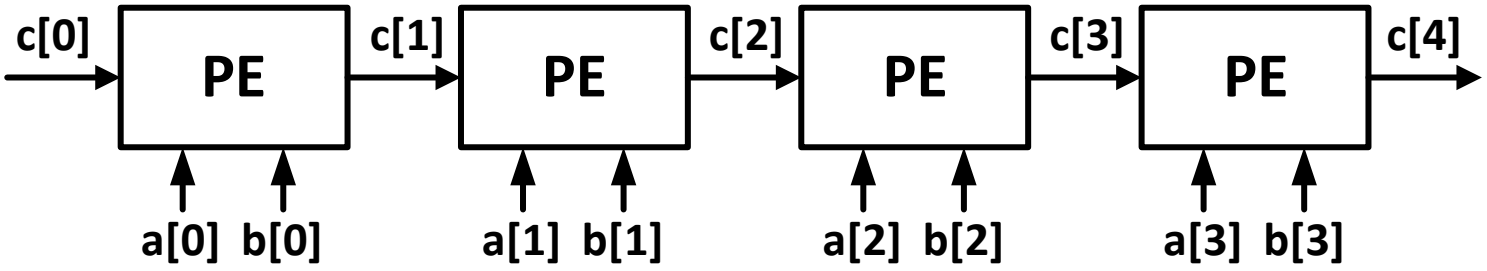
Loop Generate Constructs (1/2)

- 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 (2/2)

- 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

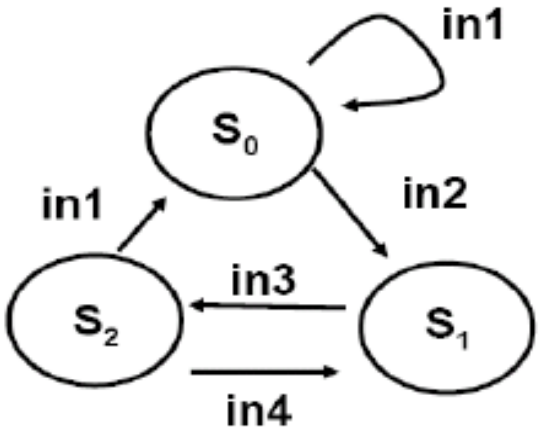
```
generate
  if((a_width < 8) || (b_width < 8)) begin: mult
    Low_Power_Module #(a_width,b_width) u1(a, b, out);
  end
  else begin: mult
    High_Performance_Module #(a_width,b_width) u1(a, b, out);
  end
endgenerate
```

Outline

- Introduction to Behavioral Level
- Procedural Construct
- Timing Control
- Procedural Statements
- Functional Partition
- Finite State Machine (FSM)
  - Mealy Machine & Moore 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



