

Verilog class

M. Mahdi Assefzadeh
Spring 2009

1

**SESSION 3
APRIL 13TH**

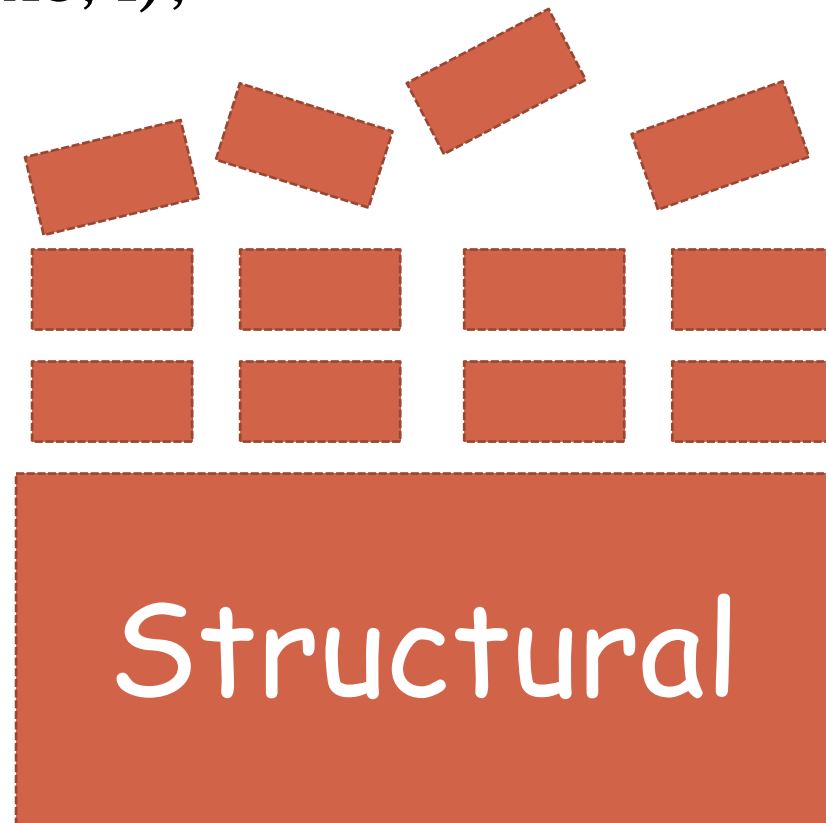
**QUICK REVIEW
TRANSMISSION GATE
QUICK REVIEW OF NUMBERS
VECTORS
FULL ADDER AND HALF ADDER IN VERILOG
RIPPLE CARRY ADDER IN VERILOG**

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

Quick review

2

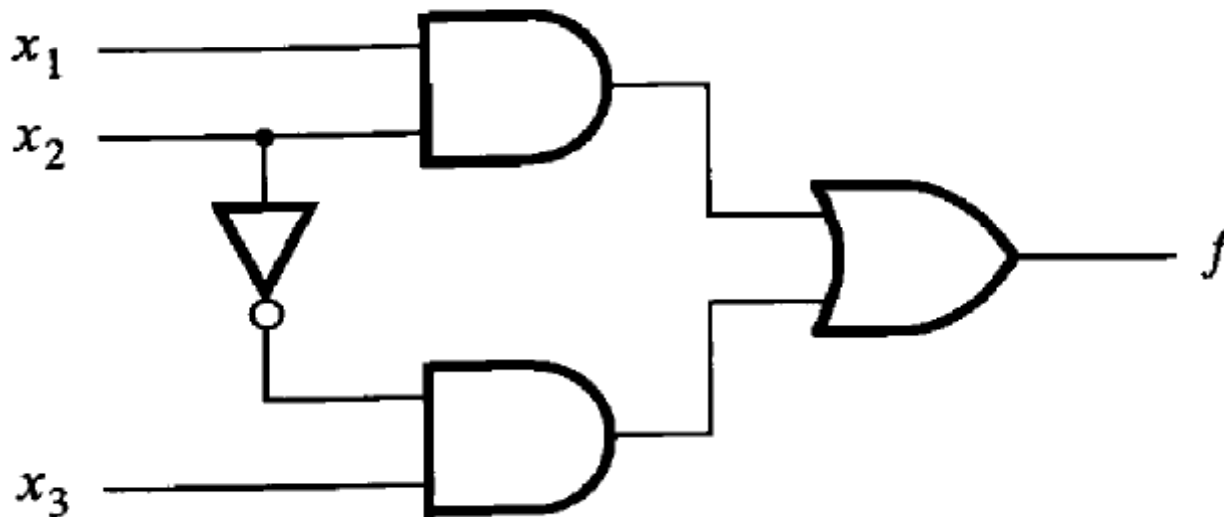
```
module example1 (x1, x2, x3, f);  
  input x1, x2, x3;  
  output f;  
  
  and (g, x1, x2);  
  not (k,x2);  
  and (h, k, x3);  
  or (f, g, h);  
  
endmodule
```



