

Chapter 2

Boolean Arithmetic

These slides support chapter 2 of the book

The Elements of Computing Systems

By Noam Nisan and Shimon Schocken

MIT Press

Usage Notice

You are welcome to use this presentation, or any part of it, in Nand to Tetris courses, or in other courses, as you see fit.

Usage is free provided that it supports instruction in an educational, non-profit setting.

Feel free to use the slides as-is, or modify them as needed.

We'll appreciate it if you will give credit somewhere to Nand to Tetris, and put a reference to www.nand2tetris.org

You are welcome to remove this slide from the presentation. If you make extensive changes to the slides, you can remove the copyright notice also.

Happy teaching!

Noam Nisam / Shimon Schocken

Chapter 2: Boolean arithmetic



Binary numbers

- Binary addition
- Negative numbers
- Arithmetic Logic Unit
- Project 2 overview

| | | | |
|----|----|----|----|
| | 0 | 1 | |
| 00 | 01 | 10 | 11 |

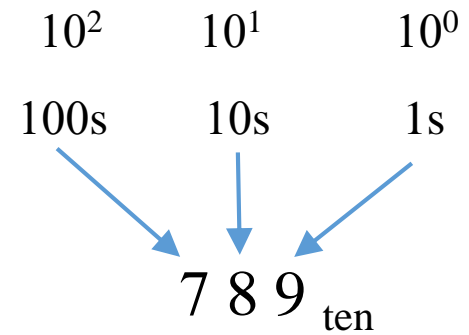
3 bits – 8 possibilities

N bits – 2^N possibilities

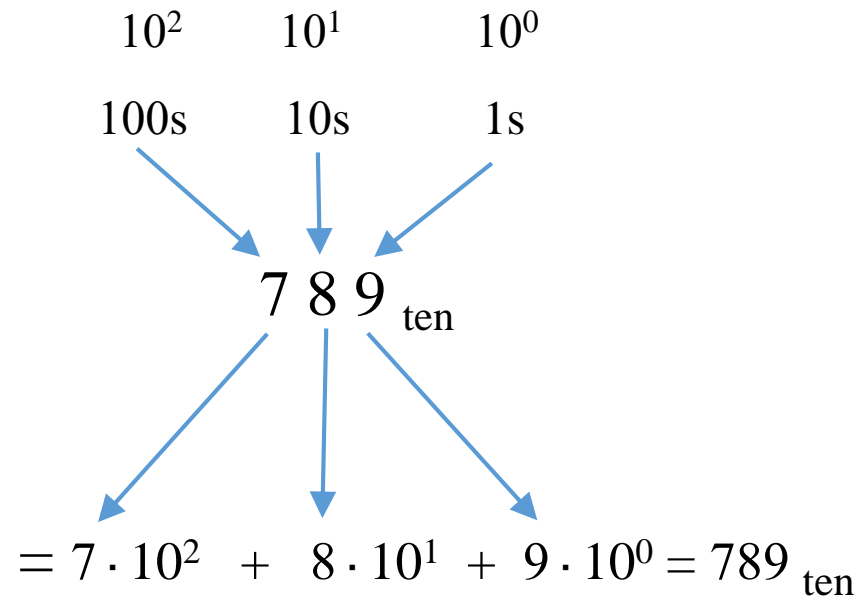
Representing numbers

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

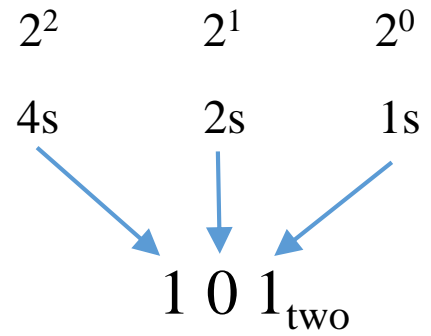
Representing numbers



Representing numbers



Binary \rightarrow Decimal



Binary \rightarrow Decimal

Diagram illustrating the conversion of binary 101_{two} to decimal 5_{ten} :

The binary digits are weighted by powers of 2:

- 2^2 (4s) for the first digit (1)
- 2^1 (2s) for the second digit (0)
- 2^0 (1s) for the third digit (1)

The calculation is shown as:

$$= 1 \cdot 10^2 + 0 \cdot 10^1 + 1 \cdot 10^0 = 5_{\text{ten}}$$

Binary → Decimal

Binary \rightarrow Decimal

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

Binary \rightarrow Decimal

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

$$= \sum_{i=0}^n b_i \cdot 2^i$$

Binary \rightarrow Decimal

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

$$= \sum_{i=0}^n b_i \cdot 2^i$$

Maximum value represented by k bits:

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

Fixed word size

Fixed word size


We will use a fixed number of bits.

Say 8 bits.

Fixed word size

We will use a fixed number of bits.

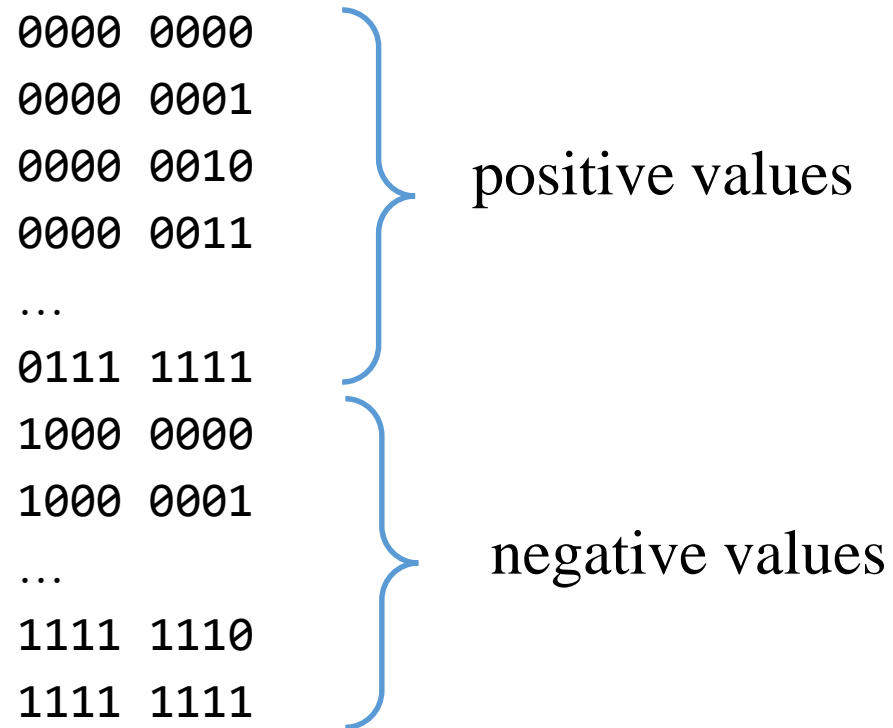
Say 8 bits.

| | | | |
|------|------|--|--------------------|
| 0000 | 0000 |  | $2^8 = 256$ values |
| 0000 | 0001 | | |
| 0000 | 0010 | | |
| 0000 | 0011 | | |
| ... | | | |
| 0111 | 1111 | | |
| 1000 | 0000 | | |
| 1000 | 0001 | | |
| ... | | | |
| 1111 | 1110 | | |
| 1111 | 1111 | | |

Representing signed numbers

We will use a fixed number of bits.

