Lecture 3 – Boolean Arithmetic

18/10/2024

Dr. Libin HONG

Overview

- Review previous weeks slides
- Review lab session
- Boolean arithmetic
  - Decimal to Binary
  - Binary Addition
  - Adders
  - Negative numbers
  - Binary subtraction

x	y	z	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

(Not(x) And Not(y) And Not(z))

Or

(Not(x) And y And Not(z))

Or

(x And Not(y) And Not(z))

(Not(x) And Not(y) And Not(z)) Or
(Not(x) And y And Not(z)) Or
(x And Not(y) And Not(z)) =

x	y	z	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Reminder

Complement law: **x Or (Not(x))=1**

Distributive law: **x and (y or z) = (x and y) or (x and z)**

(Not(x) And **Not(y)** And Not(z)) Or
(Not(x) And ***y*** And Not(z)) Or
(x And Not(y) And Not(z)) =

(Not(x) And Not(z)) Or (x And Not(y) And Not(z)) =

x	y	z	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Reminder

Complement law: **x Or (Not(x))=1**

Distributive law: **x and (y or z) = (x and y) or (x and z)**

(**Not(*x*)** And Not(*y*) And Not(*z*)) Or
(Not(*x*) And *y* And Not(*z*)) Or
(***x*** And Not(*y*) And Not(*z*)) =

(Not(*x*) And Not(*z*)) Or (*x* And Not(*y*) And Not(*z*)) =

(Not(*x*) And Not(*z*)) Or (Not(*y*) And Not(z)) =

x	y	z	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Reminder

Complement law: **x Or (Not(x))=1**

Distributive law: **x and (y or z) = (x and y) or (x and z)**

(Not(x) And Not(y) And Not(z)) Or
(Not(x) And y And Not(z)) Or
(x And Not(y) And Not(z)) =

(Not(x) And Not(z)) Or (x And Not(y) And Not(z)) =

2 * Complement Law

(Not(x) And Not(z)) Or (Not(y) And Not(z)) =

Distributive law

(Not(x) Or Not(y)) And Not(z)

Lab sessions

- Use the API!

```
Add16 (a= ,b= ,out= );
ALU (x= ,y= ,zx= ,nx= ,zy= ,ny= ,f= ,no= ,out= ,zr= ,ng= );
And16 (a= ,b= ,out= );
And (a= ,b= ,out= );
Aregister (in= ,load= ,out= );
Bit (in= ,load= ,out= );
CPU (inM= ,instruction= ,reset= ,outM= ,writeM= ,addressM= ,pc= );
DFF (in= ,out= );
DMux4Way (in= ,sel= ,a= ,b= ,c= ,d= );
DMux8Way (in= ,sel= ,a= ,b= ,c= ,d= ,e= ,f= ,g= ,h= );
Dmux (in= ,sel= ,a= ,b= );
Dregister (in= ,load= ,out= );
FullAdder (a= ,b= ,c= ,sum= ,carry= );
HalfAdder (a= ,b= ,sum= , carry= );
Inc16 (in= ,out= );
Keyboard (out= );
Memory (in= ,load= ,address= ,out= );
Mux16 (a= ,b= ,sel= ,out= );
Mux4Way16 (a= ,b= ,c= ,d= ,sel= ,out= );
Mux8Way16 (a= ,b= ,c= ,d= ,e= ,f= ,g= ,h= ,sel= ,out= );
```

```
Mux8Way (a= ,b= ,c= ,d= ,e= ,f= ,g= ,h= ,sel= ,out= );
Mux (a= ,b= ,sel= ,out= );
Nand (a= ,b= ,out= );
Not16 (in= ,out= );
Not (in= ,out= );
Or16 (a= ,b= ,out= );
Or8Way (in= ,out= );
Or (a= ,b= ,out= );
PC (in= ,load= ,inc= ,reset= ,out= );
PCLoadLogic (cinstr= ,j1= ,j2= ,j3= ,load= ,inc= );
RAM16K (in= ,load= ,address= ,out= );
RAM4K (in= ,load= ,address= ,out= );
RAM512 (in= ,load= ,address= ,out= );
RAM64 (in= ,load= ,address= ,out= );
RAM8 (in= ,load= ,address= ,out= );
Register (in= ,load= ,out= );
ROM32K (address= ,out= );
Screen (in= ,load= ,address= ,out= );
Xor (a= ,b= ,out= );
```

Nand, Not, And, Or

x	y	Nand
0	0	1
0	1	1
1	0	1
1	1	0

```
CHIP Nanda {
    IN a, b;
    OUT out;

    PARTS:
    And (a=a, b=b, out=x);
    Not  (in=x, out=out);
}
```

```
CHIP Not {
    IN in;
    OUT out;

    PARTS:
    Nand (a=in, b=in, out=out);
}
```

(*x* Nand *y*) = Not(*x* And *y*)

```
CHIP And {
    IN a, b;
    OUT out;

    PARTS:
    Nand (a=a, b=b, out=x);
    Not  (in=x, out=out);
}
```

- Not(x) = (x Nand x)
- (x And y) = Not(x Nand y)
- (x Or y) = Not(Not(x) And Not(y))

```
CHIP Ora {
    IN a, b;
    OUT out;

    PARTS:
    Not  (in=a, out=nota);
    Not  (in=b, out=notb);
    And (a=nota, b=notb, out=anda);
    Not (in=anda, out=out);
}
```

```
CHIP Or {
    IN a, b;
    OUT out;

    PARTS:
    Not  (in=a, out=nota);
    Not  (in=b, out=notb);
    Nand (a=nota, b=notb, out=out);
}
```

