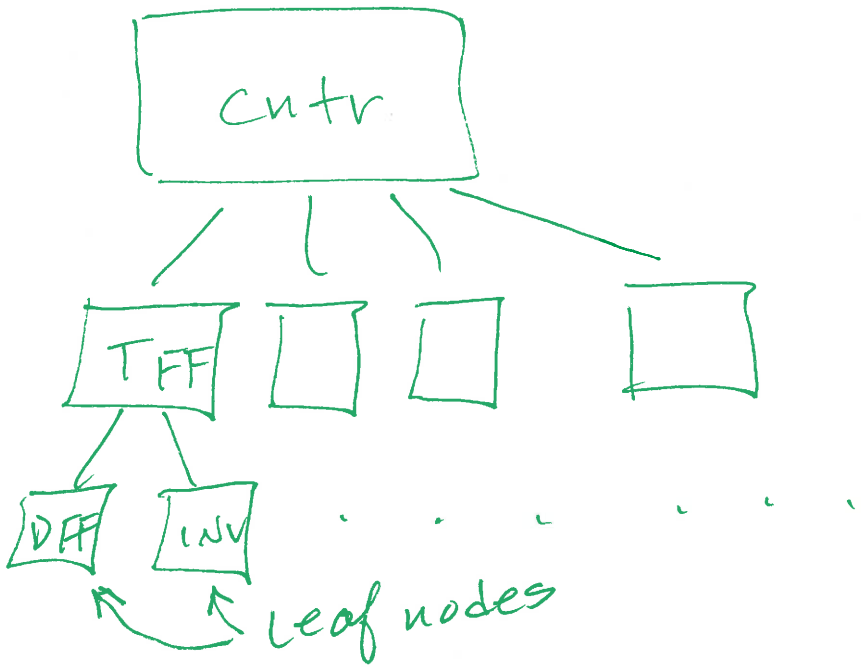
Defn (partitioning)

