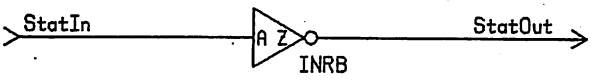
2-1

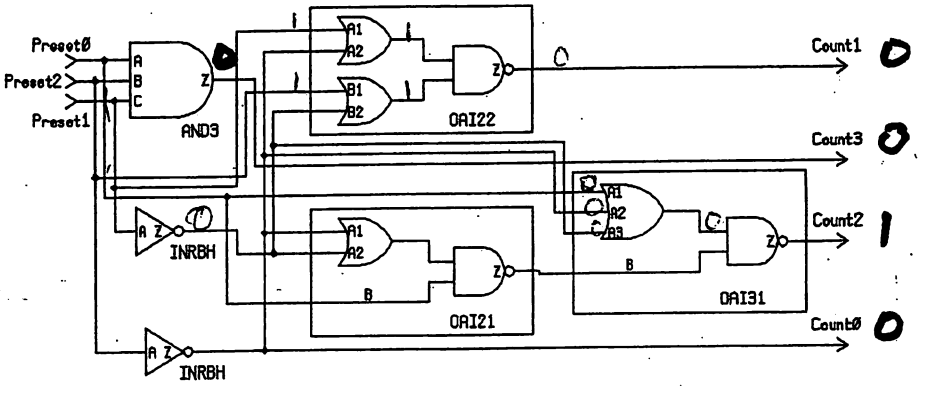
```
module Continuous (StatIn, StatOut);
  input StatIn;
  output StatOut;

  assign StatOut = ~ StatIn; // Continuous assignment.
endmodule
// Synthesized netlist is shown in Figure
```



2-2

```
module Blocking (Preset, Count);
  input [0:2] Preset;
  output [3:0] Count;
  reg [3:0] Count;

  always @ (Preset)
    Count = Preset + 1;
    // Blocking procedural assignment.
endmodule
// Synthesized netlist is shown in Figure
```

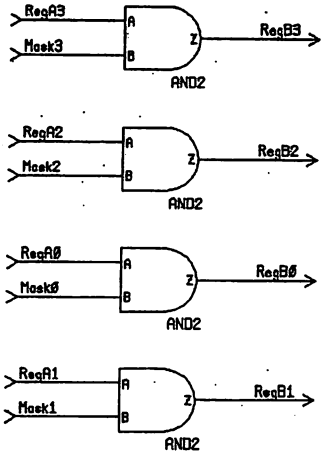
3_{10} 1 1



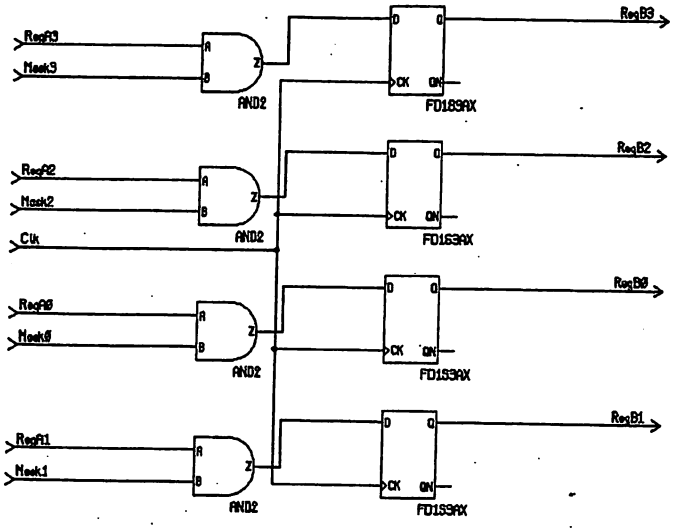
Count 3 = 0 ✓✓
Count 2 = 1 ✓✓
Count 1 = 0 ✓✓
Count 0 = 0 ✓✓

