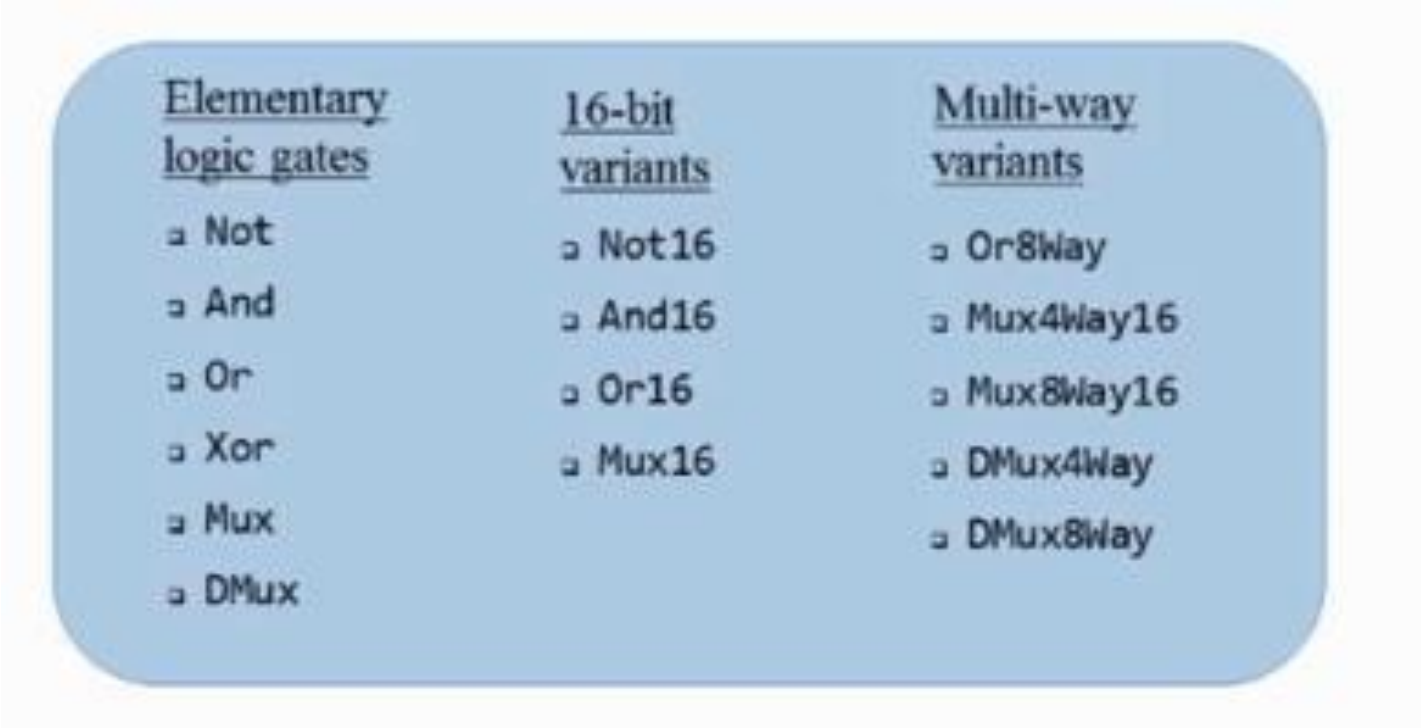
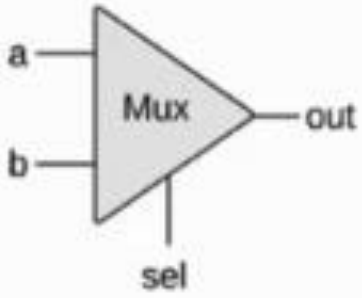
Computer Architecture

Big Picture

Basic CPU Parts



Multiplexor




```
if (sel==0)
   out=a
else
   out=b
```

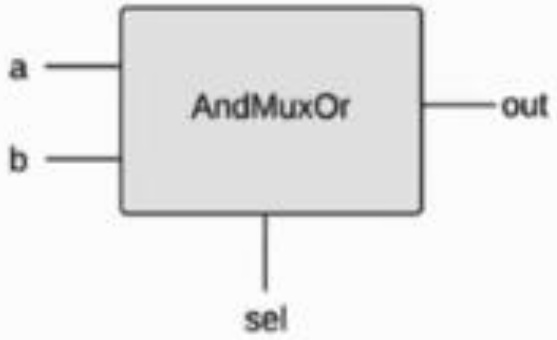
a	b	sel	out
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1

sel	out
0	a
1	b

abbreviated truth table

- A 2-way multiplexor enables selecting, and outputting, one out of two possible inputs
- Widely used in:
  - Digital design
  - Communications networks

Example: using mux logic to build a programmable gate

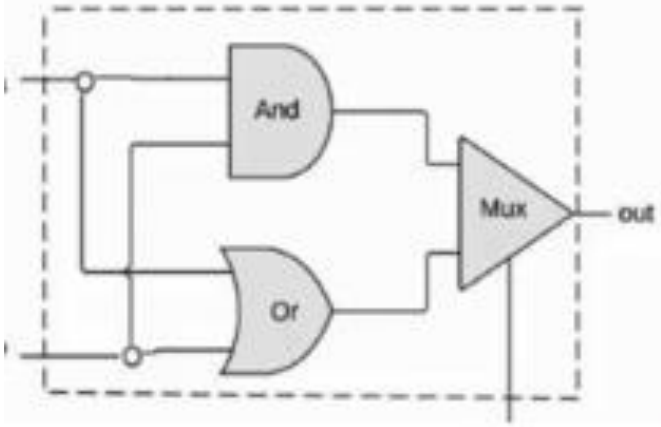


a	b	sel	out
0	0	0	0
0	1	0	0
1	0	0	0
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	1

When `sel==0` the gate acts like an And gate

When `sel==1` the gate acts like an Or gate

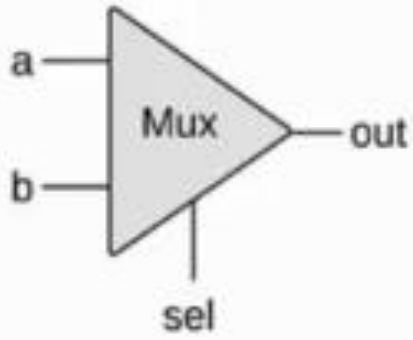
```
if (sel==0)
   out = (a And b)
else
   out = (a Or b)
```



Mux.hdl

```
CHIP AndMuxOr {
    IN a, b, sel;
    OUT out;

    PARTS:
    And (a=a, b=b, out=andOut);
    Or  (a=a, b=b, out=orOut);
    Mux (a=andOut, b=orOut, sel=sel, out=out);
}
```

Multiplexor implementation




```
if (sel==0)
   out=a
else
   out=b
```

sel	out
0	a
1	b

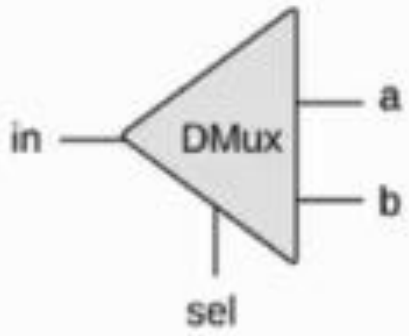
Mux.hdl

```
CHIP Mux {
    IN a, b, sel;
    OUT out;

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

Implementation tip:

Can be implemented with And, Or, and Not gates

Demultiplexor




