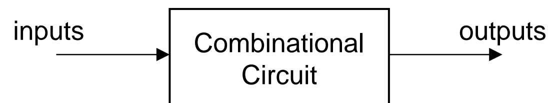
Modeling Combinational Circuits with Verilog

Prof. Chien-Nan Liu
TEL: 03-4227151 ext:34534
Email: jimmy@ee.ncu.edu.tw

3-1

Combinational Circuit Design

- Outputs are functions of inputs



- Description styles
  - Gate-level
  - Data-flow
  - Behavior
- Some guidelines are given for synthesis

3-2

Sensitivity List

- The sensitivity list must include all inputs of the block
  - All variables in condition statements
  - All variables on the right hand side of procedural assignments
- If not all inputs are listed
  - The changes of inputs may not change outputs immediately
  - May cause functional mismatch in the synthesized circuits
    - The sensitivity list will be skipped during synthesis

Incomplete sensitivity list

```
always@(s1) begin
  if (!s1) q=a;
  else q=b;
end
```

Complete sensitivity list

```
always@(s1 or a or b) begin
  if (!s1) q=a;
  else q=b;
end
```

3-3

Non-Synthesizable Verilog Constructs

Not commonly supported by synthesis tools !!

- initial
- Loops
  - repeat
  - forever
  - while
- Data types
  - event
  - real
  - time
- UDPs
- fork …join blocks
- procedural assignments
  - assign and deassign
  - force and release
  - disable
- Some operators
  - / and %
  - === and !==

3-4

Inconsistent Results

- The following situations may cause simulation to disagree with synthesis
  - Incomplete sensitivity list
    - Sensitivity list is ignored in synthesis routines
  - Code with delays
    - The specified delay values are also ignored
  - Non-local reference within a function
  - Order dependency of concurrent statements
  - Comparisons to X or Z
- Those cases should be avoided at all

3-5

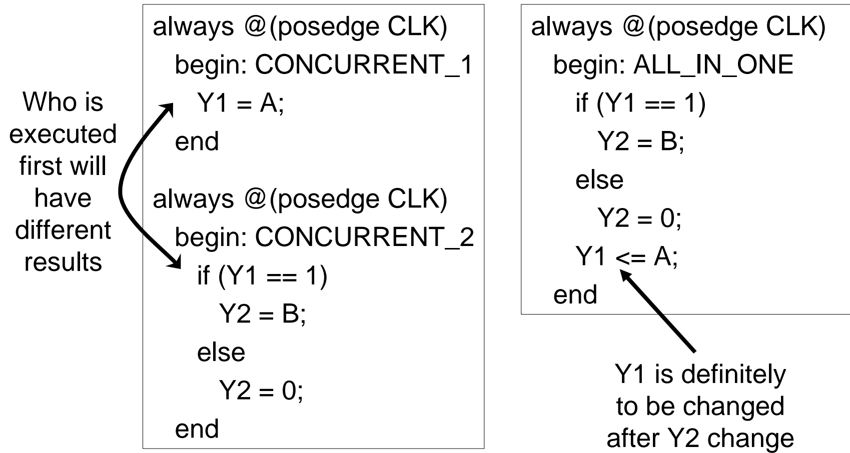
Non-Local Reference

- Not all inputs are declared in the input list (sensitivity list) of a function

```
function byte_compare;
  input [15:0] vector1, vector2;
  input [7:0] length;
                                   non-local variable
  begin
    if (byte_sel)
      // compare  the upper byte
    else
      // compare the lower byte
  end
endmodule
```

3-6

Order Dependency



3-7

Comparisons to X or Z

```
always @(A)
  begin
    if (A === 1’bx)
      B = 0;
    else
      B = 1;
  end
```

Comparisons to a “don’t care” are treated as always being false in synthesis routines

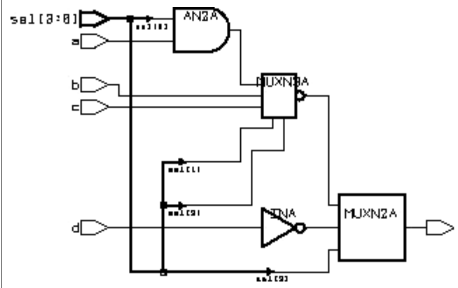
3-8

Priority for If Statements

The last "if" has highest priority

```
always@(a or b or c or d or sel)
begin
  z=0;
  if (sel[0]) z=a;
  if(sel[1]) z=b;
  if(sel[2]) z=c;
  if (sel[3]) z=d;
end
```