2-3

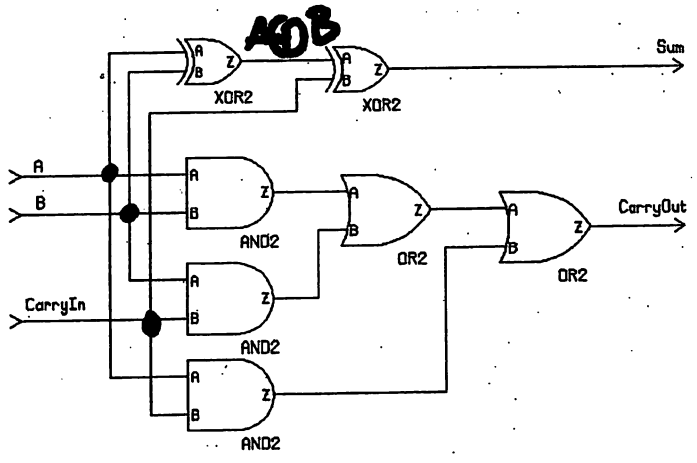
```
module NonBlocking (RegA, Mask, RegB);
  input [3:0] RegA, Mask;
  output [3:0] RegB;
  reg [3:0] RegB;

  always @ (RegA or Mask)
    RegB <= RegA & Mask;
    // Non-blocking procedural assignment.
endmodule
// Synthesized netlist is shown in Figure
```



```
module Target (Clk, RegA, RegB, Mask);
  input Clk;
  input [3:0] RegA, Mask;
  output [3:0] RegB;
  reg [3:0] RegB;

  always @ (posedge Clk)
    RegB <= RegA & Mask;
endmodule
// Synthesized netlist is shown in Figure
```

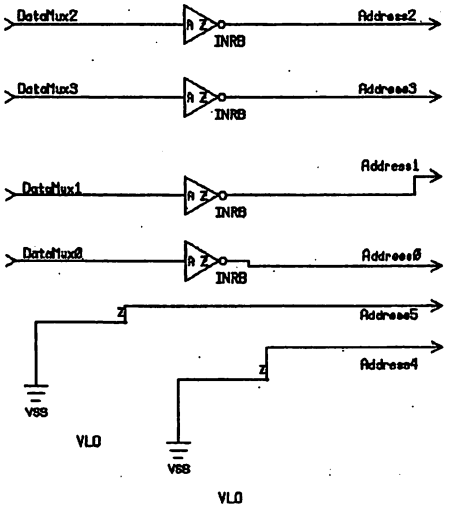


```
module FullAdder (A, B, CarryIn, Sum, CarryOut);
  input A, B, CarryIn;
  output Sum, CarryOut;

  assign Sum = (A ^ B) ^ CarryIn;
  assign CarryOut = (A & B) | (B & CarryIn) |
                    (A & CarryIn);
endmodule
// Synthesized netlist is shown in Figure
```

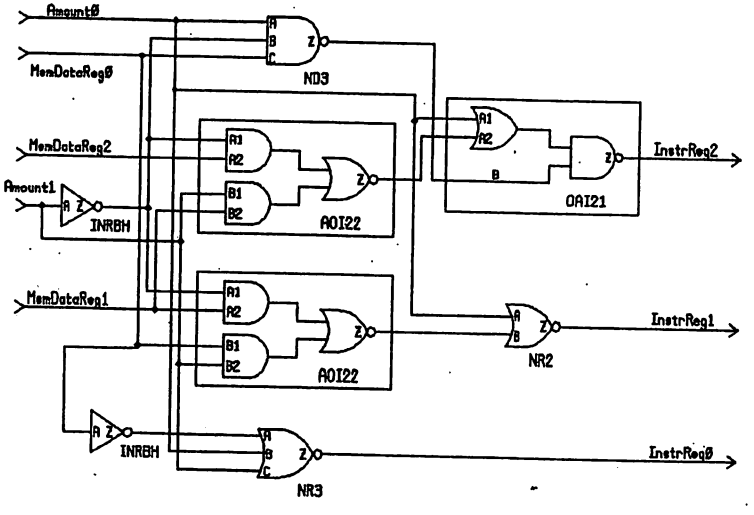


2 - 11

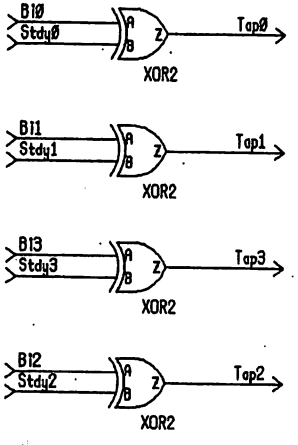
```
module ConstantShift (DataMux, Address);
  input [0:3] DataMux;
  output [0:5] Address;

  assign Address = (~ DataMux) << 2;
endmodule
// Synthesized netlist is shown in Figure
```