```
if (sel==0)
   {a,b}={in,0}
else
   {a,b}={0,in}
```

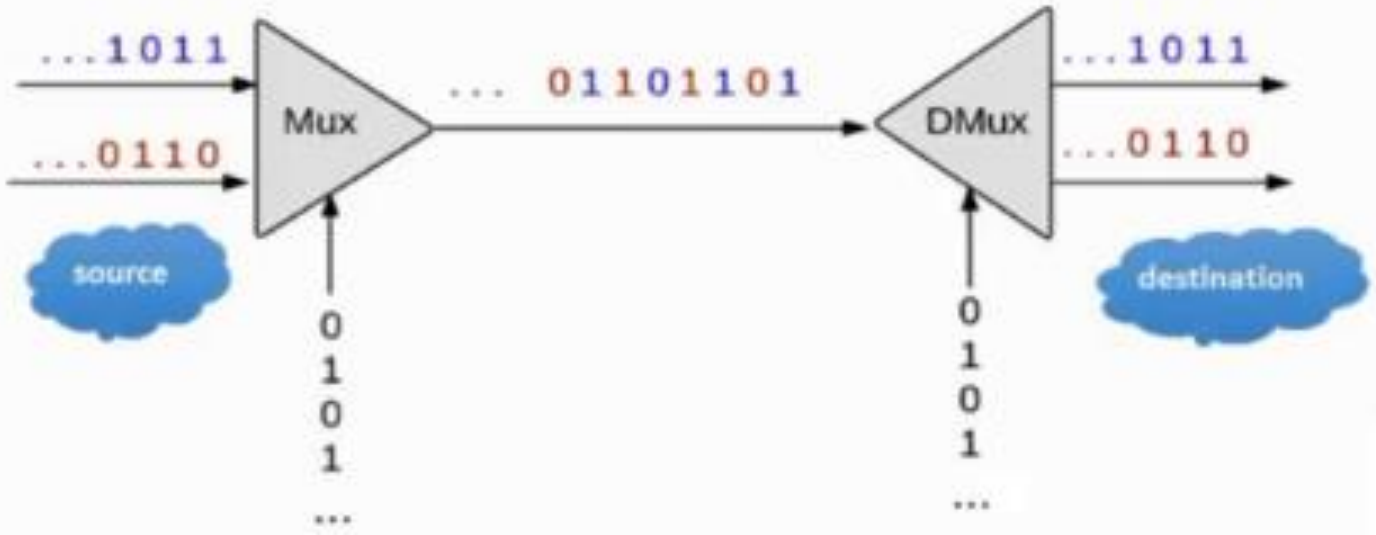
in	sel	a	b
0	0	**0**	**0**
0	1	**0**	**0**
1	0	**1**	**0**
1	1	**0**	**1**

- Acts like the "inverse" of a multiplexor
- Distributes the single input value into one of two possible destinations

DMux.hdl

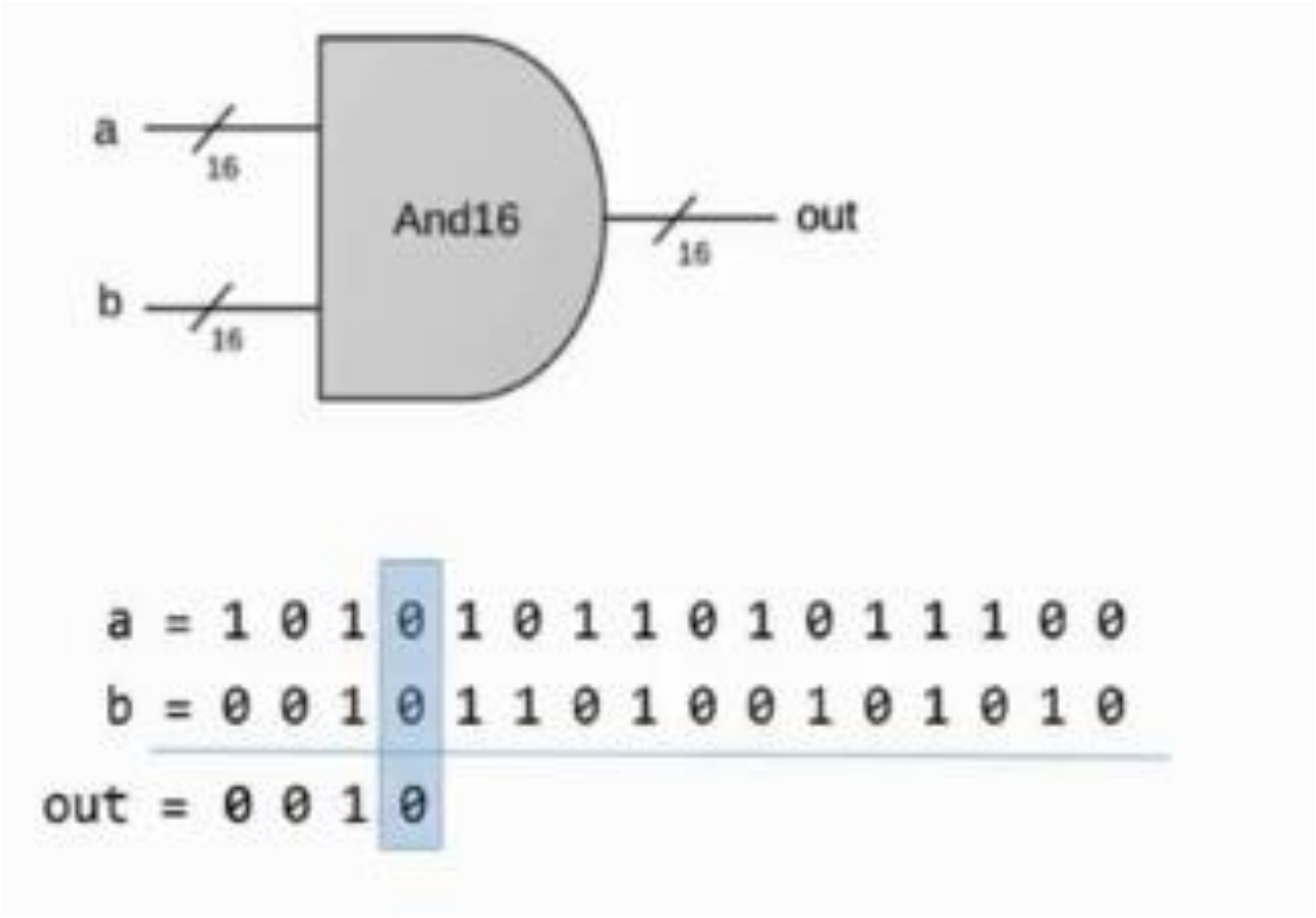
```
CHIP DMux {
    IN in, sel;
    OUT a, b;

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

Example: Multiplexing / demultiplexing in communications networks

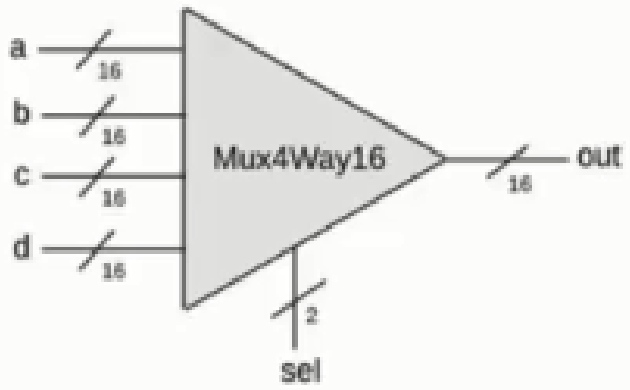


- Each `sel` bit is connected to an oscillator that produces a repetitive train of alternating 0 and 1 signals
- Enables transmitting multiple messages on a single, shared communications line

AND-16


a
16
b
16
And16
16
out
a = 1 0 1 0 1 0 1 1 0 1 0 1 1 1 0 0
b = 0 0 1 0 1 1 0 1 0 0 1 0 1 0 1 0
out = 0 0 1 0

