

Verilog class

M. Mahdi Assefzadeh
Spring 2009



SESSION 4

**CARRY-LOOKAHEAD ADDER (CLA)
MULTIPLICATION IN VERILOG
COMBINATIONAL CIRCUITS (DECODERS, ENCODERS, MULTIPLEXERS) IN VERILOG**

Some figures belong to “Fundamentals of digital logic with Verilog design” by S. Brown and Z. Vranesic

CLA

2

Compare these two, 2-bit Ripple carry adder and 2-bit CLA :

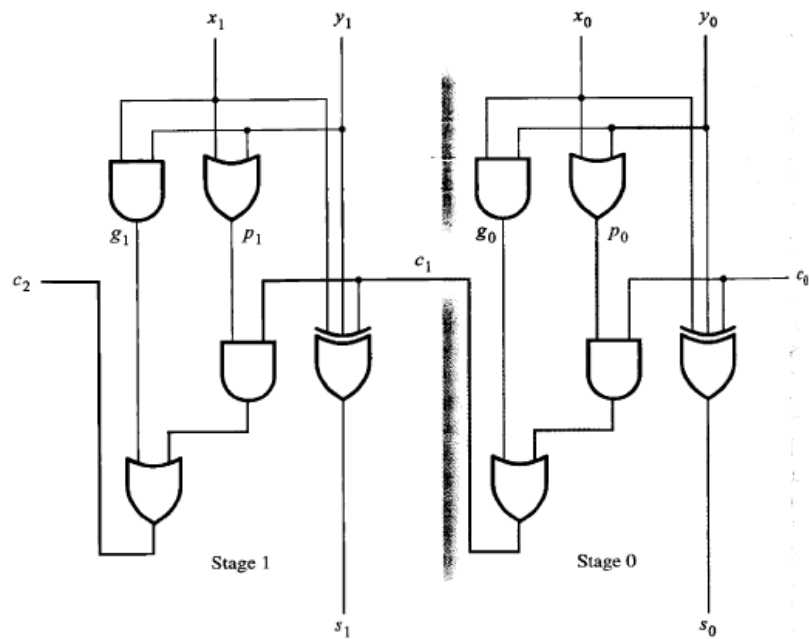


Figure 5.15 A ripple-carry adder based on expression 5.3.

VS.

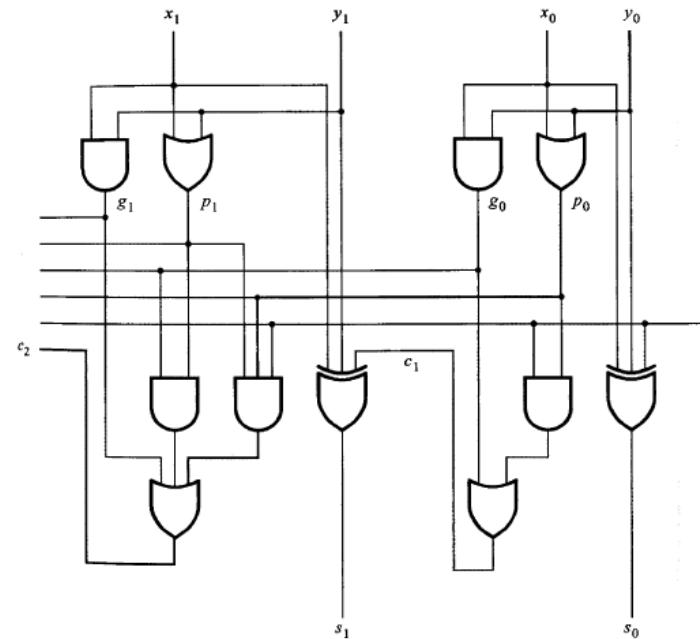


Figure 5.16 The first two stages of a carry-lookahead adder.

CLA vs. RCA

3

- First, what is the difference in boolean expressions?
RCA uses these expressions: ('i' is number of stage)

$$c_{i+1} = g_i + p_i c_i$$

$$s_i = x_i \oplus y_i \oplus c_i$$

While CLA uses these expressions:

$$\begin{aligned} c_{i+1} &= g_i + p_i(g_{i-1} + p_{i-1}c_{i-1}) \\ &= g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-1} \\ &= g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \dots + p_i p_{i-1} \dots p_2 p_1 g_0 + p_i p_{i-1} \dots p_1 p_0 c_0 \end{aligned}$$

$$s_i = x_i \oplus y_i \oplus c_i$$

CLA vs. RCA

4

- CLA is much faster than RCA. We can see an example of this speed comparison in the worst case delays of this two circuits. For RCA and CLA, from "y0" to "c2" we have 5 and 3 gates. So it shows that a 2-bit CLA's delay is 3/5 of a 2-bit RCA's delay. Also we can calculate these for 3-bit adders.

