数字设计和计算机体系结构

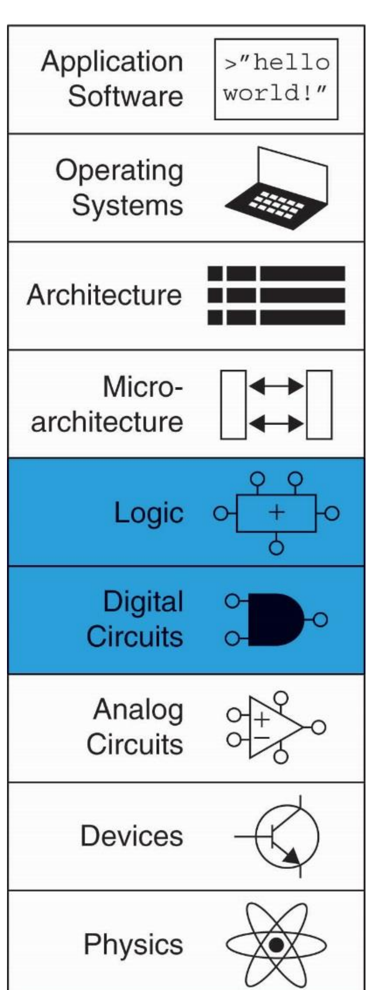
第四章 硬件描述语言HDL

刘龙军 副教授

2020年11月7日

第四章 硬件描述语言

- 介绍
- 组合逻辑
- 结构化模块
- 时序逻辑
- 更多组合逻辑
- 有限状态机
- 参数化模块
- 测试程序



介绍

- Hardware description language (HDL):
  - specifies logic function only
  - Computer-aided design (CAD) tool produces or *synthesizes* the optimized gates
- Most commercial designs built using HDLs
- Two leading HDLs:
  - **SystemVerilog**
    - developed in 1984 by Gateway Design Automation
    - IEEE standard (1364) in 1995
    - Extended in 2005 (IEEE STD 1800-2009)
  - **VHDL 2008**
    - Developed in 1981 by the Department of Defense
    - IEEE standard (1076) in 1987
    - Updated in 2008 (IEEE STD 1076-2008)

HDL to Gates

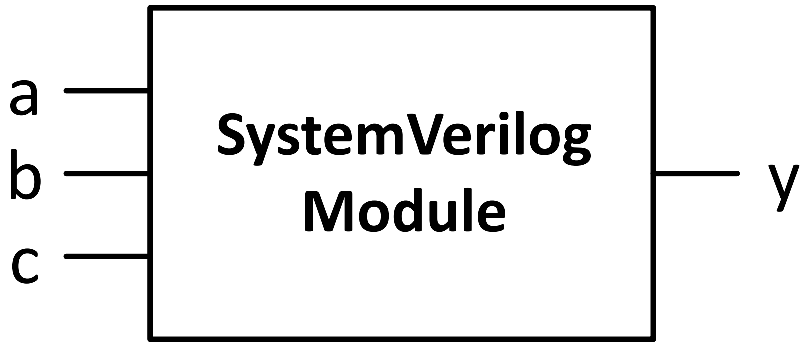
- 仿真
  - Inputs applied to circuit
  - Outputs checked for correctness
  - Millions of dollars saved by debugging in simulation instead of hardware
- 综合
  - Transforms HDL code into a *netlist* describing the hardware (i.e., a list of gates and the wires connecting them)

HDL to Gates

- 仿真
  - Inputs applied to circuit
  - Outputs checked for correctness
  - Millions of dollars saved by debugging in simulation instead of hardware
- 综合
  - Transforms HDL code into a *netlist* describing the hardware (i.e., a list of gates and the wires connecting them)

IMPORTANT: When using an HDL, think of the **hardware** the HDL should implies（当使用HDL时，要时时想到实现的硬件）

SystemVerilog Modules



Two types of Modules:

- **行为级:** describe what a module does
- **结构:** describe how it is built from simpler modules

行为级描述 SystemVerilog

SystemVerilog:

```
module example(input  logic a, b, c,
               output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

Behavioral SystemVerilog

SystemVerilog:

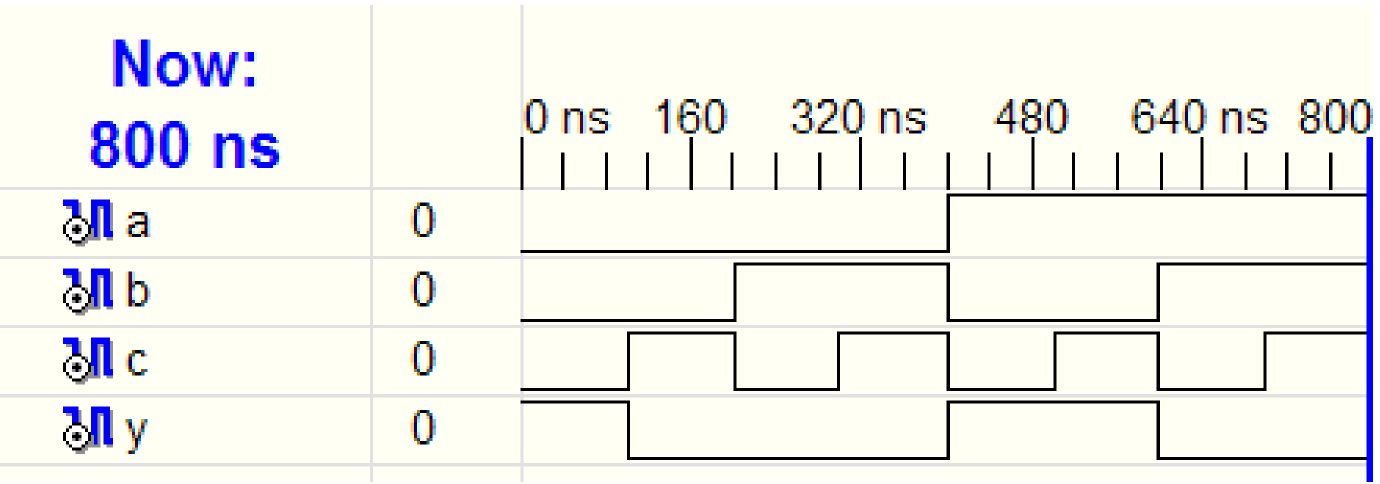
```
module example(input  logic a, b, c,
               output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

- `module/endmodule`: required to begin/end module
- `example`: name of the module
- Operators:
  - `~`: NOT
  - `&`: AND
  - `|`: OR

HDL 仿真

SystemVerilog:

```
module example(input  logic a, b, c,
                      output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

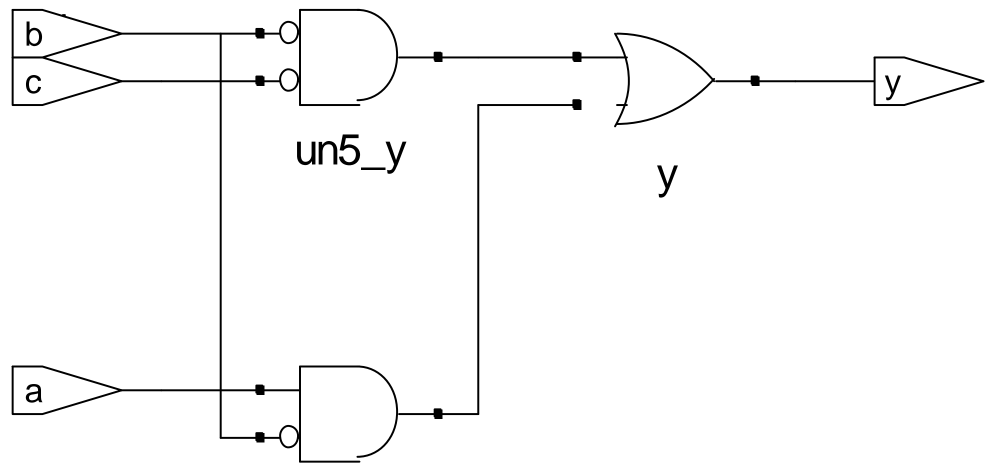


HDL 综合

SystemVerilog:

```
module example(input  logic a, b, c,
                      output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

Synthesis:



SystemVerilog 语法

- 大小写敏感的
  - **Example:** `reset` and `Reset` are not the same signal.
- 命名不能以数字开头
  - **Example:** `2mux` is an invalid name
- 空格被忽略
- 注释:
  - `// single line comment`
  - `/* multiline`

```
        comment */
```

Structural Modeling - 层次化

```
module and3(input  logic a, b, c,
            output logic y);
  assign y = a & b & c;
endmodule


module inv(input  logic a,
           output logic y);
  assign y = ~a;
endmodule


module nand3(input  logic a, b, c
             output logic y);
  logic n1;                   // 内部信号
  and3 andgate(a, b, c, n1);  // 实例：and3
  inv  inverter(n1, y);       // 实例：inv
endmodule
```

位操作符Operators

```
module gates(input  logic [3:0]  a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
   /* Five different two-input logic
      gates acting on 4 bit busses */
   assign y1 = a & b;    // AND
   assign y2 = a | b;    // OR
   assign y3 = a ^ b;    // XOR
   assign y4 = ~(a & b); // NAND
   assign y5 = ~(a | b); // NOR
endmodule
```

综合后:

`//` single line comment

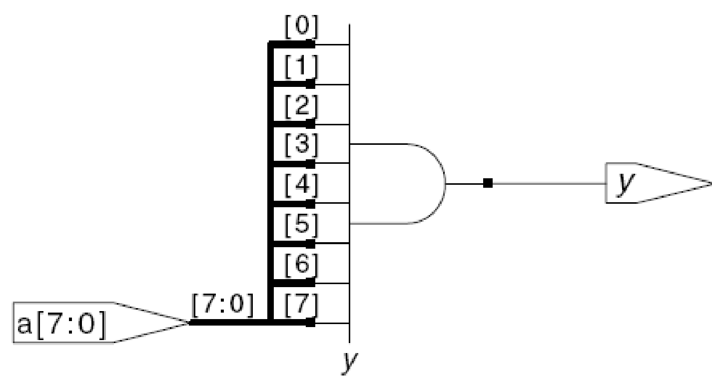
`/*…*/` multiline comment

位与运算Operators

SystemVerilog:

```
module and8(input  logic [7:0] a,
            output logic       y);
  assign y = &a;
  // &a is much easier to write than
  // assign y = a[7] & a[6] & a[5] & a[4] &
  //            a[3] & a[2] & a[1] & a[0];
endmodule
```

综合后:



条件赋值语句

SystemVerilog:

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);
   assign y = s ? d1 : d0;
endmodule
```

综合后:

s
d0[3:0]
d1[3:0]
[3:0]
[3:0]
0
1
[3:0]
y[3:0]
y[3:0]

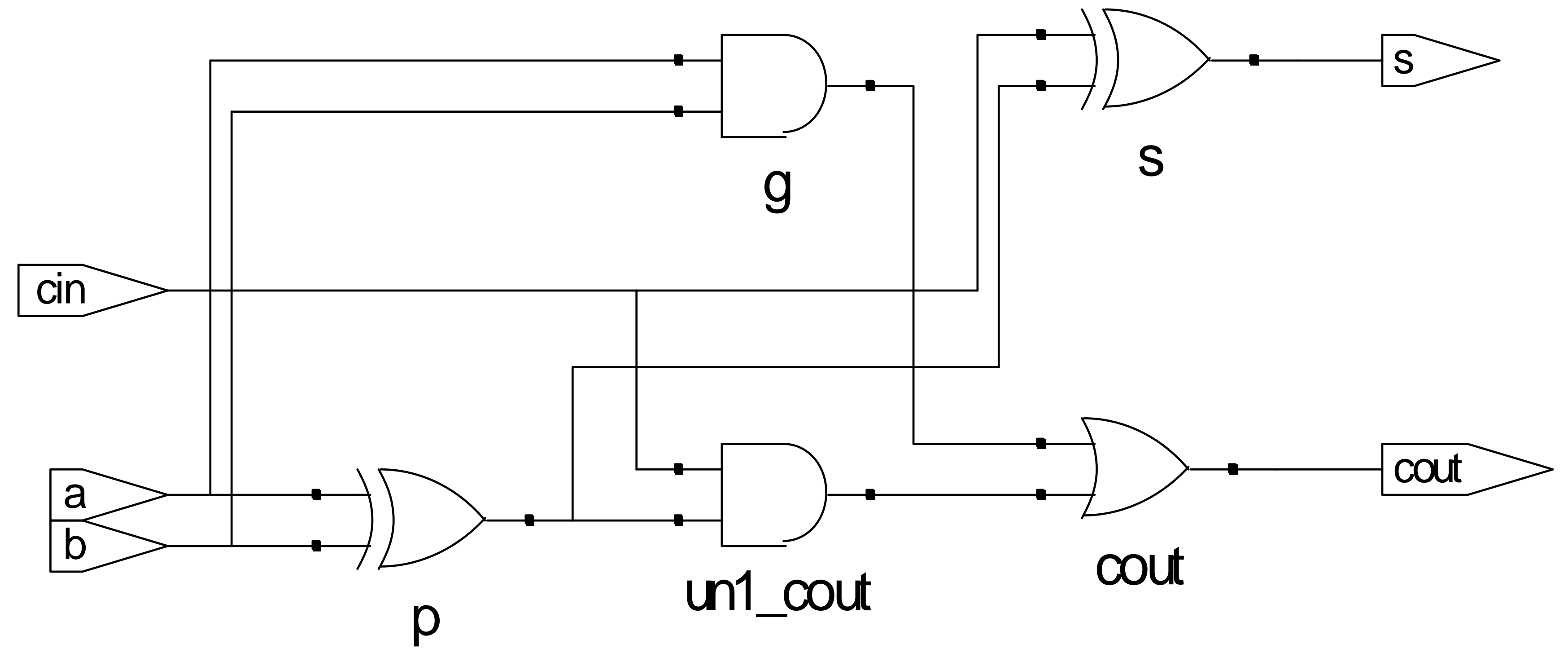
? : is also called a *ternary operator* because it operates on 3 inputs: `s`, `d1`, and `d0`.

内部变量

SystemVerilog:

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;   // internal nodes

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

Synthesis:


s
g
s
cin
cout
a
b
p
un1_cout
cout

测试模块：

```
module test_full_adder;
logic     a, b;
logic     cin;
initial  begin
              a = 1'b1; b=1'b1; cin=1'b1;
    #10   a = 1'b0; b=1'b0; cin=1'b0;
    #10   a = 1'b0; b=1'b1; cin=1'b1;
    #10   a = 1'b1; b=1'b0; cin=1'b1;
    #10   $stop;    end
full_adder   n1(a, b, cin, s, cout);
endmodule
```

优先级

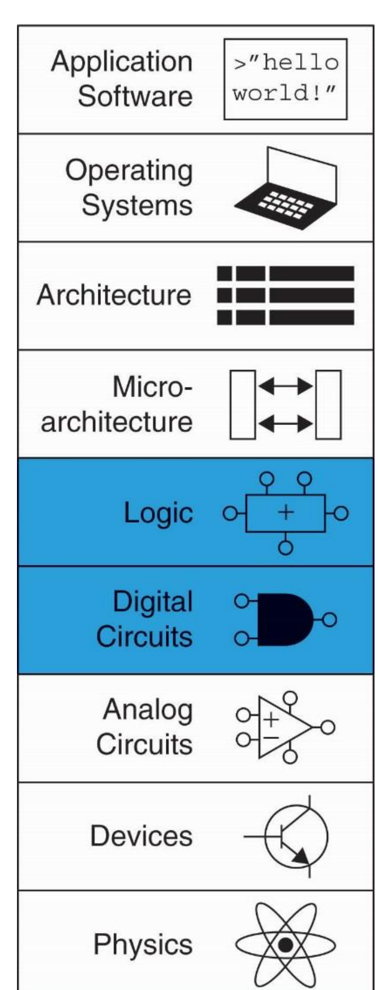
Highest

Operator	Meaning
~	NOT
*, /, %	mult, div, mod
+, -	add,sub
<<, >>	shift
<<<, >>>	arithmetic shift
<, <=, >, >=	comparison
==, !=	equal, not equal
&, ~&	AND, NAND
^, ~^	XOR, XNOR
|, ~|	OR, NOR
? :	ternary operator

Lowest

第四章 硬件描述语言

- 介绍
- 组合逻辑
- 结构化模块
- 时序逻辑
- 更多组合逻辑
- 有限状态机
- 参数化
- 测试程序



数制

Format: N'B value

N = number of bits, **B** = base

N'B is optional but recommended **(default is decimal)**

Number	# Bits	Base	Decimal Equivalent	Stored
3'b101	3	binary	5	101
'b11	unsized	binary	3	00…0011
8'b11	8	binary	3	00000011
8'b1010_1011	8	binary	171	10101011
3'd6	3	decimal	6	110
6'o42	6	octal	34	100010
8'hAB	8	hexadecimal	171	10101011
42	Unsized	decimal	42	00…0101010

位操作: Example 1