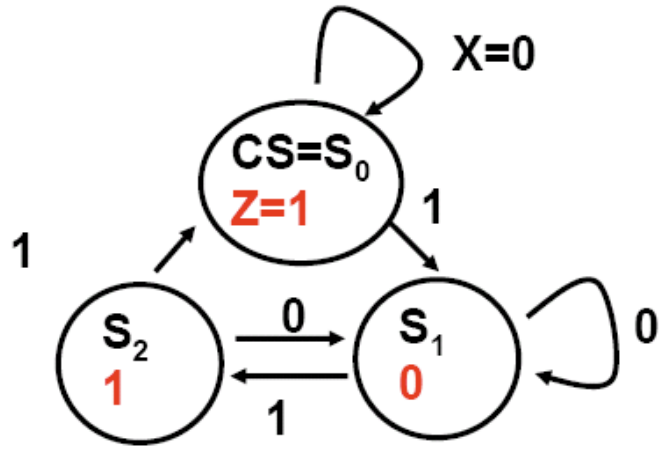
State transition diagram

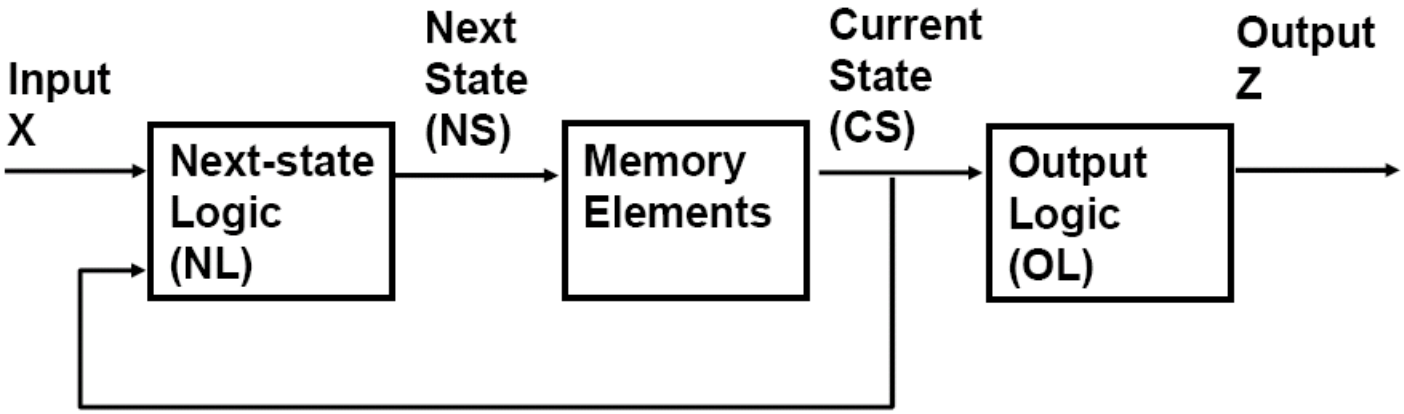
Elements of FSM

- Memory Elements
  - Memorize Current States (CS)
  - Usually consist of FF or latch
  - N-bit FF have 2n 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 – Moore machine
    - Based on current state and input – Mealy machine

Moore Machine

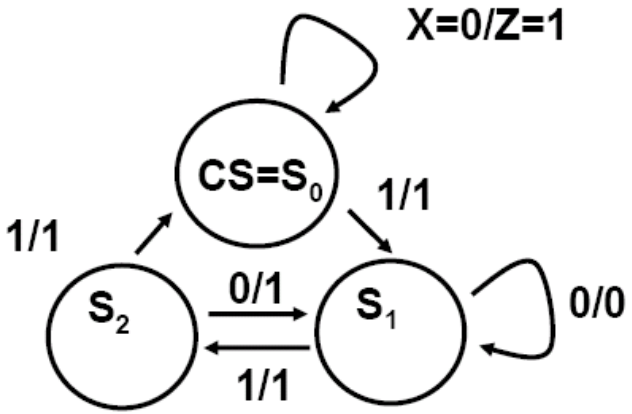
- Output is a function of
  - **Only current state**

