Verilog HDL – II
Sequential Logics

黃元豪
Yuan-Hao Huang

國立清華大學電機工程學系
Department of Electrical Engineering
National Tsing-Hua University

Outline

- Verilog Coding for Sequential Logics
- Behavior Modeling
- Examples of Sequential Circuits

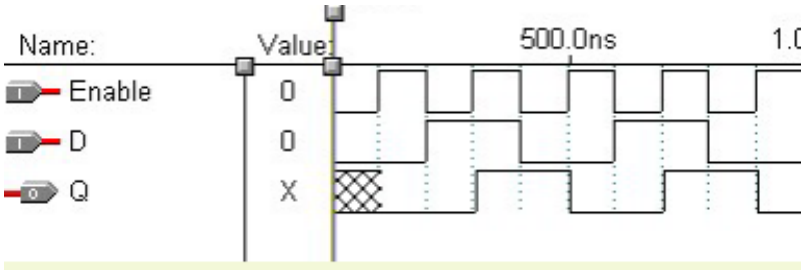
Sequential Logic Circuits

- Memory Devices
  - Latch
  - Flip-flop
  - Register
- Sequential Circuits
  - Synchronous Counters
  - General Synchronous Sequential Circuits
- Finite State Machine
  - Moore Machine
  - Mealy Machine

Latch

- Latch is an level-triggered storage with a trigger signal enable.
- It is suggested not to use the level-triggering latch in the experiment.
  - Transparent Q

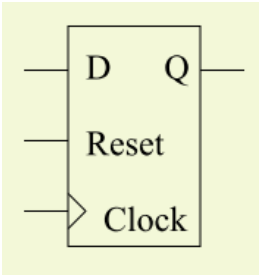
```
module latch_d(Enable, D, Q);
input  Enable, D;
output Q;
reg    Q;

always@(Enable or D)
if (Enable)
Q = D;

endmodule
```

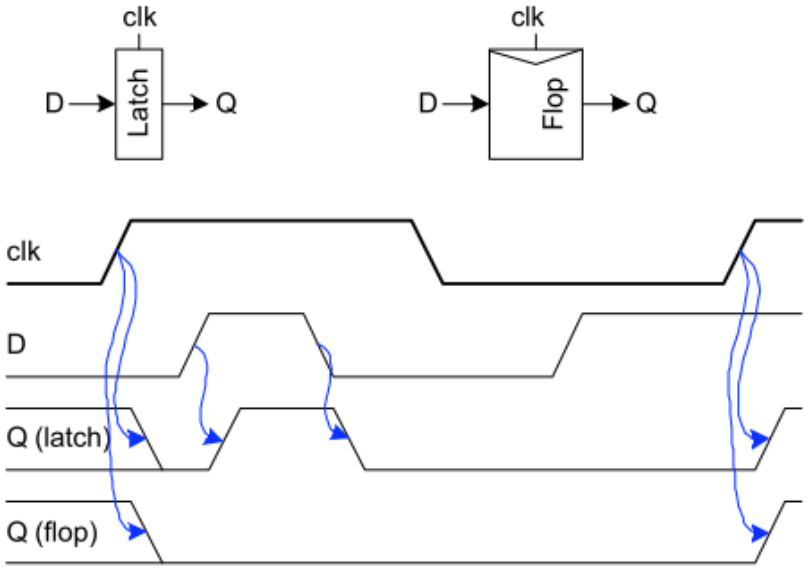


Flip-Flop

- Flip-flop with synchronous clock and asynchronous reset
  - Opaque flip-flop

```
module D_FF_SR(clock, reset, D, Q);
input  clock, reset, D;
output Q;
reg    Q;

always @(posedge clock or negedge reset)
begin
if (reset == 1'b0)
Q = 1'b0;  // reset
else
Q = D;
end

endmodule
```

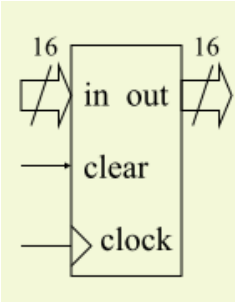




Register

- 16-bit synchronous register with clear

```
module reg_16(clear, clock, in, out);
input     clear, clock;
input     [0:15] in;
output    [0:15] out;
reg       [0:15] out;

always@(posedge clock or posedge clear)
begin
if (clear == 1'b1)
out = 16'b0;
else out = in;
end
endmodule
```



Synchronous Counter

- It is recommended that the logic circuits and the flip-flops are coded separately.

Logic circuits and flip-flops are jointly coded.