```
assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100_010};
```

```
// if y is a 12-bit signal, the above statement produces:
y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0
```

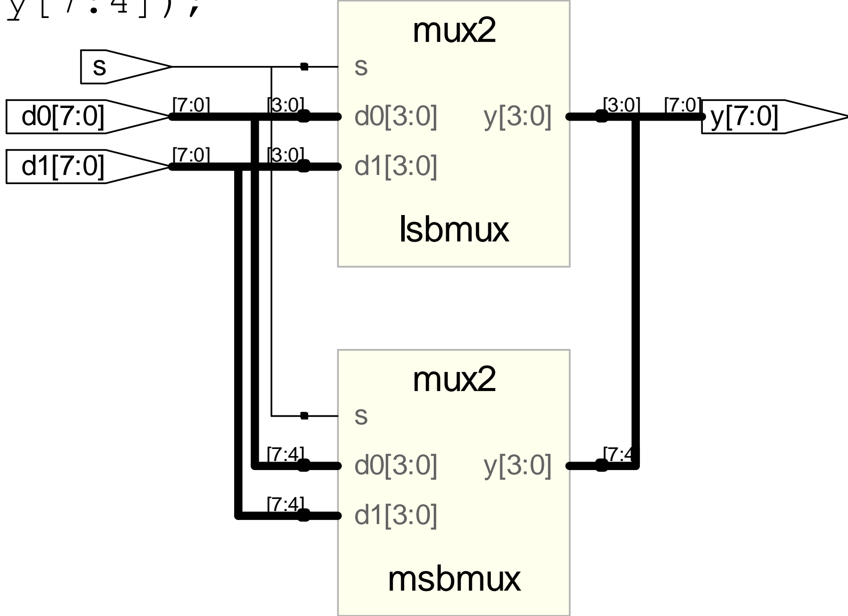
```
// 下划线  _  are used for formatting only to make
// it easier to read. SystemVerilog ignores them.
```

位操作: Example 2

SystemVerilog:

```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```

Synthesis:

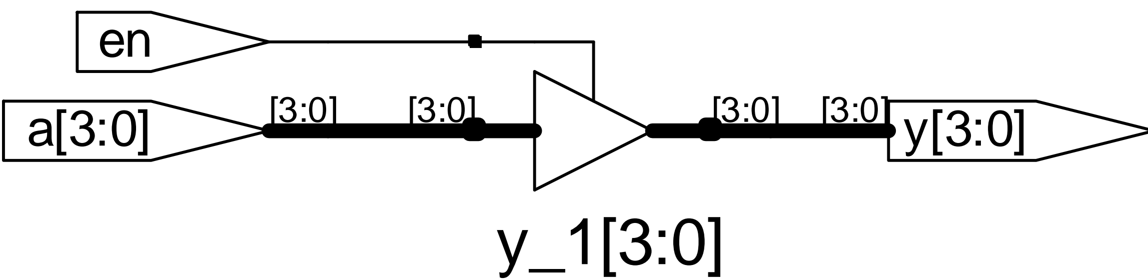


Z: 高阻输出

SystemVerilog:

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output tri   [3:0] y);
   assign y = en ? a : 4'bz;
endmodule
```

Synthesis:



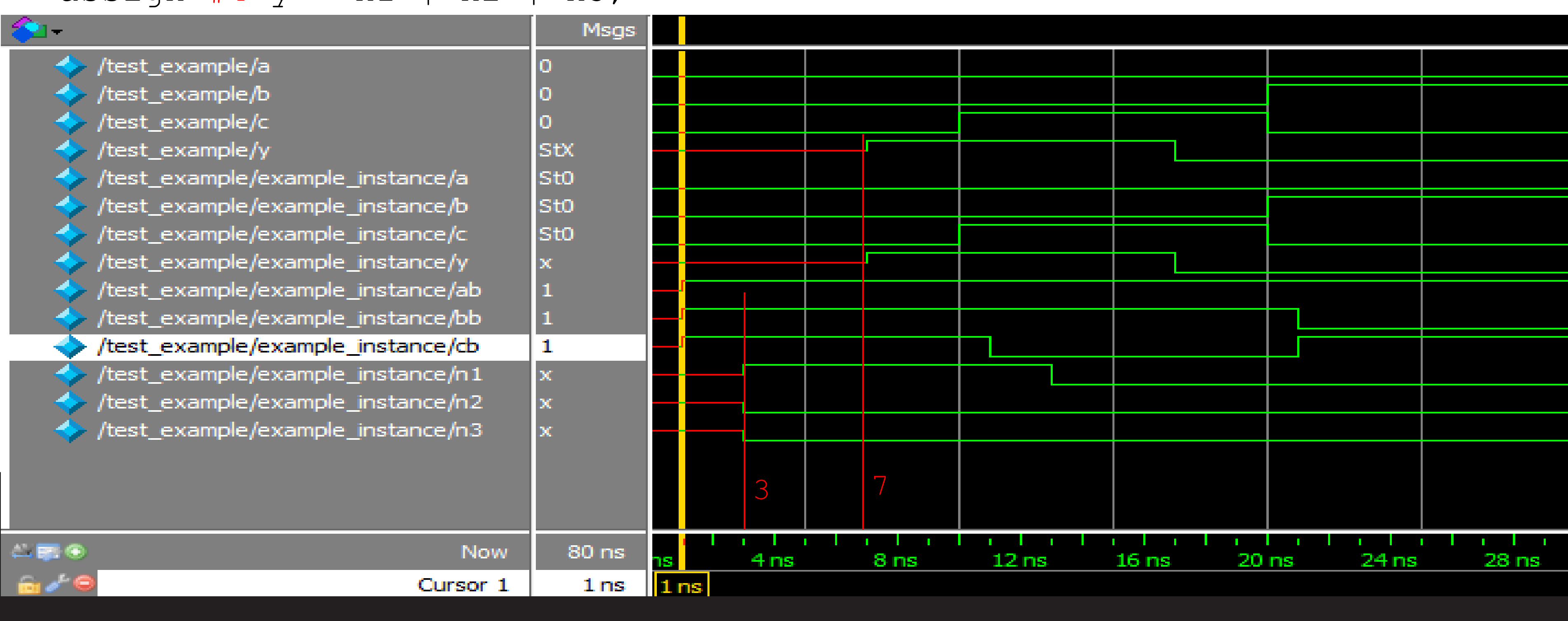
延迟设置

```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

```
module test_example;
reg       a,b,c;
initial  begin
          a = 1'b0; b=1'b0; c=1'b0;
     #10  a = 1'b0; b=1'b0; c=1'b1;
     #10  a = 1'b0; b=1'b1; c=1'b0;
     #10  a = 1'b0; b=1'b1; c=1'b1;
     #10  a = 1'b1; b=1'b0; c=1'b0;
     #10  a = 1'b1; b=1'b0; c=1'b1;
     #10  a = 1'b1; b=1'b1; c=1'b0;
     #10  a = 1'b1; b=1'b1; c=1'b1;
     #10  $stop;
end
example    example_instance(a, b, c, y);
endmodule
```

延迟设置

```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
```



时序逻辑

- SystemVerilog uses **idioms**（**always**） to describe latches, flip-flops and FSMs

always 语句

General Structure:

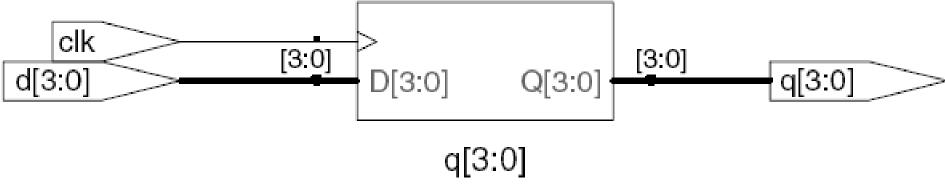
```
always @(sensitivity list)
    statement;
```

