06 Arithmetic circuits:

- Full adder
- Ripple-carry adder
- Inverting numbers
- Adder/subtractor circuit
- Multiplier
- Verilog for adders
    - signal concatenation
    - vectored signals

Full addition:

Example:

	1 1 1 0		c_i
X	0 1 1 1 1	$= 15_{10}$	x_i
Y	+ 0 1 0 1 0	$= 10_{10}$	$+y_i$
S	0 1 1 0 0 1	$= 25_{10}$	$c_{i+1}s_i$

Full addition:

Example:

Generated Carries

		1	1	1	0		
X		0	1	1	1	1	$= 15_{10}$
Y	+	0	1	0	1	0	$= 10_{10}$
S		0	1	1	0	0	1 $= 25_{10}$

$$\begin{array}{r} c_i \\ x_i \\ \underline{+y_i} \\ c_{i+1}s_i \end{array}$$

Half adder is ok for the least significant bit (LSB), but we need 3 inputs for other bits.

3 inputs are x and y at that bit, plus c from the "previous" bit.

Full addition (cont):

List all possible input combinations:

Inputs			Outputs	
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

Truth table

Full addition (cont):

[M]

K-map for s_i

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

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

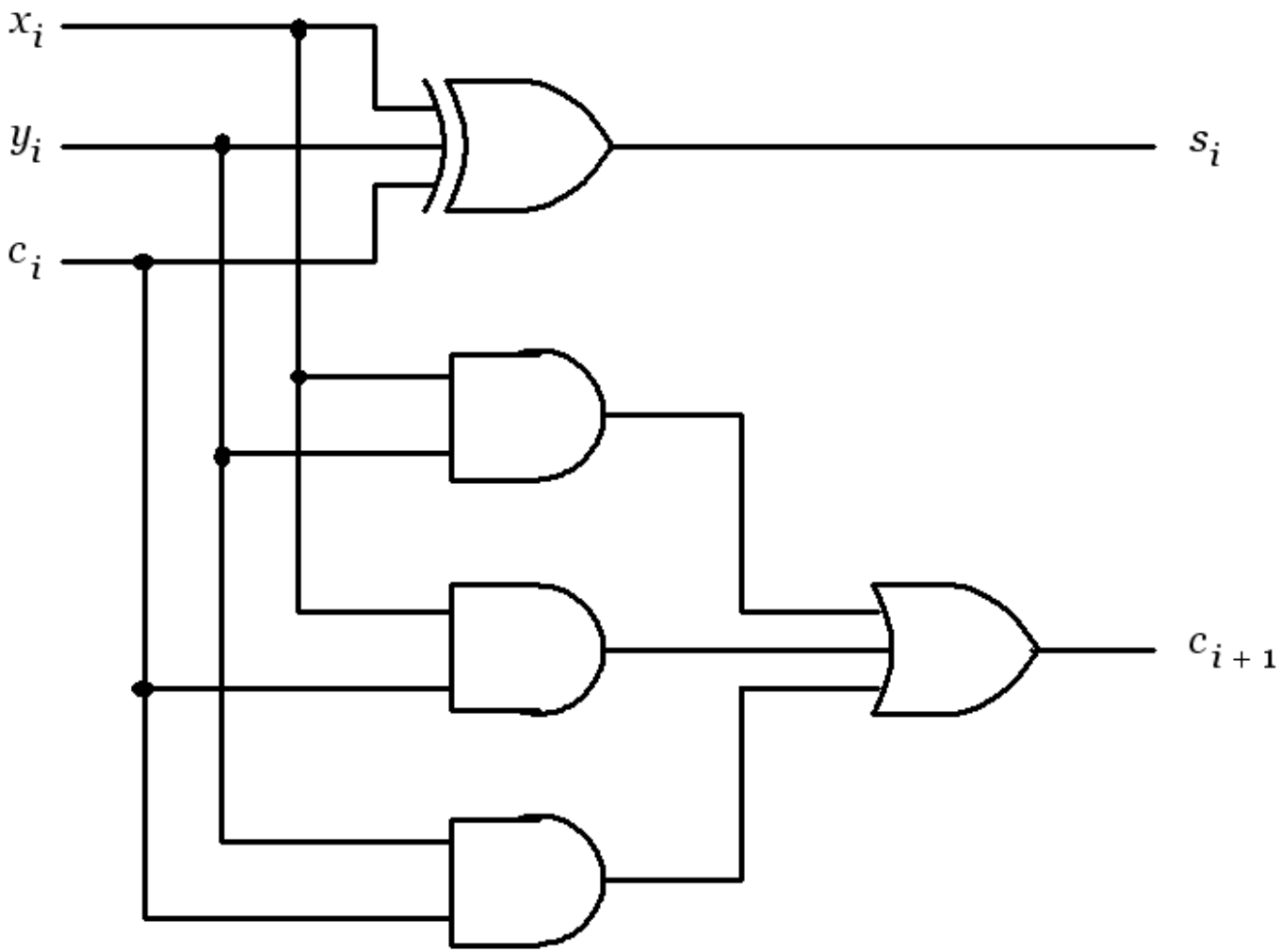
K-map for c_{i+1}

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

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

[BV]

