Verilog HDL语言

主讲：李榕

Email: lr@hust.edu.cn

QQ: 179825425

华中科技大学计算机科学与技术学院

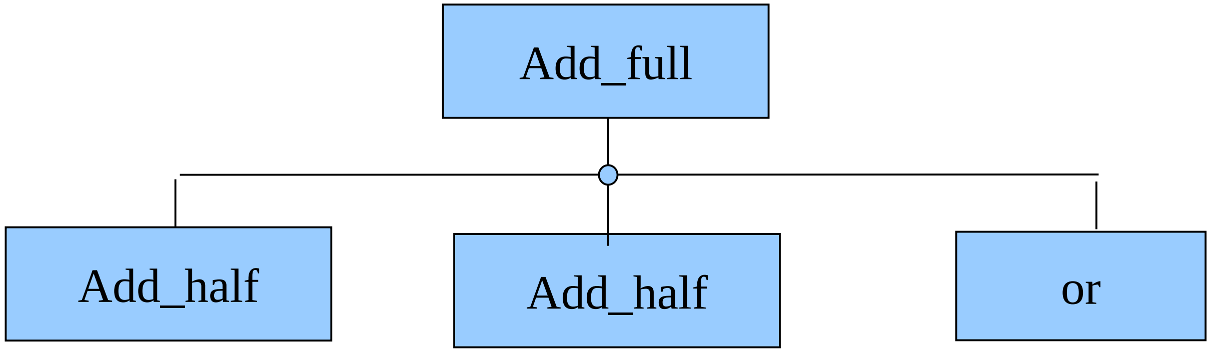
Verilog组合电路设计

- 运算器
- 编码器（带优先级）
- 译码器
- 数据选择器
- 数据分配器
- 数值比较器

```
// 1位全加器

module fa_rtl (A, B, CI, S, CO) ;

input A, B, CI ;
output S, CO ;

// use continuous assignments
assign S = A ^  B ^ CI;
assign C0 = (A & B) | (A & CI) | (B & CI);

endmodule
```

```
// 1位全加器

module fa_bhv (A, B, CI, S, CO) ;

input A, B, CI;
output S, CO;
reg S, CO;                // assignment made in an always block
                          // must be made to registers
// use procedural assignments
always@(A or B or CI)
   begin
        S = A ^ B ^ CI;
        CO = (A & B) | (A & CI) | (B & CI);
    end
endmodule
```

```
// 1位全加器

module Add_full(c_out, sum, a, b, c_in) ;
    output sum, c_out;
    input a, b, c_in;
    wire w1, w2, w3;

    Add_half AH1(.sum(w1), .c_out(w2), .a(a), .b(b));
    Add_half AH2(.sum(sum), .c_out(w3), .a(c_in), .b(w1));
    or carry_bit(c_out, w2, w3);
endmodule
```

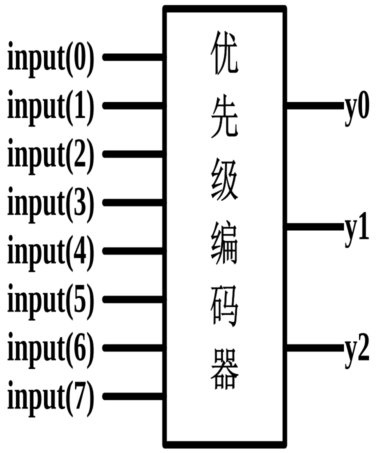


```
// 4位全加器

module full_adder(a, b, c, s);
    input   [3:0]    a, b;
    output [3:0]    s;
    output               c;

    assign {c,s} = a + b;

endmodule
```

```
// 多位全加器

module full_adder(a, b, c, s);
    parameter WIDTH = 8;

    input   [WIDTH-1:0]     a, b;
    output [WIDTH-1:0]     s;
    output                          c;

    assign {c,s} = a + b;

endmodule
```

- **优先级编码器：** 常用于中断的优先级控制。例如，74LS148是一个8输入，3位二进制码输出的优先级编码器，当其某一个输入有效时，就可以输出一个对应的3位二进制编码。另外，当同时有几个输入有效时，将输出优先级最高的那个输入所对应的二进制编码。

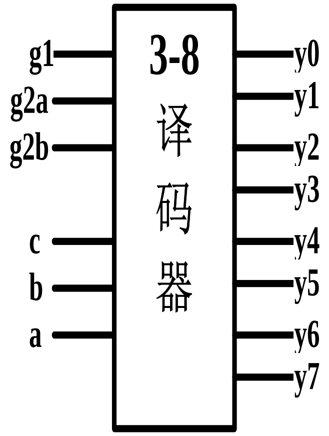
输　　入								二进制编码输出		
input(7)	input(6)	input(5)	input(4)	input(3)	input(2)	input(1)	input(0)	y2	y1	y0
X	X	X	X	X	X	X	0	1	1	1
X	X	X	X	X	X	0	1	1	1	0
X	X	X	X	X	0	1	1	1	0	1
X	X	X	X	0	1	1	1	1	0	0
X	X	X	0	1	1	1	1	0	1	1
X	X	0	1	1	1	1	1	0	1	0
X	0	1	1	1	1	1	1	0	0	1
X	1	1	1	1	1	1	1	0	0	0



```
module Priority_Encoder(in, y);
    input         [7:0]     in;
    output reg [2:0]     y;

    always @(in) begin
        if(in[0]==1'b0)                   y = 3'b111;
        else if(in[1]==1'b0)          y = 3'b110;
        else if(in[2]==1'b0)          y = 3'b101;
        else if(in[3]==1'b0)          y = 3'b100;
        else if(in[4]==1'b0)          y = 3'b011;
        else if(in[5]==1'b0)          y = 3'b010;
        else if(in[6]==1'b0)          y = 3'b001;
        else                                   y = 3'b000;
    end
endmodule
```

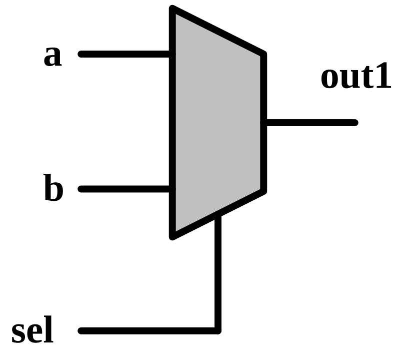
- **3-8译码器：3-8译码器是最常用的一种小规模集成电路。对输入a、b、c的值进行译码，就可以确定输出端y0～y7的哪一个输出端变为有效，从而达到译码的目的。**

选 通 输 入			二进制输入端			译 码 输 出 端							
g1	g2a	g2b	c	b	a	y0	y1	y2	y3	y4	y5	y6	y7
X	1	X	X	X	X	1	1	1	1	1	1	1	1
X	X	1	X	X	X	1	1	1	1	1	1	1	1
0	X	X	X	X	X	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	0	1	1	0	1	1	1	1	1	1
1	0	0	0	1	0	1	1	0	1	1	1	1	1
1	0	0	0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	0	0	1	1	1	1	0	1	1	1
1	0	0	1	0	1	1	1	1	1	1	0	1	1
1	0	0	1	1	0	1	1	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	1	1	1	0



```
module decoder_38(out,in,g1,g2a,g2b);
    output reg [7:0] out;
    input        [2:0] in;
    input              g1,g2a,g2b;
    always @(in,g1,g2a,g2b) begin
        if(g1==1'b1 && g2a==1'b0 && g2b==1'b0)
            case(in)
                3'd0: out=8'b11111110;
                3'd1: out=8'b11111101;
                3'd2: out=8'b11111011;
                3'd3: out=8'b11110111;
                3'd4: out=8'b11101111;
                3'd5: out=8'b11011111;
                3'd6: out=8'b10111111;
                3'd7: out=8'b01111111;
            endcase
    end
endmodule
```

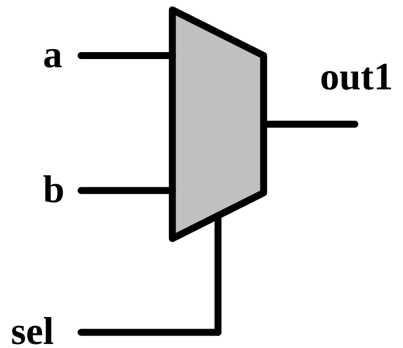
❑ 1位数据宽度2选1多路选择器



```
module mux2_1(out1, a, b, sel) ;
    output  reg out1;
    input  a, b;
    input sel;
always @(sel or a or b)
begin
    if (sel)
        out1 = b;
    else
        out1 = a;
end
endmodule
```

```
module mux2_1(out1, a, b, sel) ;
    output  reg out1;
    input  a, b;
    input sel;
always @(sel or a or b)
begin
    case (sel)
        1’b0 :  out1 = a;
        1’b1 :  out1 = b;
    endcase
end
endmodule
```

❑ 多位数据宽度2选1多路选择器



```
module mux2_1(out1, a, b, sel) ;
    output  reg [3:0] out1;
    input          [3:0] a, b;
    input                    sel;
always @(sel or a or b)
begin
    if (sel)
        out1 = b;
    else
        out1 = a;
end
endmodule
```

```
module mux2_1(out1, a, b, sel) ;
    output  reg [3:0] out1;
    input          [3:0] a, b;
    input                    sel;
always @(sel or a or b)
begin
    case (sel)
        1'b0 :  out1 = a;
        1'b1 :  out1 = b;
    endcase
end
endmodule
```

❑ 4选1多路选择器（用if-else 语句描述）

```
module MUX4_1(out, in0, in1, in2, in3, sel);
    output out;
    input in0, in1, in2, in3;
    input[1:0] sel;
    reg out;

    always @ (in0 or in1 or in2 or in3 or sel)
    begin
        if (sel==2'b00)         out=in0;
        else if (sel==2'b01) out=in1;
        else if (sel==2'b10) out=in2;
        else                           out=in3;
    end
endmodule
```

❑ 4选1多路选择器（用case 语句描述）

```
module MUX4_1(out, in0, in1, in2, in3, sel);
    output out;
    input in0,in1,in2,in3;
    input[1:0] sel;
    reg out;

    always @ (in0 or in1 or in2 or in3 or sel)
    begin
        case(sel)
            2'b00: out=in0;
            2'b01: out=in1;
            2'b10: out=in2;
            default: out=in3;
        endcase
    end
endmodule
```

```
// 数值比较器

module comparator(A, B, is_equal, is_great, is_less);
    parameter WIDTH = 8;
    input [WIDTH-1:0] A, B;
    output is_equal, is_great, is_less;

    assign is_equal = (A==B)? 1'b1 : 1'b0;
    assign is_great = (A>B)?    1'b1 : 1'b0;
    assign is_less    = (A<B)?    1'b1 : 1'b0;
endmodule
```

Verilog时序电路设计

- 触发器（锁存器）
- 寄存器
- 计数器
- 存储器

■ 基本D触发器及其Verilog表述

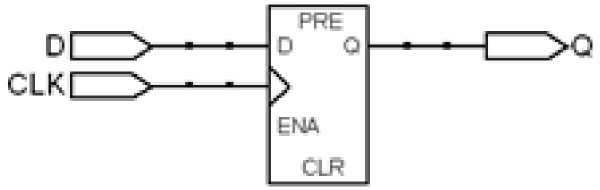


图 5-1 边沿触发型 D 触发器

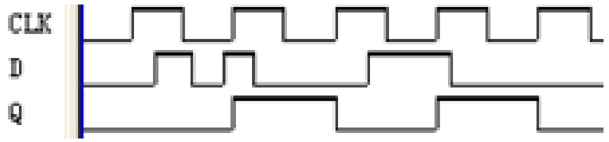


图 5-2 D 触发器时序波形

【例 5-1】

```
module DFF1(CLK,D,Q);
    output Q ;
    input  CLK, D ;
     reg Q;
  always @(posedge CLK )
     Q <= D;
Endmodule
```

■ 含异步复位和时钟使能的D触发器及其Verilog表述

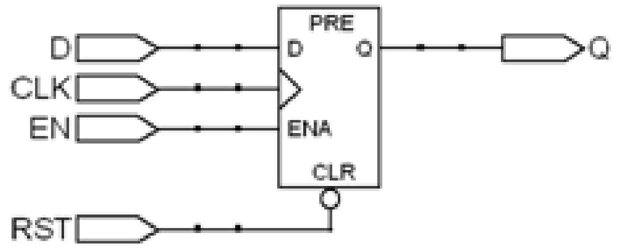


图 5-4 含使能和复位控制的 D 触发器

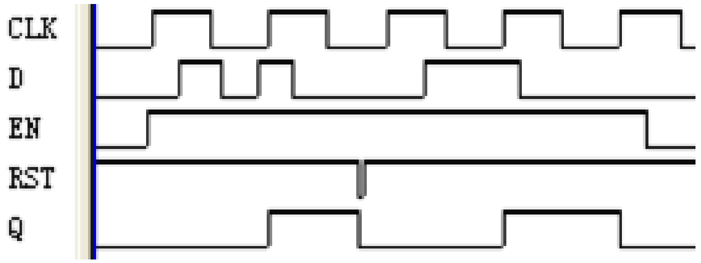


图 5-5 图 5-4 的 D 触发器的时序图

【例 5-4】

```
module DFF2(CLK,D,Q,RST,EN);
  output Q;
  input CLK,D,RST,EN;
  reg Q;
always @(posedge CLK or negedge RST)
   begin
        if (!RST)  Q <= 0;
   else if (EN)    Q <= D;
   end
endmodule
```

■ 含同步复位控制的D触发器及其Verilog表述

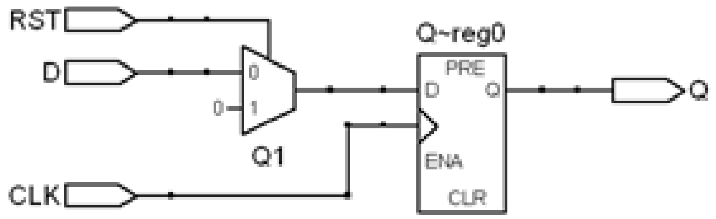


图 5-6 含同步清 0 控制的 D 触发器

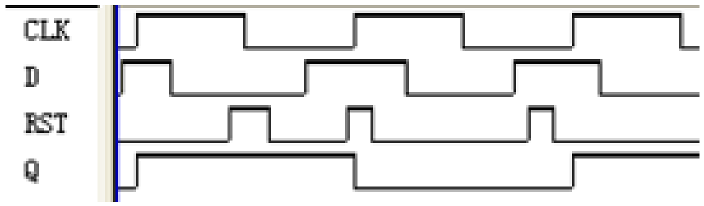


图 5-7 含同步清 0 控制 D 触发器的时序图

【例 5-5】
```
module DFF3(CLK,D,Q,RST);
   output Q ;
   input CLK,D,RST ;
   reg Q;
   always @(posedge CLK )
        if (RST==1)  Q = 0;
  else if (RST==0)  Q = D;
endmodule
```

【例 5-6】
```
module DFF1(CLK,D,Q,RST);
    output Q;  input CLK,D,RST ;
    reg Q, Q1;  //注意定义了 Q1 信号
    always @(RST)  //纯组合过程
     if (RST==1)  Q1=0;  else  Q1=D;
    always @(posedge CLK )
           Q <= Q1;
 endmodule
```

■ 基本锁存器及其Verilog表述

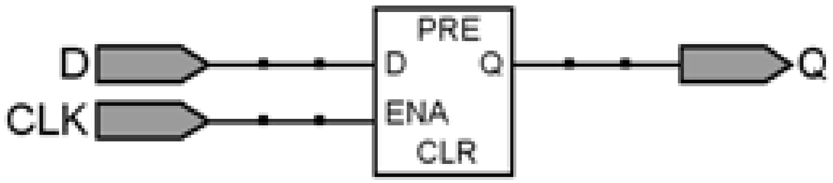


图 5-8 锁存器模块

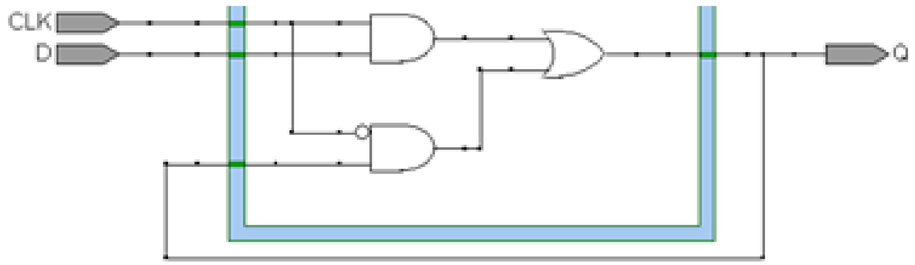


图 5-9 锁存器模块内部逻辑电路

【例 5-8】

```
module LATCH1(CLK,D,Q);
     output Q ; input CLK,D;
     reg Q;
   always @(D or CLK)
     if(CLK)  Q <= D;
endmodule
```

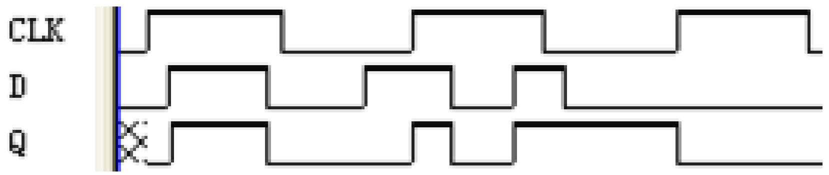


图 5-10 例 5-8 锁存器的时序波形

■ 含清0控制的锁存器及其Verilog表述

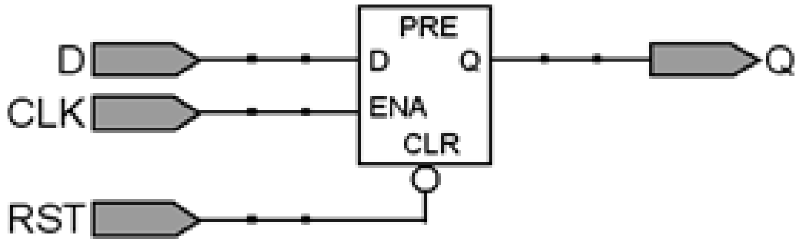


图 5-11 含异步清 0 的锁存器

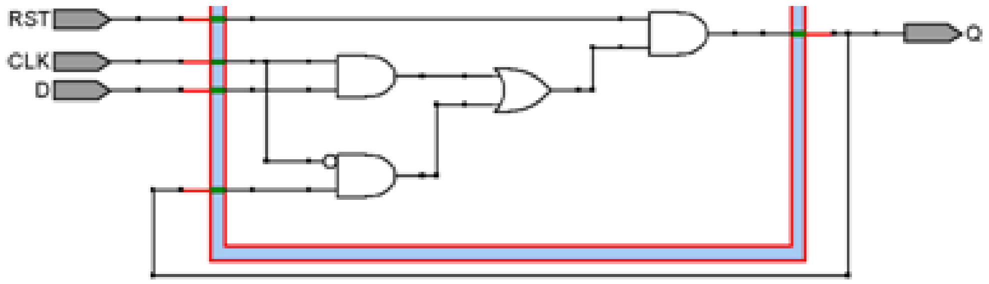
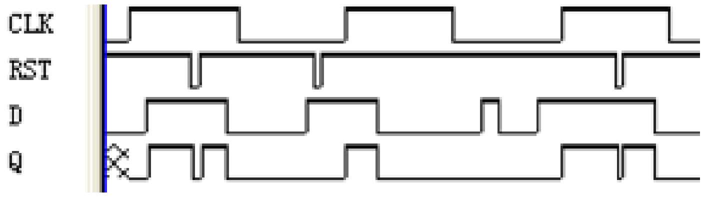