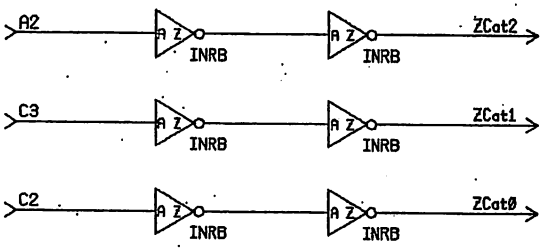
```
module VariableShift (MemDataReg, Amount, InstrReg);
  input [0:2] MemDataReg;
  input [0:1] Amount;
  output [0:2] InstrReg;

  assign InstrReg = MemDataReg >> Amount;
endmodule
// Synthesized netlist is shown in Figure
```



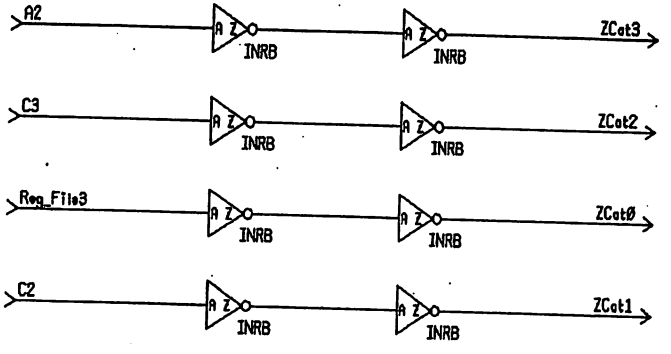
```
module VectorOperands (Bi, Stdy, Tap);
  input [0:3] Bi, Stdy;
  output [0:3] Tap;

  assign Tap = Bi ^ Stdy;
endmodule
// Synthesized netlist is shown in Figure
```



```
module PartSelect (A, C, ZCat);
  input [3:0] A, C;
  output [3:0] ZCat;

  assign ZCat[2:0] = {A[2], C[3:2]};
endmodule
// Synthesized netlist is shown in Figure
```



```
module ConstantIndex (A, C, Reg_File, ZCat);
  input [3:0] A, C;
  input [3:0] Reg_File;
  output [3:0] ZCat;

  assign ZCat[3:1] = {A[2], C[3:2]};
  assign ZCat[0] = Reg_File[3];
endmodule
// Synthesized netlist is shown in Figure
```



```
module NonComputeRight (Data, Index, Dout);
  input [0:3] Data;
  input [1:2] Index;
  output Dout;

  assign Dout = Data [Index];
endmodule
// Synthesized netlist is shown in Figure 2-17.
```