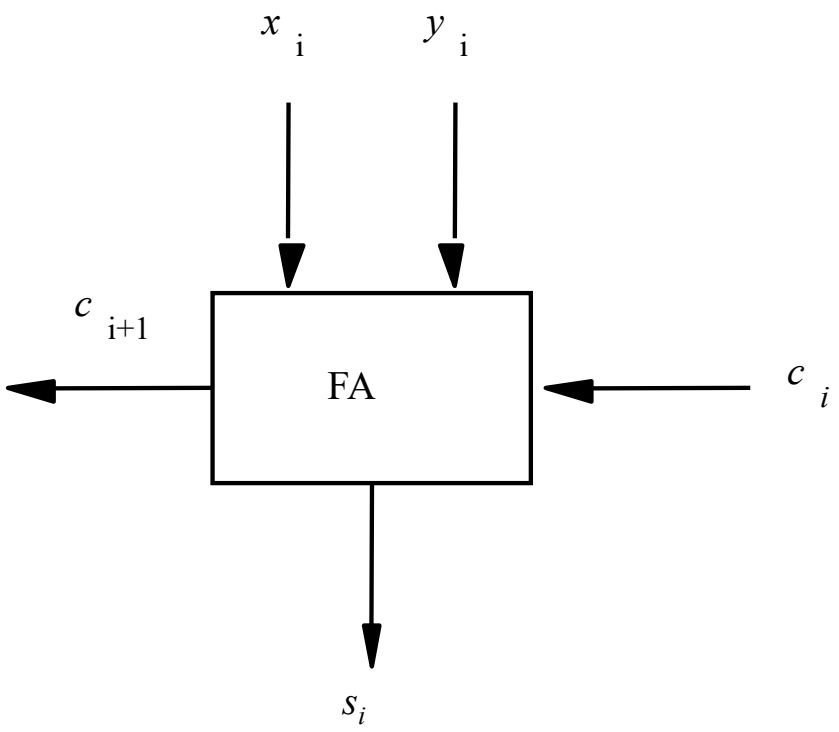
Full addition (cont):



Circuit

[BV]

Full adder symbol:



[BV]

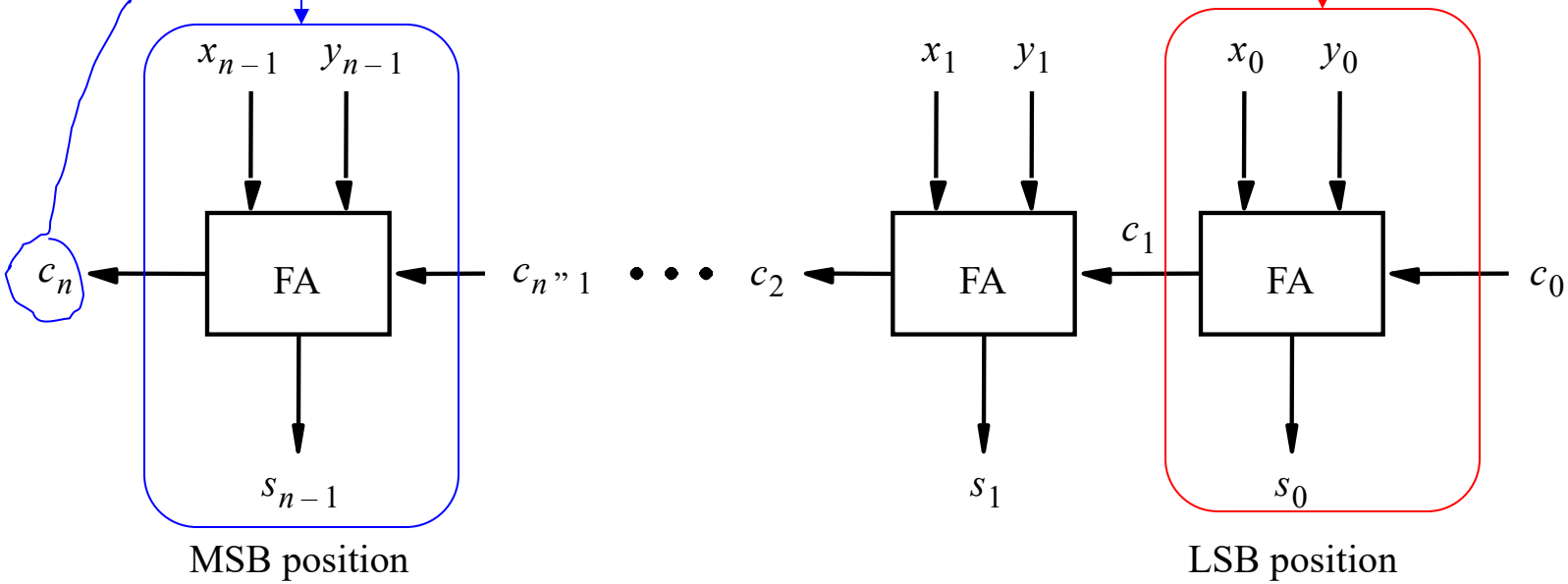
Ripple-carry adder:

IDEA: Add two *n*-bit numbers (n can be anything) from *right* to *left*, just like humans, using *n* **full adders**.

Possible carry can affect the addition result.