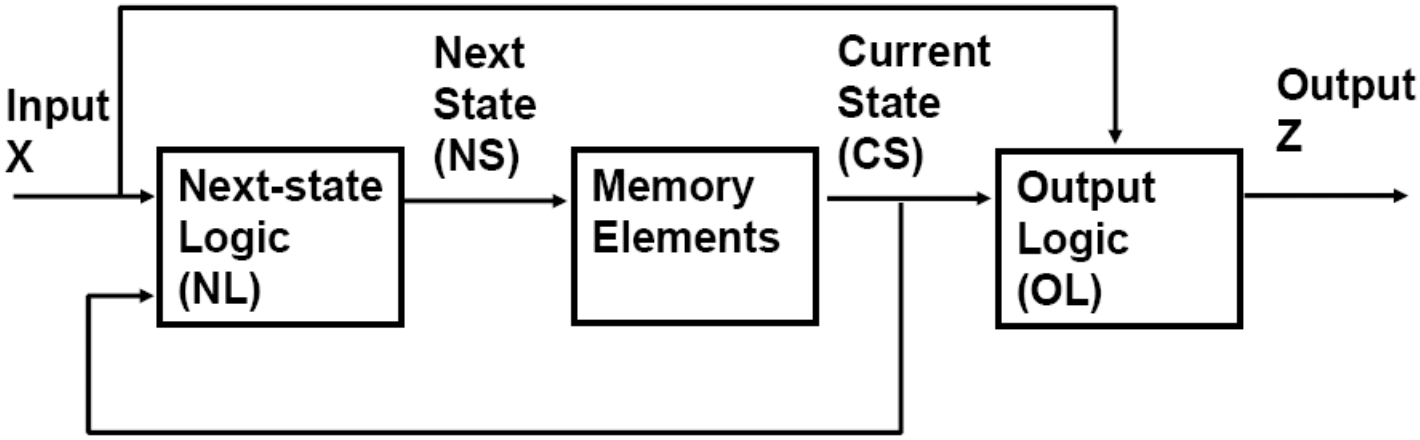




Mealy Machine

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





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
```

Avoid mixing comb. and seq. in one always block

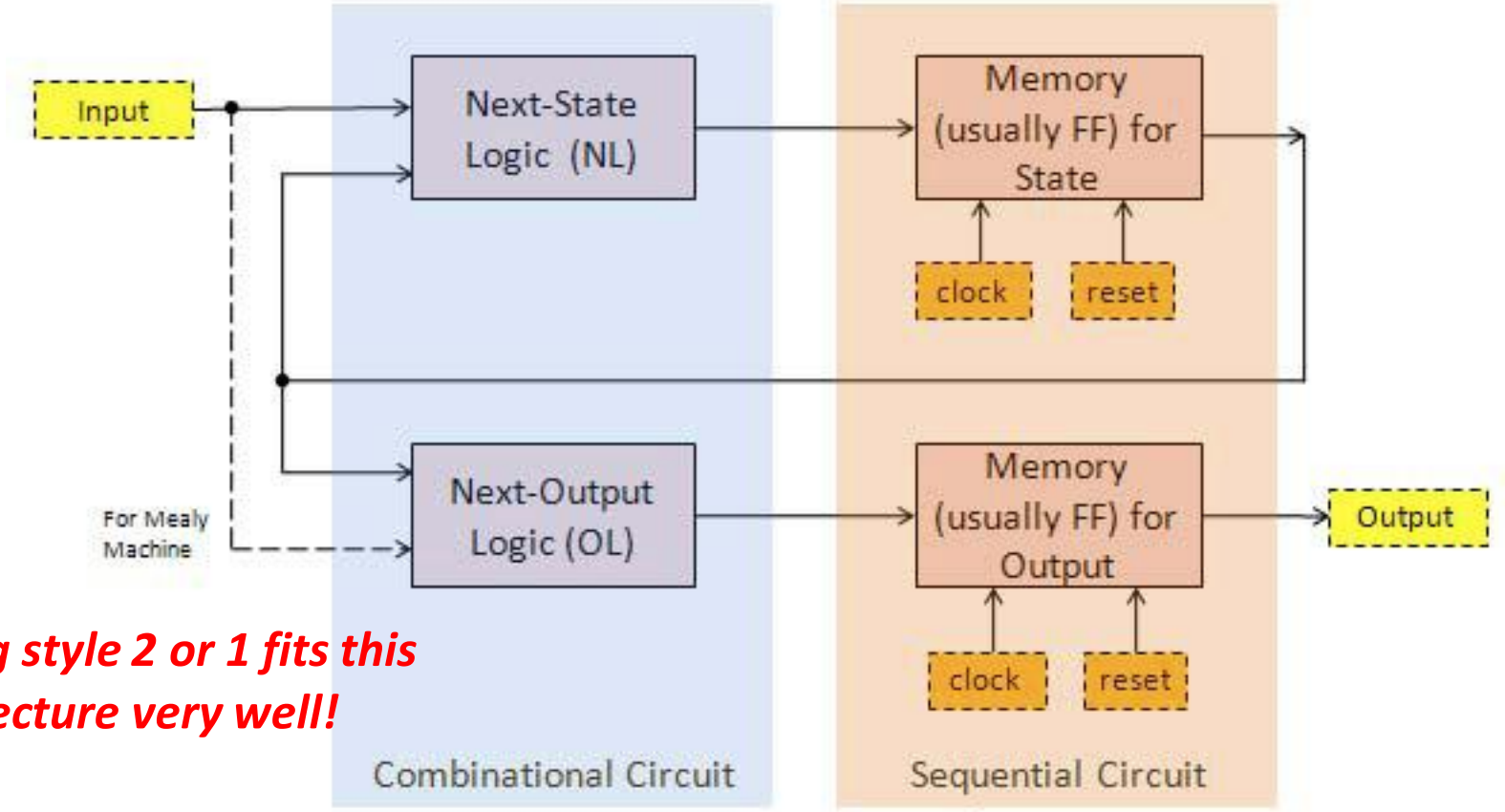
- OL

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

