Arithmetic Logic Unit (ALU)

Introduction to Computer

Yung-Yu Chuang

with slides by Sedgewick & Wayne (introcs.cs.princeton.edu*), Nisan & Schocken (*www.nand2tetris.org*) and Harris & Harris (DDCA)*

Let's Make an Adder Circuit

Goal. x + y = z for 4-bit integers.

- We build 4-bit adder: 9 inputs, 4 outputs.
- Same idea scales to 128-bit adder.
- Key computer component.

$$\begin{array}{ccccc}
 & 1 & 1 & 1 & 0 \\
 & 2 & 4 & 8 & 7 \\
+ & 3 & 5 & 7 & 9 \\
\hline
 & 6 & 0 & 6 & 6
\end{array}$$

Binary addition

Assuming a 4-bit system:

$$\begin{array}{ccccc}
\underline{0} & \underline{0} & \underline{0} & \underline{1} & \\
 & 1 & 0 & 0 & 1 \\
 & 0 & 1 & 0 & 1 \\
\hline
0 & 1 & 1 & 1 & 0
\end{array} +
\qquad\qquad
\begin{array}{ccccc}
\underline{1} & \underline{1} & \underline{1} & \underline{1} & \\
 & 1 & 0 & 1 & 1 \\
 & 0 & 1 & 1 & 1 \\
\hline
1 & 0 & 0 & 1 & 0
\end{array} +$$

no overflow

overflow

- Algorithm: exactly the same as in decimal addition
- Overflow (MSB carry) has to be dealt with.

Representing negative numbers (4-bit system)

0	0000		
1	0001	1111	-1
2	0010	1110	-2
3	0011	1101	-3
4	0100	1100	-4
5	0101	1011	-5
6	0110	1010	-6
7	0111	1001	-7
		1000	-8

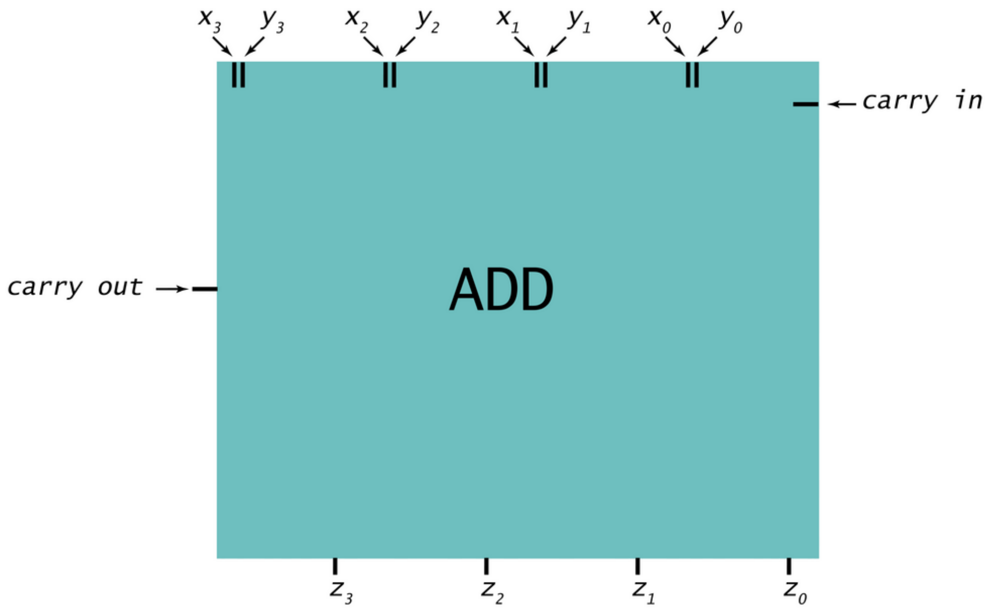
- The codes of all positive numbers begin with a "0"
- The codes of all negative numbers begin with a "1"
- To convert a number: leave all trailing 0's and first 1 intact, and flip all the remaining bits

Example: 2 - 5 = 2 + (-5) =

$$\begin{array}{r} 0\,0\,1\,0 \\ +\,1\,0\,1\,1 \\ \hline 1\,1\,0\,1 \end{array} \quad = \quad -3$$

Let's Make an Adder Circuit

Step 1. Represent input and output in binary.

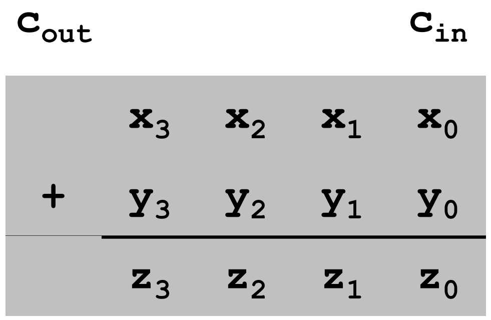


	1	1	0	0
	0	0	1	0
+	0	1	1	1
	1	0	0	1

	x_3	x_2	x_1	x_0
+	y_3	y_2	y_1	y_0
	z_3	z_2	z_1	z_0

Let's Make an Adder Circuit

Goal. x + y = z for 4-bit integers.



Step 2. [first attempt]
Build truth table.

4-Bit Adder Truth Table

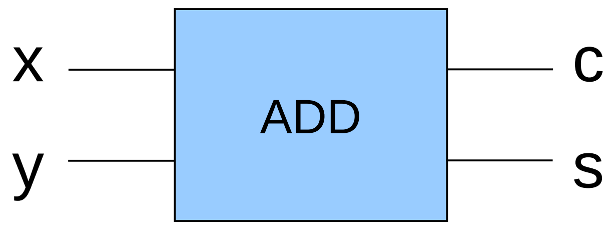
c_0	x_3	x_2	x_1	x_0	y_3	y_2	y_1	y_0	z_3	z_2	z_1	z_0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	0	0	0	1	1	0	0	1	1
0	0	0	0	0	0	1	0	0	0	1	0	0
.	.	.	.	.	.	.	.	.	.	.	.	.
1	1	1	1	1	1	1	1	1	1	1	1	1

2^{8+1} = 512 rows!

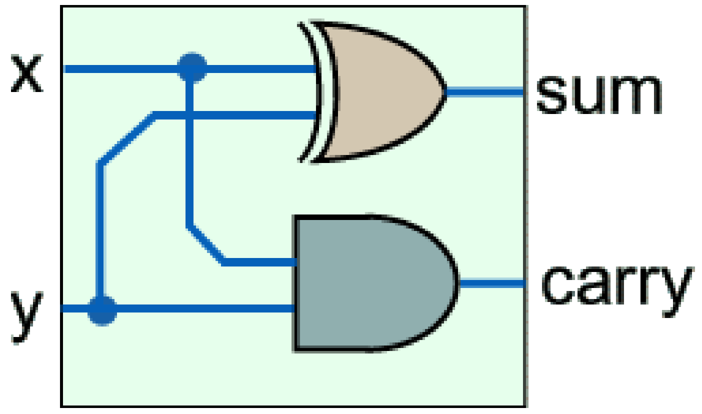
Q. Why is this a bad idea?
A. 128-bit adder: 2^{256+1} rows >> # electrons in universe!

1-bit half adder

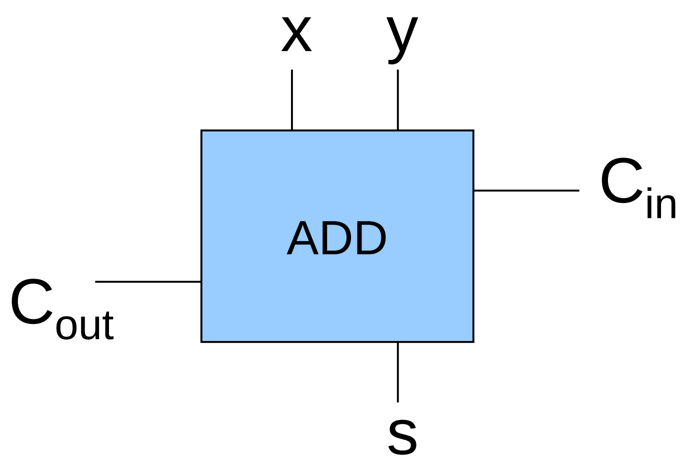
We add numbers one bit at a time.

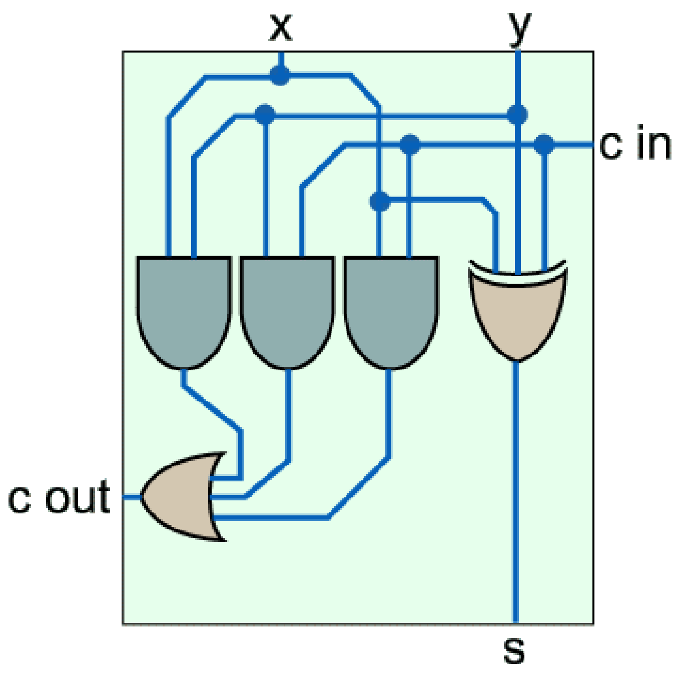


x	y	s	c



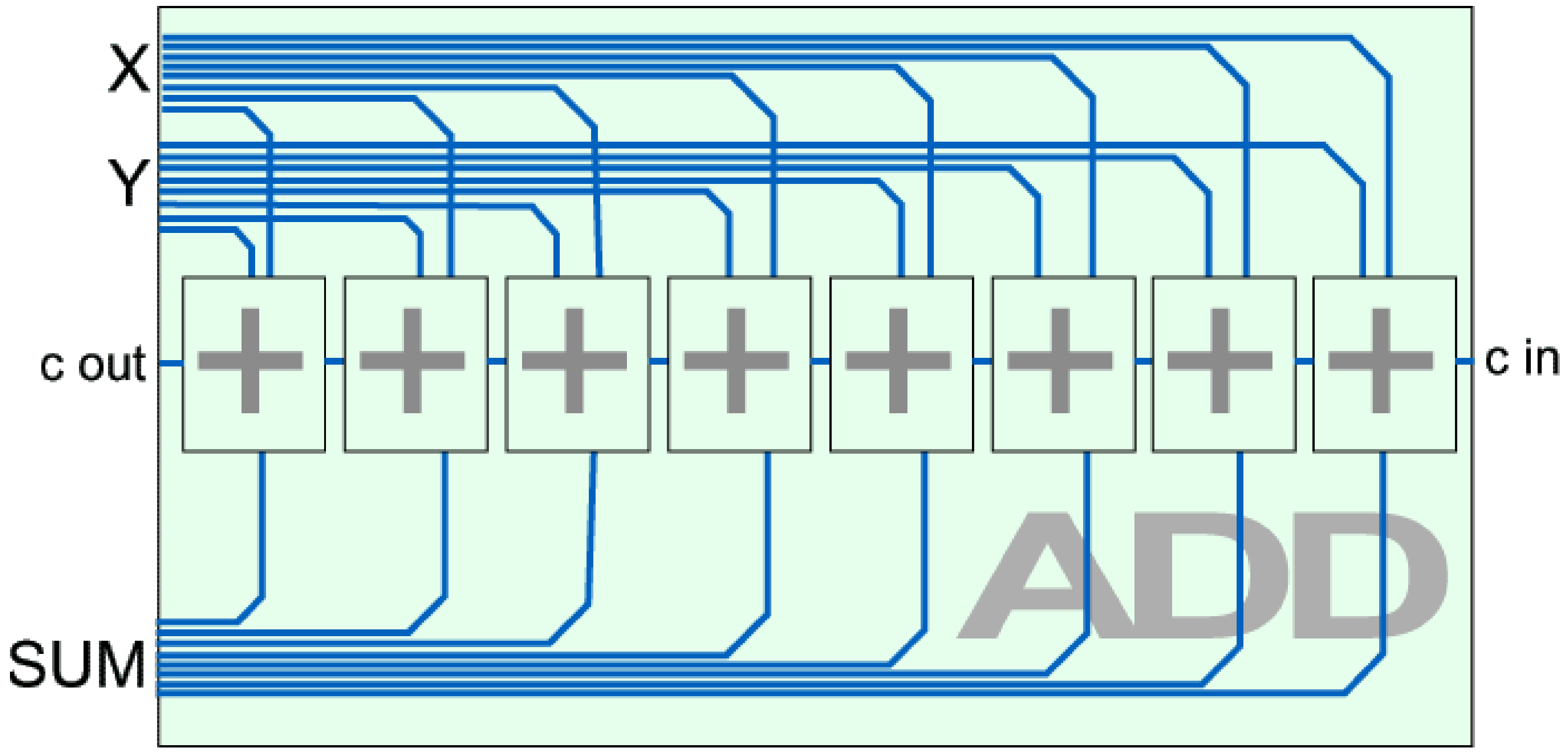
1-bit full adder





x	y	C_{in}	C_{out}	s

8-bit adder



Let's Make an Adder Circuit

Goal. x + y = z for 4-bit integers.

$$\begin{array}{ccccc}
c_{out} & c_3 & c_2 & c_1 & c_0 = 0 \\
 & x_3 & x_2 & x_1 & x_0 \\
+ & y_3 & y_2 & y_1 & y_0 \\
\hline
 & z_3 & z_2 & z_1 & z_0
\end{array}$$

Step 2. [do one bit at a time]

Build truth table for carry bit.

Build truth table for summand bit.

Carry Bit

x_i	y_i	c_i	c_{i+1}
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Summand Bit

x_i	y_i	c_i	z_i
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Let's Make an Adder Circuit

Goal. $x + y = z$ for 4-bit integers.

Step 3.

Derive (simplified) Boolean expression.

Carry Bit

x_i	y_i	c_i	c_{i+1}	MAJ
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Summand Bit

x_i	y_i	c_i	z_i	ODD
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

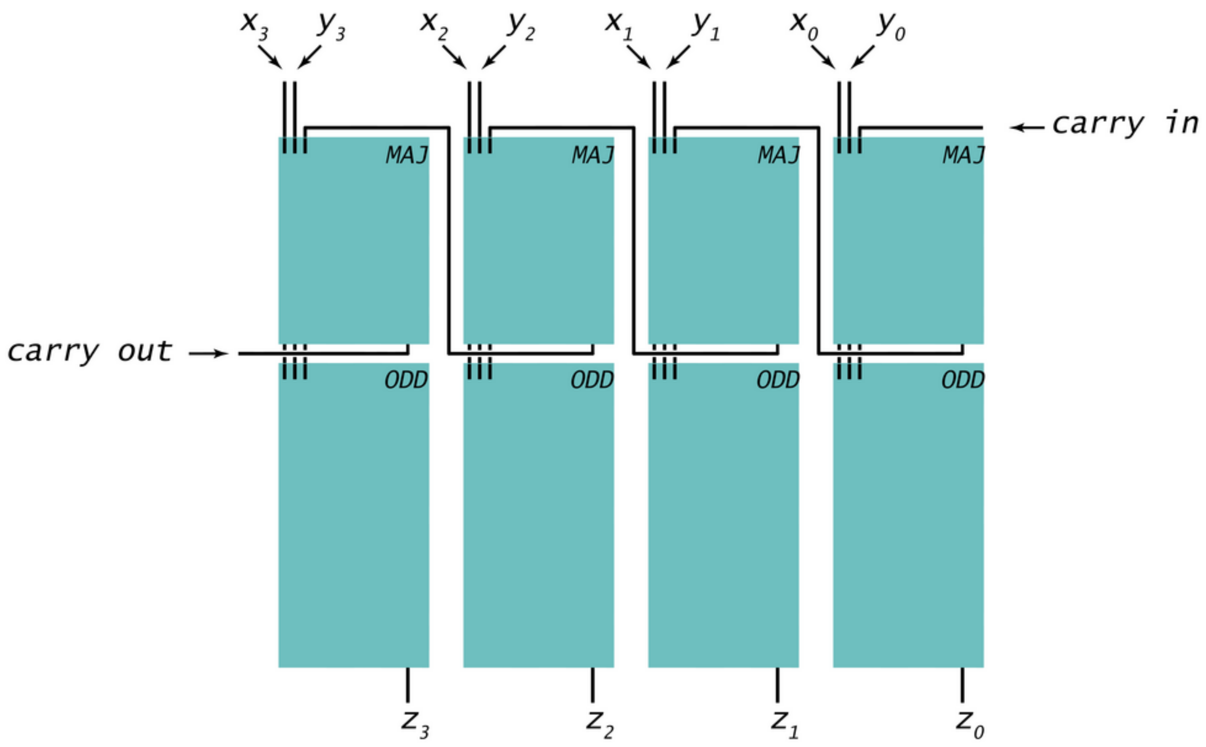
Let's Make an Adder Circuit

Goal. $x + y = z$ for 4-bit integers.