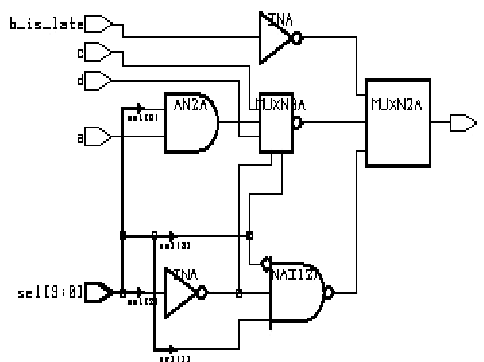
highest priority



3-9

Late Arriving Data Signal

```
always@(a or b_is_late or c or
d or sel)
begin
  z=0;
  if (sel[0]) z1=a;
  if(sel[2]) z1=c;
  if (sel[3]) z1=d;
  if ( sel[1] & ~sel[2] | sel [3])
    z=b_is_late;
  else
    z=z1;
end
```



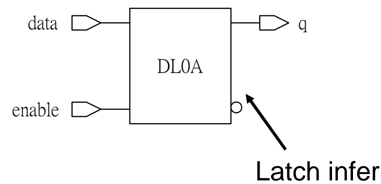
Put the last arriving signal to the last if condition

3-10

Avoid Latch Inference (1/3)

- When if or case statements are used without specifying outputs in all possible condition, a latch will be created

```
always@(enable or data)
  if (enable)
  q =data ;
```



Latch infer

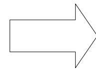
```
always@(enable or data)
begin
  if (enable)
    q =data ;
  else
    q=0;
end
```

Specify all output value

Avoid Latch Inference (2/3)

```
always@(a or b or c)
 case(a)
    2'b11 : e = b;
    2'b10 : e = ~c;
 end case
```

No latch



```
always@(a or b or c)
 case(a)
    2'b11 : e = b;
    2'b10 : e = ~c;
    default : e = 0 ;
 end case
```

Add default statement

result in a latch

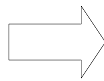
```
always@(a or b or c)
 e =0;
 case(a)
    2'b11 : e = b;
    2'b10 : e = ~c;
 end case
```

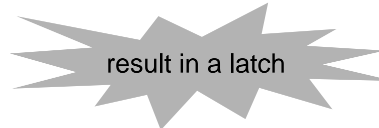
Give initial value

Avoid Latch Inference (3/3)

Not all possible cases are defined

```
always@(a or b or c)
 case(a)
    2'b11 : e = b;
    2'b10 : e = ~c;
 end case
```





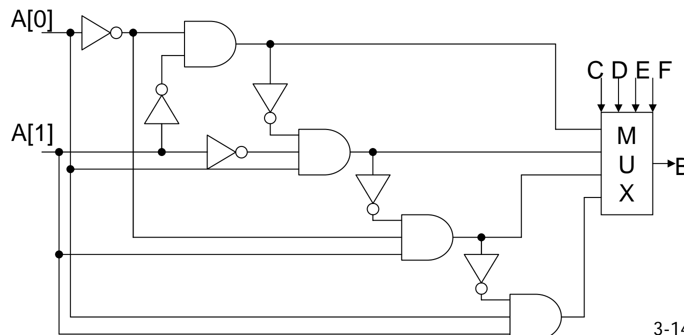

```
always@(a or b or c)
case(a)  // synopsys full_case
    2'b11 : e = b;
    2'b10 : e = ~c;
 end case
```

- To avoid latch inference and the need for default logic, add case directive :
  - //synopsys full_case
  - // ambit synthesis case = full

3-13

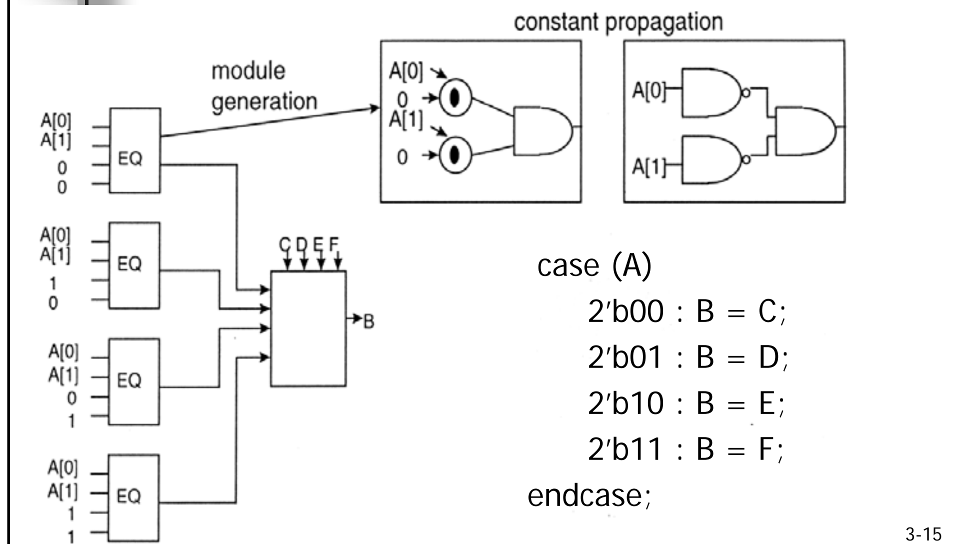
If vs. Case (1/2)

```
if (A(0) == 0 and A(1) == 0) then
    B = C;
else if (A(0) == 1 and A(1) == 0) then
    B = D;
else if (A(0) == 0 and A(1) == 1) then
    B = E;
else
    B = F;
end if;
```



