Digital System Design

Logic Design at Register-Transfer Level - Procedural Assignments

Lecturer: 王景平

Date: 2025.03.13

Based on: Ch.8-10 & Appendix A. of the textbook
Review: Logic Design
(Concepts of Combinational /Sequential Circuits and Finite State Machines)

Operators

Concatenation and replications	{,}
Bitwise	~, &, ~^, ^, |
Reduction	&, |, ^, ^~
Arithmetic	+ , - , * , / , %, **
Shift	>> , <<, >>>, <<<
Relational	< , <= , > , >=
Equality	== , != , === , !==
Logical	!, && , || **Not Synthesizable**
Conditional	? :

Singed for Arithmetic Operators

```
module test;
reg  [3:0] A,B;
wire [4:0] sum;

assign sum = A+B;

initial begin
  #5 A=5;  B=-2;
  $display(" A  B  sum");
  $display("%d %d %d",A,B,sum);
end

endmodule
```

A	B	Sum
5	14	19
00101	01110	10011
	↑ unsigned	↑ unsigned

```
reg signed [3:0] A,B;
assign sum= A+B;
```
←Recommended

```
reg [3:0] A,B;
assign sum = $signed(A)+$signed(B);
```

```
module test;
reg  signed [3:0] A,B;
wire signed [4:0] sum;

assign sum = A+B;

initial begin
  #5 A=5;  B=-2;
  $display(" A  B  sum");
  $display("%d %d %d",A,B,sum);
end

endmodule
```

A	B	Sum
5	-2	3
00101	11110	00011

Bit Length of Arithmetic Operations

- (Signed or Unsigned) addition bit length
  - A(8 bits) + B(8 bits) → C(8+1 bits)
  - A(M bits) + B (N bits) → C(max(M, N)+1 bits)

```
wire [7:0] A, B;
wire [8:0] C;
assign C = {A[7], A} + {B[7], B};
assign C = $signed(A) + $ signed(B);
```

```
wire signed [7:0] A, B;
wire signed [8:0] C;
assign C = A+B;
```

- (Signed or Unsigned) multiplication bit length
  - A(3 bits) x B(5 bits) → C(3 + 5 bits)

```
wire signed [2:0];
wire signed [4:0];
wire signed [7:0];
assign C = A * B;
```

Ref: https://zipcpu.com/dsp/2017/07/21/bit-growth.html

Assignments

- Assignment: Drive value onto nets and registers
- There are two basic forms of assignment
  - continuous assignment, which assigns values to wire type
  - procedural assignment, which assigns values to reg type
- Basic form

Assignments	Description	Left Hand Side	Example
Continuous Assignment	appear outside procedures	wire	wire a; assign a = 1'b1;
Procedural Assignment	appear inside procedures	reg	reg a; always@(*) a = 1'b1;

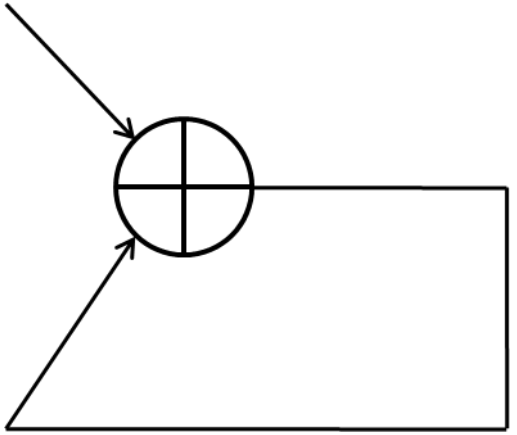
P.S. Left hand side (LHS) = Right hand side (RHS)

✓ ***Synthesizable!***

Avoiding Combinational Loops

- Avoid combinational loops (or logic loops)
  - Without disabling the combinational feedback loop, the static timing analyzer can't resolve
  - Example

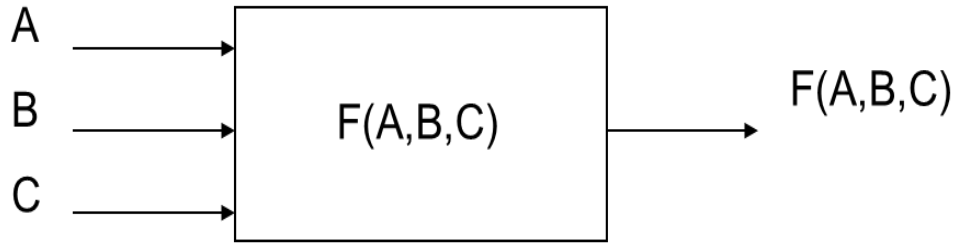
```
wire [3:0] a;
wire [3:0] b;

assign a = b + a;
```

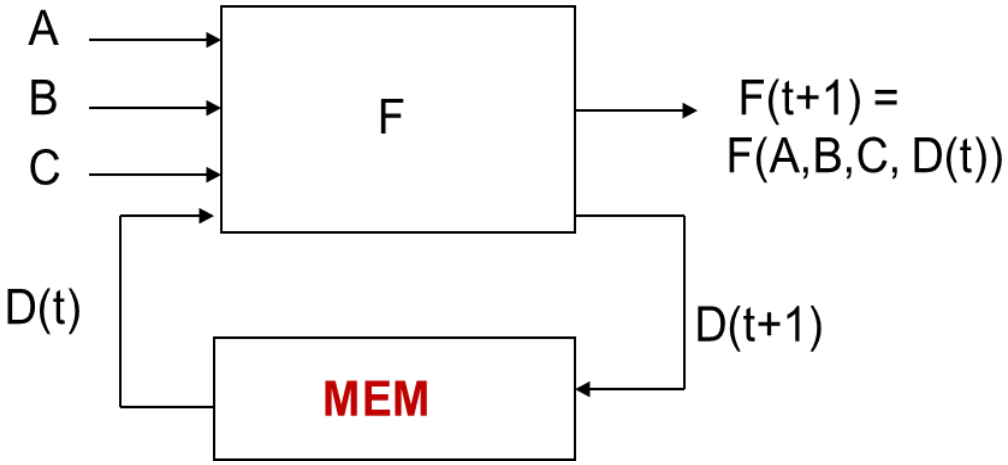


But how to implement a loop in Verilog ➔ Sequential Circuits

Combinational & Sequential Circuits

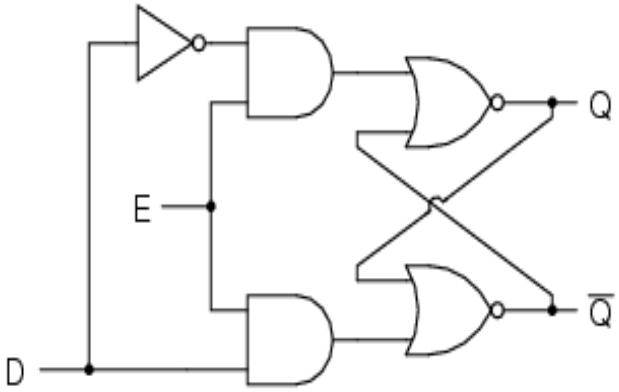
- Combinational Circuits
  - Without memory
  - output only depends on current input
  - Adder tree, ALU
- Sequential Circuits
  - With memory
  - output depends on current input and previous stored value
  - Latch, D F/F





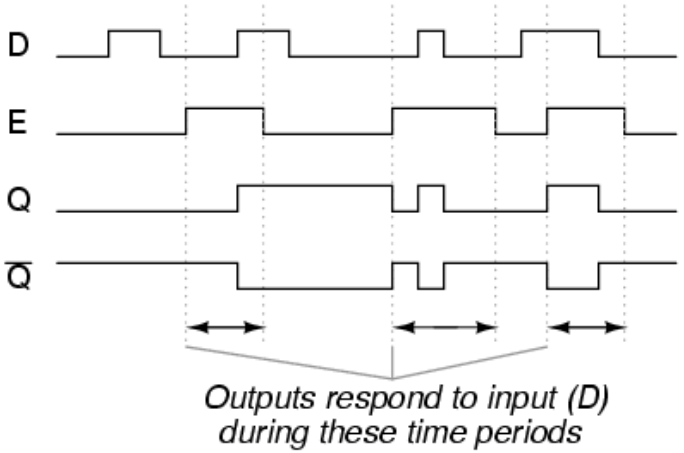
Latch

- Level-sensitive register
- Most of designs should not infer latches
  - Most industry chip designs follow a “synchronous” design methodology
  - Latch-based designs are susceptible to timing problems
  - Glitches in the enable pin of the latches that can cause unrecoverable failure (transparent)
  - **Inferred latches after synthesis due to bad coding styles!**