图 5-12 含异步清 0 锁存器的逻辑电路图

【例 5-9】	【例 5-10】

`module LATCH2 (CLK,D,Q,RST);`
`  output Q ;    input CLK,D,RST;`
`  assign Q = (!RST)? 0:(CLK ? D:Q);`
`endmodule`

 	
图 5-13 含异步清 0 的锁存器的仿真波形
	`module LATCH3 (CLK,D,Q,RST);`
	`    output Q ;`
	`    input CLK,D,RST;`
	`    reg Q;`
	`    always @(D or CLK or RST)`
	`            if(!RST)  Q<=0;`
	`       else  if(CLK)   Q<=D;`
	`endmodule`

■ 异步时序电路的Verilog表述特点

【例 5-11】

```
module AMOD(D,A,CLK,Q);
 output Q ;  input A,D,CLK;
 reg Q,Q1;
 always @(posedge CLK)  Q1 = ~(A|Q);
 always @(posedge Q1)  Q =  D;
endmodule
```

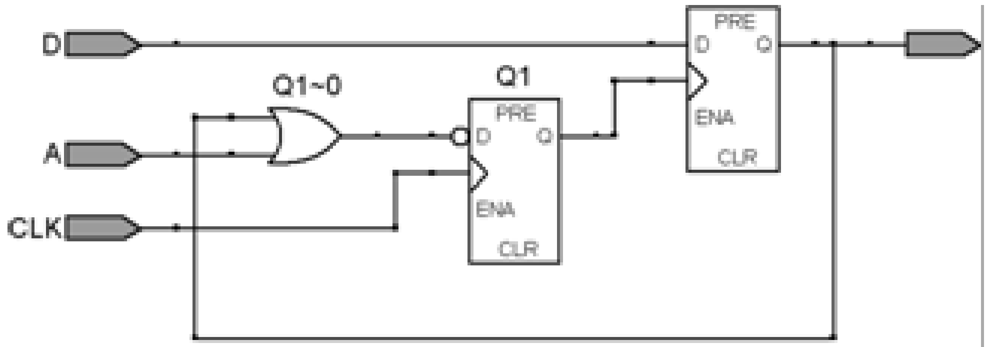


图 5-14 例 5-11 的时序电路图

■ 简单加法计数器及其Verilog表述

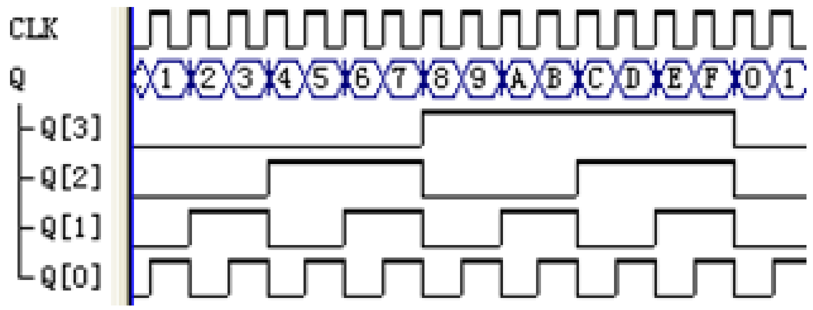


图 5-16　4 位加法计数器工作时序

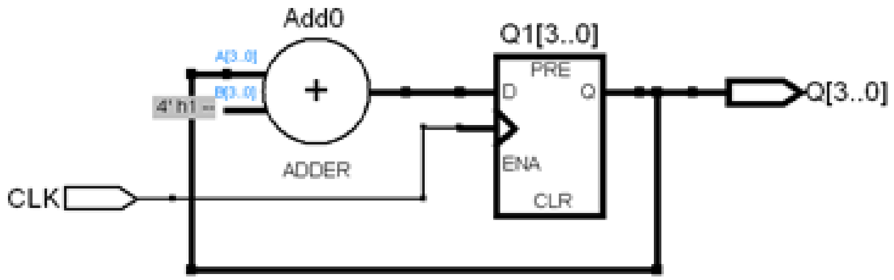


图 5-17　4 位加法计数器 RTL 电路图

【例 5-13】	【例 5-14】

```
module CNT4(CLK,Q);
   output [3:0] Q;   input  CLK;
   reg [3:0] Q1 ;
   always @(posedge CLK)
        Q1 <= Q1+1 ;
   assign Q=Q1;
endmodule
```
	```
	module CNT4 (CLK,Q);
	     output [3:0] Q ;
	     input  CLK;
	      reg [3:0] Q ;
	    always @(posedge CLK)
	         Q <= Q+1 ;
	   endmodule
	```

■ 实用加法计数器

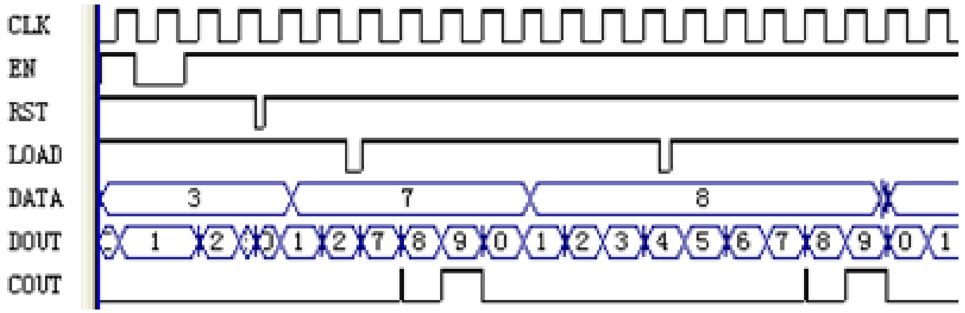


图 5-18 例 5-15 的仿真波形图

```
module CNT10 (CLK,RST,EN,LOAD,COUT,DOUT,DATA);
   input CLK,EN,RST,LOAD ;       // 时钟，时钟使能，复位，数据加载控制信号；
    input  [3:0] DATA ;          // 4 位并行加载数据
    output  [3:0] DOUT ;         // 4 位计数输出
    output COUT ;                // 计数进位输出
   reg  [3:0] Q1 ;     reg COUT ;
   assign DOUT = Q1;             // 将内部寄存器的计数结果输出至 DOUT
   always @(posedge CLK or negedge RST)   //时序过程
     begin
          if (!RST)   Q1 <= 0;      //RST=0 时，对内部寄存器单元异步清 0
     else  if (EN)  begin           //同步使能 EN=1，则允许加载或计数
            if (!LOAD)     Q1<=DATA;  //当 LOAD=0，向内部寄存器加载数据
         else  if (Q1<9)  Q1 <= Q1+1; //当 Q1 小于 9 时，允许累加
         else   Q1 <= 4'b0000; end    //否则一个时钟后清 0 返回初值
     end
   always @(Q1)                          //组合过程
      if (Q1==4'h9)  COUT = 1'b1;  else   COUT = 1'b0;
   endmodule
```

■ 含同步预置功能的移位寄存器

【例 5-16】

```
module SHFT1(CLK,LOAD,DIN,QB);
  output QB;  input CLK,LOAD;  input[7:0] DIN; reg[7:0]  REG8;
   always @(posedge CLK )
     if (LOAD)     REG8<=DIN ;  else REG8[6:0]<=REG8[7:1];
   assign QB = REG8[0] ;
endmodule
```

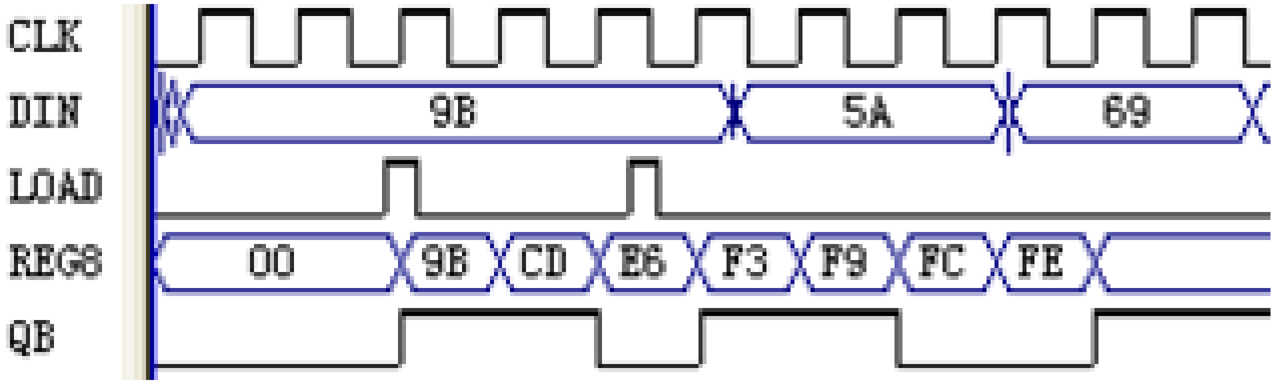


图 5-20 例 5-16 的工作时序图

■ 使用移位操作符设计移位寄存器

【例5-17】

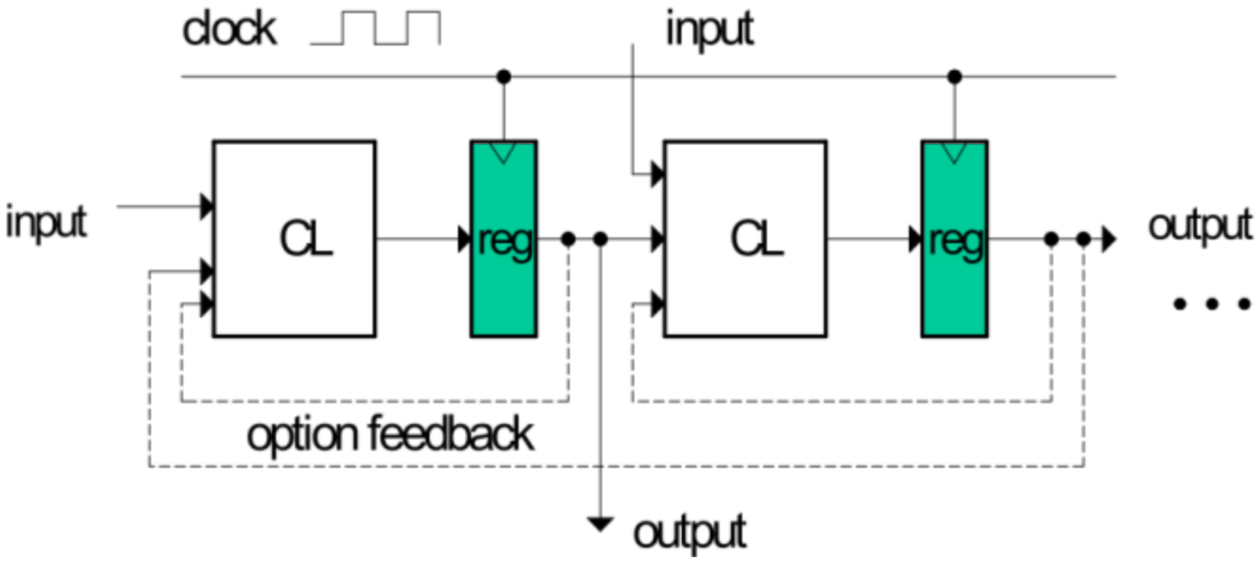
```
module SHIF4 (DIN,CLK,RST,DOUT);
 input CLK,DIN,RST;   output DOUT;   reg [3:0] SHFT;
 always@(posedge CLK or posedge RST)
  if(RST)  SHFT<=4'B0;
    else begin  SHFT <=(SHFT >> 1);  SHFT[3] <= DIN; end
  assign  DOUT  = SHFT[0];
endmodule
```

```
// 双端口存储器
module dual_port_ram
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=6)(
    input [(DATA_WIDTH-1):0] data,
    input [(ADDR_WIDTH-1):0] read_addr, write_addr,
    input we, clk,
    output reg [(DATA_WIDTH-1):0] q);

    // Declare the RAM variable
    reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

    initial begin
      $readmemh("ram_init.txt", ram);     end

    always @ (posedge clk) begin
            // Write
            if (we)    ram[write_addr] <= data;
            q <= ram[read_addr];
    end
endmodule
```

同步时序电路之数据通路

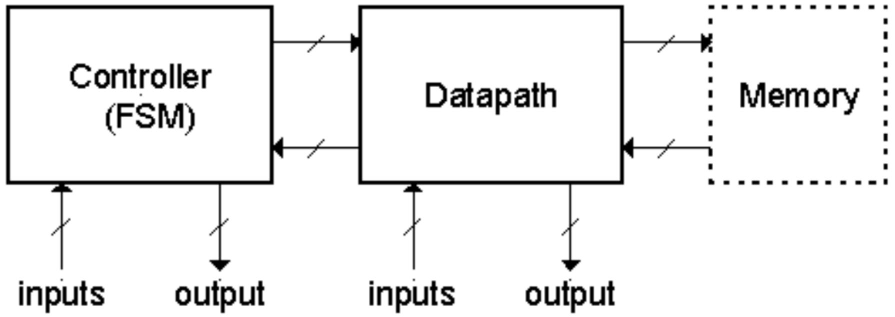
Only Two Types of Circuits Exist

- Combinational Logic Blocks (CL) ← out=f(inputs)
- State Elements (registers) ← $y_{n+1} = f(\text{inputs}, y_n)$



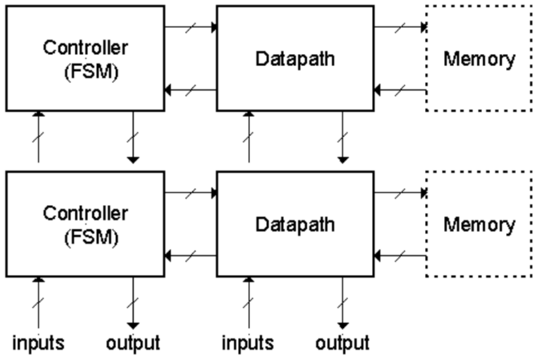
- State elements are mixed in with CL blocks to control the flow of data.