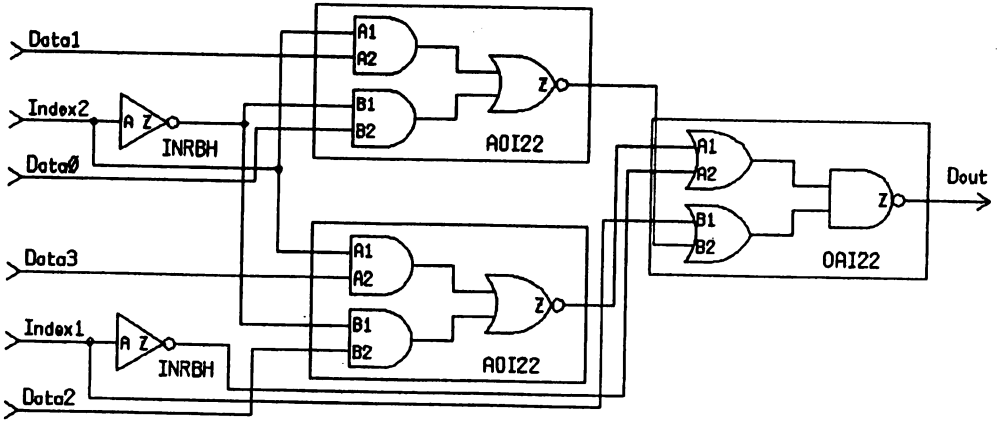
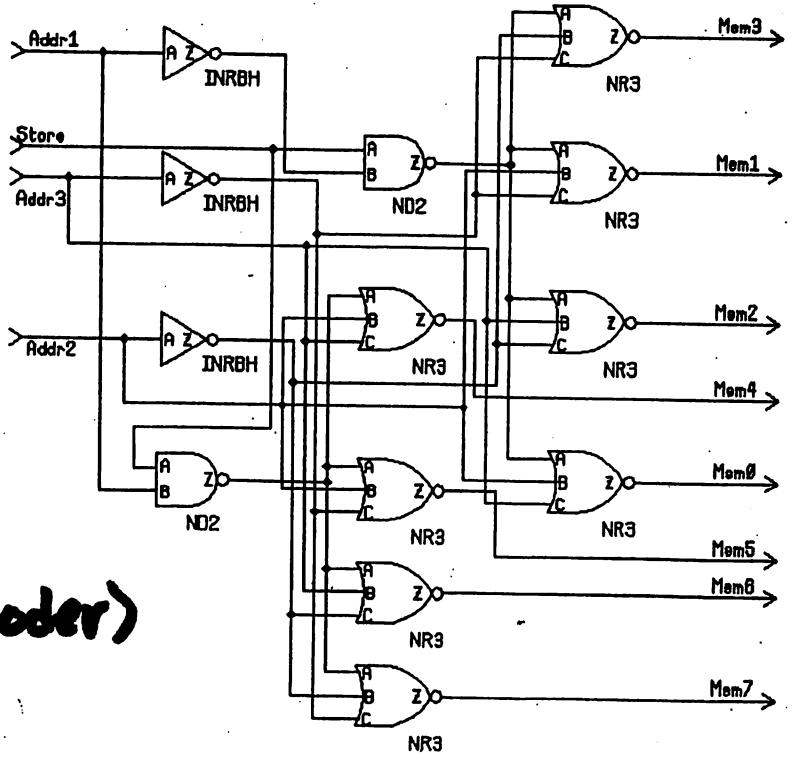


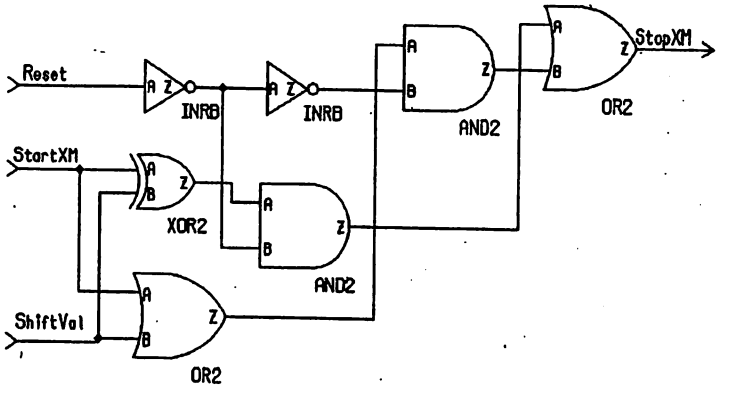
Figure 2-17 Non-constant bit-select generates a multiplexer.