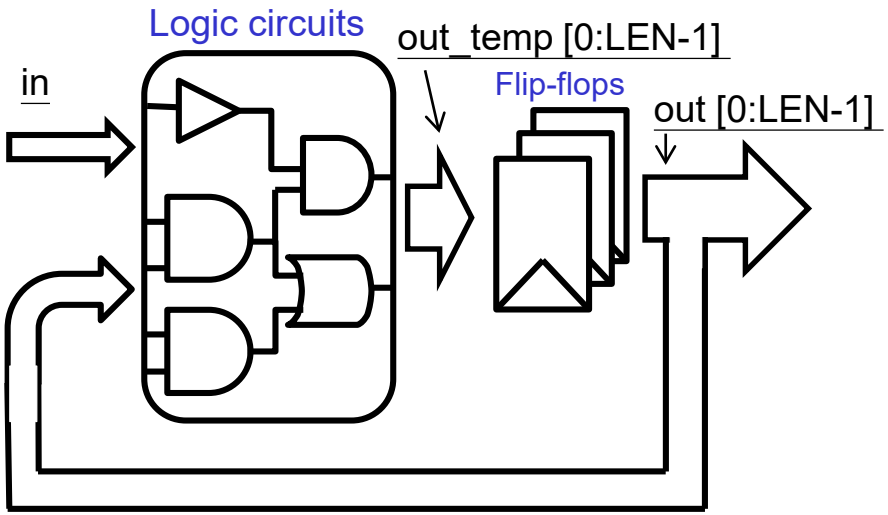
```
module counter1(direct, clk, reset, out);
input      direct, clk, reset;
output     [0:3] out;
reg        [0:3] out;

always@(posedge clk or negedge reset) begin
if (~reset)
out <= 4'b0000;
else
    begin
    if (direct)
    out <= out + 1;
    else
    out <= out -1;
    end
end
endmodule
```

Logic circuits and flip-flops are separately coded.

```
module counter1(direct, clk, reset, out);
input      direct, clk, reset;
output     [0:3] out;
reg        [0:3] out;
wire     [0:3] out_temp;

// Logic circuits
assign out_temp = (direct) ?  out+1 : out-1;

// Flip-flops
always@(posedge clk or nededge reset) begin
If(~reset)
out <= 4'b0000;
else
out <= out_temp;
end

endmodule
```

Generalized Sequential Circuits

- Recommended coding styles
  - Separate coding of combinational logics and flip-flops

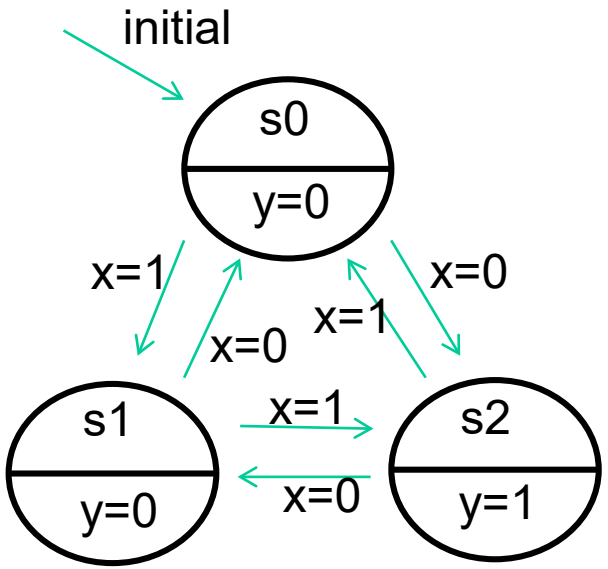



```
reg       [0:LEN-1] out;
reg    [0:LEN-1] out_temp;

// Logic circuits
always @(in1 or in2 or …)
out_temp <= combinational_circuits_codes;

// Flip-flops
always@(posedge clk or nededge reset) begin
If(~reset)
out <= 0;
else
out <= out_temp;
end
```

```
reg       [0:LEN-1] out;
wire    [0:LEN-1] out_temp;

// Logic circuits
assign out_temp = combinational_circuits_codes;
// Flip-flops
always@(posedge clk or nededge reset) begin
If(~reset)
out <= 0;
else
out <= out_temp;
end
```

Finite State Machine

- Moore Machine
  - Output y depends on the current state, not input x.
  - Output y is synchronized with the state s.