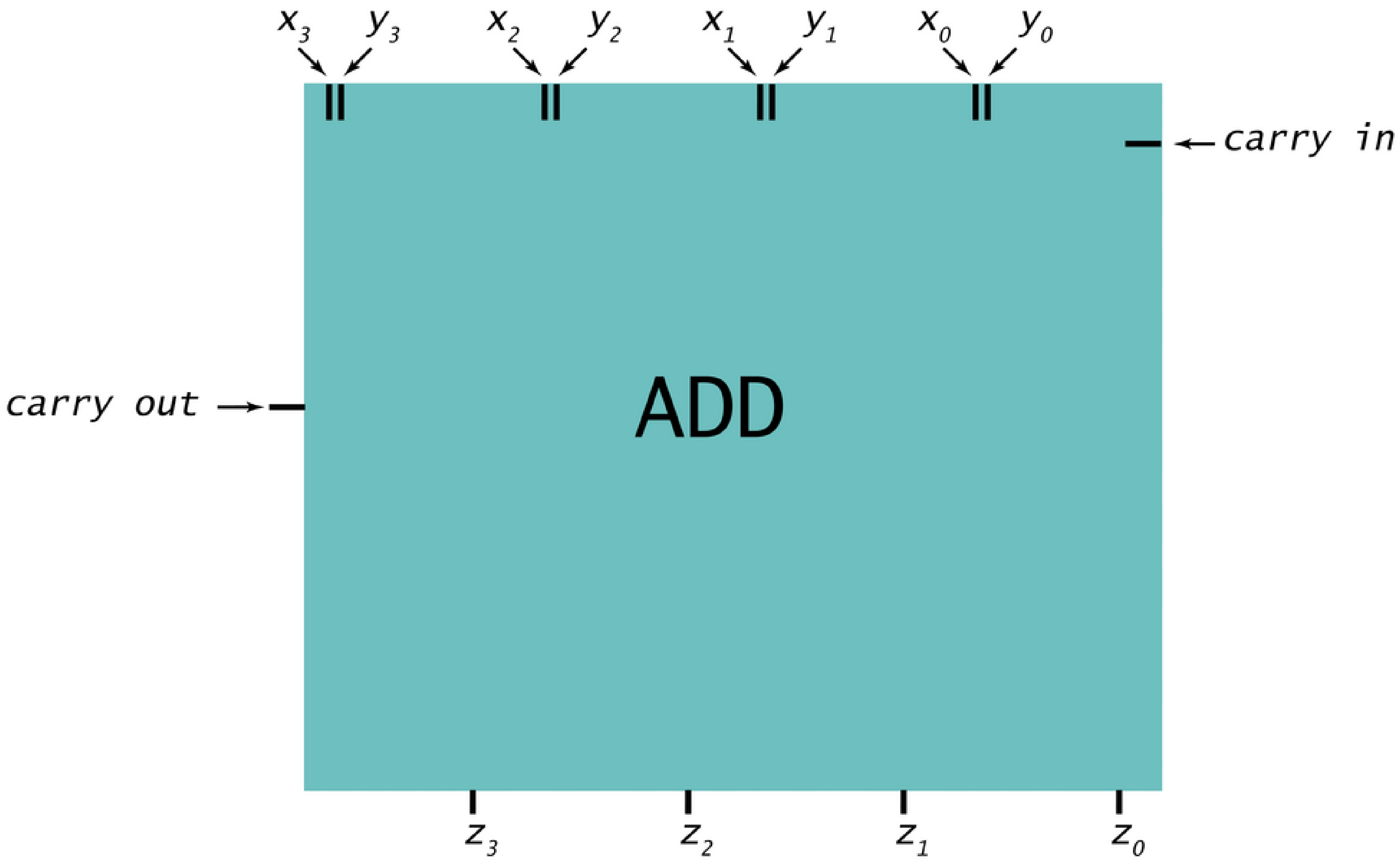
Step 4.

Transform Boolean expression into circuit.

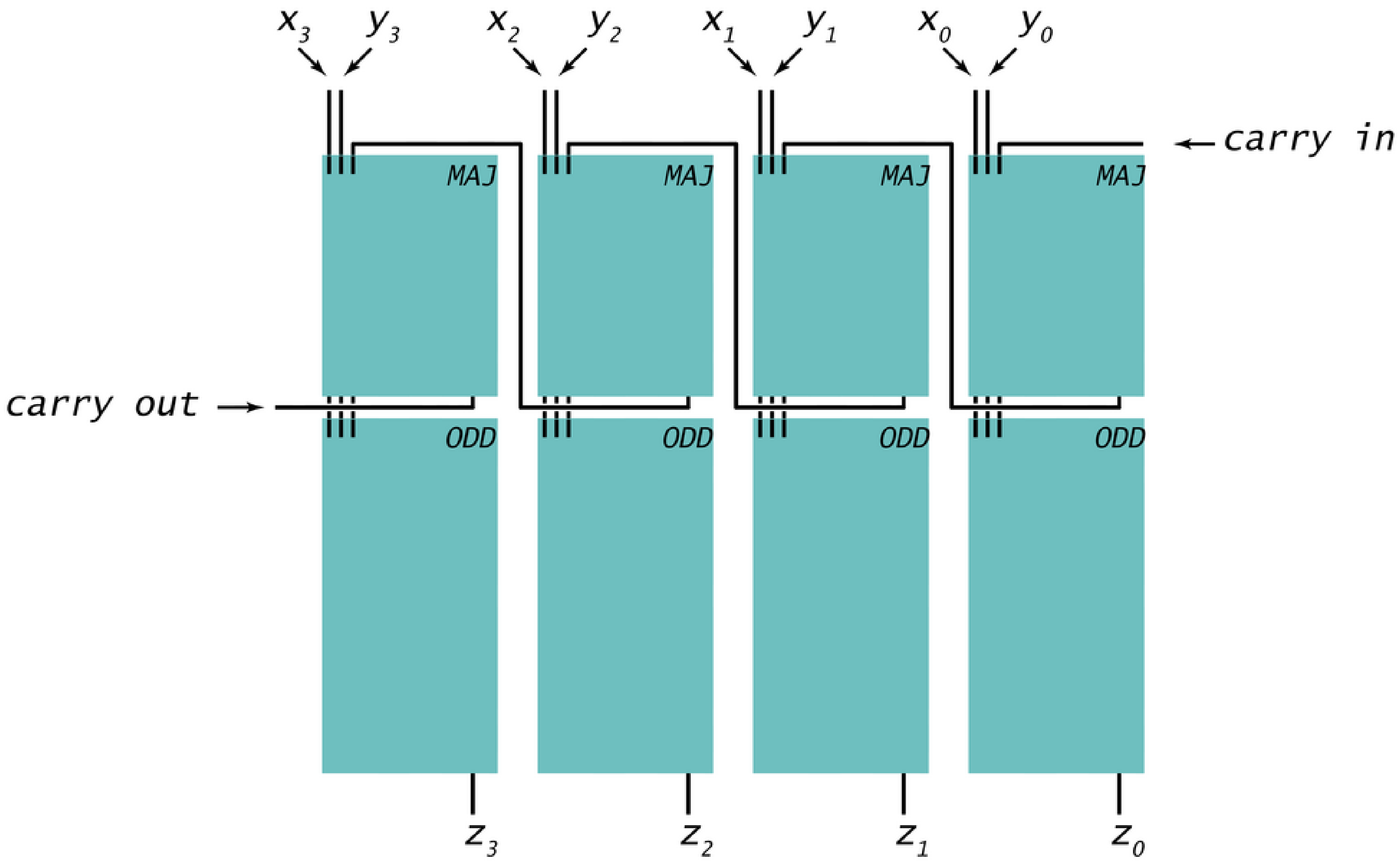
Chain together 1-bit adders.



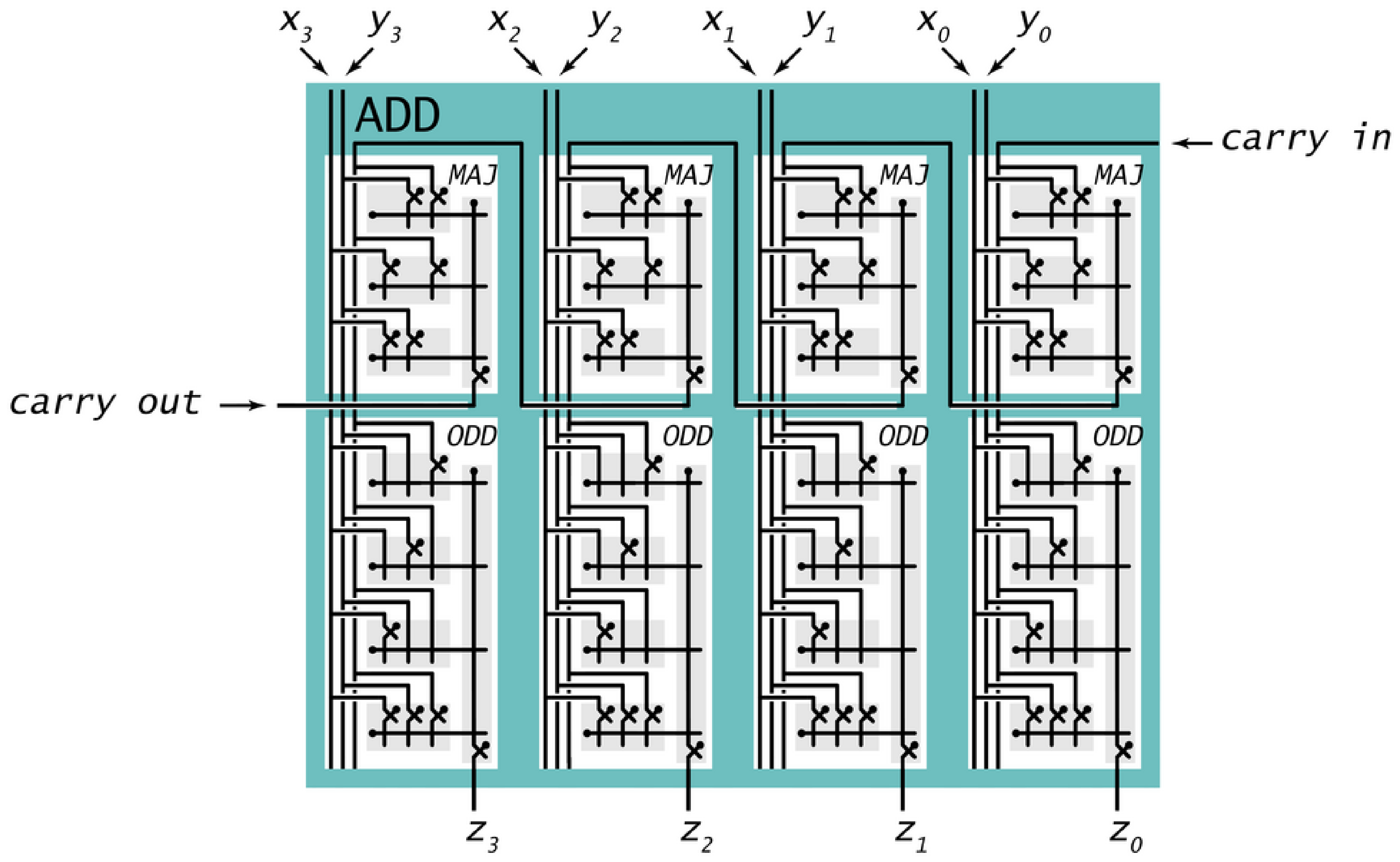
Adder: Interface



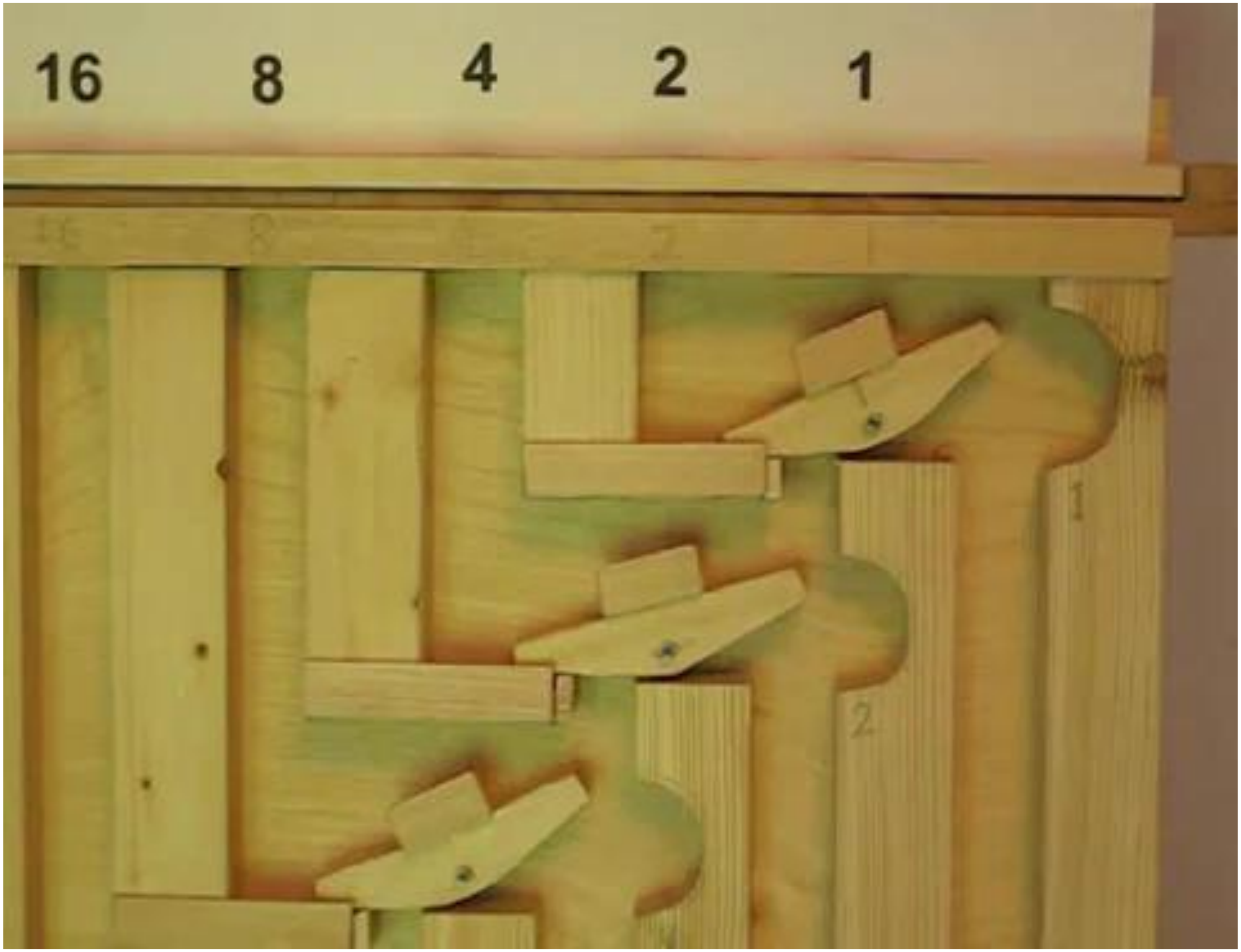
Adder: Component Level View



Adder: Switch Level View



Marble adding machine



Subtractor

Subtractor circuit: $z = x - y$.