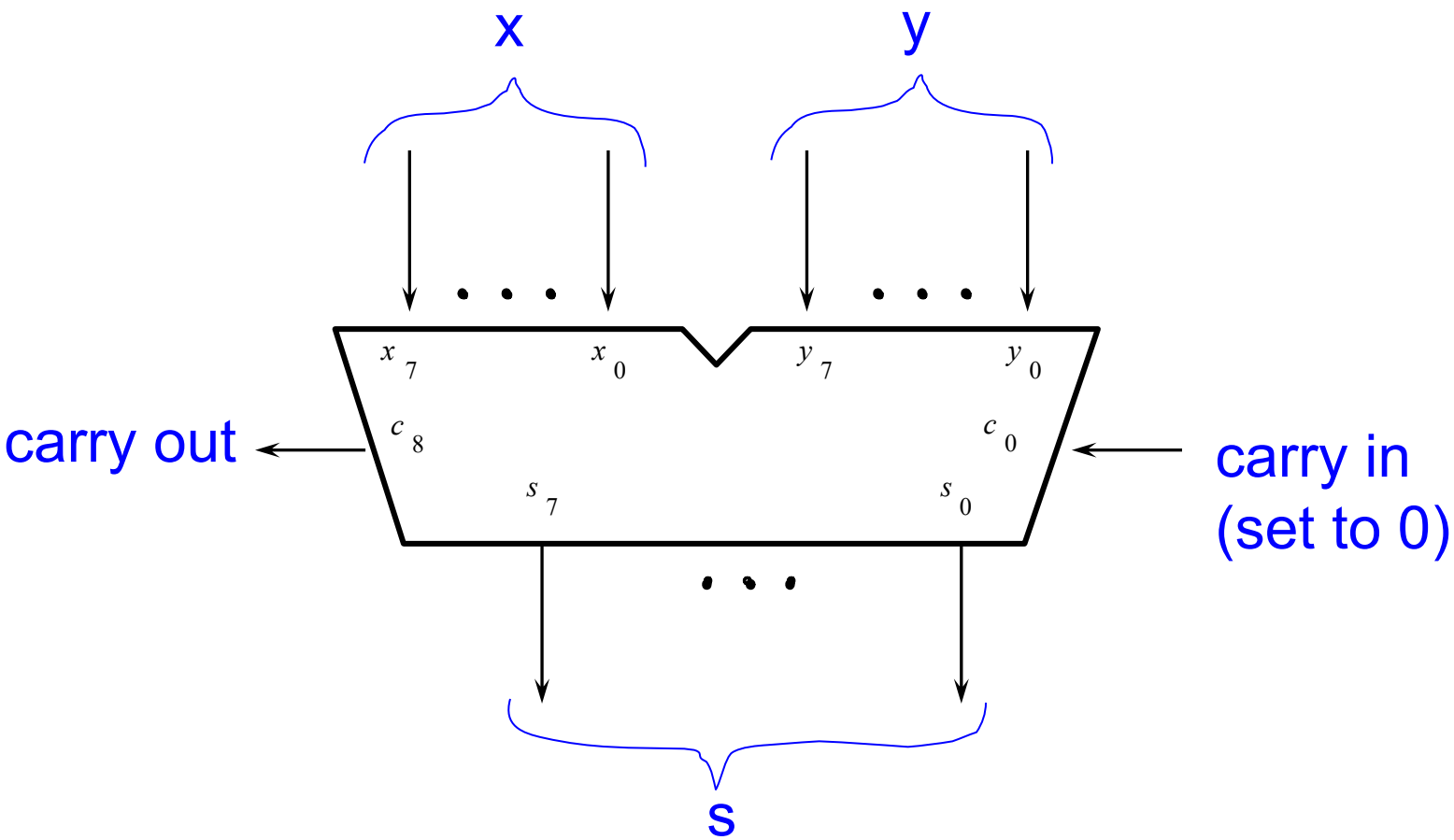
	1 1 1 0		c_i
X	0 1 1 1 1	$= 15_{10}$	x_i
Y	+ 0 1 0 1 0	$= 10_{10}$	$+y_i$
S	0 1 1 0 0 1	$= 25_{10}$	$c_{i+1}s_i$

		1	1	1	0			c_i
X		0	1	1	1	1	$= 15_{10}$	x_i
Y	+	0	1	0	1	0	$= 10_{10}$	$+y_i$
S	0	1	1	0	0	1	$= 25_{10}$	$c_{i+1}s_i$



[BV]