3-14

If vs. Case (2/2)



3-15

Case Directive

- A non-parallel (overlapped) case statement will generate the logic for a priority encoder
  - The first case item has the highest priority
- If only one case item is executed at a time
  - Force to generate multiplexer logic instead
    - Use // synopsys parallel_case
    - Use //ambit synthesis case = parallel
- The results are unexpected when more than one case items are executed together

3-16

Inferring Multiplexers

- Use directives to force inferring multiplexers

```
//synopsys infer_mux "mux_lab"
always@(SEL or DIN)
begin: mux_lab
case(SEL)
  2'b00:DOUT=DIN[0];
  2'b01:DOUT=DIN[1];
  2'b10:DOUT=DIN[2];
  2'b11:DOUT=DIN[3];
endcase
```

```
always@(SEL or DIN)
begin: mux_lab
case(SEL) //synopsys infer_mux
  2'b00:DOUT=DIN[0];
  2'b01:DOUT=DIN[1];
  2'b10:DOUT=DIN[2];
  2'b11:DOUT=DIN[3];
endcase
```

3-17

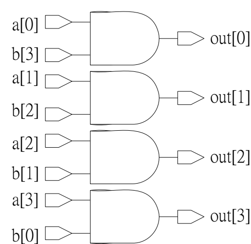
Loop Synthesis

- In synthesis, for loops are "unrolled", and then synthesized

```
integer i;
always@(a or b) begin
  for ( i=0; i<=3; i=i+1)
    out [i]=a[i]&b[3-i];
end
```

unroll

```
integer i;
always@(a or b) begin
  out[0]=a[0]&b[3];
  out[1]=a[1]&b[2];
  out[2]=a[2]&b[1];
  out[3]=a[3]&b[0];
end
```



3-18

Non-Static Loops

- Non-static loops are not synthesizable

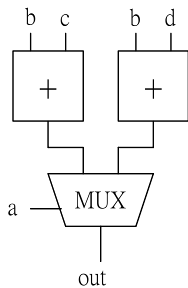
```
module NonStaticLoop (A, B, R, Y);
  input [7:0] A, B;
  input [2:0] R;
  output [7:0] Y;
  reg [7:0] Y;
  integer i;
  always @(A) begin
    Y = 8'b0;
    for ( i=0; i<R; i=i+1)
      Y[i] = A[i] & B[i];
  end
endmodule
```

R is non-static

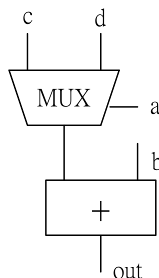
Resource Sharing (1/2)

```
always@(a or b or c or d)
  out =(a) ? (b+c):(b+d);
```

```
always@(a or b or c or d)
  if (a)  out = b+c;
  else out = b+d ;
```



without resource sharing



with resource sharing

Keep sharable resource in the
- ✓ same conditional statement
- ✓ same always block
- ✓ same module

Resource Sharing (2/2)

- The operators that can share resources must be in the mutual exclusive paths

```
if (sel1)  out1=a1+b1;
else begin
  out1 = c1+d1 ;
  if ( sel2) out2=a2+b2;
  else out2=c2+d2;
end
```

```
if (sel1)  out1=a1+b1;
else begin
  out1 = c1+d1 ;   (crossed out)
  if ( sel2) out2=a2+b2;
  else out2=c2+d2;
end
```

operator "c1+d1" and "a2+b2" or "c2+d2" are not in the mutual exclusive paths

All operators are in the mutual exclusive paths

Resource sharing

3-21

Modeling Examples with Verilog for Combinational Circuits

3-22

Code Converter (1/3)

- A code converter transforms one representation of data to another
- Ex: A BCD to excess-3 code converter
  - BCD: Binary Coded Decimal
  - Excess-3 code: the decimal digit plus 3

TABLE 4-2
Truth Table for Code Converter Example

Input BCD				Output Excess-3			
A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

Code Converter (2/3)

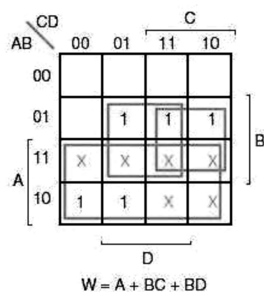
- Equations: (share terms to minimize cost)

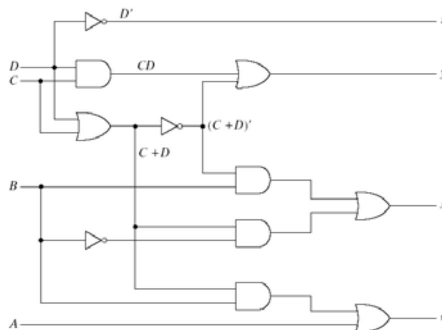
$W = A + BC + BD = A + B(C+D)$

$X = \overline{B}C + \overline{B}D + B\overline{C}\,\overline{D} = \overline{B}(C+D) + B\overline{C}\,\overline{D}$