- One approach: design like adder circuit

Subtractor

Subtractor circuit: $z = x - y$.

One approach: design like adder circuit

Better idea: reuse adder circuit

- 2's complement: to negate an integer, flip bits, then add 1

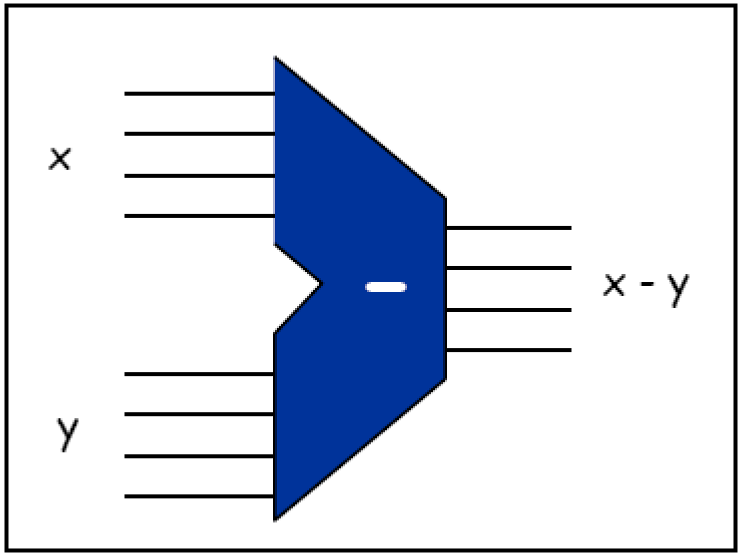
Subtractor

Subtractor circuit: $z = x - y$.

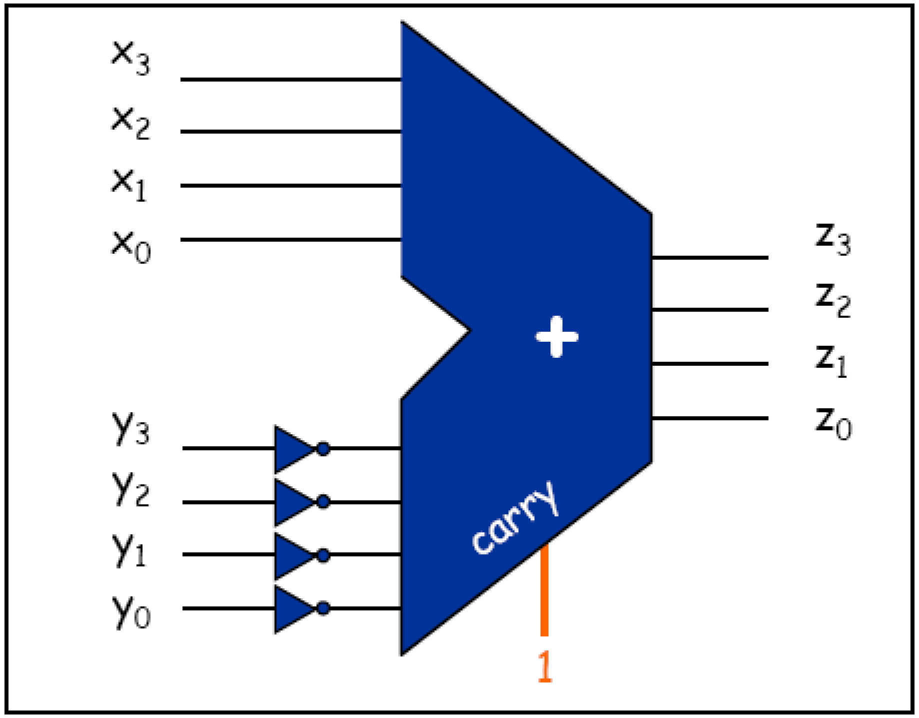
One approach: design like adder circuit

Better idea: reuse adder circuit

- 2's complement: to negate an integer, flip bits, then add 1



4-Bit Subtractor Interface

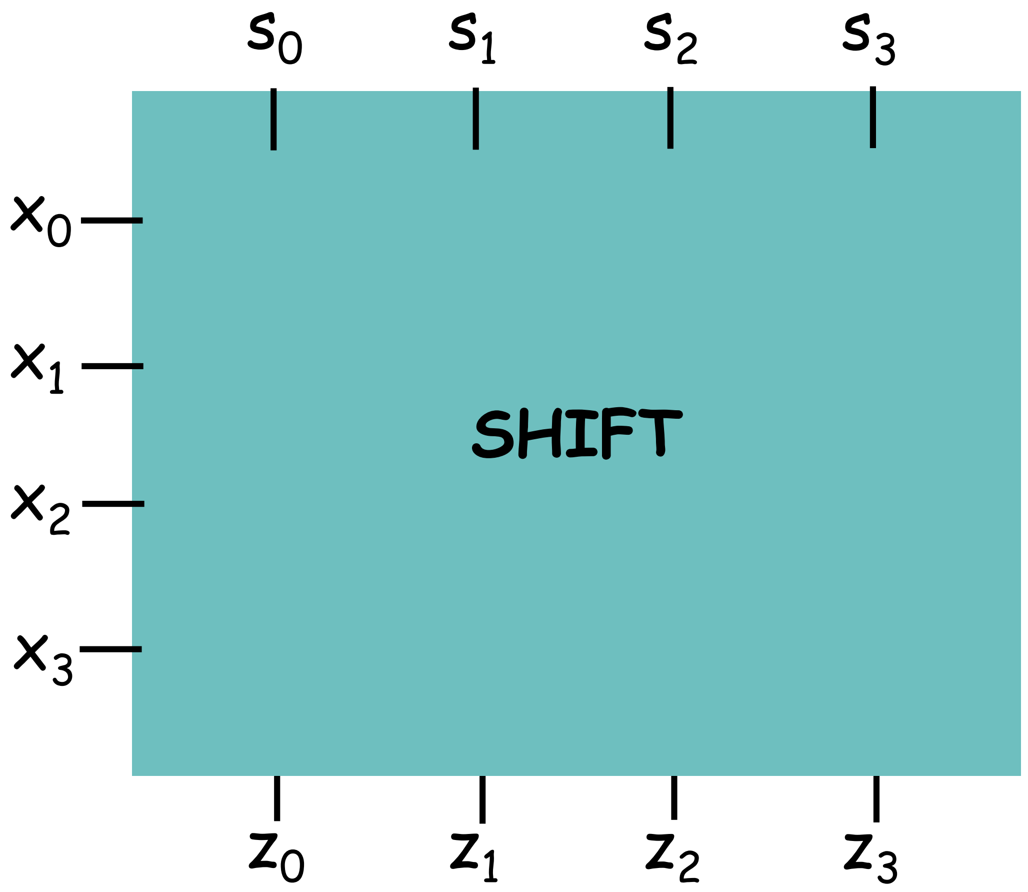


4-Bit Subtractor Implementation

Put adder and subtractor together

Shifter

Only one of them will be on at a time.



4-bit Shifter

Shifter

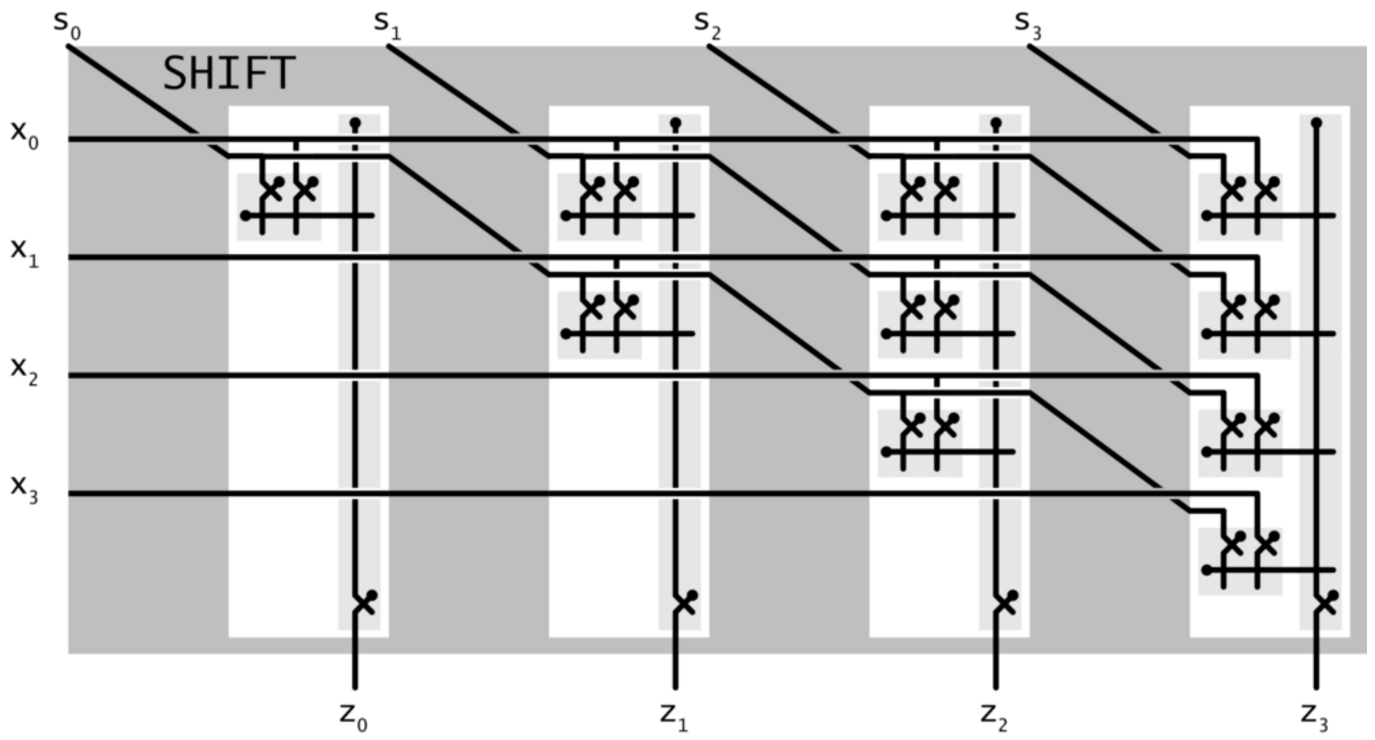
	z_3	z_2	z_1	z_0
s_0				
s_1				
s_2				
s_3				

Shifter

	z_3	z_2	z_1	z_0
s_0	x_3	x_2	x_1	x_0
s_1	x_2	x_1	x_0	0
s_2	x_1	x_0	0	0
s_3	x_0	0	0	0

$$z_0 = s_0 \cdot x_0 + s_1 \cdot 0 + s_2 \cdot 0 + s_3 \cdot 0$$
$$z_1 = s_0 \cdot x_1 + s_1 \cdot x_0 + s_2 \cdot 0 + s_3 \cdot 0$$
$$z_2 = s_0 \cdot x_2 + s_1 \cdot x_1 + s_2 \cdot x_0 + s_3 \cdot 0$$
$$z_3 = s_0 \cdot x_3 + s_1 \cdot x_2 + s_2 \cdot x_1 + s_3 \cdot x_0$$

Shifter

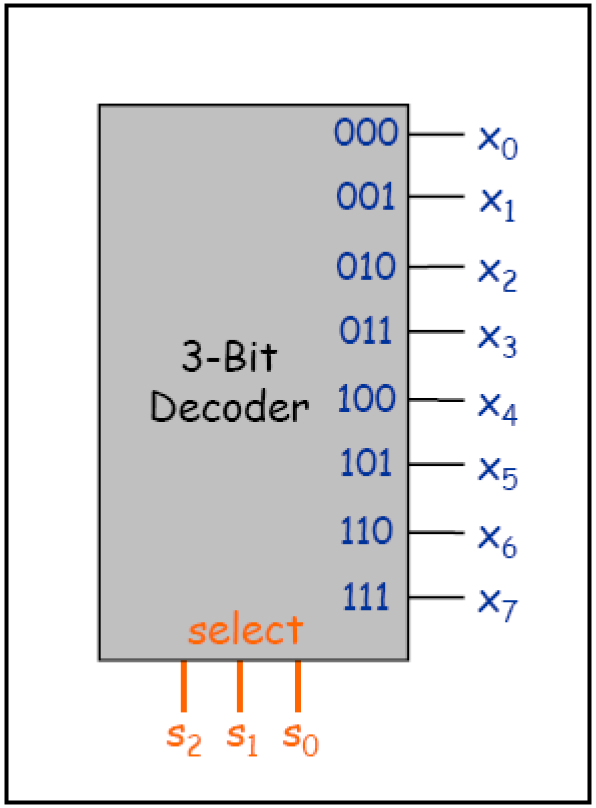


$$z_0 = s_0 \cdot x_0 + s_1 \cdot 0 + s_2 \cdot 0 + s_3 \cdot 0$$
$$z_1 = s_0 \cdot x_1 + s_1 \cdot x_0 + s_2 \cdot 0 + s_3 \cdot 0$$
$$z_2 = s_0 \cdot x_2 + s_1 \cdot x_1 + s_2 \cdot x_0 + s_3 \cdot 0$$
$$z_3 = s_0 \cdot x_3 + s_1 \cdot x_2 + s_2 \cdot x_1 + s_3 \cdot x_0$$