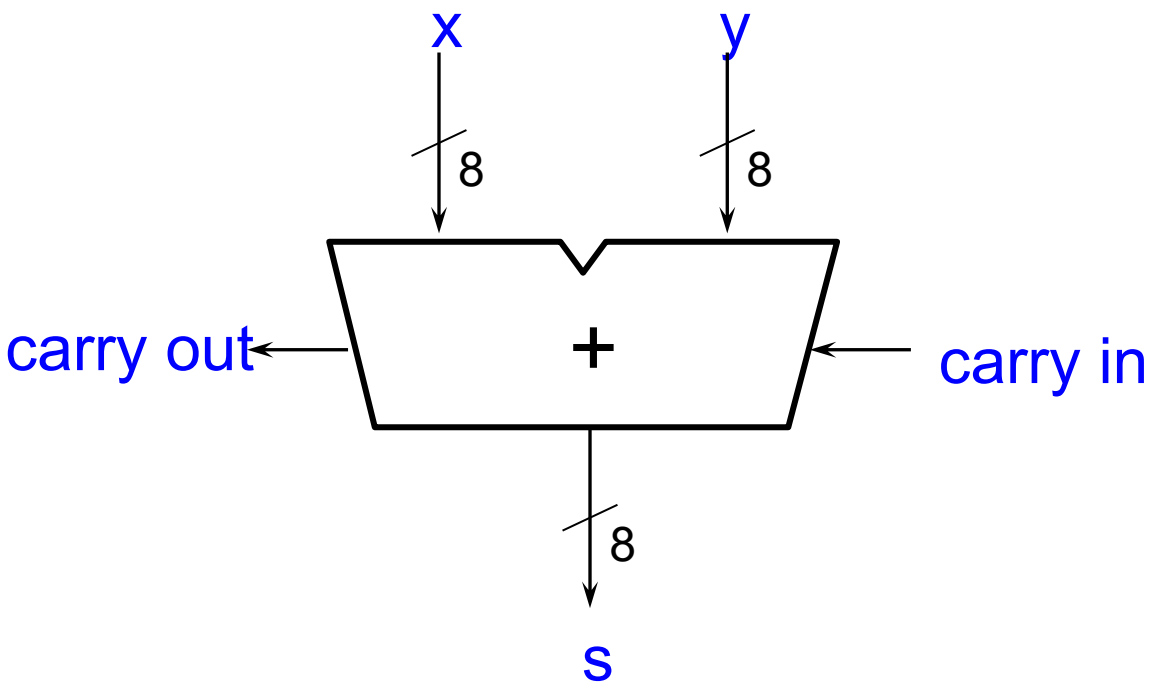
Adder symbol:



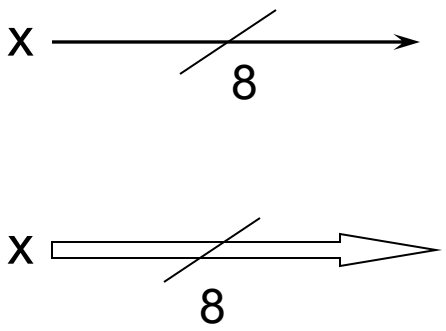
8-bit adder: x and y are 8-bit numbers

[BV]

Alternate adder symbol:

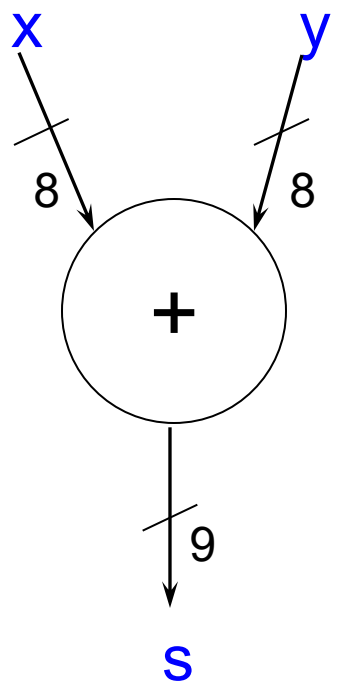


[BV]



Bus: Draw x7 … x0 together into a bus. A bus is a collection of related signals. Drawn by thick or thin lines. The number indicates number of signals.

Alternate adder symbol:



Inverting numbers:

Converting a positive number into a negative number, or vice versa.

We will focus on 2's complement.
Recap from last lecture set below:

Positive/Negative 2's comp. conversions

Invert every bit then add 1 to LSB

001_100 ← This is +12

110_011 (inversion of 001_100)
+ 1
110_100 ← This is -12

Using an adder to invert a number:

We want to find –A from A:

- Use a full adder
- Let one input (say x) be 0.
- Invert every bit of input y
- Set carry in to 1 (This adds 1 to ~y)