E	D	Q	$\overline{Q}$
0	0	latch	latch
0	1	latch	latch
1	0	0	1
1	1	1	0

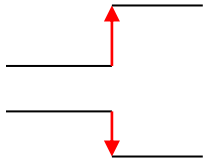
Regular D-latch response

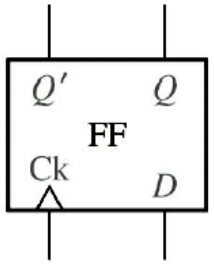


Edge-triggered D Flip-Flop

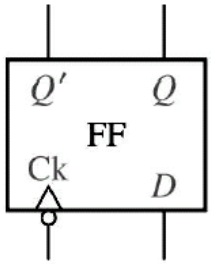
- Positive (Rising edge) trigger
- Negative (Falling edge) trigger

→to align with clock edges





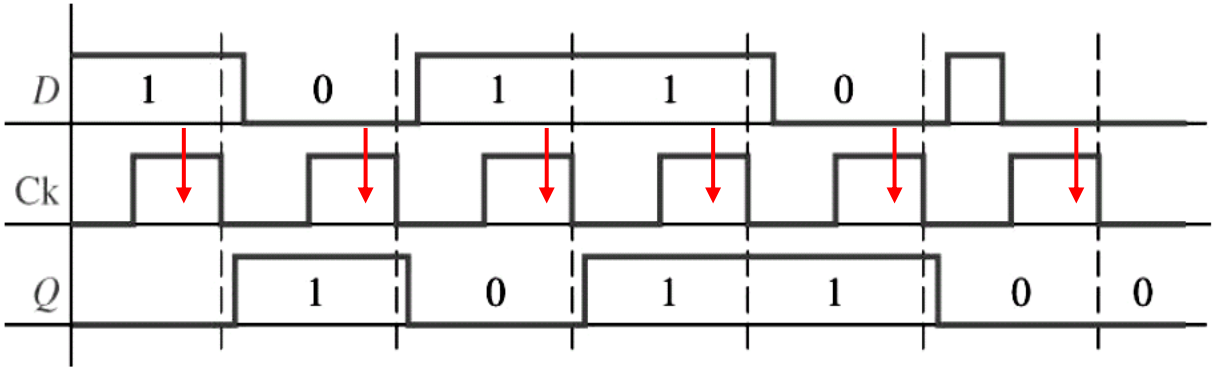
(a) Rising-edge trigger



(b) Falling-edge trigger

D Q	Q^+
0 0	0
0 1	0
1 0	1
1 1	1

$Q^+ = D$



Timing for D Flip-Flop ***(Falling-Edge Trigger)***

```
1  /*=======================================
2  Author: Yu Chuan, Chuang
3  Module: Counter
4  Description:
5  When getting start_i signal, counter starts
6  to count from 0 to 15.
7  =======================================*/
8  module counter (
9      input            clk,
10     input            rst,
11     input            start_i,
12     output  [3:0]    count_o
13 );
14
15 //============  Parameter ==============
16     localparam  STATE_IDLE  = 1'b0;
17     localparam  STATE_CNT   = 1'b1;
18
19 //=========  Reg/Wire Declaration ======
20     reg         state, nxt_state;
21     reg [3:0]   cnt, nxt_cnt;
22
23 //========= Finite State Machine =======
24     always@(posedge clk or posedge rst) begin
25         if(rst)
26             state <= STATE_IDLE;
27         else
28             state <= nxt_state;
29     end
30
31     always@(*) begin
32         case(state)
33             STATE_IDLE: begin
34                 if(start_i)
35                     nxt_state = STATE_CNT;
36                 else
37                     nxt_state = STATE_IDLE;
38             end
39             STATE_CNT: begin
40                 if(cnt == 4'd15)
41                     nxt_state = STATE_IDLE;
42                 else
43                     nxt_state = STATE_CNT;
44             end
45             default: nxt_state = STATE_IDLE;
46         endcase
47     end
48 //============ Combinational ============
49     assign count_o = cnt;
50
51     always@(*) begin
52         if(state == STATE_CNT) begin
53             nxt_cnt = cnt + 1;
54         end
55         else begin
56             nxt_cnt = 0;
57         end
58     end
59
60 //============= Sequential =============
61     always@(posedge clk or posedge rst) begin
62         if(rst)
63             cnt <= 0;
64         else
65             cnt <= nxt_cnt;
66     end
67
68
69 endmodule
```

Header/Comment

Module instantiation

Input/output declaration

Parameter

Number Representation

Reg/WIre

If statement

FSM

Procedure Block

case statement

Combinational

Continuous Assignment

Sensitivity list

Operator

Procedure Assignment

Sequential

Outline

- Procedural Construct and Assignment
  - initial block
  - always block
  - Procedural assignment
- Control Structure
- Finite State Machine (FSM)
  - Moore Machine & Mealy Machine
  - Modeling of FSM
- Loop Statements
- Functional Block
  - Sub-modules
  - function and task

Procedural Blocks (1/3)

- 2 types of blocks
  - Sequential: **begin … end**
  - Concurrent: **fork … join (Not Synthesizable)**
  - Can be omitted for **one-line** block
- Blocks can be treated as a **statement**
  - Statement: `a = b + c;`
- Statements can be decorated with event/delay
  - Event: `@(event) a = b + c;`
  - Delay: `#(delay) a = b + c;` **(Not Synthesizable)**
- Statements are either **inside a block** or **described by**
  - `initial`: once at the beginning of simulation
  - `always`: every time the statement finishes

Only always block is synthesizable

Procedural Blocks (2/3)

- Statements can be
  - Assignment: **`a = b + c;`**
  - Loop structure: **`while (cond) Statement`**
  - Condition structure: **`if (cond) Statement`**
- Assignment
  - **`{ cout, sum } = a + b + cin;`**
  - RHS can be either wires or regs
  - LHS can only be **regs**
    - Continuous assignment(**assign**) is for wires

Procedural Blocks (3/3)

```
initial begin
    $display("Initial");
end
```

```
initial begin
    #5 $display("Initial");
    #5;
    #5 $display("15t passed");
end
```

```
initial fork
    #5  $display("Initial");
    #10;
    #15 $display("15t passed");
join
```

```
always #5 clk = ~clk;
```

```
initial begin
  fork
    begin
      in_valid = 0;
      @(posedge clk);
      in_valid = 1;
      @(posedge clk);
      in_valid = 0;
    end
    begin
      while (out_valid !== 0)
        @(negedge clk);
    end
  join
end
```

Sensitivity List (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
```

Sensitivity List (2/2)

- Easy way to use sensitivity list for combinational circuit
  - Use always @ (*)

```
always@(a or b or x) begin
    y = ~x;
    x = a | b;
end
```

```
always@(*) begin
    y = ~x;
    x = a | b;
end
```

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
```