the process of dividing a design into modules thereby creating a

design hierarchy



Can partition

- horizontally
- vertically

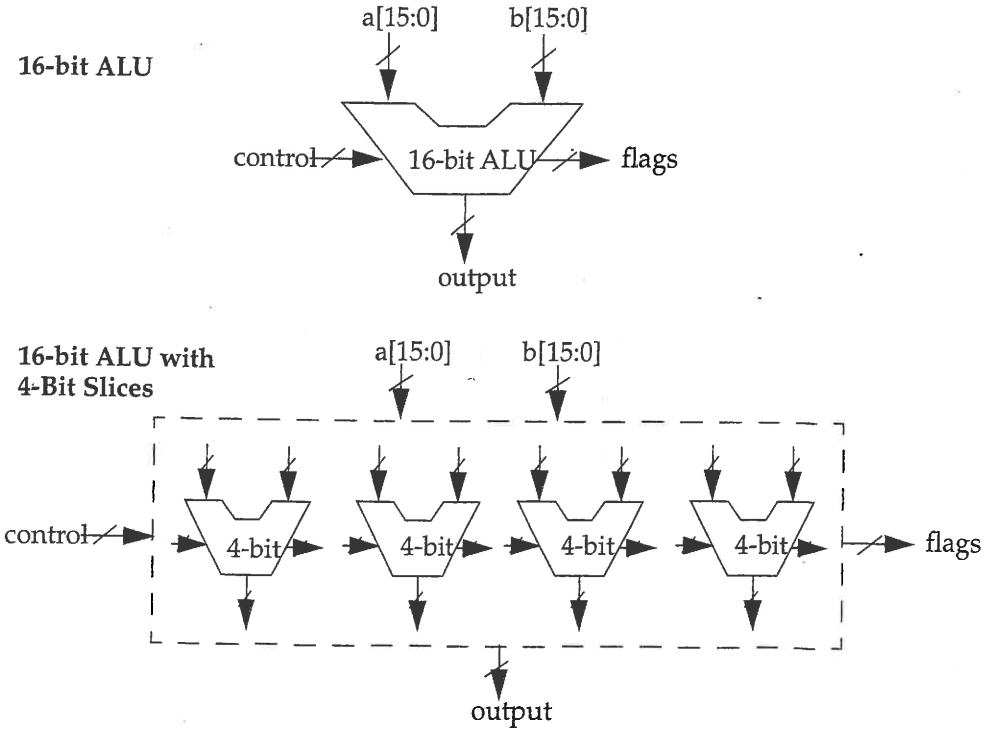


Figure 14-7 Horizontal Partitioning of 16-bit ALU

easier to optimize small bit circuitry

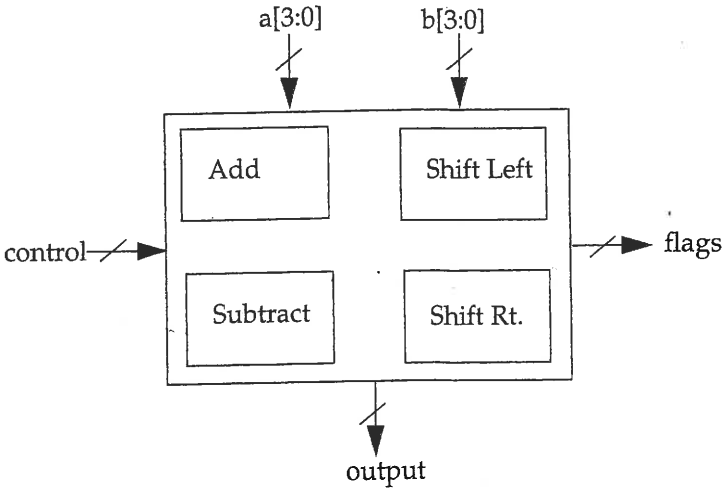


Figure 14-8 Vertical Partitioning of 4-bit ALU

easier to optimize individual functions

General partitioning rules

- only leaf nodes in the hierarchy should be gate level of abstraction

  makes the synthesis faster

- critical paths should be in one module entirely

defn (critical path)

slowest combination logic path between registers

makes timing analysis easier and makes it easier to optimize the timing

- whenever possible, register the module output

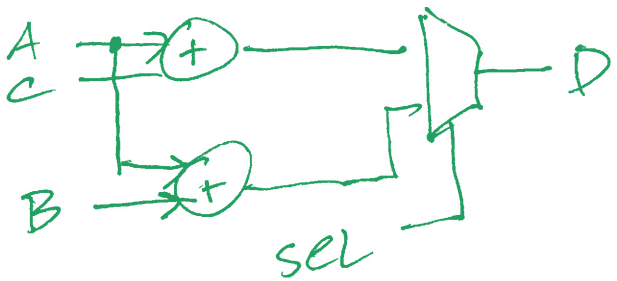
  makes it easier to determine timing between modules

- potentially sharable resources should be in the same module

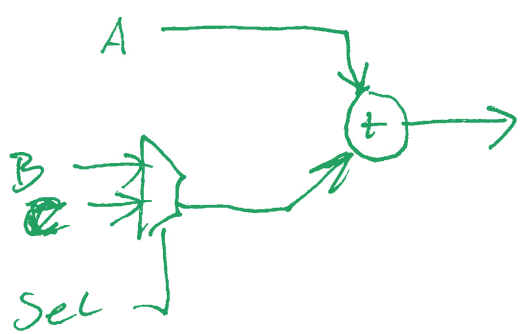
e.g.

```
always @( A or B or C or Sel )
    D = Sel ? A+C : A+B;
```

w/o sharing



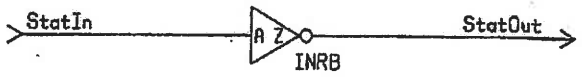
w/ sharing



- Keep modules as small as possible
  (consistent with above guidelines)