Lab sessions

- There are inbuilt versions of all chips (these work)
- If a chip with the same name exists in the working directory (eg. And) you HDL code will use that instead of the inbuilt one
  - It will try and use this chip even if it is wrong or empty
- Leads to some interesting situations
  - Eg. Never try and build your own Nand chip
  - Nand uses And, And uses Nand - infinite cycle (Loading Chip…..)
- HDL does allow looping BUT don’t use it (inefficient in real chips)

Designing Logic Gates - Reminder

- Helpful to start by building up the truth table
- Then work out the equations for each output for each line based on the input
- *OR* each of these together for each output (or multiple outputs)
- Then simplify the equations
  - Use the algebraic rules,
  - Look for common sub-equations
  - Use as few gates as possible
  - Saves money/Reduces delay*

x	y	z	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

(Not(x) And Not(y) And Not(z)) Or

(Not(x) And y And Not(z)) Or

(x And Not(y) And Not(z))

(Not(x) And Not(y) And Not(z)) Or
(Not(x) And y And Not(z)) Or
(x And Not(y) And Not(z)) =

* Propagation delay is the time taken for a change in the inputs to be reflected in the output

Simplifying Truth tables (eg Mux)

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

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

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

    PARTS:
    Not (in=sel, out=notSel);
    And (a=notSel, b=a, out=and1);
    And (a=sel, b=b, out=and2);
    Or  (a=and1, b=and2, out=out);
}
```

Boolean Arithmetic (based on nand2tetris chapter 2)

- INTRODUCTION

  - Given a set of NAND gates, we can design any logic circuit we want
  - Can get programmable logic chips that are just arrays of Nand gates
  - Easier to design using And, Or and Not gates
  - Only Two output states — True or False, 0 or 1
  - Can describe logic circuits using a *Hardware Description Language*
  - Express how the various AND, OR and NOT gates connect

  - But how do we represent a number with just two states?

Numbers

- Various symbols have been used over the ages to represent numbers
  - Roman numerals - XX
  - Arabic numbers (as we use today) - 20
- All encode a quantity
- We use the decimal system for counting
  - Based around 10 because we have ten fingers
  - But we also have counting systems using other bases
  - Time, eggs…..
- Computers just use a different encoding using just two symbols
  - (computers only have 2 fingers)

Decimal vs Binary Counting

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

For Binary, Maximum value represented by *k* bits:

$$1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1$$

Representing numbers

10^2	10^1	10^0
100s	10s	1s

$$7\ 8\ 9_{\text{ten}}$$

Representing numbers

10^2	10^1	10^0
100s	10s	1s

$$7\ 8\ 9_{\text{ten}}$$

$$= 7 \cdot 10^2 + 8 \cdot 10^1 + 9 \cdot 10^0 = 789_{\text{ten}}$$

Binary → Decimal

2^2	2^1	2^0
4s	2s	1s

$1\ 0\ 1_{two}$

Binary → Decimal

2^2	2^1	2^0
4s	2s	1s

$$1\ 0\ 1_{two}$$

$$= 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5_{ten}$$

Fixed word size

We will use a fixed number of bits.

Say 8 bits.

```
0000 0000
0000 0001
0000 0010
0000 0011
…
0111 1111
1000 0000
1000 0001
…
1111 1110
1111 1111
```

$2^8 = 256$ values

Representing signed numbers

We will use a fixed number of bits.

Say 8 bits.

```
0000 0000
0000 0001
0000 0010
0000 0011
…
0111 1111
1000 0000
1000 0001
…
1111 1110
1111 1111
```

positive values

negative values

Representing signed numbers

We will use a fixed number of bits.

Say 8 bits.

```
0000 0000
0000 0001
0000 0010
0000 0011
…
0111 1111
1000 0000
1000 0001
…
1111 1110
1111 1111
```

positive values

negative values

- That's one possible representation
- We'll use a better one, later

Decimal → Binary

87_{ten}

Decimal → Binary

87_{ten} = ? ? ? ? ? ? ? ? $_{two}$

Decimal → Binary

$87_{ten} = 64$ (64 = 2^6, the biggest 2^n that 87 is divisible by)

remainder 23

Decimal → Binary

$87_{ten} = 64 + 16$ (16 = 2^4, the biggest 2^n that 23 is divisible by)

remainder 7

Decimal → Binary

$87_{ten} = 64 + 16 + 4$ (4 = 2^2, the biggest 2^n that 7 is divisible by)

remainder 3

Decimal → Binary

$87_{ten} = 64 + 16 + 4 + 2$ (2 = 2^1, the biggest 2^n that 7 is divisible by)

remainder 1

Decimal → Binary

$87_{ten} = 64 + 16 + 4 + 2 + 1$ (1 = 2^0, the biggest 2^n that 1 is divisible by)

remainder 0 (stop when remainder = 0)

Decimal → Binary

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

$= ?\ ?\ ?\ ?\ \ ?\ ?\ ?\ ?_{two}$

Decimal → Binary

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

$= 2^6 + 2^4 + 2^2 + 2^1 + 2^0$

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

= 01010111

Decimal → Binary

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

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

32 8

Binary Numbers

- Often written out with leading zeros
  - Up to a certain number of bits — usually a multiple of eight
  - So 42 is usually written as 00101010
  - Might also see it written as 0b00101010 to signify it is binary
- Octal and Hexadecimal often used to compress binary
  - Octal is a base-8 system, Hexadecimal is base-16
  - Both of these are powers of two
- Each octal digit equates to three consecutive bits of a binary number
  - Useful for 3 bit adders
- Each hex digit equates to four consecutive bits
- Binary 00111011 | Decimal 59 |Hex 3B | Octal 73

BINARY	HEXADECIMAL	OCTAL	DECIMAL
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	10	8
1001	9	11	9
1010	A	12	10
1011	B	13	11
1100	C	14	12
1101	D	15	13
1110	E	16	14
1111	F	17	15

Binary Addition

- Challenge - Build a circuit to add two binary numbers together
- First let’s recap how addition works
  - Add each column together from right
  - If bigger than 9, we carry over into the next column
  - Binary addition is the same, except we carry if the value is greater than one

Addition

Addition