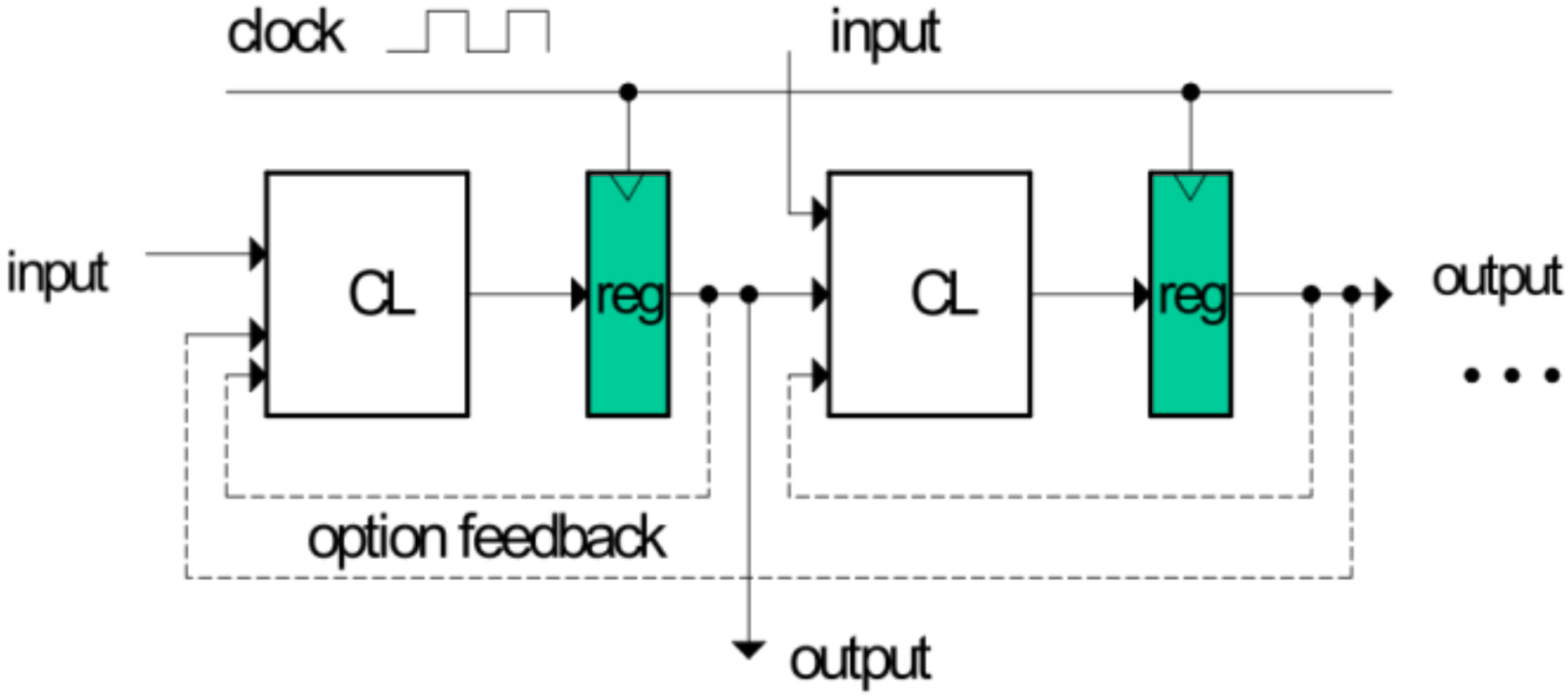
One Standard High-level Template

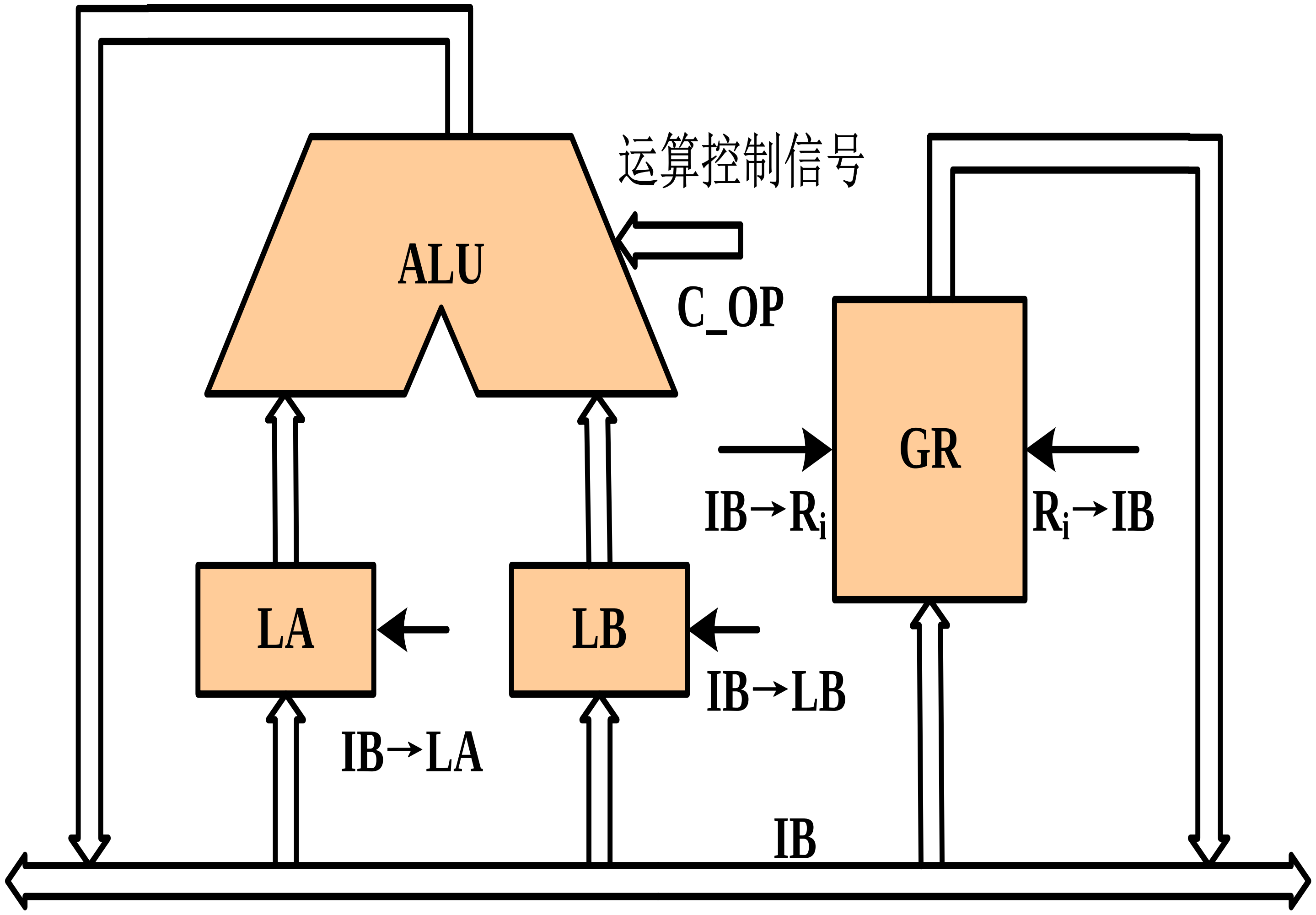


- Controller
  - accepts external and control input, generates control and external output and sequences the movement of data in the datapath. (puppeteer)
- Datapath
  - is responsible for data manipulation. Usually includes a limited amount of storage. (puppet)
- Memory
  - optional block used for long term storage of data structures.

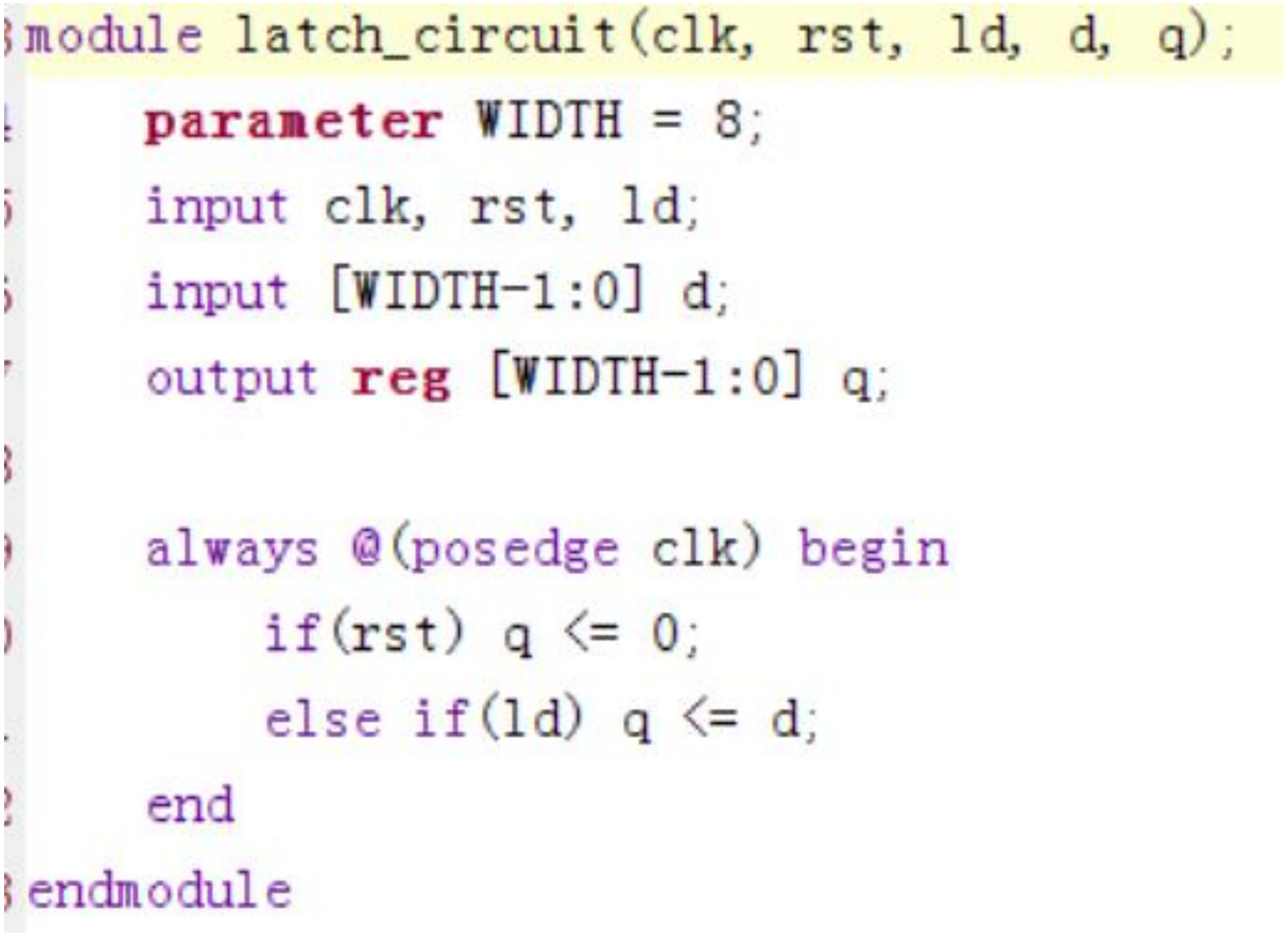
- Standard model for CPUs, micro-controllers, many other digital sub-systems.
- Usually *not* nested.
- Sometimes cascaded:



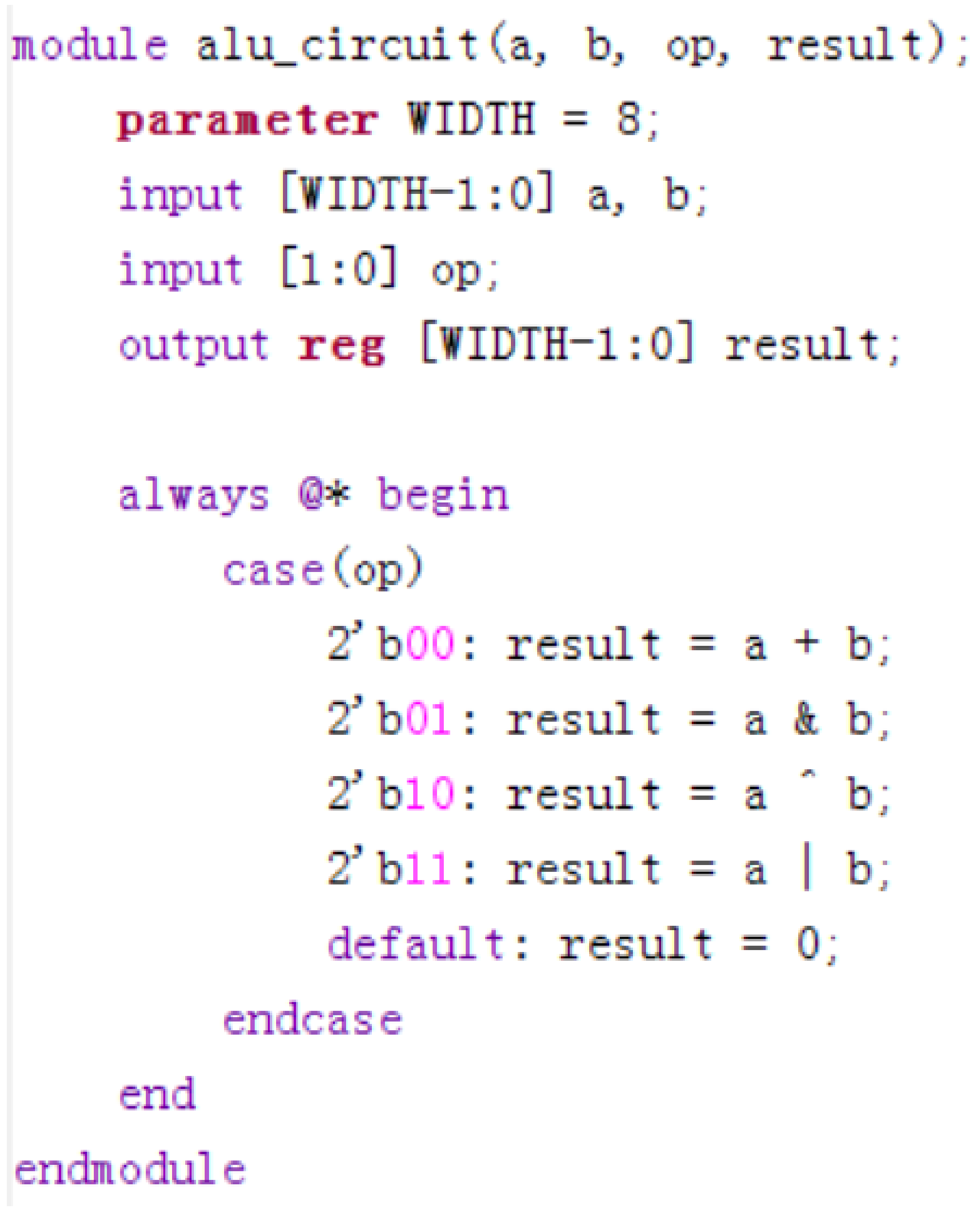

clock
input
input
CL
reg
CL
reg
output
option feedback
output



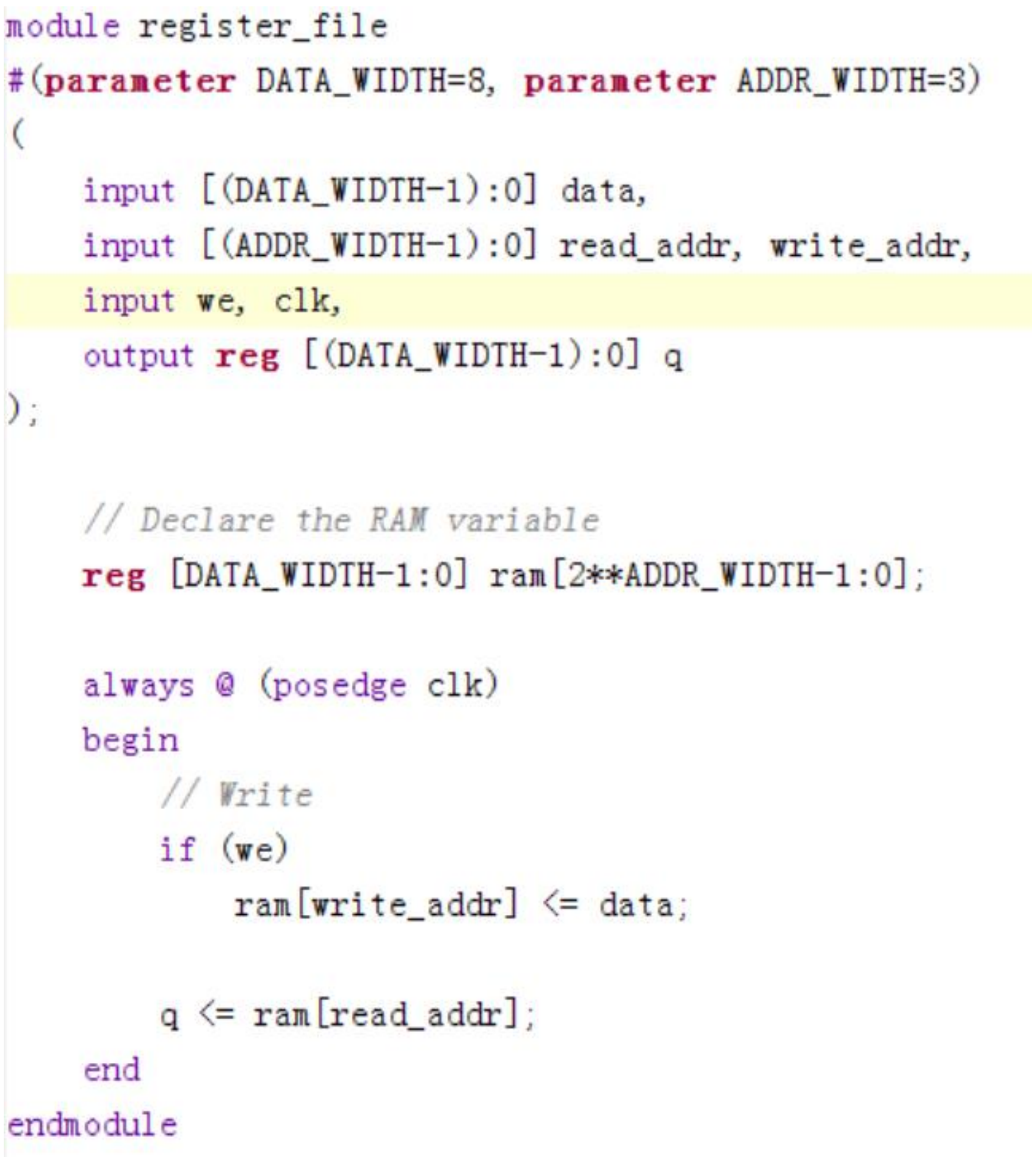
运算控制信号
ALU
C_OP
GR
IB→R$_i$
R$_i$→IB
LA
LB
IB→LB
IB→LA
IB