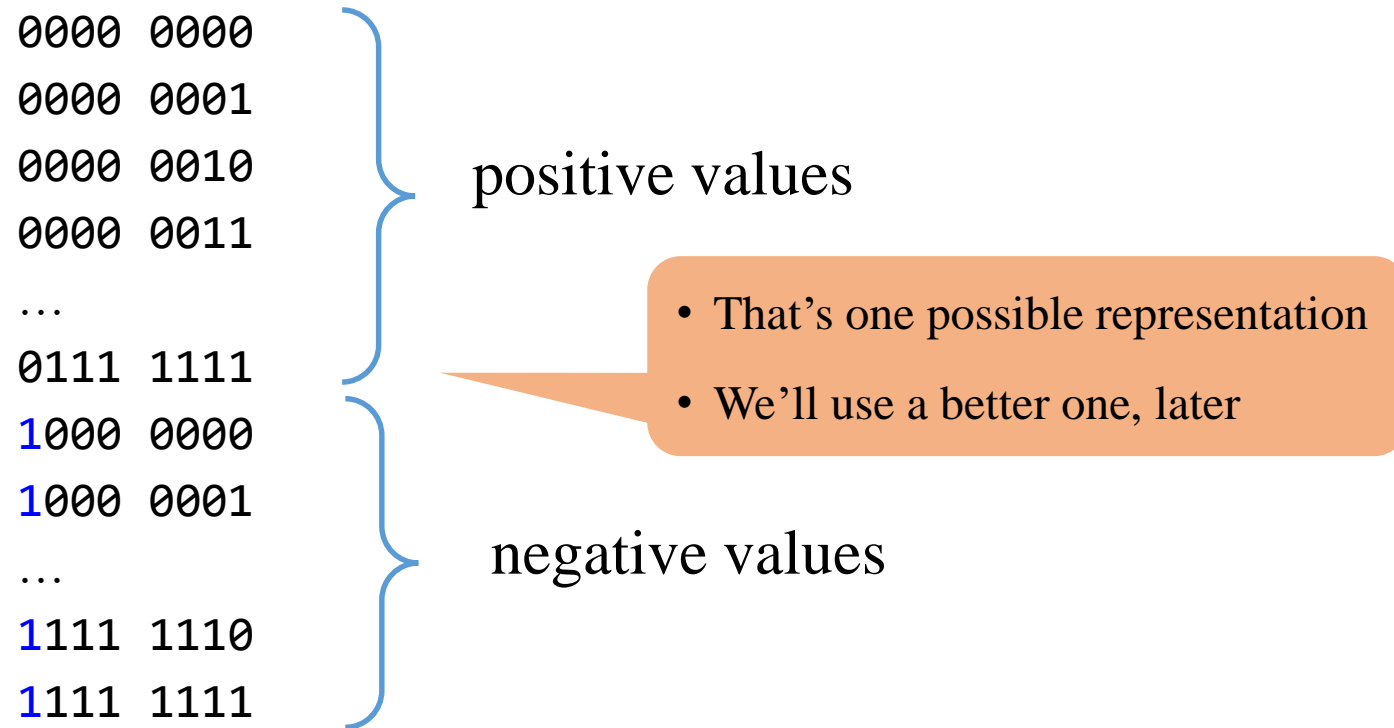
Say 8 bits.



Representing signed numbers

We will use a fixed number of bits.

Say 8 bits.



Decimal \rightarrow Binary

87_{ten}

Decimal \rightarrow Binary

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

Decimal \rightarrow Binary

$$87_{\text{ten}} = 64$$

Decimal \rightarrow Binary

$$87_{\text{ten}} = 64 + 16$$

Decimal \rightarrow Binary

$$87_{\text{ten}} = 64 + 16 + 4$$

Decimal \rightarrow Binary

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

Decimal \rightarrow Binary

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

Decimal \rightarrow Binary

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

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

Decimal \rightarrow Binary

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

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

Decimal \rightarrow Binary

$$\begin{array}{ccccccc} 87_{\text{ten}} & = & 64 & + & 16 & + & 4 & + & 2 & + & 1 \\ & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\ & = & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 &_{\text{two}} \\ & & & \uparrow & & \uparrow & & & & & \\ & & & 32 & & 8 & & & & & \end{array}$$

Chapter 2: Boolean arithmetic



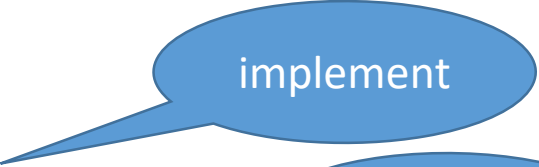



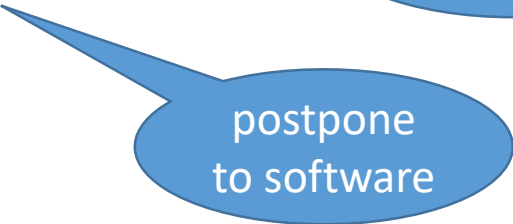
Binary numbers



Binary addition

- Negative numbers
- Arithmetic Logic Unit
- Project 2 overview

Boolean arithmetic

- Addition  implement
- Subtraction  get for free
- Comparison ($<$, $>$, $=$)  get for free
- Multiplication  postpone to software
- Division  postpone to software

Addition

Addition

| | | | | | | | | |
|---|-------|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| + | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| | <hr/> | | | | | | | |
| | ? | ? | ? | ? | ? | ? | ? | ? |

Addition

$$\begin{array}{r} + \quad 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \\ \quad 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \\ \hline \end{array}$$

$$\begin{array}{r} + \quad 21 \\ \quad 92 \end{array}$$

Addition

$$\begin{array}{r} + \quad 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \\ \quad 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \\ \hline \end{array}$$

$$\begin{array}{r} + \quad 21 \\ \quad 92 \\ \hline 113 \end{array}$$

Addition

$$\begin{array}{r} + \quad 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \\ \quad 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \\ \hline \quad 0 \ 1 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \end{array}$$

$$\begin{array}{r} + \quad 21 \\ \quad 92 \\ \hline \quad 113 \end{array}$$

Addition

Addition

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

Addition

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

Addition

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

Addition

$$\begin{array}{r} 1 \\ + 5783 \\ 2456 \\ \hline 39 \end{array}$$

Addition

$$\begin{array}{r} 11 \\ + 5783 \\ 2456 \\ \hline 239 \end{array}$$

Addition

$$\begin{array}{r} 11 \\ + 5783 \\ 2456 \\ \hline 8239 \end{array}$$

Addition

Addition

$$\begin{array}{r} + \quad 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \\ \quad 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \\ \hline \quad ? \ ? \ ? \ ? \ ? \ ? \ ? \ ? \end{array}$$

Addition

$$\begin{array}{r} + \quad 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \\ \quad 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \\ \hline \end{array}$$

Addition

$$\begin{array}{r} + \quad 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \\ \quad 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \\ \hline \qquad \qquad \qquad 1 \end{array}$$

Addition

$$\begin{array}{r} + \quad 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \\ \quad 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \\ \hline \qquad \qquad \qquad 0 \ 1 \end{array}$$

Addition

[illegible]

Addition

$$\begin{array}{r} 11 \\ + 00010101 \\ 01011100 \\ \hline 0001 \end{array}$$

Addition

$$\begin{array}{rcccccccc}
 & & & 1 & 1 & 1 & & & \\
 + & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
 \hline
 & & & & 1 & 0 & 0 & 0 & 1
 \end{array}$$

Addition

$$\begin{array}{rcccccccc}
 & & & 1 & 1 & 1 & & & \\
 + & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
 \hline
 & & & 1 & 1 & 0 & 0 & 0 & 1
 \end{array}$$