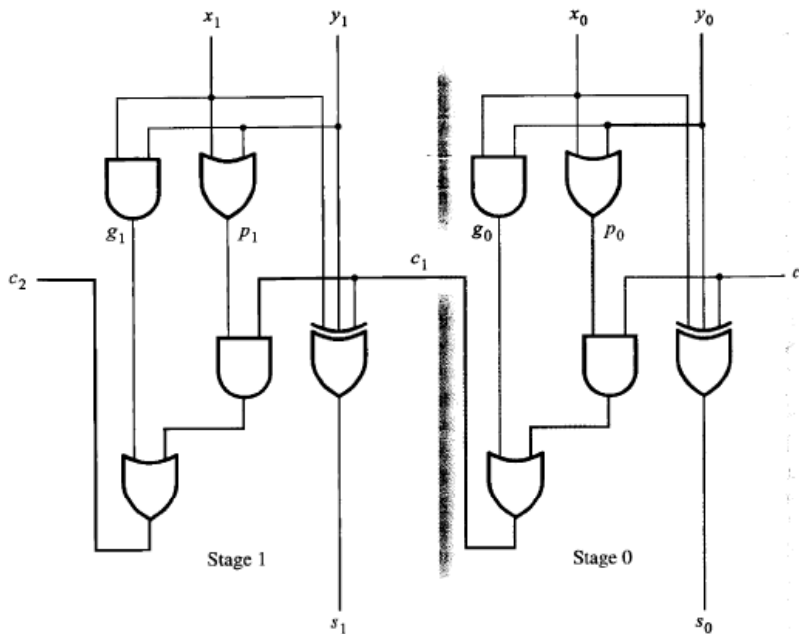


Figure 5.15 A ripple-carry adder based on expression 5.3.

VS.

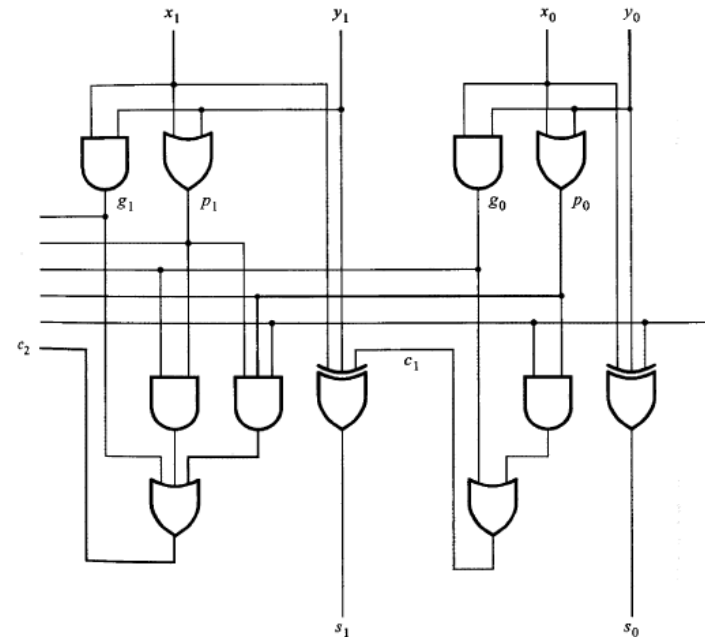


Figure 5.16 The first two stages of a carry-lookahead adder.

Hierarchical CLA

5

- Like what we did to FAs and we reached RCA, we can have a **Hierarchical carry-lookahead adder WITH ripple carry between blocks**, for example a 32-bit adder made by ripple-carry between 8-bit CLA blocks.