Positive/Negative 2's comp. conversions

Invert every bit then add 1 to LSB

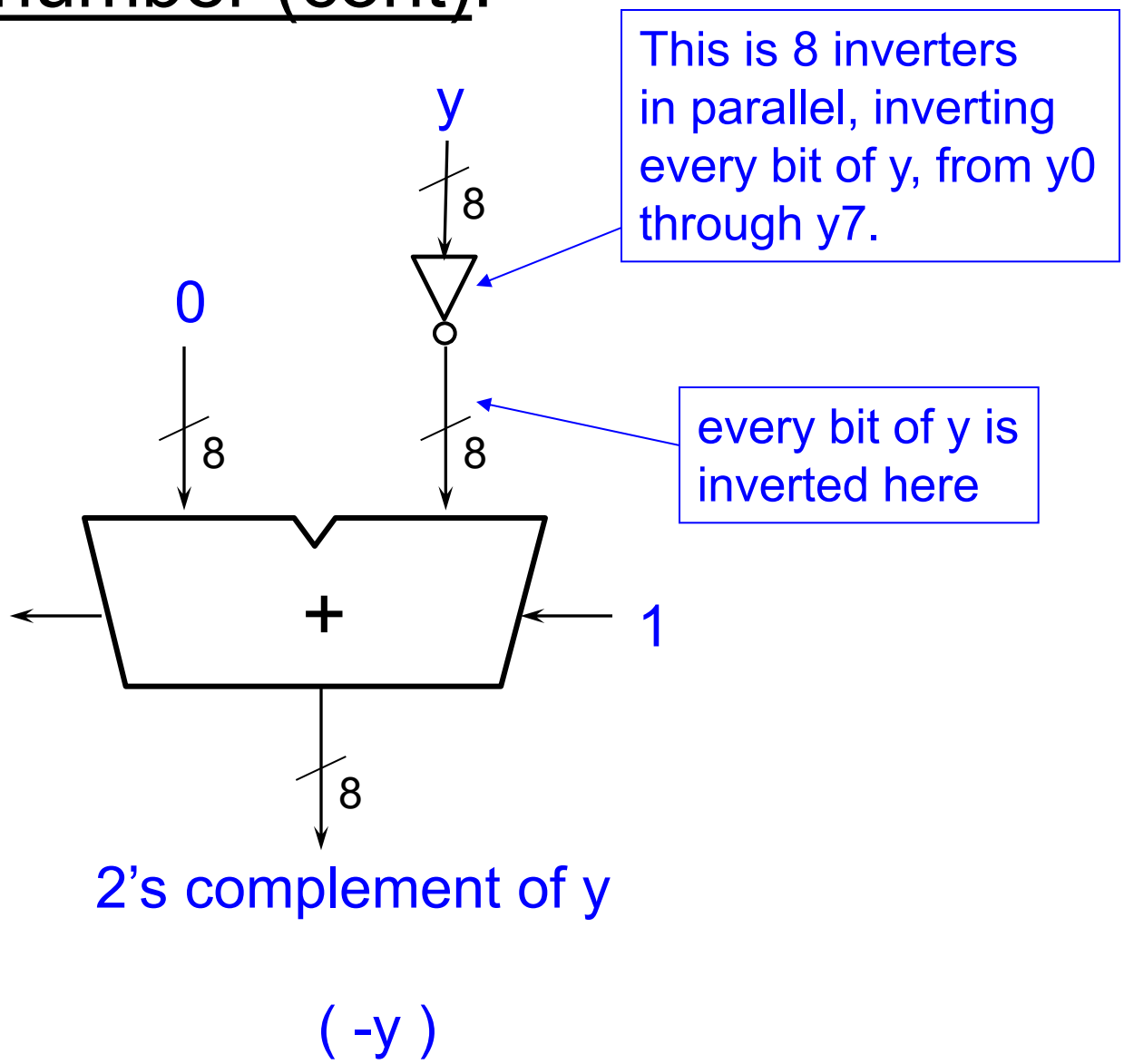
001_100 ← This is +12

110_011 (inversion of 001_100)

\+ 1

110_100 ← This is -12

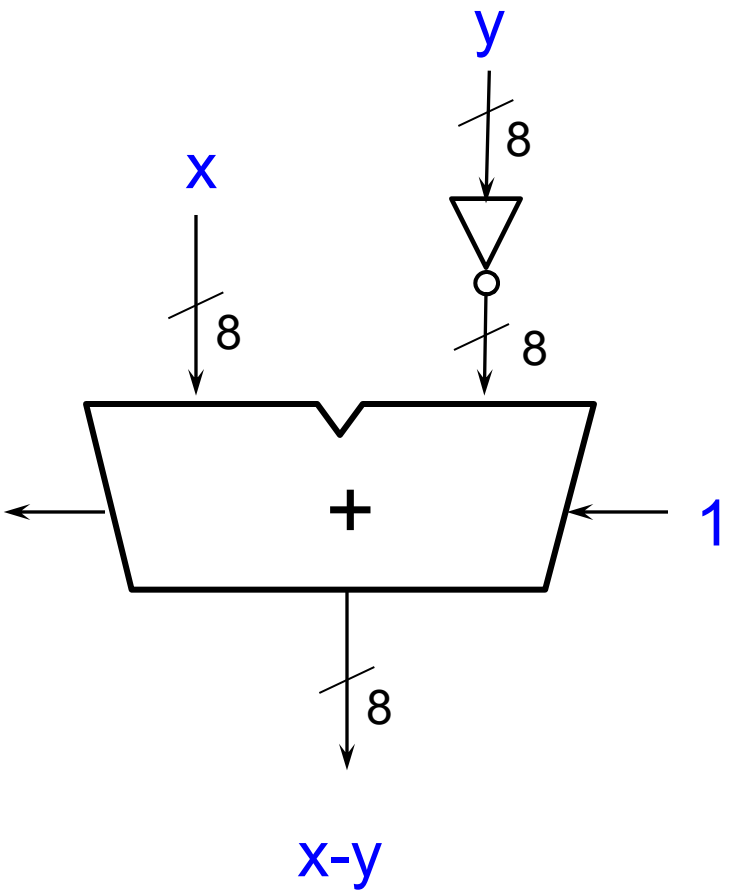
Inverting a number (cont):



[BV/K]

Subtraction:

- We can get –y from y using adder
- Putting in x input will give us (x – y)