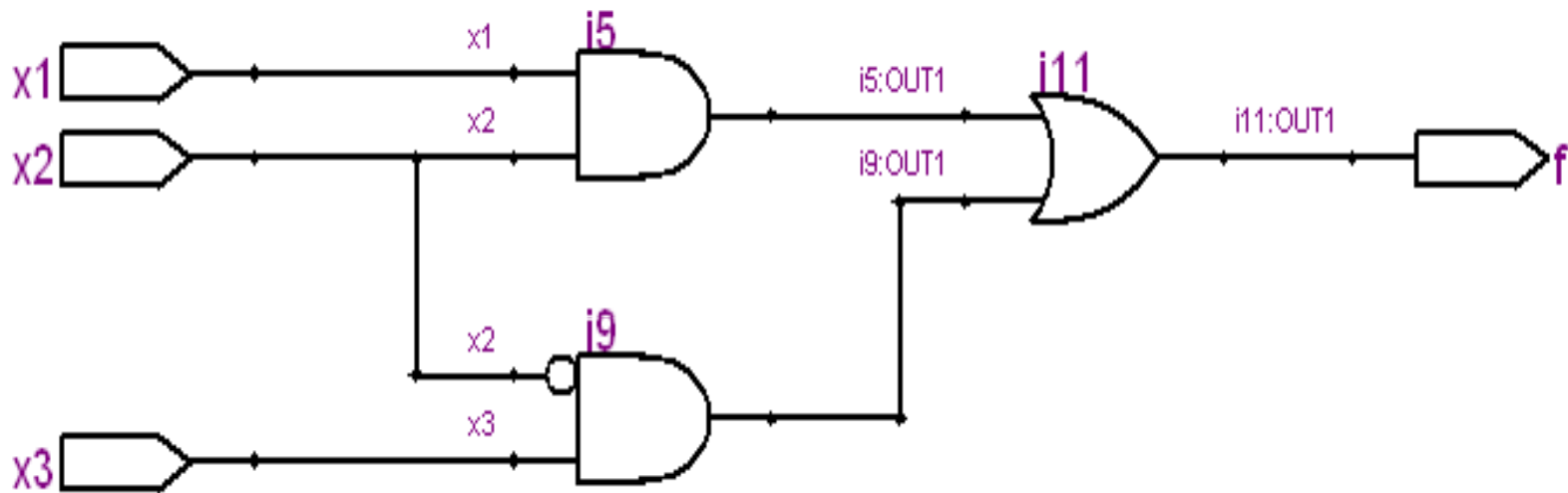
Quick Review

3

The circuit made by structural code.



From RTL (register transfer level) viewer :



- **What was the difference?**

NOT gate , **A bubble**

Is there any problem in the code? 'g', 'h' and 'k' are not defined as type "wire"?

NO! no problem, compiler passes it.

```
module example1 (x1, x2, x3, f);
```

```
    input x1, x2, x3;
```

```
    output f;
```

```
    and (g, x1, x2);
```

```
    not (k,x2);
```

```
    and (h, k, x3);
```

```
    or (f, g, h);
```