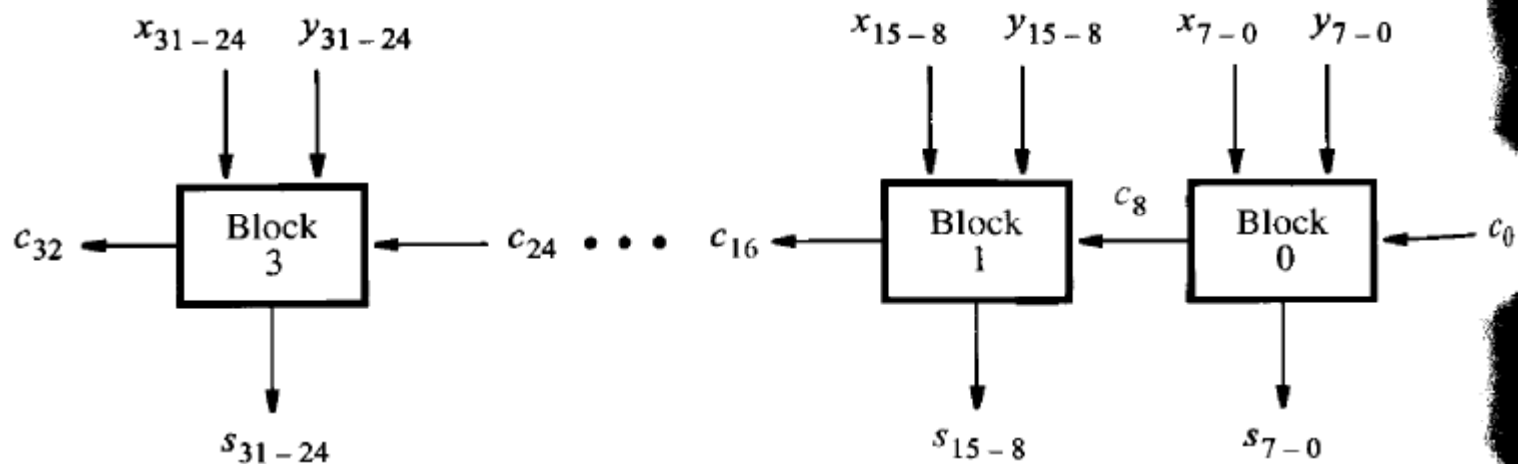


Figure 5.17 A hierarchical carry-lookahead adder with ripple-carry between blocks.

Hierarchical CLA

6

- Again we can have something faster using the former CLA algorithm, **Hierarchical carry-lookahead adder.**

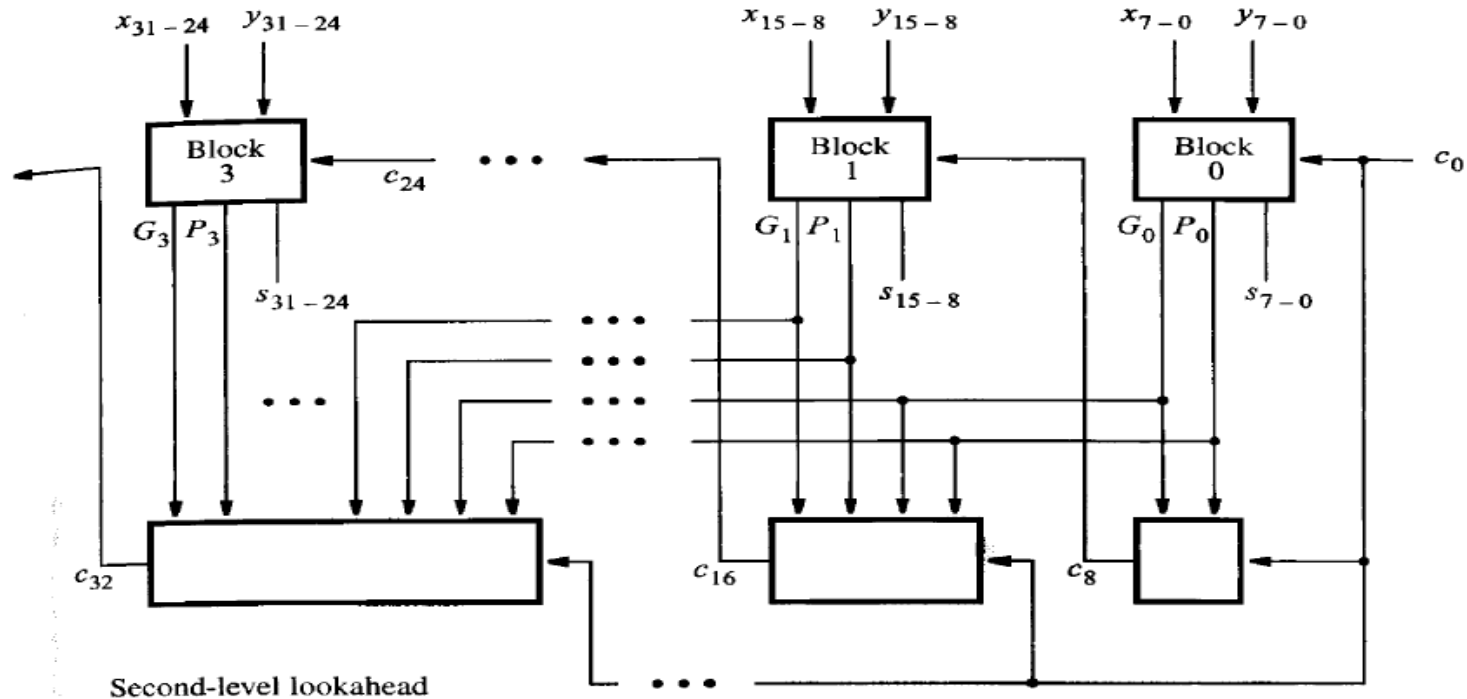


Figure 5.18 A hierarchical carry-lookahead adder.

74S182

4-bit CLA

CLA's Verilog?!

8

- What happens to CLA's Verilog code?

we need to be careful about that “beside CAD Tools, what we write in HDL is also responsible for the resulting circuit!”.