```
  0 0 0 1 0 1 0 1
+ 0 1 0 1 1 1 0 0
  ---------------
  ? ? ? ? ? ? ? ?
```

Addition

```
  0 0 0 1 0 1 0 1        21
+ 0 1 0 1 1 1 0 0      + 92
  -----------------
```

Addition

```
  0 0 0 1 0 1 0 1
+ 0 1 0 1 1 1 0 0
-----------------
```

```
  21
+ 92
----
 113
```

Addition

```
  0 0 0 1 0 1 0 1        21
+ 0 1 0 1 1 1 0 0      +  92
  -----------------     ----
  0 1 1 1 0 0 0 1       113
```

0+64+32+16+0+0+0+1 = 113

Decimal Addition

Decimal Addition

$$
\begin{array}{r}
5\ 7\ 8\ 3 \\
+\ 2\ 4\ 5\ 6 \\
\hline
?\ ?\ ?\ ?
\end{array}
$$

Decimal Addition

$$\begin{array}{r} 5\ 7\ 8\ 3 \\ +\ 2\ 4\ 5\ 6 \\ \hline \end{array}$$

Decimal Addition

$$
\begin{array}{r}
5\ 7\ 8\ 3 \\
+\ 2\ 4\ 5\ 6 \\
\hline
9
\end{array}
$$

Decimal Addition

```
    1
  5 7 8 3
+ 2 4 5 6
---------
      3 9
```

Decimal Addition

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

Decimal Addition

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

Binary Addition

Binary Addition

```
  0 0 0 1 0 1 0 1
+ 0 1 0 1 1 1 0 0
  ---------------
  ? ? ? ? ? ? ? ?
```

Binary Addition

```
  0 0 0 1 0 1 0 1
+ 0 1 0 1 1 1 0 0
  ---------------
```

Binary Addition

```
  0 0 0 1 0 1 0 1
+ 0 1 0 1 1 1 0 0
  ---------------
                1
```

Binary Addition

```
  0 0 0 1 0 1 0 1
+ 0 1 0 1 1 1 0 0
  ---------------
            0 1
```

Binary Addition

```
          1
  0 0 0 1 0 1 0 1
+ 0 1 0 1 1 1 0 0
  ---------------
            0 0 1
```

Binary Addition

```
        1 1
  0 0 0 1 0 1 0 1
+ 0 1 0 1 1 1 0 0
  ---------------
          0 0 0 1
```

Binary Addition

```
      1 1 1
  0 0 0 1 0 1 0 1
+ 0 1 0 1 1 1 0 0
  ---------------
        1 0 0 0 1
```

Binary Addition

```
      1 1 1
  0 0 0 1 0 1 0 1
+ 0 1 0 1 1 1 0 0
  -----------------
      1 1 0 0 0 1
```

Binary Addition

```
      1 1 1
  0 0 0 1 0 1 0 1
+ 0 1 0 1 1 1 0 0
  -----------------
    1 1 1 0 0 0 1
```

Binary Addition

```
      1 1 1
  0 0 0 1 0 1 0 1
+ 0 1 0 1 1 1 0 0
  -----------------
  0 1 1 1 0 0 0 1
```

Overflow

Overflow

```
  1 0 0 1 0 1 0 1
+ 1 1 0 1 1 1 0 0
  ---------------
  ? ? ? ? ? ? ? ?
```

Overflow

```
1 0 0 0 1 1 1 0 0
  1 0 0 1 0 1 0 1      128+16+4+1          =149
+
  1 1 0 1 1 1 0 0      128+64+16+8+4       =220
  -----------------
1 0 1 1 1 0 0 0 1                          =369
```

Building an Adder

- Half adder: adds two bits
- Full adder: adds three bits
- Adder: adds two integers

Half adder

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

carry bit