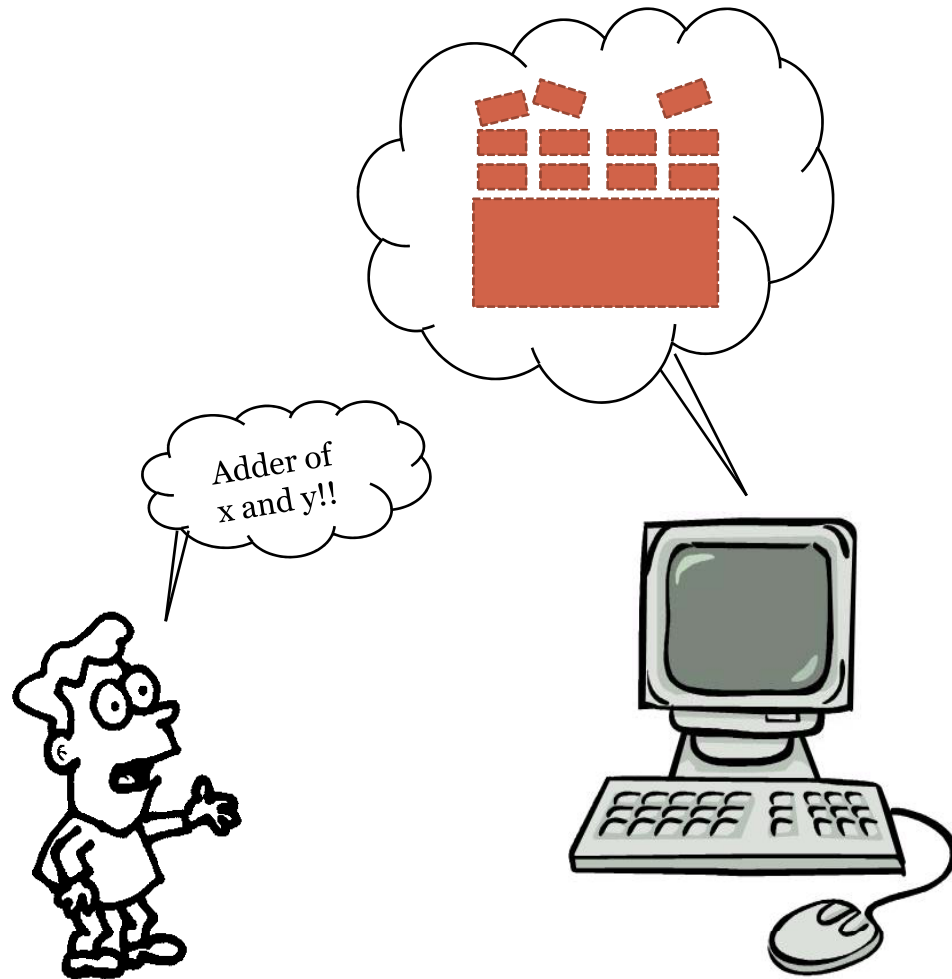
```
endmodule
```

Last example, behavioral code:

```
// Behavioral specification
module example5 (x1, x2, x3, f);
  input x1, x2, x3;
  output f;
  reg f;

  always @(x1 or x2 or x3)
    if (x2 == 1)
      f = x1;
    else
      f = x3;

endmodule
```



Last example, behavioral code:

```
// Behavioral specification
module example5 (x1, x2, x3, f);
  input x1, x2, x3;
  output f;
  reg f;

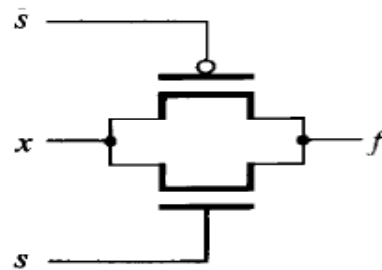
  always @(x1 or x2 or x3)
    if (x2 == 1)
      f = x1;
    else
      f = x3;
endmodule
```

Sequential

Concurrent

Transmission gate

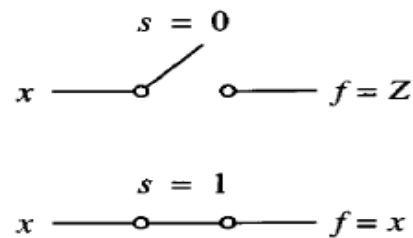
9



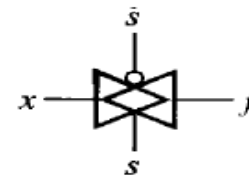
(a) Circuit

s	f
0	Z
1	x

(b) Truth table



(c) Equivalent circuit



(d) Graphical symbol

Figure 3.60 A transmission gate.