$Y = CD + \overline{C}\,\overline{D} = \overline{C \oplus D}$

$Z = D$





Code Converter (3/3)

- Data-flow style

```
assign W = A|(B&(C|D));
assign X = ~B&(C|D)|(B&~C&~D);
assign Y = ~(C^D);
assign Z = ~D;
```

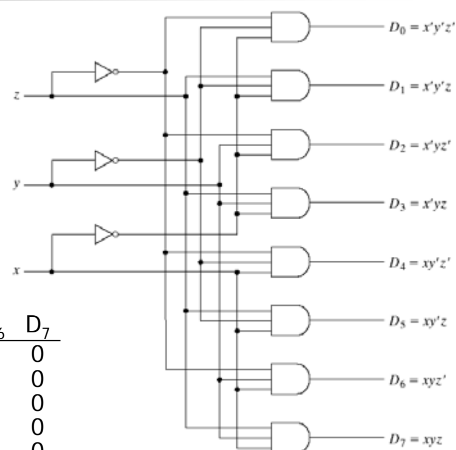
- Behavior style

```
assign ROM_in = {A, B, C, D};
assign {W, X, Y, Z} = ROM_out;
always @(ROM_in) begin
  case (ROM_in)
    4`b0000:  ROM_out = 4`b0011;
    4`b0001:  ROM_out = 4`b0100;
       ⋮
    4`b1001:  ROM_out = 4`b1100;
    default: ROM_out = 4`b0000;
  endcase
end
```

3-25

Decoder (1/2)

- A decoder is to generate the 2^n (or fewer) minterms of n input variables
- Ex: a 3-to-8 line decoder



Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

3-26

Decoder (2/2)

- Behavior style 1

```
input x, y, z;
reg [7:0] D;
always @(x or y or z) begin
  case ({x, y, z})
    3`b000:  D = 8`b00000001;
    3`b001:  D = 8`b00000010;
    ⋮
    3`b111:  D = 8`b10000000;
    default: D = 8`b0;
  endcase
end
```

- Behavior style 2

```
input x, y, z;
wire [2:0] addr;
reg [7:0] D;
assign addr = {x, y, z};

always @(addr) begin
  D = 8`b0;
  D[addr] = 1;
end
```

D is unknown when addr is unknown

Encoder (1/2)

- An encoder performs the inverse operation of a decoder
  - Have 2^n (or fewer) input lines and n output lines
  - The output lines generate the binary code of the input positions
- Only one input can be active at any given time
- Ex: a octal-to-binary encoder

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

$z = D_1 + D_3 + D_5 + D_7$
$y = D_2 + D_3 + D_6 + D_7$
$x = D_4 + D_5 + D_6 + D_7$

Encoder (2/2)

- Data-flow style

```
assign z = D[1] | D[3] | D[5] | D[7];
assign y = D[2] | D[3] | D[6] | D[7];
assign x = D[4] | D[5] | D[6] | D[7];
```

- Behavior style

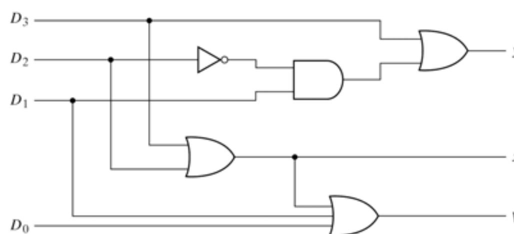
```
always @(D) begin
  case (D)
    8`b00000001:
       {x, y, z} = 3`b000;
    8`b00000010:
       {x, y, z} = 3`b001;
          ⋮
    8`b10000000:
        {x, y, z} = 3`b111;
    default: {x, y, z} = 3`b000;
  endcase
end
```


Priority Encoder (1/2)

- An encoder circuit that includes the priority function
- If two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence

V = 0 : no valid inputs

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1



$x = D_2 + D_3$
$y = D_3 + D_1D'_2$
$V = D_0 + D_1 + D_2 + D_3$

Priority Encoder (2/2)

- Data-flow style

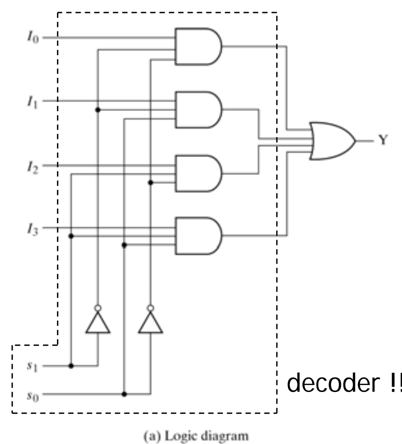
```
assign x = D[2] | D[3];
assign y = D[3] | (~D[2] & D[1]);
assign V = D[0] | D[1] | D[2] | D[3];
```