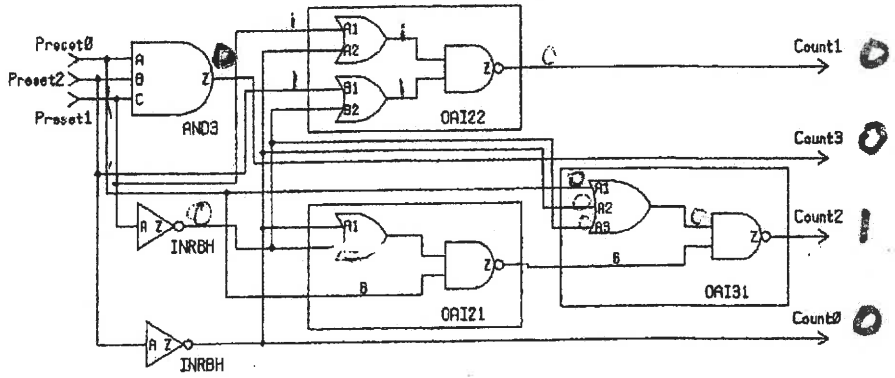
2-1

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

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



2-2

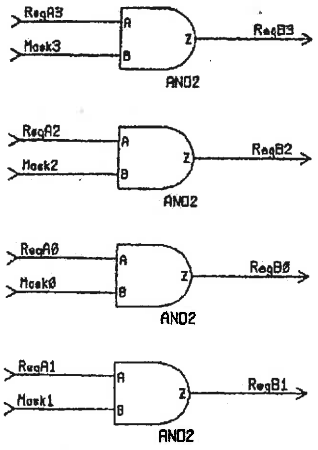
```
module Blocking (Preset, Count);
  input [0:2] Preset;
  output [3:0] Count;
  reg [3:0] Count;

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



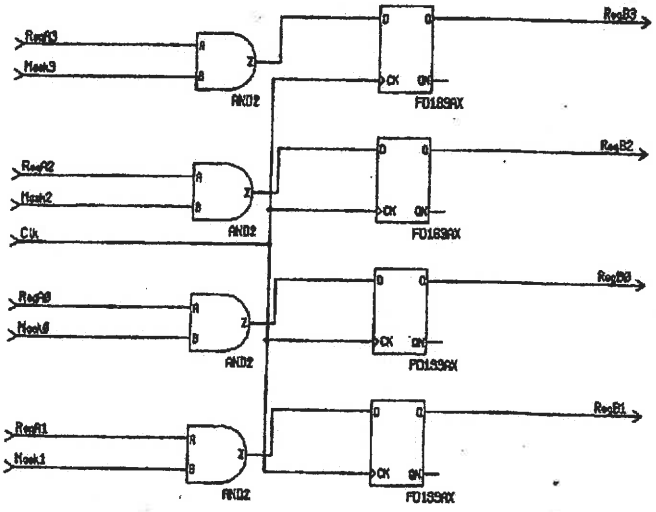
Count 3 = 0
Count 2 = 1
Count 1 = 0
Count 0 = 0

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
```



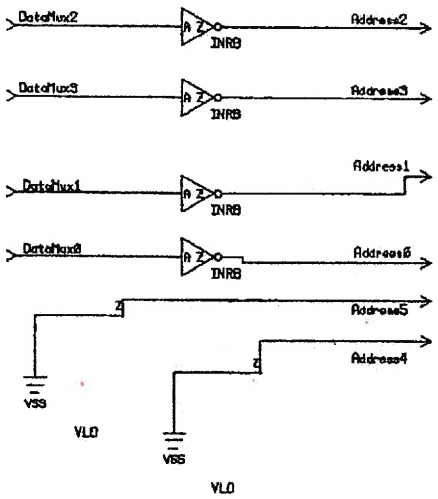
```
module Target (Clk, RegA, RegB, Mask);
  input Clk;
  input [3:0] RegA, Mask;
  output [3:0] RegB;
  reg [3:0] RegB;

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



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
```

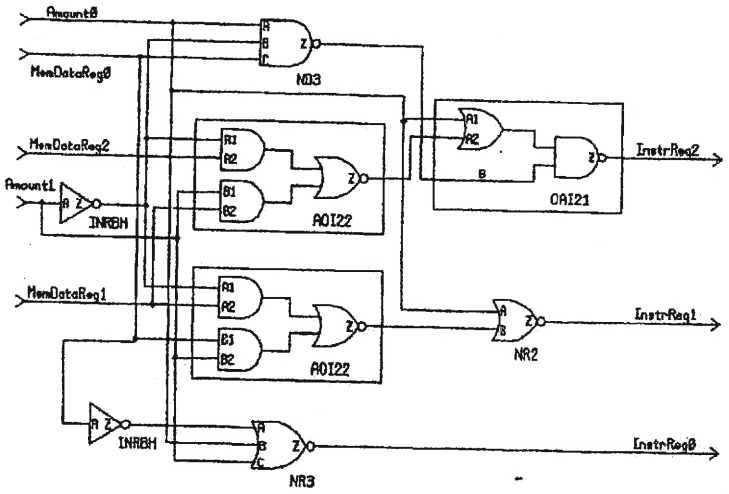
a constant



2-12

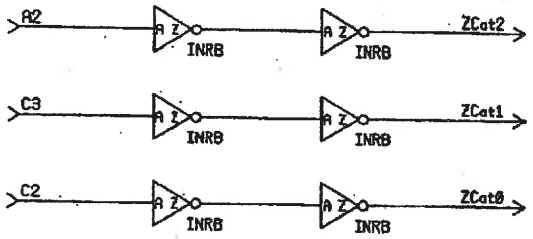
```
module VariableShift (MemDataReg, Amount, InstrReg);
  input [0:2] MemDataReg;
  input [0:1] Amount;
  output [0:2] InstrReg;

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

a variable



2-12