Addition

$$\begin{array}{ccccccc}
 & & 1 & 1 & 1 \\
 + & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
 \hline
 & 1 & 1 & 1 & 0 & 0 & 0 & 1
 \end{array}$$

Addition

$$\begin{array}{r}
 \\
 + \quad \begin{array}{ccccccc} & & 1 & 1 & 1 & & \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ \hline 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}
 \end{array}$$

Overflow

Overflow

$$\begin{array}{r} + \quad 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \\ \quad 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \\ \hline \quad ? \ ? \ ? \ ? \ ? \ ? \ ? \ ? \end{array}$$

Overflow

$$\begin{array}{r} 1\ 0\ 0\ 0\ 1\ 1\ 1\ 0\ 0 \\ +\ 1\ 0\ 0\ 1\ 0\ 1\ 0\ 1 \\ 1\ 1\ 0\ 1\ 1\ 1\ 0\ 0 \\ \hline 1\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 1 \end{array}$$

Building an Adder

Building an Adder

$$\begin{array}{rcccccccc} & 0 & 0 & 1 & 0 & 1 & 1 & 0 & \\ + & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ \hline & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{array}$$

- 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 |

Diagram illustrating the addition of two 8-bit numbers, showing the carry bit and the resulting sum bit.

The numbers being added are:

0 0 1 0 1 1 0 0

+ 1 0 0 1 0 1 1 1

The result of the addition is:

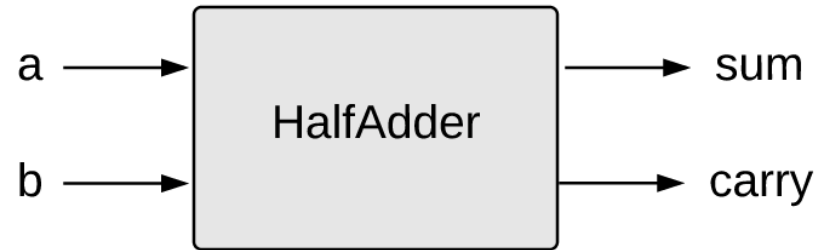
1 1 1 0 1 0 0 1

The carry bit is 0, and the sum bit is 1.

The diagram shows an 8-bit addition. The first number is 00101100 and the second is 10010111. The sum is 11101001. The carry bit is 0, and the sum bit is 1. The carry bit is highlighted with a blue box and a callout. The sum bit is highlighted with a blue box and a callout.

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:  
}
```

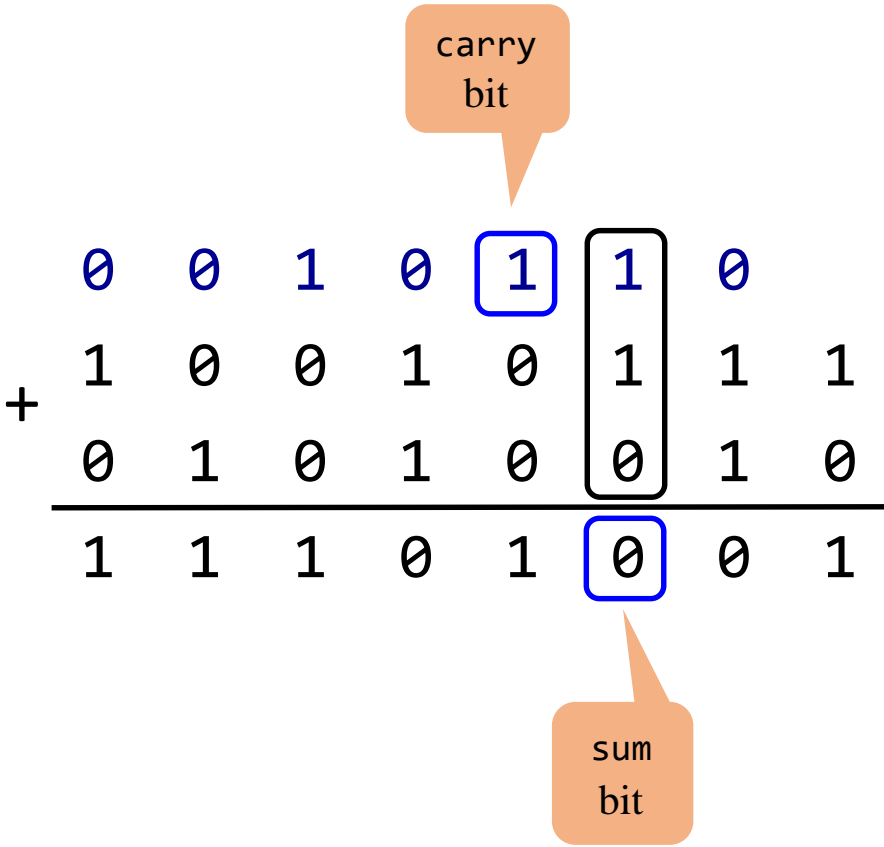
Full adder

Full adder

$$\begin{array}{rcccccccc} & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ + & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ \hline & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{array}$$

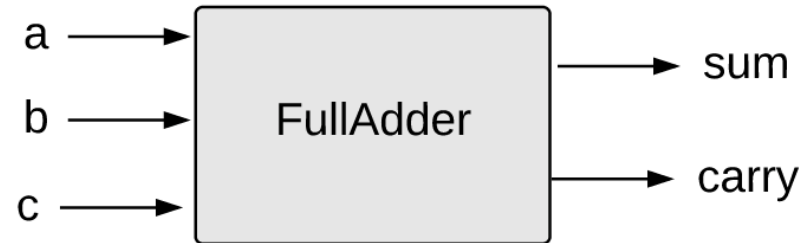
Full adder

| 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 |



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:  
}
```


Multi-bit Adder

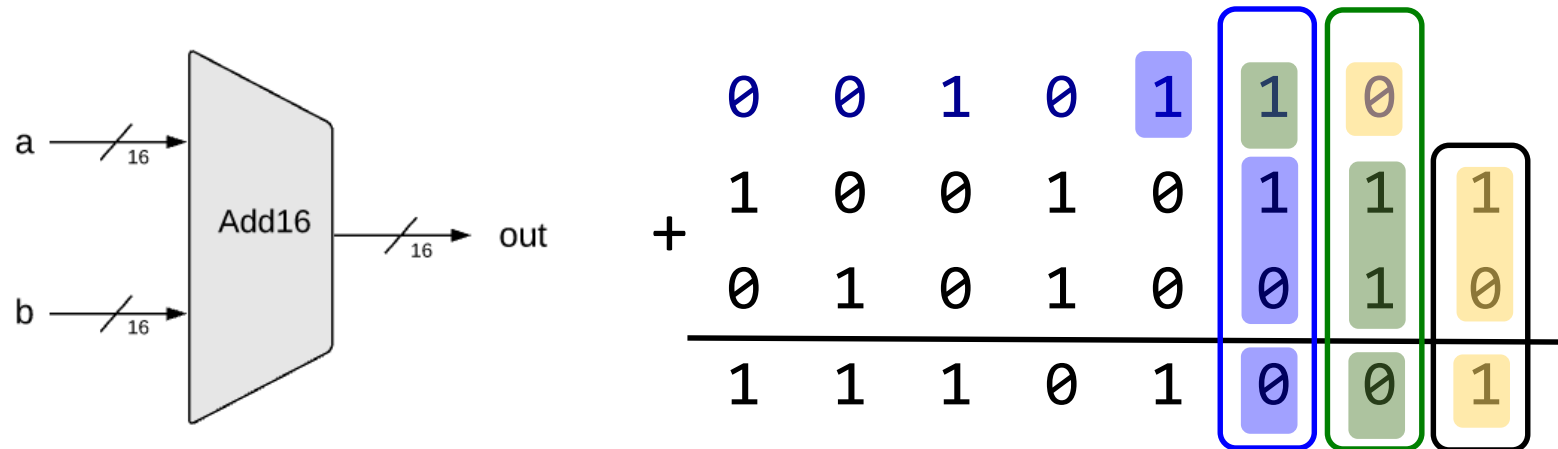
Multi-bit Adder

$$\begin{array}{rcccccccc} & 0 & 0 & 1 & 0 & 1 & 1 & 0 & \\ + & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ \hline & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{array}$$