- Behavior style

```
always @(D) begin
  V = 1;
  casex (D)
    4`b1???:  {x, y} = 2`b11;
    4`b01??:  {x, y} = 2`b10;
    4`b001?:  {x, y} = 2`b01;
    4`b0001:  {x, y} = 2`b00;
    default: begin
      {x, y} = 2`bx;
      V = 0;
    end
  endcase
end
```


Multiplexer (1/2)

- A multiplexer uses *n* selection bits to choose binary info. from a maximum of 2^n unique input lines
- Like a decoder, it decodes all minterms internally
- Unlike a decoder, it has only one output line



(a) Logic diagram

s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

Multiplexer (2/2)

- Behavior style 1

```
input [1:0] S;
input [3:0] I;
output Y;
always @(S or I) begin
  case (S)
    2`b00:  Y = I[0];
    2`b01:  Y = I[1];
    2`b10:  Y = I[2];
    2`b11:  Y = I[3];
    default: Y = 0;
  endcase
end
```

suitable for the cases that inputs are not coded (ex: A, B, C, …)

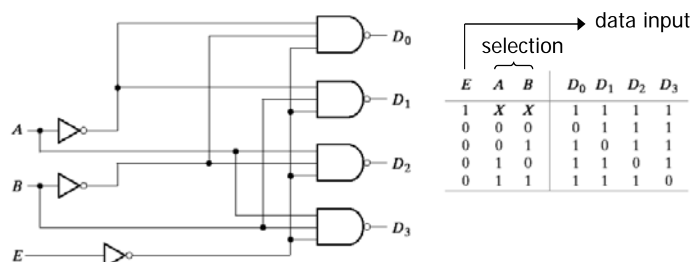
- Behavior style 2

```
input [1:0] S;
input [3:0] I;
output Y;
always @(S or I) begin
  Y = I[S];
end
```

3-33

Demultiplexer (1/2)

- It performs the inverse function of a multiplexer
- It receives info from a single line and transmits it to one of 2^n possible output lines
- A decoder with enable input can function as a demultiplexer
  - Often referred to as a ***decoder/demultiplexer***



E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

3-34

Demultiplexer (2/2)

- Behavior style 1

```
input A, B, E;
reg [3:0] D;
always @(A or B or E) begin
  D = 4`b1111;
  case ({A, B})
    2`b00:  D[0] = E;
    2`b01:  D[1] = E;
    2`b10:  D[2] = E;
    2`b11:  D[3] = E;
    default: D = 4`b0;
  endcase
end
```

- Behavior style 2

```
input A, B, E;
wire [1:0] S;
reg [3:0] D;
assign S = {A, B};
always @(S or E) begin
  D = 4`b1;
  D[S] = E;
end
```


Binary Adder Cell (1/2)

□ **TABLE 3-7**
Truth Table of Half Adder

Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 3-7 Truth Table of Half Adder

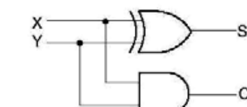


Fig. 3-25 Logic Diagram of Half Adder

□ **TABLE 3-8**
Truth Table of Full Adder

Inputs			Outputs	
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 3-8 Truth Table of Full Adder

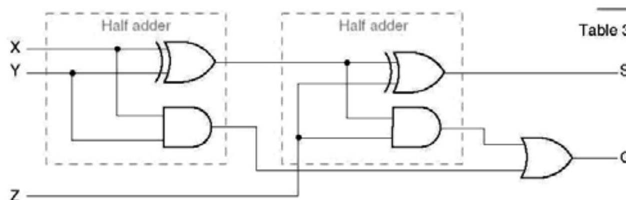


Fig. 3-27 Logic Diagram of Full Adder

Binary Adder Cell (2/2)

- Half adder

```
assign {C, S} = X + Y;

assign C = X & Y;
assign S = X ^ Y;
```

- Full adder

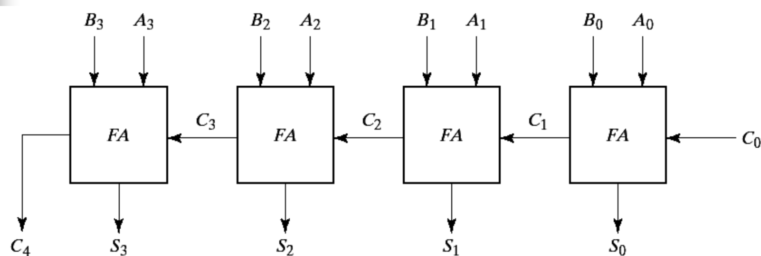
```
assign {C, S} = X+Y+Z;

assign C = (X&Y) | Z&(X^Y);
assign S = X ^ Y ^ Z;
```

preferred writing style that can tell synthesizer the existence of an adder

3-37

Ripple Carry Adder




```
module FA4 (S, C4, A, B, C0);
  input [3:0] A, B;
  input C0;
  output [3:0] S;
  output C4;
  FA1 U0(S[0], C1, A[0], B[0], C0);
  FA1 U1(S[1], C2, A[1], B[1], C1);
  FA1 U2(S[2], C3, A[2], B[2], C2);
  FA1 U3(S[3], C4, A[3], B[3], C3);
endmodule
```