Whenever the event in **sensitivity list** occurs, **statement** is executed

D 触发器（自己写）

```
module flop(input  logic       clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff @(posedge clk)
    q <= d;              // pronounced "q gets（得到） d"

endmodule
```

Synthesis:



可复位的 D 触发器(Flip-Flop) （自己写）

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // 同步复位
  always_ff @(posedge clk)
    if (reset) q <= 4'b0;
    else       q <= d;

endmodule
```

Synthesis:

带复位 D Flip-Flop

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // 异步复位
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;

endmodule
```

Synthesis:

D Flip-Flop with 使能

```
module flopren(input  logic       clk,
               input  logic       reset,
               input  logic       en,
               input  logic [3:0] d,
               output logic [3:0] q);

  // 使能与异步复位
  always_ff @(posedge clk, posedge reset)
    if      (reset) q <= 4'b0;
    else if (en)    q <= d;

endmodule
```

综合后:

clk
d[3:0]
en
reset
[3:0]
[3:0]
D[3:0]
E
Q[3:0]
R
[3:0]
[3:0]
q[3:0]
q[3:0]

锁存器

```
module latch(input  logic       clk,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_latch
    if (clk) q <= d;

endmodule
```

综合后:

d[3:0]
clk
[3:0]
[3:0]
lat
D[3:0]
C
Q[3:0]
q[3:0]
[3:0]
[3:0]
q[3:0]

Warning: We don’t use latches in this text. But you might write code that inadvertently implies a latch. Check synthesized hardware – if it has latches in it, there’s an error.

小结

通用结构:

```
always @(sensitivity list)
    statement;
```

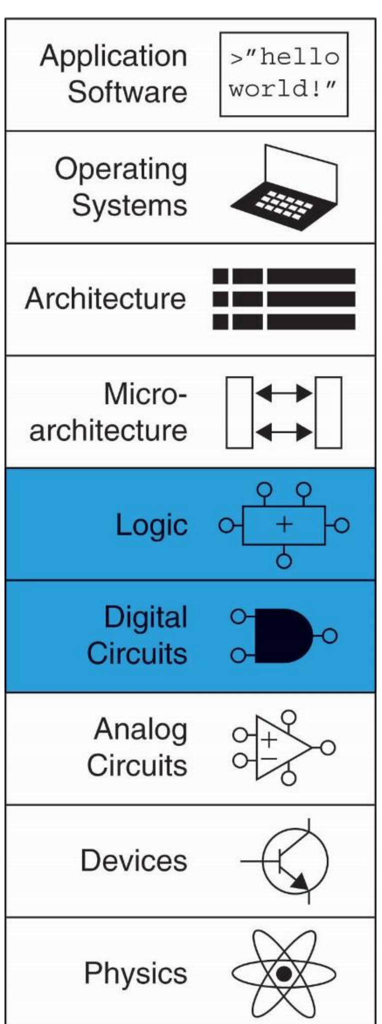
- **Flip-flop:** `always_ff`
- **Latch:** `always_latch` **(don’t use)**

其他行为语句（类似其他编程语言）

- Statements that must be inside `always` statements: （必须在always语句内）
  - `if/else`
  - `case`, `casez`

第四章 硬件描述语言

- 介绍
- 组合逻辑
- 结构化模块
- 时序逻辑
- 更多组合逻辑
- 有限状态机
- 参数化模块
- 测试程序



画出HDL代码描述的电路图，简化以便使用最少的门

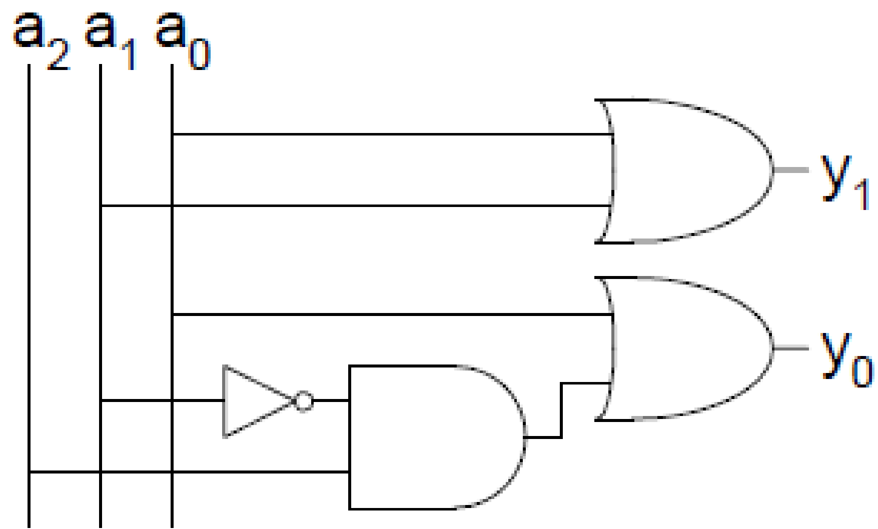
SystemVerilog

```
module exercise2(input  logic [3:0] a,
                 output logic [1:0] y);
  always_comb
    if      (a[0]) y = 2'b11;
    else if (a[1]) y = 2'b10;
    else if (a[2]) y = 2'b01;
    else if (a[3]) y = 2'b00;
    else           y = a[1:0];
endmodule
```

先画真值表，得到逻辑表达式，再化简。

$$Y_0 = a_0 + a_2\overline{a_1}\overline{a_0} = a_0 + a_2\overline{a_1}$$

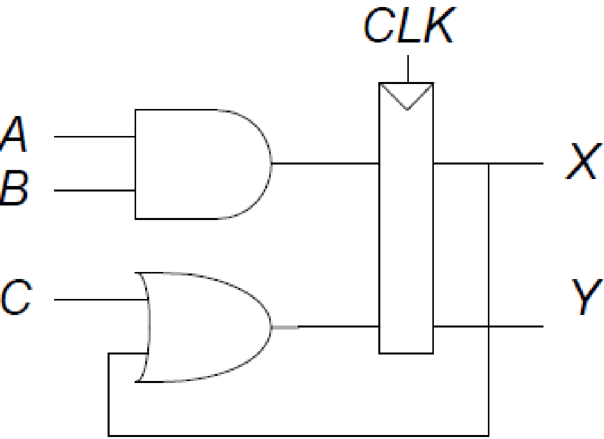
$$Y_1 = a_0 + a_1\overline{a_0} = a_0 + a_1$$



这两段代码功能一样吗？画出他们的硬件电路图

```
module code1(input  logic clk, a, b, c,
             output logic y);
  logic x;

  always_ff @(posedge clk) begin
    x <= a & b;
    y <= x | c;
  end
endmodule
```

```
module code2 (input  logic a, b, c, clk,
              output logic y);
  logic x;

  always_ff @(posedge clk) begin
    y <= x | c;
    x <= a & b;
  end
endmodule
```



```
module priority(input  logic [3:0] a,
                output logic [3:0] y);

  always_comb
    if      (a[3]) y = 4'b1000;
    else if (a[2]) y = 4'b0100;
    else if (a[1]) y = 4'b0010;
    else if (a[0]) y = 4'b0001;
endmodule
```

```
  always_comb
    if      (a[3]) y = 4'b1000;
    else if (a[2]) y = 4'b0100;
    else if (a[1]) y = 4'b0010;
    else if (a[0]) y = 4'b0001;
    else           y = 4'b0000;
endmodule
```

```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5)
always @(a)
  begin
    y1 = a & b;
    y2 = a | b;
    y3 = a ^ b;
    y4 = ~(a & b);
    y5 = ~(a | b);
  end
endmodule
```

```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);

  always_comb
    begin
      y1 = a & b;
      y2 = a | b;
      y3 = a ^ b;
      y4 = ~(a & b);
      y5 = ~(a | b);
    end
endmodule
```

组合逻辑中使用 always

```
// combinational logic using an always statement
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
  always_comb          // need begin/end because there is
    begin              // more than one statement in always
      y1 = a & b;      // AND
      y2 = a | b;      // OR
      y3 = a ^ b;      // XOR
      y4 = ~(a & b);   // NAND
      y5 = ~(a | b);   // NOR
    end
endmodule
```

This hardware could be described with assign statements using fewer lines of code, so it's better to use assign statements in this case.