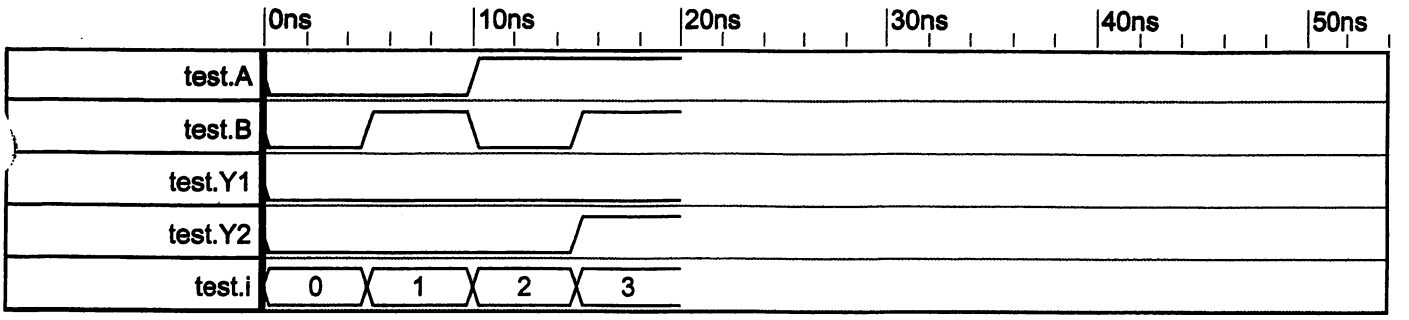
```
module NonComputeLeft (Mem, Store, Addr);
  output [7:0] Mem;
  input Store;
  input [1:3] Addr;

  assign Mem [Addr] = Store;
endmodule
// Synthesized netlist is shown in Figure .
```



(a decoder)

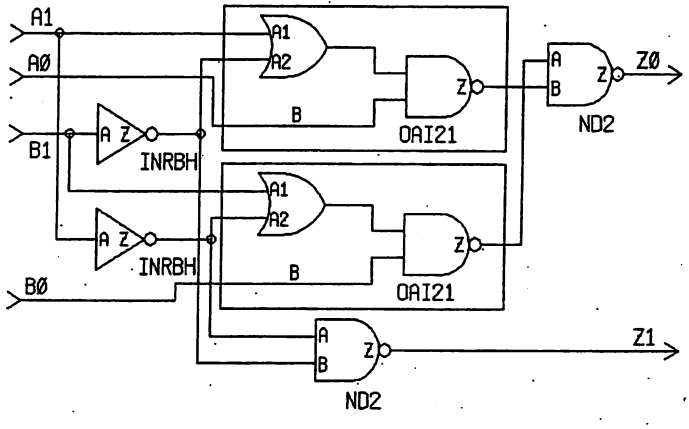
```
module ConditionalExpression (StartXM, ShiftVal,
                              Reset, StopXM);
  input StartXM, ShiftVal, Reset;
  output StopXM;

  assign StopXM = (! Reset) ? StartXM ^ ShiftVal :
                              StartXM | ShiftVal;
endmodule
// Synthesized netlist is shown in Figure .
```



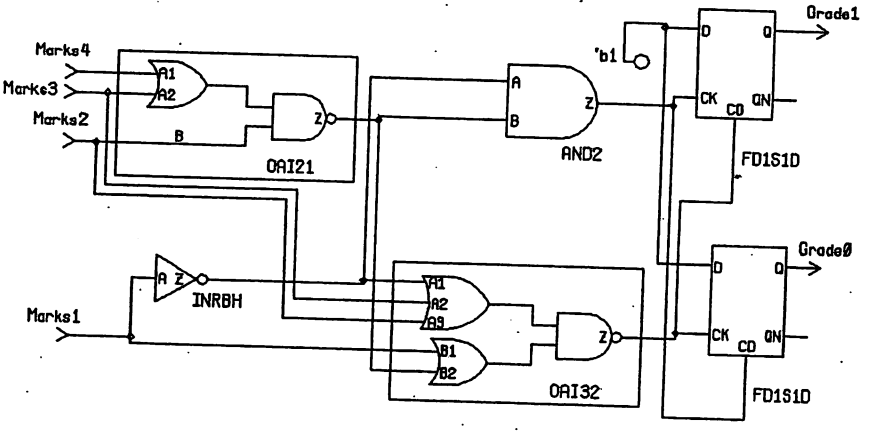



```
module my_and1(a,b,y);
input a,b;
output y;
reg y;

always@(a)
        if(a & b)
        y = 1;
  else
    y = 0;

endmodule


module my_and2(a,b,y);
input a,b;
output y;
reg y;

always@(a or b)
  if(a & b)
        y = 1;
  else
    y = 0;

endmodule


module test;
reg A,B;
wire Y1,Y2;

integer i;

my_and1 g2(A,B,Y1);
my_and2 g3(A,B,Y2);

initial
 begin
  for(i=0;i<=3;i=i+1)
    begin
          {A,B}=i;
       #5;
    end
 end


 initial #20 $finish;

endmodule
```

```
module SelectOneOf (A, B, Z);
  input [1:0] A, B;
  output [1:0] Z;
  reg [1:0] Z;

  always @ (A or B)
    if (A > B)
      Z = A;
    else
      Z = B;
endmodule
```