Multi-bit Adder

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 1 | 0 | 1 | 1 | 0 | |
| | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| + | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |

Multi-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:
}
```

Chapter 2: Boolean arithmetic



Binary numbers



Binary addition



Negative numbers

- Arithmetic Logic Unit
- Project 2 overview

Representing numbers (using 4 bits)

0000

0001

0010

0011

0100

0101

0110

0111

1000

1001

1010

1011

1100

1101

1110

1111

Representing numbers (using 4 bits)

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

Using n bits, we can represent
the positive integers in the range
 $0 \dots 2^n - 1$

Representing negative numbers

0000

0001

0010

0011

0100

0101

0110

0111

1000

1001

1010

1011

1100

1101

1110

1111

Possible solution: use a 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 |

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

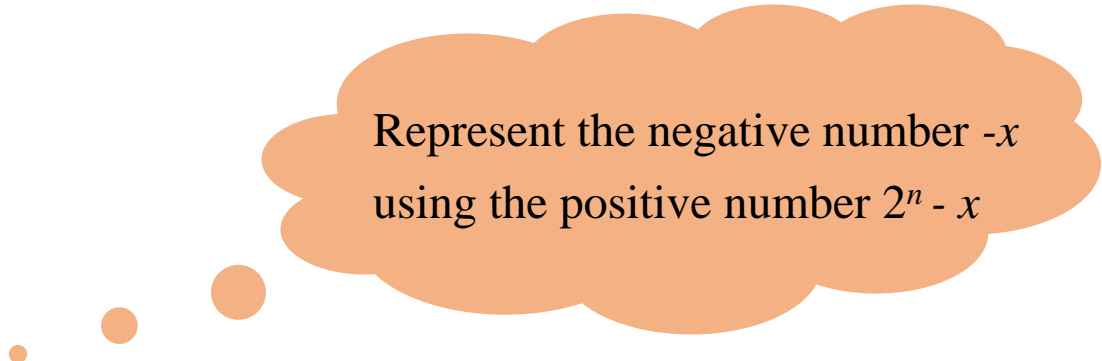
Use the remaining bits to
represent a positive number

Complications:

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

Two's Complement

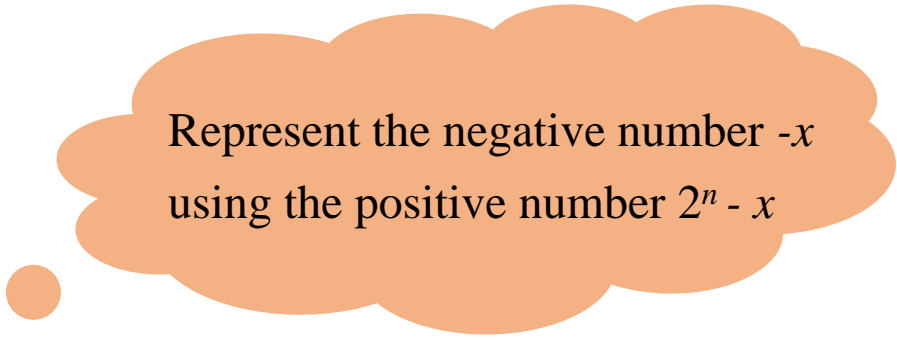
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111



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 $2^n - x$

Two's Complement

| | | | |
|------|----|-----------|--|
| 0000 | 0 | | positive numbers range: $0 \dots 2^{n-1} - 1$ |
| 0001 | 1 | | |
| 0010 | 2 | | |
| 0011 | 3 | | |
| 0100 | 4 | | |
| 0101 | 5 | | |
| 0110 | 6 | | |
| 0111 | 7 | | |
| 1000 | -8 | (16 - 8) | negative numbers range: $-1 \dots -2^n - 1$ |
| 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) | |

Addition using two's complement

$$\begin{array}{r} -2 \\ + \\ -3 \\ \hline \end{array}$$

Addition using two's complement

$$\begin{array}{r} + \quad -2 \\ \quad -3 \\ \hline \end{array}$$

$$\begin{array}{r} + \quad 14 \\ \quad 13 \\ \hline \end{array}$$

Addition using two's complement

$$\begin{array}{r} + \quad -2 \\ + \quad -3 \\ \hline \end{array}$$

$$\begin{array}{r} + \quad 14 \\ + \quad 13 \\ \hline \end{array}$$