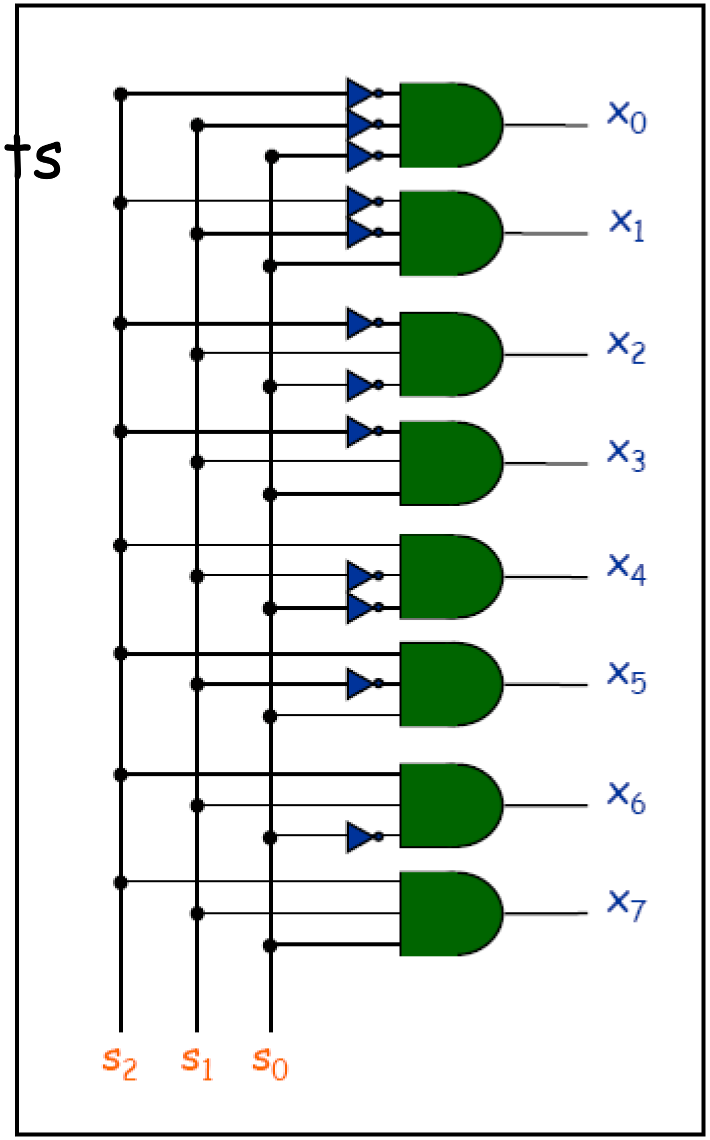
N-bit Decoder

N-bit decoder

N address inputs, 2^N data outputs
Addresses output bit is 1;
all others are 0



3-Bit Decoder Interface

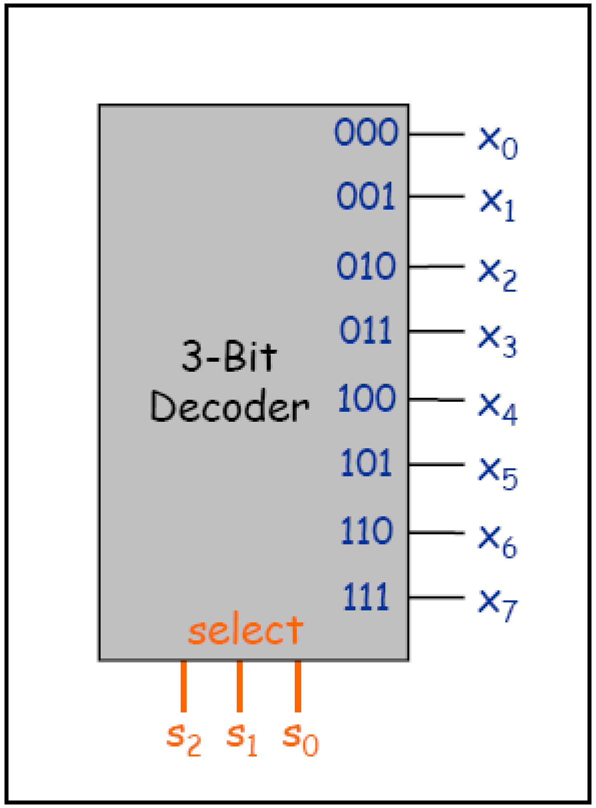


3-Bit Decoder Implementation

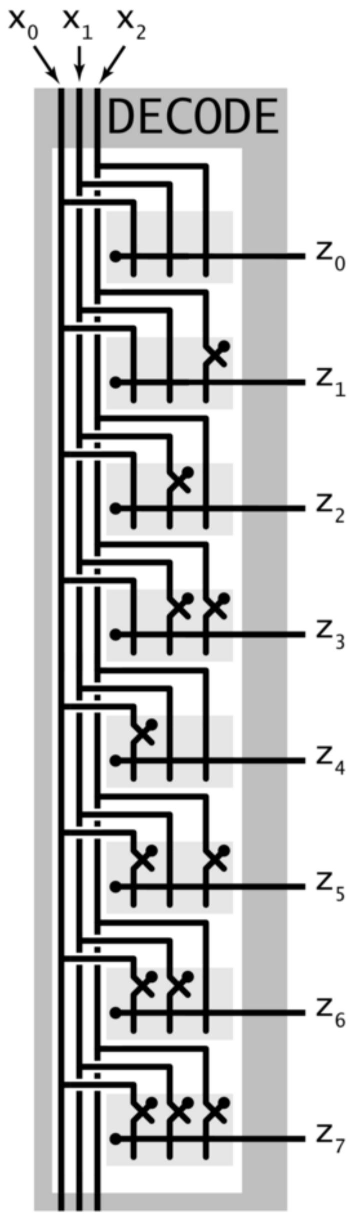
N-bit Decoder

N-bit decoder

N address inputs, 2^N data outputs
Addresses output bit is 1;
all others are 0



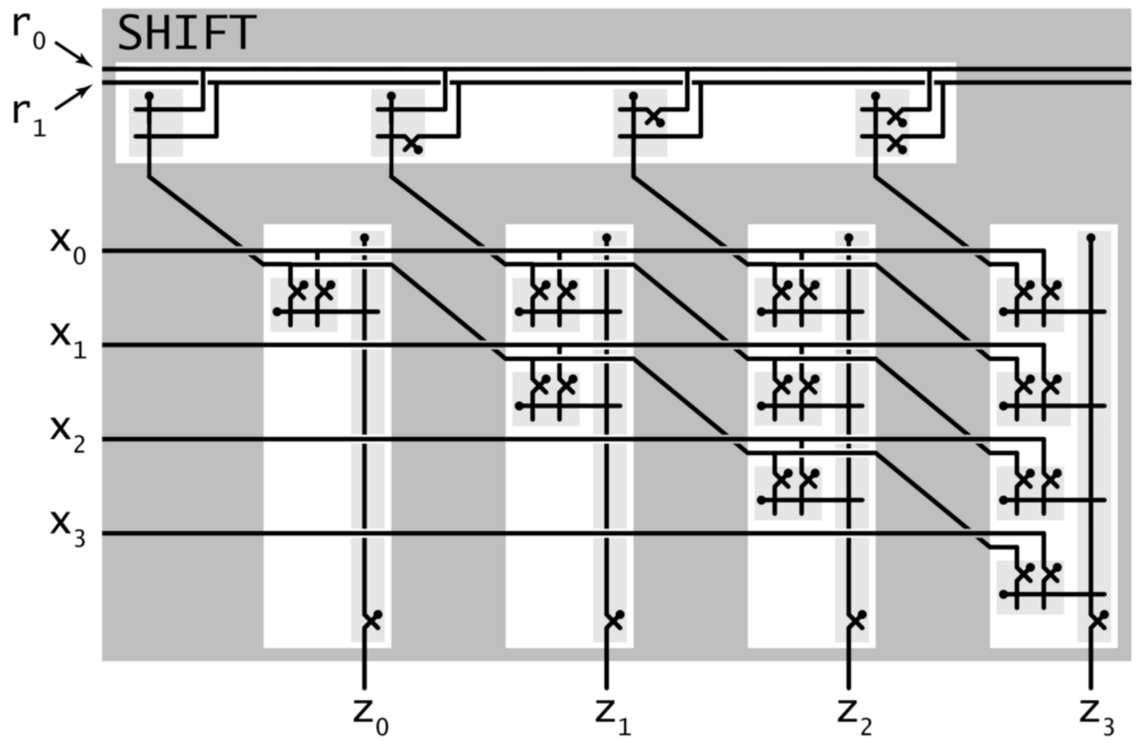
3-Bit Decoder Interface



Decoder

2-Bit Decoder Controlling 4-Bit Shifter

Ex. Put in a binary amount $r_0 r_1$ to shift.



shifter with decoder

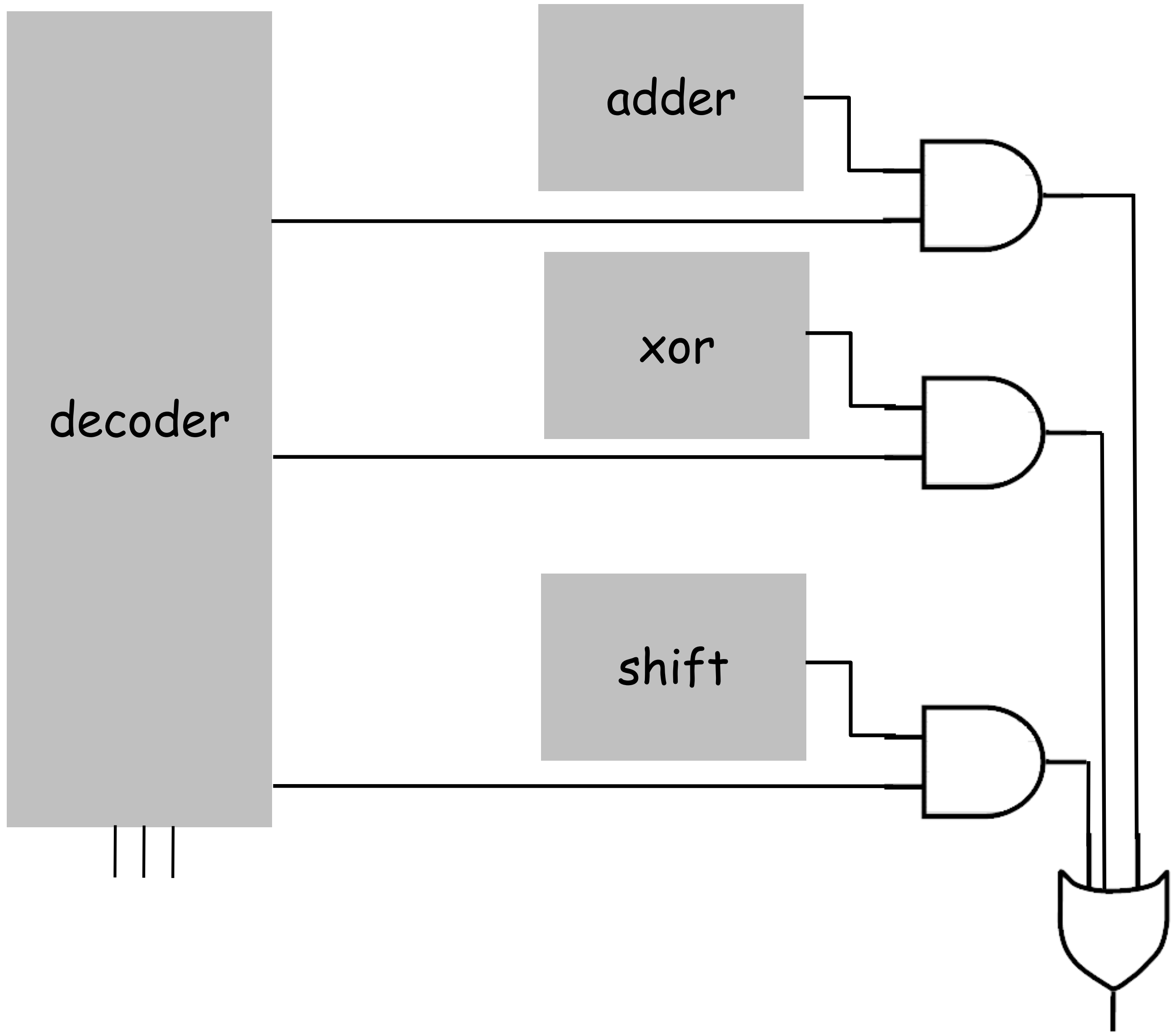
Arithmetic Logic Unit

Arithmetic logic unit (ALU). Computes all operations in parallel.

- Add and subtract.
- Xor.
- And.
- Shift left or right.

Q. How to select desired answer?

1 Hot OR



$x \cdot 1 = x$
$x \cdot 0 = 0$

$x + 0 = x$

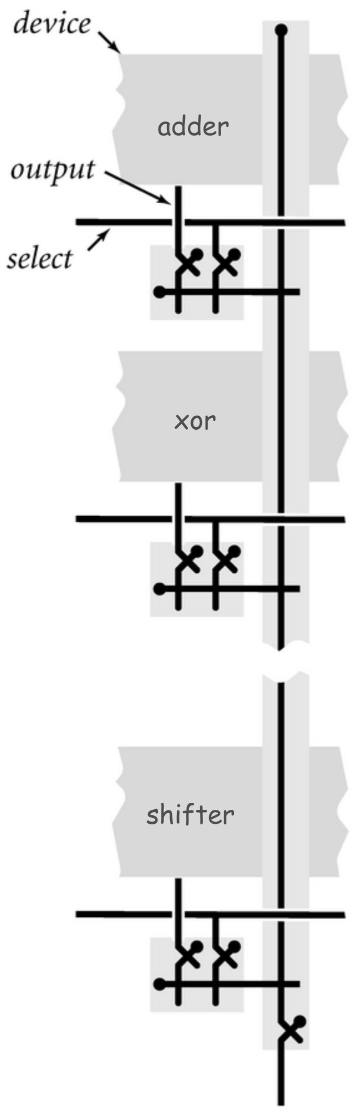
1 Hot OR

1 hot OR.

All devices compute their answer; we pick one.
Exactly one select line is on.
Implies exactly one output line is relevant.

$x \cdot 1 = x$
$x \cdot 0 = 0$

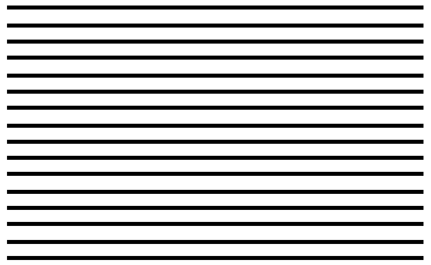
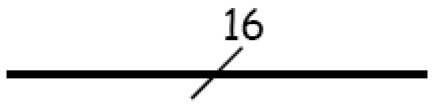
$x + 0 = x$



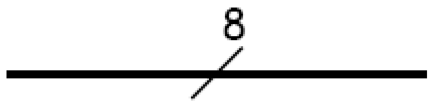
Output select with one-hot OR

Bus

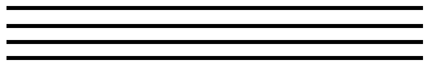
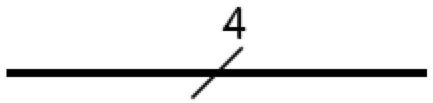
16-bit bus
- Bundle of 16 wires
- Memory transfer
- Register transfer

8-bit bus
- Bundle of 8 wires
- TOY memory address



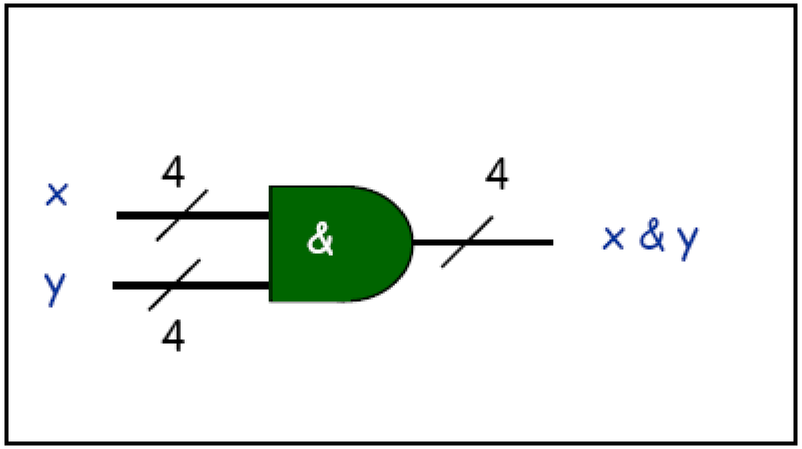
4-bit bus
- Bundle of 4 wires
- TOY register address

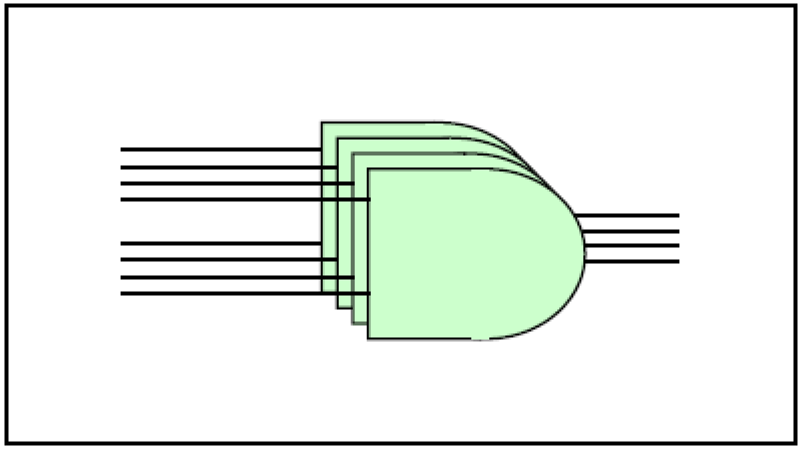
Bitwise AND, XOR, NOT

Bitwise logical operations

Inputs x and y: n bits each
Output z: n bits
Apply logical operation to each corresponding pair of bits