```
  0 0 1 0 1 1 0
+ 1 0 0 1 0 1 1 1
  0 1 0 1 0 0 1 0
  -----------------
  1 1 1 0 1 0 0 1
```

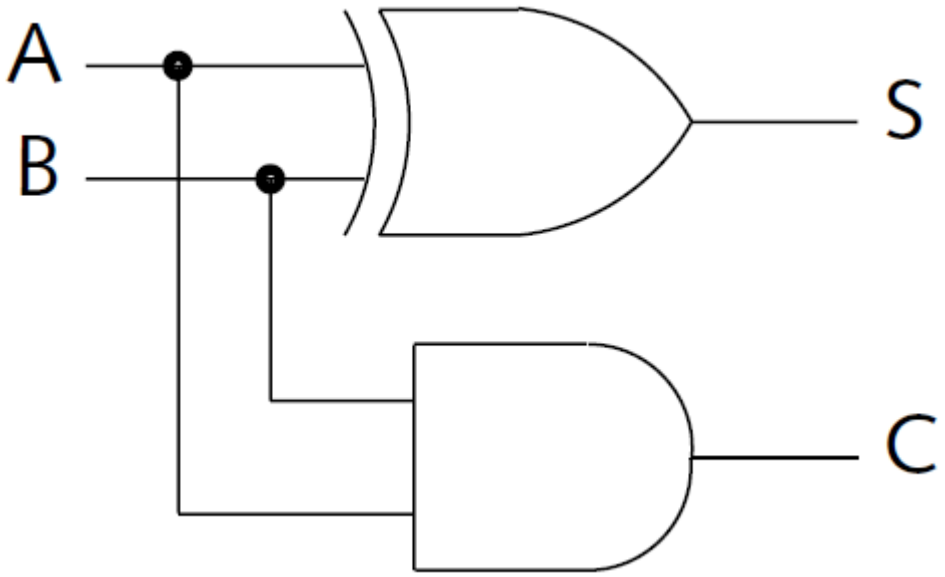
sum bit

Half Adder never has a situation where sum and carry are both 1

Half-Adder

- Each column takes in two input bits
- And produces a sum bit and a carry bit

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1
		(Xor)	(And)



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

carry bit

```
  0 0 1 0 1 1 0
+ 1 0 0 1 0 1 1 1
  0 1 0 1 0 0 1 0
  ---------------
  1 1 1 0 1 0 0 1
```

sum bit

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

Chip Name : FullAdder Time : 0

Input pins

Name	Value
a	0
b	0
c	0

Output pins

Name	Value
sum	0
carry	0

HDL

```
// This file is part of www.nanc
// and the book "The Elements of
// by Nisan and Schocken, MIT Pr
// File name: projects/02/FullAd

/**
 * Computes the sum of three bit
 */

CHIP FullAdder {
    IN a, b, c;  // 1-bit inputs
    OUT sum,     // Right bit of
        carry;   // Left bit of
```

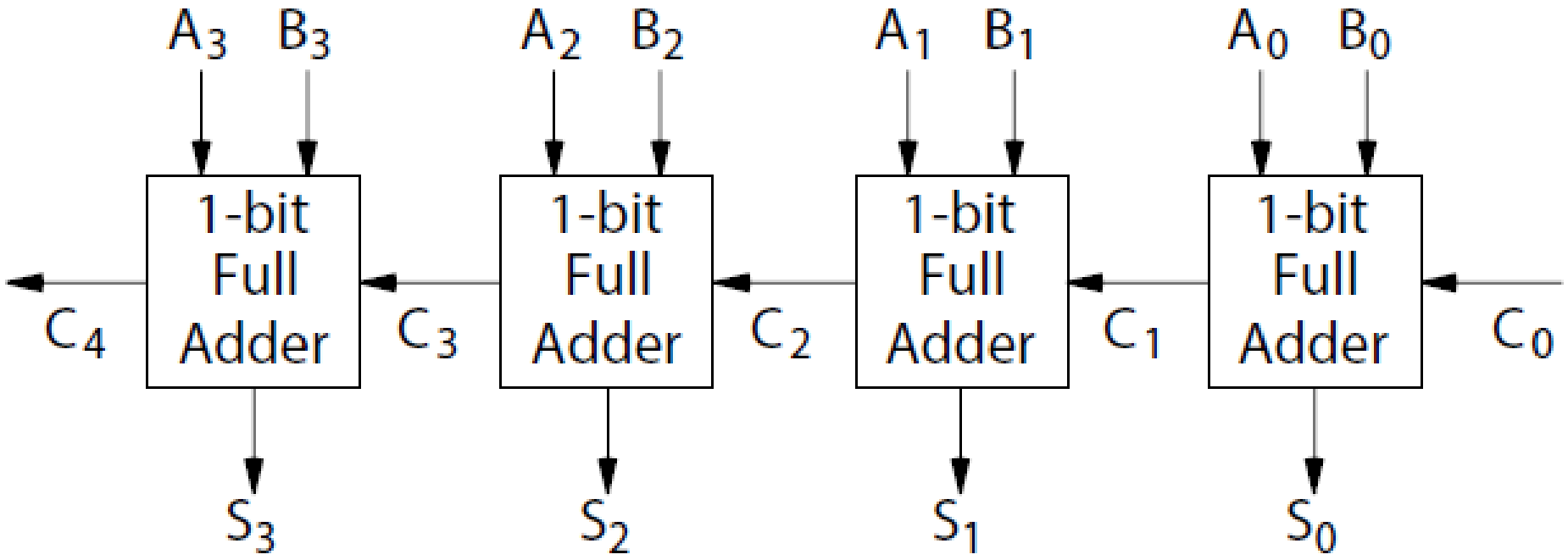
Internal pins

Name	Value
sum1	0
carry1	0
carry2	0

Full-Adder – Truth Table

C_{in}	A	B	S	C_{out}
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

Ripple Carry 4 bit Adder



Subtraction

- Can design a logic circuit to perform the subtraction
- Half-subtractor and full-subtractors
- But another way – subtraction by addition
- 4096-2048 is the same as 4096 + (-2048)
- How to represent a negative number in binary?

Representing negative numbers in binary

- Four approaches
  - Sign and Magnitude
    - Use top bit to represent sign (just as we do in decimal)
    - Has 2 zeroes
    - Addition is ‘tricky’
  - One’s complement
    - Negative values are inverted (Not-ed)
    - Has two zeroes, but addition now works as with unsigned numbers (more or less)
  - Excess-*n*
    - Add a fixed offset to all values
  - Two’s complement
    - Very similar to one’s complement but we invert negative numbers and add one
    - Only one zero, but we have one more negative number than positive
    - Addition same as with unsigned numbers

4-BIT SIGNED ENCODINGS

Sign and Magnitude		Excess-3		One's Complement		Two's Complement	
1111	-7	0000	-3	1000	-7	1000	-8
1110	-6	0001	-2	1001	-6	1001	-7
1101	-5	0010	-1	1010	-5	1010	-6
1100	-4	0011	0	1011	-4	1011	-5
1011	-3	0100	+1	1100	-3	1100	-4
1010	-2	0101	+2	1101	-2	1101	-3
1001	-1	0110	+3	1110	-1	1110	-2
1000	-0	0111	+4	1111	-0	1111	-1
0000	+0	1000	+5	0000	+0	0000	0
0001	+1	1001	+6	0001	+1	0001	+1
0010	+2	1010	+7	0010	+2	0010	+2
0011	+3	1011	+8	0011	+3	0011	+3
0100	+4	1100	+9	0100	+4	0100	+4
0101	+5	1101	+10	0101	+5	0101	+5
0110	+6	1110	+11	0110	+6	0110	+6
0111	+7	1111	+12	0111	+7	0111	+7

Representing negative numbers

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Possible solution: sign and magnitude

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

Use the left-most bit to represent the sign, -/+

Use the remaining bits to represent a positive number

<u>Complications:</u>

- -0?
- $x + (-x)$ is not 0
- more complications

Two's Complement