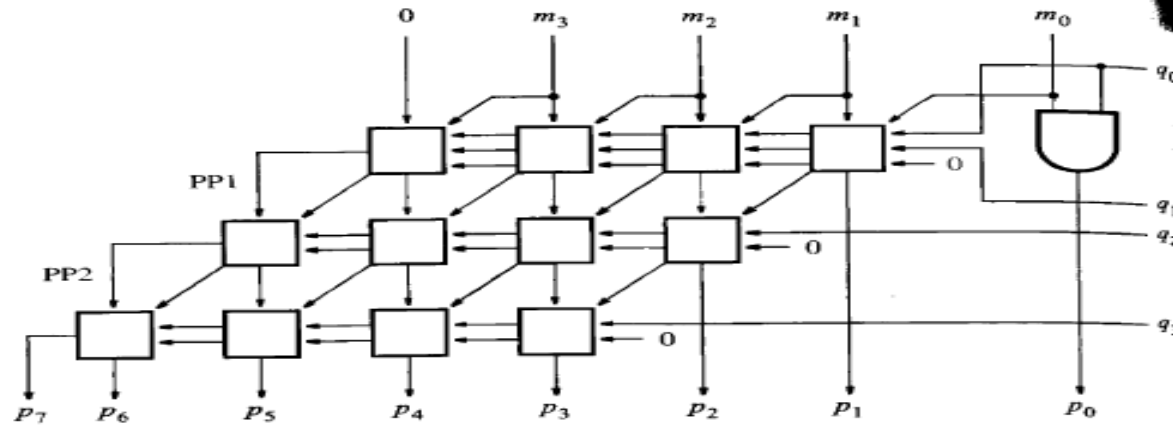
for example, the difference between RCA and CLA is in Timing, not in functionality and ...

so, in these situations we should be more Structural than just describing the functionality, because timing is important.

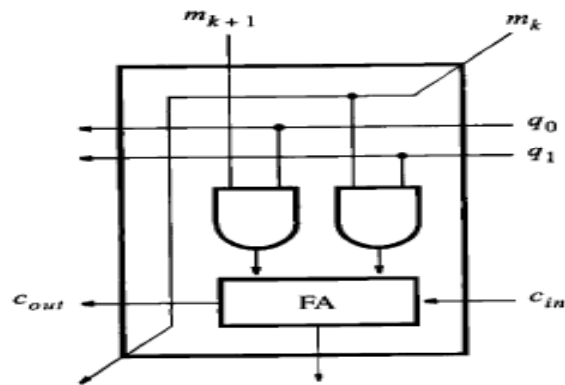
Multiplication

9

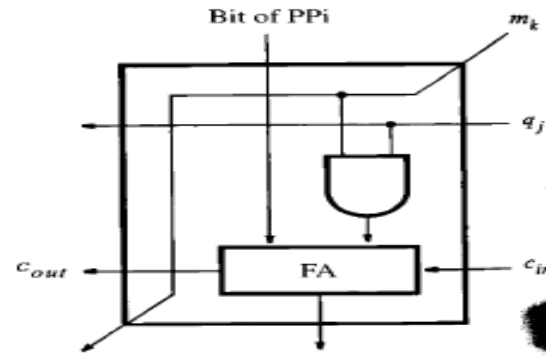
- Array multiplier for unsigned numbers structure,



(a) Structure of the circuit



(b) A block in the top row



(c) A block in the bottom two rows

Multiplication

10

- An example code for Array multiplier for unsigned numbers.

```
module newca2 (s, v, m_in, q_in);

    input [7:0] m_in, q_in;
    output [15:0] s;
    output v;
    wire [7:0] m, q, m_in, q_in;
    wire [15:0] pp0, pp1, pp2, pp3, pp4, pp5, pp6, pp7;
    reg pp0, pp1, pp2, pp3, pp4, pp5, pp6, pp7;
    reg m, q, s, v;

    integer i, ctl;
    always@(*)
    begin

        ctl=0;
        m = m_in;
        q = q_in;
        if (q_in[7]==1)
        begin
            q = ~q_in + 1;
            ctl = 1;
        end
        if (m_in[7]==1)
        begin
            m = ~m_in + 1;
            ctl = 1;
        end
        if ( (q_in[7]==1)&(m_in[7]==1) )
            ctl = 0;
    end
endmodule
```

```
pp0=0;
pp1=0;
pp2=0;
pp3=0;
pp4=0;
pp5=0;
pp6=0;
pp7=0;
```

```
for (i=0; i<=7; i=i+1)
    pp0[i] = q[0] & m[i];
```

```
for (i=0; i<=7; i=i+1)
    pp1[i] = q[1] & m[i];
pp1 = pp1 + pp0/2;
```

```
for (i=0; i<=7; i=i+1)
    pp2[i] = q[2] & m[i];
pp2 = pp2 + pp1/2;
```

```
for (i=0; i<=7; i=i+1)
    pp3[i] = q[3] & m[i];
pp3 = pp3 + pp2/2;
```

```
for (i=0; i<=7; i=i+1)
    pp4[i] = q[4] & m[i];
pp4 = pp4 + pp3/2;
```

```
for (i=0; i<=7; i=i+1)
    pp5[i] = q[5] & m[i];
pp5 = pp5 + pp4/2;

for (i=0; i<=7; i=i+1)
    pp6[i] = q[6] & m[i];
pp6 = pp6 + pp5/2;

for (i=0; i<=7; i=i+1)
    pp7[i] = q[7] & m[i];
pp7 = pp7 + pp6/2;

s[15:7] = pp7[8:0];
s[6] = pp6[0];
s[5] = pp5[0];
s[4] = pp4[0];
s[3] = pp3[0];
s[2] = pp2[0];
s[1] = pp1[0];
s[0] = pp0[0];

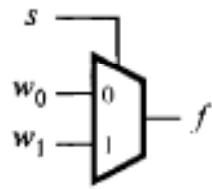
v=0;
if (s[15]==1) v=1;

if (ctl)
    s = ~s + 1;
end
endmodule
```