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

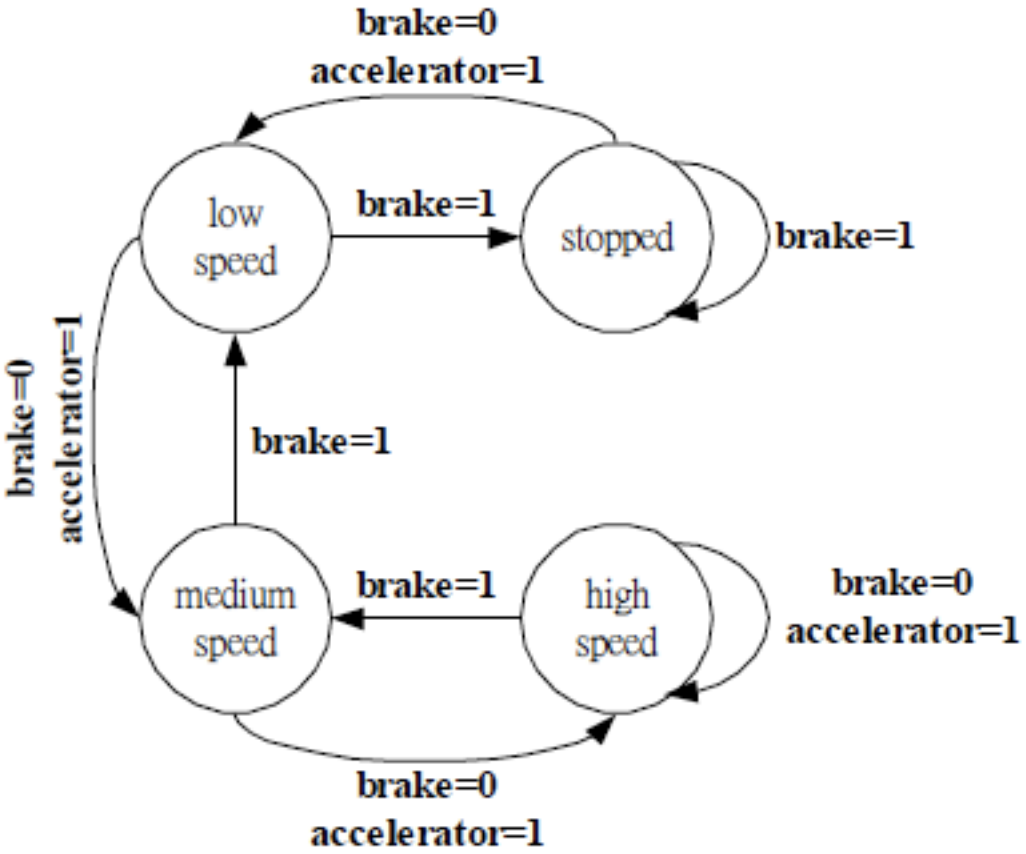
Architecture of FSM

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

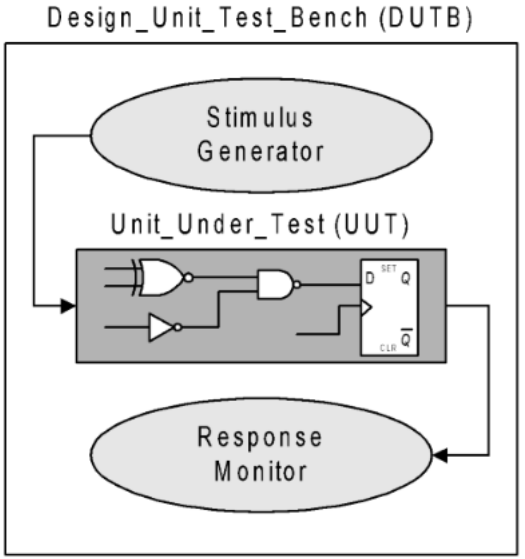
Lecturer: 徐以帆

Graduate Institute of Electronics Engineering, National Taiwan University

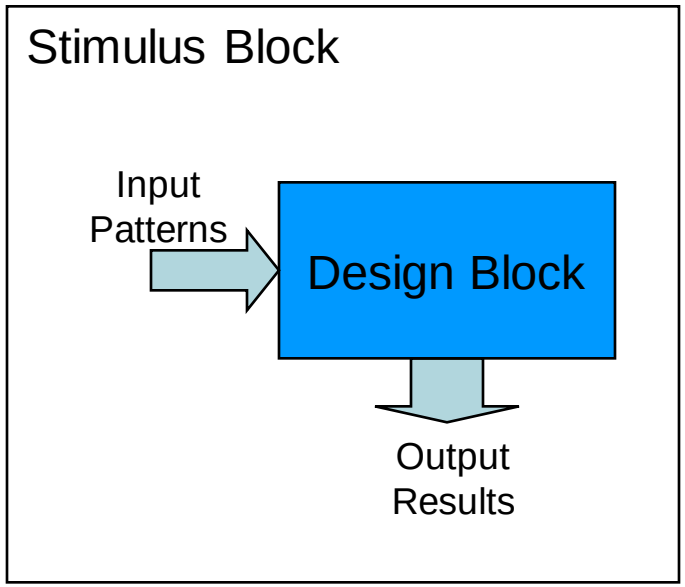
NTU GIEE

Verification Methodology