Current State	Next State		output y
	Input x=0	Input x=1	
s0	s2	s1	0
s1	s0	s2	0
s2	s1	s0	1

Moore Machine

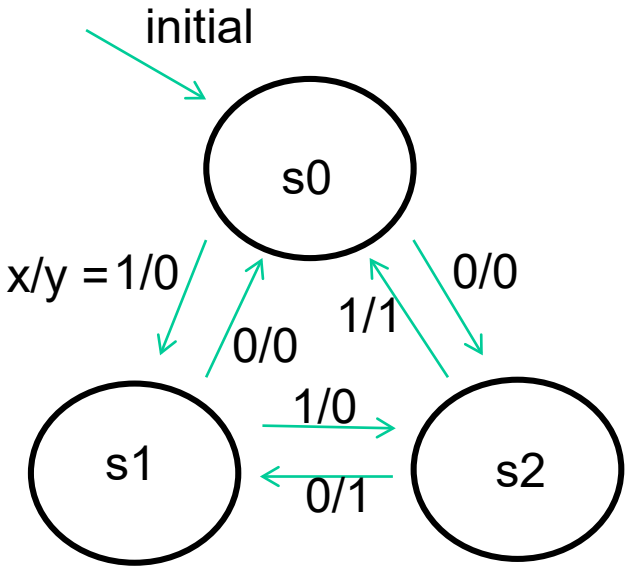
- Moore Machine Coding Style

```
module moore(reset, clk, x, y);
input reset, clk, x;
output  y;
reg     y;
parameter  s0=2'b00, s1=2'b01, s2=2'b10;
reg  [0:1] ps, ns;

always@(reset or ps)
begin
   if (reset)
   begin
      ns = s0; y = 0;
   end
   else
   begin
      case(ps)
```

```
s0: begin
       if (x == 0)
          ns = s2;
       else ns = s1;
      y = 0;
     end
s1: begin
       if (x == 0)
          ns = s0;
        else ns = s2;
      y = 0;
     end
……
endcase
end end // else & always

always@(posedge clk or negedge reset)
If (~reset)
    ps<= s0;
else
    ps<= ns;
endmodule
```

Finite State Machine

- Mealy Machine
  - Output y depends on both the current state and input x.
  - Output y is synchronized with the input x.



Current State	Next State		Output y	
	Input x=0	Input x=1	Input x=0	Input x=1
s0	s2	s1	0	0
s1	s0	s2	0	0
s2	s1	s0	1	1

Mealy Machine

- Mealy Machine Coding Style

```
module mealy(reset, clk, x, y);
input reset, clk, x;
output  y;
reg     y;
parameter  s0=2'b00, s1=2'b01, s2=2'b10;
reg  [0:1] ps, ns;
always@(reset or ps)
begin
   if (reset)
   begin
      ns = s0; y = 0;
   end
   else
   begin
```

```
    case(ps)
      s0: begin
            if (x == 0)
               ns = s2;
            else ns = s1;
            if (x == 0)
               y = 0;
            else y = 0;
          end
       s1: begin
……
      endcase
end end // else & always

always@(posedge clk or negedge reset)
If (~reset)
    ps<= s0;
else
    ps<= ns;
endmodule
```

Outline

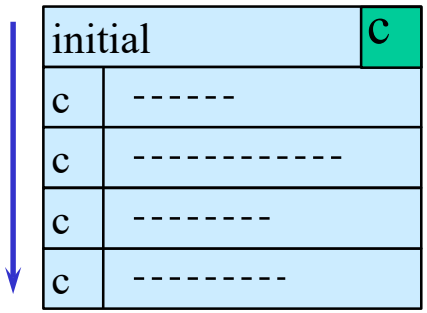
- Verilog Coding for Sequential Logics
- Behavior Modeling
- Examples of Sequential Circuits

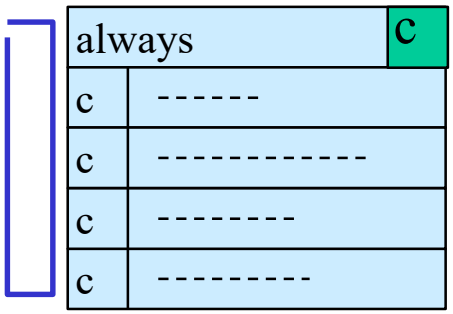
Behavior Modeling

- In behavior modeling, you must specify your circuit's
  - Action
    - How to model the circuit's behavior
  - Timing control
    - Timing
    - Condition
- Verilog supports the following structures for behavior modeling
  - Procedural block
  - Procedural assignment
  - Timing control
  - Control statement