组合逻辑中使用case

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);
  always_comb
    case (data)
      //                       abc_defg
      0: segments =      7'b111_1110;
      1: segments =      7'b011_0000;
      2: segments =      7'b110_1101;
      3: segments =      7'b111_1001;
      4: segments =      7'b011_0011;
      5: segments =      7'b101_1011;
      6: segments =      7'b101_1111;
      7: segments =      7'b111_0000;
      8: segments =      7'b111_1111;
      9: segments =      7'b111_0011;
      default: segments = 7'b000_0000; // required
    endcase
endmodule
```

组合逻辑中使用case

- `case` statement implies combinational logic **only if** all possible input combinations described
- Remember to use **`default`** statement

组合逻辑中使用 casez

```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);
  always_comb
    casez(a)
      4'b1???: y = 4'b1000; // ?=don't care
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
    endcase
endmodule
```

Synthesis:

阻塞blocking vs. 非阻塞赋值nonblocking

- <= is **nonblocking** assignment
  - Occurs simultaneously with others （并行）
- = is **blocking** assignment
  - Occurs in order it appears in file （串行）

```
// Good synchronizer using
// nonblocking assignments
module syncgood(input  logic clk,
                input  logic d,
                output logic q);
  logic n1;
  always_ff @(posedge clk)
    begin
      n1 <= d;  // nonblocking
      q  <= n1; // nonblocking
    end
endmodule
```

```
// Bad synchronizer using
// blocking assignments
module syncbad(input logic  clk,
               input  logic d,
               output logic q);
  logic n1;
  always_ff @(posedge clk)
    begin
      n1  = d;  // blocking
      q   = n1; // blocking
    end
endmodule
```

信号赋值规则

- **Synchronous sequential logic:** use **always_ff @(posedge clk)** and nonblocking assignments (**<=**)

```
always_ff @(posedge clk)
  q <= d; // nonblocking
```

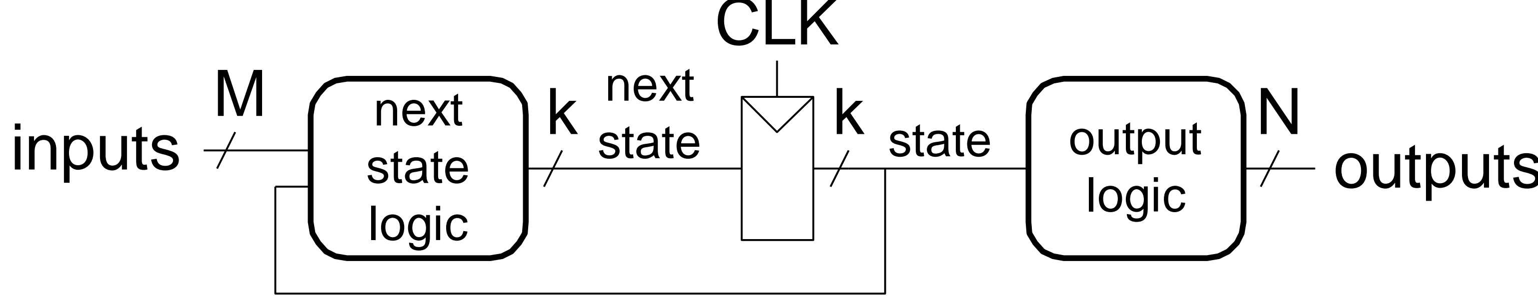
- **Simple combinational logic:** use continuous assignments (**assign…**)

```
assign y = a & b;
```

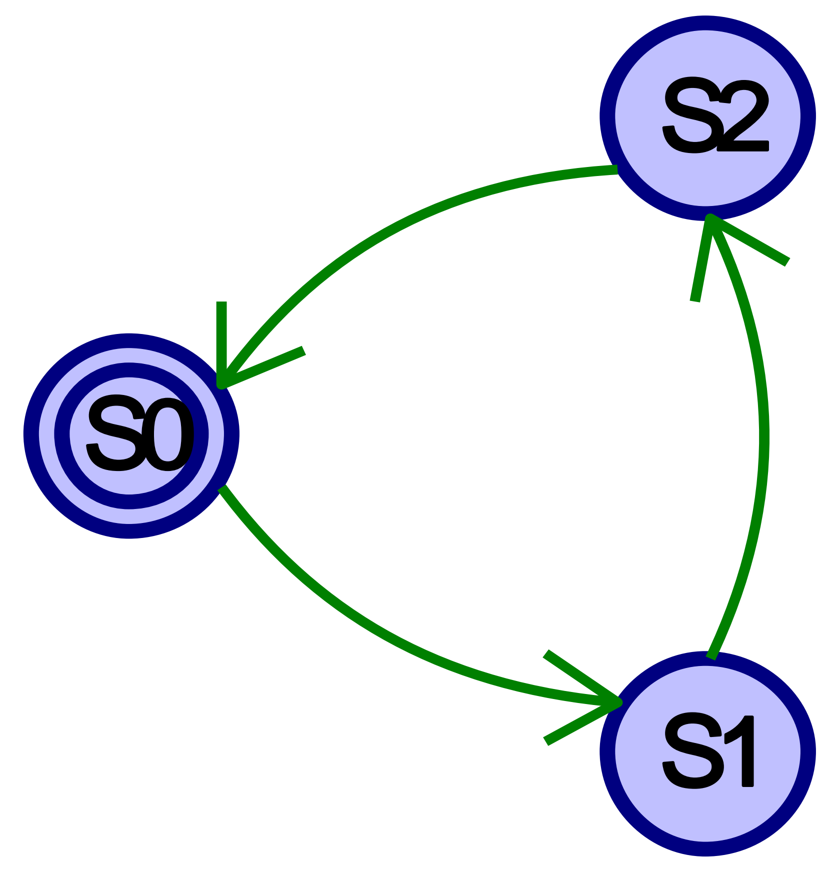
- **More complicated combinational logic:** use **always_comb** and blocking assignments (**=**)
- Assign a signal in **only one** `always` statement or continuous assignment statement.

有限状态机 (FSMs)

- 三大模块:
  - next state logic
  - state register
  - output logic



FSM Example: Divide by 3

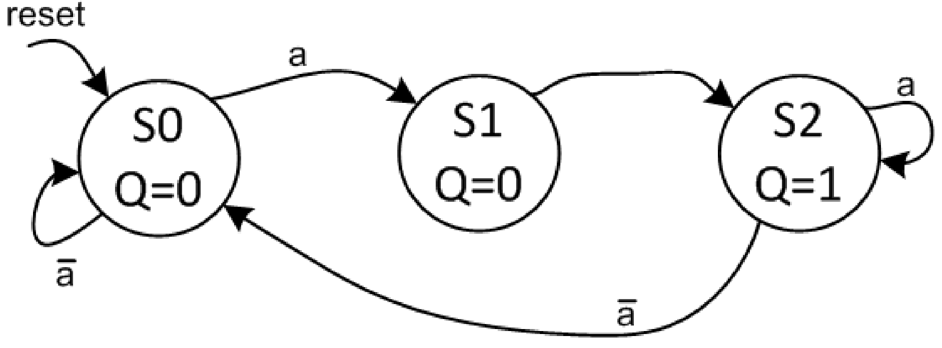


The double circle indicates the reset state

FSM in SystemVerilog

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic q);

   typedef enum logic [1:0] {S0, S1, S2} statetype;
   statetype state, nextstate;

   // state register
   always_ff @(posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // next state logic
   always_comb
      case (state)
         S0:      nextstate = S1;
         S1:      nextstate = S2;
         S2:      nextstate = S0;
         default: nextstate = S0;
      endcase

   // output logic
   assign q = (state == S0);
endmodule
```