```
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 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.
```

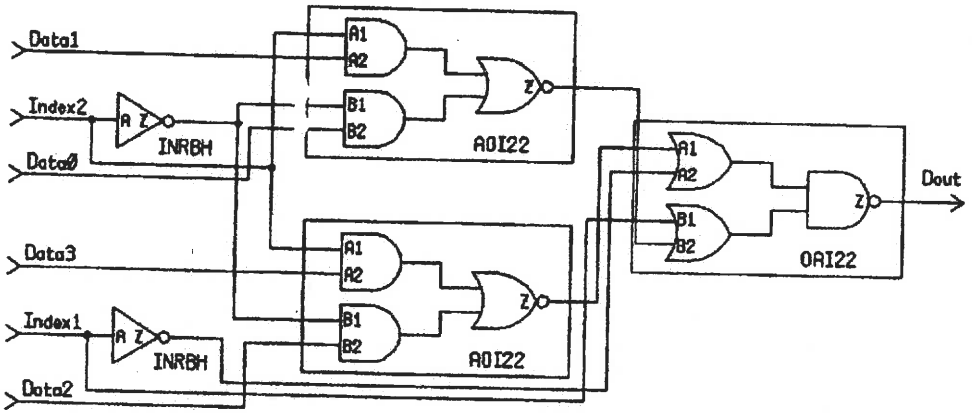
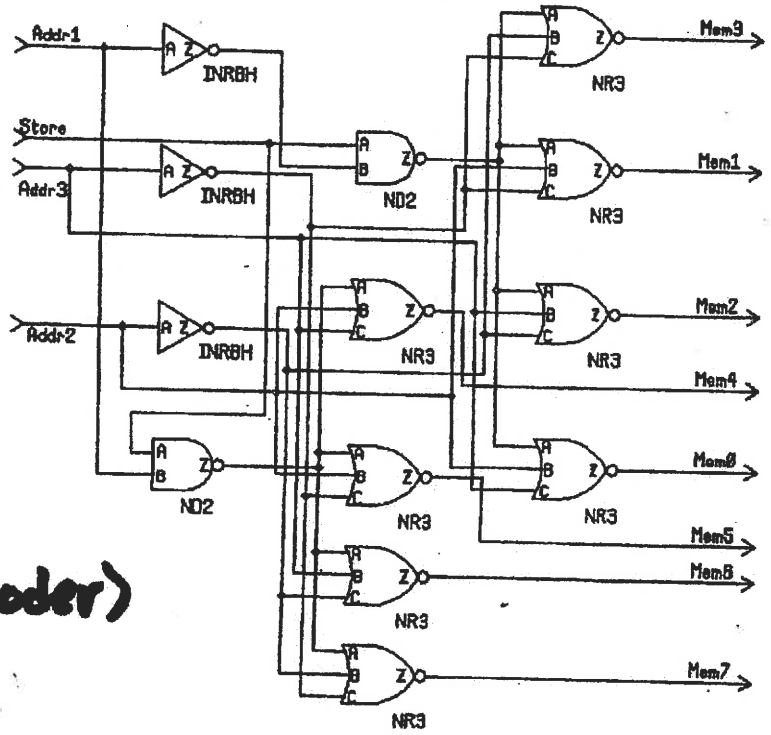


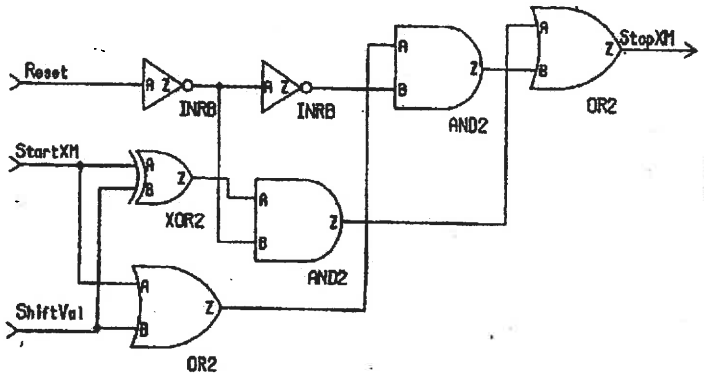
Figure 2-17 Non-constant bit-select generates a multiplexer. (if on RHS)