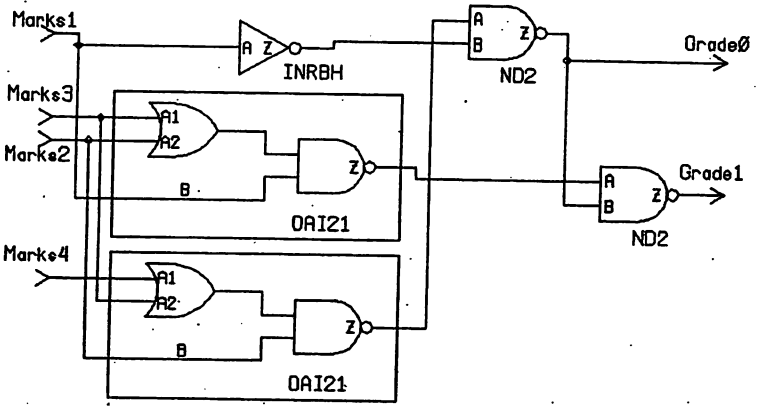


```
module Compute (Marks, Grade);
  input [1:4] Marks;
  output [0:1] Grade;
  reg [0:1] Grade;

parameter FAIL = 1, PASS = 2, EXCELLENT = 3;

always @ (Marks)
  if (Marks < 5)
    Grade = FAIL;
  else if ((Marks >= 5) & (Marks < 10))
    Grade = PASS;
endmodule
```



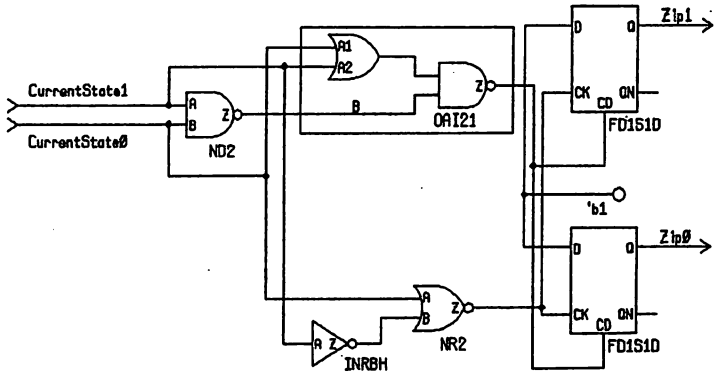
2-27

```
module ComputeNoLatch (Marks, Grade);
  input [1:4] Marks;
  output [0:1] Grade;
  reg [0:1] Grade;
  parameter FAIL = 1, PASS = 2, EXCELLENT = 3;

  always @ (Marks)
    if (Marks < 5)
      Grade = FAIL;
    else if ((Marks >= 5) && (Marks < 10))
      Grade = PASS;
    else
      Grade = EXCELLENT;
endmodule
```



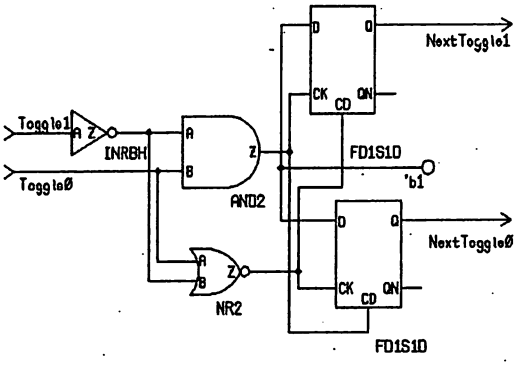
2-27

```
module StateUpdate (CurrentState, Zip);
  input [0:1] CurrentState;
  output [0:1] Zip;
  reg [0:1] Zip;

  parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;

  always @ (CurrentState)
    case (CurrentState)
      S0,
      S3: Zip = 0;
      S1: Zip = 3;
    endcase
endmodule
```