Tristate buffer

10

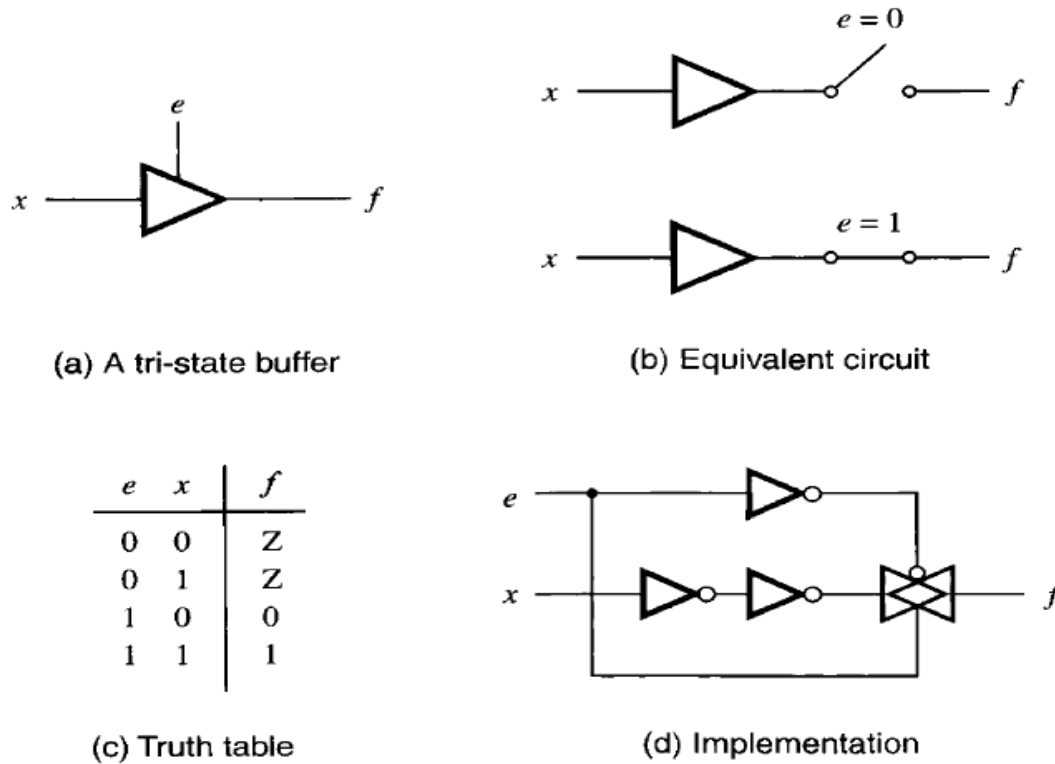


Figure 3.57 Tri-state buffer.

Tristate buffer and Transmission gate

11

- **What are the differences between them?**
 - So, we can define a Tristate buffer both behavioral and structural, Transmission gate is only defined as a structural (transistor level) code. Because we want the compiler to create a circuit which is a switch, not a buffer.
- **These two are useful in gates, e.g. XOR and MUX.**

Numbers in Verilog

12

- `<length>'<radix><digits>`

of bits

b	binary	0-1
o	octal	0-7
d	decimal	0-9
h	hexadecimal	0-F

Examples:

`12'b 1000 =`

If the system of viewing is
unsigned decimal : 8

If the system of viewing is
signed decimal (two's
complement) : -8

Vectors

13

input [3:0] w;

MSB



w[3]w[2]w[1]w[0]



LSB

wire [3:1] A, B;

wire [0:7] z;

...

Vectors

14

- Making smaller vectors out of bigger vectors:

wire [3:0] A;

A[3:2]

A[2:1]

A[1]

A[2]

A[0]

- CONCAT : { , }

MSB
↑
 $X = \{w, y\}$ w, y : maybe vectors

An example of adder in Verilog

15

- In the first view of an adder, how much it is easy to write:

```
wire [3:0] x, y, s;  
assign s = x + y;
```

But a little problem in above, what happens to the “Carry”?
So, we may have 2 solutions.

Carry problem

16

- **Solution 1**

```
wire[3:0] x, y;  
wire[4:0] s;  
assign s = x + y;
```

Then, the MSB of 's' is the “carry bit”.