```
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 .
```

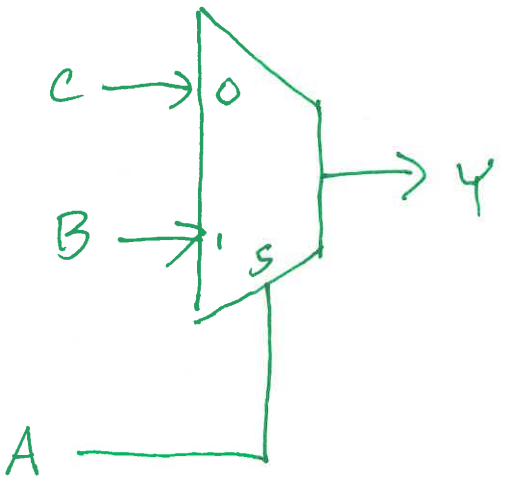


⇒ a mux

if-then-else code synthesizes as a multiplexer

e.g.

```
if (A)
    Y = B;
else
    Y = C;
```



Latch inferrance

a really, really bad thing (really)

occurs when

- an if-then-else doesn't contain a default else

or

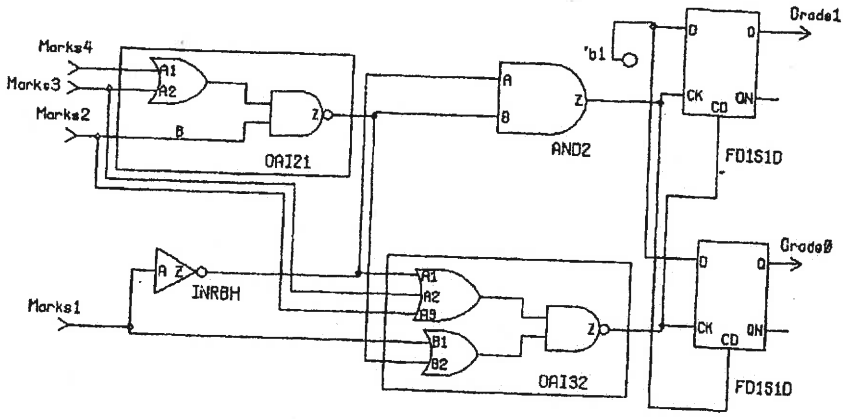
- case statement where not all case values listed

and

no default

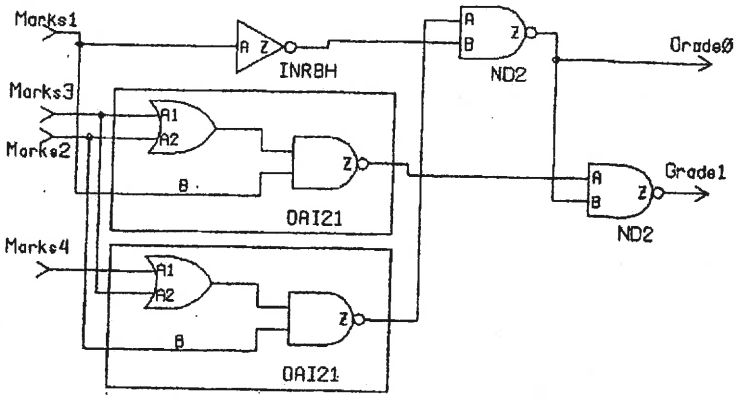
2-26

```
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
```



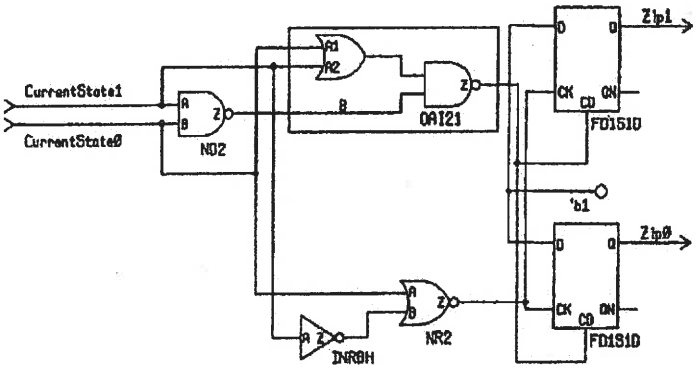
2-27

```
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
```

(a preassignment)

Consider the following...

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

always @(abc)
   case (abc)                    // synthesis full_case
   S0,S1 : zip=0;
      S3 : zip=2;
   endcase
```

Known as a synthesis directive

Prevent latch inferrance w/o

- preassignment
- default in the case statement

when synthesis full-case makes more sense ...