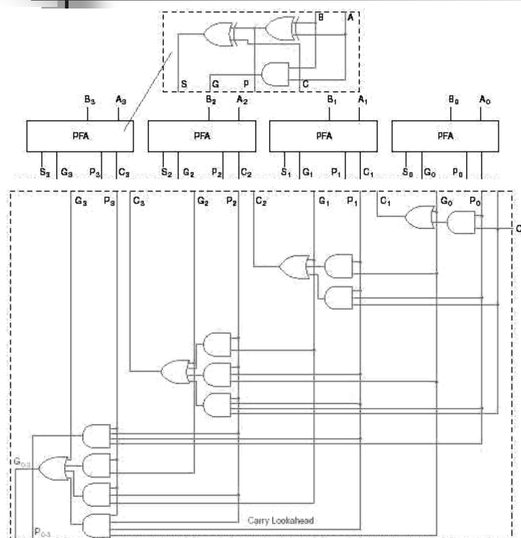
3-38

Carry Propagation

- The ***carry propagation time*** is a limiting factor on the speed with which two numbers are added
- All other arithmetic operations are implemented by successive additions
  - The time consumed during the addition is very critical
- To reduce the carry propagation delay
  - Employ faster gates with reduced delays
  - Increase the equipment complexity
- Several techniques for reducing the carry propagation time in a parallel adder
  - The most widely used technique employs the principle of ***carry lookahead***

3-39

Carry Lookahead Adder

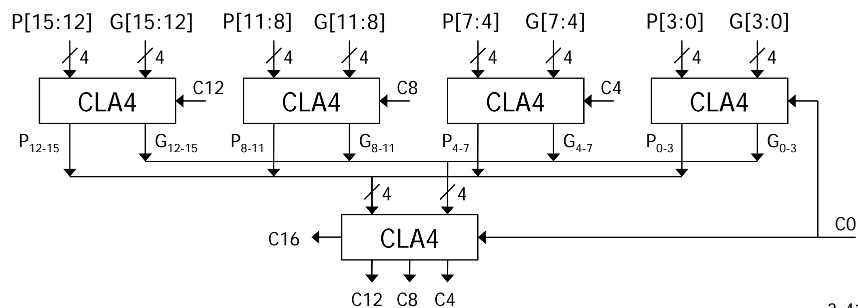


- The critical path "carry" is calculated separately to reduce delay
- $G_n = A_nB_n$
  $P_n = A_n \wedge B_n$
  $S_n = P_n \wedge C_n$
  $C_n = G_{n-1} + P_{n-1}C_{n-1}$
- $C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$
- Gate-level descriptions are often used

3-40

Hierarchical CLA

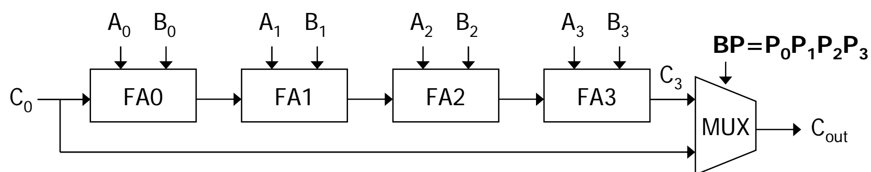
- $C_n = G_{n-1} + P_{n-1}C_{n-1}$
- $C_4 = (G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0)+(P_3P_2P_1P_0)C_0$
  - ☞ $P_{0\text{-}3} = P_3P_2P_1P_0$
    $G_{0\text{-}3} = G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0$



3-41

Carry Skip Adder

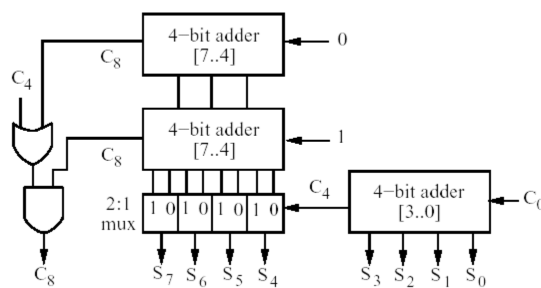
- Looks for the cases in which the carry out is identical to carry in
  - All propagation
  - The most time-consuming path
- For such cases, ***bypass*** the original carry path
  - Generate carry out directly (Cout=Cin)
  - Use a MUX to select



3-42

Carry Select Adder

- Compute two results in parallel, each for different carry assumptions
- Use actual carry in to select correct results
- Reduce delay to a multiplexer