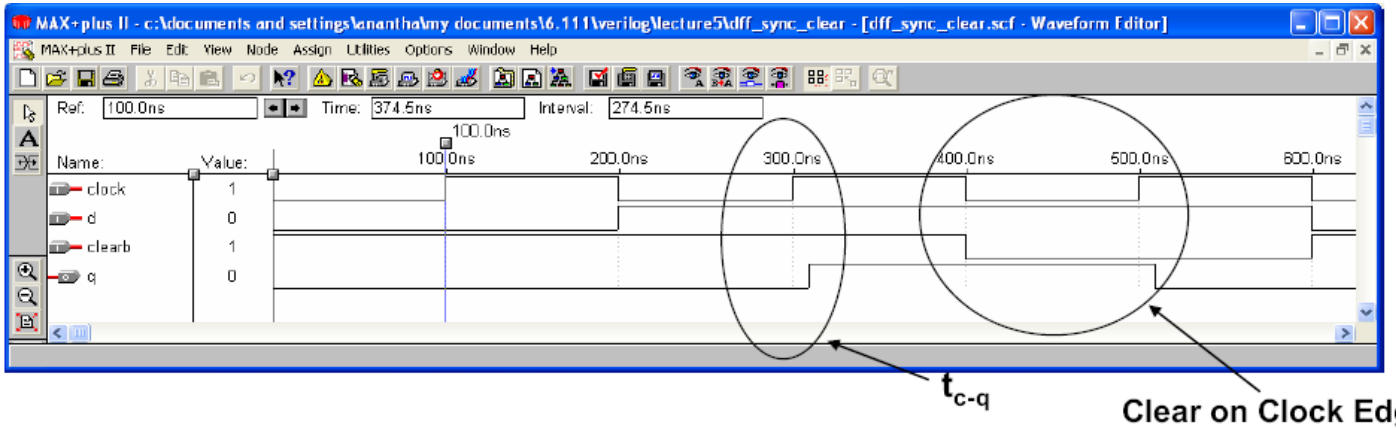
`always` block entered immediately when (active-low) clearb is asserted

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

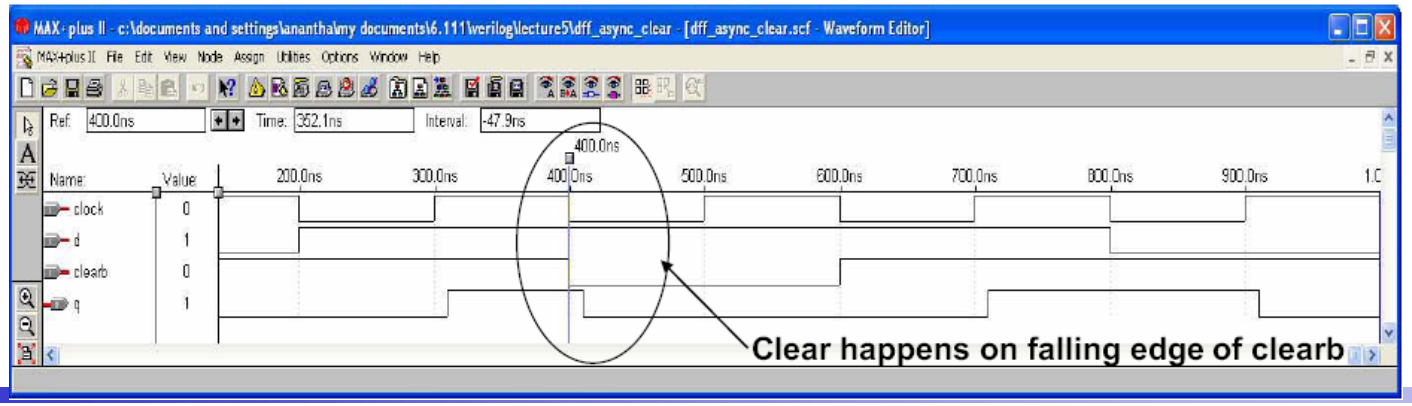
If one signal in the sensitivity list uses posedge/negedge, then all signals must.

Modeling of Flip-Flops (2/2)

- Synchronous Reset



- Asynchronous Reset



Synthesizable Always Blocks

- Combinational Circuit

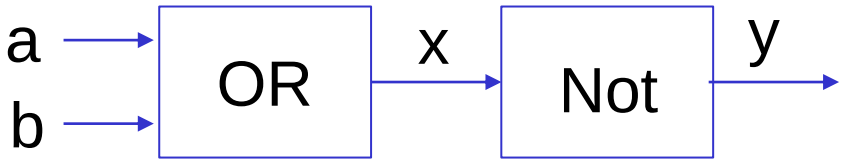
```
always @(*) begin
    y = x;
    …
end
```

- Sequential Circuit

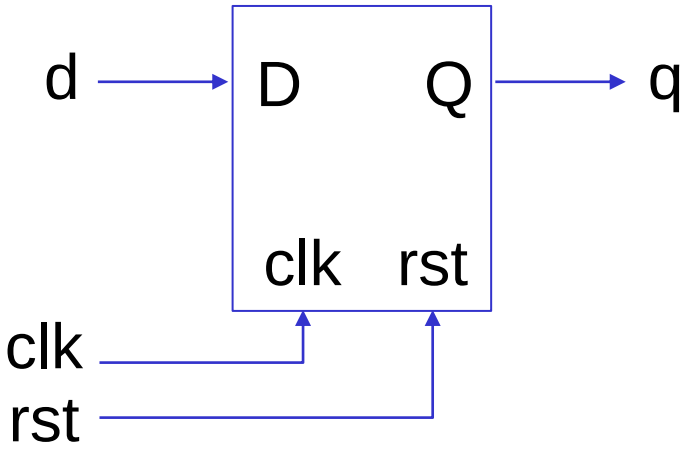
```
always @(posedge clk or negedge rst_n) begin
    if (~rst_n) y <= 'b0;
    else        y <= x;
end
```

Combination & Sequential

- Combinational Ckt.



- Sequential Ckt.




```
reg x;
reg y;
always@(*) begin
    y = ~x;
    x = a | b;
end
```

```
reg q;
always@(posedge clk) begin
    if (rst) q <= 0;
    else     q <= d;
end
```

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
  - Difficult to model the concurrency
  - Usually be used for design the **Combinational** circuit
- Non-blocking assignment (<=)
  - Evaluate the RHS, but schedule the LHS
  - Update LHS only after evaluate all RHS
  - Greatly simplify modeling concurrency
  - Suitable for design the **Sequential** circuit

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
```

- Non-blocking assignment (<=)
  - All assignment deferred until all right-hand sides have been evaluated (end of the virtual timestamp)

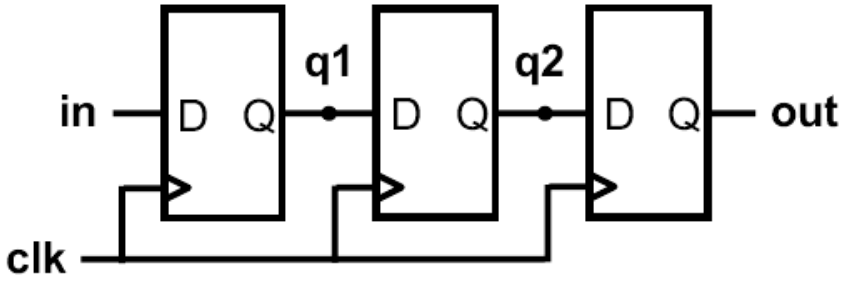
```
always @ (a or b or c)
begin
   x <= a | b;         1. Evaluate a | b but defer assignment of x
   y <= a ^ b ^ c;     2. Evaluate a^b^c but defer assignment of y
   z <= b & ~c;        3. Evaluate b&(~c) but defer assignment of z
end                    4. Assign x, y, and z with their new values
```

Non-Blocking for Sequential Logic

- Blocking assignments do not reflect the intrinsic behavior of multi-stage **sequential logic**

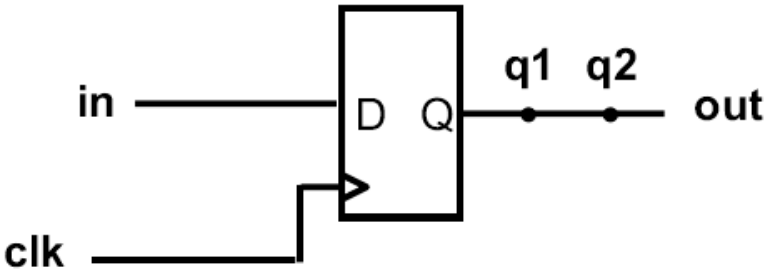
```
always @ (posedge clk)
begin
  q1 <= in;
  q2 <= q1;
  out <= q2;
end
```

"At each rising clock edge, *q1*, *q2*, and *out* simultaneously receive the old values of *in*, *q1*, and *q2*."