16-bit, 4-way multiplexor



sel[1]	sel[0]	out
0	0	a
0	1	b
1	0	c
1	1	d

Mux4Way16.hdl

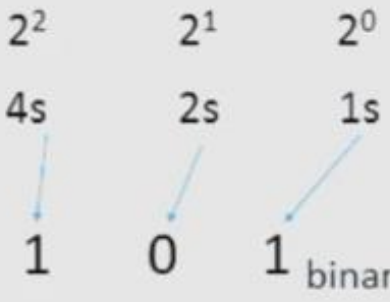
```
CHIP Mux4Way16 {
    IN a[16], b[16], c[16], d[16],
       sel[2];
    OUT out[16];

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

Boolean Arithmetic and the ALU

- Representing Numbers

Binary	Decimal
0	0
1	1
10	2
11	3
100	4
101	5

Binary → Decimal



$$= 1\times2^2 + 0\times2^1 + 1\times2^0 = 5_{\text{Decima}}$$

$$b_n\ b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0$$

$$= \sum_i b_i \cdot 2^i$$

Fixed word size

We will use a fixed number of bits.

Say 8 bits.

0000 0000

0000 0001

.....

1111 1111

$2^8 = 256$ numbers

Decimal → Binary

$87_{decimal} = 64 + 16 + 4 + 2 + 1$

$= 0\ 1\ 0\ 1\ 0\ 1\ 1\ 1_{binary}$

32 8

Binary Addition

```
   1 1 1
 00010101
+
 01011100
 ________
 01110001
```

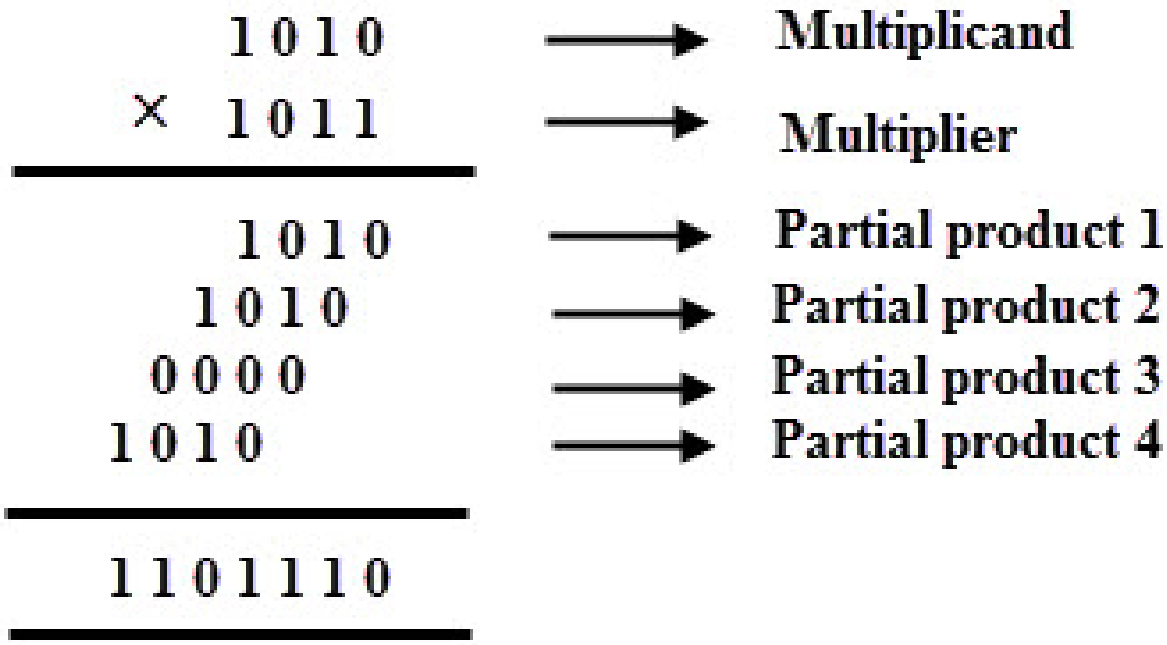
Overflow