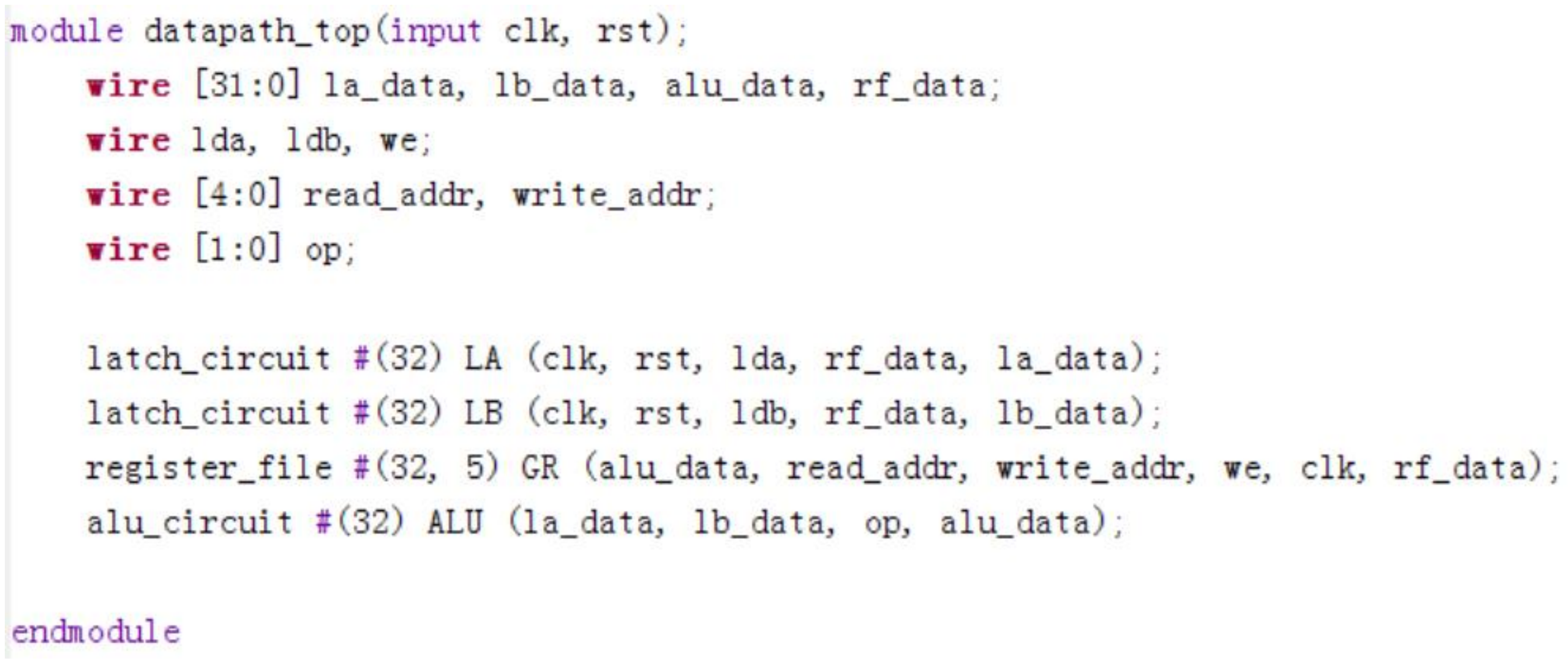
```
module latch_circuit(clk, rst, ld, d, q);
    parameter WIDTH = 8;
    input clk, rst, ld;
    input [WIDTH-1:0] d;
    output reg [WIDTH-1:0] q;

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



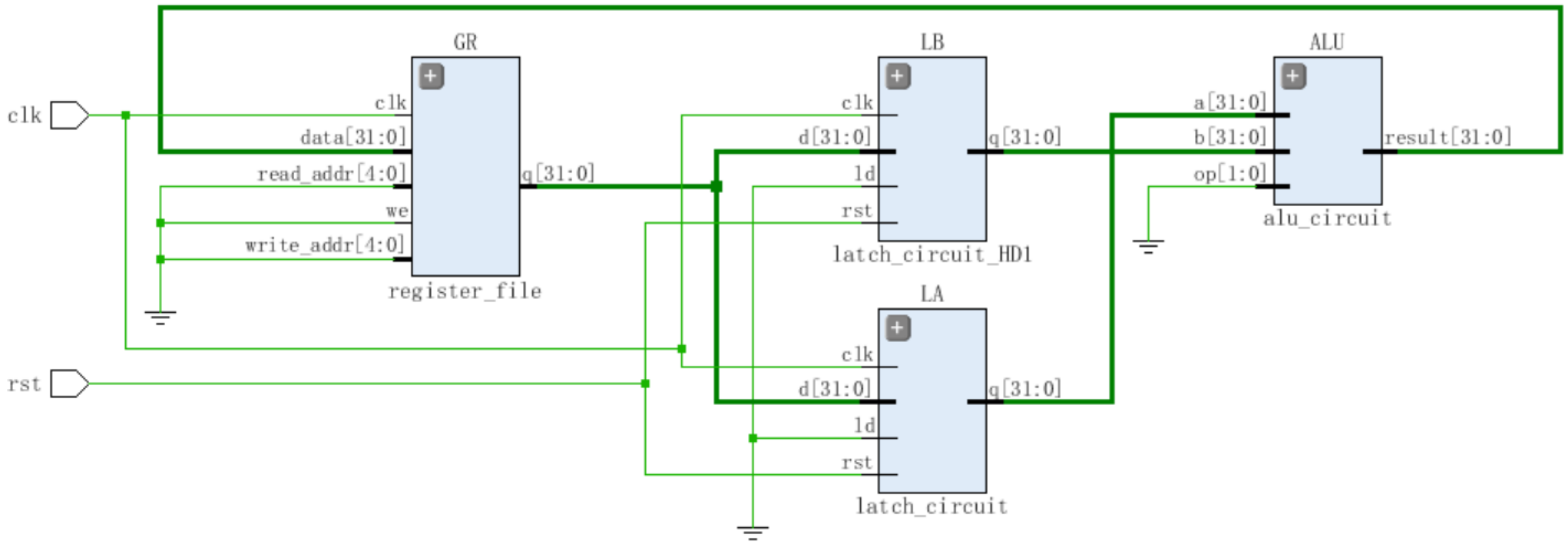
```
module alu_circuit(a, b, op, result);
    parameter WIDTH = 8;
    input [WIDTH-1:0] a, b;
    input [1:0] op;
    output reg [WIDTH-1:0] result;

    always @* begin
        case(op)
            2’b00: result = a + b;
            2’b01: result = a & b;
            2’b10: result = a ^ b;
            2’b11: result = a | b;
            default: result = 0;
        endcase
    end
endmodule
```



```
module register_file
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=3)
(
    input [(DATA_WIDTH-1):0] data,
    input [(ADDR_WIDTH-1):0] read_addr, write_addr,
    input we, clk,
    output reg [(DATA_WIDTH-1):0] q
);

    // Declare the RAM variable
    reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

    always @ (posedge clk)
    begin
        // Write
        if (we)
            ram[write_addr] <= data;

        q <= ram[read_addr];
    end
endmodule
```



```
module datapath_top(input clk, rst);
    wire [31:0] la_data, lb_data, alu_data, rf_data;
    wire lda, ldb, we;
    wire [4:0] read_addr, write_addr;
    wire [1:0] op;

    latch_circuit #(32) LA (clk, rst, lda, rf_data, la_data);
    latch_circuit #(32) LB (clk, rst, ldb, rf_data, lb_data);
    register_file #(32, 5) GR (alu_data, read_addr, write_addr, we, clk, rf_data);
    alu_circuit #(32) ALU (la_data, lb_data, op, alu_data);

endmodule
```

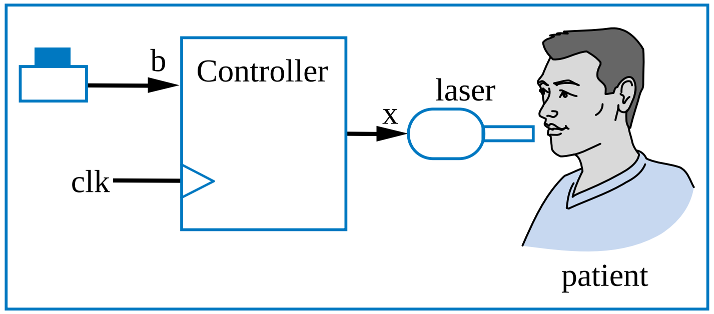

clk
rst
GR
clk
data[31:0]
read_addr[4:0]
we
write_addr[4:0]
q[31:0]
register_file
LB
clk
d[31:0]
ld
rst
q[31:0]
latch_circuit_HD1
LA
clk
d[31:0]
ld
rst
q[31:0]
latch_circuit
ALU
a[31:0]
b[31:0]
op[1:0]
result[31:0]
alu_circuit

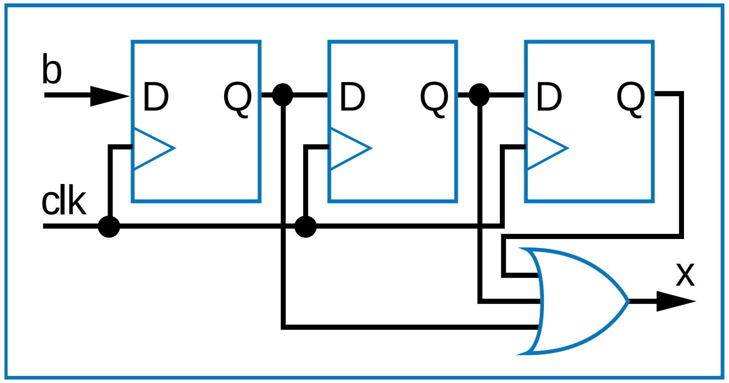
有限状态机FSM

Finite State Machine

Finite-State Machines (FSMs) and Controllers

- Want sequential circuit with particular behavior over time
- Example: Laser timer
  - Push button: x=1 for 3 clock cycles
  - How? Let's try three flip-flops
    - b=1 gets stored in first D flip-flop
    - Then 2nd flip-flop on next cycle, then 3rd flip-flop on next
    - OR the three flip-flop outputs, so x should be 1 for three cycles



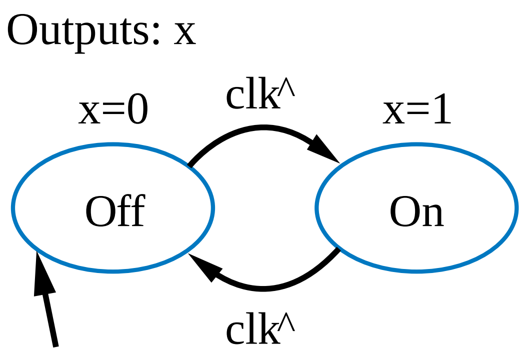


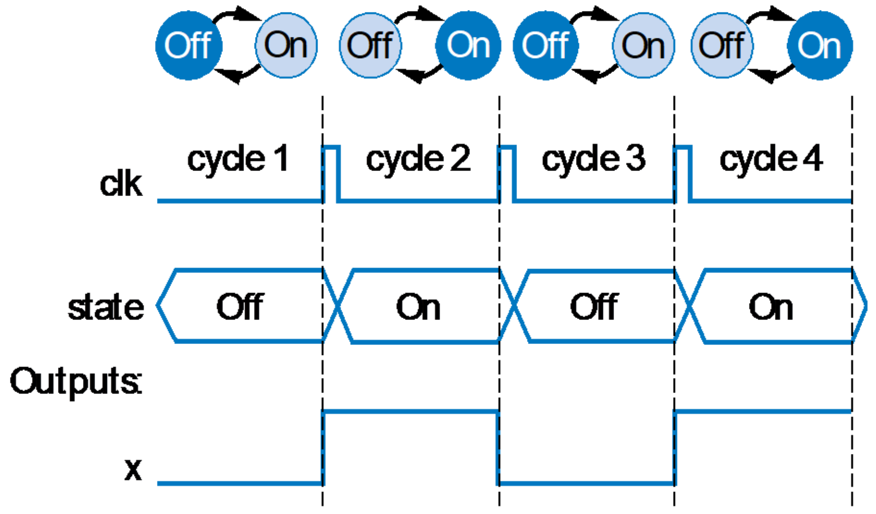
Need a Better Way to Design Sequential Circuits

- Trial and error is not a good design method
  - Will we be able to "guess" a circuit that works for other desired behavior?
    - How about counting up from 1 to 9? Pulsing an output for 1 cycle every 10 cycles? Detecting the sequence 1 3 5 in binary on a 3-bit input?
  - And, a circuit built by guessing may have undesired behavior
    - Laser timer: What if press button again while x=1? x then stays one another 3 cycles. Is that what we want?
- Combinational circuit design process had two important things
  1. A formal way to describe desired circuit behavior
     - Boolean equation, or truth table
  2. A well-defined process to convert that behavior to a circuit
- We need those things for sequence circuit design

Describing Behavior of Sequential Circuit: FSM

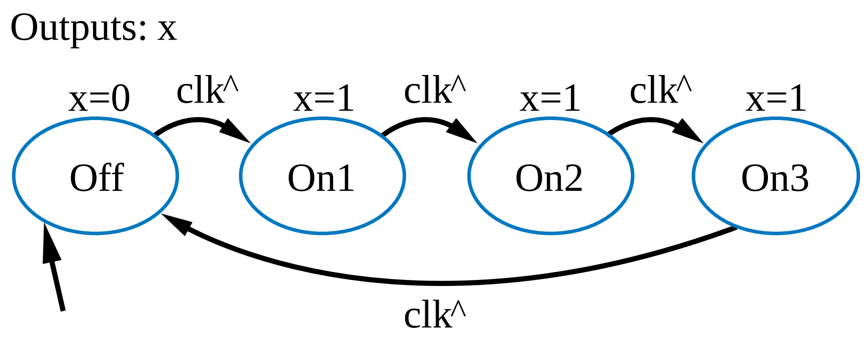
- Finite-State Machine (FSM)
  - A way to describe desired behavior of sequential circuit
    - Akin to Boolean equations for combinational behavior
  - List states, and transitions among states
    - Example: Make x change toggle (0 to 1, or 1 to 0) every clock cycle
    - Two states: “Off” (x=0), and “On” (x=1)
    - Transition from Off to On, or On to Off, on rising clock edge
    - Arrow with no starting state points to initial state (when circuit first starts)

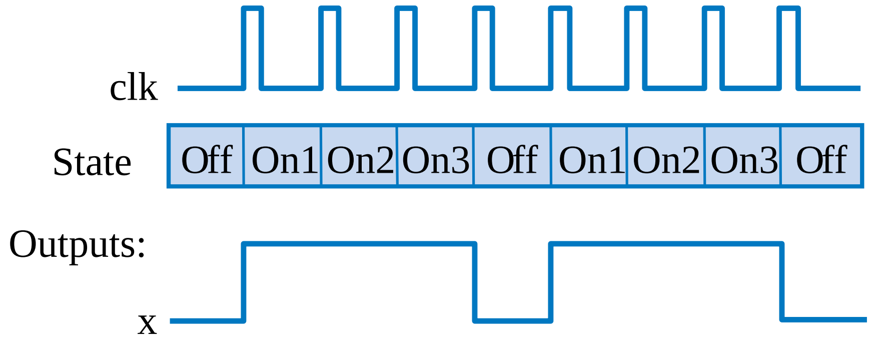




FSM Example: 0,1,1,1,repeat

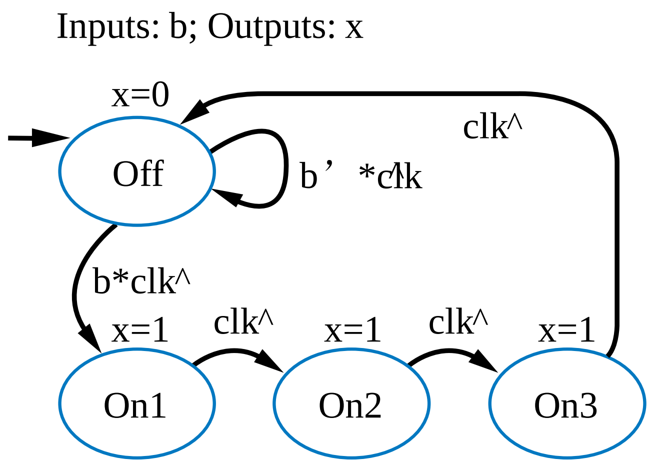
- Want 0, 1, 1, 1, 0, 1, 1, 1, ...
  - Each value for one clock cycle
- Can describe as FSM
  - Four states
  - Transition on rising clock edge to next state

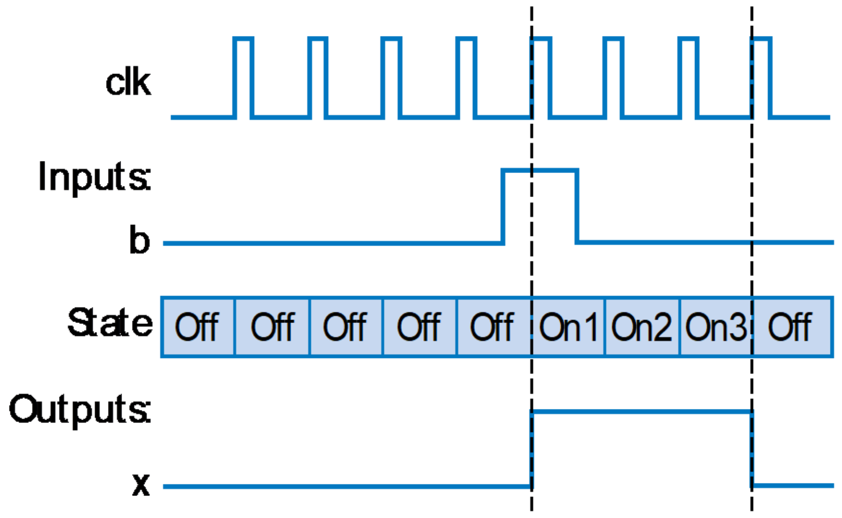




Extend FSM to Three-Cycles High Laser Timer

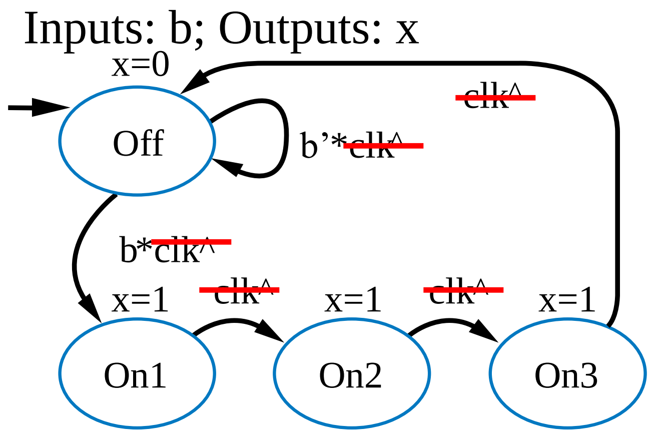
- Four states
- Wait in “Off” state while b is 0 (b’)
- When b is 1 (and rising clock edge), transition to On1
  - Sets x=1
  - On next two clock edges, transition to On2, then On3, which also set x=1
- So x=1 for three cycles after button pressed

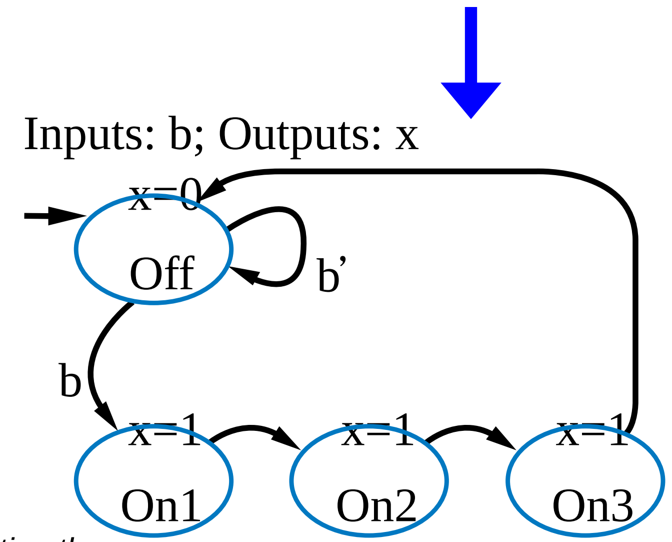




FSM Simplification: Rising Clock Edges Implicit

- Showing rising clock on every transition: cluttered
  - Make implicit -- assume every edge has rising clock, even if not shown
  - What if we wanted a transition *without* a rising edge
    - We don't consider such asynchronous FSMs -- less common, and advanced topic
    - Only consider ***synchronous*** FSMs -- rising edge on *every* transition

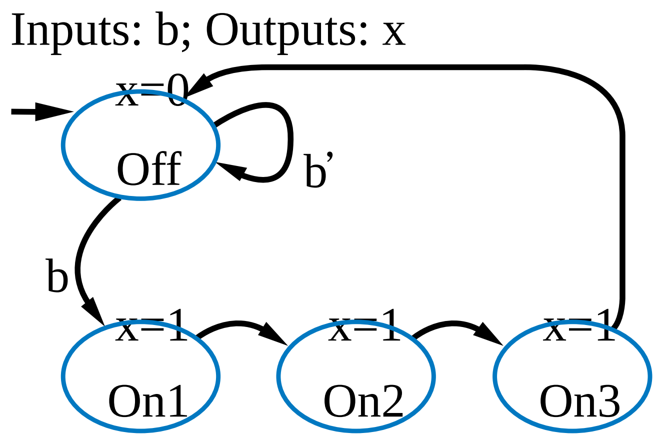




Note: Transition with no associated condition thus transistions to next state on next clock cycle

FSM Definition

- FSM consists of
  - Set of states
    - Ex: {Off, On1, On2, On3}
  - Set of inputs, set of outputs
    - Ex: Inputs: {x}, Outputs: {b}
  - Initial state
    - Ex: “Off”
  - Set of transitions
    - Describes next states
    - Ex: Has 5 transitions
  - Set of actions
    - Sets outputs while in states
    - Ex: x=0, x=1, x=1, and x=1

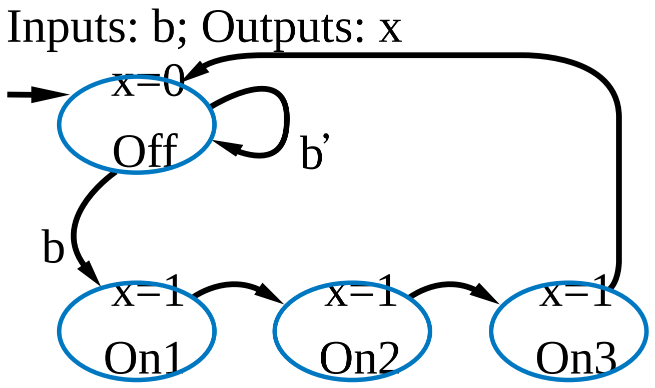


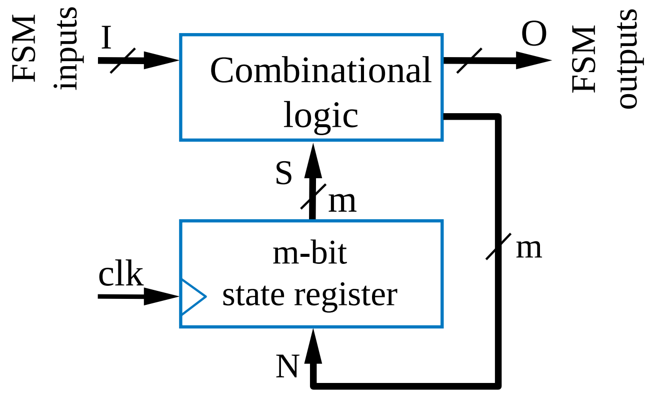
We often draw FSM graphically, known as ***state diagram***

Can also use table (state table), or textual languages

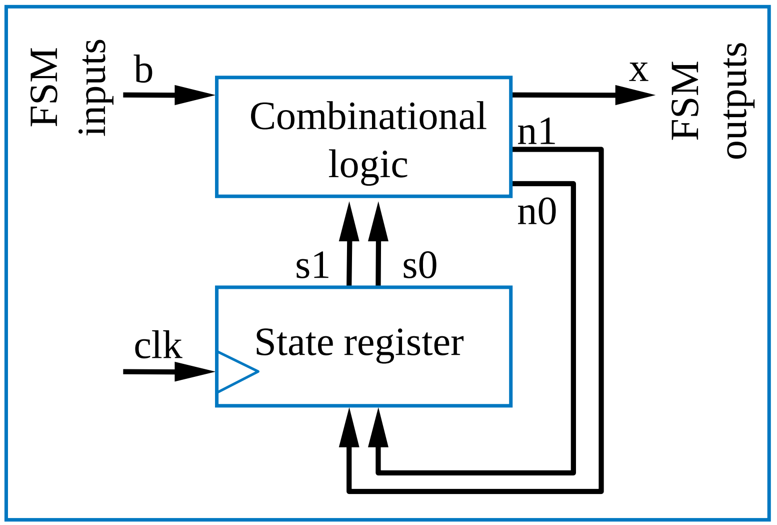
Standard Controller Architecture

- How implement FSM as sequential circuit?
  - Use standard architecture
    - State register -- to store the present state
    - Combinational logic -- to compute outputs, and next state
    - For laser timer FSM
      - 2-bit state register, can represent four states
      - Input b, output x
  - Known as ***controller***



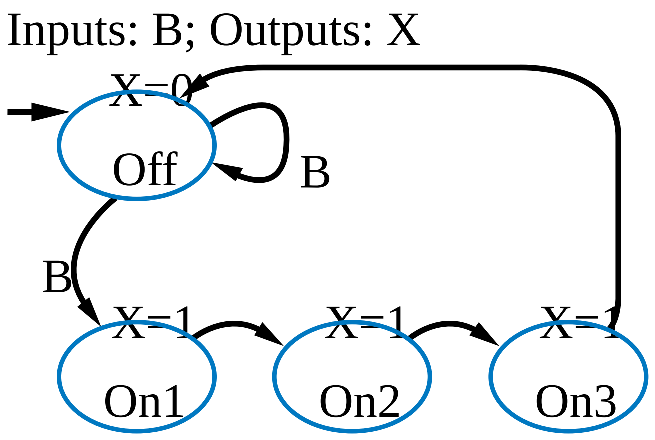


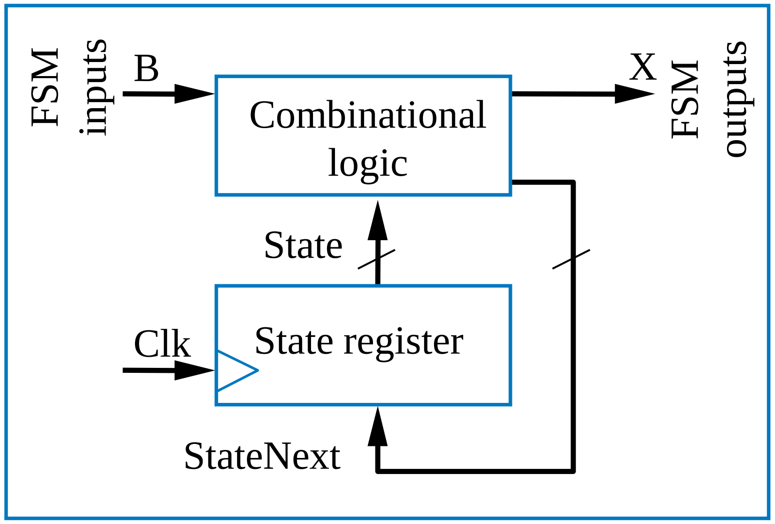
General version



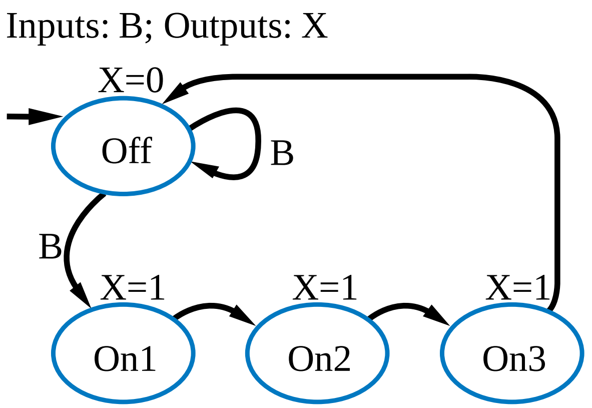
Finite-State Machines (FSMs)—Sequential Behavior

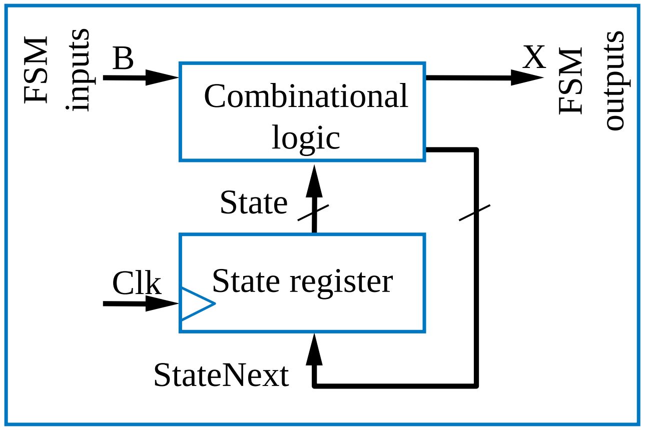
- Finite-state machine (FSM) is a common model of sequential behavior
  - Example: If B=1, hold X=1 for 3 clock cycles
    - Note: Transitions implicitly ANDed with rising clock edge
  - Implementation model has two parts:
    - State register
    - Combinational logic
  - HDL model will reflect those two parts





Finite-State Machines (FSMs)—Sequential Behavior
Modules with Multiple Procedures and Shared Variables





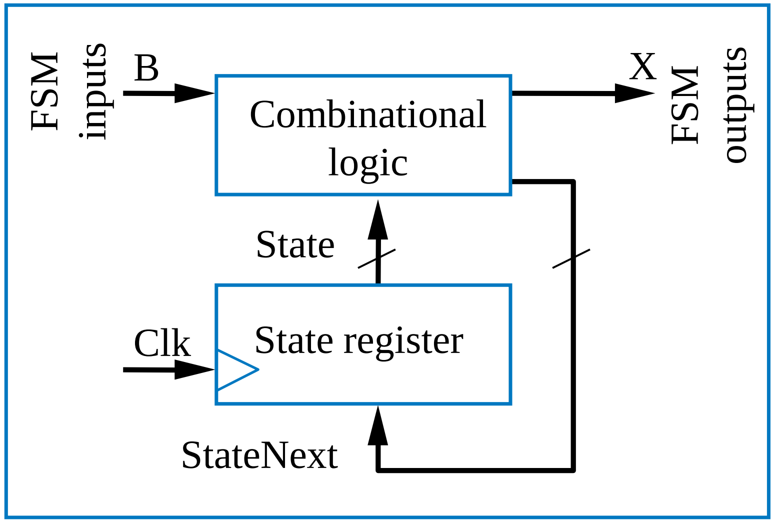
```
`timescale 1 ns/1 ns