```
always @ (posedge clk)
begin
  q1 = in;
  q2 = q1;
  out = q2;
end
```

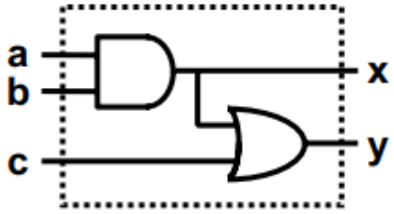
"At each rising clock edge, *q1* = *in*.
After that, *q2* = *q1* = *in*.
After that, *out* = *q2* = *q1* = *in*.
Therefore *out* = *in*."



Blocking for Combinational Logic

- Non-blocking assignments do not reflect the intrinsic behavior of multi-stage **combinational logic**
- While nonblocking assignments can be hacked to simulate correctly (expand the sensitivity list), it's not elegant

Blocking Behavior	a b c x y
(Given) Initial Condition	1 1 0 1 1
a changes; always block triggered	0 1 0 1 1
`x = a & b;`	0 1 0 0 1
`y = x | c;`	0 1 0 0 0




```
module blocking(a,b,c,x,y);
  input a,b,c;
  output x,y;
  reg x,y;

  always @ (a or b or c)
  begin
    x = a & b;
    y = x | c;
  end

endmodule
```

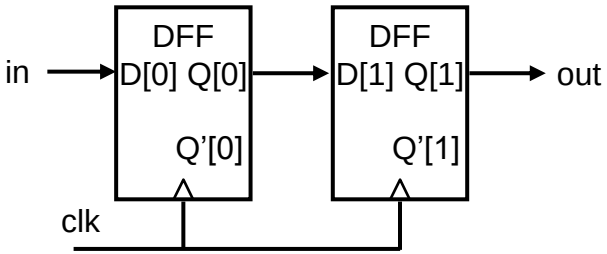
Nonblocking Behavior	a b c x y	Deferred
(Given) Initial Condition	1 1 0 1 1	
a changes; always block triggered	0 1 0 1 1	
`x <= a & b;`	0 1 0 1 1	x<=0
`y <= x | c;`	0 1 0 1 1	x<=0, y<=1
Assignment completion	0 1 0 0 1	

```
module nonblocking(a,b,c,x,y);
  input a,b,c;
  output x,y;
  reg x,y;

  always @ (a or b or c)
  begin
    x <= a & b;
    y <= x | c;
  end

endmodule
```

Race Condition

Example: shift register

```
module shift_r(out, in, clk);
    input in, clk;
    output out;
    wire a
    DFF FF0(a, in, clk);
    DFF FF1(out, a, clk);
endmodule
```



1. Blocking assignment

```
module DFF(q, d, clk);
    input d, clk;
    output q;
    reg a
    always@(posedge clk) q=d;
endmodule
```

2. Non-Blocking assignment

```
module DFF(q, d, clk);
    input d, clk;
    output q;
    reg a
    always@(posedge clk) q<=d;
endmodule
```

Outline

- Procedural Construct and Assignment
  - initial block
  - always block
  - Procedural assignment
- Control Structure
- Finite State Machine (FSM)
  - Moore Machine & Mealy Machine
  - Modeling of FSM
- Loop Statements
- Functional Block
  - Sub-modules
  - function and task

Conditional Statements (1/2)

- If and If-else statements

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

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

```
always@(*) begin
  nxt_a = a;
  nxt_b = b;
  if (sel)
    nxt_a = data;
  else
    nxt_b = data;
end
```

- 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!

```
always@(*) begin
  if (sel)
    nxt_a = data;
  else
    nxt_b = data;
end
```

```
always@(*) begin
  if (sel)
    nxt_a = data;
end
```

Conditional Statements (2/2)

Example 1

```
if (sel==3)
  y = d;
else
  if (sel==2)
    y = c;
  else
    if (sel==1)
      y = b;
    else
      if (sel==0)
        y = a;
```

vs.

Example 2

```
if (sel[1])
  if (sel[0])
    y = d;
  else
    y = c;
else
  if (sel[0])
    y = b;
  else
    y = a;
```

Multiway Branching (Case)

- 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
```

if-else statement

```
if (sel==3)
  y = d;
else
  if (sel==2)
    y = c;
  else
    if (sel==1)
      y = b;
    else
      if (sel==0)
        y = a;
```

case statement

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

```
always@(*)
  y= (sel==3) ? d :
     (sel==2) ? c :
     (sel==1) ? b : a;
```

Example: Binary-Coded-Decimal Counter (1/3)

- Binary-Coded-Decimal (BCD) Counter
  - Input
    - Clock (1 bit)
    - Clear: reset (1 bit)
    - E: enable (1 bit)
  - Output
    - BCD1: tens digit (4 bits)
    - BCD0: unit digit (4 bits)

Decimal Count	BCD Output			
	D	*C*	*B*	*A*
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
0	0	0	0	0

Example: Binary-Coded-Decimal Counter (2/3)

```
    reg [3:0] nxt_BCD1, nxt_BCD0;
//============ Combinational ==============
    always@(*) begin
        nxt_BCD0 = BCD0;
        nxt_BCD1 = BCD1;
        if(E) begin
            if(BCD0 == 4'b1001) begin
                nxt_BCD0 = 0;
                if(BCD1 == 4'b1001)
                    nxt_BCD1 = 0;
                else
                    nxt_BCD1 = BCD1 + 1;
            end
            else begin
                nxt_BCD0 = BCD0 + 1;
            end
        end
        else begin
            nxt_BCD0 = BCD0;
            nxt_BCD1 = BCD1;
        end
    end
```

```
//============ Sequential ==============
    always@(posedge Clock) begin
        if(Clear) begin
            BCD0 <= 0;
            BCD1 <= 0;
        end
        else begin
            BCD0 <= nxt_BCD0;
            BCD1 <= nxt_BCD1;
        end
    end
```

Separate Combinational and Sequential !!!!!!

Example: Binary-Coded-Decimal Counter (3/3)

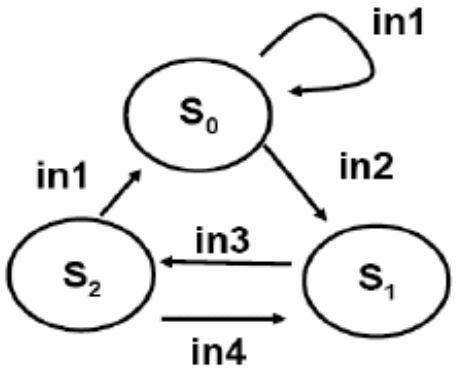
```
1   reg [3:0] BCD1_n, BCD0_n;
2
3   always @(posedge Clock) begin
4       if (Clear) { BCD1, BCD0 } <= 'b0;
5       else       { BCD1, BCD0 } <= {BCD1_n, BCD0_n};
6   end
7
8   always @(*) begin
9       if (~E)                        BCD0_n = BCD0;
10      else if (BCD0 == 4'b1001)      BCD0_n = 4'b0000;
11      else                           BCD0_n = BCD0 + 'b1;
12  end
13
14  always @(*) begin
15      if (~E)                        BCD1_n = BCD1;
16      else if (BCD0 == 4'b1001
17              && BCD1 == 4'b1001)    BCD1_n = 4'b0000;
18      else                           BCD1_n = BCD1 + 'b1;
19  end
```

Outline

- Procedural Construct and Assignment
  - initial block
  - always block
  - Procedural assignment
- Control Structure
- Finite State Machine (FSM)
  - Moore Machine & Mealy Machine
  - Modeling of FSM
- Loop Statements
- Functional Block
  - Sub-modules
  - function and task

Finite State Machine (FSM)

- Model of computation consisting of
  - A set (of finite number) of states
  - An initial state
  - Input symbols (Not necessary)
  - Transition function that maps input symbols and current states to a next state.