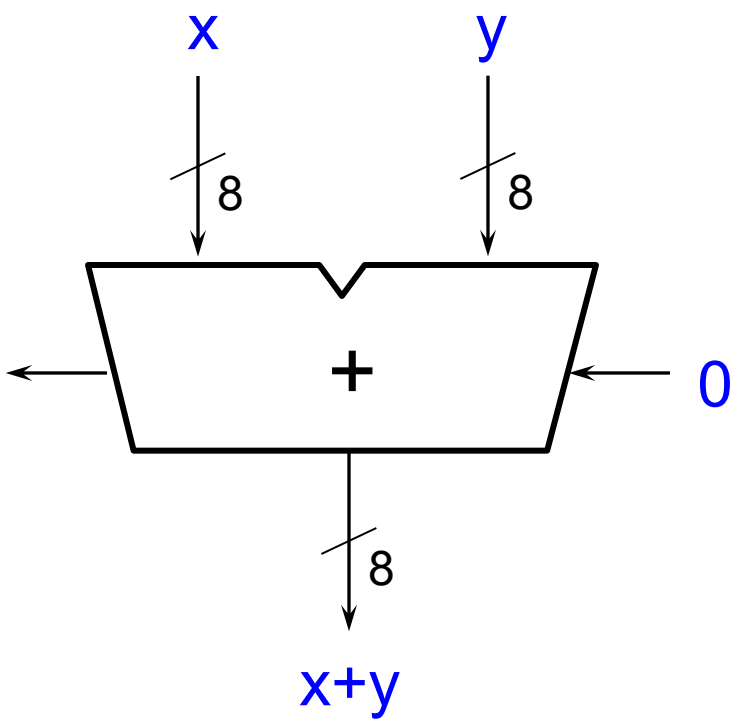


[BV/K]

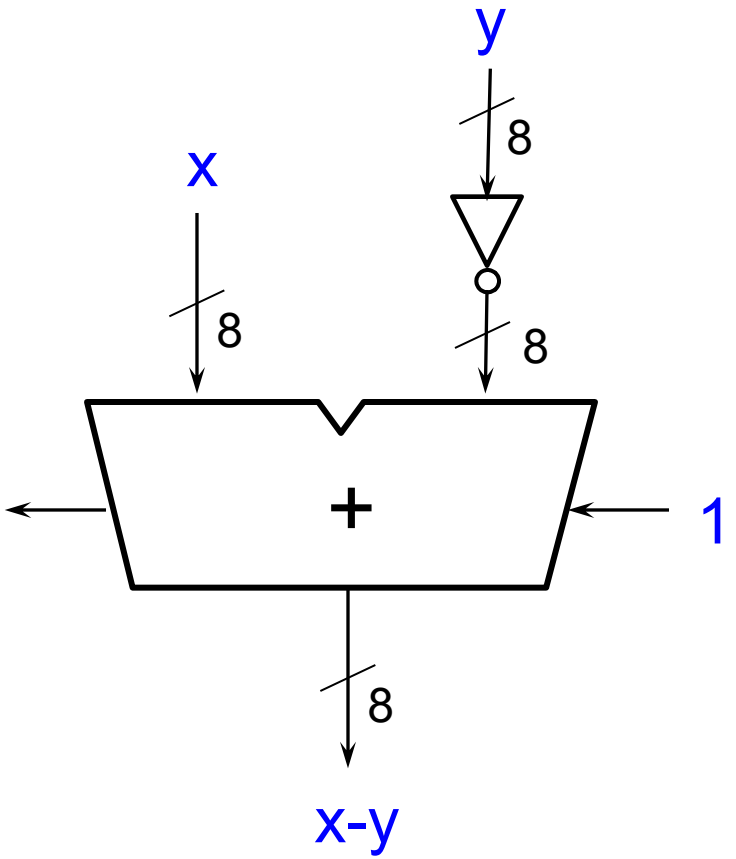
Adder/subtractor:

Is there a way to combine them?

Adder



Subtractor



[BV/K]

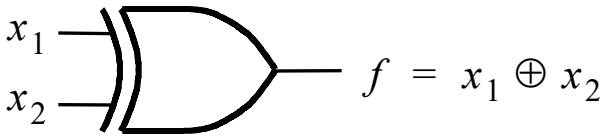
Adder/subtractor (cont):

- Remember XOR?
- x2 ^ 0 is x2
- x2 ^ 1 is ~x2

One input of XOR can be used to control bit inversion of the other input.