Procedural Blocks

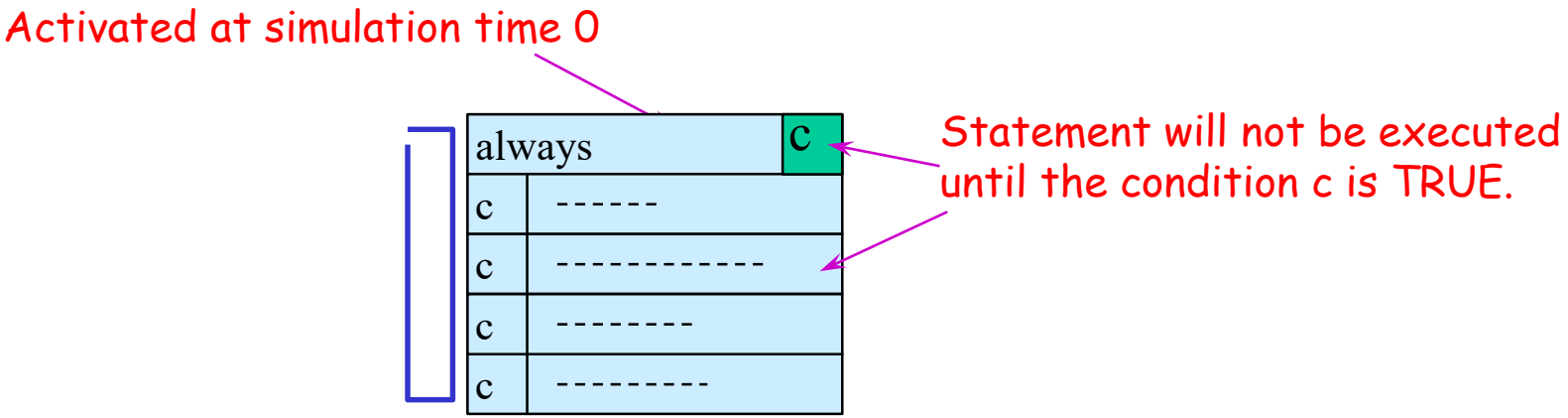
- In Verilog, procedural blocks are basis of behavior modeling
- Procedural blocks are of two types
  - initial procedural block, which executes only once
  - always procedural block, which executes in a loop





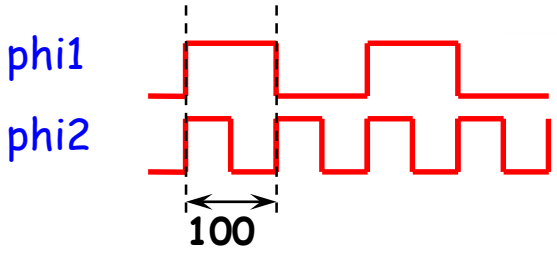
Procedural Blocks

- All procedural blocks are activated at simulation time 0.
  - The block will not be executed until the enabling condition evaluates TRUE.
  - Without the enabling condition, the block will be executed immediately.



Procedural Blocks

```
module clock_gen(phi1,phi2);
output  phi1,phi2;
reg     phi1,phi2;

initial
begin
        phi1=0;phi2=0
end

always
        #100 phi1=~phi1;

always  @(posedge phi1)
begin
        phi2=1;
        #50         phi2=0;
        #50         phi2=1;
        #50         phi2=0;
end
endmodule
```



This procedural block is activated and executed at simulation time 0

This procedural block is activated at simulation time 0 and is always executed

This procedural block is activated at simulation time 0 but executed at positive edge of phi1

Procedural Blocks

- Three components
  - Procedural assignment statements
  - High-level programming language constructs
  - Timing controls
- Using the first two components to model the actions of the circuit.
- Using timing controls to model when should these actions happen.

Procedural Timing Control

- Three types
  - Simple delay control
    - #50 clk=~clk;
  - Event control
    - @(a or b or ci) sum=a+b+ci;
    - @(posedge clk) q=d;
  - Level-sensitive timing control

Block Statements

- Group two or more statements together
  - Sequential blocks
    - Enclosed by keyword **begin** and **end**
  - Parallel blocks
    - Enclosed by keyword **fork** and **join**

```
begin
    #10 out='d10;
    #10 out='d43;
    #10 out='d25;
    #10 out='d86;
end
```

Equivalent

```
fork
    #10 out='d10;
    #20 out='d43;
    #30 out='d25;
    #40 out='d86;
join
```

Blocking and Non-blocking Assignments

- Procedural assignments update the value of register under the control of the procedure flow constructs.
- Blocking procedure assignment