带输入的FSM



看FSM状态图，写出Systemverilog代码

带输入的FSM

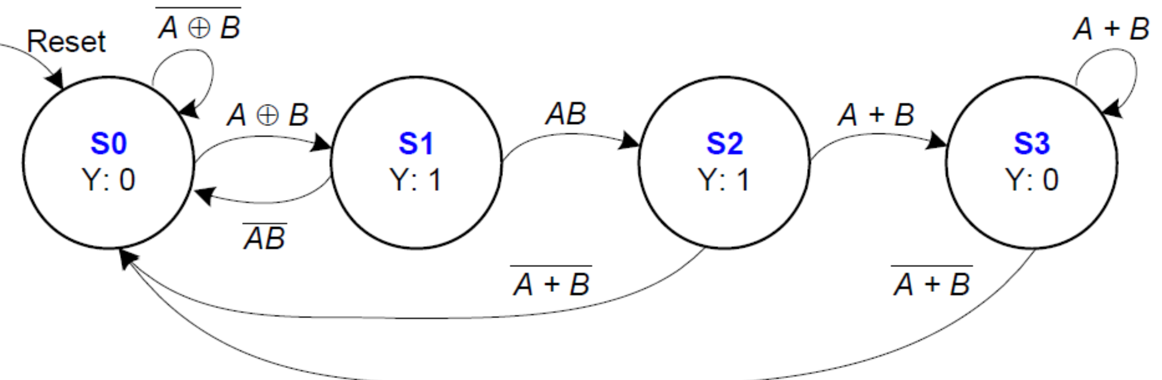
```
module fsmWithInputs(input  logic clk,
                     input  logic reset,
                     input  logic a,
                     output logic q);

   typedef enum logic [1:0] {S0, S1, S2} statetype;
   statetype state, nextstate;

   // state register
   always_ff @(posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // next state logic
   always_comb
      case (state)
         S0:      if (a) nextstate = S1;
                  else   nextstate = S0;
         S1:             nextstate = S2;
         S2:      if (a) nextstate = S2;
                  else   nextstate = S0;
         default:        nextstate = S0;
      endcase

   // output logic
   assign q = (state == S2);
endmodule
```


看FSM状态图，写出Systemverilog代码

按照HDL代码画出FSM1

SystemVerilog

```
module fsm2(input  logic clk, reset,
            input  logic a, b,
            output logic y);

  logic [1:0] state, nextstate;

  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;
  parameter S3 = 2'b11;

  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  always_comb
    case (state)
      S0: if (a ^ b) nextstate = S1;
          else       nextstate = S0;
      S1: if (a & b) nextstate = S2;
          else       nextstate = S0;
      S2: if (a | b) nextstate = S3;
          else       nextstate = S0;
      S3: if (a | b) nextstate = S3;
          else       nextstate = S0;
    endcase

  assign y = (state == S1) | (state == S2);
endmodule
```



```
module fsm1(input  logic clk, reset,
            input  logic taken, back,
            output logic predicttaken);

  logic [4:0] state, nextstate;

  parameter  S0  =  5'b00001;
  parameter  SI  =  5'b00010;
  parameter  S2  =  5'b00100;
  parameter  S3  =  5'b01000;
  parameter  S4  =  5'b10000;

  always_ff @(posedge clk, posedge  reset)
    if (reset) state <= S2;
    else       state <= nextstate;

  always_comb
    case (state)
      S0: if (taken)  nextstate = S1;
          else        nextstate = S0;
      S1: if (taken)  nextstate = S2;
          else        nextstate = S0;
      S2: if (taken)  nextstate = S3;
          else        nextstate = S1;
      S3: if (taken)  nextstate = S4;
          else        nextstate = S2;
      S4: if (taken)  nextstate = S4;
          else        nextstate = S3;
      default:        nextstate = S2;
    endcase

  assign predicttaken = (state  ==  S4)  |
                        (state  ==  S3)  |
                        (state  ==  S2 && back);
endmodule
```

出FSM2

ystemverilog代码，画出FSM状态图

按照HDL代码画出FSM2

```
module fsm1(input  logic clk, reset,
            input  logic taken, back,
            output logic predicttaken);

  logic [4:0] state, nextstate;

  parameter  S0  =  5'b00001;
  parameter  SI  =  5'b00010;
  parameter  S2  =  5'b00100;
  parameter  S3  =  5'b01000;
  parameter  S4  =  5'b10000;

  always_ff @(posedge clk, posedge  reset)
    if (reset) state <= S2;
    else       state <= nextstate;

  always_comb
    case (state)
      S0: if (taken)  nextstate = S1;
          else        nextstate = S0;
      S1: if (taken)  nextstate = S2;
          else        nextstate = S0;
      S2: if (taken)  nextstate = S3;
          else        nextstate = S1;
      S3: if (taken)  nextstate = S4;
          else        nextstate = S2;
      S4: if (taken)  nextstate = S4;
          else        nextstate = S3;
      default:        nextstate = S2;
    endcase

  assign predicttaken = (state  ==  S4)  |
                        (state  ==  S3)  |
                        (state  ==  S2 && back);
endmodule
```

reset
S0
S1
S2
if (back)
predicttaken
S3
predicttaken
S4
predicttaken
taken
$\overline{\text{taken}}$
taken
$\overline{\text{taken}}$
taken
$\overline{\text{taken}}$
taken
$\overline{\text{taken}}$
taken
$\overline{\text{taken}}$
taken

参数化模块

2:1 mux:

```
module mux2
  #(parameter width = 8)   // name and default value
   (input  logic [width-1:0] d0, d1,
    input  logic             s,
    output logic [width-1:0] y);
  assign y = s ? d1 : d0;
endmodule
```

Instance with 8-bit bus width (uses default):

```
  mux2 myMux(d0, d1, s, out);
```

Instance with 12-bit bus width:

```
   mux2 #(12) lowmux(d0, d1, s, out);
```

测试程序

- HDL that tests another module: *device under test* (dut)
- 不能综合 Not synthesizeable
- Uses different features of SystemVerilog
- Types:
  - Simple
  - Self-checking
  - Self-checking with testvectors

测试程序例子

- Write SystemVerilog code to implement the following function in hardware:

$$y = \overline{b}\,\overline{c} + a\overline{b}$$

- Name the module `sillyfunction`

Testbench Example

- Write SystemVerilog code to implement the following function in hardware:

$$y = \overline{b}\,\overline{c} + a\overline{b}$$

```
module sillyfunction(input  logic a, b, c,
                     output logic y);
  assign y = ~b & ~c | a & ~b;
endmodule
```

简单的 Testbench

```
module testbench1();
  logic a, b, c;
  logic y;
  // instantiate device under test
  sillyfunction dut(a, b, c, y);
  // apply inputs one at a time
  initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
    a = 1; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
  end
endmodule
```

带自检查的 Testbench

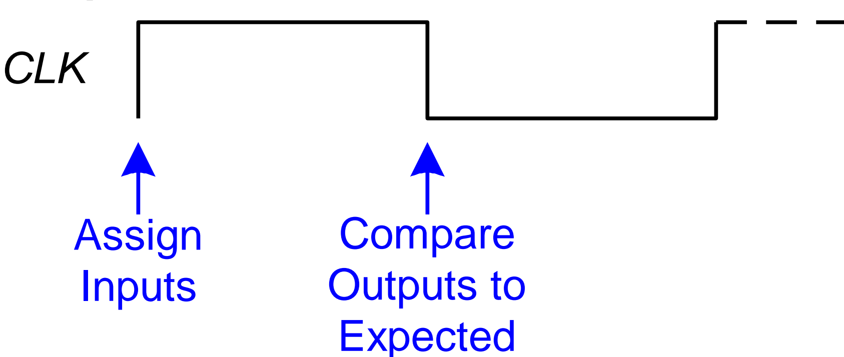
```
module testbench2();
  logic  a, b, c;
  logic y;
  sillyfunction dut(a, b, c, y);  // instantiate dut
  initial begin // apply inputs, check results one at a time
    a = 0; b = 0; c = 0; #10;
    if (y !== 1) $display("000 failed.");
    c = 1; #10;
    if (y !== 0) $display("001 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("010 failed.");
    c = 1; #10;
    if (y !== 0) $display("011 failed.");
    a = 1; b = 0; c = 0; #10;
    if (y !== 1) $display("100 failed.");
    c = 1; #10;
    if (y !== 1) $display("101 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("110 failed.");
    c = 1; #10;
    if (y !== 0) $display("111 failed.");
  end
endmodule
```

Testbench with 测试向量

- Testvector file: inputs and expected outputs
- Testbench:
  1. Generate clock for assigning inputs, reading outputs
  2. Read testvectors file into array
  3. Assign inputs, expected outputs
  4. Compare outputs with expected outputs and report errors

Testbench with 测试向量

- Testbench clock:
  - assign inputs (on rising edge)
  - compare outputs with expected outputs (on falling edge).