$$\begin{array}{r} + \quad 1110 \\ + \quad 1101 \\ \hline \end{array}$$

Addition using two's complement

$$\begin{array}{r} + -2 \\ + -3 \\ \hline \end{array}$$

$$\begin{array}{r} + 14 \\ + 13 \\ \hline \end{array}$$

$$\begin{array}{r} + 1110 \\ + 1101 \\ \hline 11011 \end{array}$$

$11011 = 27_{\text{ten}}$

$1011 = 11_{\text{ten}}$

Addition using two's complement

$$\begin{array}{r} + \quad -2 \\ + \quad -3 \\ \hline \end{array}$$

$$\begin{array}{r} + \quad 14 \\ + \quad 13 \\ \hline 11 \end{array}$$

$$\begin{array}{r} + \quad 1110 \\ + \quad 1101 \\ \hline 11011 \end{array}$$

$11011 = 27_{\text{ten}}$

$1011 = 27_{\text{ten}}$

Addition using two's complement

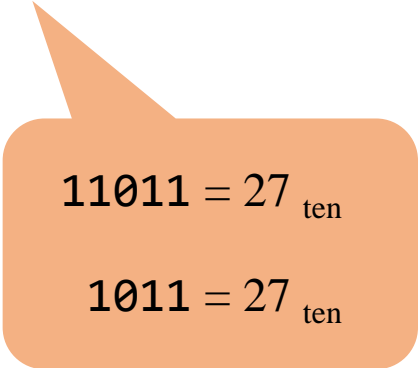
$$\begin{array}{r} + \quad -2 \\ + \quad -3 \\ \hline -5 \end{array}$$

$$\begin{array}{r} + \quad 14 \\ + \quad 13 \\ \hline 11 \end{array}$$

$$\begin{array}{r} + \quad 1110 \\ + \quad 1101 \\ \hline 11011 \end{array}$$

Two's complement rationale:

- representation is modulu 2^n
- addition is modulu 2^n


$$11011 = 27_{\text{ten}}$$

$$1011 = 27_{\text{ten}}$$

Computing $-x$

Input: x

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

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

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

Computing $-x$

Input: x

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

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

11111111_{two}

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

Computing $-x$

Input: x

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

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

11111111_{two}

$$\begin{array}{r} 11111111 \\ - 10101100 \text{ (some } x \text{ example)} \\ \hline 01010011 \end{array}$$

Computing $-x$

Input: x

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

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

11111111_{two}

$$\begin{array}{r} 11111111 \\ - 10101100 \text{ (some x example)} \\ \hline 01010011 \text{ (flip all the bits)} \end{array}$$

Now add 1 to the result

Computing $-x$ (example)

Input: 4

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

Input: 0100