  Basic form : <lvalue> = <timing_control> <expression>
- Non-Blocking procedure assignment

  Basic form : <lvalue> <= <timing_control> <expression>

```
initial begin
   a=0;
   b=1;
   c=0;
end
always c = #5 ~c;
```

```
always @(posedge c)
begin
   a=b;  // 1
   b=a;  // 1
end
```

Blocking

```
always @(posedge c)
begin
   a<=b;    // 1
   b<=a;    // 0
end
```

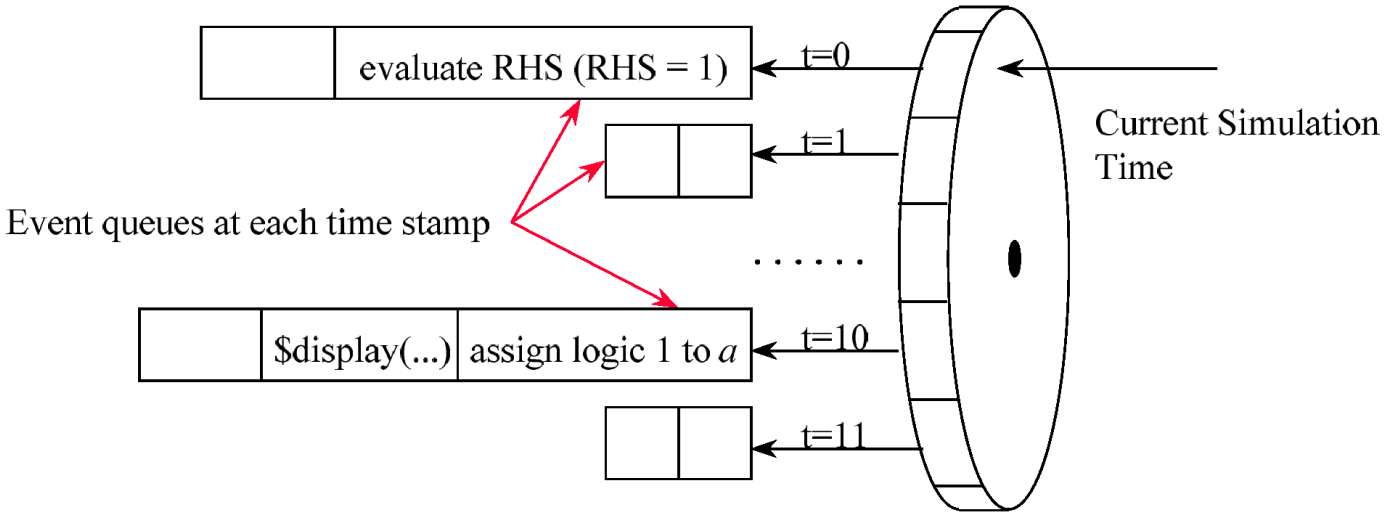
Non-Blocking

Blocking and Non-blocking Assignments

- Blocking assignments
  - Evaluate the RHS expression and **stores** the value in the LHS register immediately

```
initial begin
  a = #10 1;
  $display("current time = %t  a = %b", $time, a);
  end
```