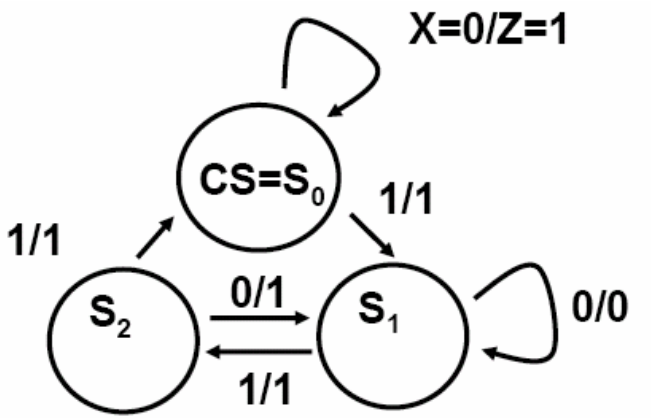
State transition diagram

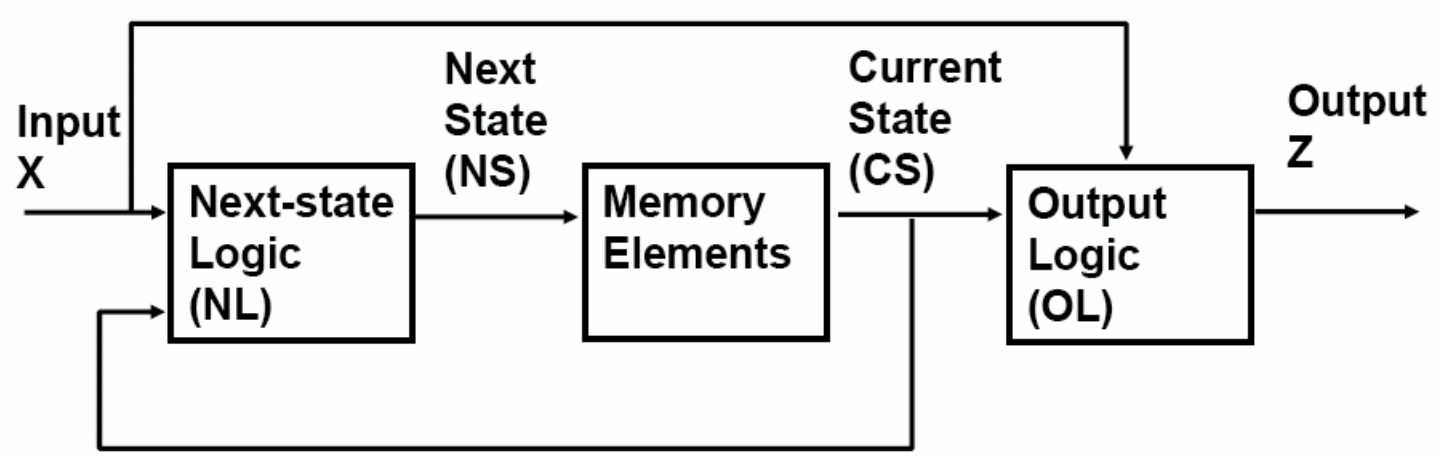
Elements of FSM

- Memory Elements (ME)
  - Memorize Current States (CS)
  - Usually consist of FF
  - 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, or
    - Based on current state and input

Mealy Machine

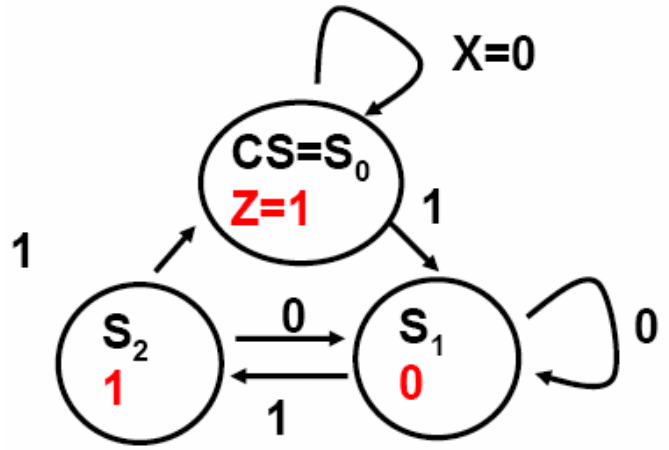
- Output is a function of
  - **both current state & input**

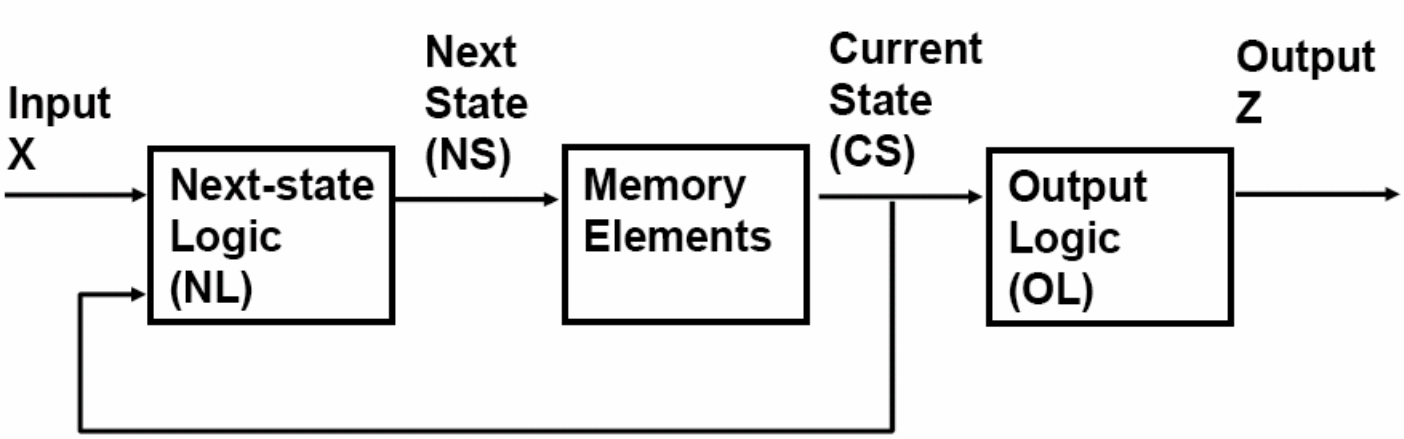




Moore Machine

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

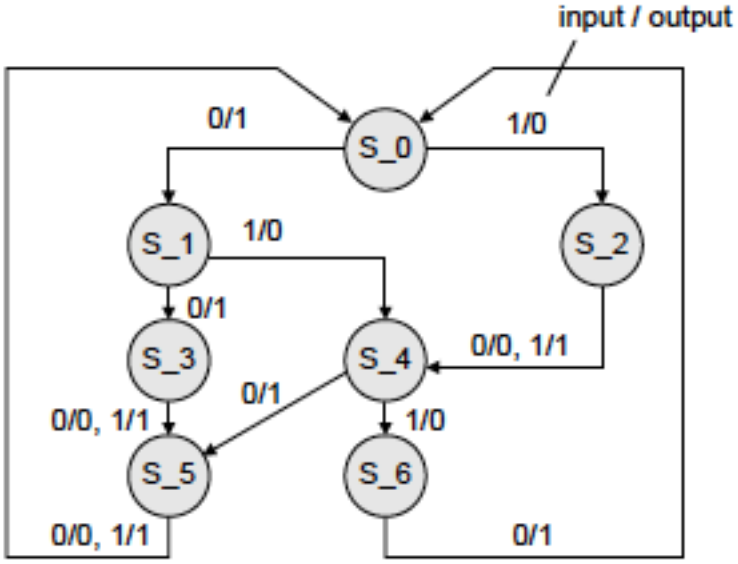




Manual Design of FSM (1/2)

- State Transition Graph describes the mechanism of FSM
- Then encode each state and derive the encoded table