Carry problem

17

- **Solution 2**

```
wire [3:0] x, y, s;
```

```
wire carryout;
```

```
assign {carryout, s} = x + y; ← Using "Concat"
```

Arithmetic circuits in Verilog

18

- **Full adder,**

Having “cin”, “x” and “y” reaching to “s”, “cout”, so the functionality is like below,

$$\{\text{cout}, \text{s}\} = \text{cin} + \text{x} + \text{y};$$

- **Half adder** is simpler than full adder because it doesn't have the “carry in” (“cin”) input. So then ,

$$\{\text{cout}, \text{s}\} = \text{x} + \text{y};$$

First Take a look at FA's structure and Boolean expressions for reviewing,

c_i	x_i	y_i	c_{i+1}	s_i
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

(a) Truth table

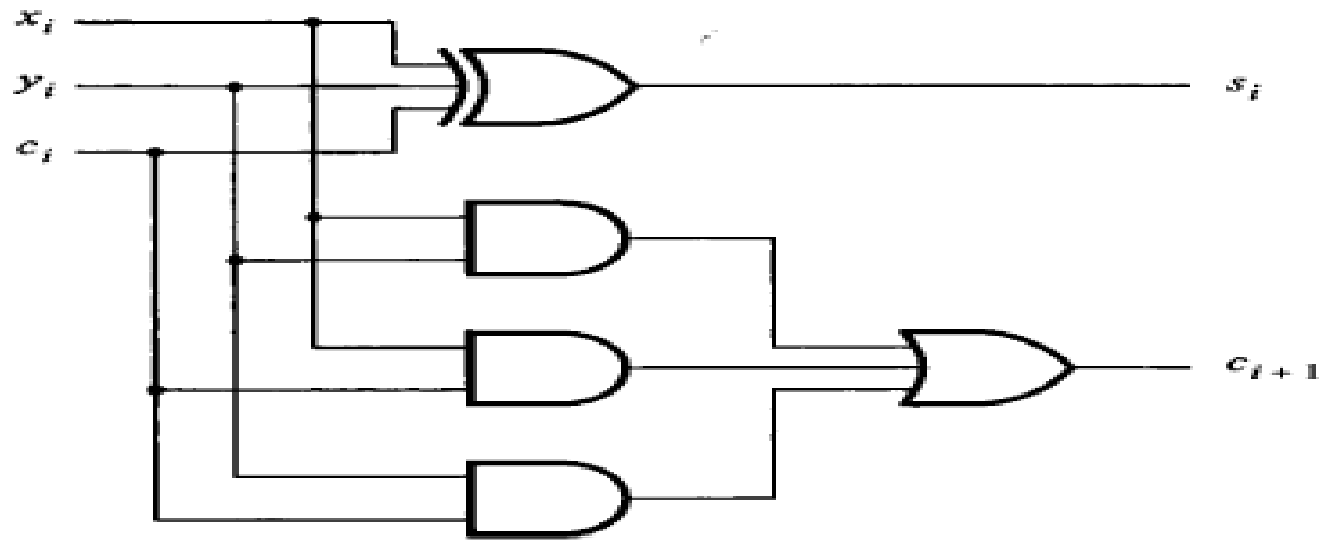
$c_i \backslash x_i y_i$	00	01	11	10
0		1		1
1	1		1	

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

$c_i \backslash x_i y_i$	00	01	11	10
0			1	
1		1	1	1

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

(b) Karnaugh maps



(c) Circuit

Figure 5.4 Full-adder.

FA

20

- Then, we have some Verilog codes:

- Structural:

...

- Behavioral 1:

```
assign s = x ^ y ^ cin;
```

```
assign cout = (x & y) | (x & cin) | (y & cin);
```

- Behavioral 2:

```
reg s, cout;
```

```
always@(*)
```

```
{cout, s} = x + y + cin;
```

And ...

FA

21

- Another representation of Simple **Full adder** formulas:

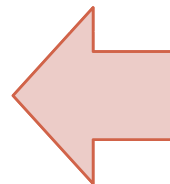
$$s = x \oplus y \oplus cin$$

$$cout = g + p.cin$$

Which:

$$g = x.y$$

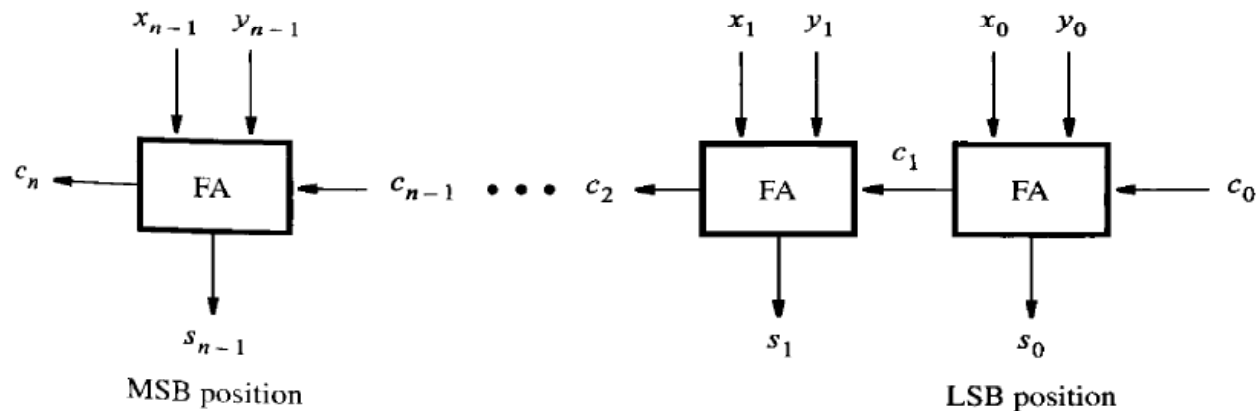
$$p = x + y$$



Ripple carry adder

22

- Gathering FAs together, for an n-bit Ripple carry adder:



Worst case delay = $n * t$
 t : delay from cin to cout of an FA

A problem of Ripple carry adder, very slow!!

Ripple carry adder

23

- Structural code by using “fulladd” modules in our code, which are written previously:

```
module adder4(cin, x, y, s, cout);  
    input cin;  
    input [3:0] x, y;  
    output [3:0] s;  
    output cout;  
    wire[3:1] c;  
  
    fulladd stage0(cin, x[0], y[0], s[0], c[1]);  
    fulladd stage0(c[1], x[1], y[1], s[1], c[2]);  
    fulladd stage0(c[2], x[2], y[2], s[2], c[3]);  
    fulladd stage0(c[3] x[3], y[3], s[3], cout);  
endmodule
```

Ripple carry adder, behavioral code, showing overflow

24

```
module addern(cin, x, y, s, cout, overflow);
```

```
    parameter n = 32;
```

```
    input [n-1:0] x, y;
```

```
    output cout, overflow;
```

```
    output [n-1:0] s;
```

```
    reg cout;
```

```
    reg [n-1:0] s;
```

```
    reg [n:0] c;
```

```
    integer k;
```

```
    always@(*)
```

```
    begin
```

```
        c[0] = cin;
```



the compiler puts "32"



Ripple carry adder, behavioral code, showing overflow



```
for (k=0; k<n; k = k + 1)
begin
    s[k] = x[k]^y[k]^c[k];
    c[k+1] = ( x[k] & y[k] ) | ( x[k] & y[k] ) | ( y[k] & c[k] );
end
cout = c[n];
end
assign overflow = c[n-1] ^ c[n];
Endmodule
```

Overflow/underflow

26

- **Arithmetic overflow/underflow :**

In the 2's complement system(that we are in) and **if n bits are used to represent signed numbers**, The result number in decimal is not in the range $-2^{(n-1)}$ to $2^{(n-1)} - 1$.

We call this Arithmetic overflow/underflow (easier to say only overflow!).

In the ripple carry adder, we have two cases for overflow:

1. **Overflow :**

Adding two positive numbers resulting in negative number.

2. **Underflow:**

Adding two negative numbers resulting in positive number.

Please pay attention using both positive and negative numbers, subtracting is a subset of adding.

Overflow

27

- By the two sections described about overflow and underflow in our adder, we define a bit named “overflow” which when it is ‘1’, it shows that we have a overflow/underflow :

$$\text{overflow} = \overline{c[n-1]} \cdot c[n] + c[n-1] \cdot \overline{c[n]} =$$

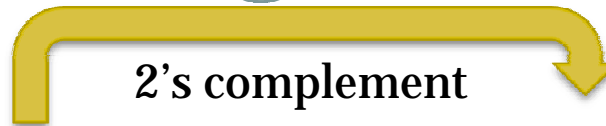
$\text{overflow} = c[n-1] \oplus c[n];$

underflow

Subtraction

28

We have,



2's complement

$$A - B = A + (-B) = A + (\sim B + 1)$$

So a subtractor, needs to add 1 to the addition of A and $\sim B$, and therefore “cin” is used by this subtractor and is not free any more.