- A problem of last page code is the use of “ctl” variable to use an unsigned multiplication for signed mult.
 - There is a direct way for mult. of signed numbers (when the multiplicand is positive) and it goes through sign extension subject and using of sign extended extra bit.
- for more info : Digital Logic (Brown-Vranesic) part 5.6.2 .

2-to-1 MUX

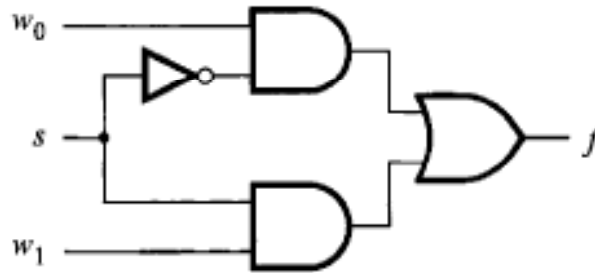
14



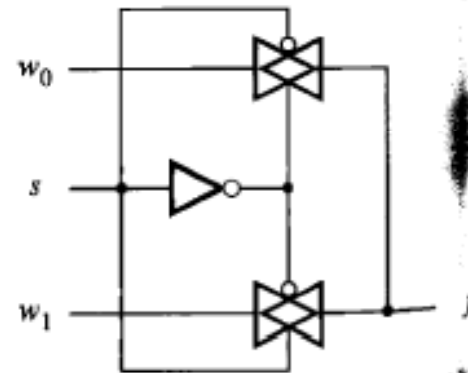
(a) Graphical symbol

s	f
0	w_0
1	w_1

(b) Truth table



(c) Sum-of-products circuit

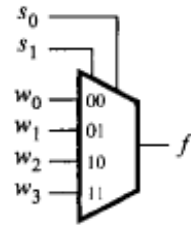


(d) Circuit with transmission gates

Figure 6.1 A 2-to-1 multiplexer.

4-to-1 MUX

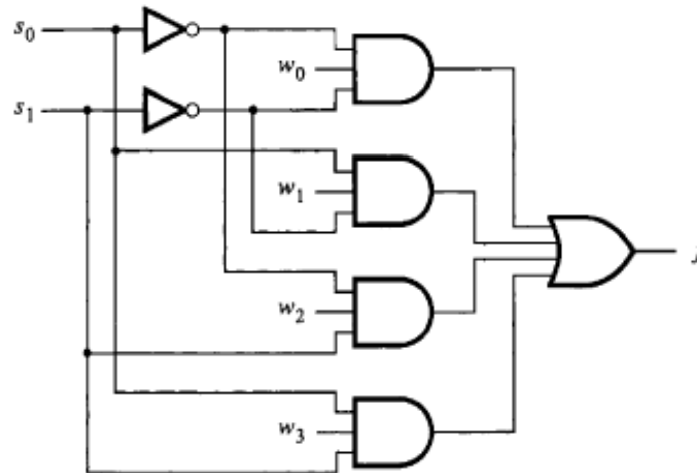
15



(a) Graphical symbol

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

(b) Truth table



(c) Circuit

Figure 6.2 A 4-to-1 multiplexer.