Bitwise And Interface



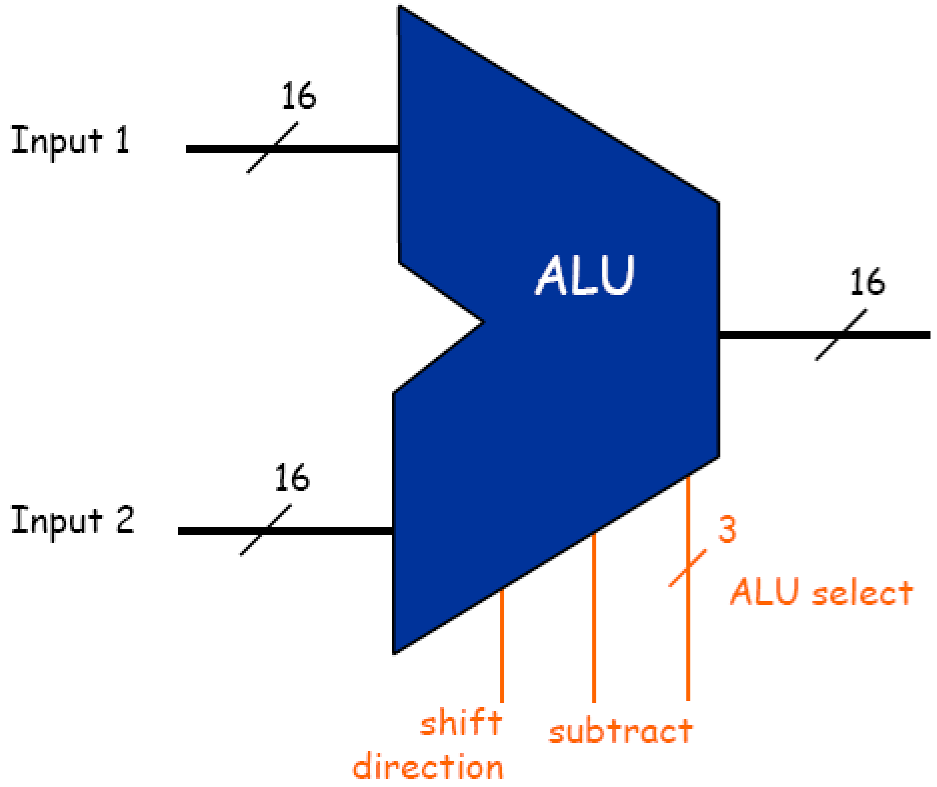
Bitwise And Implementation

TOY ALU

TOY ALU

Big combinational logic
16-bit bus
Add, subtract, and, xor, shift left, shift right, copy input 2.

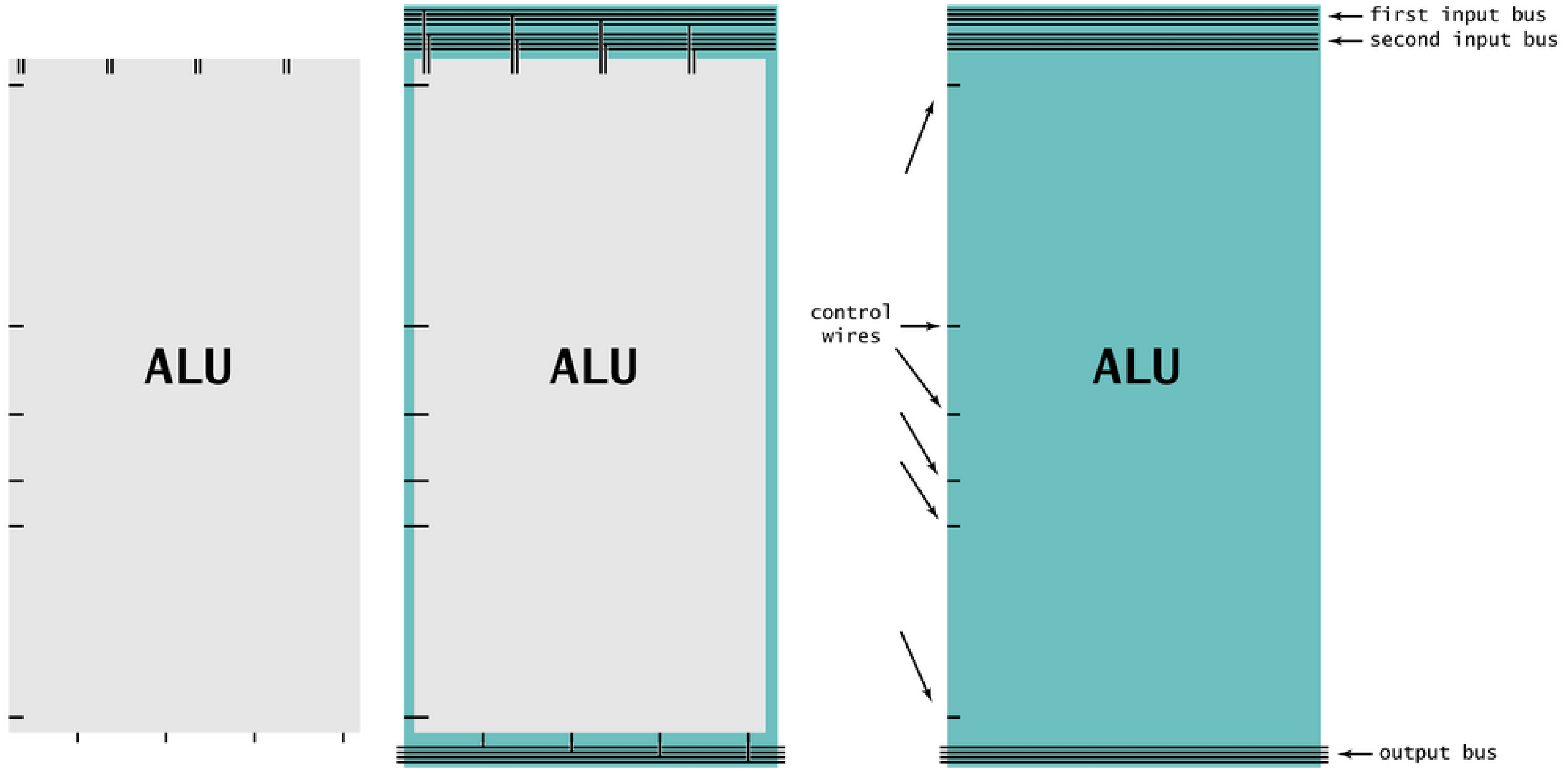
op	2	1	0
+, -	0	0	0
&	0	0	1
^	0	1	0
<<, >>	0	1	1
input 2	1	0	0



Device Interface Using Buses

16-bit words for TOY memory

Device. Processes a word at a time.
Input bus. Wires on top.
Output bus. Wires on bottom.
Control. Individual wires on side.



ALU

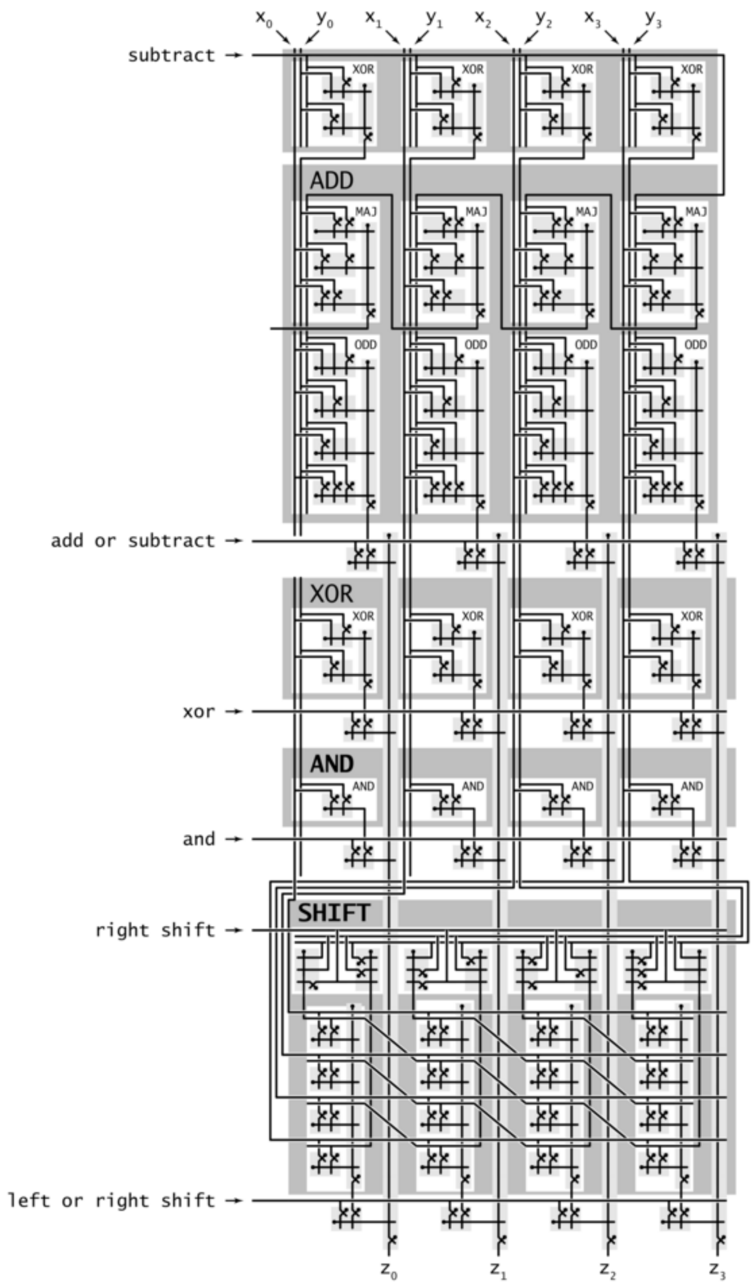
Arithmetic logic unit.

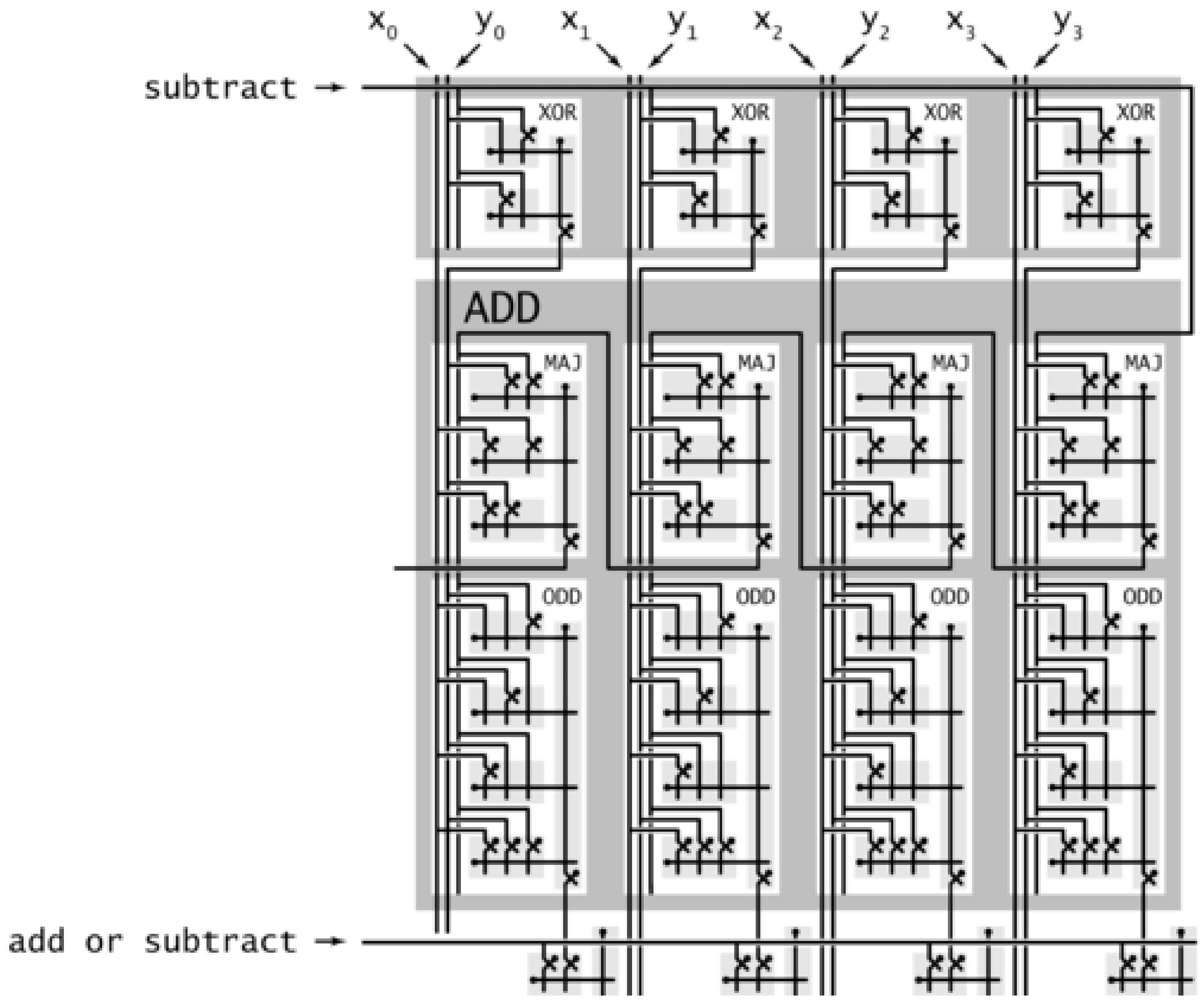
- Add and subtract.
- Xor.
- And.
- Shift left or right.

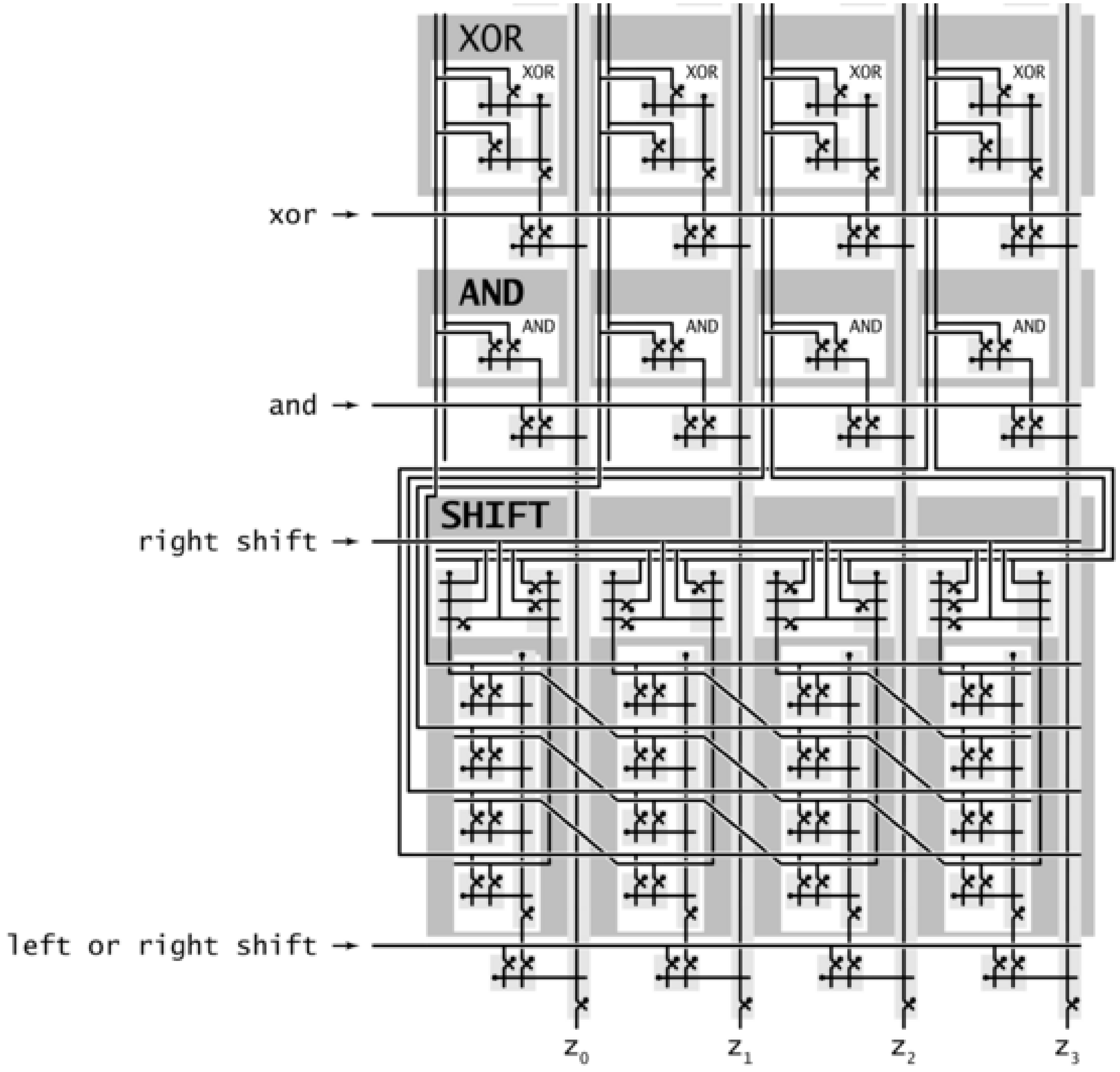
Arithmetic logic unit.