```
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
```

Represent the negative number $-x$ using the positive number $2^n - x$

Two's Complement

0000	0	
0001	1	
0010	2	
0011	3	
0100	4	
0101	5	
0110	6	
0111	7	
1000	-8	(16 - 8)
1001	-7	(16 - 9)
1010	-6	(16 - 10)
1011	-5	(16 - 11)
1100	-4	(16 - 12)
1101	-3	(16 - 13)
1110	-2	(16 - 14)
1111	-1	(16 - 15)

Represent the negative number $-x$ using the positive number $2^n - x$

Note - we use 2^n when the biggest value available is $2^n - 1$

Two's Complement

0000	0	
0001	1	
0010	2	
0011	3	
0100	4	
0101	5	
0110	6	
0111	7	
1000	-8	(16 - 8)
1001	-7	(16 - 9)
1010	-6	(16 - 10)
1011	-5	(16 - 11)
1100	-4	(16 - 12)
1101	-3	(16 - 13)
1110	-2	(16 - 14)
1111	-1	(16 - 15)

positive numbers range:

$0 \ldots 2^{n-1} - 1$

negative numbers range:

$-1 \ldots -2^{n-1}$

Computing -*x*

Input: x

Output: $-x$ (in two's complement)

Insight: if we solve this we'll know how to subtract:

$$y - x = y + (-x)$$

By product is that:

We get out subtracter for free

We get our comparator for free

Two’s complement

- To get the negative version of a number
  - Invert the bits
  - Add 1

- So, if we want -29
  - 29 = 0001 1101
  - Invert = 1110 0010
  - Add 1 = 1110 0011
- So, if we want -30
  - 30 = 0001 1110
  - Invert = 1110 0001
  - Add 1 = 1110 0010

Example

- 30-29
- 30+ (-29)

- 30 = 0001 1110
- -29 = 1110 0011
- ----------

  = 0000 0001 = 1

Example

- 29-30
- 29+ (-30)

- 29 = 0001 1101
- -30 = 1110 0010

  = 1111 1111 = -1

Example (4 bits)

- -3 -4
- -3 +(-4)

- -3 = 1101
- -4 = 1100

  = 1001

Computing -*x*

Input: x

Output: $-x$ (in two's complement)

Idea: $2^n - x = 1 + (2^n - 1) - x$

11111111_{two}

$$\begin{array}{r} 11111111 \\ -\ 10101100 \\ \hline \end{array}$$ (some x example)

Computing -*x*

Input: x

Output: $-x$ (in two's complement)

Idea: $2^n - x = 1 + (2^n - 1) - x$

11111111_{two}

```
  11111111
- 10101100   (some x example)
  --------
  01010011
```

Computing -*x*

Input: x

Output: $-x$ (in two's complement)

Idea: $2^n - x = 1 + (2^n - 1) - x$

11111111_{two}