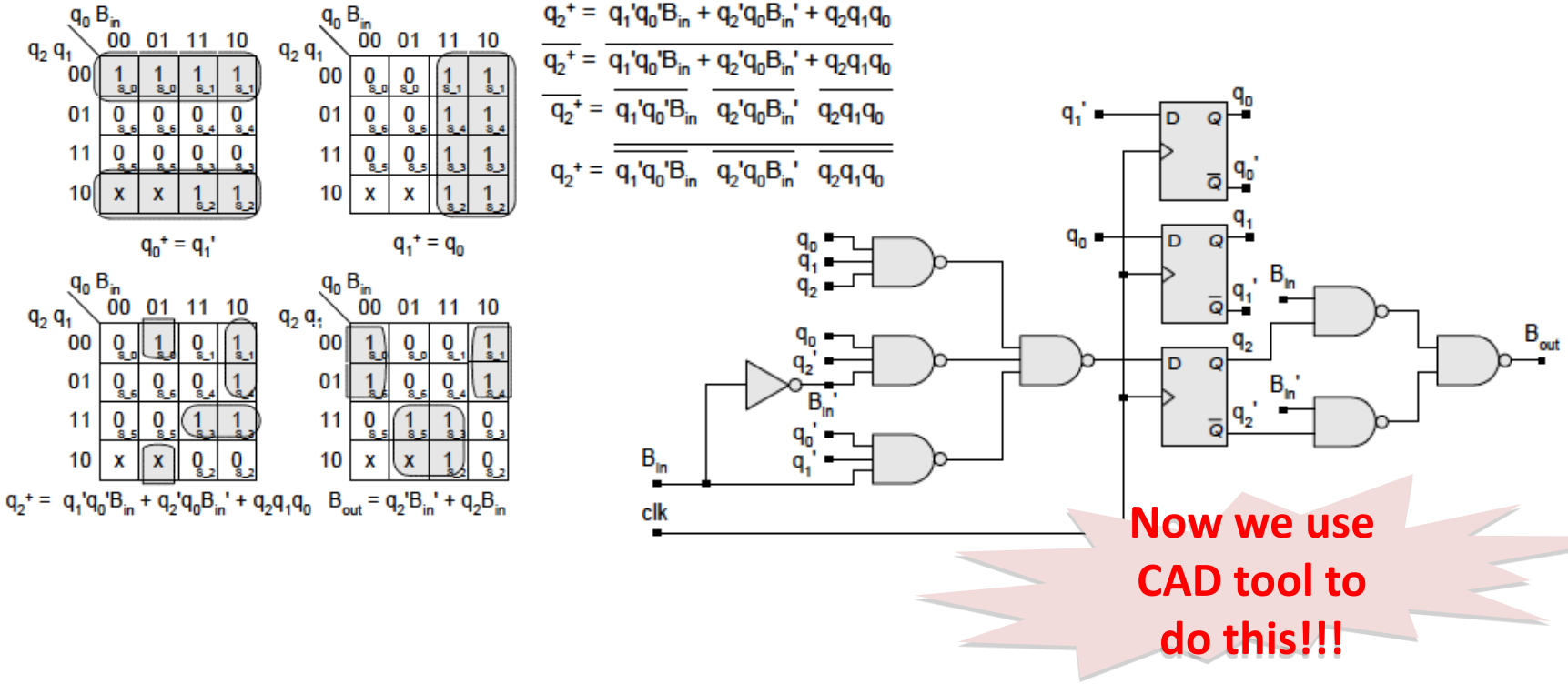
State Transition Graph



Encoded Next state/ Output Table					
	state	next state		output	
	$q_2\ q_1\ q_0$	$q_2^+\ q_1^+\ q_0^+$			
		input		input	
		0	1	0	1
S_0	000	001	101	1	0
S_1	001	111	011	1	0
S_2	101	011	011	0	1
S_3	111	110	110	0	1
S_4	011	110	010	1	0
S_5	110	000	000	0	1
S_6	010	000	-	1	-
	100	-	-	-	-

Manual Design of FSM (2/2)

- Use the K-Map to derive the gate-level FSM design



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
```

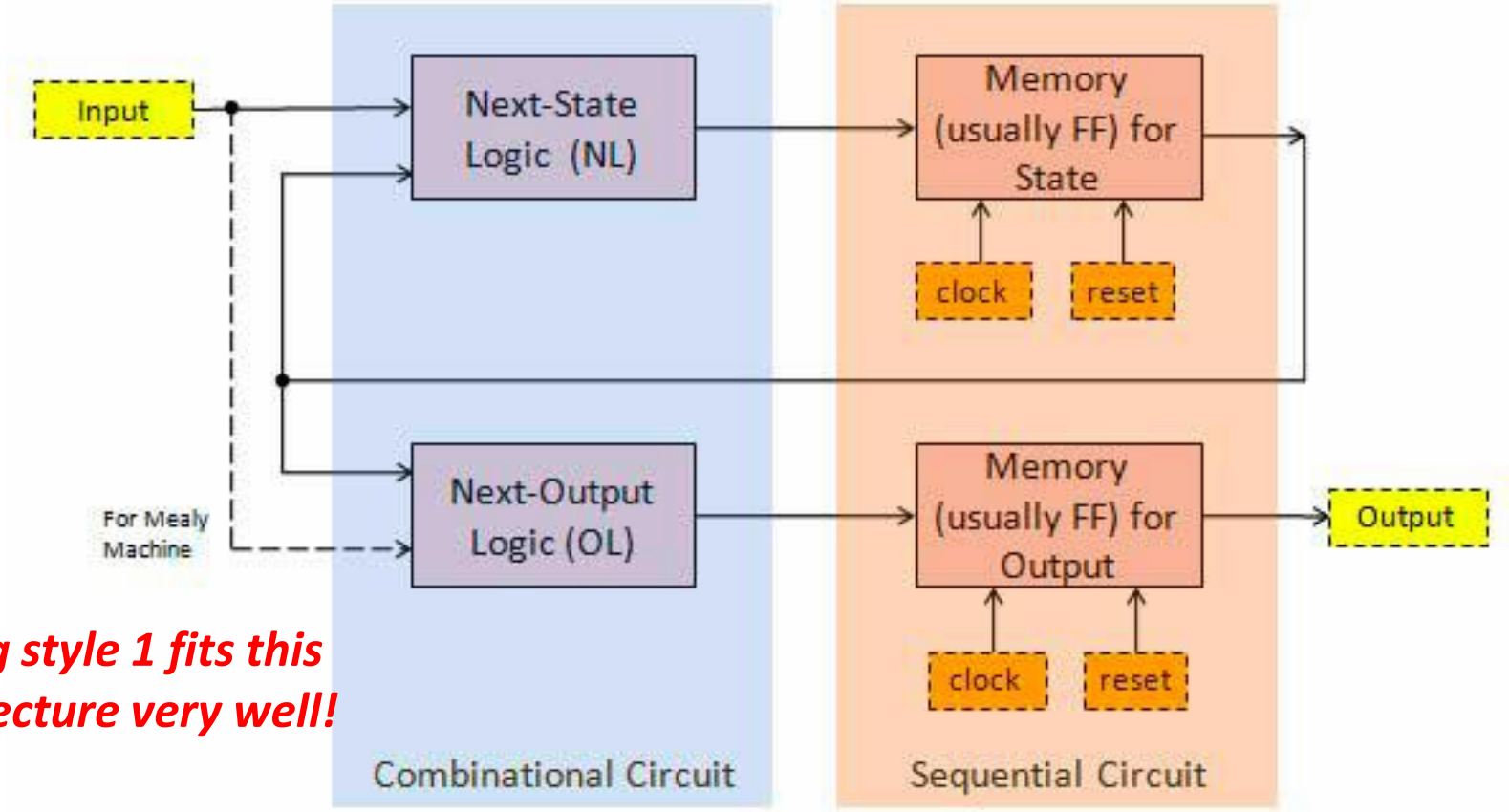
- OL

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

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

Architecture of FSM

Build combinational and sequential parts separately!



Coding style 1 fits this architecture very well!