```
1    1 1 1
  10010101
+
  11011100
  ________
101110001
```

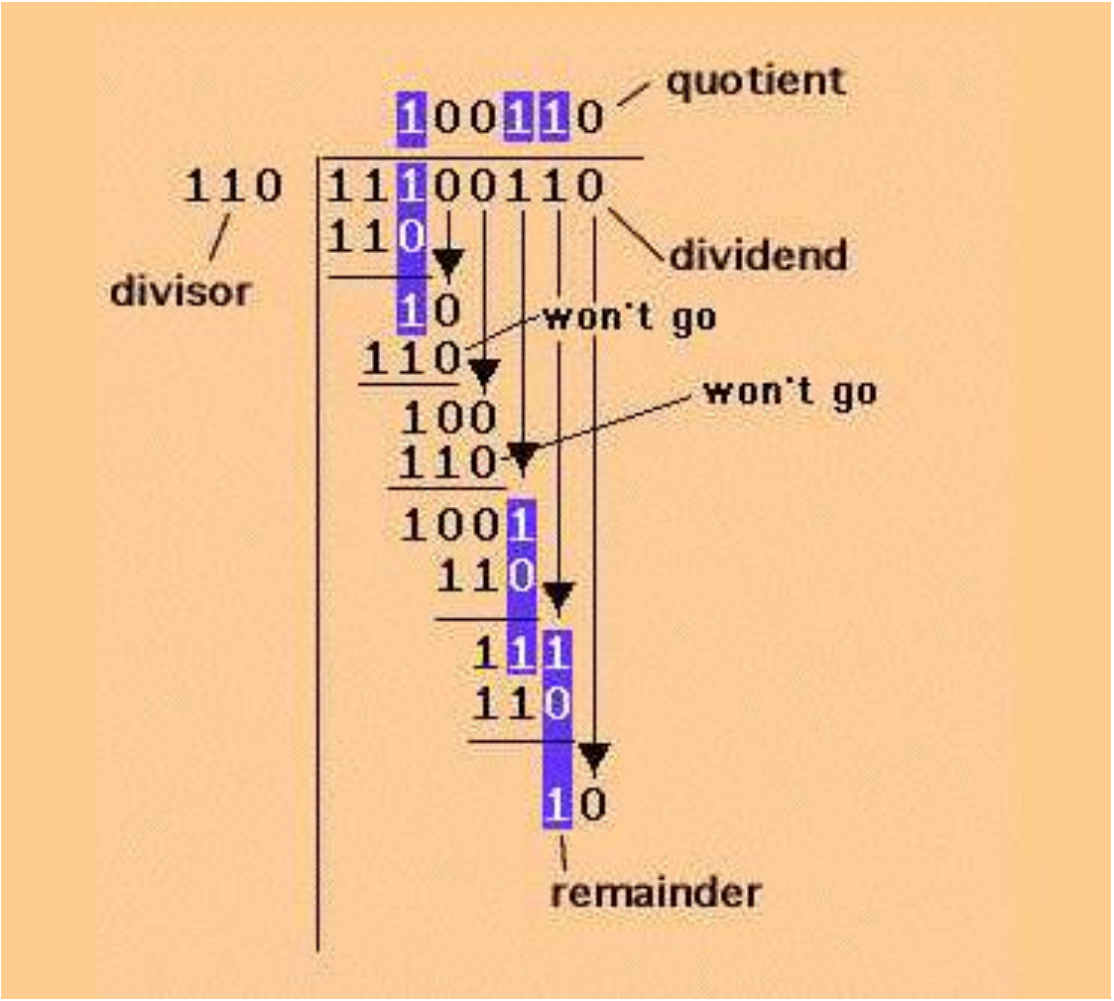
Building an Adder

1. Half Adder – adds two bits
2. Full Adder – adds three bits
3. Adder – Adds two numbers

Binary Multiplication Concept

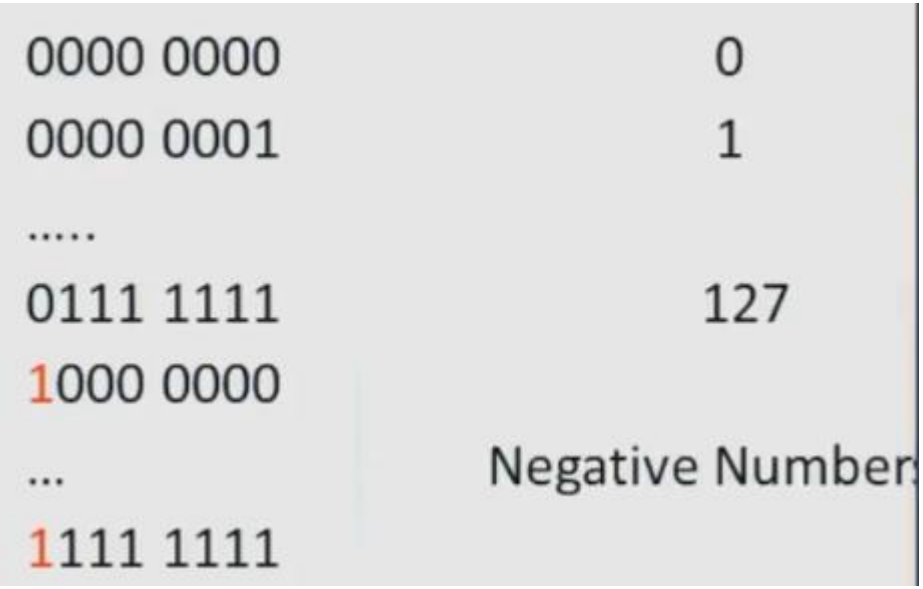


Binary Division Concept



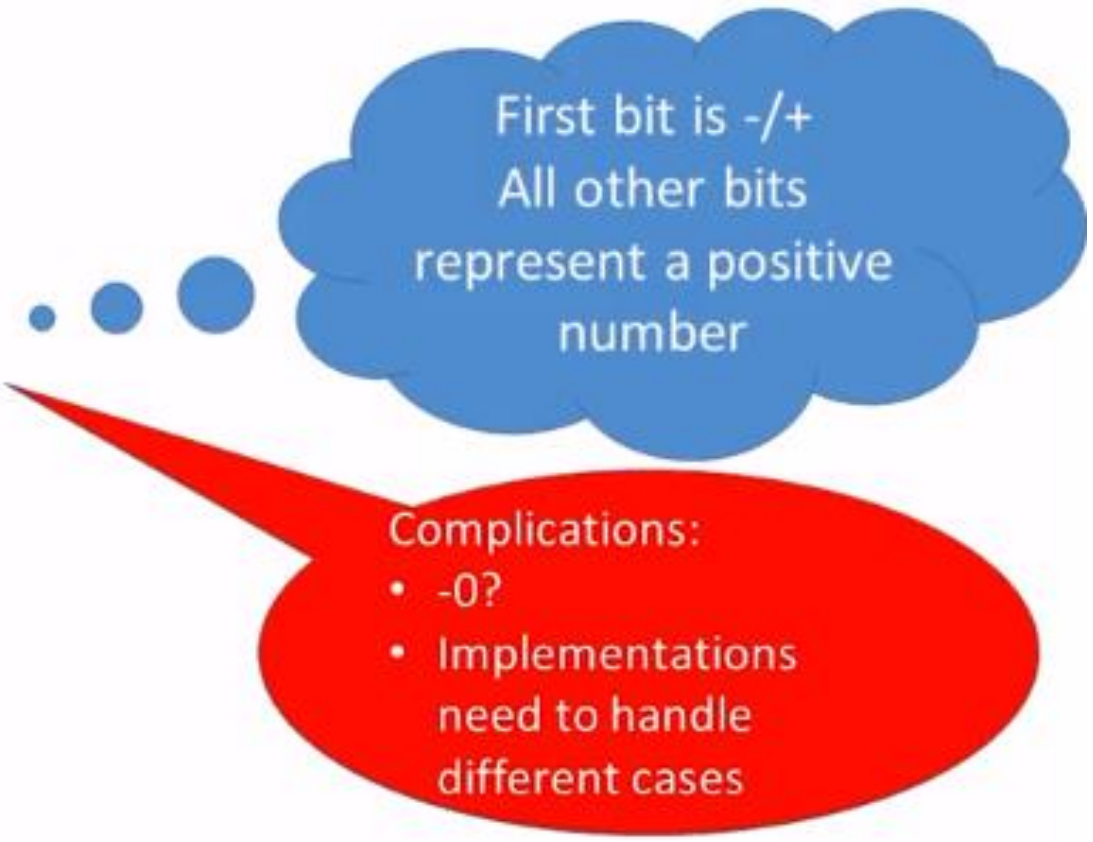
How can we represent negative numbers?

- Old methods

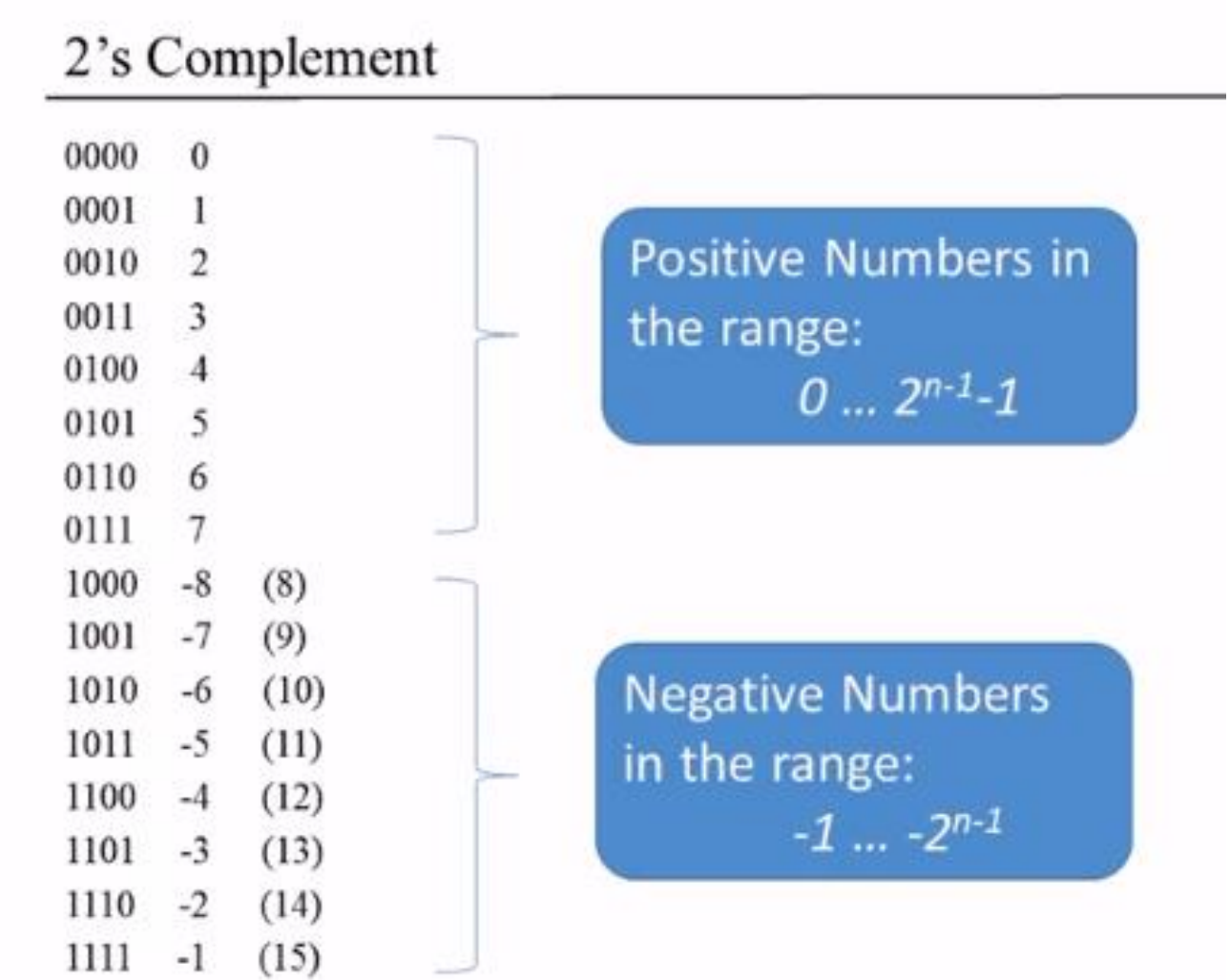


Negative Numbers – Sign bit

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



Representing Negative Numbers



The methods

1- Take inverse for each bit and add 1

For finding the negative of 1 (the number)

0001 Take inverse for each bir and add 1

1110+1=1111

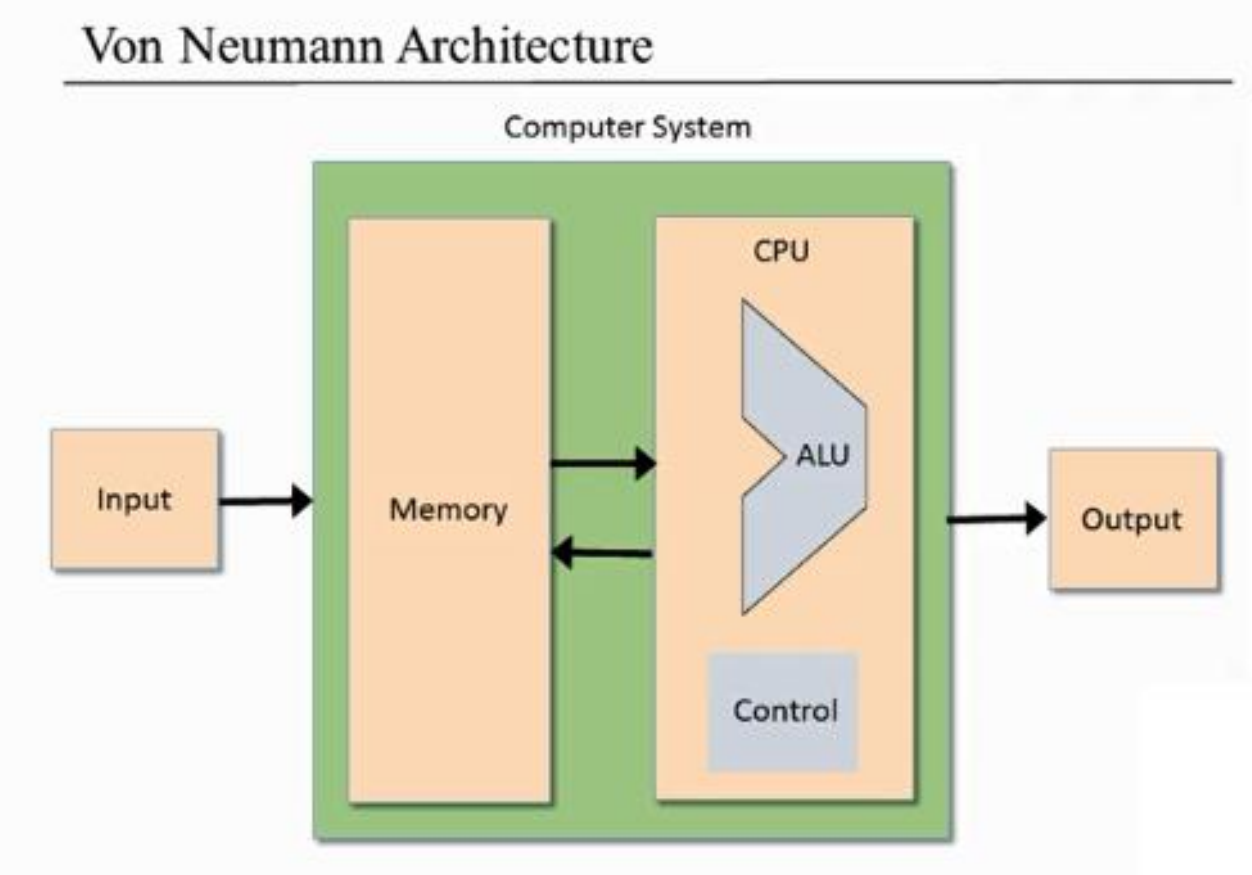
2- $2^n - number = -number$

16-1=15 = 1111 (-1 is represented with 1111)