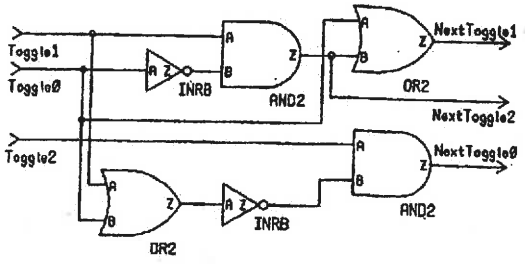
```
parameter S0 = 3'b001, S1 = 3'b010, S2 = 3'b100;
```

a one-hot encoded FSM

```
case (state) // synthesis full-case
    S0: ____
    S1: ____
    S2: ____
endcase
```

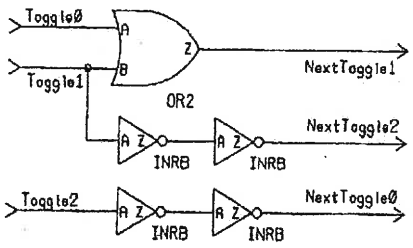
```
module PriorityLogic (NextToggle, Toggle);
  input [2:0] Toggle;
  output [2:0] NextToggle;
  reg [2:0] NextToggle;

  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;
```



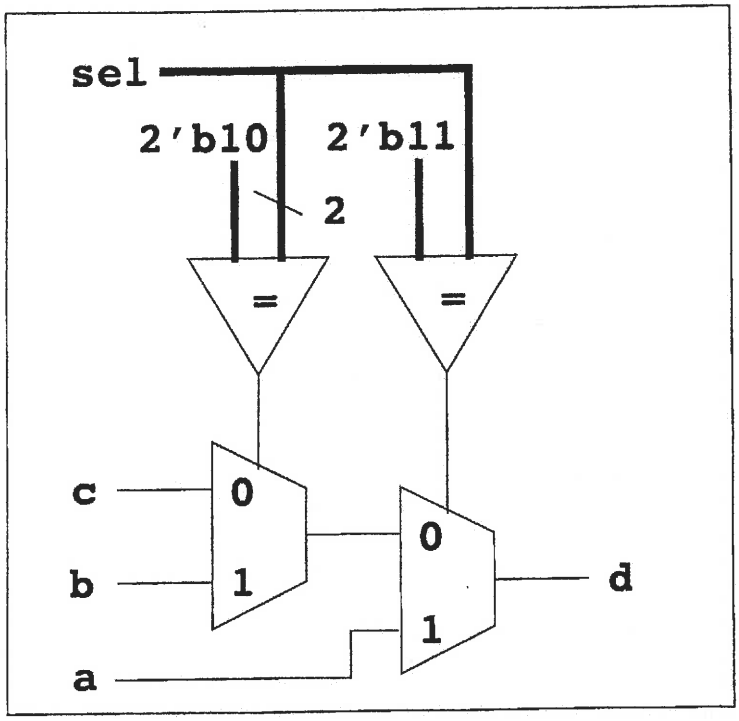
```
module ParallelCase (NextToggle, Toggle);
  input [2:0] Toggle;
  output [2:0] NextToggle;
  reg [2:0] NextToggle;

  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;
```



2.3.2 Priority Decoder using an if/else statement

```
// 2. using an if statement
always @ (sl or a or b or c)
   if (sel == 2'b11)
      d = a;
   else if (sel ==2'b10)
      d = b;
   else
      d = c;
```