Flip the bits: 1011

Computing $-x$ (example)

Input: 4

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

Input: 0100

Flip the bits: 1011

Add one:
$$\begin{array}{r} + \\ 1011 \\ \hline 1000 \end{array}$$

Computing $-x$ (example)

Input: 4

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

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 complement)

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

Chapter 2: Boolean arithmetic

✓ Binary numbers

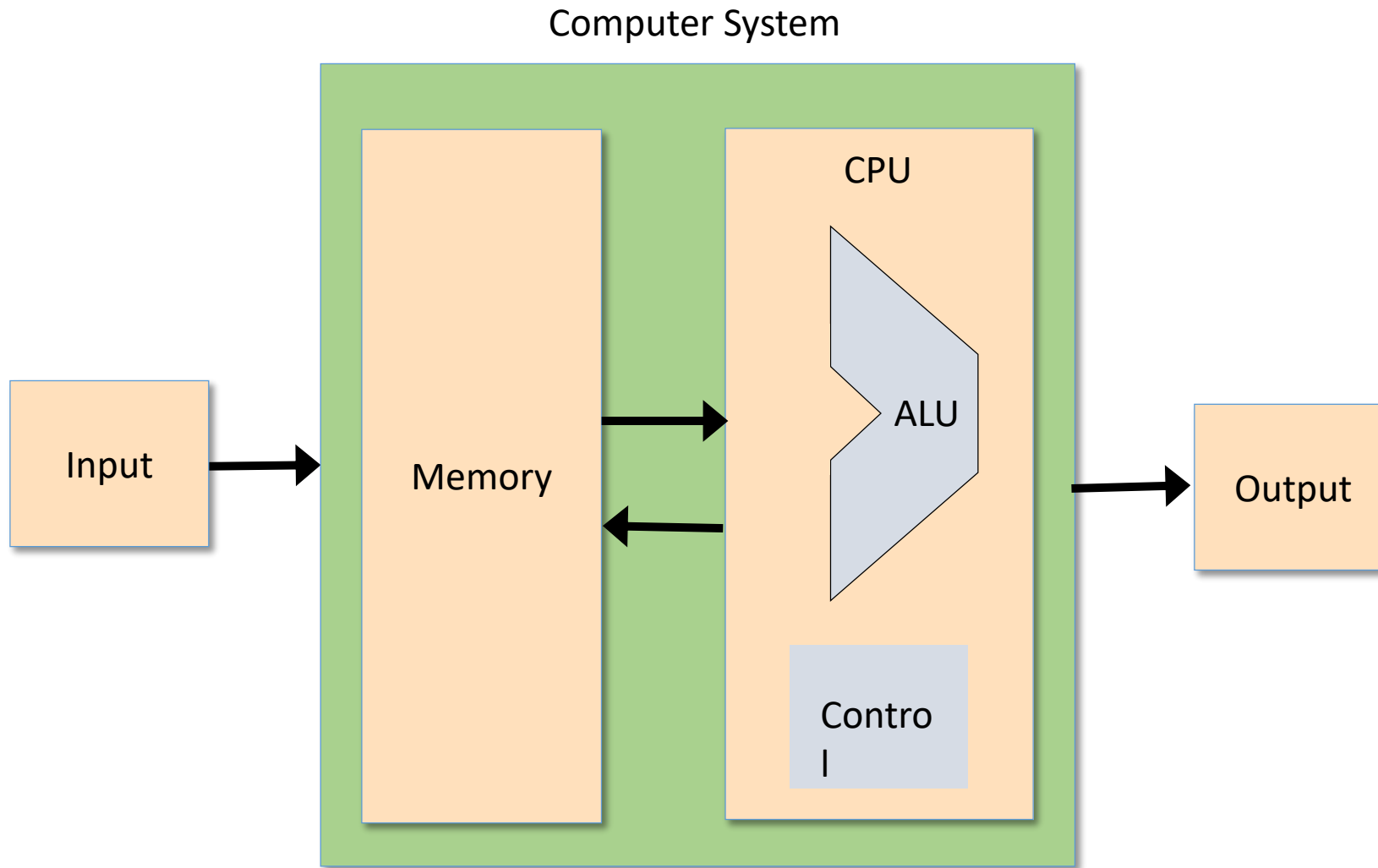
✓ Binary addition

✓ Negative numbers

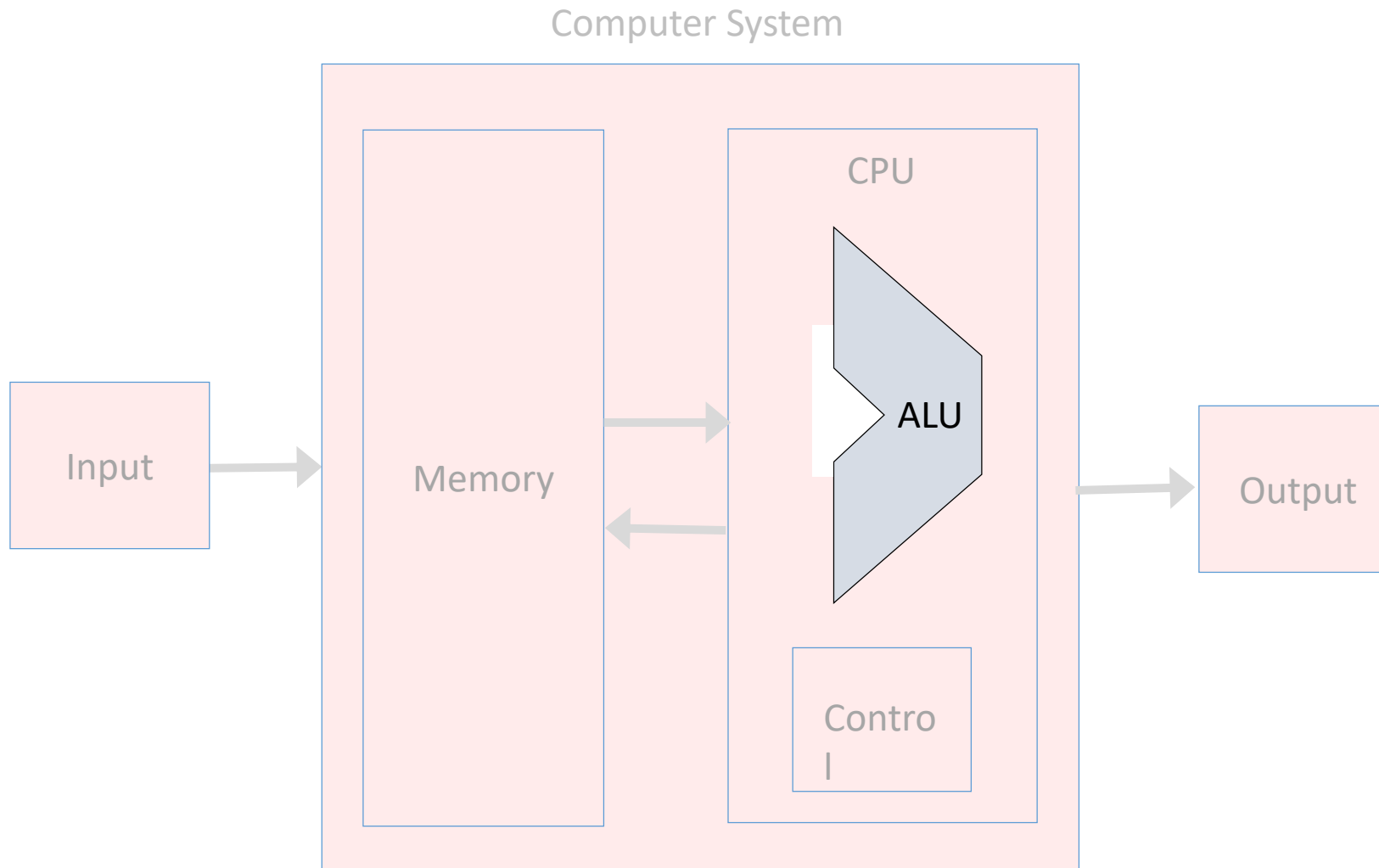
➡ Arithmetic Logic Unit

- Project 2 overview

Von Neumann Architecture



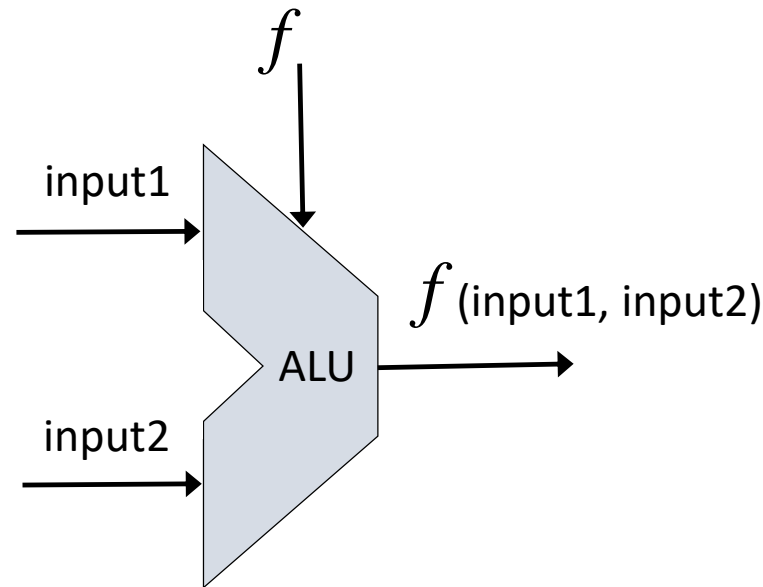
The Arithmetic Logical Unit



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

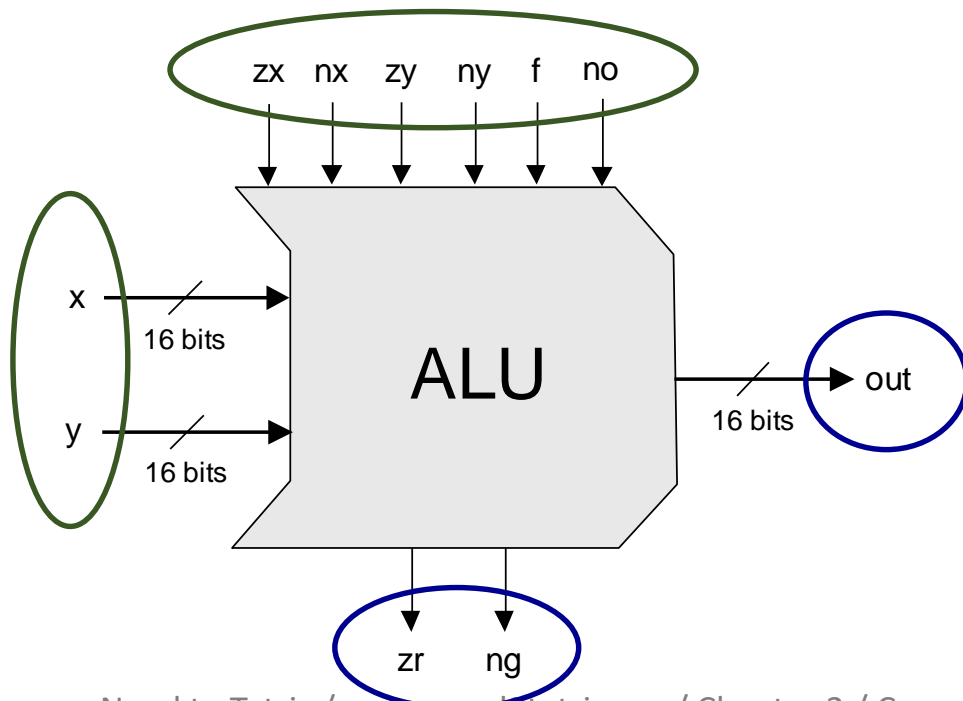


- Arithmetic functions: integer addition, multiplication, division, ...
- logical functions: And, Or, Xor, ...

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

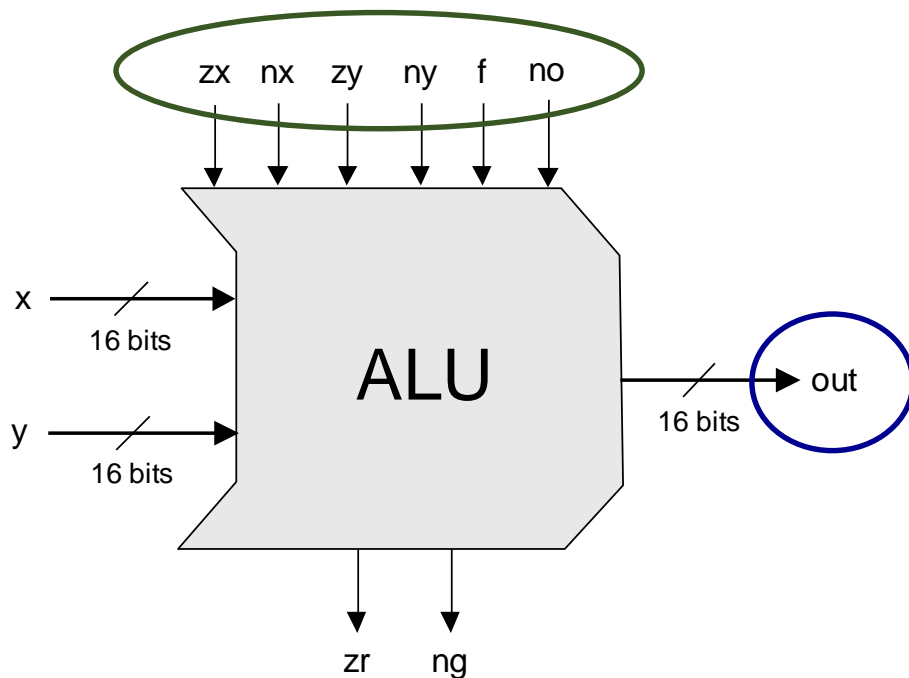
The Hack ALU

- 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 |

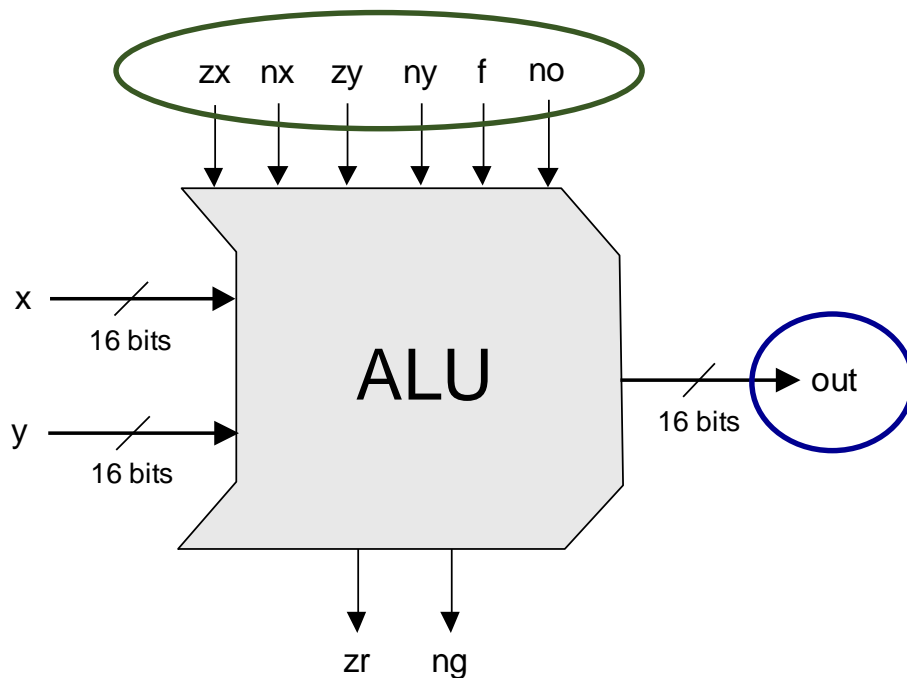
The Hack ALU



| 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 |

The Hack ALU in action: compute $y-x$

The screenshot shows the Nand2Tetris IDE interface. The top toolbar contains icons for loading, running, and inspecting. The main window is divided into several sections:

- Input/Output Table:** A table with columns for Name, Value, and Name. It shows the ALU's inputs and outputs.
- HDL Code Editor:** Displays the HDL code for the ALU implementation.
- Logic Diagram:** A diagram of the ALU implementation, showing the D Input, M/A Input, and the ALU output.

Annotations (orange callouts) describe the steps to compute $y-x$:

1. Set the ALU's inputs and control bits to some test values (000111 codes "output $y-x$ ")
2. Evaluate the chip logic
3. Inspect the ALU outputs

Other annotations include:

- Load tools/builtInChips/ALU.hdl
- Built-in ALU implementation
- The built-in ALU implementation has some GUI side-effects

The HDL code in the editor is as follows:

```
// This file is part of the material for "The Elements of Computing Systems"
// by Noam Nisan and Shimon Schocken
// MIT Press. Book site: www.nand2tetris.org
// File name: tools/builtIn/ALU.hdl

/**
 * The ALU. Computes a pre-defined operation on two 16-bit
 * integers x and y, where x and y are two 16-bit integers
 * by a set of 6 control bits defined by the ALU's inputs.
 * The ALU operation can be described as follows:
 * if zx=1 set x = 0
 * if nx=1 set x = !x
 * if zy=1 set y = 0
 * if ny=1 set y = !y
 */
```

The logic diagram shows the ALU implementation with the following inputs and outputs:

- D Input: 30
- M/A Input: 20
- ALU output: -10

The Hack ALU in action: compute $x \& y$

The screenshot shows the Nand2Tetris ALU simulator interface. The top menu bar includes File, View, Run, and Help. Below the menu is a toolbar with icons for running, stepping, and other functions. The main window is divided into several sections:

- Chip Name:** ALU, Time: 0
- Input pins:** A table with columns Name and Value. The values for x[16] and y[16] are circled in green.
- Output pins:** A table with columns Name and Value. The values for out[16], zr, and ng are circled in blue.
- HDL:** A text area showing the Verilog code for the ALU.
- ALU Diagram:** A diagram showing the ALU operation with D Input, M/A Input, and ALU output.

Annotations (orange callouts) provide additional information:

- Set to binary I/O format:** Points to the Format dropdown menu.
- Inspect the ALU outputs:** Points to the Output pins table.
- Set the ALU's inputs and control bits to some test values (000000 codes "compute x&y"):** Points to the Input pins table.

Input pins table:

| Name | Value |
|-------|------------------|
| x[16] | 1110101110000110 |
| y[16] | 0001100001101101 |
| zx | 0 |
| nx | 0 |
| zy | 0 |
| ny | 0 |
| f | 0 |
| no | 0 |

Output pins table:

| Name | Value |
|---------|------------------|
| out[16] | 0000100000000100 |
| zr | 0 |
| ng | 0 |

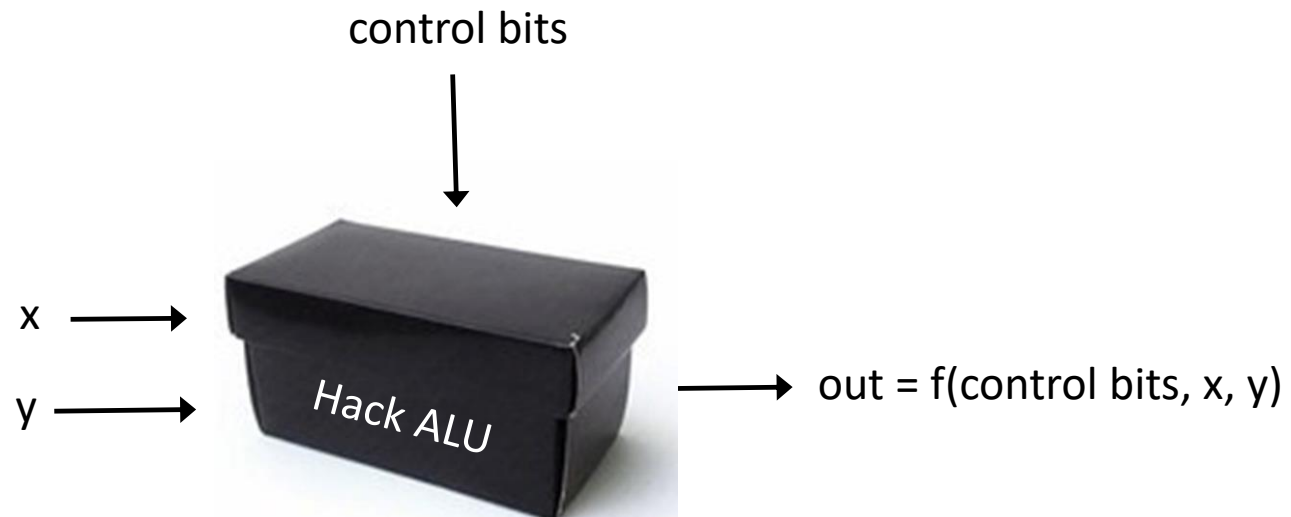
HDL code:

```
// This file is part of the material for the book "The Elements of Computing Systems" by Noam Nisan and Shimon Schocken. MIT Press. Book site: www.nand2tetris.org. File name: tools/builtIn/ALU.  
  