```
  11111111
- 10101100   (some x example)
  --------
  01010011   (flip all the bits)
```

Now add 1 to the result

Computing -*x* (example)

Input: 4

Output: should be 12 (representing -4 in two's compelemt)

Input: `0100`

Flip the bits: `1011`

Computing -*x* (example)

Input: 4

Output: should be 12 (representing -4 in two's compelemt)

Input:	`0100`
Flip the bits:	`1011`
Add one:	`+ 1`

Computing -*x* (example)

Input: 4

Output: should be 12 (representing -4 in two's compelemt)

Input:	`0100`
Flip the bits:	`1011`
Add one:	`+ 1`
Output:	`1100`

Computing -*x* (example)

Input: 4

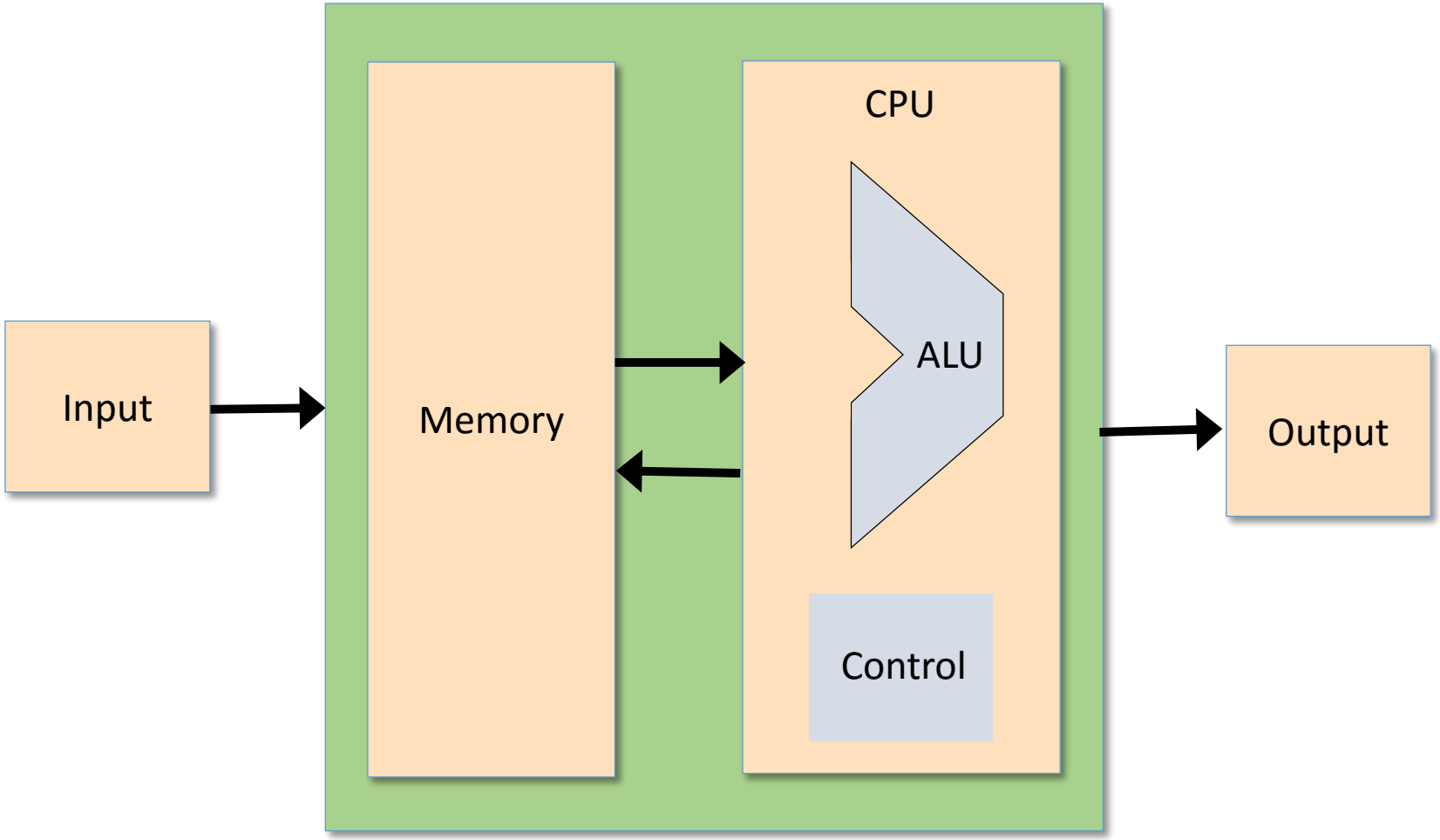
Output: should be 12 (representing -4 in two's compelemt)

Input:	`0100`
Flip the bits:	`1011`
Add one:	`+ 1`
Output:	`1100`
	= 12_{ten}

To add 1:

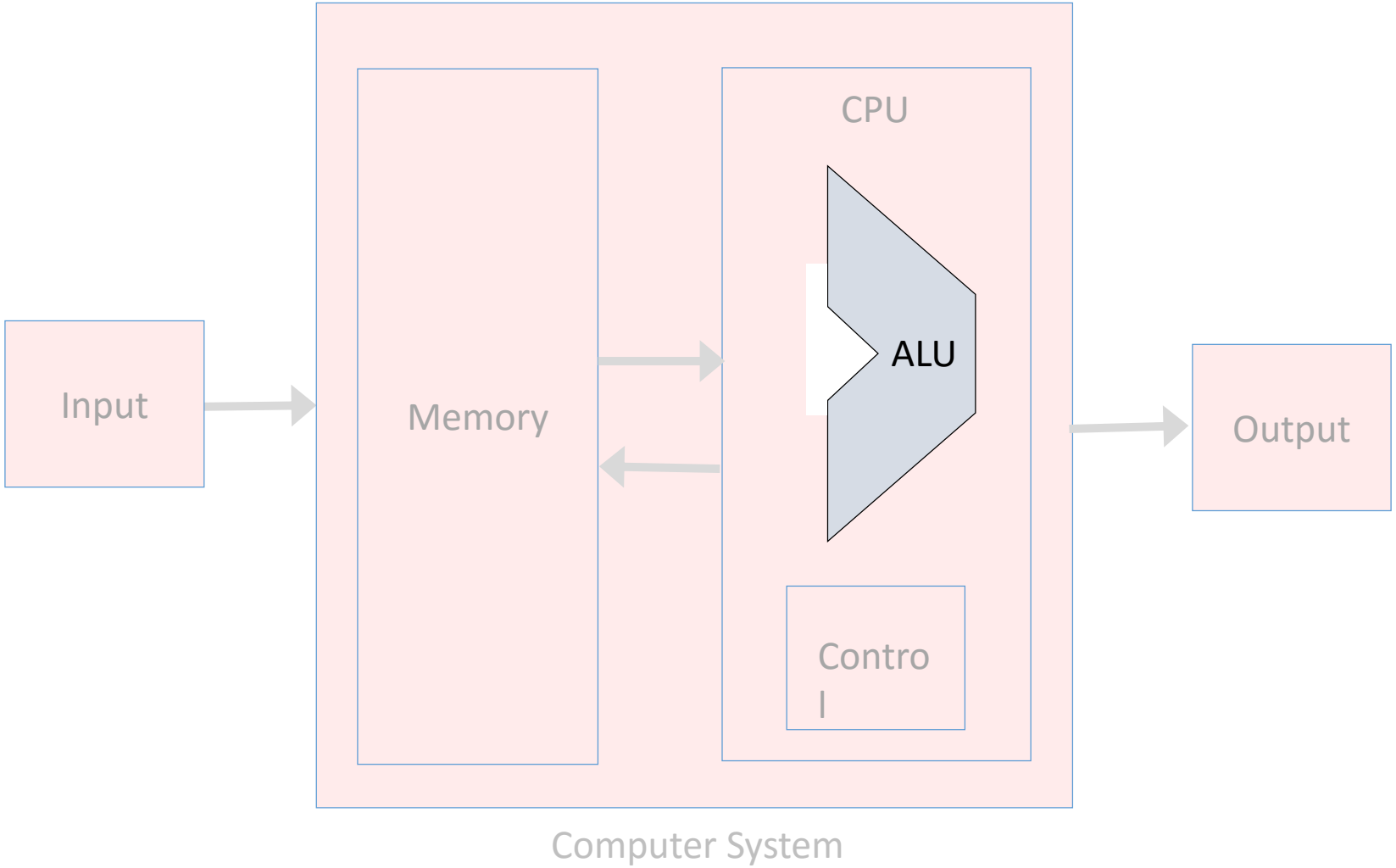
Flip all the bits from right to left, stop when the first 0 flips to 1

Von Neumann Architecture



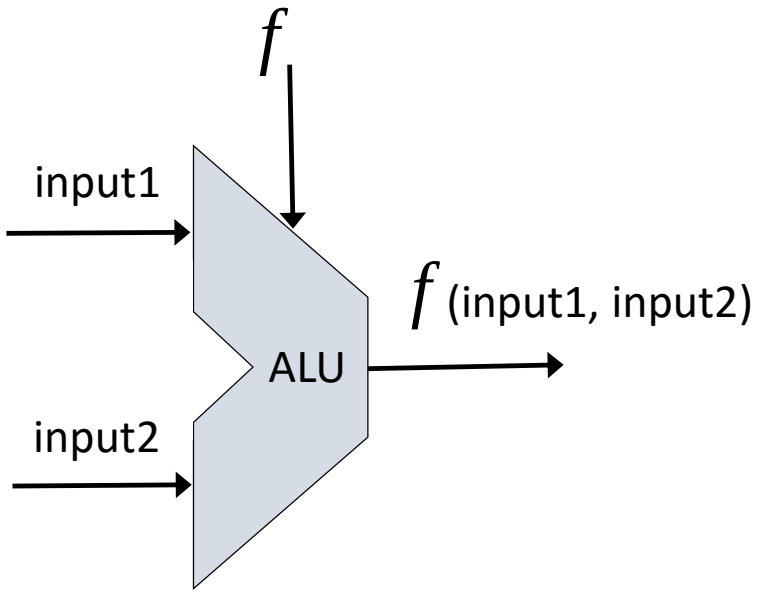
Computer System

The Arithmetic Logical Unit



The Arithmetic Logical Unit

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



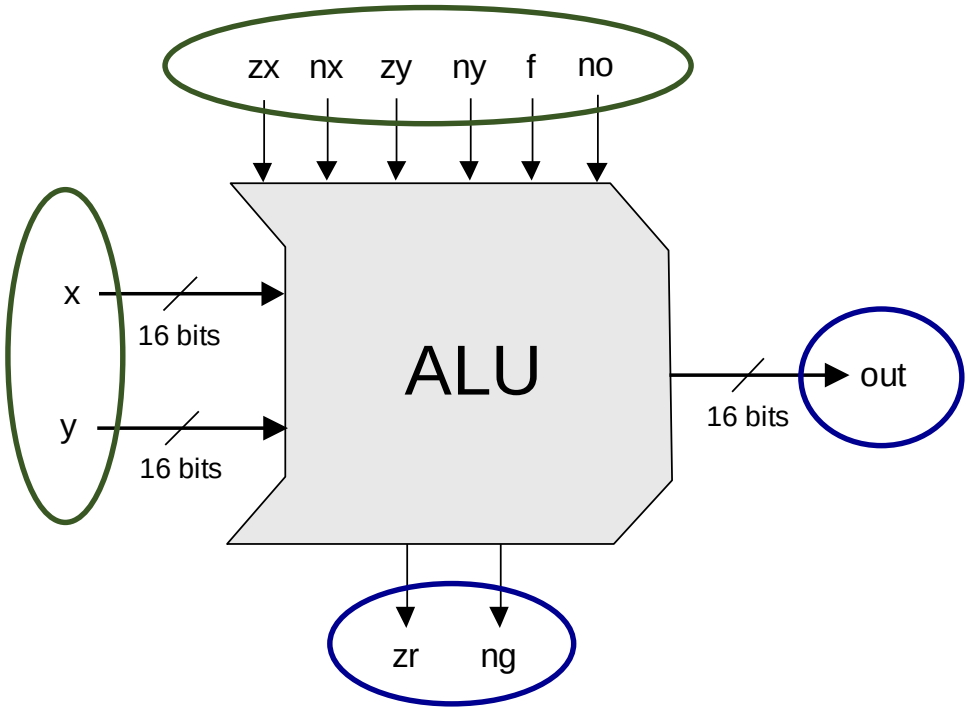
f : one out of a family of pre-defined arithmetic and logical functions