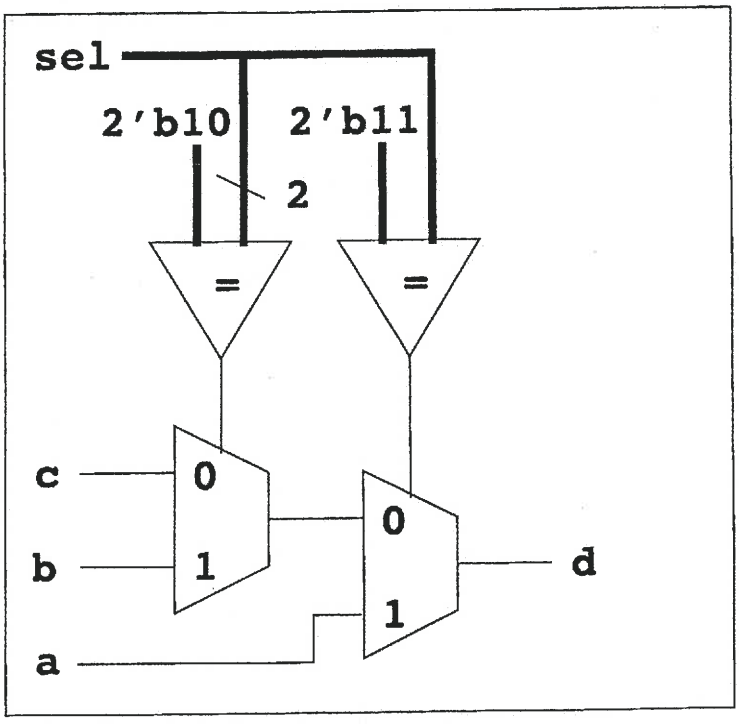
2.3 Priority Decoders

2.3.1 Priority Decoder using a case statement

```
// 1. using a case statement
always @ (sl or a or b or c)
   case (sel)
      2'b11:  d = a;
      2'b10:  d = b;
      default: d = c;
   endcase
```

1. Both case and if statements result in priority structures.

2. The order of the variables determines the priority

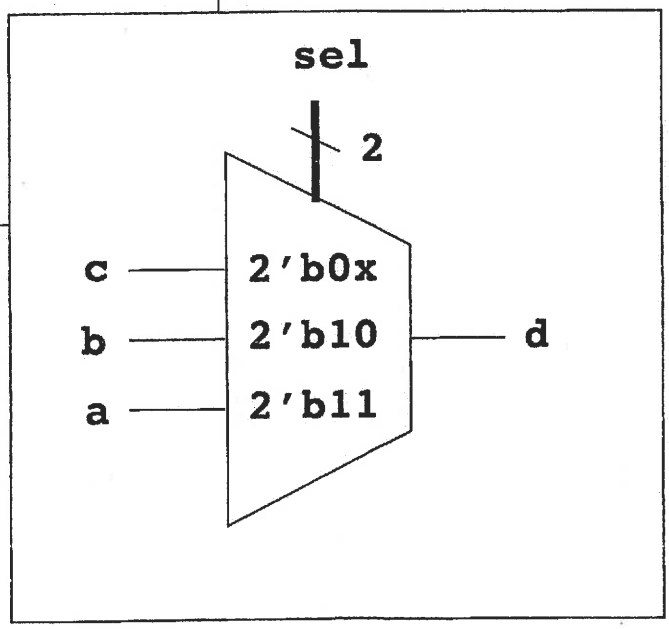


2.4 Parallel Priority Decoders

2.4.1 Parallel Priority Decoders Using a Synthesis Directive

```
// using a synthesis directive
always @ (sl or a or b or c)
   case (sel) // parallel_case
      2'b11: d = a;
      2'b10: d = b;
      default:d = c;
   endcase
```

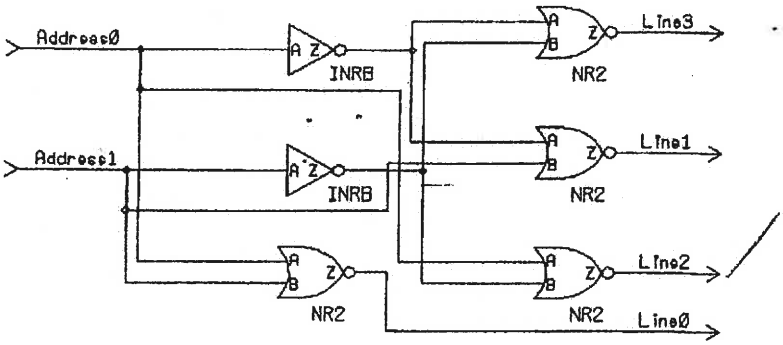
Note: full_case not used



The parallel_case synthesis directive prevents the cascaded priority logic you get with the if-then-else-if-then....