3-43

Comparison of Adders

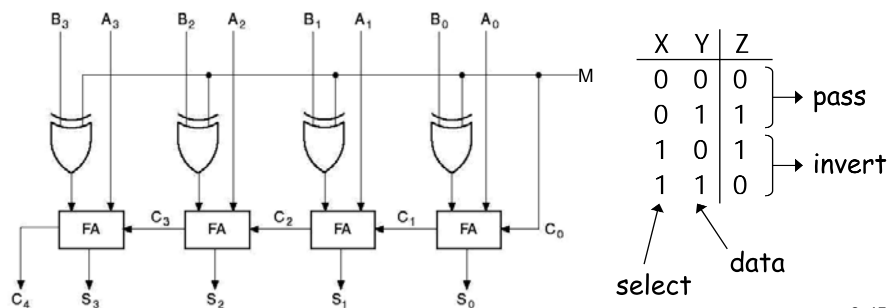
- Make comparison between Time ***(speed)*** and Space ***(area)***
- Empirical average power consumption of 16-bit adders

Adder	Time	Space	Power (mW)
Ripple carry adder	$O(n)$	$O(n)$	0.117
Carry look-ahead adder	$O(\lg n)$	$O(n \lg n)$	0.171
Carry skip adder	$O(\sqrt{n})$	$O(n)$	0.109
Carry select adder	$O(\sqrt{n})$	$O(n)$	0.216

3-44

Binary Adder-Subtractor (1/2)

- Subtraction can be achieved by adding 2's compliment
  - A – B = A + (-B)
  - Compliment each bit of B then add 1 to the result
- M = 0: addition
- M = 1: supply the additional "1" in subtraction



3-45

Binary Adder-Subtractor (2/2)

```
module Add_Sub (S, C4, A, B, M);
  input [3:0] A, B;
  input M;  // 0: Add, 1: Sub
  output [3:0] S;
  output C4;
  wire [3:0] K;
  assign K = B ^ {4{M}};
  FA1 U0(S[0], C1, A[0], K[0], M);
  FA1 U1(S[1], C2, A[1], K[1], C1);
  FA1 U2(S[2], C3, A[2], K[2], C2);
  FA1 U3(S[3], C4, A[3], K[3], C3);
endmodule
```

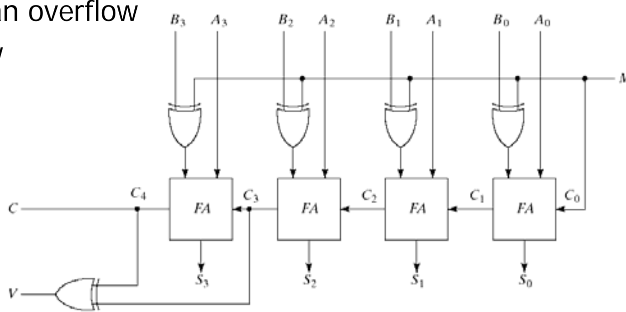
3-46

Signed Adder-Subtractor

- Unsigned
  - C detects a ***carry*** after addition or a ***borrow*** after subtraction
- Signed
  - V bit detects an overflow

    0: no overflow

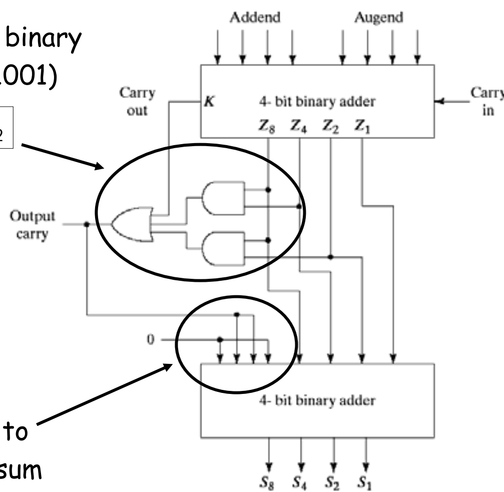
    1: overflow



BCD Adder (1/2)

C detects whether the binary sum is greater than 9(1001)

$C = K + Z_8Z_4 + Z_8Z_2$



If C=1, it is necessary to add 6(0110) to binary sum

BCD Adder (2/2)

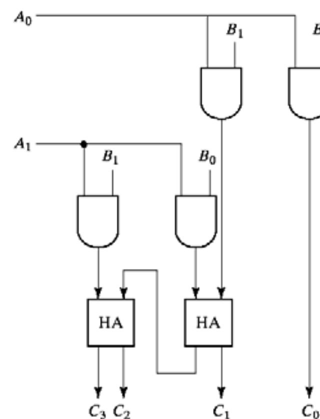
```
module bcd_add (S, Cout, A, B, Cin);
  input [3:0] A, B;
  input Cin;
  output [3:0] S;
  output Cout;
  reg [3:0] S;
  reg Cout;
  always @(A or B or Cin) begin
     {Cout, S} = A + B + Cin;
     if (Cout != 0  ||  S > 9) begin
        S = S + 6;
        Cout = 1;
     end
  end
endmodule
```