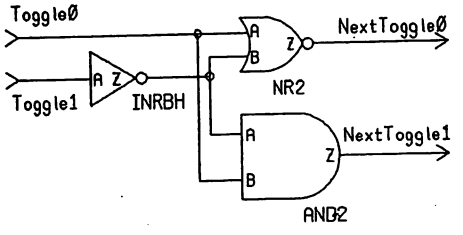



```
. . .
always @ (CurrentState)
begin
  Zip = 0;                    // This statement added.

  case (CurrentState)
  . . .
  endcase
end
```

```
module NextStateLogic (NextToggle, Toggle);
  input [1:0] Toggle;
  output [1:0] NextToggle;
  reg [1:0] NextToggle;

  always @ (Toggle)
    case (Toggle)
      2'b01 : NextToggle = 2'b10;
      2'b10 : NextToggle = 2'b01;
    endcase

endmodule
```

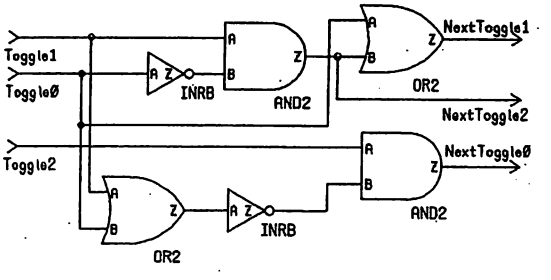


```
module NextStateLogicFullCase (NextToggle, Toggle);
  input [1:0] Toggle;
  output [1:0] NextToggle;
  reg [1:0] NextToggle;

    always @ (Toggle)
      case (Toggle)                    // synthesis full_case
        2'b01 : NextToggle = 2'b10;
        2'b10 : NextToggle = 2'b01;
      endcase
  endmodule
```



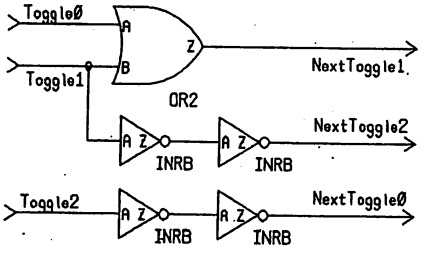
```
module PriorityLogic (NextToggle, Toggle);
  input [2:0] Toggle;
  output [2:0] NextToggle;
  reg [2:0] NextToggle;

  always @ (Toggle)
    casex (Toggle)
      3'bxx1 : NextToggle = 3'b010;
      3'bx1x : NextToggle = 3'b110;
      3'b1xx : NextToggle = 3'b001;
      default : NextToggle = 3'b000;
    endcase
endmodule
```

```
  if (Toggle[0] == 'b1)
    NextToggle = 3'b010;
  else if (Toggle[1] == 'b1)
    NextToggle = 3'b110;
  else if (Toggle[2] == 'b1)
    NextToggle = 3'b001;
  else
    NextToggle = 3'b000;
```