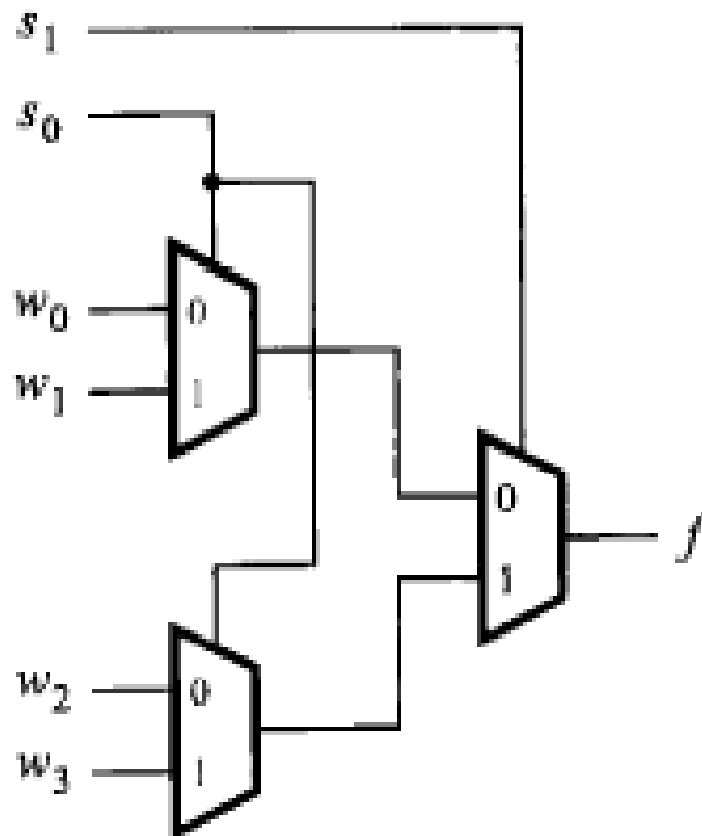


Figure 6.3 Using 2-to-1 multiplexers to build a 4-to-1 multiplexer.

MUX

17

- Last page came of from here,

$$f = \bar{s}_1 \bar{s}_0 w_0 + \bar{s}_1 s_0 w_1 + s_1 \bar{s}_0 w_2 + s_1 s_0 w_3$$

MUX

18

- **Shannon's Expansion Theorem:**

Any Boolean function $f(w_1, \dots, w_n)$ can be written in the form

$$f(w_1, w_2, \dots, w_n) = \bar{w}_1 \cdot f(0, w_2, \dots, w_n) + w_1 \cdot f(1, w_2, \dots, w_n)$$

16-to-1 MUX using 4-to-1 MUXEs

19

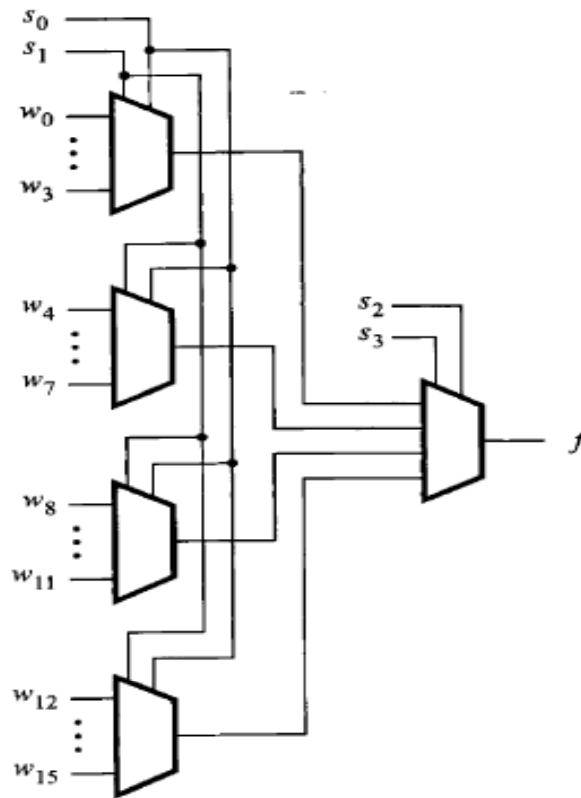


Figure 6.4 A 16-to-1 multiplexer.

2-to-1 MUX code

20

```
module mux2to1(w0, w1, s, f)
  input w0, w1, s;
  output f;
  assign f = s ? w1 : w0;
endmodule
```

2-to-1 MUX code

21

- **First code:**

```
module mux4to1(w0, w1, w2, w3, s, f);  
    input w0, w1, w2, w3;  
    input [1:0] s;  
    output f;  
    assign f = s[1] ? (s[0] ? w3:w2) : (s[0]? w1:w0);  
endmodule
```

2-to-1 MUX code

22

- Second code, using case statement :

HOLD ON!

**where do we use “case” or “if” statements?
in an “always@”.**

- **Reviewing case statement:**

case (expression)

 alternative1: statement;

 alternative2: statement;

•

•

•

 alternativej: statement

 [**default**: statement]

endcase

2-to-1 MUX code

24

- Second code, using case statement :

...

```
reg f;
```

```
always@(*)
```

```
  case (s)
```

```
    0 : f = w[0];
```

```
    1 : f = w[1];
```

```
    2 : f = w[2];
```

```
    3 : f = w[3];
```

```
  endcase
```

```
end
```

...