3-49

Binary Multiplier

- Ex: 2 x 2 multiplier

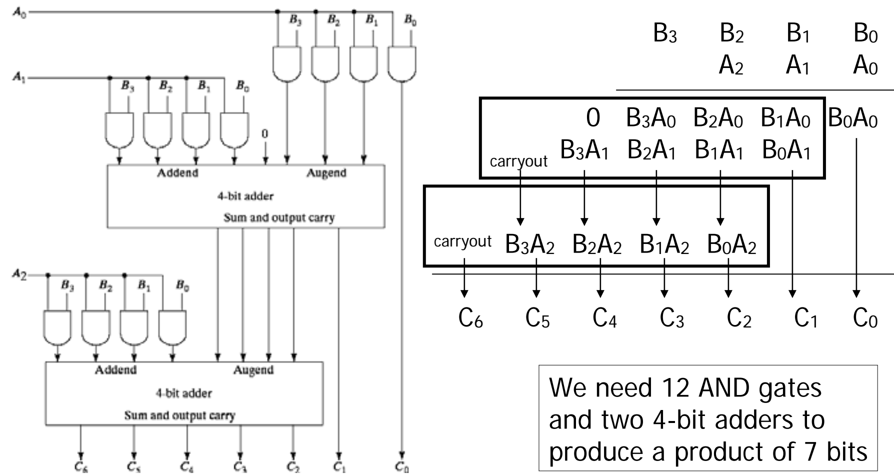
		B_1	B_0
		A_1	A_0
		A_0B_1	A_0B_0
	A_1B_1	A_1B_0	
C_3	C_2	C_1	C_0



- Compute partial products, and justify the sum of those partial products
- *m* x *n* digit multiplication generates up to an (*m*+*n*) digit result
- Can be described using * (not recommended)

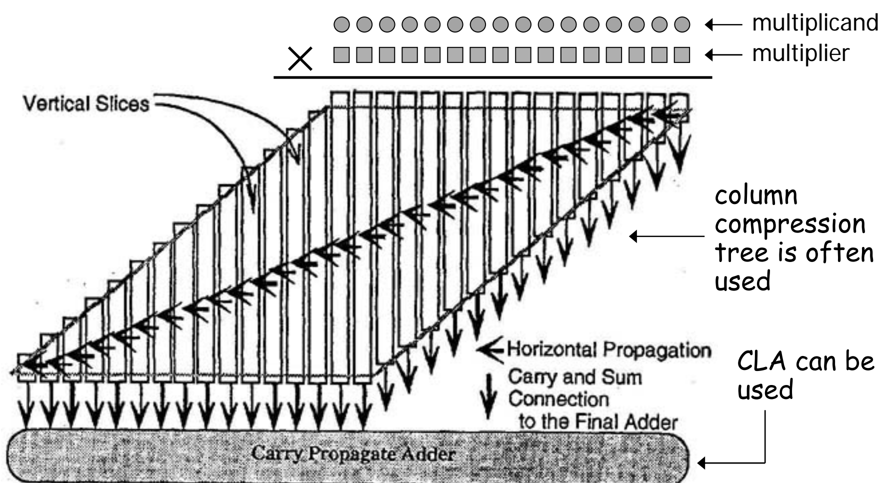
3-50

4-Bit By 3-Bit Binary Multiplier



3-51

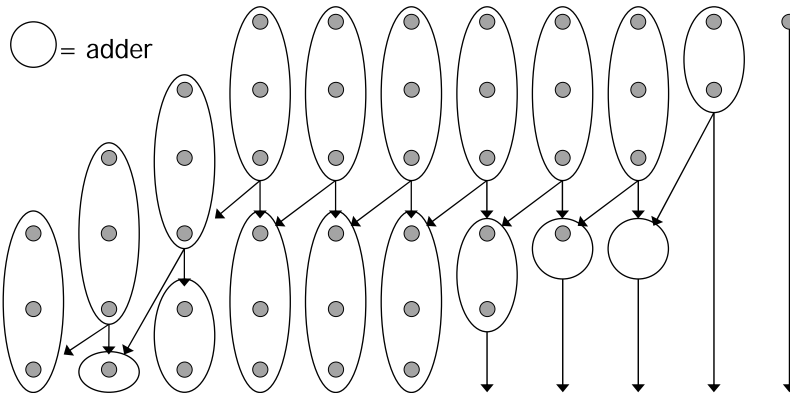
General Multiplier Architecture



3-52

Column Compression Tree

- Also called ***Wallace Tree***
- Use “carry in” to add three components at a time
- “carry out” is added in next column



Signed Multiplication

- Case: negative multiplicand, positive multiplier → sign extension works

```
                              1 1 0 1 0   (−6)
                            × 0 1 1 0 1   (+13)
              ---------------------------
             ┌ 1 1 1 1 1 1 1 0 1 0
    sign     │ 0 0 0 0 0 0 0 0 0
  extension  │ 1 1 1 1 1 0 1 0
             │ 1 1 1 1 0 1 0
             └ 0 0 0 0 0 0
              ---------------------------
               1 1 1 0 1 1 0 0 1 0   (−78)
```

- Other cases can be handled by applying 2’s complement to appropriate numbers