- Computes all operations in parallel.
- Uses 1-hot OR to pick each bit answer.

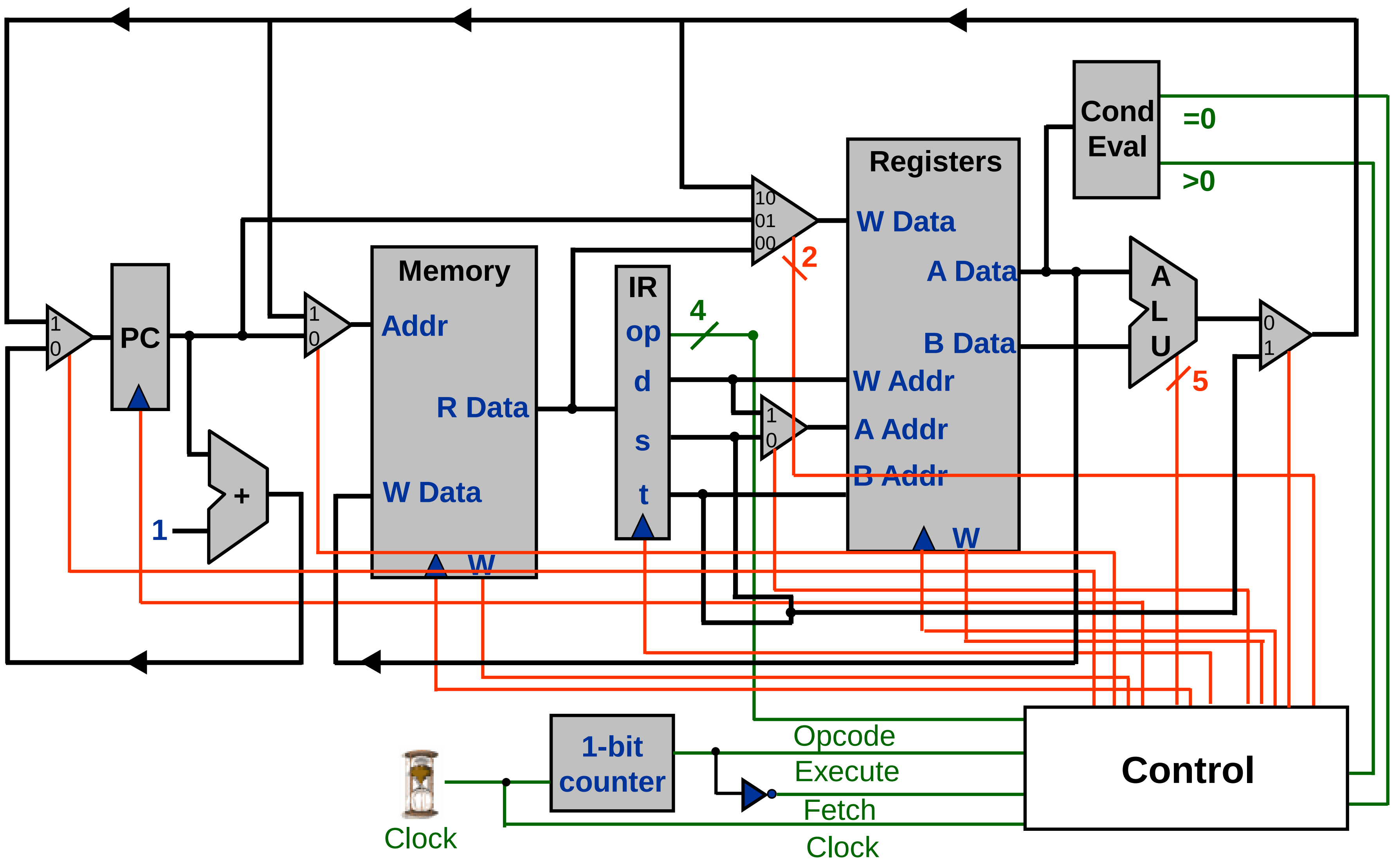
How to convert opcode to 1-hot OR signal?



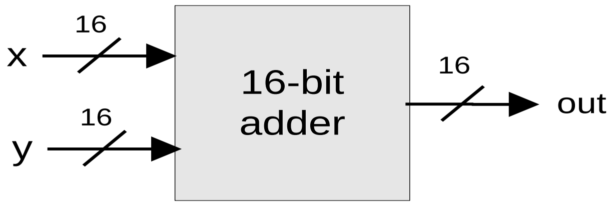

x_0
y_0
x_1
y_1
x_2
y_2
x_3
y_3
subtract →
XOR
XOR
XOR
XOR
ADD
MAJ
MAJ
MAJ
MAJ
ODD
ODD
ODD
ODD
add or subtract →

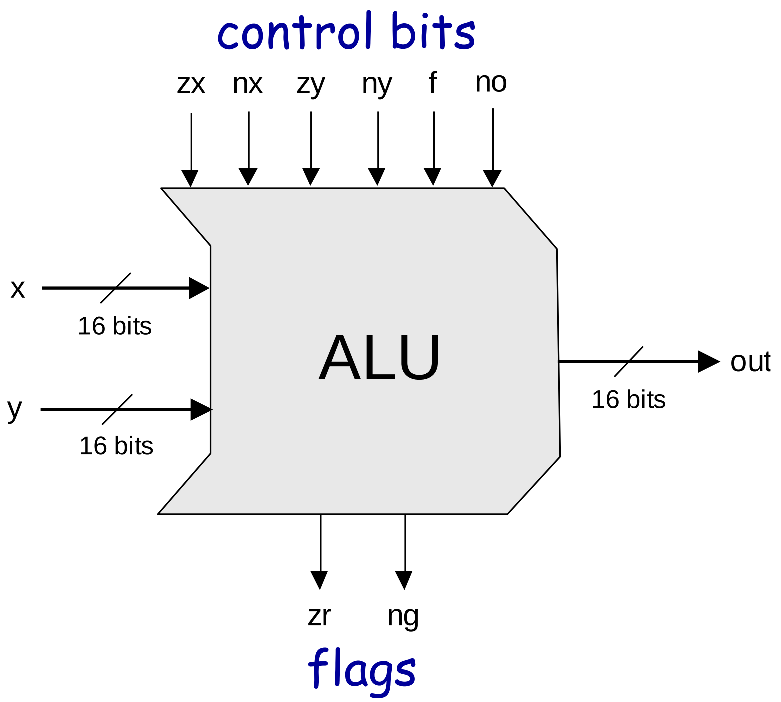

XOR
XOR
XOR
XOR
XOR
xor →
AND
AND
AND
AND
AND
and →
SHIFT
right shift →
left or right shift →
z_0
z_1
z_2
z_3

The ALU in the CPU context



Hack ALU





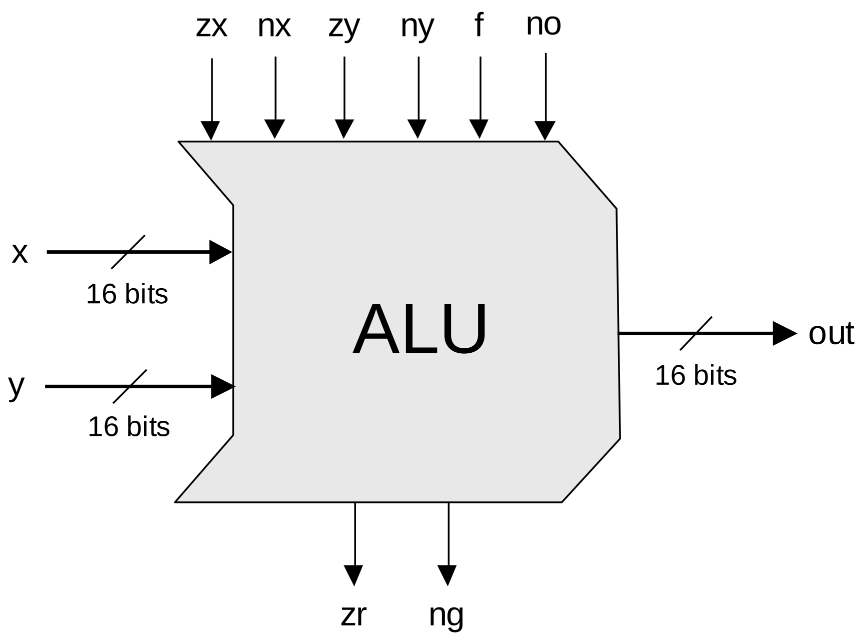
out(x, y, control bits**) =**

```
x+y, x-y, y-x,
0, 1, -1,
x, y, -x, -y,
x!, y!,
x+1, y+1, x-1, y-1,
x&y, x|y
```

Hack ALU

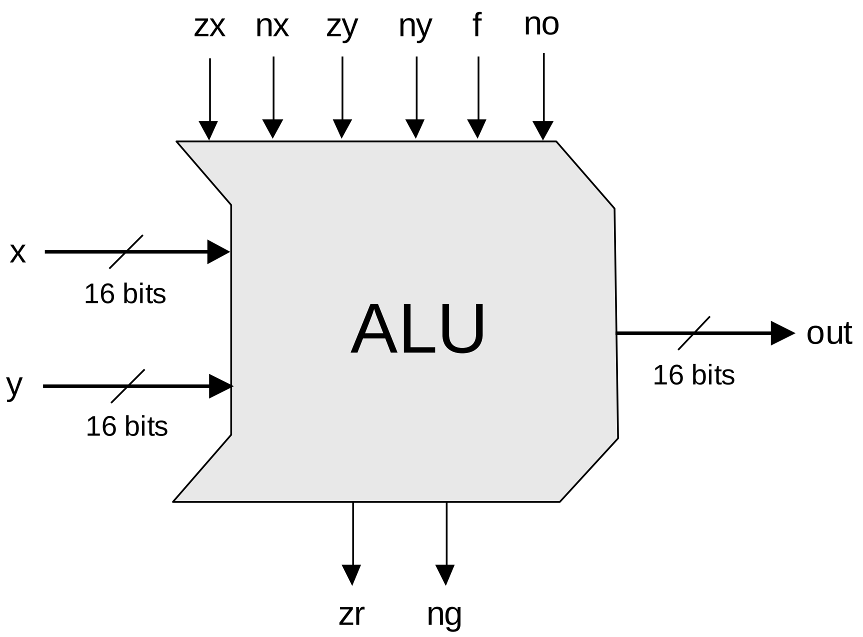
pre-setting
the x input

zx	nx
if zx then x=0	if nx then x=!x



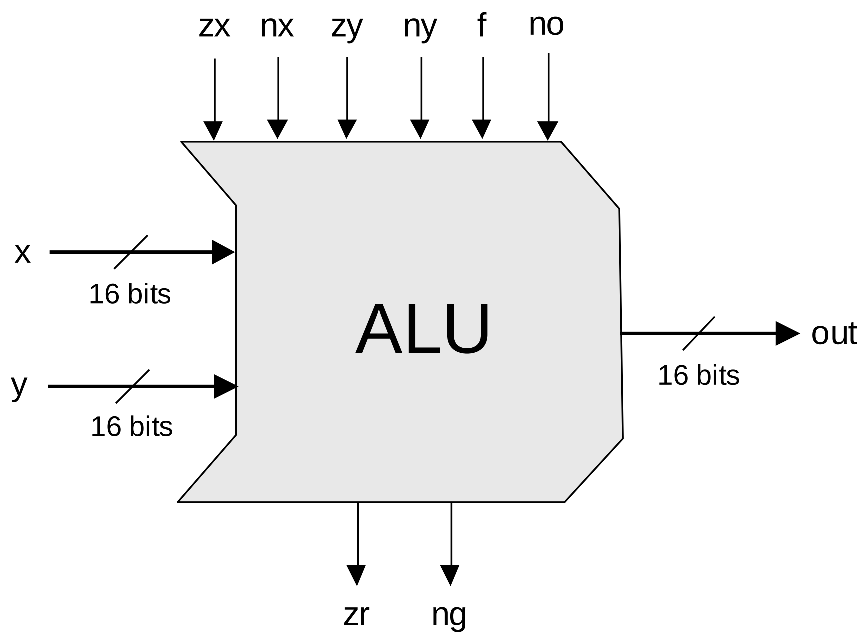
Hack ALU

pre-setting the x input		pre-setting the y input	
zx	**nx**	**zy**	**ny**
`if zx then x=0`	`if nx then x=!x`	`if zy then y=0`	`if ny then y=!y`



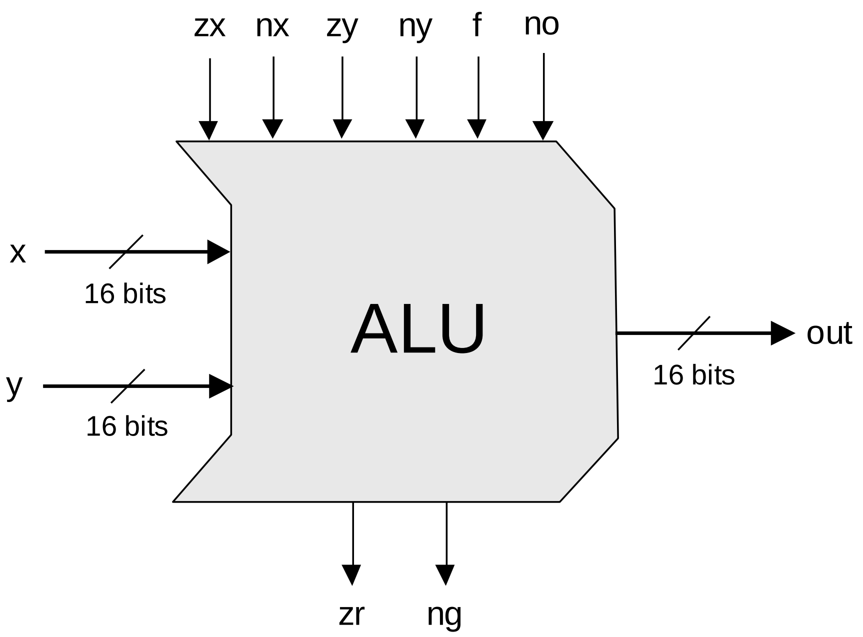
Hack ALU

pre-setting the x input		pre-setting the y input		selecting between computing + or &
zx	**nx**	**zy**	**ny**	**f**
`if zx then x=0`	`if nx then x=!x`	`if zy then y=0`	`if ny then y=!y`	`if f then out=x+y else out=x&y`