simulation result → current time = 10, a = 1

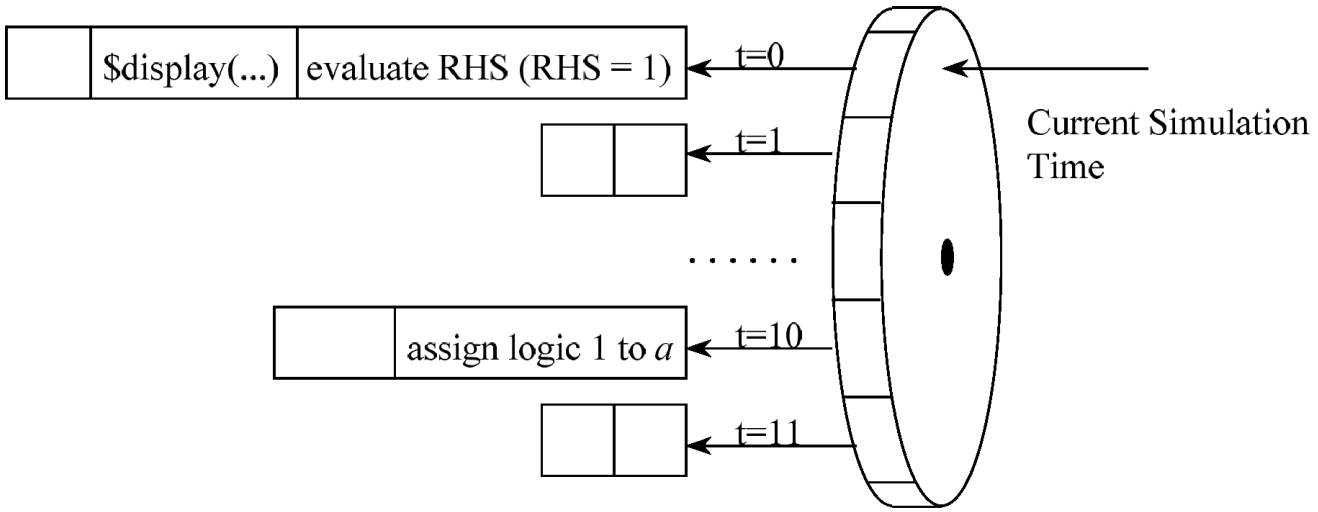


Blocking and Non-blocking Assignments

- Non-blocking assignments
  - It evaluate the RHS expression and **schedules** to update the value in the LHS register
  - It updates the LHS only after evaluating all the RHS

```
initial begin
  a <= #10 1;
  $display("current time = %t  a = %b", $time, a);
  end
```

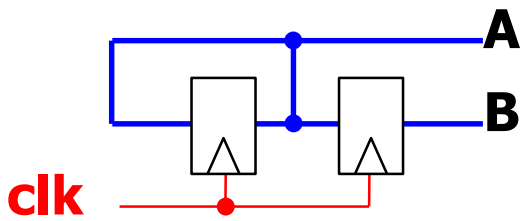
simulation result → current time = 0, a = x



Blocking and Non-blocking Assignments

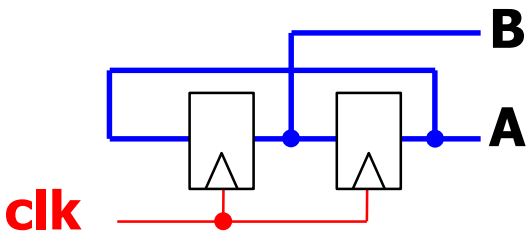
Bad: Circuit from blocking assignment.

```
always @(posedge clk)
begin
    b=a;
    a=b;
end
```



Good: Circuit from nonblocking assignment.

```
always @(posedge clk)
begin
    b<=a;
    a<=b;
end
```

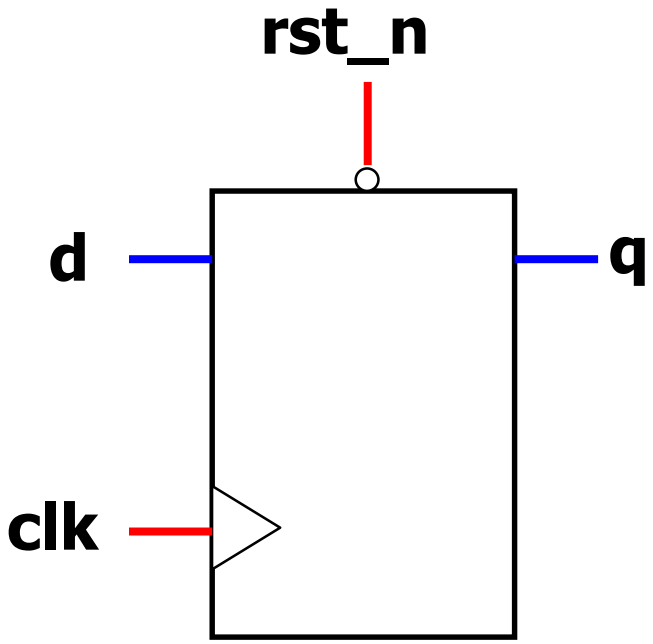


Outline

- Verilog Coding for Sequential Logics
- Behavior Modeling
- Examples of Sequential Circuits