The Next slide shows an Adder and subtractor unit.

Adder and Subtractor Unit

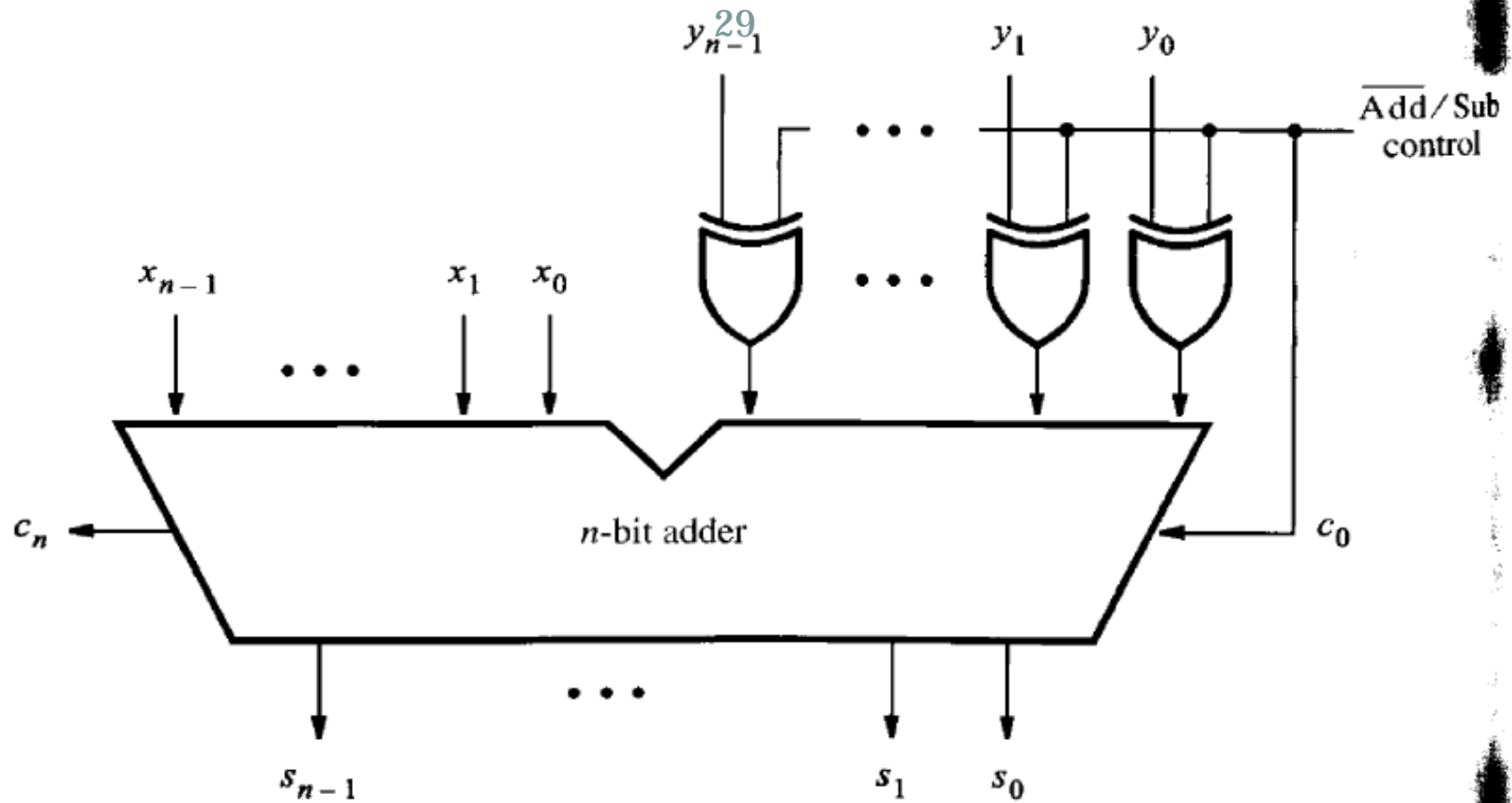


Figure 5.13 Adder/subtractor unit.