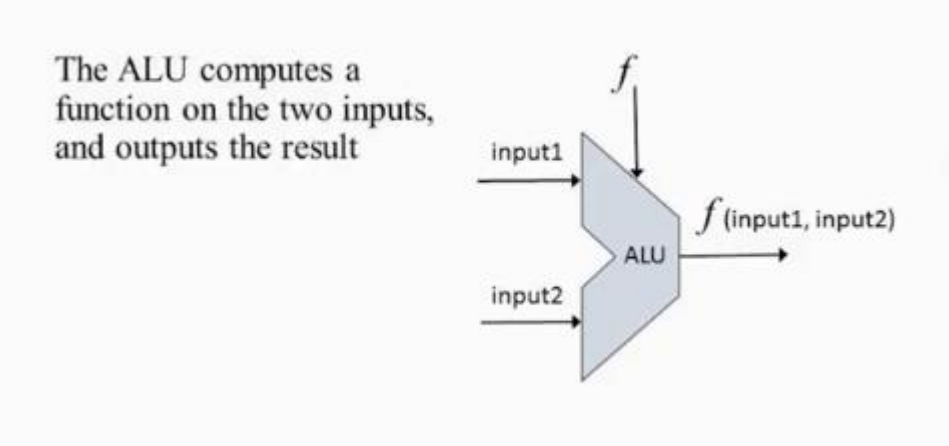
- How can a computer make a substruction?

ALU UNIT

- The grey part is called CPU



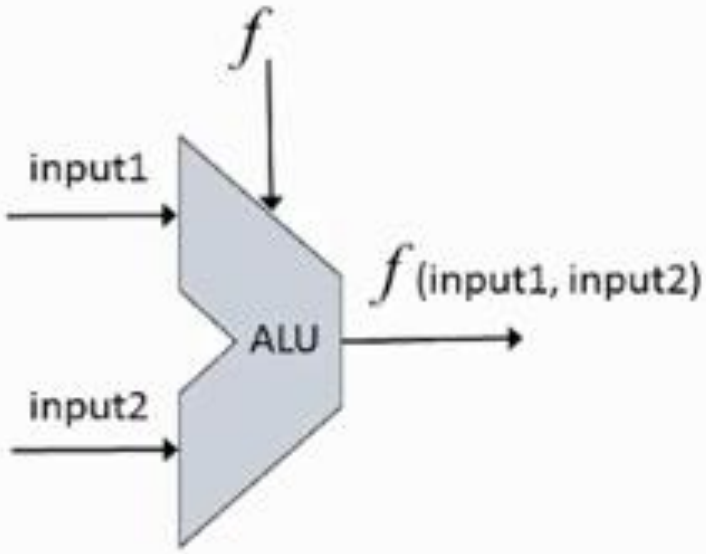
- Multi bit input 1 and multi bit input2



The Arithmetic Logic Unit

The ALU computes a function on the two inputs, and outputs the result

f: one out of a family of pre-defined arithmetic and logical functions

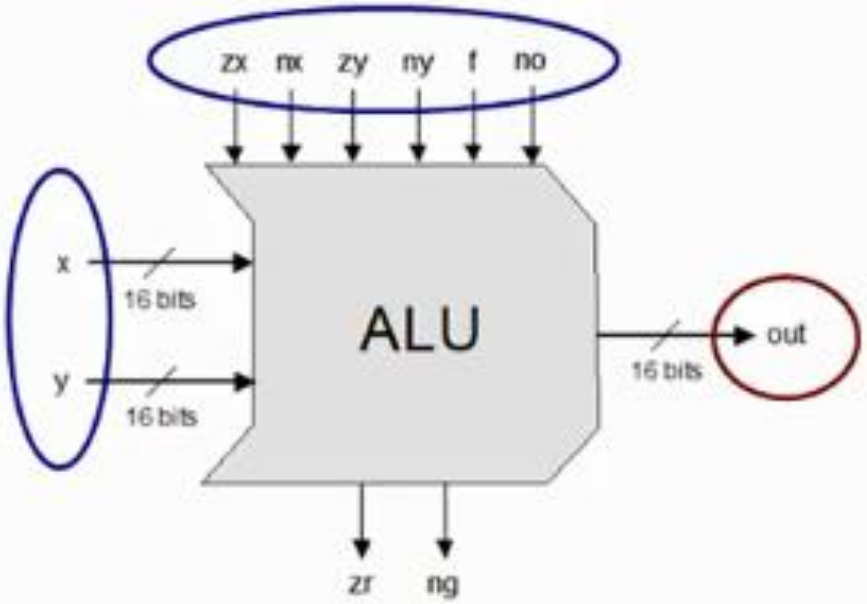


- Arithmetic operations: integer addition, multiplication, division, ...
- logical operations: And, Or, Xor, …

Which operations should the ALU perform?
A hardware / software tradeoff.

The Hack ALU

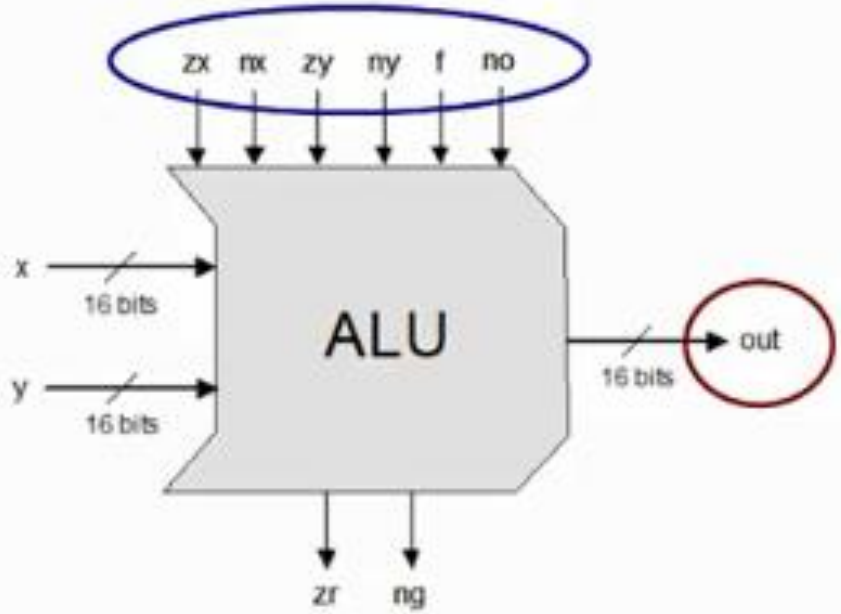
- Operates on two 16-bit, two's complement values
- Outputs a 16-bit, two's complement value
- Which function to compute is set by six 1-bit inputs
- Computes one out of a family of 18 functions