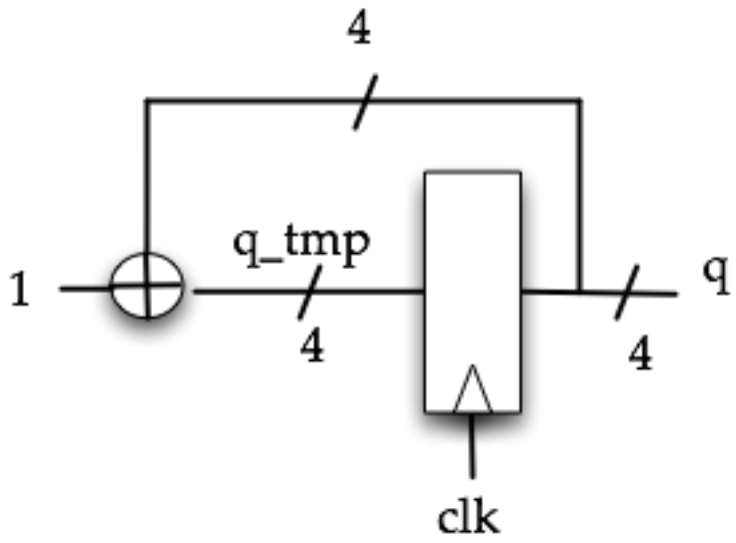
D-type Flip Flop

```
module dff(
  q,  // output
  d,  // input
  clk,  // global clock
  rst_n // active low reset
);

output q;  // output
input d;  // input
input clk;  // global clock
input rst_n;  // active low reset

reg q; // output (in always block)

always @(posedge clk or negedge
rst_n)
  if (~rst_n)
    q<=0;
  else
    q<=d;

endmodule
```



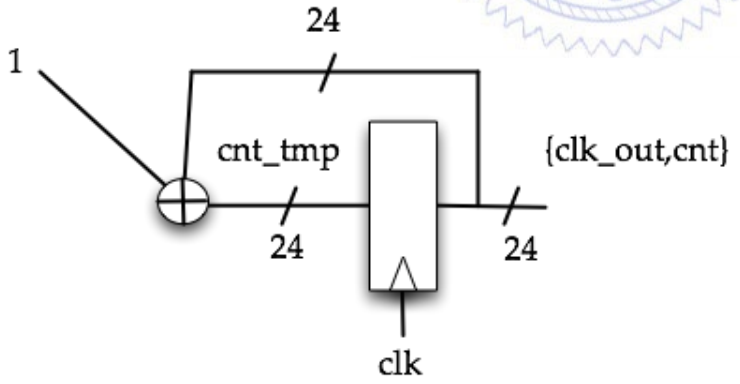
Binary Up Counter

```
`define BCD_BIT_WIDTH 4
`define BCD_ZERO 4'd0
`define BCD_ONE 4'd1
`define BCD_NINE 4'd9
module bcdcounter(
  q,  // output
  clk,  // global clock
  rst_n  // active low reset
);

output [`BCD_BIT_WIDTH-1:0] q;  // output
input clk;  // global clock
input rst_n;  // active low reset

reg [`BCD_BIT_WIDTH-1:0] q; // output (in always block)
reg [`BCD_BIT_WIDTH-1:0] q_tmp; // input to dff (in always block)

// Combinational logics
always @(q)
  q_tmp = q + `BCD_ONE;

// Sequential logics: Flip flops
always @(posedge clk or negedge rst_n)
  if (~rst_n) q<=`BCD_BIT_WIDTH'd0;
  else q<=q_tmp;

endmodule
```



Frequency Divider

```
`define FREQ_DIV_BIT 24
module freq_div(
  clk_out,  // divided clock output
  clk,  // global clock input
  rst_n  // active low reset
);

output clk_out;  // divided output
input clk;  // global clock input
input rst_n;  // active low reset

reg clk_out; // clk output (in always block)
reg [`FREQ_DIV_BIT-2:0] cnt; // remainder of the counter
reg [`FREQ_DIV_BIT-1:0] cnt_tmp; // input to dff (in always block)

// Combinational logics: increment, neglecting overflow
always @(clk_out or cnt)
  cnt_tmp = {clk_out,cnt} + 1'b1;

// Sequential logics: Flip flops
always @(posedge clk or negedge rst_n)
  if (~rst_n) {clk_out, cnt}<=`FREQ_DIV_BIT'd0;
  else {clk_out,cnt}<=cnt_tmp;

endmodule
```