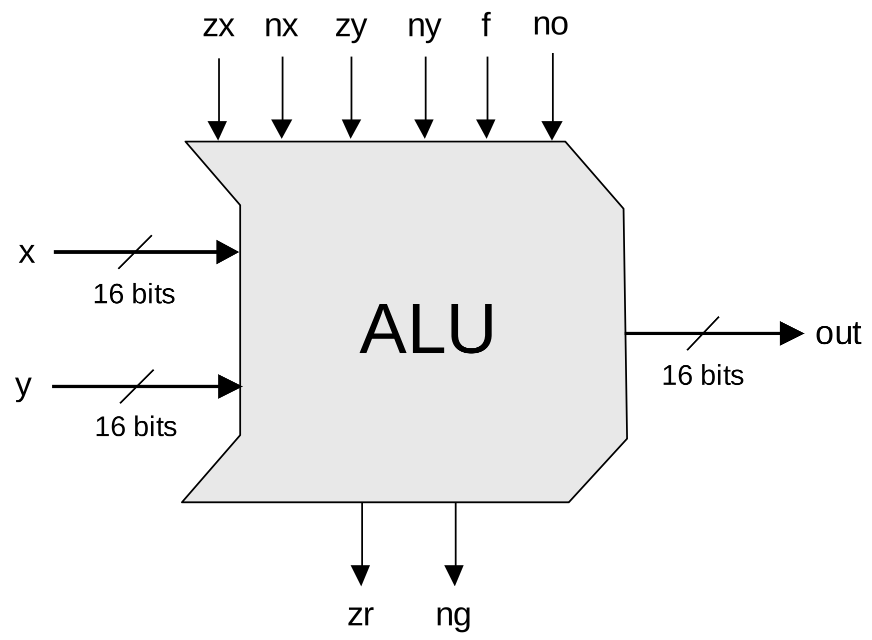
Hack ALU

pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output
zx	**nx**	**zy**	**ny**	**f**	**no**
`if zx then x=0`	`if nx then x=!x`	`if zy then y=0`	`if ny then y=!y`	`if f then out=x+y else out=x&y`	`if no then out=!out`



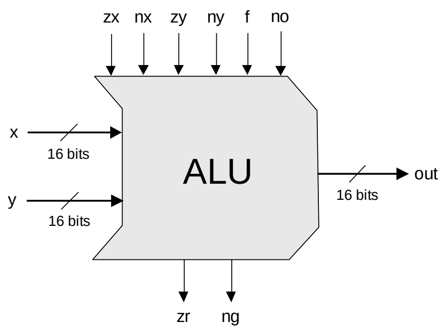
Hack ALU

pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	**nx**	**zy**	**ny**	**f**	**no**	**out**
`if zx then x=0`	`if nx then x=!x`	`if zy then y=0`	`if ny then y=!y`	`if f then out=x+y else out=x&y`	`if no then out=!out`	`out(x,y)=`



Hack ALU

pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	**nx**	**zy**	**ny**	**f**	**no**	**out**
`if zx then x=0`	`if nx then x=!x`	`if zy then y=0`	`if ny then y=!y`	`if f then out=x+y else out=x&y`	`if no then out=!out`	`out(x,y)=`



Hack ALU

pre-setting the x input **zx**	 **nx**	pre-setting the y input **zy**	 **ny**	selecting between computing + or & **f**	post-setting the output **no**	Resulting ALU output **out**
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	out(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Hack ALU: !x

pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	**nx**	**zy**	**ny**	**f**	**no**	**out**
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	out(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Hack ALU: !x

pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	**nx**	**zy**	**ny**	**f**	**no**	**out**
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	out(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0					-x
1	1					-y
0	1					x+1
1	1					y+1
0	0					x-1
1	1					y-1
0	0					x+y
0	1					x-y
0	0					y-x
0	0					x&y
0	1					x|y

Example: compute `!x`

```
x:        1 1 0 0
y:        1 0 1 1  (irrelevant)
```

Following pre-setting:

```
x:        1 1 0 0
y:        1 1 1 1
```

Computation and post-setting:

```
x&y:      1 1 0 0
!(x&y):   0 0 1 1   (!x)
```

Hack ALU: x|y

pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	**nx**	**zy**	**ny**	**f**	**no**	**out**
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	out(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1					-1
0	0					x
1	1					y
0	0					!x
1	1					!y
0	0					-x
1	1					-y
0	1					x+1
1	1					y+1
0	0					x-1
1	1					y-1
0	0					x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Example: compute x|y

```
x:        0 1 0 1
y:        0 0 1 1
```

Following pre-setting:

```
x:        1 0 1 0
y:        1 1 0 0
```

Computation and post-setting:

```
x&y:      1 0 0 0
!(x&y):   0 1 1 1
```

Hack ALU: y-1

pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	**nx**	**zy**	**ny**	**f**	**no**	**out**
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	out(x,y)=
1	0					0
1	1					1
1	1					-1
0	0					x
1	1					y
0	0					!x
1	1					!y
0	0					-x
1	1					-y
0	1					x+1
1	1					y+1
0	0					x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Example: compute y-1

x: 0 1 0 1 (irrelevant)
y: 0 1 1 0 (6)

Following pre-setting:

x: 1 1 1 1
y: 0 1 1 0

Computation and post-setting:

x+y: 0 1 0 1
x+y: 0 1 0 1 (5)

Hack ALU: y-x

pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	**nx**	**zy**	**ny**	**f**	**no**	**out**
`if zx then x=0`	`if nx then x=!x`	`if zy then y=0`	`if ny then y=!y`	`if f then out=x+y else out=x&y`	`if no then out=!out`	`out(x,y)=`
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1					-1
0	0					x
1	1					y
0	0					!x
1	1					!y
0	0					-x
1	1					-y
0	1					x+1
1	1					y+1
0	0					x-1
1	1					y-1
0	0					x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Example: compute `y-x`

```
x:        0 1 0 1 (5)
y:        0 1 1 0 (6)
```

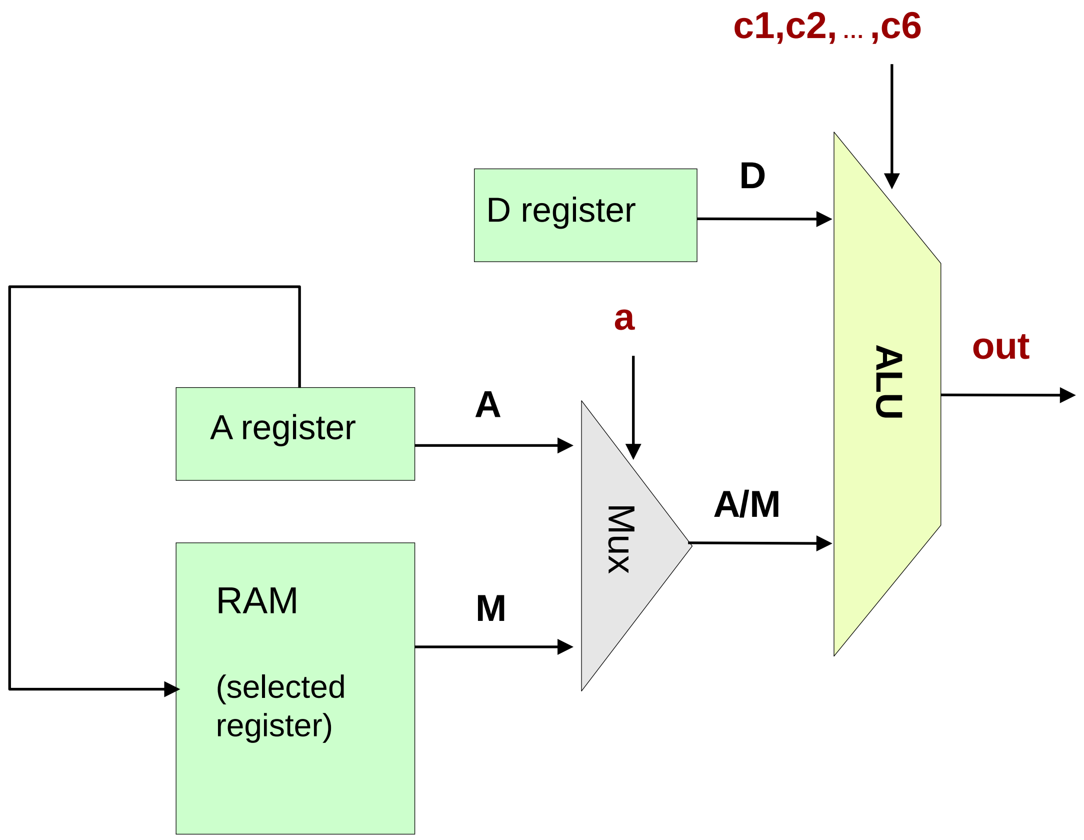
Following pre-setting:

```
x:        0 1 0 1
y:        1 0 0 1
```

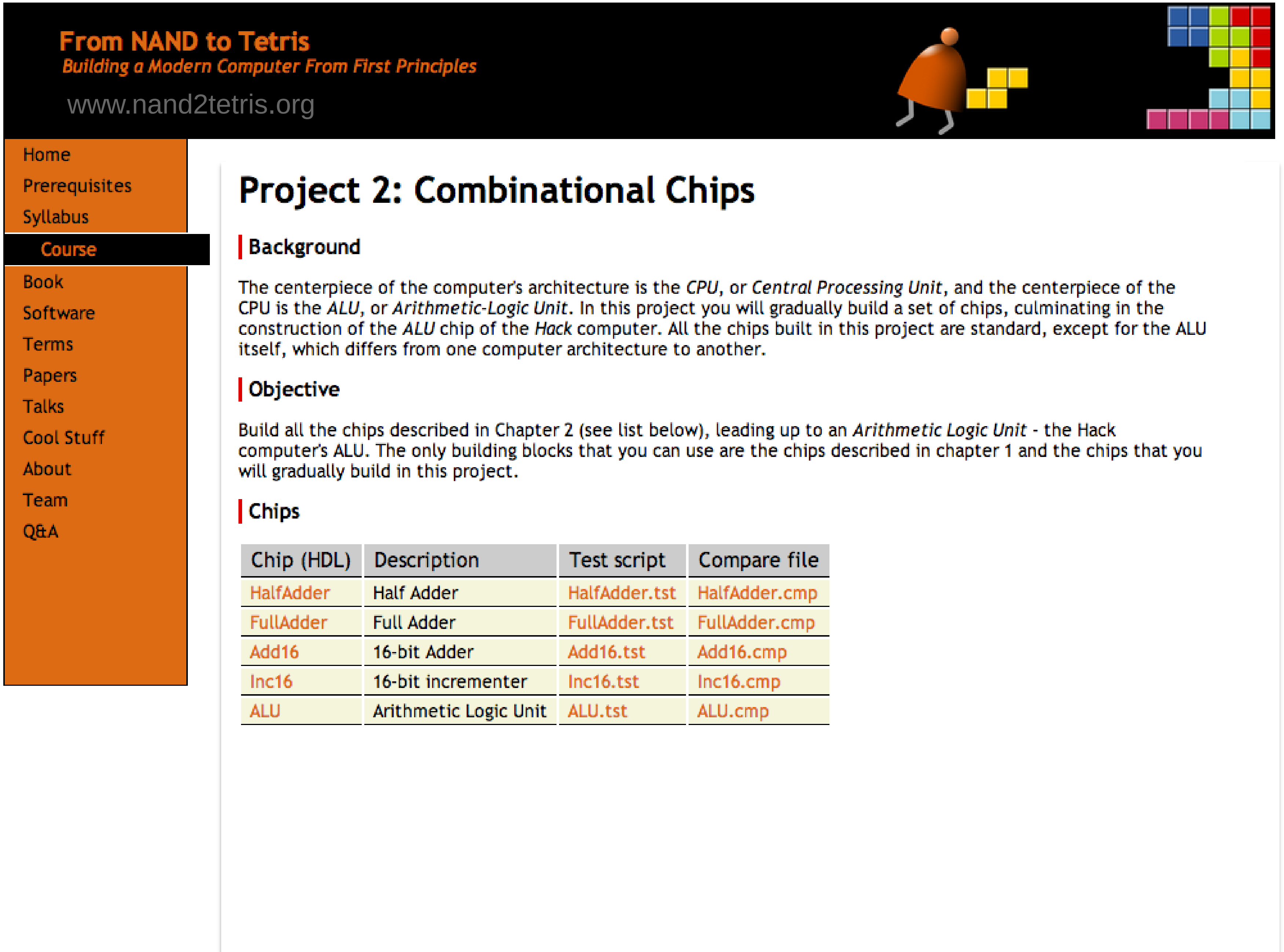
Computation and post-setting:

```
x+y:      1 1 1 0
!(x+y):   0 0 0 1 (1)
```

The ALU in the CPU context (a sneak preview of the Hack platform)



Project 2



From NAND to Tetris
Building a Modern Computer From First Principles
www.nand2tetris.org

Project 2: Combinational Chips

Background

The centerpiece of the computer's architecture is the *CPU*, or *Central Processing Unit*, and the centerpiece of the CPU is the *ALU*, or *Arithmetic-Logic Unit*. In this project you will gradually build a set of chips, culminating in the construction of the *ALU* chip of the *Hack* computer. All the chips built in this project are standard, except for the ALU itself, which differs from one computer architecture to another.

Objective

Build all the chips described in Chapter 2 (see list below), leading up to an *Arithmetic Logic Unit* - the Hack computer's ALU. The only building blocks that you can use are the chips described in chapter 1 and the chips that you will gradually build in this project.

Chips

Chip (HDL)	Description	Test script	Compare file
HalfAdder	Half Adder	HalfAdder.tst	HalfAdder.cmp
FullAdder	Full Adder	FullAdder.tst	FullAdder.cmp
Add16	16-bit Adder	Add16.tst	Add16.cmp
Inc16	16-bit incrementer	Inc16.tst	Inc16.cmp
ALU	Arithmetic Logic Unit	ALU.tst	ALU.cmp

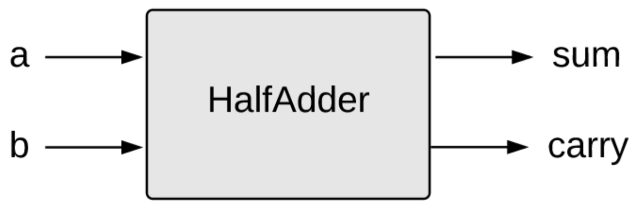
Project 2

Given: All the chips built in Project 1

Goal: Build the chips:

- `HalfAdder`
- `FullAdder`
- `Add16`
- `Inc16`
- `ALU`

Half Adder



a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

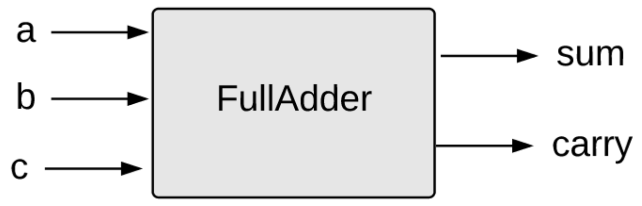
HalfAdder.hdl

```
/** Computes the sum of two bits. */
CHIP HalfAdder {
    IN a, b;
    OUT sum, carry;
    PARTS:
    // Put your code here:
}
```

Implementation tip

Can be built from two gates built in project 1.