out
0
1
-1
x
y
!x
!y
-x
-y
x+1
y+1
x-1
y-1
x+y
x-y
y-x
x&y
x|y

The Hack ALU

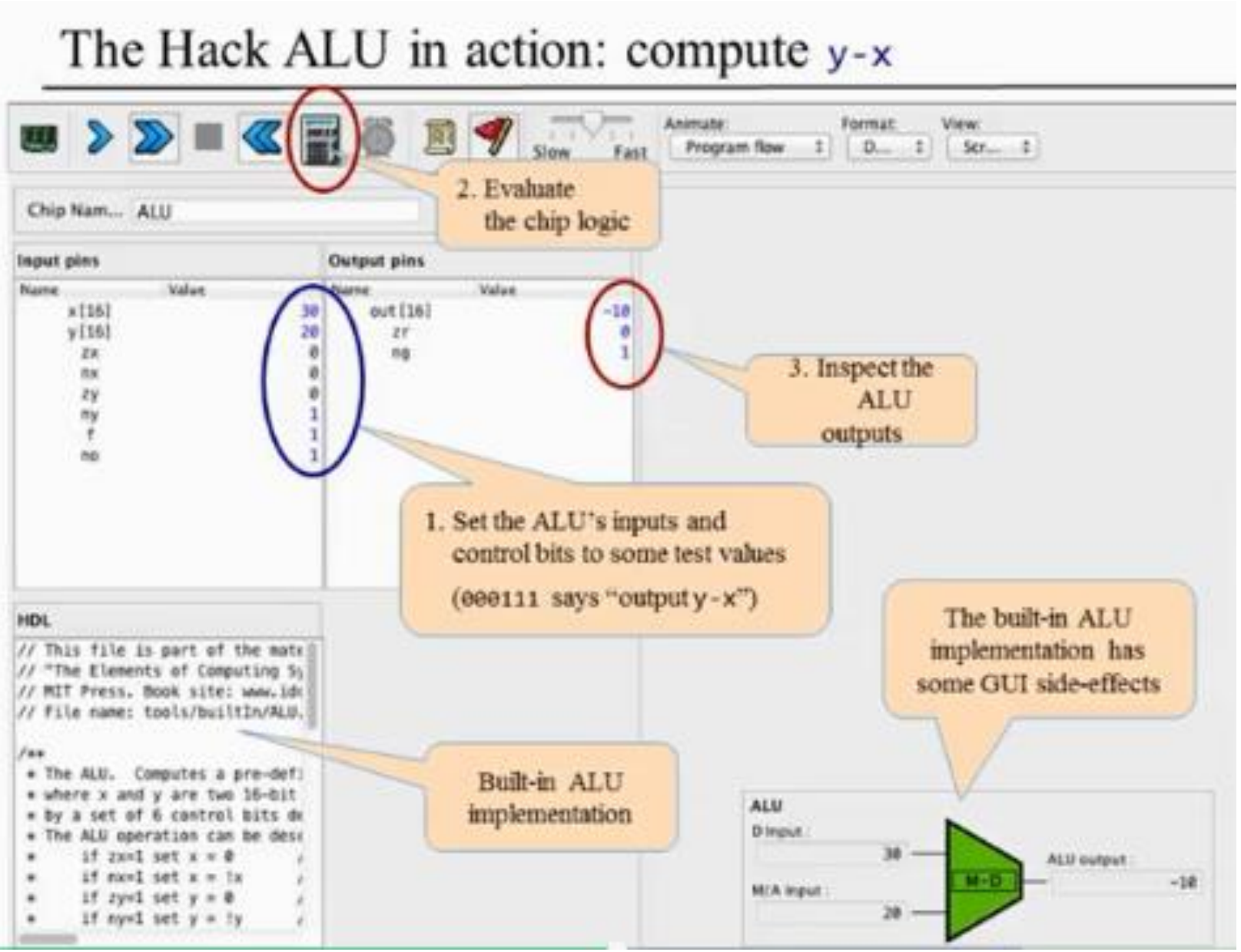
To cause the ALU to compute a function, set the control bits to the binary combination listed in the table.



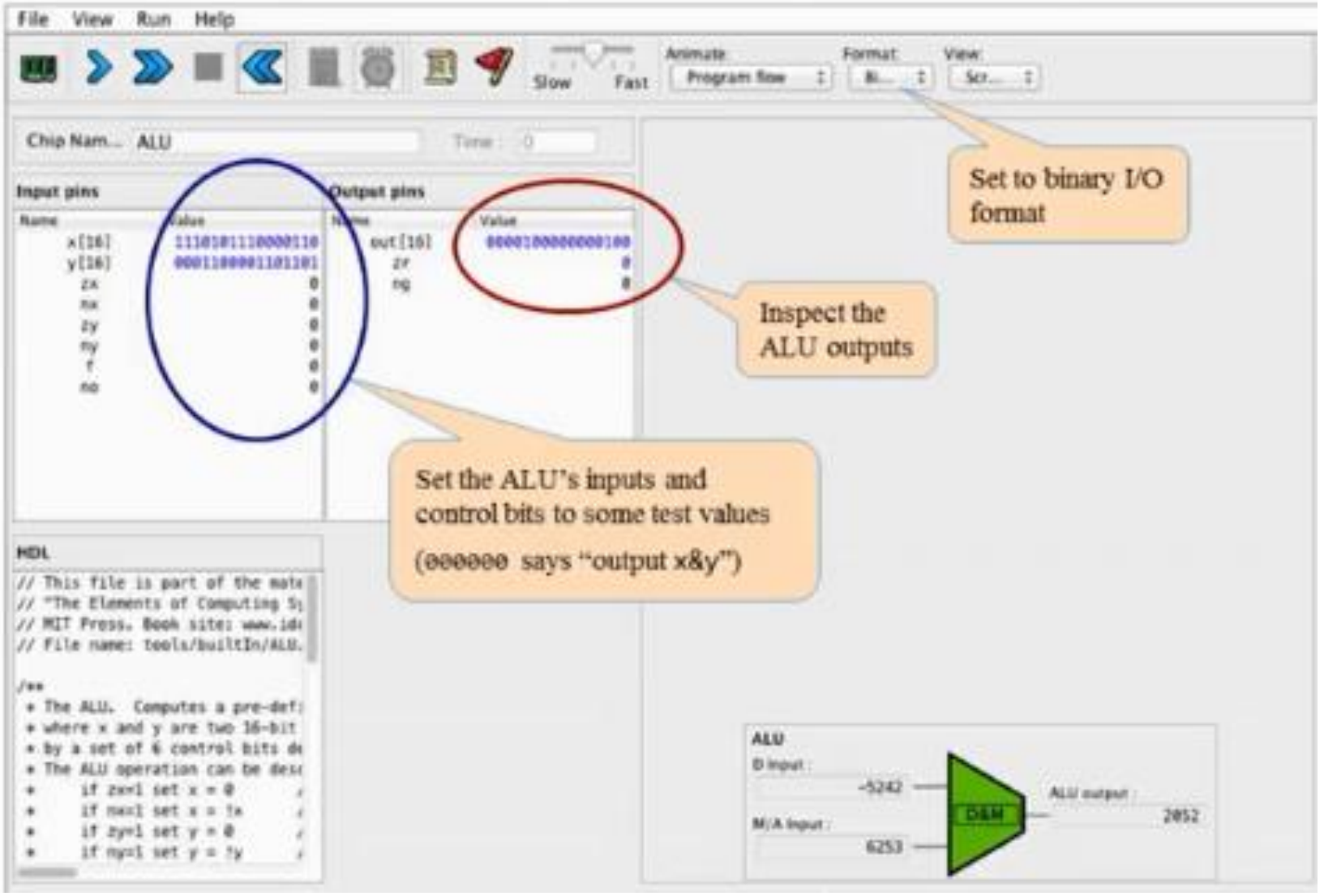
control bits

zx	nx	zy	ny	f	no	out
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

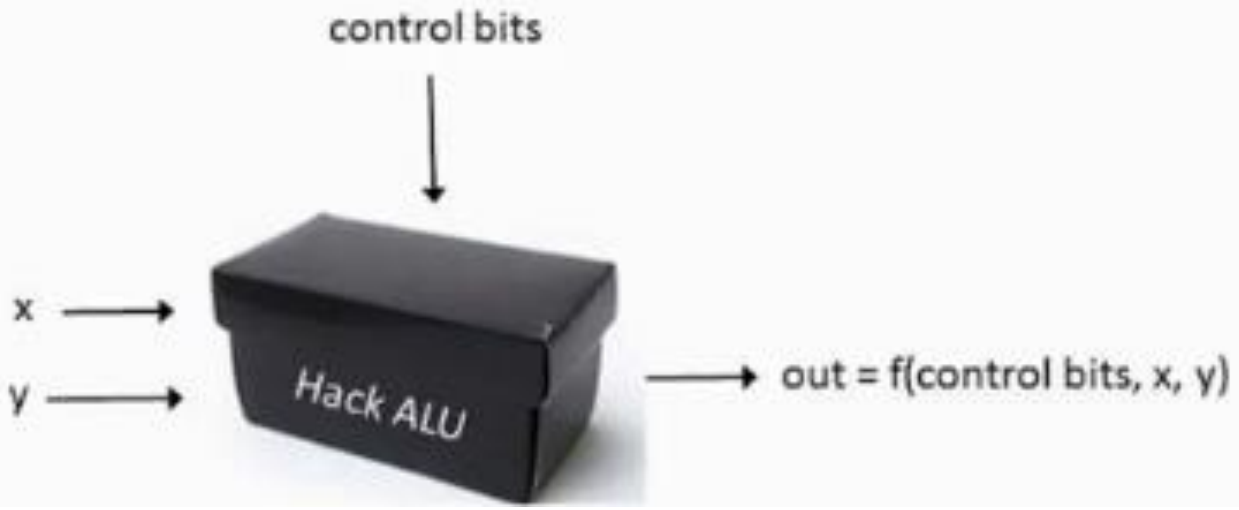
The Hack ALU in action: compute y-x



The Hack ALU in action: compute x & y

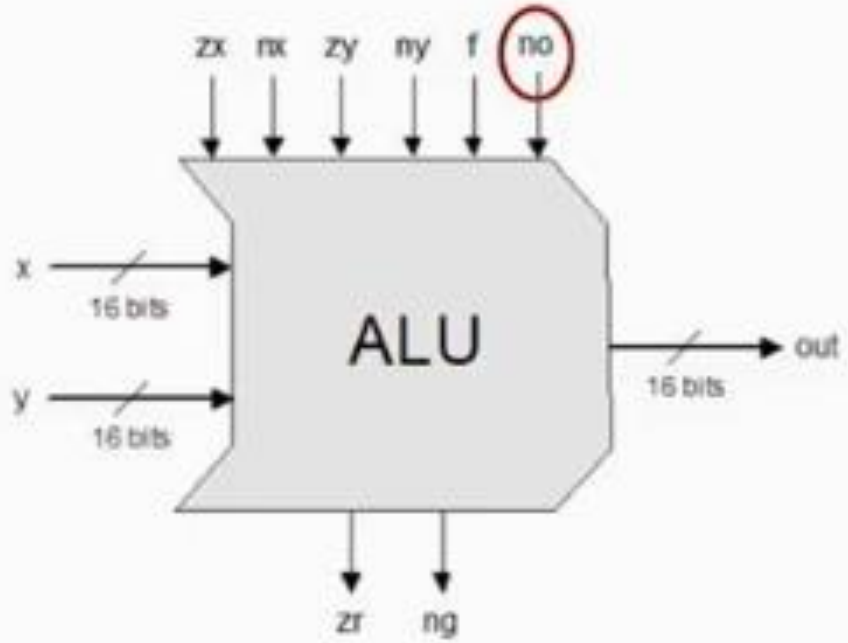


Opening up the Hack ALU black box



The Hack ALU operation

pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the 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	



The Hack ALU operation

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 operation example: compute !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				!y