module LaserTimer(B, X, Clk, Rst);

   input B;
   output reg X;
   input Clk, Rst;

   parameter S_Off = 0, S_On1 = 1,
             S_On2 = 2, S_On3 = 3;

   reg [1:0] State, StateNext;

   // CombLogic
   always @(State, B) begin
      case (State)
         S_Off: begin
            X <= 0;
            if (B == 0)
               StateNext <= S_Off;
            else
               StateNext <= S_On1;
         end
         ...
```

```
         ...
         S_On1: begin
            X <= 1;
            StateNext <= S_On2;
         end
         S_On2: begin
            X <= 1;
            StateNext <= S_On3;
         end
         S_On3: begin
            X <= 1;
            StateNext <= S_Off;
         end
      endcase
   end

   // StateReg
   always @(posedge Clk) begin
      if (Rst == 1 )
         State <= S_Off;
      else
         State <= StateNext;
   end
endmodule
```

- Code will be explained on following slides

Finite-State Machines (FSMs)—Sequential Behavior

- Modules has two procedures
  - One procedure for combinational logic
  - One procedure for state register
  - But it's still a behavioral description



```
`timescale 1 ns/1 ns

module LaserTimer(B, X, Clk, Rst);

   input B;
   output reg X;
   input Clk, Rst;

   parameter S_Off = 0, S_On1 = 1,
             S_On2 = 2, S_On3 = 3;

   reg [1:0] State, StateNext;

   // CombLogic
   always @(State, B) begin
      ...
   end

   // StateReg
   always @(posedge Clk) begin
      ...
   end
endmodule
```

Finite-State Machines (FSMs)—Sequential Behavior Parameters

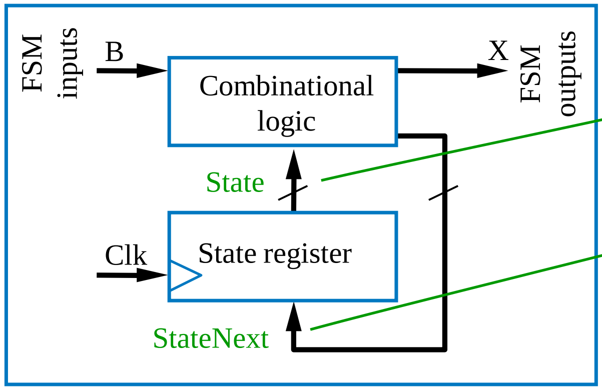
- parameter declaration
  - Not a variable or net, but rather a *constant*
  - A constant is a value that must be initialized, and that cannot be changed within the module's definition
  - Four parameters defined
    - *S_Off, S_On1, S_On2, S_On3*
    - Correspond to FSM's states
      - Should be initialized to unique values

```
`timescale 1 ns/1 ns

module LaserTimer(B, X, Clk, Rst);

   input B;
   output reg X;
   input Clk, Rst;

   parameter S_Off = 0, S_On1 = 1,
             S_On2 = 2, S_On3 = 3;

   reg [1:0] State, StateNext;

   // CombLogic
   always @(State, B) begin
      ...
   end

   // StateReg
   always @(posedge Clk) begin
      ...
   end
endmodule
```

Finite-State Machines (FSMs)—Sequential Behavior

- Module declares two reg variables
  - *State*, *StateNext*
  - Each is 2-bit vector (need two bits to represent four unique state values 0 to 3)
  - Variables are shared between CombLogic and StateReg procedures
- CombLogic procedure
  - Event control sensitive to *State* and input *B*
  - Will output *StateNext* and *X*
- StateReg procedure
  - Sensitive to *Clk* input
  - Will output *State*, which it stores

```
`timescale 1 ns/1 ns

module LaserTimer(B, X, Clk, Rst);

   input B;
   output reg X;
   input Clk, Rst;

   parameter S_Off = 0, S_On1 = 1,
             S_On2 = 2, S_On3 = 3;

   reg [1:0] State, StateNext;

   // CombLogic
   always @(State, B) begin
      ...
   end

   // StateReg
   always @(posedge Clk) begin
      ...
   end
endmodule
```



Finite-State Machines (FSMs)—Sequential Behavior Procedures with Case Statements

- Procedure may use case statement
  - Preferred over if-else-if when just one expression determines which statement to execute
  - case (expression)
    - Execute statement whose case item expression value matches case expression
      - *case item expression : statement*
      - *statement* is commonly a begin-end block, as in example
  - First case item expression that matches executes; remaining case items ignored
  - If no item matches, nothing executes
  - Last item may be "*default : statement*"
    - Statement executes if none of the previous items matched

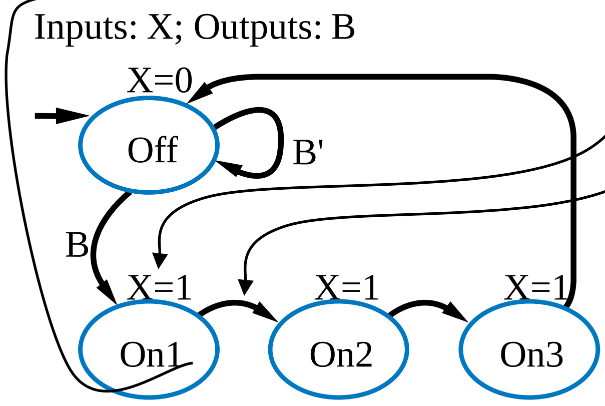
```
// CombLogic
always @(State, B) begin
   case (State)
      S_Off: begin
         X <= 0;
         if (B == 0)
            StateNext <= S_Off;
         else
            StateNext <= S_On1;
      end
      S_On1: begin
         X <= 1;
         StateNext <= S_On2;
      end
      S_On2: begin
         X <= 1;
         StateNext <= S_On3;
      end
      S_On3: begin
         X <= 1;
         StateNext <= S_Off;
      end
   endcase
end
```

Finite-State Machines (FSMs)—Sequential Behavior Procedures with Case Statements

- FSM's CombLogic procedure
  - Case statement describes states
  - case (State)
    - Executes corresponding statement (often a begin-end block) based on State's current value
  - A state's statements consist of
    - Actions of the state
    - Setting of next state (transitions)
- Ex: State is S_On1
  - Executes statements for state On1, jumps to endcase

Suppose State is S_On1

```
reg [1:0] State, StateNext;

// CombLogic
always @(State, B) begin
   case (State)
      S_Off: begin
         X <= 0;
         if (B == 0)
            StateNext <= S_Off;
         else
            StateNext <= S_On1;
      end
      S_On1: begin
         X <= 1;
         StateNext <= S_On2;
      end
      S_On2: begin
         X <= 1;
         StateNext <= S_On3;
      end
      S_On3: begin
         X <= 1;
         StateNext <= S_Off;
      end
   endcase
end
```

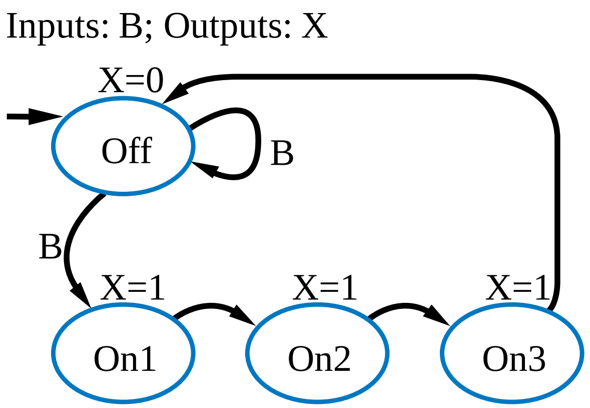


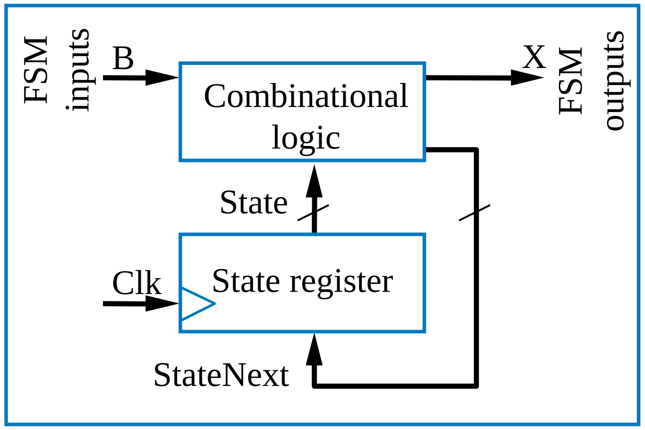
Finite-State Machines (FSMs)—Sequential Behavior

- FSM StateReg Procedure
  - Similar to 4-bit register
    - Register for State is 2-bit vector reg variable
  - Procedure has synchronous reset
    - Resets State to FSM's initial state, S_Off

```
...
   parameter S_Off = 0, S_On1 = 1,
             S_On2 = 2, S_On3 = 3;

   reg [1:0] State, StateNext;

...