- Testbench clock also used as clock for synchronous sequential circuits

Testvectors 文件

- File: `example.tv`
- contains vectors of abc_yexpected

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

1. 产生 Clock

```
module testbench3();
  logic        clk, reset;
  logic        a, b, c, yexpected;
  logic        y;
  logic [31:0] vectornum, errors;    // bookkeeping variables
  logic [3:0]  testvectors[10000:0]; // array of testvectors

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // generate clock
  always     // no sensitivity list, so it always executes
    begin
      clk = 1; #5; clk = 0; #5;
    end
```

2. Read Testvectors into Array

```
// at start of test, load vectors and pulse reset

initial
   begin
      $readmemb("example.tv", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #27; reset = 0;
   end


// Note: $readmemh reads testvector files written in
// hexadecimal
```

3. Assign Inputs & Expected Outputs

```
// apply test vectors on rising edge of clk
always @(posedge clk)
  begin
    #1; {a, b, c, yexpected} = testvectors[vectornum];
  end
```

4. Compare with Expected Outputs

```
// check results on falling edge of clk
   always @(negedge clk)
    if (~reset) begin // skip during reset
      if (y !== yexpected) begin
        $display("Error: inputs = %b", {a, b, c});
        $display("  outputs = %b (%b expected)",y,yexpected);
        errors = errors + 1;
      end

// Note: to print in hexadecimal, use %h. For example,
//       $display("Error: inputs = %h", {a, b, c});
```

4. Compare with Expected Outputs

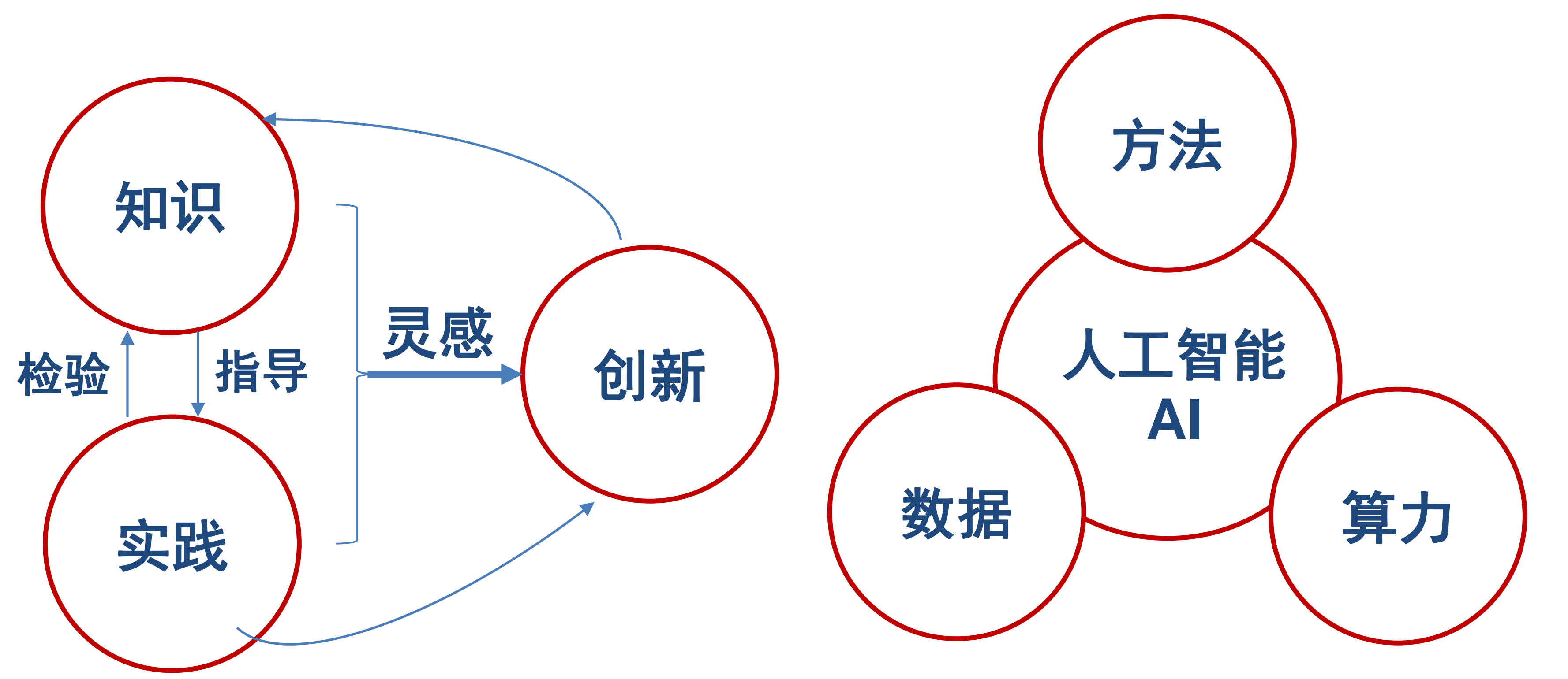
```
// increment array index and read next testvector
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 4'bx) begin
          $display("%d tests completed with %d errors",
                vectornum, errors);
        $stop;
      end
    end
endmodule


// === and !== can compare values that are 1, 0, x, or z.
```