2-44

```
    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;
```

~~Rule~~

Law:

Blocking assignments ⟶ use for combinational logic

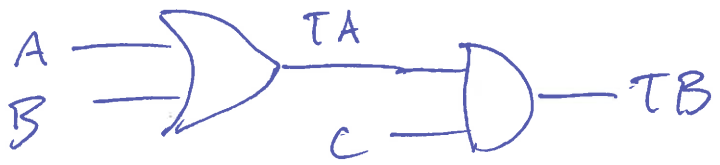
non-blocking assignments ⟶ use for sequential logic

Consider

```
reg TA, TB;
    always @ (A or B or C)
      begin
          TA = A | B;
          TB = TA & C;
      end
```

Blocking assignments

Synthesizes as



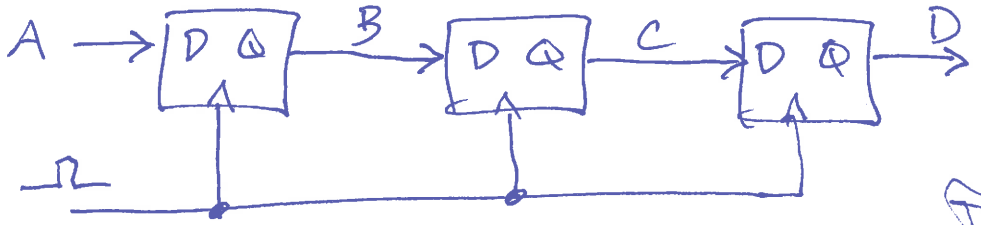
```
reg TA, TB;

always@(A or B or C)
    begin
        TA <= A | B;
        TB <= TA&C;
    end
```

uses "old" TA value

⇓

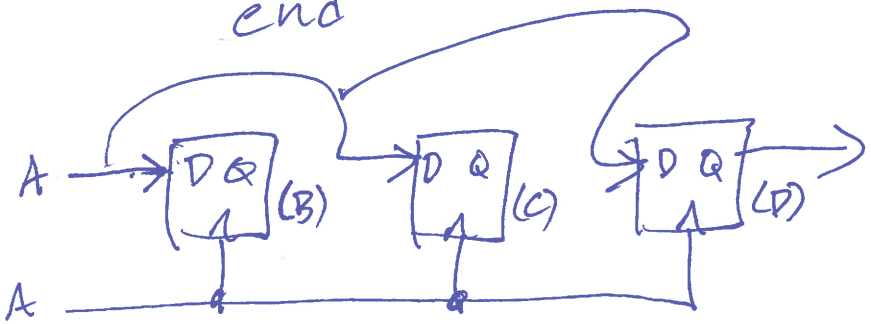
need some sort of storage...

I want



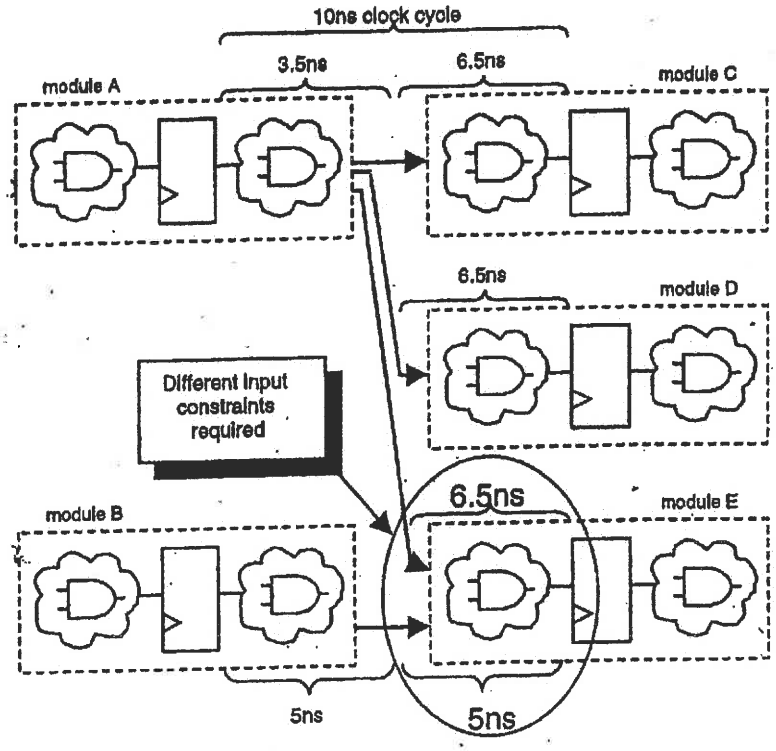
w/ Blocking

```
always@ (posedge clk)
  begin
    B=A;
    C=B;
    D=C;
  end
```



w/ non-blocking

```
always@ (posedge clk)
  begin
    B <= A;
    C <= B;
    D <= C;
  end
```



· Constraining combinational outputs that drive combinational inputs

Assume CLK period = 10 ns

Assume $t_{su} = t_h = 0$