   // StateReg
   always @(posedge Clk) begin
      if (Rst == 1 )
         State <= S_Off;
      else
         State <= StateNext;
   end
...
```

Finite-State Machines (FSMs)—Sequential Behavior
Modules with Multiple Procedures and Shared Variables





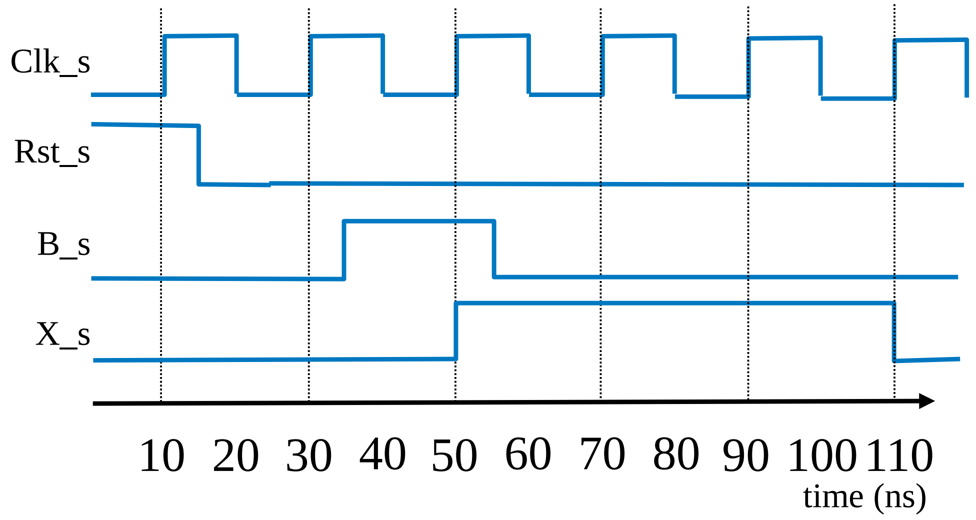
```
`timescale 1 ns/1 ns

module LaserTimer(B, X, Clk, Rst);

   input B;
   output reg X;
   input Clk, Rst;

   parameter S_Off = 0, S_On1 = 1,
             S_On2 = 2, S_On3 = 3;

   reg [1:0] State, StateNext;

   // CombLogic
   always @(State, B) begin
      case (State)
         S_Off: begin
            X <= 0;
            if (B == 0)
               StateNext <= S_Off;
            else
               StateNext <= S_On1;
         end
         ...
```

```
         ...
         S_On1: begin
            X <= 1;
            StateNext <= S_On2;
         end
         S_On2: begin
            X <= 1;
            StateNext <= S_On3;
         end
         S_On3: begin
            X <= 1;
            StateNext <= S_Off;
         end
      endcase
   end

   // StateReg
   always @(posedge Clk) begin
      if (Rst == 1 )
         State <= S_Off;
      else
         State <= StateNext;
   end
endmodule
```

- Code should now be clear

Finite-State Machines (FSMs)—Sequential Behavior Self-Checking Testbenches

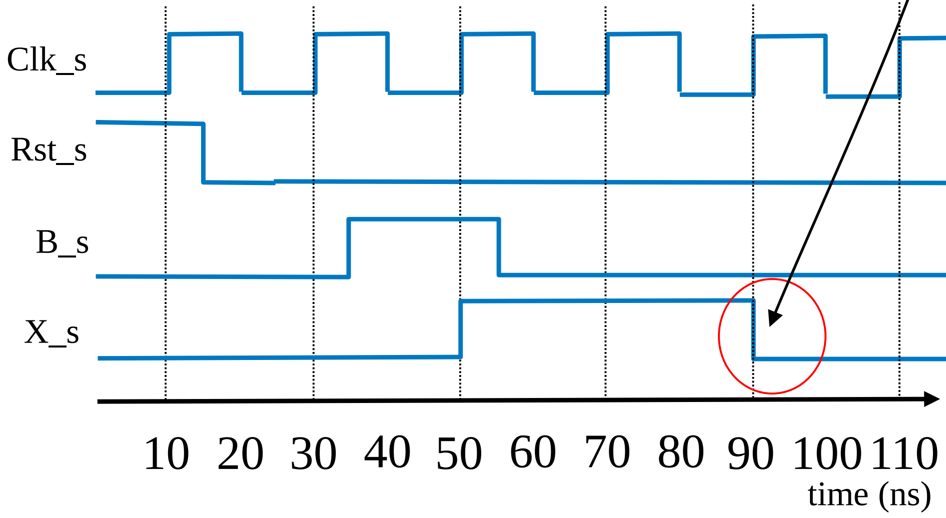
- FSM testbench
  - First part of file (variable/net declarations, module instantiations) similar to before
  - Vector Procedure
    - Resets FSM
    - Sets FSM's input values (“test vectors”)
    - Waits for specific clock cycles
  - We observe the resulting waveforms to determine if FSM behaves correctly



```
...
   // Clock Procedure
   always begin
      Clk_s <= 0;
      #10;
      Clk_s <= 1;
      #10;
   end   // Note: Procedure repeats

   // Vector Procedure
   initial begin
      Rst_s <= 1;
      B_s <= 0;
      @(posedge Clk_s);
      #5 Rst_s <= 0;
      @(posedge Clk_s);
      #5 B_s <= 1;
      @(posedge Clk_s);
      #5 B_s <= 0;
      @(posedge Clk_s);
      @(posedge Clk_s);
      @(posedge Clk_s);
   end
endmodule
```

Finite-State Machines (FSMs)—Sequential Behavior Self-Checking Testbenches

- Reading waveforms is error-prone
- Create *self-checking testbench*
  - Use *if* statements to check for expected values
    - If a check fails, print error message
    - Ex: if X_s fell to 0 one cycle too early, simulation might output:
      - *95: Third X=1 failed*



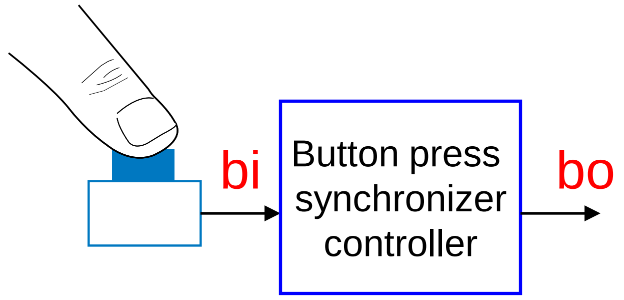
```
// Vector Procedure
initial begin
   Rst_s <= 1;
   B_s <= 0;
   @(posedge Clk_s);
   #5 if (X_s != 0)
      $display("%t: Reset failed", $time);
   Rst_s <= 0;
   @(posedge Clk_s);
   #5 B_s <= 1;
   @(posedge Clk_s);
   #5 B_s <= 0;
   if (X_s != 1)
      $display("%t: First X=1 failed", $time);
   @(posedge Clk_s);
   #5 if (X_s != 1)
      $display("%t: Second X=1 failed", $time);
   @(posedge Clk_s);
   #5 if (X_s != 1)
      $display("%t: Third X=1 failed", $time);
   @(posedge Clk_s);
   #5 if (X_s != 0)
      $display("%t: Final X=0 failed", $time);
end
```

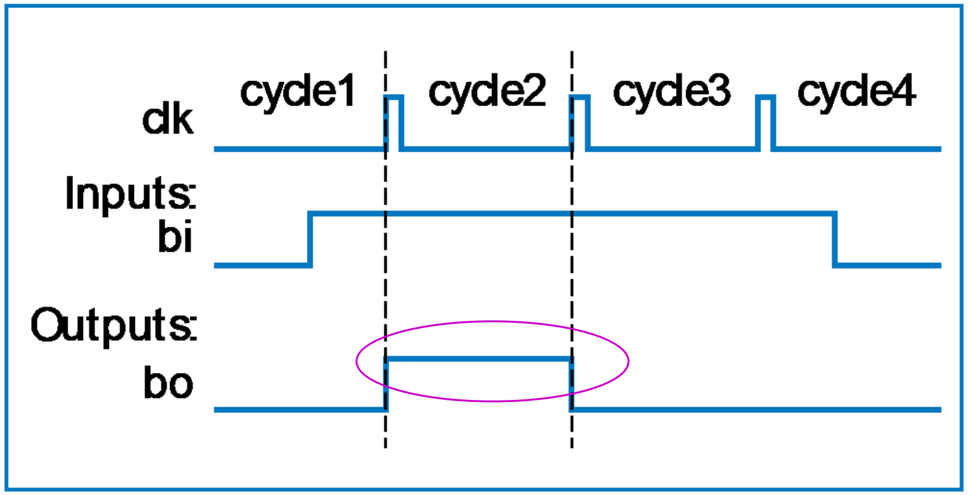
Finite-State Machines (FSMs)—Sequential Behavior $display System Procedure

- $display – built-in Verilog system procedure for printing information to display during simulation
  - A system procedure interacts with the simulator and/or host computer system
    - To write to a display, read a file, get the current simulation time, etc.
    - Starts with $ to distinguish from regular procedures
- String argument is printed literally...
  - $display("Hello") will print "Hello"
  - Automatically adds newline character
- ...except when special sequences appear
  - *%t*: Display a time expression
  - Time expression must be next argument
    - $time – Built-in system procedure that returns the current simulation time
      - *95: Third X=1 failed*

```
// Vector Procedure
initial begin
   Rst_s <= 1;
   B_s <= 0;
   @(posedge Clk_s);
   #5 if (X_s != 0)
      $display("%t: Reset failed", $time);
   Rst_s <= 0;
   @(posedge Clk_s);
   #5 B_s <= 1;
   @(posedge Clk_s);
   #5 B_s <= 0;
   if (X_s != 1)
      $display("%t: First X=1 failed", $time);
   @(posedge Clk_s);
   #5 if (X_s != 1)
      $display("%t: Second X=1 failed", $time);
   @(posedge Clk_s);
   #5 if (X_s != 1)
      $display("%t: Third X=1 failed", $time);
   @(posedge Clk_s);
   #5 if (X_s != 0)
      $display("%t: Final X=0 failed", $time);
end
```

Controller Example:
Button Press Synchronizer

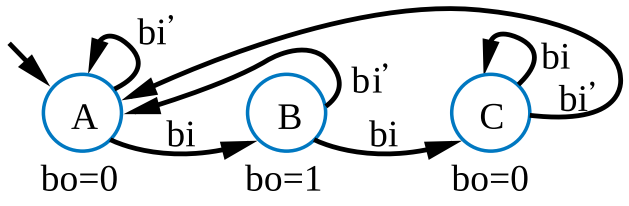




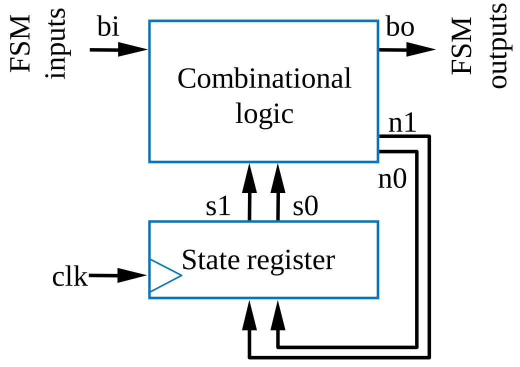
- Want simple sequential circuit that converts button press to single cycle duration, regardless of length of time that button actually pressed
  - We assumed such an ideal button press signal in earlier example, like the button in the laser timer controller

Controller Example: Button Press Synchronizer (cont)

FSM inputs: bi; FSM outputs: bo

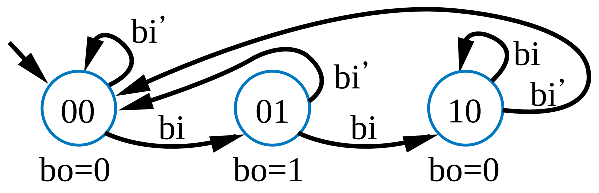


Step 1: FSM



Step 2: Create architecture

FSM inputs: bi; FSM outputs: bo

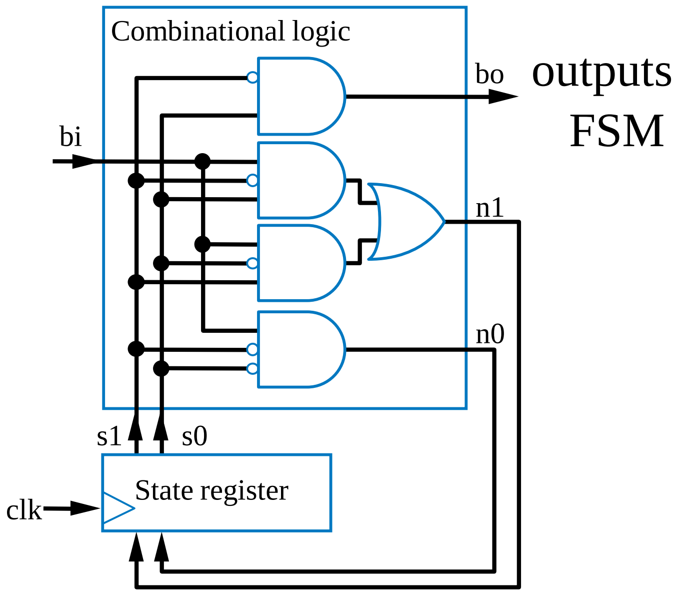


Step 3: Encode states

	Combinational logic					
	Inputs			Outputs		
	s1	s0	bi	n1	n0	bo
A	0	0	0	0	0	0
	0	0	1	0	1	0
B	0	1	0	0	0	1
	0	1	1	1	0	1
C	1	0	0	0	0	0
	1	0	1	1	0	0
unused	1	1	0	0	0	0
	1	1	1	0	0	0

Step 4: State table

n1 = s1's0bi + s1s0'bi
n0 = s1's0'bi
bo = s1's0bi' + s1's0bi = s1s0

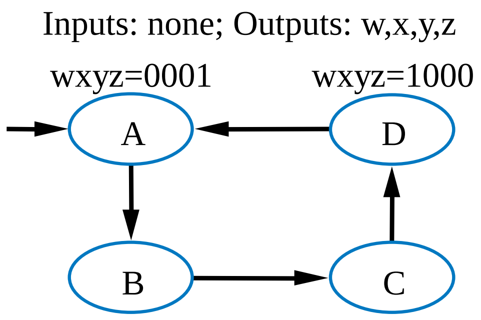


Step 5: Create combinational circuit

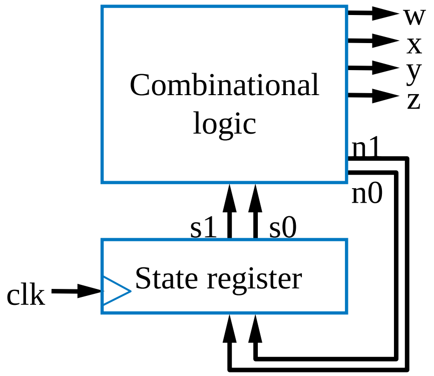
Step 5: Create combinational circuit

Controller Example: Sequence Generator

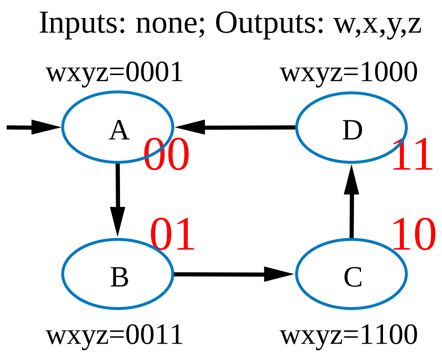
- Want generate sequence 0001, 0011, 1100, 1000, (repeat)
  - Each value for one clock cycle
  - Common, e.g., to create pattern in 4 lights, or control magnets of a "stepper motor"



Step 1: Create FSM



Step 2: Create architecture

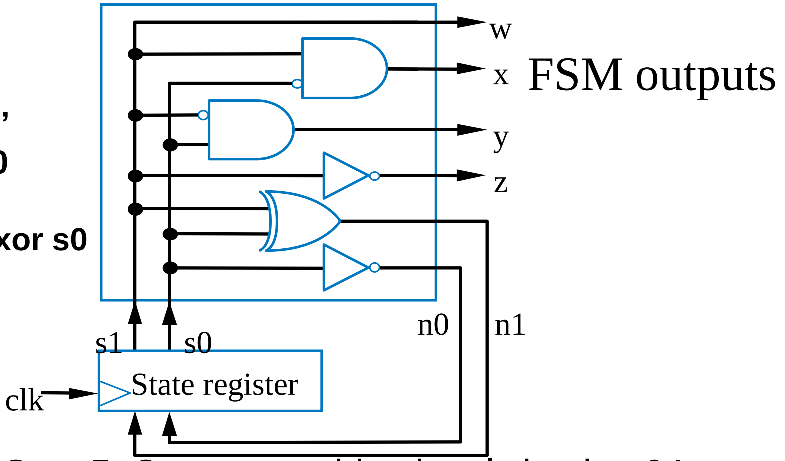


Step 3: Encode states

	Inputs		Outputs					
	s1	s0	w	x	y	z	n1	n0
A	0	0	0	0	0	1	0	1
B	0	1	0	0	1	1	1	0
C	1	0	1	1	0	0	1	1
D	1	1	1	0	0	0	0	0

Step 4: Create state table

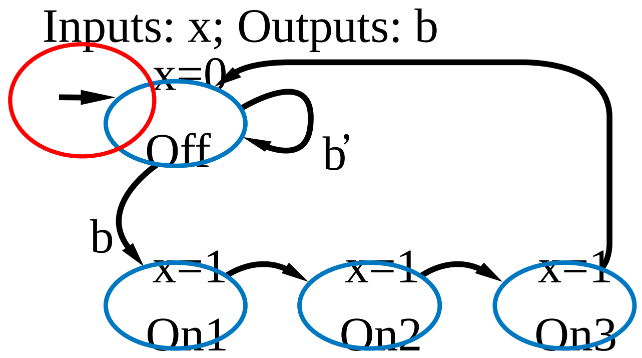
w = s1
x = s1s0'
y = s1's0
z = s1'
n1 = s1 xor s0
n0 = s0'

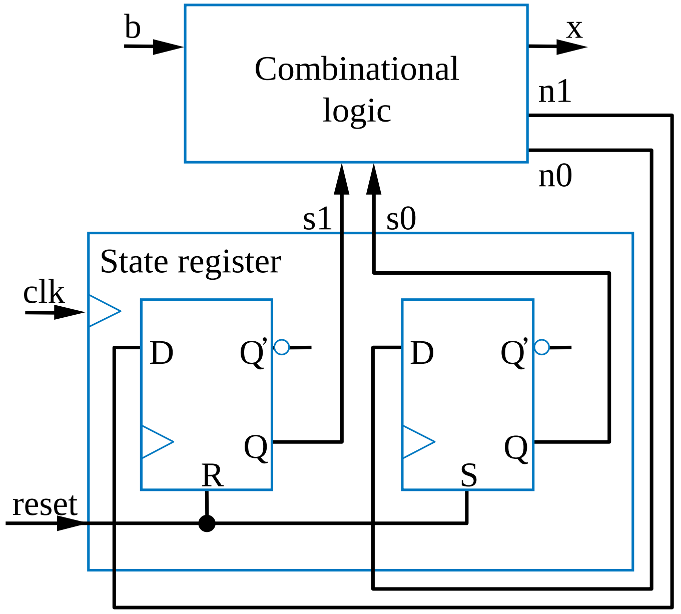


Step 5: Create combinational circuit

Initial State of a Controller

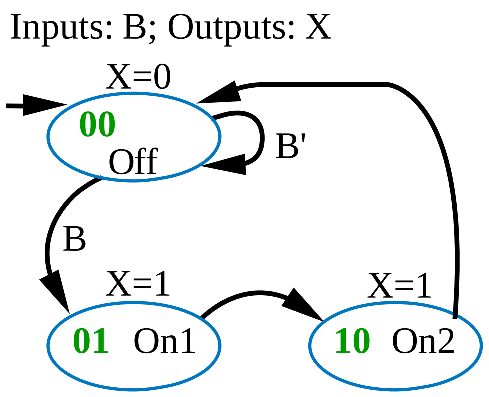
- All our FSMs had initial state
  - But our sequential circuit designs did not
  - Can accomplish using flip-flops with reset/set inputs
    - Shown circuit initializes flip-flops to 01
  - Designer must ensure reset input is 1 during power up of circuit
    - By electronic circuit design





Describing Safe FSMs

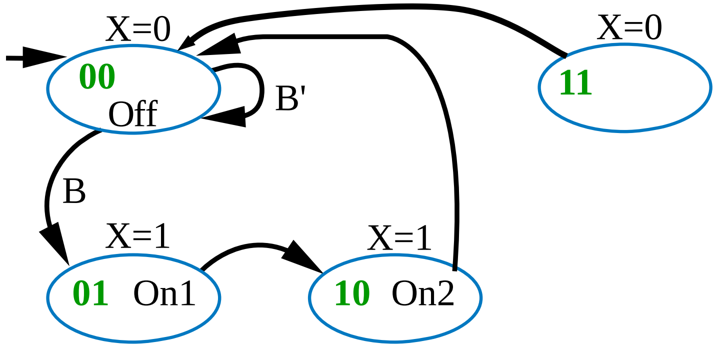
- Safe FSM – If enters illegal state, transitions to a legal state
- Example
  - Suppose example has only three states
  - Two-bit encoding has illegal state encoding "11"
    - Also known as "unreachable" state
    - Not possible to enter that state under normal FSM operation
    - But actually possible to enter that state due to circuit error – e.g., electrical noise that causes state register bit to switch



Describing Safe FSMs

- Safe FSM – If enters illegal state, transitions to a legal state
- Example
  - Suppose example has only three states
  - Two-bit encoding has illegal state encoding "11"
  - Safe implementation
    - Transition to appropriate legal state
    - Even though that undefined state appears to be unreachable
    - Thus, FSM recovers from the error

Inputs: B; Outputs: X



Describing Safe FSMs in Verilog HDL

- Unsafe FSM description
  - Only describes legal states, ignores illegal states
- Some synthesis tools support "safe" option during synthesis
  - Automatically creates safe FSM from an unsafe FSM description