本章结束

作业：2.38 3.29，并编写Systemverilog代码及Testbench。

其它课后题及自己练习编程

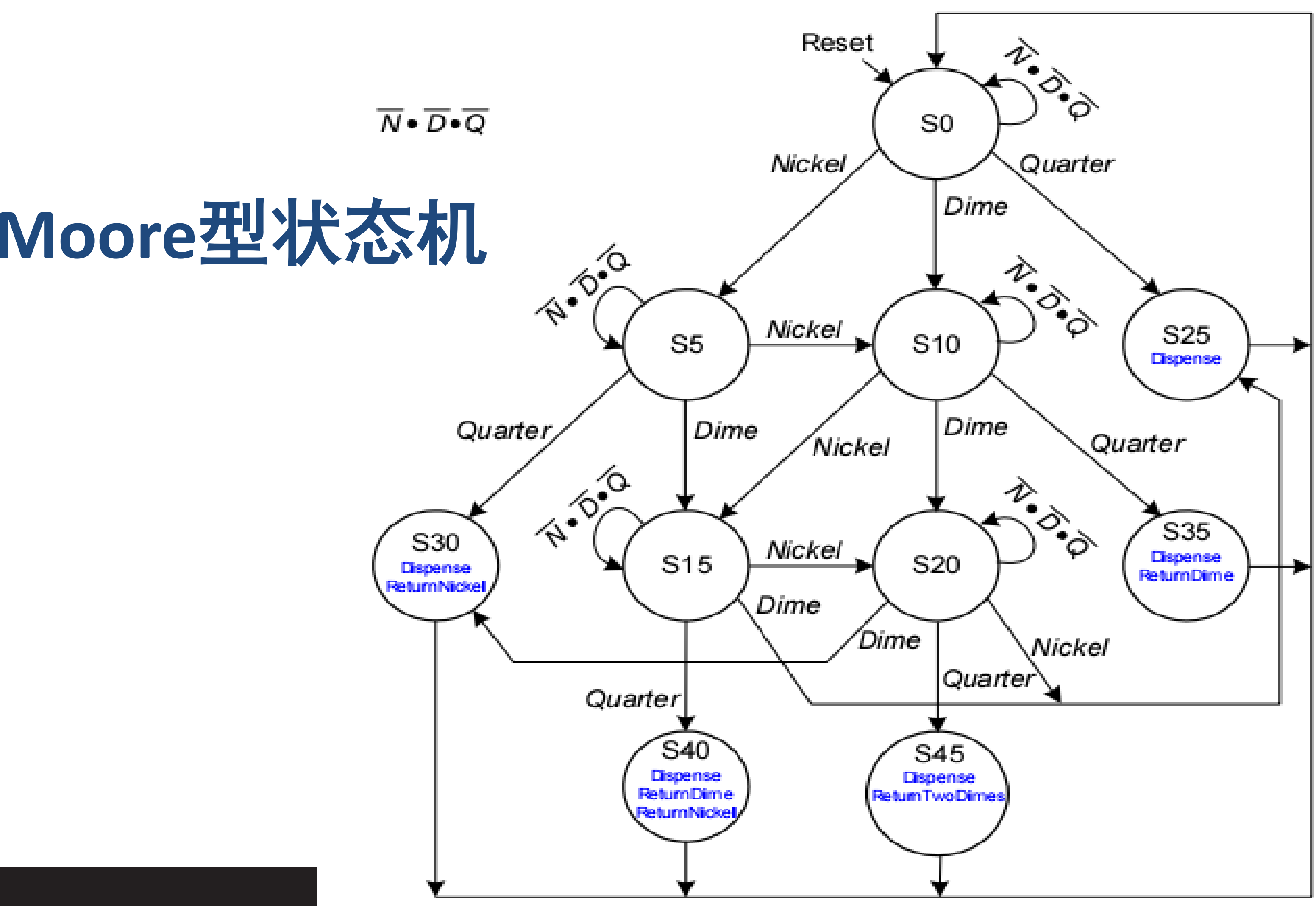
对创新的理解



生成式AI将会引起一场机器智能新的革命，创业黄金时期

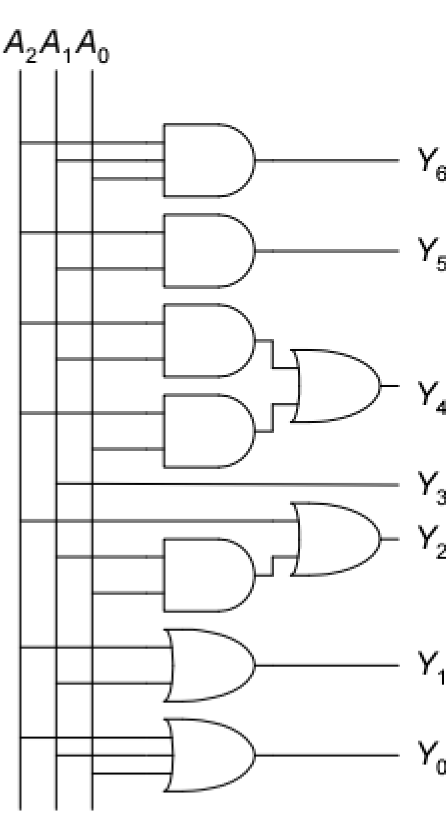
实验测试：作业题3.26

Moore型状态机



作业题2.38

$$Y_6 = A_2A_1A_0$$
$$Y_5 = A_2A_1$$
$$Y_4 = A_2A_1 + A_2A_0$$
$$Y_3 = A_2$$
$$Y_2 = A_2 + A_1A_0$$
$$Y_1 = A_2 + A_1$$
$$Y_0 = A_2 + A_1 + A_0$$



SystemVerilog

```
module thermometer(input  logic [2:0] a,
                   output logic [6:0] y);

  always_comb
    case (a)
      0: y = 7'b0000000;
      1: y = 7'b0000001;
      2: y = 7'b0000011;
      3: y = 7'b0000111;
      4: y = 7'b0001111;
      5: y = 7'b0011111;
      6: y = 7'b0111111;
      7: y = 7'b1111111;
    endcase
endmodule
```

作业题3.29

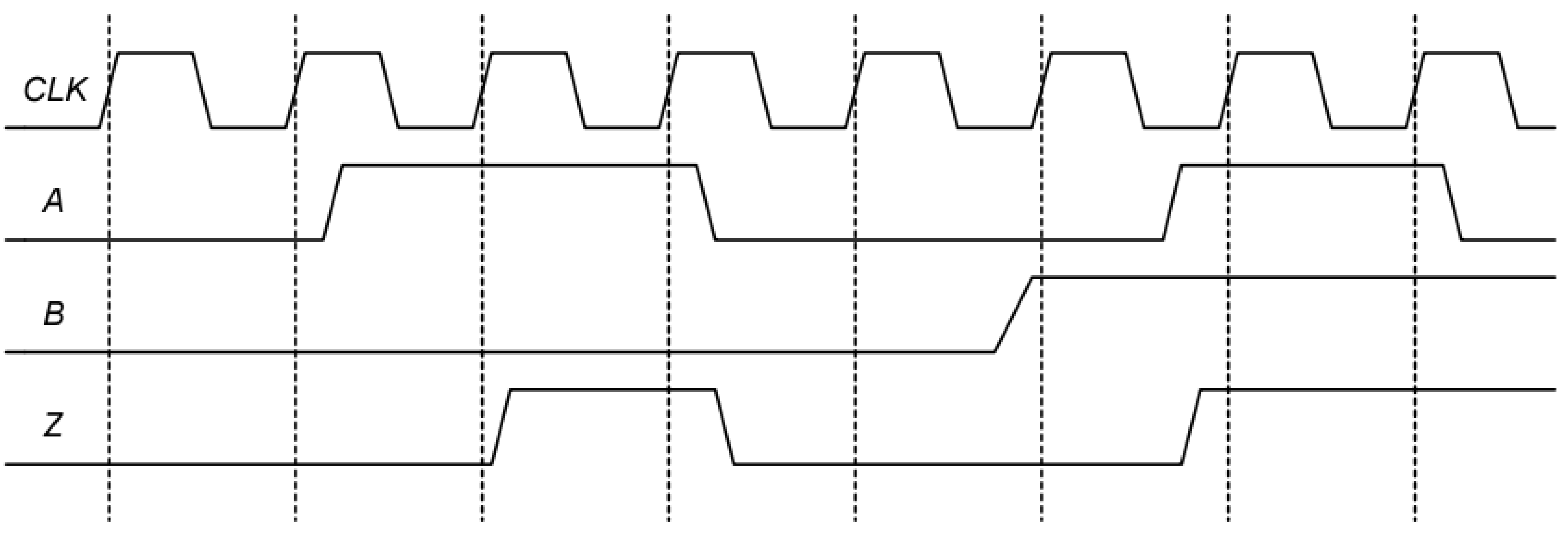
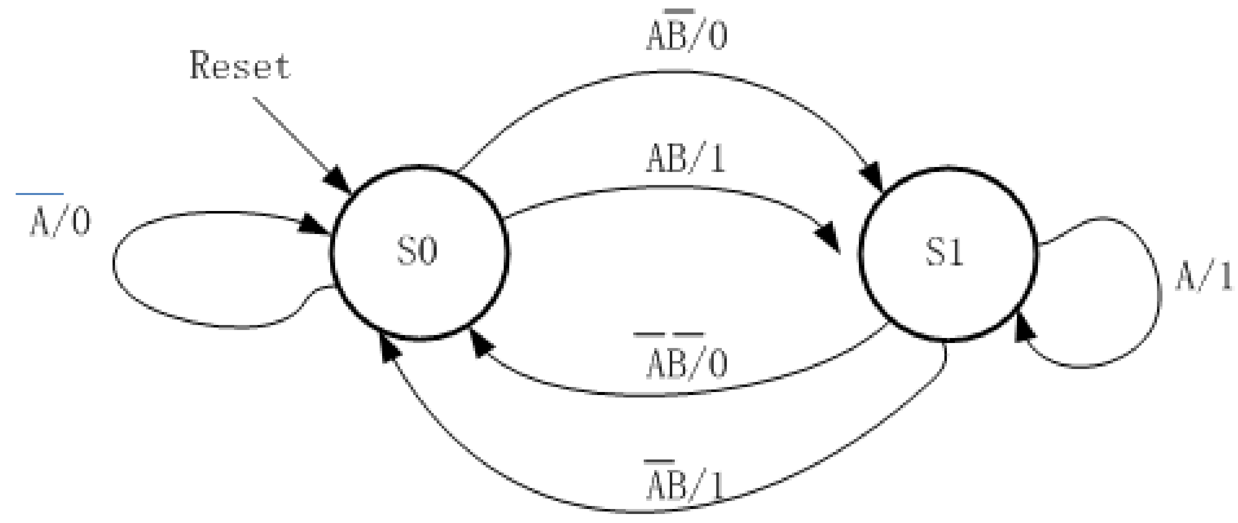


FIGURE 3.8 Waveform showing Z output for Exercise 3.29

作业题3.29

设状态表示为{An-1}，四个状态S0=0，S1=1

Mealy型状态机



$$Z=(A_n+A_{n-1})B+A_nA_{n-1}\&\overline{B}$$

$$Z= AB+S_0(A+B)$$

作业题3.29

Moore型状态机

设状态表示为{An-1，An}，四个状态S0=00，S1=01，S2=10，S3=11