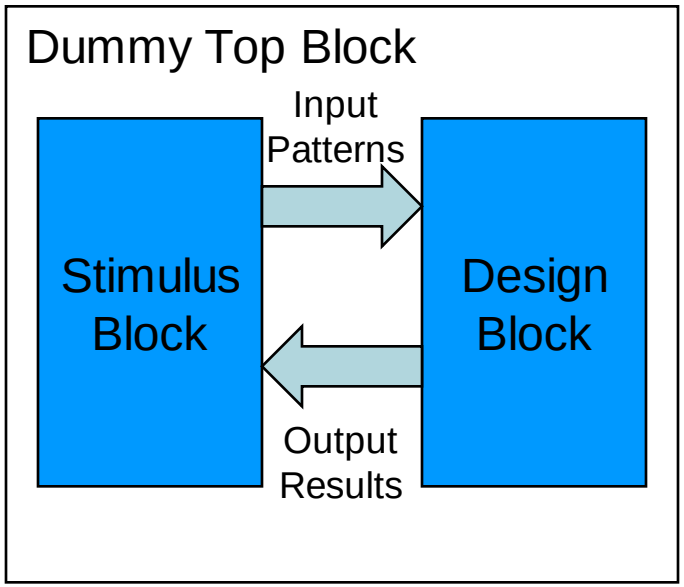
- Goal: systematically verify the functionality of a model
- Approaches: Simulation and/or formal verification
- Simulation:
  - (1) detect syntax violations in source code
  - (2) simulate behavior
  - (3) monitor results
- Formal verification:
  - (1) mathematically prove the correctness