```
/*=======================================
Author: Yu Chuan, Chuang
Module: Counter
Description:
When getting start_i signal, counter starts
to count from 0 to 15.
=======================================*/
module counter (
    input           clk,
    input           rst,
    input           start_i,
    output  [3:0]   count_o
);

//============  Parameter ==============
    localparam  STATE_IDLE  = 1'b0;
    localparam  STATE_CNT   = 1'b1;

//=========  Reg/Wire Declaration ======
    reg             state, nxt_state;
    reg [3:0]       cnt, nxt_cnt;

//========== Finite State Machine =======
    always@(posedge clk or posedge rst) begin
        if(rst)
            state <= STATE_IDLE;
        else
            state <= nxt_state;
    end

    always@(*) begin
        case(state)
            STATE_IDLE: begin
                if(start_i)
                    nxt_state = STATE_CNT;
                else
                    nxt_state = STATE_IDLE;
            end
            STATE_CNT: begin
                if(cnt == 4'd15)
                    nxt_state = STATE_IDLE;
                else
                    nxt_state = STATE_CNT;
            end
            default: nxt_state = STATE_IDLE;
        endcase
    end

//============ Combinational ===========
    assign count_o = cnt;

    always@(*) begin
        if(state == STATE_CNT) begin
            nxt_cnt = cnt + 1;
        end
        else begin
            nxt_cnt = 0;
        end
    end

//============= Sequential =============
    always@(posedge clk or posedge rst) begin
        if(rst)
            cnt <= 0;
        else
            cnt <= nxt_cnt;
    end

endmodule
```

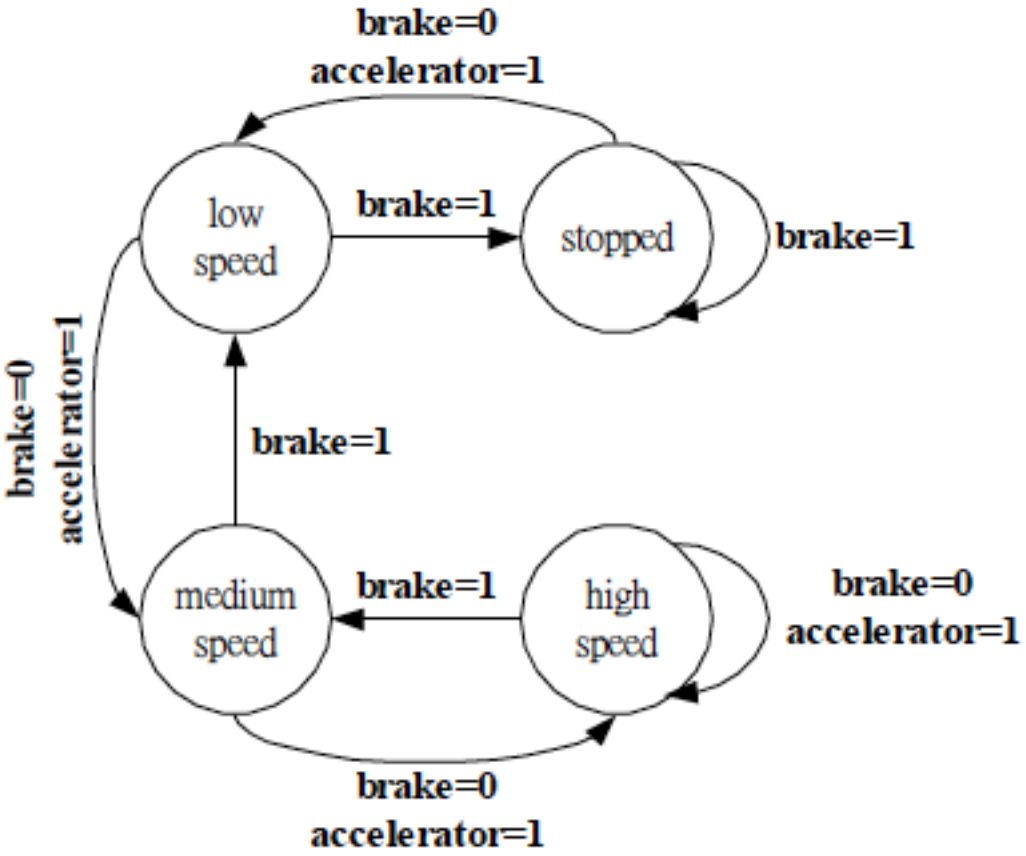
FSM

CS

NL

OL

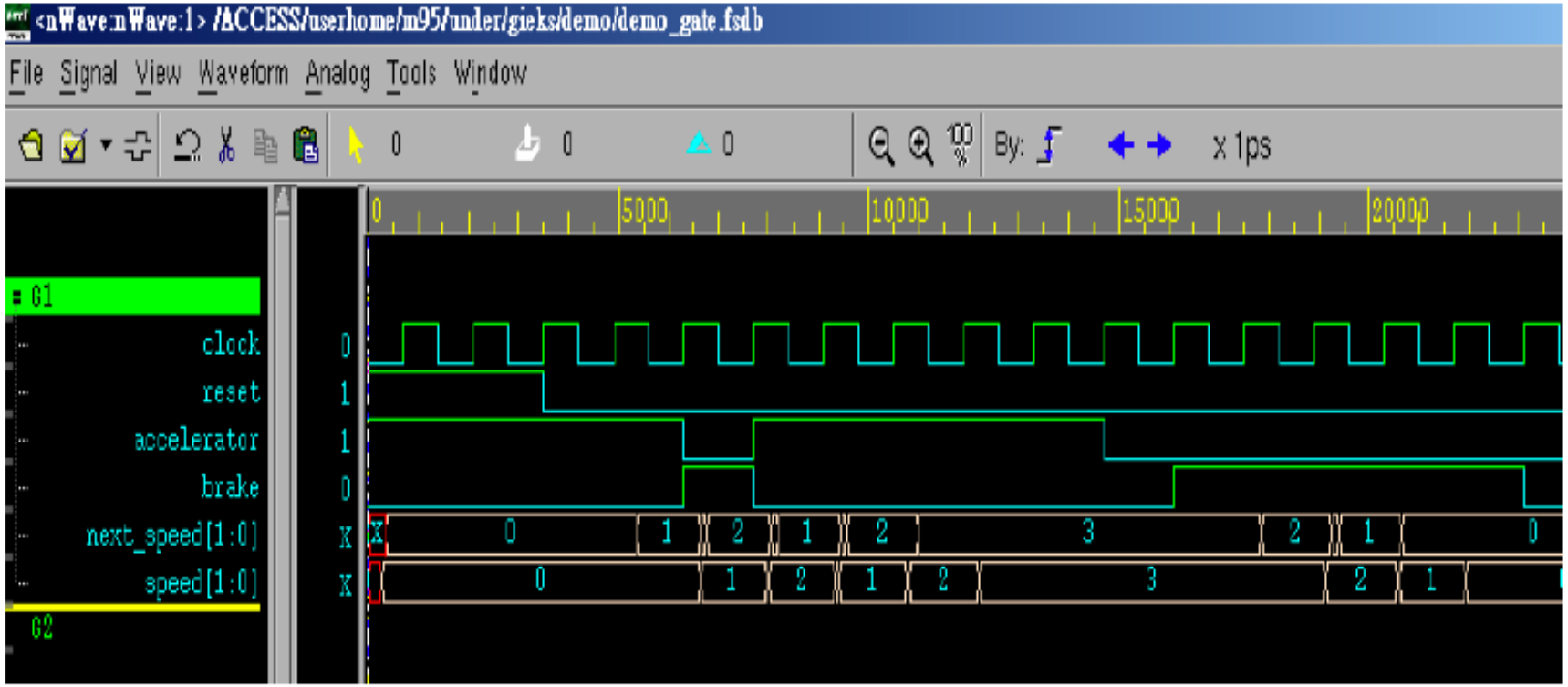
FSM Example: Speed Machine



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 Example: Waveform



FSM Design Notice

- Partition FSM and non-FSM logic
- Partition combinational part and sequential part
  - Coding style 1 is preferred
- Use parameter to define names of the state vector
- Assign a default (reset) state

Coding Overview

- Initialization
  - Input / output definition, wire / reg declaration
- Finite state machine (FSM)
  - State define (using localparameter)
  - Separating CS, NL, OL
- Combinational circuit
- Sequential circuit (only declare DFF !!!)

Outline

- Procedural Construct and Assignment
  - initial block
  - always block
  - Procedural assignment
- Control Structure
- Finite State Machine (FSM)
  - Moore Machine & Mealy Machine
  - Modeling of FSM
- Loop Statements
- Functional Block
  - Sub-modules
  - function and task

Looping Statements

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

- Execute **inside Procedure Block!!!!!!**

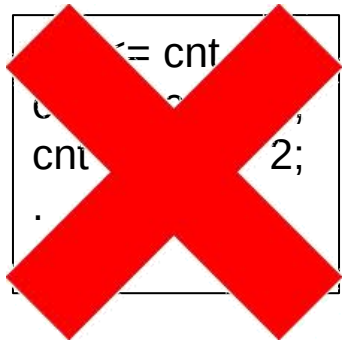
For Loop

- 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