0	0	1				-x
1	1	0				-y
0	1	1				x+1
1	1	0				y+1
0	0	1				x-1
1	1	0				y-1
0	0	0				x+y
0	1	0				x-y
0	0	0				y-x
0	0	0				x&y
0	1	0				x|y

Example: compute !x

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

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)

ALU operation example: compute 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	0	-1
				0	0	x
				0	0	y
				0	1	!x
				0	1	!y
				1	1	-x
				1	1	-y
				1	1	x+1
				1	1	y+1
				1	0	x-1
				1	0	y-1
				1	0	x+y
				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 0 1 0 (2)
y: 0 1 1 1 (7)

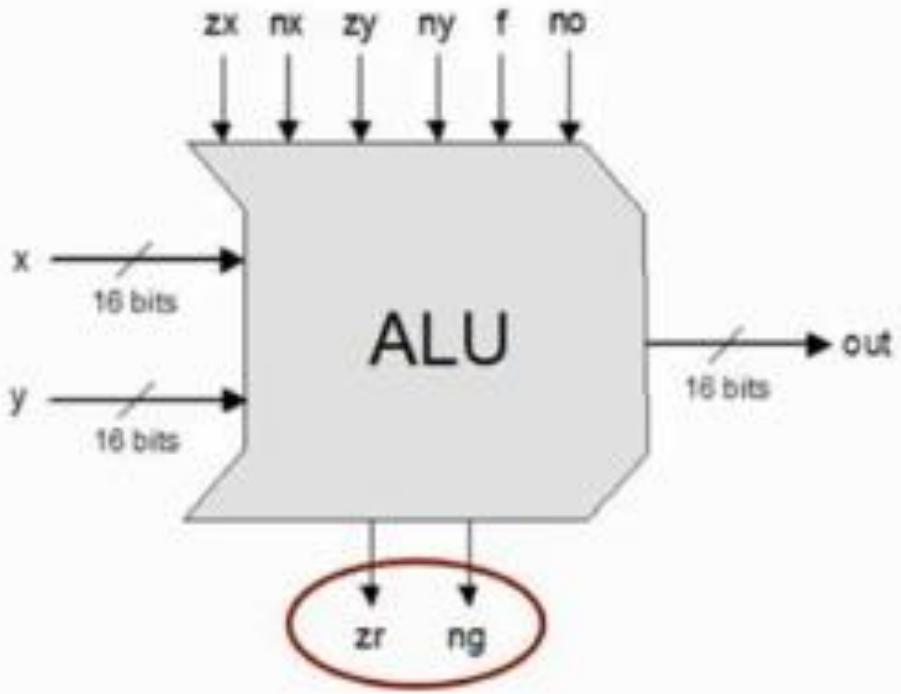
Following pre-setting:

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

Computation and post-setting:

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

The Hack ALU output control bits