Full Adder



a	b	c	sum	carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

FullAdder.hdl

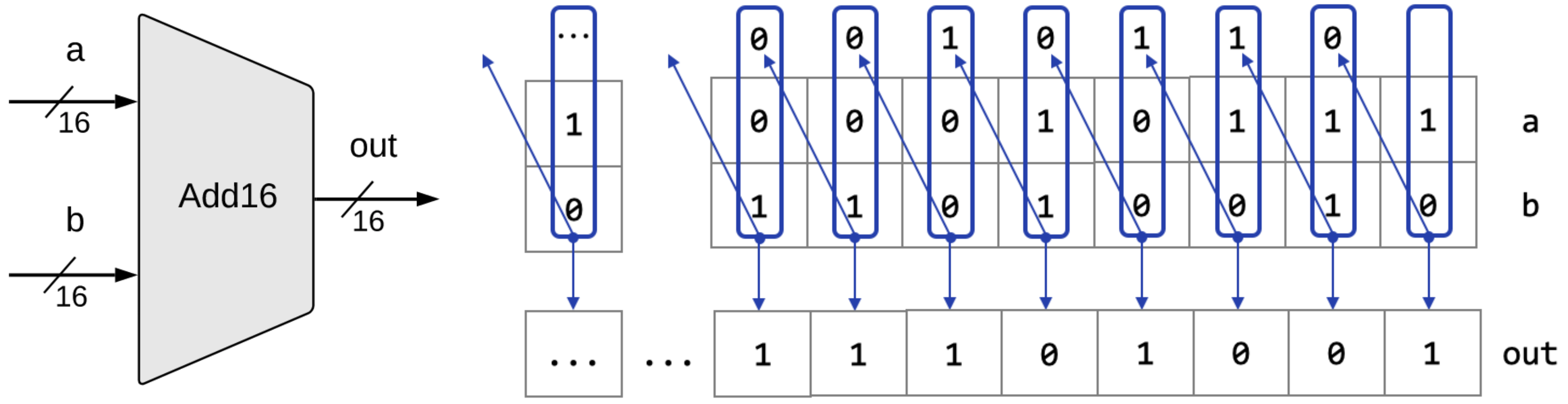
```
/** Computes the sum of three bits. */
CHIP FullAdder {
    IN a, b, c;
    OUT sum, carry;

    PARTS:
    // Put your code here:
}
```

Implementation tip

Can be built from two half-adders.

16-bit Adder



Add16.hdl

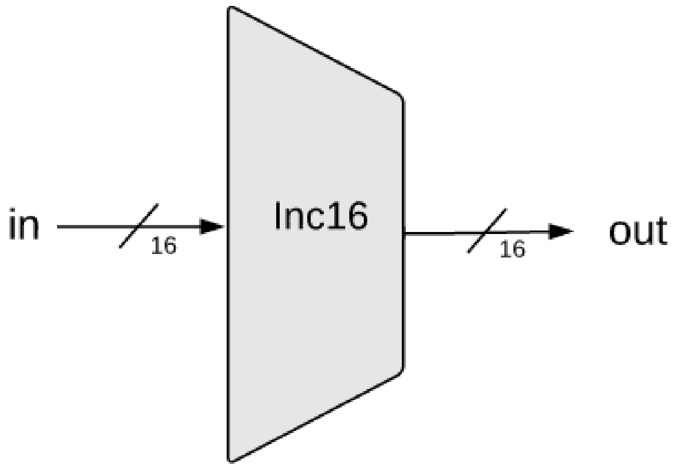
```
/* Adds two 16-bit, two's-complement values.
   The most-significant carry bit is ignored. */
CHIP Add16 {
    IN a[16], b[16];
    OUT out[16];
    PARTS:
    // Put you code here:
}
```

- The bitwise additions are done in parallel
- The carry propagation is sequential
- Yet… it works fine, as is.

<u>Implementation note</u>

If you need to set a pin `x` to `0` (or `1`) in HDL, use: `x = false` (or `x = true`)

16-bit incrementor



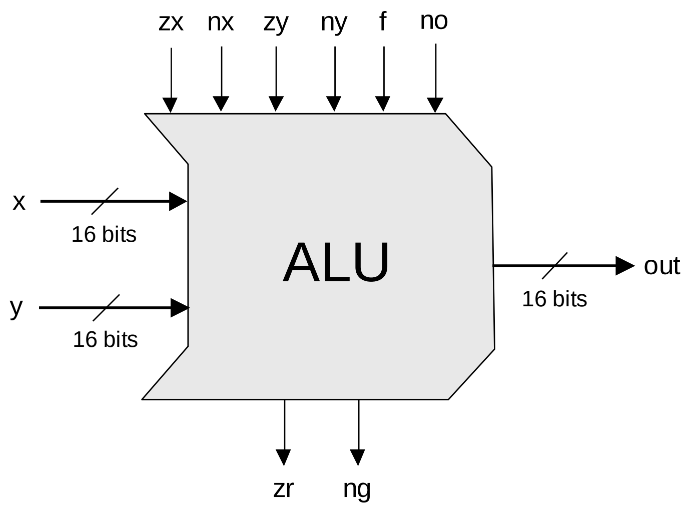
Inc16.hdl

```
/** Outputs in + 1. */
CHIP Inc16 {
    IN in[16];
    OUT out[16];
    PARTS:
    // Put you code here:
}
```

Implementation:

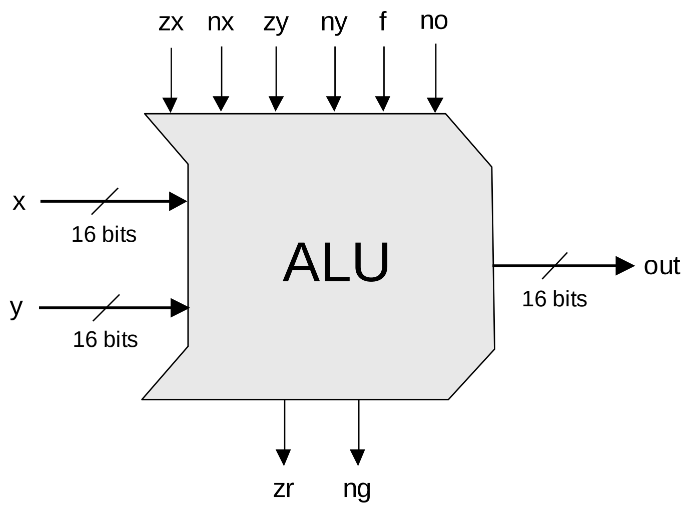
Simple.

ALU



pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	**nx**	**zy**	**ny**	**f**	**no**	**out**
`if zx then x=0`	`if nx then x=!x`	`if zy then y=0`	`if ny then y=!y`	`if f then out=x+y else out=x&y`	`if no then out=!out`	`out(x,y)=`
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

ALU

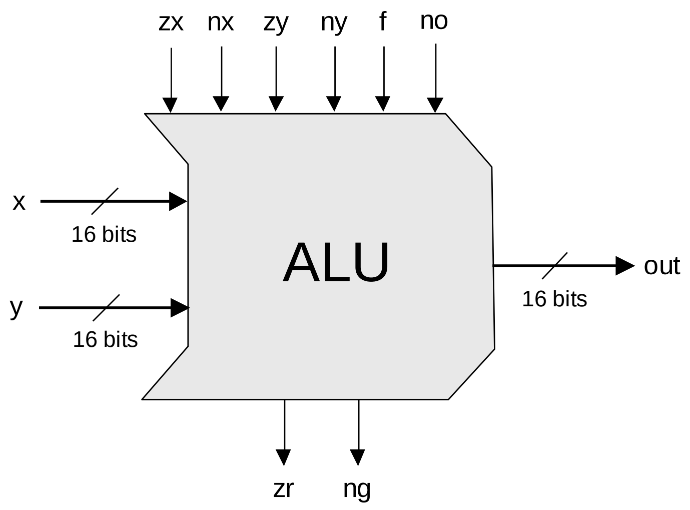


pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	**nx**	**zy**	**ny**	**f**	**no**	**out**
`if zx then x=0`	`if nx then x=!x`	`if zy then y=0`	`if ny then y=!y`	`if f then out=x+y else out=x&y`	`if no then out=!out`	`out(x,y)=`

ALU.hdl

```
/** The ALU */
// Manipulates the x and y inputs as follows:
// if (zx  == 1) sets x = 0        // 16-bit true
// if (nx  == 1) sets x = !x       // 16-bit Not
// if (zy  == 1) sets y = 0        // 16-bit true
// if (ny  == 1) sets y = !y       // 16-bit Not
// if (f   == 1) sets out = x + y  // 2's-complement addition
// if (f   == 0) sets out = x & y  // 16-bit And
// if (no  == 1) sets out = !out   // 16-bit Not
// if (out == 0) sets zr = 1       // 1-bit true
// if (out < 0)  sets ng = 1       // 1-bit true
...
```

ALU



Implementation tips

We need logic for:

- Implementing "if bit == `0`/`1`" conditions
- Setting a 16-bit value to `0000000000000000`
- Setting a 16-bit value to `1111111111111111`
- Negating a 16-bit value (bitwise)
- Computing `Add` and `And` on two 16-bit values

`ALU.hdl`

```
/** The ALU */
// Manipulates the x and y inputs as follows:
// if (zx  == 1) sets x = 0        // 16-bit true
// if (nx  == 1) sets x = !x       // 16-bit Not
// if (zy  == 1) sets y = 0        // 16-bit true
// if (ny  == 1) sets y = !y       // 16-bit Not
// if (f   == 1) sets out = x + y  // 2's-complement addition
// if (f   == 0) sets out = x & y  // 16-bit And
// if (no  == 1) sets out = !out   // 16-bit Not
// if (out == 0) sets zr = 1       // 1-bit true
// if (out < 0)  sets ng = 1       // 1-bit true
...
```

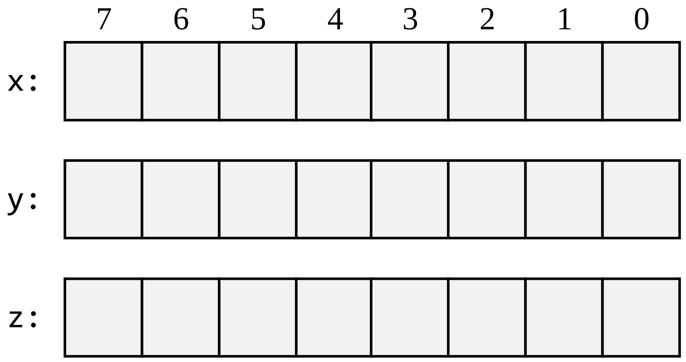
Implementation strategy

- Start by building an ALU that computes `out`
- Next, extend it to also compute `zr` and `ng`.

Relevant bus tips

Using multi-bit `truth` / `false` constants:

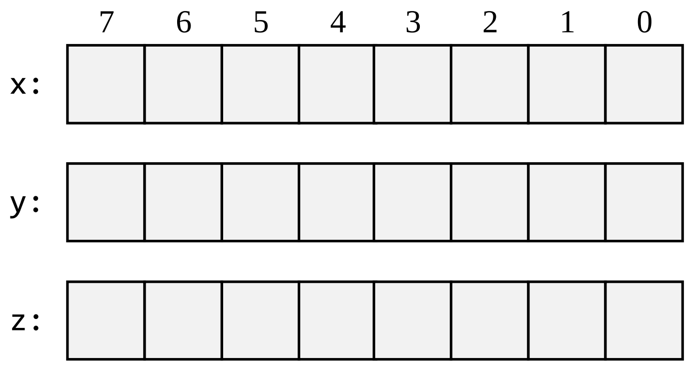
```
...
// Suppose that x, y, z are 8-bit bus-pins:
```



Relevant bus tips

Using multi-bit `truth` / `false` constants:

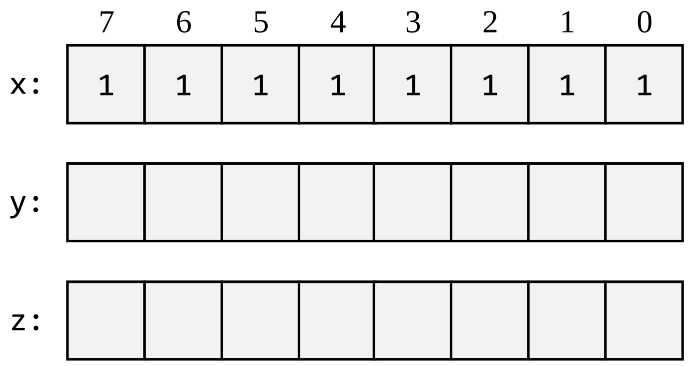
```
...
// Suppose that x, y, z are 8-bit bus-pins:
chipPart(..., x=true, y=false, z[0..2]=true, z[6..7]=true);
...
```



Relevant bus tips

Using multi-bit `truth` / `false` constants:

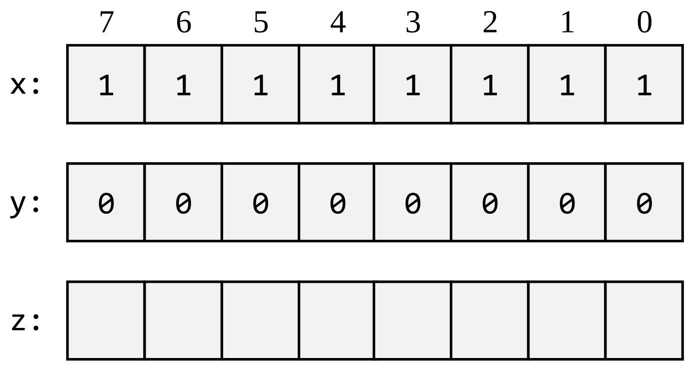
```
...
// Suppose that x, y, z are 8-bit bus-pins:
chipPart(..., x=true, y=false, z[0..2]=true, z[6..7]=true);
...
```



Relevant bus tips

Using multi-bit `truth` / `false` constants:

```
...
// Suppose that x, y, z are 8-bit bus-pins:
chipPart(..., x=true, y=false, z[0..2]=true, z[6..7]=true);
...
```



Relevant bus tips

Using multi-bit `truth` / `false` constants:

```
...
// Suppose that x, y, z are 8-bit bus-pins:
chipPart(..., x=true, y=false, z[0..2]=true, z[6..7]=true);
...
```

We can assign values to sub-buses

	7	6	5	4	3	2	1	0
x:	1	1	1	1	1	1	1	1
y:	0	0	0	0	0	0	0	0
z:	1	1	0	0	0	1	1	1

Unassigned bits are set to `0`

Relevant bus tips

Sub-bussing:

- We can assign *n*-bit values to sub-buses, for any *n*
- We can create *n*-bit bus pins, for any *n*

```
/* 16-bit adder */
CHIP Add16 {
    IN a[16], b[16];
    OUT out[16];
    PARTS:
    ...
}
```

```
CHIP Foo {
   IN  x[8], y[8], z[16]
   OUT out[16]
   PARTS
   ...
   Add16(                                    );
   ...
   Add16(                                        );
   ...
}
```

Relevant bus tips

Sub-bussing:

- We can assign *n*-bit values to sub-buses, for any *n*
- We can create *n*-bit bus pins, for any *n*

```
/* 16-bit adder */
CHIP Add16 {
    IN a[16], b[16];
    OUT out[16];
    PARTS:
    ...
}
```

```
CHIP Foo {
   IN  x[8], y[8], z[16]
   OUT out[16]
   PARTS
   ...
   Add16(a[0..7]=x, a[8..15]=y, b=z, out=...);
   ...
   Add16(                                       );
   ...
}
```

Another example of assigning a multi-bit value to a sub-bus

Relevant bus tips

Sub-bussing:

- We can assign *n*-bit values to sub-buses, for any *n*
- We can create *n*-bit bus pins, for any *n*

```
/* 16-bit adder */
CHIP Add16 {
    IN a[16], b[16];
    OUT out[16];
    PARTS:
    ...
}
```

```
CHIP Foo {
   IN  x[8], y[8], z[16]
   OUT out[16]
   PARTS
   ...
   Add16 (a[0..7]=x, a[8..15]=y, b=z, out=...);
   ...
   Add16 (a=..., b=..., out[0..3]=t1, out[4..15]=t2);
   ...
}
```

Another example of assigning a multi-bit value to a sub-bus

Creating an *n*-bit bus (internal pin)

Perspective

- Combinational logic
- Our adder design is very basic: no parallelism
- It pays to optimize adders
- Our ALU is also very basic: no multiplication, no division
- Where is the seat of more advanced math operations?
  a typical hardware/software tradeoff.