```
...
   reg [1:0] State, StateNext;

   always @(State, B) begin
     case (State)
        S_Off: begin
           X <= 0;
           if (B == 0)
              StateNext <= S_Off;
           else
              StateNext <= S_On1;
        end
        S_On1: begin
           X <= 1;
           StateNext <= S_On2;
        end
        S_On2: begin
           X <= 1;
           StateNext <= S_Off;
        end
     endcase
   end
...
```

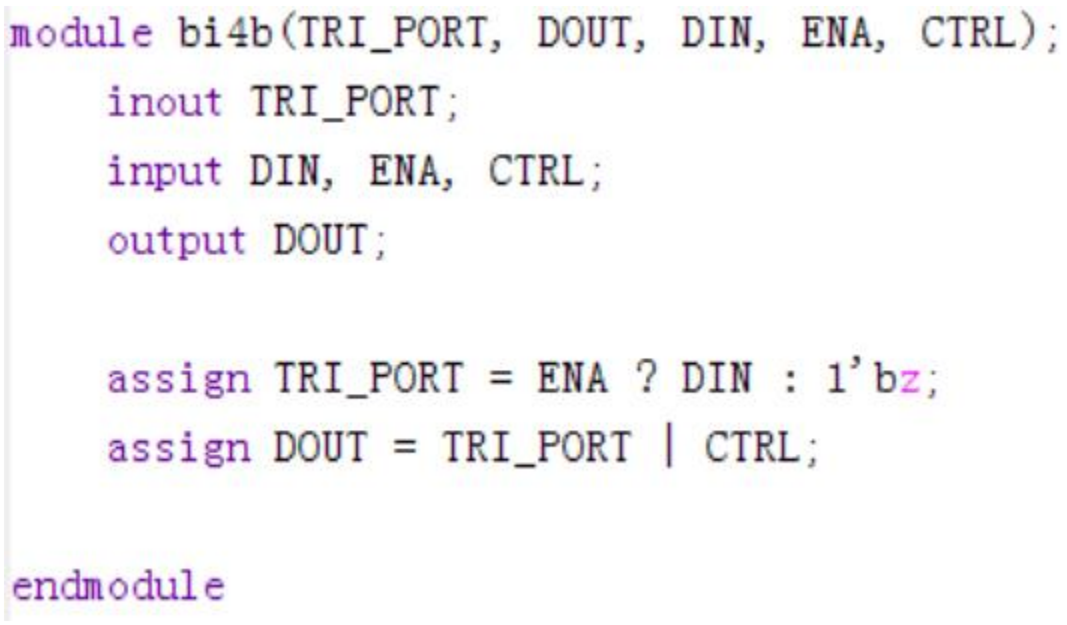
Describing Safe FSMs in Verilog HDL

- Explicitly describing a safe FSM
  - Include case item(s) to describe illegal states
  - Can use "default" case item
    - Executes if State equals anything other than S_Off, S_On1, or S_On2
- Note: Use of *default* is wise regardless of number of states
  - Even if number is power of two, because state encoding may use more than minimum number of bits
    - e.g., one-hot encoding has many more illegal states than legal states
- Note: If synthesis tool support "safe" option, *use it*
  - Otherwise, tool may automatically optimize away unreachable states to improve performance and size, but making state machine unsafe

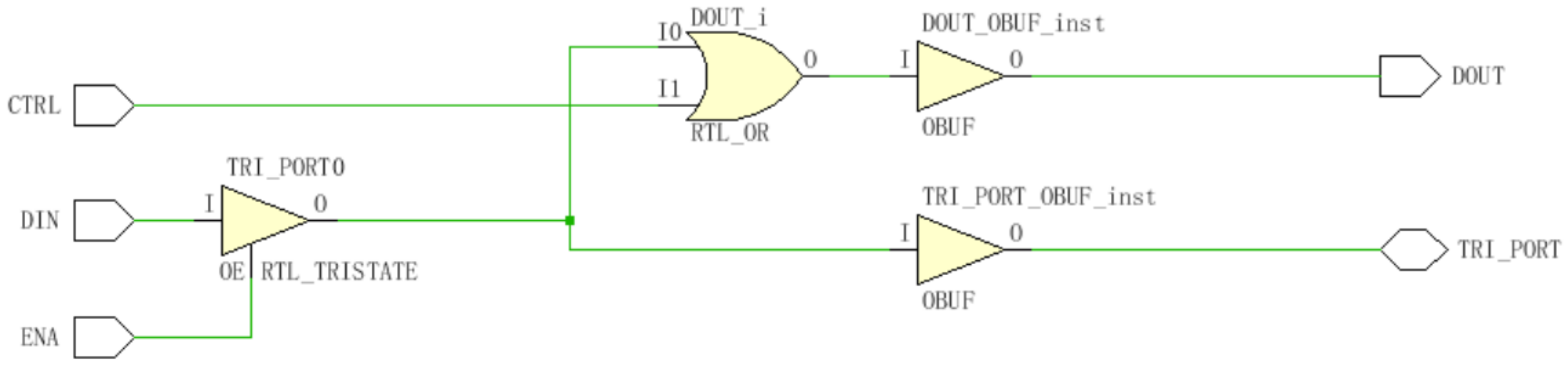
```
...
   reg [1:0] State, StateNext;

   always @(State, B) begin
     case (State)
        S_Off: begin
           X <= 0;
           if (B == 0)
              StateNext <= S_Off;
           else
              StateNext <= S_On1;
        end
        S_On1: begin
           X <= 1;
           StateNext <= S_On2;
        end
        S_On2: begin
           X <= 1;
           StateNext <= S_Off;
        end
        default: begin
           X <= 0;
           StateNext <= S_Off;
        end
     endcase
   end
...
```

Verilog设计深入

双向端口设计

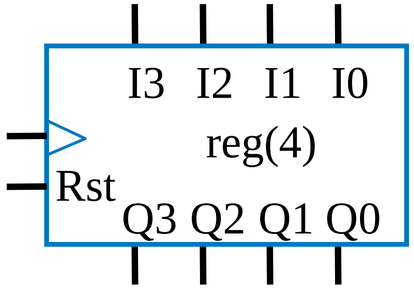


```
module bi4b(TRI_PORT, DOUT, DIN, ENA, CTRL);
    inout TRI_PORT;
    input DIN, ENA, CTRL;
    output DOUT;

    assign TRI_PORT = ENA ? DIN : 1'bz;
    assign DOUT = TRI_PORT | CTRL;

endmodule
```



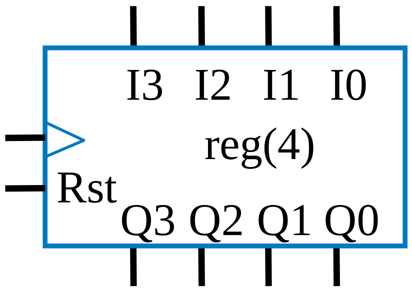
Resets

- Reset – Behavior of a register when a reset input is asserted
- Good practice dictates having defined reset behavior for every register
- Reset behavior should always have priority over normal register behavior
- Reset behavior
  - Usually clears register to 0s
  - May initialize to other value
    - e.g., state register of a controller may be initialized to encoding of initial state of FSM
- Reset usually asserted externally at start of sequential circuit operation, but also to restart due to failure, user request, or other reason

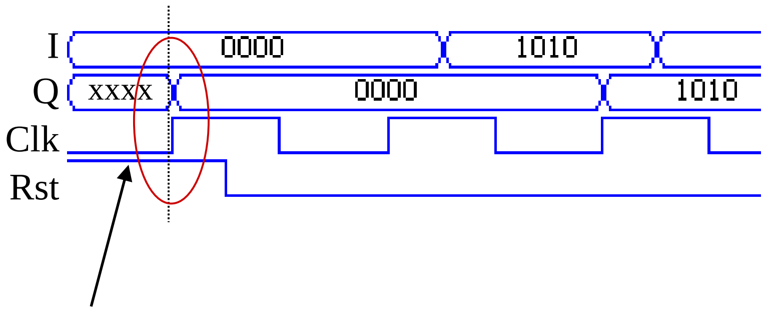


```
`timescale 1 ns/1 ns

module Reg4(I, Q, Clk, Rst);

   input [3:0] I;
   output [3:0] Q;
   reg [3:0] Q;
   input Clk, Rst;

   always @(posedge Clk) begin
      if (Rst == 1 )
        Q <= 4'b0000;
      else
        Q <= I;
   end
endmodule
```

Synchronous Reset



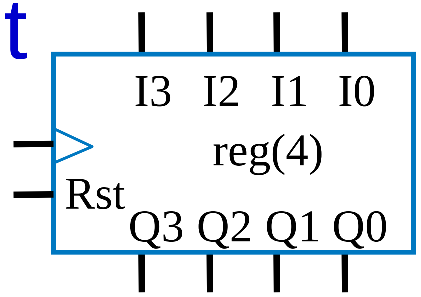
- Previous examples used synchronous resets
  - Rst input only considered during rising clock



Rst=1 has no effect until rising clock

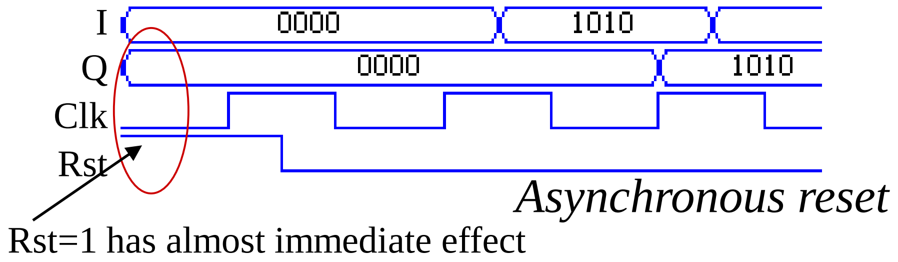
```
`timescale 1 ns/1 ns

module Reg4(I, Q, Clk, Rst);

   input [3:0] I;
   output [3:0] Q;
   reg [3:0] Q;
   input Clk, Rst;

   always @(posedge Clk) begin
      if (Rst == 1 )
         Q <= 4'b0000;
      else
         Q <= I;
   end
endmodule
```

Asynchronous Reset



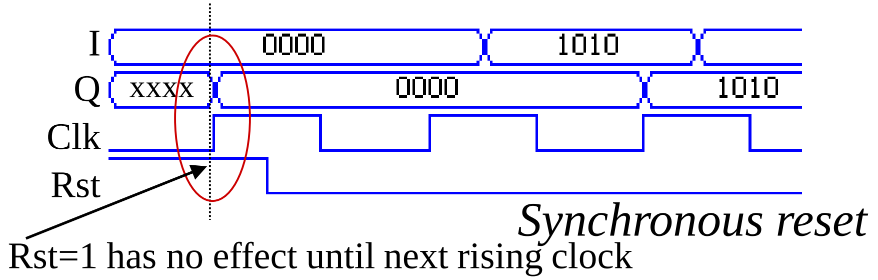
- Can also use asynchronous reset
  - Rst input considered independently from clock
    - Add "posedge Rst" to sensitivity list

```
`timescale 1 ns/1 ns

module Reg4(I, Q, Clk, Rst);

   input [3:0] I;
   output [3:0] Q;
   reg [3:0] Q;
   input Clk, Rst;

   always @(posedge Clk, posedge Rst) begin
      if (Rst == 1 )
         Q <= 4'b0000;
      else
         Q <= I;
   end
endmodule
```



Asynchronous reset

Rst=1 has almost immediate effect



Synchronous reset

Rst=1 has no effect until next rising clock

Asynchronous Reset

- Could have used asynchronous reset for FSM state register too

```
...
// StateReg
always @(posedge Clk) begin
  if (Rst == 1 )
    State <= S_Off;
  else
    State <= StateNext;
end
...
```

Synchronous