2-to-1 MUX code

25

- Third, using “if-else” :

...

```
if (s==0)  
    f = w[0];  
else if (s==1)  
    f = w[1];  
else if (s==2)  
    f = w[2];  
else if (s==3)  
    f = w[3];
```

...

Cascading Methods

26

- We have two cascading methods,
 - 1. using enable inputs, and for example using four 4-to-1 resulting to one 16-to-1
 - **2. using five 2-to-1 resulting to one 16-to-1.**

We look at a structural code for the 2nd Algorithm of cascading.

Cascading method No. 2

27

```
module mux16to1(w, s, f, m);  
    input [0:15] w;  
    input [3:0] s;  
    output f;  
    output [3:0] m;  
    wire [3:0] m;  
    mux4to1 mux1(w[0:3], s[1:0], m[0]);  
    mux4to1 mux2(w[4:7], s[1:0], m[1]);  
    mux4to1 mux3(w[8:11], s[1:0], m[2]);  
    mux4to1 mux4(w[12:15], s[1:0], m[3]);  
    mux4to1 mux5(m[0:3], s[3:2], f);  
endmodule
```

Decoder

28

Example1 code of 2-to-4 decoder:

```
module dec2to4 (W, Y, En);
  input [1:0] W;
  input En;
  output [0:3] Y;
  reg [0:3] Y;

  always @(W or En)
    case ({En, W})
      3'b100: Y = 4'b1000;
      3'b101: Y = 4'b0100;
      3'b110: Y = 4'b0010;
      3'b111: Y = 4'b0001;
      default: Y = 4'b0000;
    endcase
endmodule
```

Decoder

29

- **Example2 of 2-to-4 decoder:**

```
module dec2to4 (W, Y, En);
  input [1:0] W;
  input En;
  output [0:3] Y;
  reg [0:3] Y;

  always @(W or En)
  begin
    if (En == 0)
      Y = 4'b0000;
    else
      case (W)
        0: Y = 4'b1000;
        1: Y = 4'b0100;
        2: Y = 4'b0010;
        3: Y = 4'b0001;
      endcase
    end
  end
endmodule
```

One cascading method resulting 4-to-16 decoder

30

```
module dec4to16(W, Y, En);
    input [3:0] W;
    input En;
    output [0:15] Y;
    wire [0:3] M;

    dec2to4 Dec1 (W[3:2], M[0:3], En);
    dec2to4 Dec2 (W[1:0], Y[0:3], M[0]);
    dec2to4 Dec3 (W[1:0], Y[4:7], M[1]);
    dec2to4 Dec4 (W[1:0], Y[8:11], M[2]);
    dec2to4 Dec5 (W[1:0], Y[12:15], M[3]);

endmodule
```

```
module dec2to4(W, Y, En);
    input [1:0] W;
    input En;
    output [0:3] Y;
    reg [0:3] Y;

    always @(W or En)
        case ({En, W})
            3'b100: Y = 4'b1000;
            3'b101: Y = 4'b0100;
            3'b110: Y = 4'b0010;
            3'b111: Y = 4'b0001;
            default: Y = 4'b0000;
        endcase

endmodule
```

Code for a BCD-to-7segment decoder

31

```
module seg7 (bcd, leds);
  input [3:0] bcd;
  output [1:7] leds;
  reg [1:7] leds;

  always @(bcd)
    case (bcd)      //abcdefg
      0: leds = 7'b1111110;
      1: leds = 7'b0110000;
      2: leds = 7'b1101101;
      3: leds = 7'b1111001;
      4: leds = 7'b0110011;
      5: leds = 7'b1011011;
      6: leds = 7'b1011111;
      7: leds = 7'b1110000;
      8: leds = 7'b1111111;
      9: leds = 7'b1111011;
      default: leds = 7'bx;
    endcase
endmodule
```

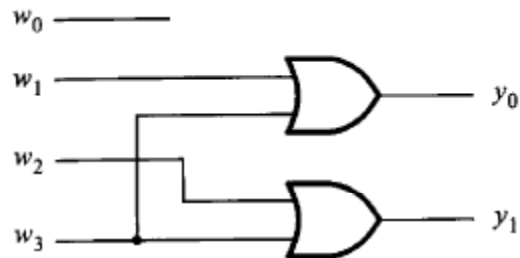
Encoder

32

- A very simple circuits for a 4-to-2 encoder

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

(a) Truth table



(b) Circuit

Figure 6.23 A 4-to-2 binary encoder.