```
module bitwise_and(a, b, out)
  parameter size = 2;
  input [size-1:0] a, b;
  output reg [size-1:0] out;

  integer i;
  always@(*) begin
    for(i=0; i<size; i=i+1)
      out[i] = a[i] & b[i]
  end
endmodule
```

=>

```
out[0] = a[0] & b[0]
out[1] = a[1] & b[1]
out[2] = a[2] & b[2]
```



<=

```
reg [7:0] cnt;
integer i;

always@(posedge clk) begin
  for(i=0; i<10; i=i+1)
    cnt <= cnt + i;
end
```

How? ➔ reference p.41

While Loop

- The **`while`** loop executes until the **`while`**-expression becomes false
- Not synthesizable

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

```
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.

```
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

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

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

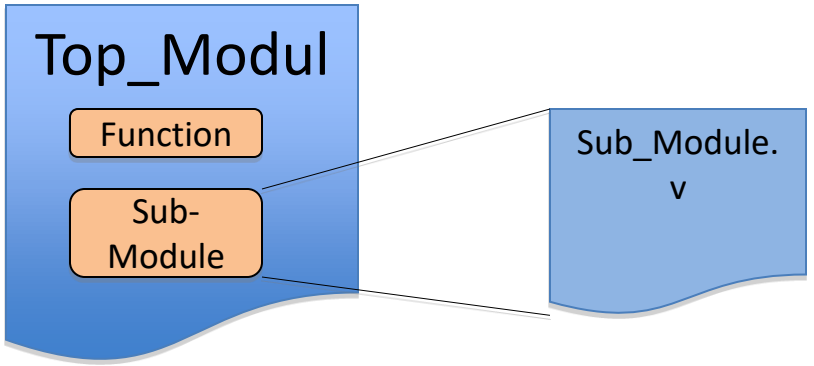
```
//Synchronize 2 register values
//at every positive edge of clock
reg clk;
reg x,y;

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

Outline

- Procedural Construct and Assignment
  - initial block
  - always block
  - Procedural assignment
- Control Structure
- Finite State Machine (FSM)
  - Moore Machine & Mealy Machine
  - Modeling of FSM
- Loop Statements
- Functional Block
  - Sub-modules
  - function and task

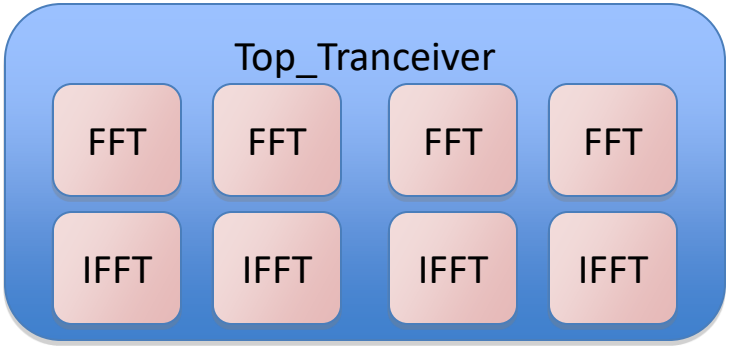
Functional Partition

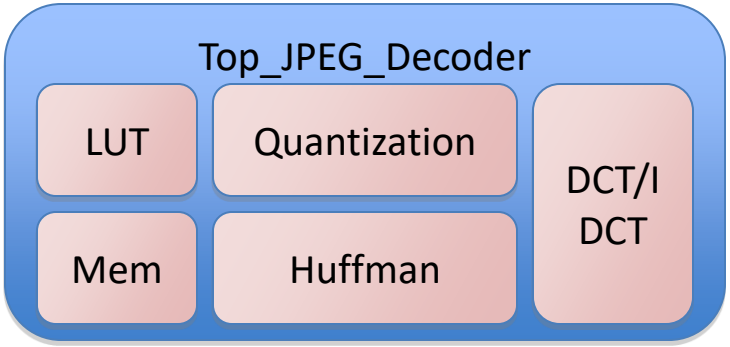
- A digital system may consist of many function blocks
  - A lumped Verilog module makes debugging & editing a great disaster
- Break the whole system into several function blocks
  - Sub-module
  - Function/Task



Sub-Module (1/2)

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

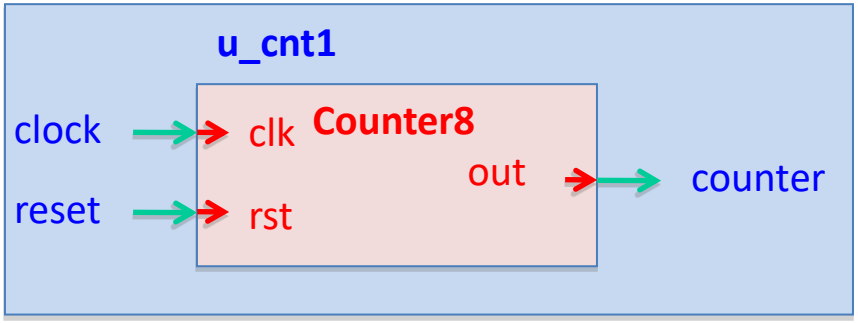




- Note: abusing sub-modules may also makes your design hard to read

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)

Synthesizable

- Functions are used if all of the following conditions are true for the procedure
  - functions can not include timing delays, like posedge, negedge, # delay, which means that functions should be executed in "zero" time delay
  - functions can have any number of inputs but only one output
  - The order of declaration within the function defines how the variables passed to the function by the caller are used.
  - functions can be used for modeling combinational logic, no always block inside
  - functions can call other functions, but can not call tasks

Function (2/3)

- Syntax
  - A function begins with keyword **function** and ends with keyword **endfunction**
  - **inputs** are declared after the keyword function.
- 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.
- Call function
  - Can be called in always block and continuous assignment
  - Have to assign output to a value, can't exist alone

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 (1/4)

Not Synthesizable

- It can be used when any one of the following conditions is true
  - tasks can include timing delays, like posedge, negedge, # delay and wait.
  - tasks can have any number of inputs and outputs
  - the order of declaration within the task defines how the variables passed to the task by the caller are used.
  - tasks can call another task or function
  - no always block inside

Task (2/4)

- Syntax
  - A task begins with keyword **task** and ends with keyword **endtask**
  - Inputs and outputs are declared after the keyword task.
  - Local variables are declared after input and output declaration.
- A task can be canceled by disabling it
- Call task
  - Must be called in a procedural block (initial or always)
  - Can exist alone

Task (3/4)

- Task example

```
// task definition
task addmult_t;
    input  [3:0] in1, in2;
    output [7:0] outa, outm;
    begin
        outa = in1 + in2;
        outm = in1 * in2;
    end
endtask
// call task
reg [7:0] var1, var2;
always@(*) begin
    addmult_t(inx, iny, var1, var2);
end
```

Task (4/4)

- Use **`disable`** to cancel the task

```
task errmon;
    forever@(poseage 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
A function can enable another function but not another task	A task can enable other tasks and functions
Function always **executes in 0 simulation time**	Tasks may execute in non-zero simulation time
Functions must **not contain any delay**, event, or timing control statements	Tasks may contain delay, event, or timing control statements
Functions must **have at least one input argument**. They can have more than one input	Task may have **zero or more arguments** of type input, output or inout
Functions **always return a single value**. They **cannot have output or inout** arguments	Tasks do not return with a value but can pass multiple values through output and inout arguments