```
...
// StateReg
always @(posedge Clk, posedge Rst) begin
  if (Rst == 1 )
    State <= S_Off;
  else
    State <= StateNext;
end
...
```

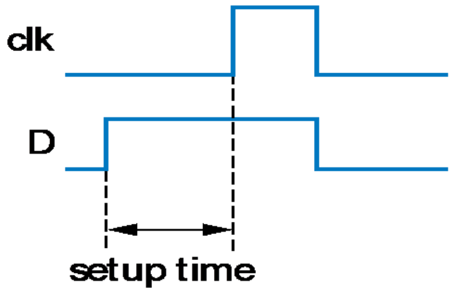
Asynchronous

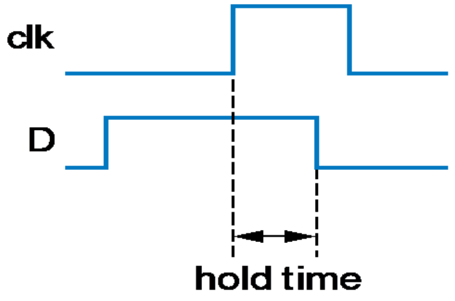
Synchronous versus Asynchronous Resets

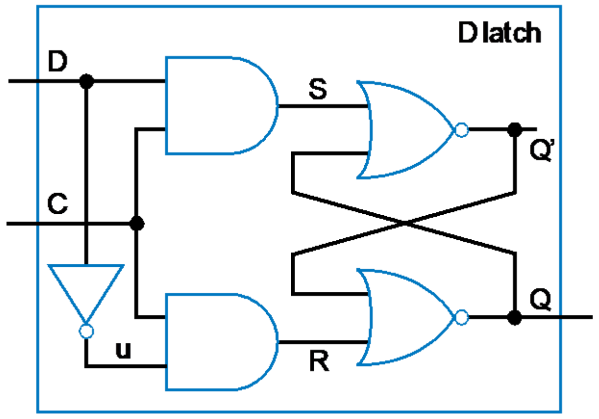
- Which is better – synchronous or asynchronous reset?
  - Hotly debated in design community
    - Each has pros and cons
      - e.g., asynchronous can still reset even if clock is not functioning, synchronous avoids timing analysis problems sometimes accompanying asynchronous designs
    - We won't try to settle the debate here
  - What's important is to be *consistent* throughout a design
    - All registers should have defined reset behavior that takes priority over normal register behavior
    - That behavior should all be synchronous reset or all be asynchronous reset
  - We will use synchronous resets in all of our remaining examples

Non-Ideal Flip-Flop Behavior

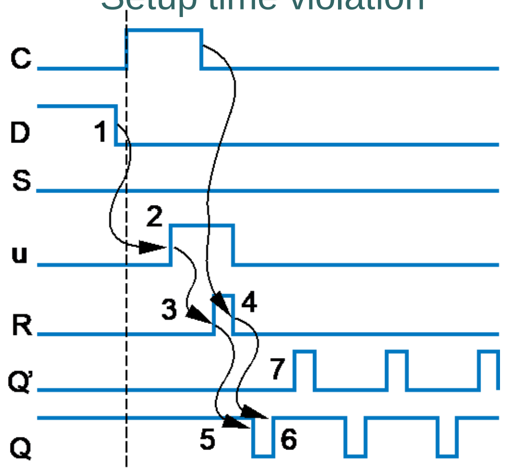
- Can't change flip-flop input too close to clock edge
  - Setup time: time that D must be stable *before* edge
    - Else, stable value not present at internal latch
  - Hold time: time that D must be held stable *after* edge
    - Else, new value doesn't have time to loop around and stabilize in internal latch







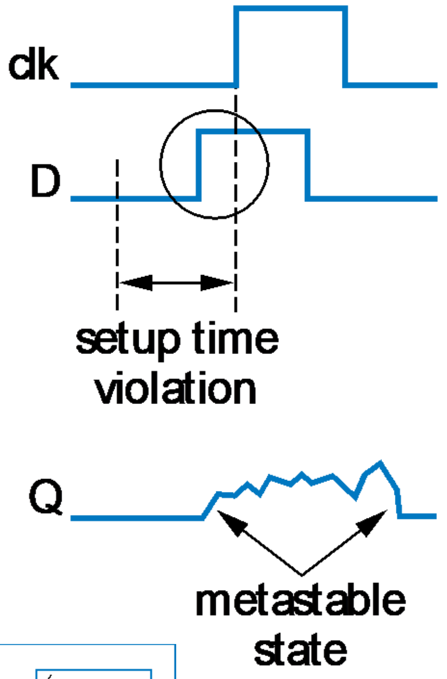
Setup time violation

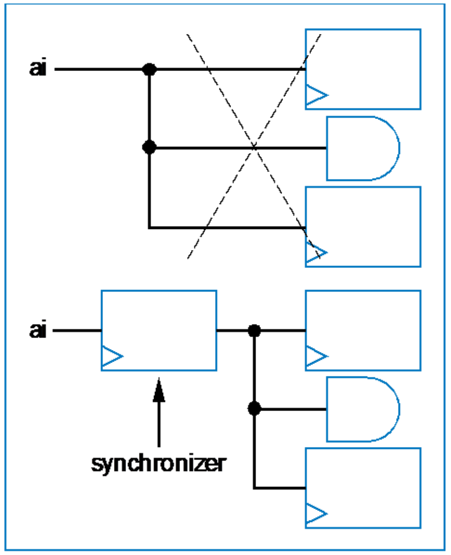


Leads to oscillation!

Metastability

- Violating setup/hold time can lead to bad situation known as ***metastable*** state
  - Metastable state: Any flip-flop state other than stable 1 or 0
    - Eventually settles to one or other, but we don't know which
  - For internal circuits, we can make sure observe setup time
  - But what if input comes from external (asynchronous) source, e.g., button press?
- Partial solution
  - Insert synchronizer flip-flop for asynchronous input
    - Special flip-flop with very small setup/hold time
  - Doesn't completely prevent metastability

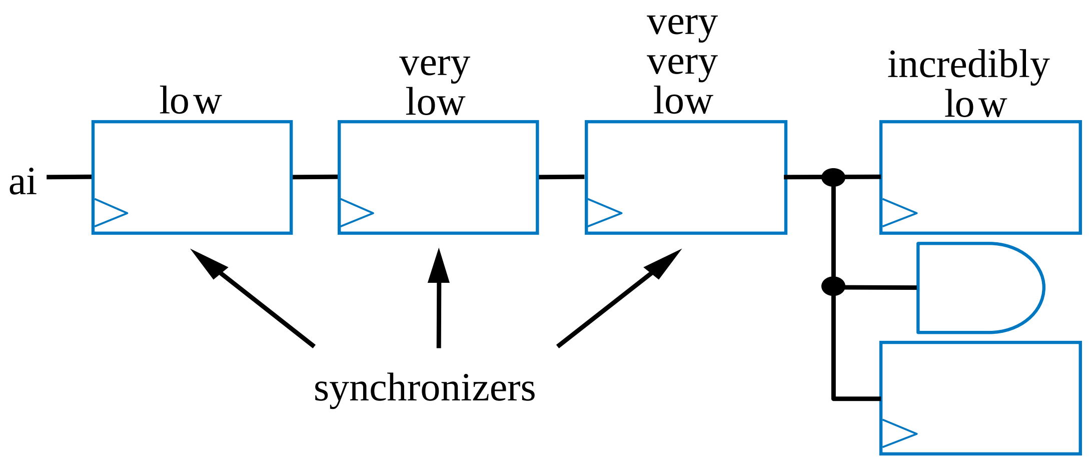




Metastability

- One flip-flop doesn't completely solve problem
- How about adding more synchronizer flip-flops?
  - Helps, but just decreases probability of metastability
- So how solve completely?
  - Can't! May be unsettling to new designers. But we just can't guarantee a design that won't ever be metastable. We can just minimize the mean time between failure (MTBF) -- a number often given along with a circuit

Probability of flip-flop being metastable is…



■ 资源优化之资源共享

【例 9-1】

```
module multmux (A0, A1, B, S, R);
  input[3:0] A0, A1, B;  input S;
  output[7:0] R;   reg[7:0] R;
  always @(A0 or A1 or B or S)
    begin
    if (S==1'b0)  R<=A0 * B;
      else  R<=A1 * B ;   end
endmodule
```

【例 9-2】

```
module multmux (A0, A1, B, S,R);
  input[3:0] A0, A1, B;  input S;
  output[7:0] R;   wire [7:0] R;
    reg [3:0] TEMP;
   always @(A0 or A1 or B or S)
   begin  if (S==1'b0)  TEMP<=A0;
    else  TEMP <= A1;   end
   assign R=TEMP * B;
endmodule
```

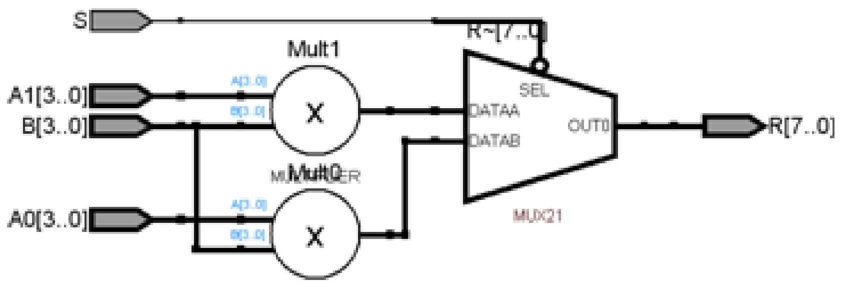


图 9-1 先乘后选择的设计方法 RTL 结构

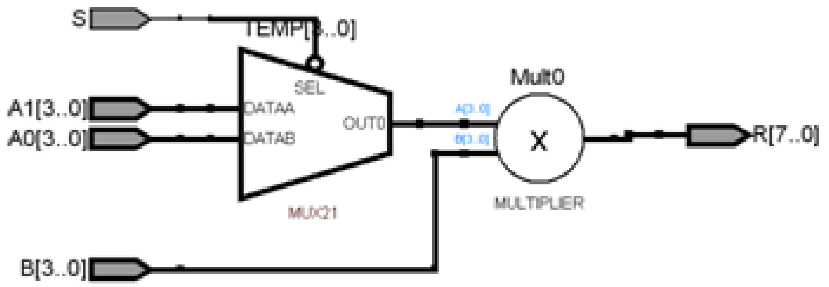
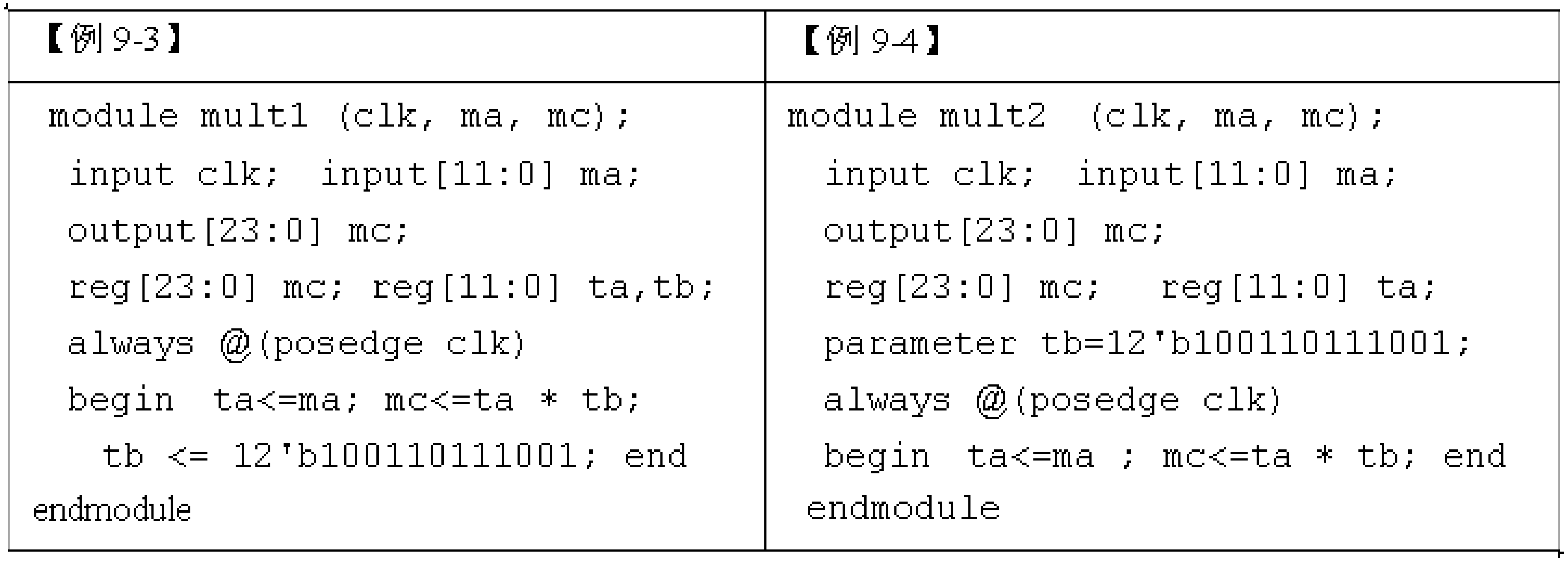


图 9-2　先选择后乘设计方法 RTL 结构

■ 资源优化之逻辑优化



【例 9-3】	【例 9-4】

`module mult1 (clk, ma, mc);`
`  input clk;  input[11:0] ma;`
`  output[23:0] mc;`
`  reg[23:0] mc; reg[11:0] ta,tb;`
`  always @(posedge clk)`
`  begin  ta<=ma; mc<=ta * tb;`
`    tb <= 12'b100110111001; end`
`endmodule`
	`module mult2  (clk, ma, mc);`
	`  input clk;  input[11:0] ma;`
	`  output[23:0] mc;`
	`  reg[23:0] mc;   reg[11:0] ta;`
	`  parameter tb=12'b100110111001;`
	`  always @(posedge clk)`
	`  begin  ta<=ma ; mc<=ta * tb; end`
	` endmodule`

■ 速度优化之流水线技术

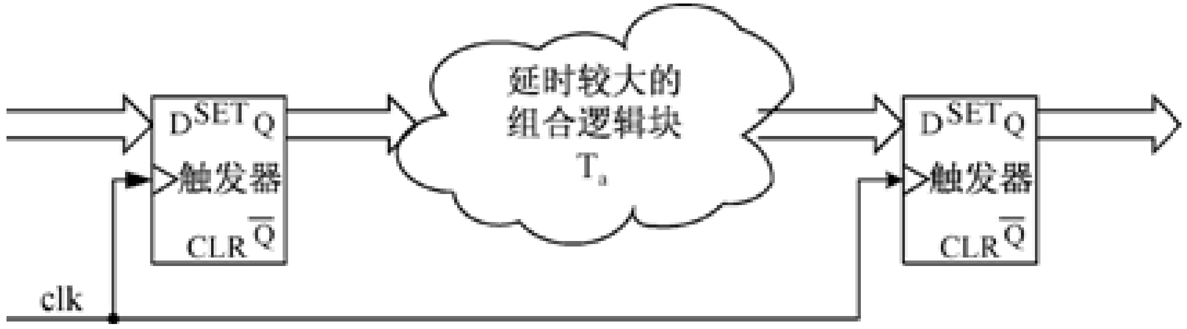


图 9-4 未使用流水线

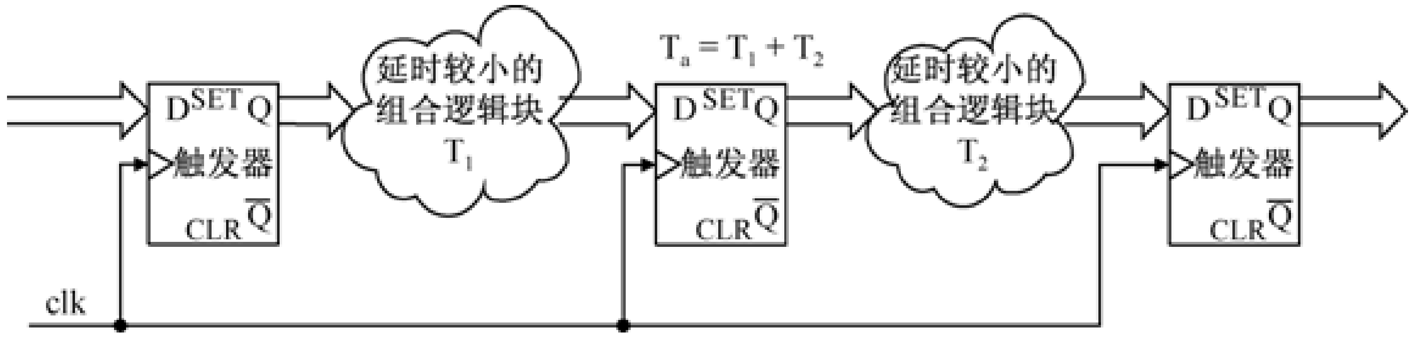


图 9-5 使用流水线结构

■ 速度优化之乒乓操作

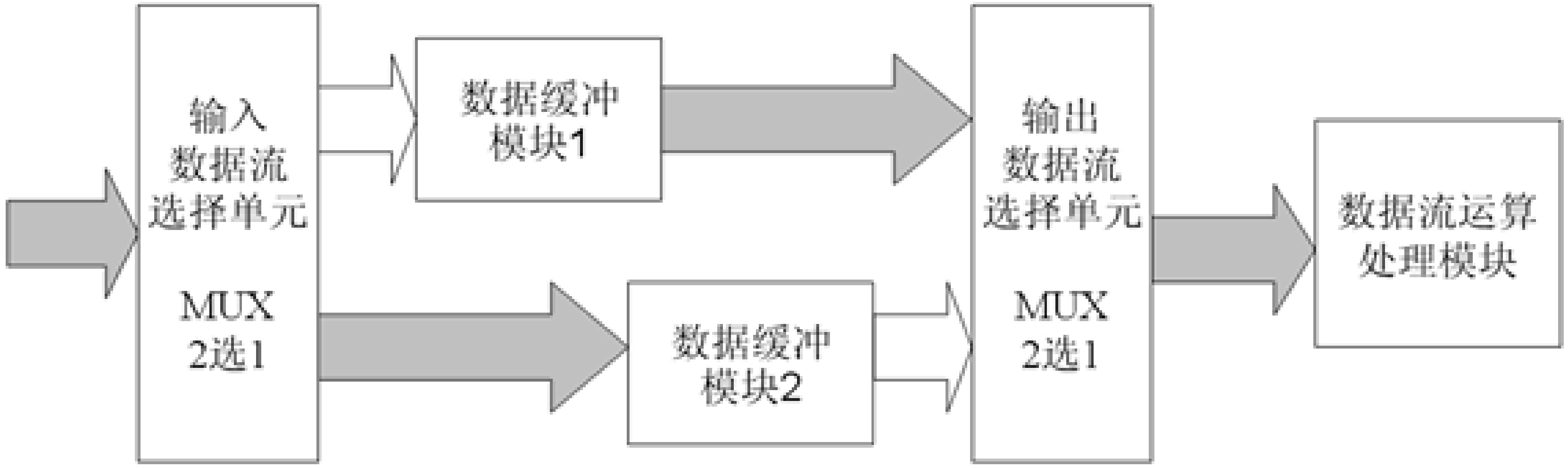


图 9-13 乒乓操作数据缓存结构示意图