Code of a priority encoder

33

- Code of a priority 4-to-2 encoder , using a new case “**casex(expression)**”:

```
module priority(w, y, z);  
  input [3:0] w;  
  output [1:0] y;  
  output z;  
  reg [1:0] y;  
  reg z;
```

```
  always@(*)  
  begin
```

```
    z = 1;  
    casex(w)
```

```
      4'b1xxx : y=3;  
      4'b01xx : y=2;  
      4'b001x : y=1;  
      4'b0001 : y=0;
```

```
    default : begin
```

```
      z = 0;  
      y = 2'bxx;  
    end
```

```
  endcase
```

```
end  
endmodule
```

'x' as a “don't care” bit



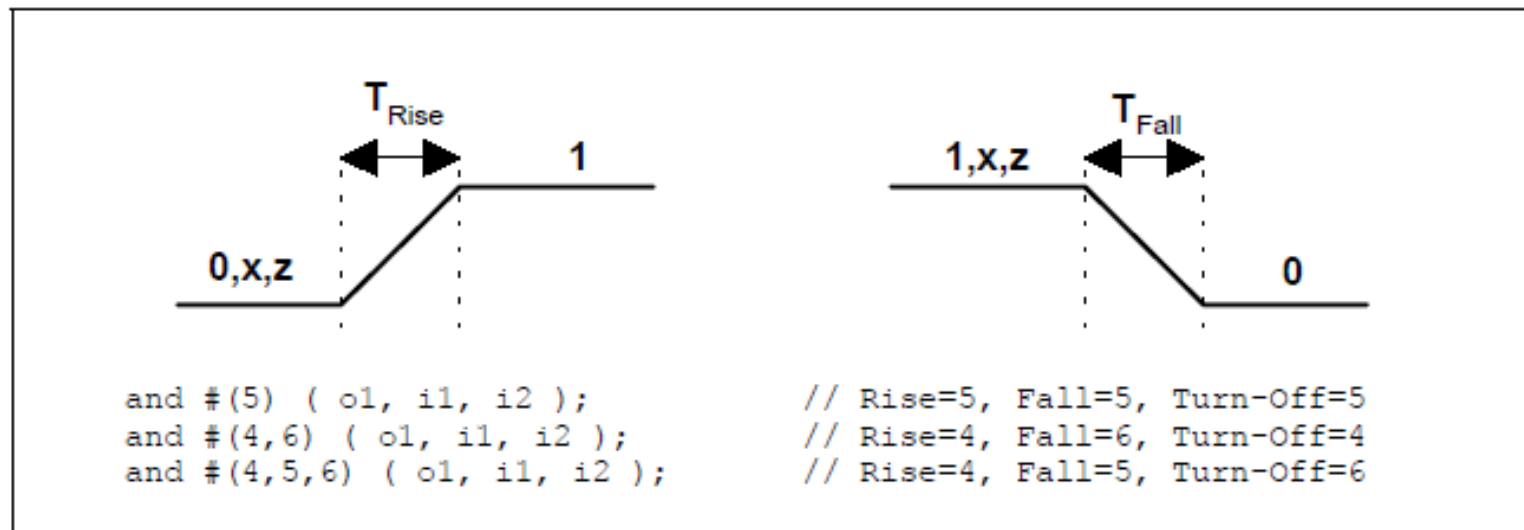
Finishing Combinational Circuits

...

Delay handling

35

- When we want to write something used for simulating, handling delays for gates or transistors is useful.



This figure is copied
from “Verilog
Tutorial” by “saeed
safari”.

Delay handling

36

- And its rules:

تاخیرهای 1-2-5 Turn-Off, Fall, Rise

- هرگونه تغییر در خروجی گیت از 0, x, z به 1 با تاخیر Rise انجام می‌شود.
- هرگونه تغییر در خروجی گیت از 1, x, z به 0 با تاخیر Fall انجام می‌شود.
- هرگونه تغییر در خروجی گیت به z با تاخیر Turn-Off انجام می‌شود.
- اگر خروجی گیت به x تغییر وضعیت بدهد، مینیموم این سه تاخیر در نظر گرفته می‌شود.
- به هنگام نمونه‌سازی از گیت اگر:
 - فقط یک تاخیر مشخص شود، این مقدار برای تمام تاخیرها در نظر گرفته می‌شود.
 - دو تاخیر مشخص شود، به ترتیب برای تاخیرهای Rise و Fall در نظر گرفته می‌شوند و تاخیر Turn-Off برابر مینیموم این دو مقدار در نظر گرفته می‌شود.
 - سه تاخیر مشخص شود، به ترتیب برای تاخیرهای Rise و Fall و Turn-Off در نظر گرفته می‌شوند.

Shift Operators

37

B = A << 1; or B = A >> 1;

While shifting the vacant bit positions are filled with '0's.

Shifting

38

- **Unlike shifting left, the pure shifting right is like dividing by 2, but this is not always correct, because...?**

Shifting right

39

Positive Numbers :

shifting is the same as mult/div by 2.

$$00101 \gg 1 == 00010$$

$$5 / 2 = 2$$

Negative Numbers:

$$11110 \gg 1 == 01111$$

$$-2 / 2 \neq 15 \times$$

we should perform a bit extension,
it means that we shift the number, but for the 2's comp.
sys. sign bit vacant we recover and keep the sign bit. So that:

$$11110/2 = 11111$$

$$-2 / 2 = -1$$