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

Which functions should the ALU perform?
A hardware / software tradeoff. (RISC – why graphics cards work)

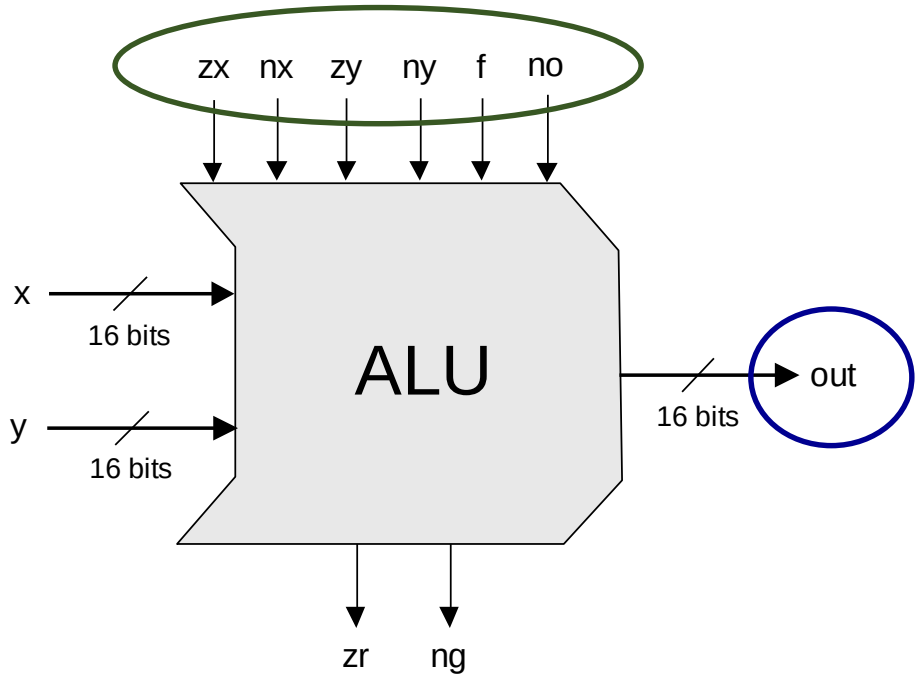
The Hack ALU

- 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
- Also outputs two 1-bit values (to be discussed later).



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

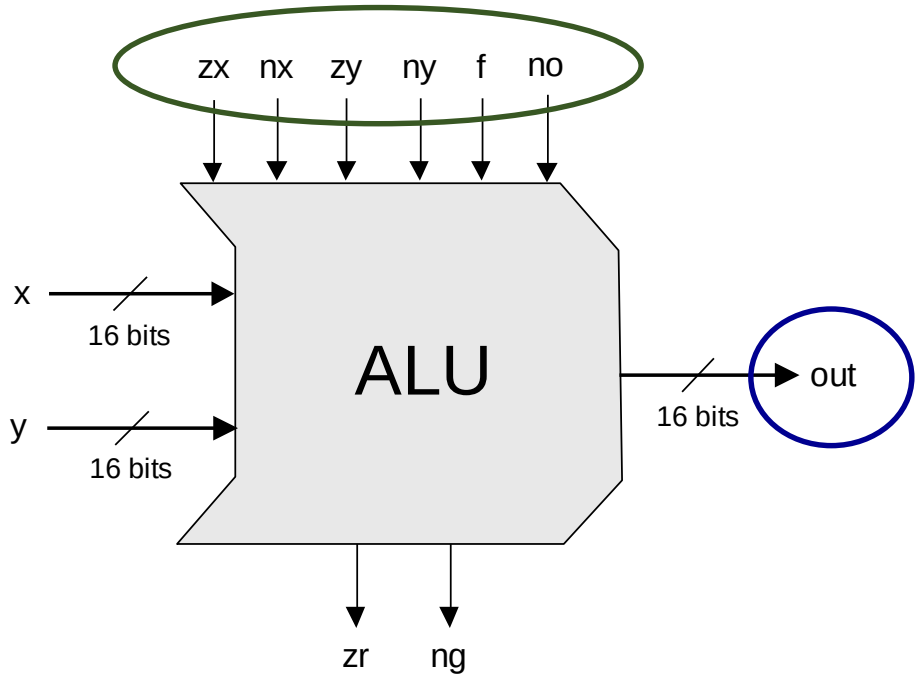
The Hack ALU



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 one of the binary combinations 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