```
if out == 0 then zr = 1, else zr = 0

if out < 0  then ng = 1, else ng = 0
```

Basic Circuits that can be used in 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 using two very elementary gates.

What two chips correspond to the sum and carry columns?

(If you can't see all the answers you should scroll down)

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

Basic Circuits that can be used in ALU

Full Adder

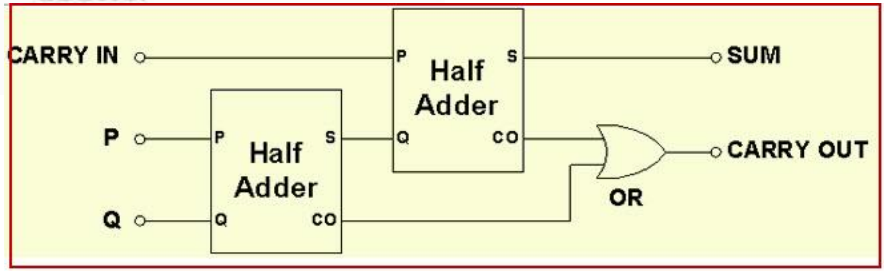
a	b	c	sum	carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

FullAdder.hdl

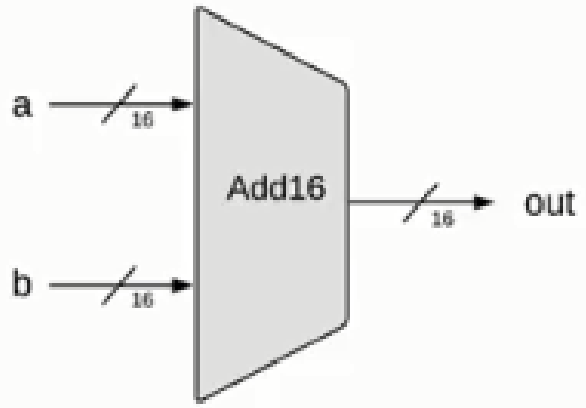
```
/** Computes the sum of three bits. */

CHIP HalfAdder {
    IN a, b, c;
    OUT sum, carry;

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

Implementation tips

Can be built from two half-adders.



16-bit adder

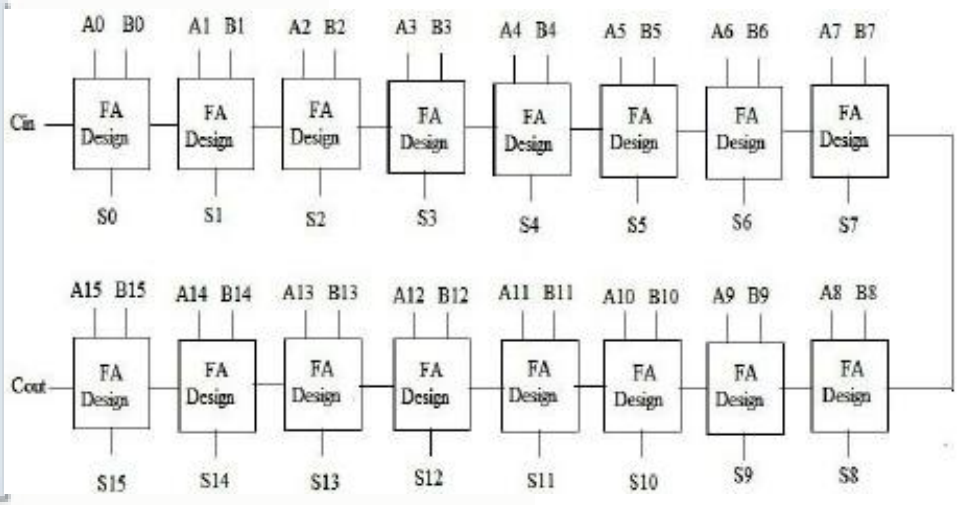


Implementation tips

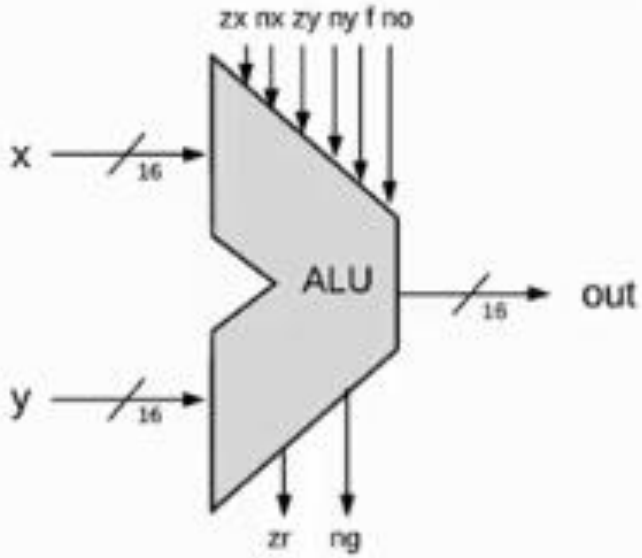
- An *n*-bit adder can be built from *n* full-adder chips
- The carry bit is "piped" from right to left
- The MSB carry bit is ignored.

Add16.hdl

```
/*
 * Adds two 16-bit, 2'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:
}
```



ALU



ALU.hdl

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