x_1	x_2	$f = x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Truth table

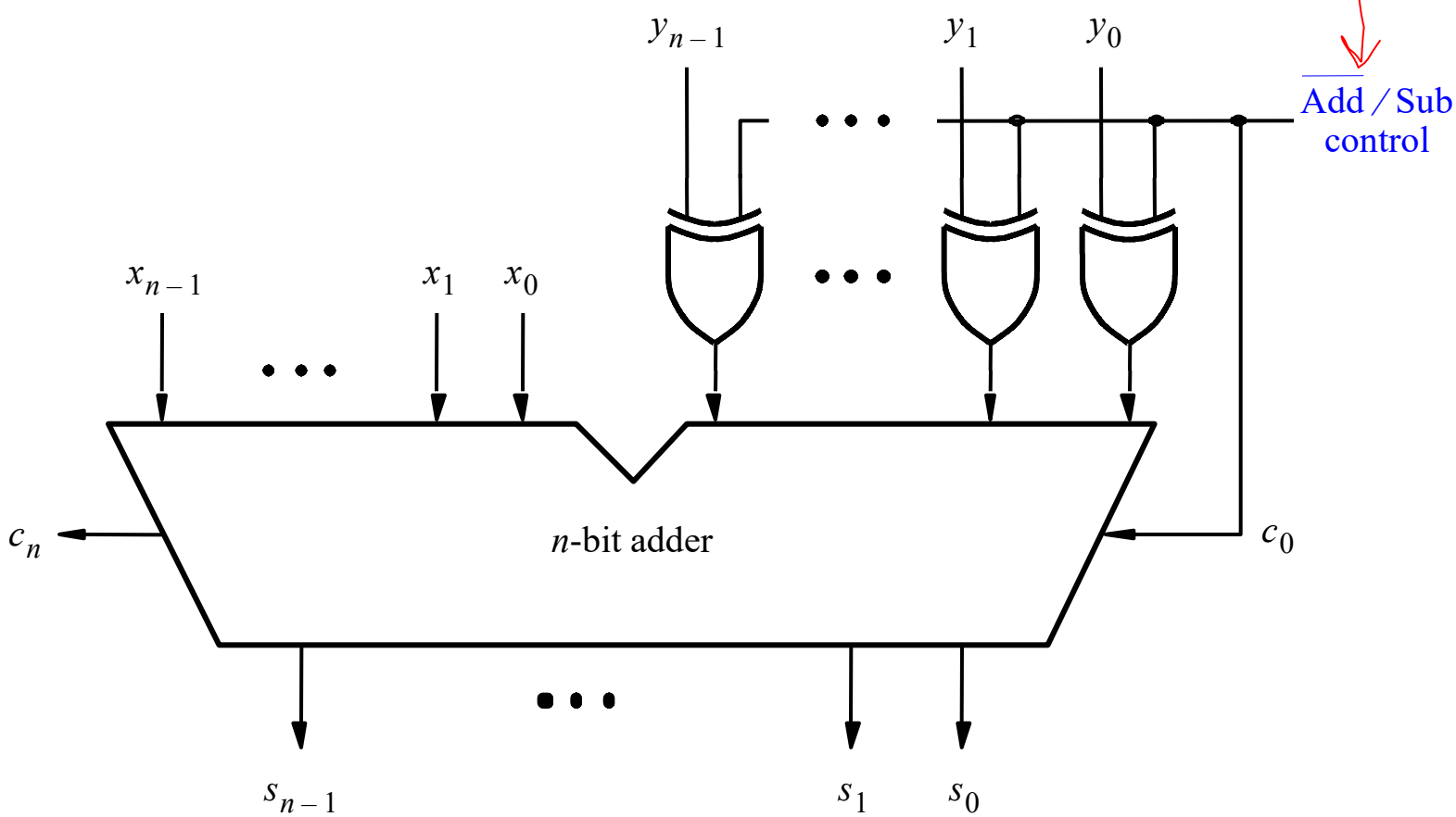


Graphical symbol

[BV]

Adder/subtractor (cont):

explain this notation!



when $\overline{\text{add}}$/sub is 0, $y_i \wedge 0 = y_i$, s is x+y

when $\overline{\text{add}}$/sub is 1, $y_i \wedge 1 = \overline{y}_i$, s is x-y

[BV]

Multiplication:

Multiplication in base 2 is easy

x multiply 0 is 0, and x multiply 1 is x

Circuit also resembles human calculation.

Multiplicand M	(14)	1 1 1 0
Multiplier Q	(11)	1 0 1 1
		1 1 1 0
		1 1 1 0
		0 0 0 0
		1 1 1 0
Product P	(154)	1 0 0 1 1 0 1 0

[BV]

Signal concatenation: (concatenate = join)

full addition:	examples		
c_{in}	0	1	0
x	1	1	0
+y	+1	+1	+1
c_{out} s	10	11	01

We use {a, b} to *concatenate* signals a, b.

So,

$\{c_{out}, s\} = c_{in} + x + y;$

+ in Verilog is arithmetic addition

Behavioral model for full adder:

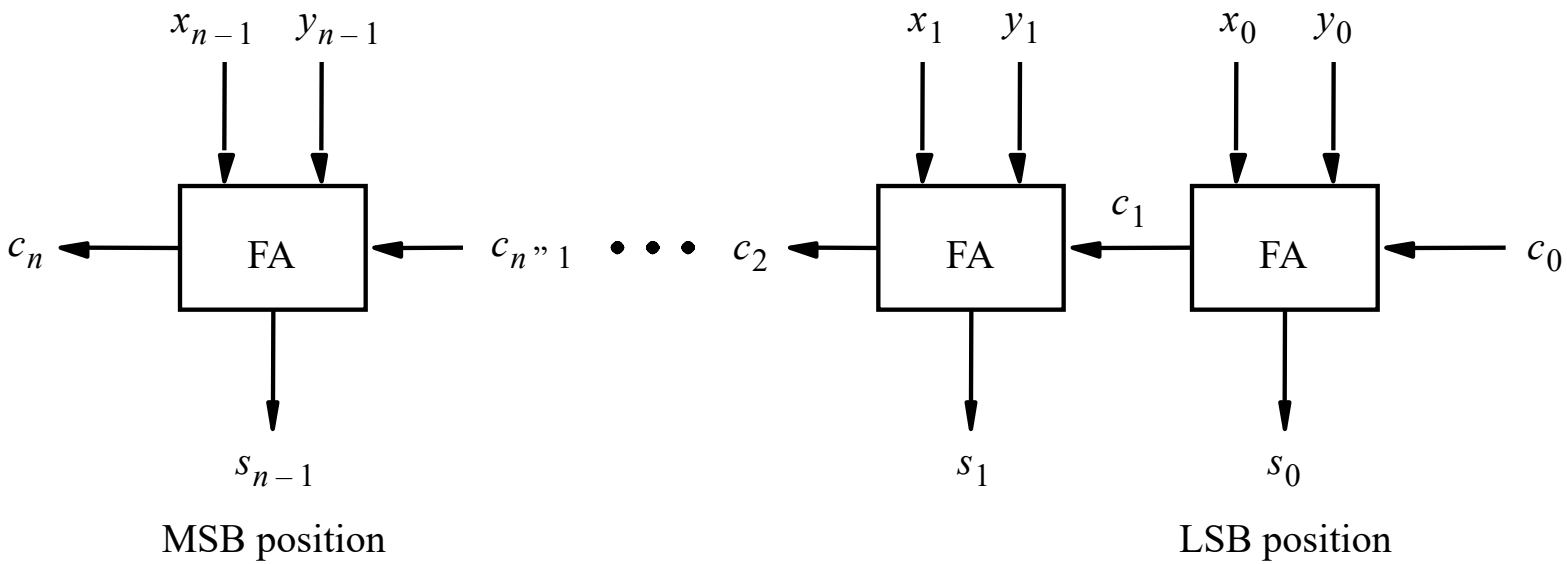
```
module fulladd (cin, x, y, s, cout);
    input cin, x, y;
    output s, cout;
    reg s, cout;

    always @(x or y or cin)
      {cout, s} = x + y + cin;

endmodule
```