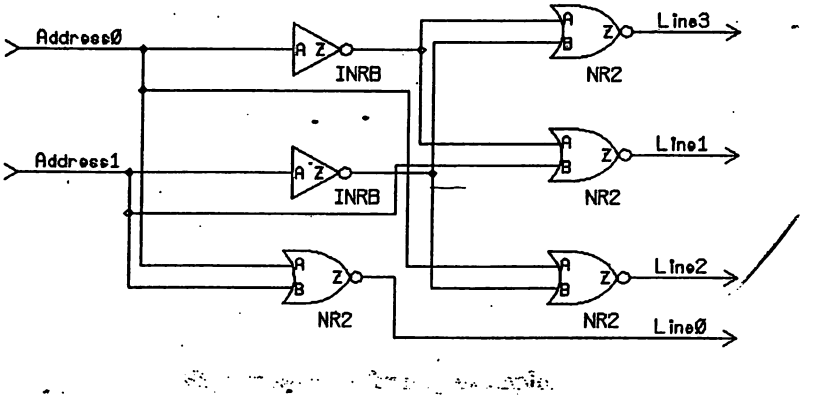


```
module ParallelCase (NextToggle, Toggle);
  input [2:0] Toggle;
  output [2:0] NextToggle;
  reg [2:0] NextToggle;

  always @ (Toggle)
    casex (Toggle)          // synthesis parallel_case
      3'bxx1 : NextToggle = 3'b010;
      3'bx1x : NextToggle = 3'b110;
      3'b1xx : NextToggle = 3'b001;
      default : NextToggle = 3'b000;
    endcase
endmodule
```

```
if (Toggle[0] == 'b1)
  NextToggle = 3'b010;

if (Toggle[1] == 'b1)
  NextToggle = 3'b110;

if (Toggle[2] == 'b1)
  NextToggle = 3'b001;

if ((Toggle[0] != 'b1) &&
    (Toggle[1] != 'b1) &&
    (Toggle[2] != 'b1))
  NextToggle = 3'b000;
```

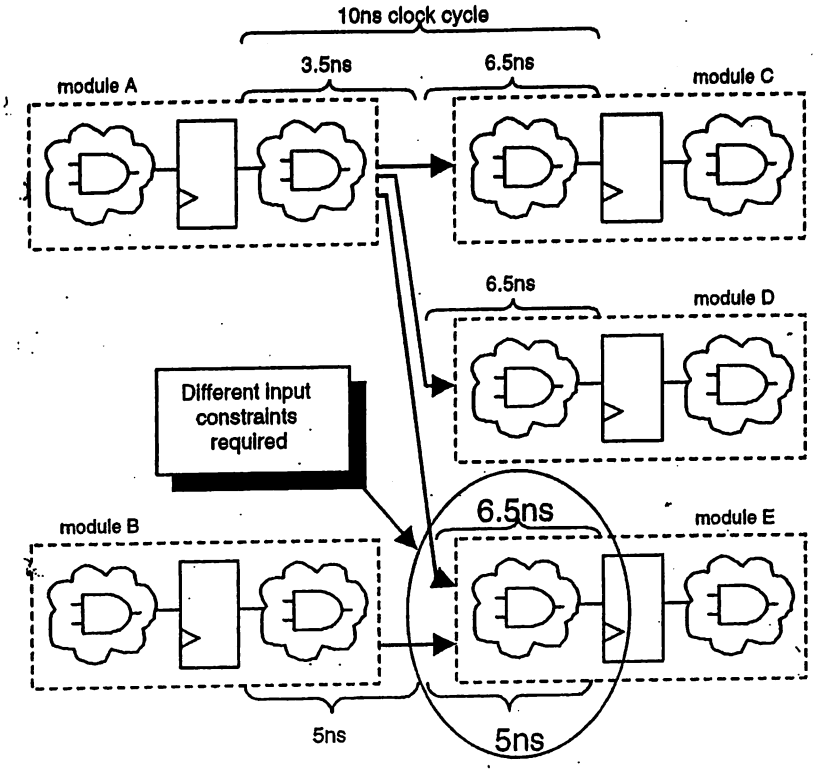


```
  output [3:0] Line;
  reg [3:0] Line;

  integer J;

  always @ (Address)
    for (J = 3; J >= 0; J = J - 1)
      if (Address == J)
        Line[J] = 1;
      else
        Line[J] = 0;
endmodule
```



When the for-loop is expanded, the following four `if` statements are obtained.

```
if (Address == 3) Line[3] = 1; else Line[3] = 0;
if (Address == 2) Line[2] = 1; else Line[2] = 0;
if (Address == 1) Line[1] = 1; else Line[1] = 0;
if (Address == 0) Line[0] = 1; else Line[0] = 0;
```



Constraining combinational outputs that drive combinational inputs

Assume CLK period = 10 ns

Assume $t_{su} = t_h = 0$