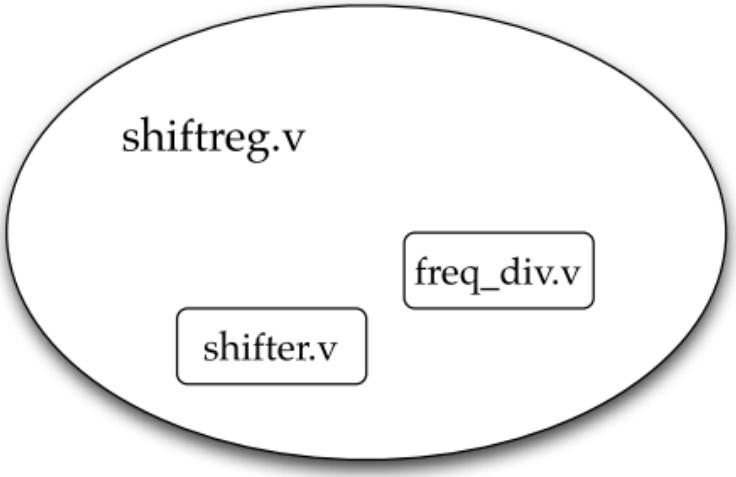


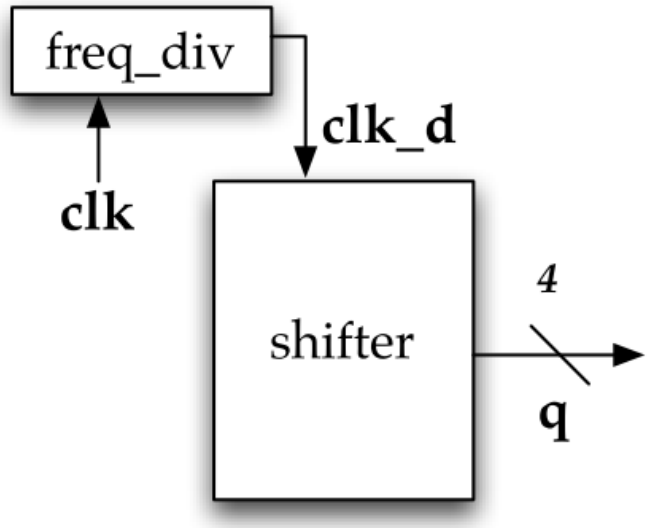
cnt_tmp[23:0]

cnt[22:0]

Modularized Shift Register Design

Shift Register

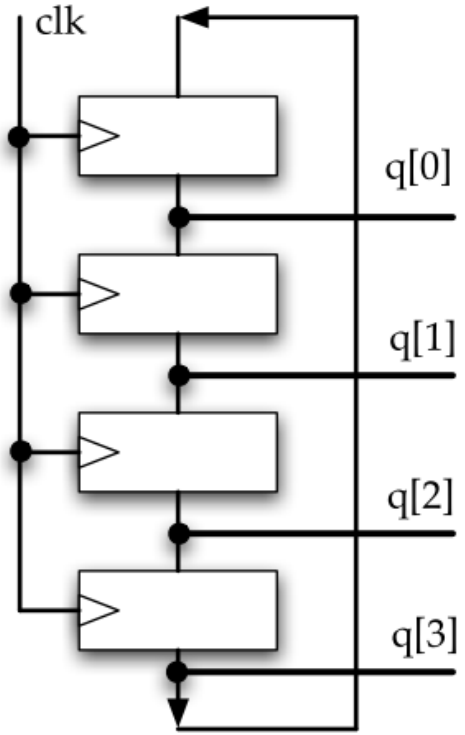




shifter.v

```
`define BIT_WIDTH 4
module shifter(
  q,  // shifter output
  clk,  // global clock
  rst_n  // active low reset
);

output [`BIT_WIDTH-1:0] q;  // output
input clk;  // global clock
input rst_n;  // active low reset

reg [`BIT_WIDTH-1:0] q;  // output

// Sequential logics: Flip flops
always @(posedge clk or negedge rst_n)
  if (~rst_n)
  begin
    q<=`BIT_WIDTH'b0101;
  end
  else
 begin
    q[0]<=q[3];
    q[1]<=q[0];
    q[2]<=q[1];
    q[3]<=q[2];
  end
endmodule
```

Initial value q = 0101



shiftreg.v

```
`define BIT_WIDTH 4
module shift_reg(
  q,  // LED output
  clk, // global clock
  rst_n  // active low reset
);

output [`BIT_WIDTH-1:0] q;  // LED output
input clk;  // global clock
input rst_n;  // active low reset

wire clk_d; // divided clock
wire [`BIT_WIDTH-1:0] q;  // LED output
```

```
// Insert frequency divider (freq_div.v)
freq_div U_FD(
  .clk_out(clk_d),  // divided clock output
  .clk(clk),  // clock from the crystal
  .rst_n(rst_n)  // active low reset
);

// Insert shifter (shifter.v)
shifter U_D(
  .q(q),  // shifter output
  .clk(clk_d),  // clock from the frequency divider
  .rst_n(rst_n)  // active low reset
);

endmodule
```