s and Cout are assigned inside the **always** block, so they must be declared as *variables* by the word **reg**.

We want to write this kind of code, and not using Verilog as a schematic description!

[BV]

Multi-bit adder:



schematic

[BV]

Module instantiations for 4-bit adder:

```
module adder4 (carryin, x3, x2, x1, x0, y3, y2, y1, y0, s3, s2, s1, s0, carryout);
    input carryin, x3, x2, x1, x0, y3, y2, y1, y0;
    output s3, s2, s1, s0, carryout;

    fulladd stage0 (.cin(carryin), .x(x0), .y(y0), .s(s0),  .cout(c1));
    fulladd stage1 (.cin(c1), .x(x1), .y(y1), .s(s1), .cout(c2));
    fulladd stage2 (.cin(c2), .x(x2), .y(y2), .s(s2), .cout(c3));
    fulladd stage3 (.cin(c3), .x(x3), .y(y3), .s(s3), .cout(carryout));

endmodule
```

```
module fulladd (cin, x, y, s, cout);
    input cin, x, y;
    output s, cout;

    assign s = x ^ y ^ cin;
    assign cout = (x & y) | (x & cin) | (y & cin);

endmodule
```

adder4 module *instantiates* fulladd module 4 times.

the name after fulladd (stage0, stage1, …, stage3) is called the *instance name*

[BV/K]

Verilog for 4-bit adder (cont):

```
module adder4 (carryin, x3, x2, x1, x0, y3, y2, y1, y0, s3, s2, s1, s0, carryout);
    input carryin, x3, x2, x1, x0, y3, y2, y1, y0;
    output s3, s2, s1, s0, carryout;

    fulladd stage0 (.cin(carryin), .x(x0), .y(y0), .s(s0),  .Cout(c1));
    fulladd stage1 (.cin(c1), .x(x1), .y(y1), .s(s1), .Cout(c2));
    fulladd stage2 (.cin(c2), .x(x2), .y(y2), .s(s2), .Cout(c3));
    fulladd stage3 (.cin(c3), .x(x3), .y(y3), .s(s3), .Cout(carryout));

endmodule

module fulladd (cin, x, y, s, cout);
    input cin, x, y;
    output s, cout;

    assign s = x ^ y ^ cin;
    assign cout = (x & y) | (x & cin) | (y & cin);

endmodule
```

The names in .name(--) MUST match module's port names

This is called *module instantiation by port name* → Use this method.

[BV/K]

Vectored signals:

What if we wanted to do a 32-bit adder?

- Naming x31 … x0, y31 … y0, s31 … s0 can be very tedious.
- We use Verilog's multi-bit signals, called *vectors* in declaring signals

input [31:0] x;

x is a *vector*, x[31] is MSB and x[0] is LSB.

output [31:0] s;

s is a *vector*, s[31] is MSB and s[0] is LSB.

Behavioral model for multi-bit adder:

Why stop at one-bit full adder?

Extend inputs x, y and output s to multi-bit

```
module fulladd32 (Cin, x, y, s, Cout);
    input cin;
    input  [31:0] x, y;
    output [31:0] s;
    output cout;
    reg [31:0] s;
    reg cout;

    always @(x or y or cin)
      {cout, s} = x + y + Cin;

endmodule
```

output port size and **reg** size must match

when ANY bit of x or y or Cin changes, the **always** block will be executed

[BV]

Parameterized module:

The code from previous page works well with any # of bits.

```
module fulladd (Cin, x, y, s, Cout, overflow);
    parameter n = 32;
    input cin;
    input  [n-1:0] x, y;
    output [n-1:0] s;
    output cout, overflow;
    reg [n-1:0] s;
    reg cout, overflow;

    always @(x or y or cin)
      begin
        {cout, s} = x + y + cin;
        overflow = cout ^ x[n-1] ^ y[n-1] ^ s[n-1];
      end

endmodule
```

If we want to build an n-bit adder, we can modify the previous code to do so.

parameter keyword indicates that this Verilog module can change according to module's parameter values. **32** is the default value for n

[BV]