Components of a Simulation

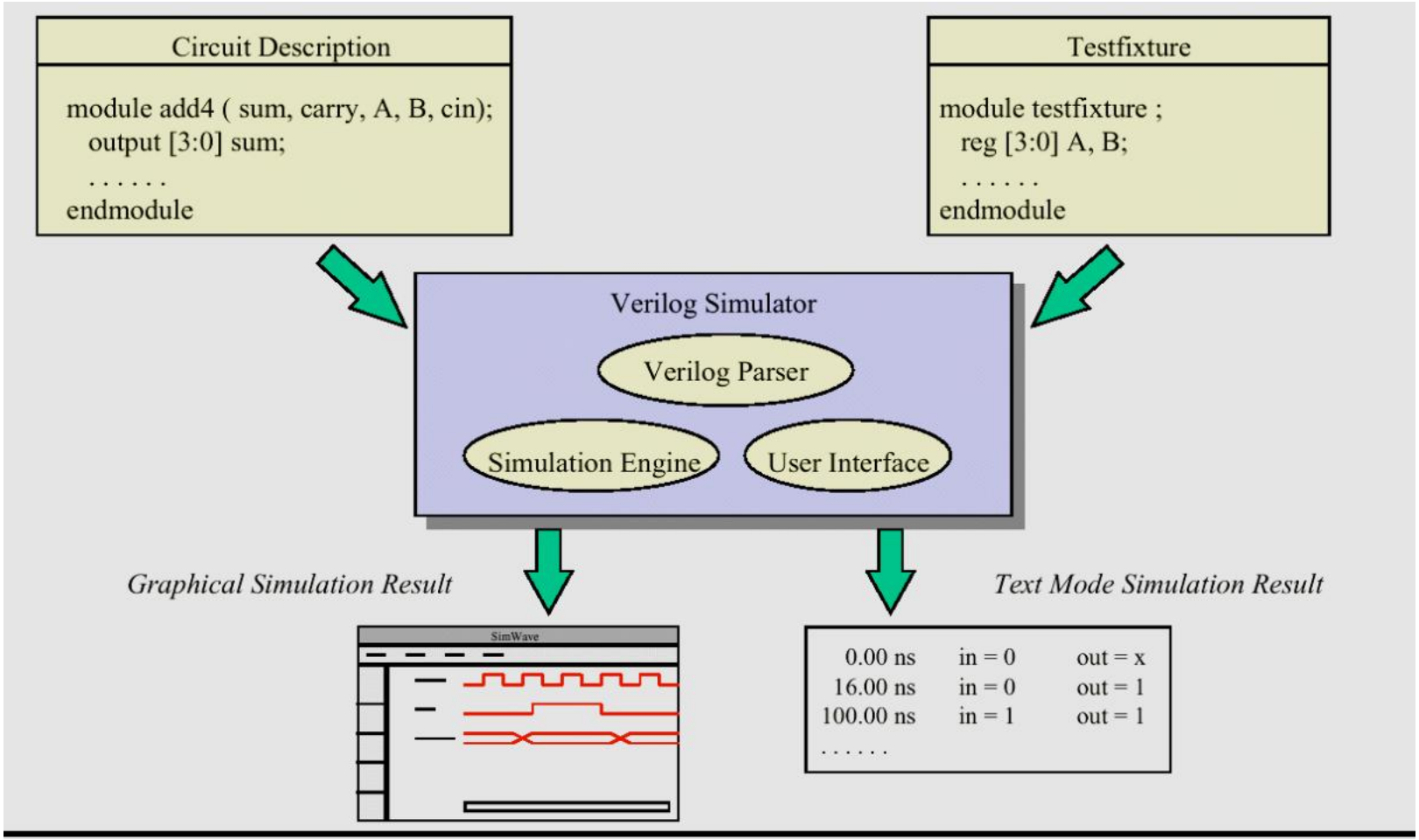


The output results are verified by console/waveform viewer



The output results are verified by testbench or stimulus block

Verilog Simulator



Testbench Template

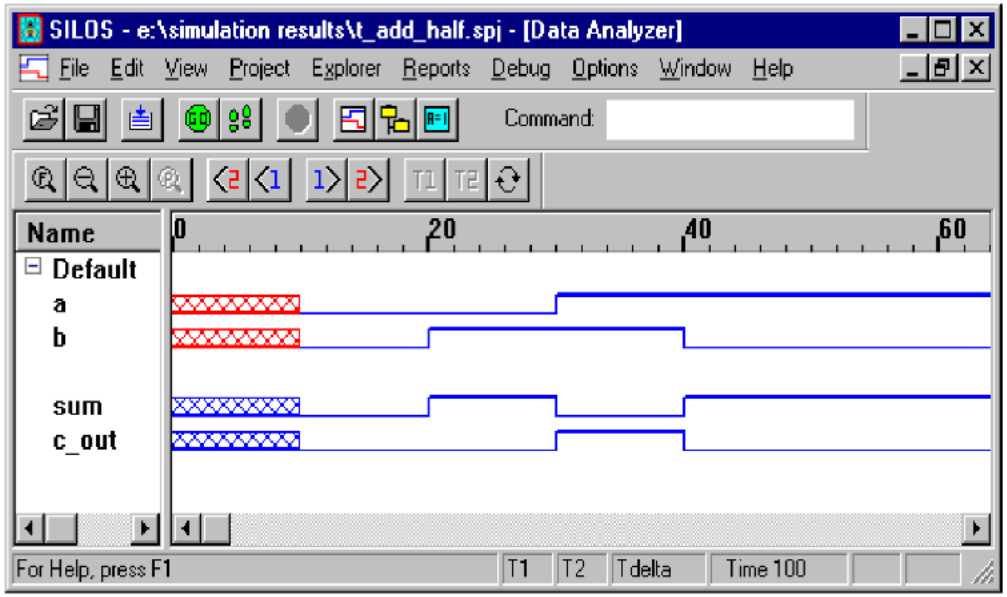
- Consider the following template as a guide for simple testbenches:

```
`timescale 1ns/10ps

module tb_DUT_name;              // substitute the name of the DUT
  parameter time_out = 1000;     // Provide a value
  reg …;                         // Declaration of primary inputs of the DUT
  wire …;                        // Declaration of primary outputs of the DUT

  DUT_name u_DUT_instance ( /* DUT ports */ );

  initial $monitor(…);           // Signals to be monitored and displayed as text
  initial #(time_out) $stop;     // (Also $finish) Stopwatch to assure
                                 // termination of simulation
  initial                        // Behaviors for pattern generation and/or
    begin                        // error detection
      // Behavioral statements generating waveforms to the input ports,
      // and comments documenting the test.
      // Use all kinds of behavioral constructs for loops and conditionals.
    end
endmodule
```

Example: Testbench

```
`timescale 1ns/10ps

module tb_Add_half;
  parameter time_out = 100;
  reg  a, b;
  wire sum, c_out;

  Add_half M1 (sum, c_out, a, b); // DUT

  initial #(time_out) $finish;    // Time out stopwatch

  initial begin                   // Stimulus patterns
    #10 a = 0; b = 0;             // Statements execute
    #10 b = 1;                    // in sequence
    #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,"Values: clk=%b rst=%b",clk,rst);`

```
> 0 Values: clk=0 rst=1
> 5 Values: clk=1 rst=1
> 10 Values: clk=0 rst=0
```

- Stopping and finishing in a simulation
  - `$stop;` // pause the simulation and enter interactive mode
  - `$finish;` // terminate the simulator

System Tasks and Function: $ (2/2)

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

```
parameter  ram_depth  = 256;
localparam addr_width = $clog2(ram_depth);
```

- 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 <macro_name> <macro_text>
  - `define RAM_SIZE 16
  - Defining a macro and the substituting text
  - The identifier `RAM_SIZE will be replaced by 16
- `include "<filename>"
  - `include "adder.v"
  - Including the entire contents of another file
- `timescale <time_unit> / <time_precision>
  - `timescale 1ns/10ps
  - Setting the simulation time unit and precision of your simulation
  - Make sure the precision is small enough to represent all timing control delays

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: Simulate on real time scale, used by SPICE simulators
  - **Event-based**: Simulate on events of signal transition, used by Verilog simulators.
    - Each event must occurs on discrete timestep specified by the simulation precision
  - Cycle-based: Simulate on cycle, used by system level verification, less used in cell-based IC design