/**  
 * The ALU. Computes a pre-defined operation on two 16-bit integers x and y.  
 * where x and y are two 16-bit integers.  
 * by a set of 6 control bits described in the book.  
 * The ALU operation can be described as follows:  
 *   if zx=1 set x = 0  
 *   if nx=1 set x = !x  
 *   if zy=1 set y = 0  
 *   if ny=1 set y = !y  
 */
```

ALU Diagram:

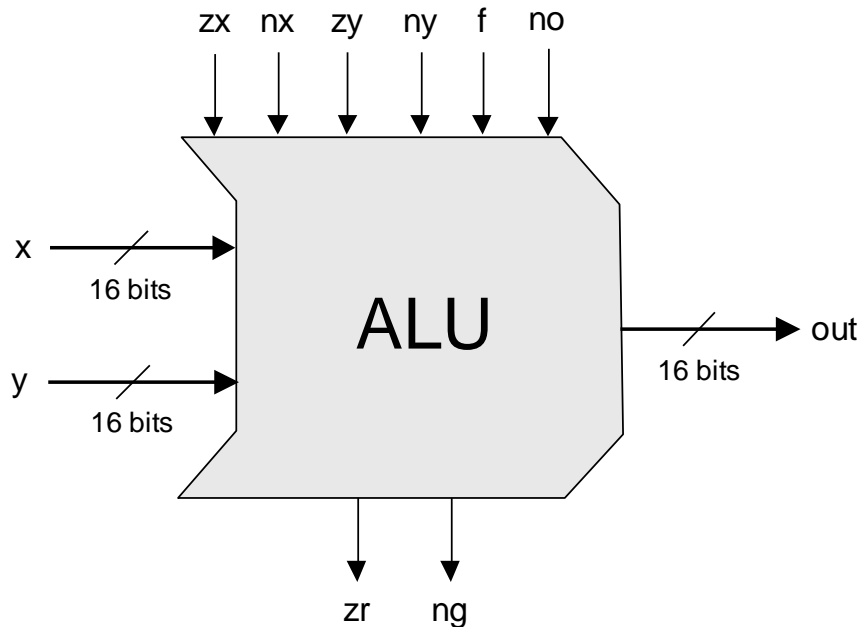
ALU
D Input : -5242
M/A Input : 6253
ALU output : 2052

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 | 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)= |



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

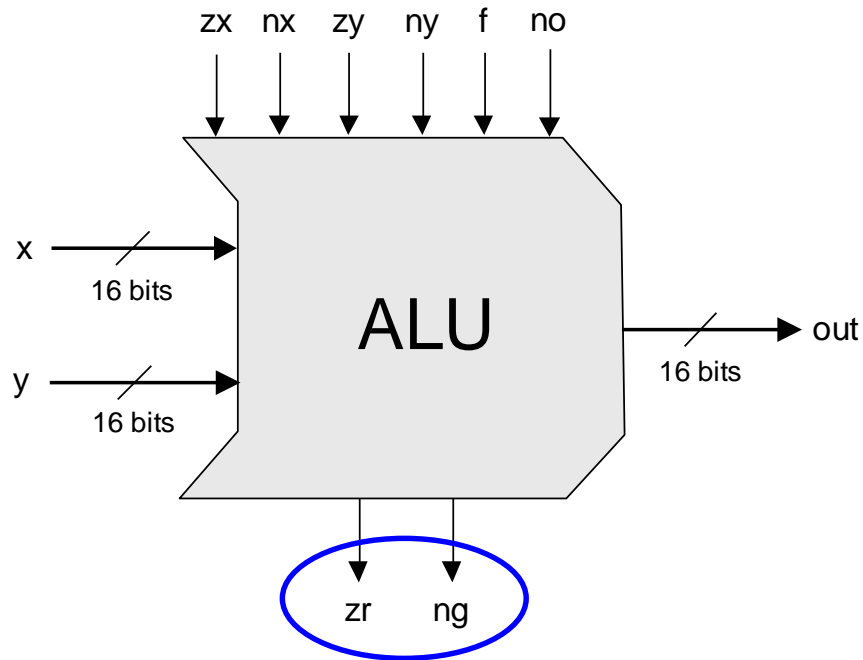
y: 1 0 0 0

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`

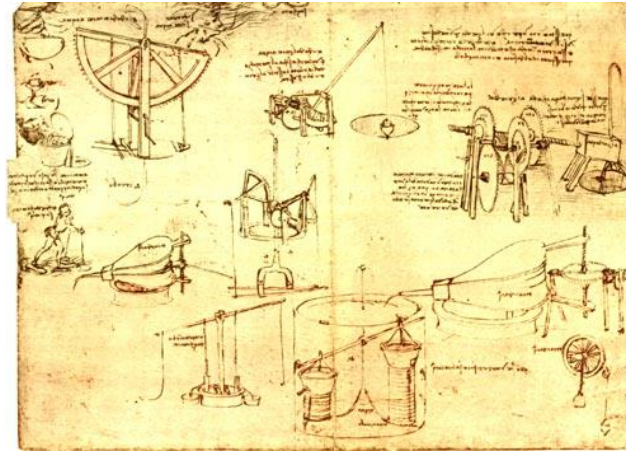
`if (out < 0) then ng = 1, else ng = 0`

These two control bits will come into play when we build the complete computer's architecture.

Perspective

The Hack ALU is:

- Simple
 - Elegant
 - To implement it this ALU, you only need to know how to:
 - Set a 16-bit value to 0000000000000000
 - Set a 16-bit value to 1111111111111111
 - Negate a 16-bit value (bit-wise)
 - Compute plus or And on two 16-bit values
- That's it!



“Simplicity is the ultimate sophistication.”
— Leonardo da Vinci

Chapter 2: Boolean arithmetic



Binary numbers



Binary addition



Negative numbers



Arithmetic Logic Unit



Project 2 overview

Project 2

Given: All the chips built in Project 1

Goal: Build the following chips:

- HalfAdder
- FullAdder
- Add16
- Inc16
- ALU

A family of *combinational* chips,
from simple adders to an
Arithmetic Logic Unit.

Half Adder



| a | b | sum | carry |
|---|---|-----|-------|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

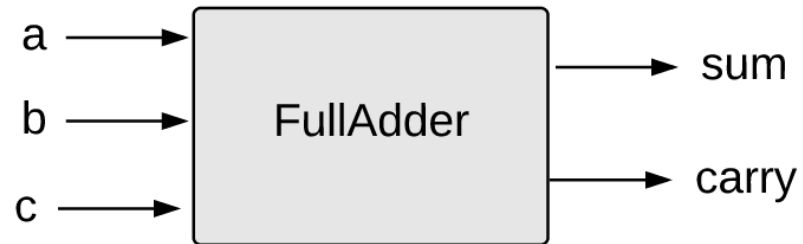
HalfAdder.hdl

```
/** 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.

Full Adder



FullAdder.hdl

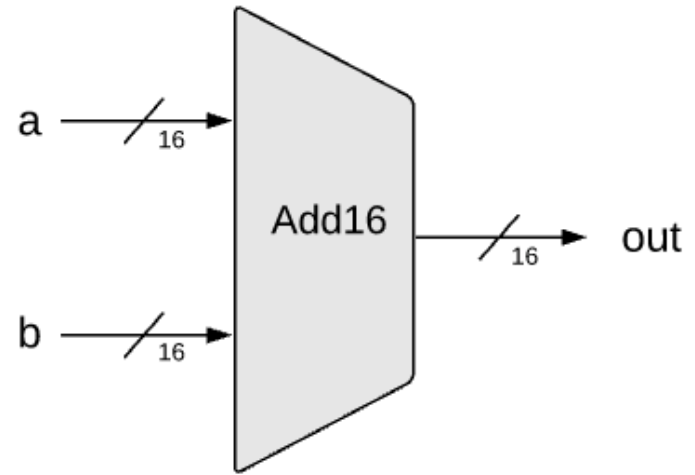
```
/** Computes the sum of three bits. */  
  
CHIP HalfAdder {  
    IN a, b, c;  
    OUT sum, carry;  
  
    PARTS:  
        // Put your code here:  
}
```

| 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 |

Implementation tips

Can be built using 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.

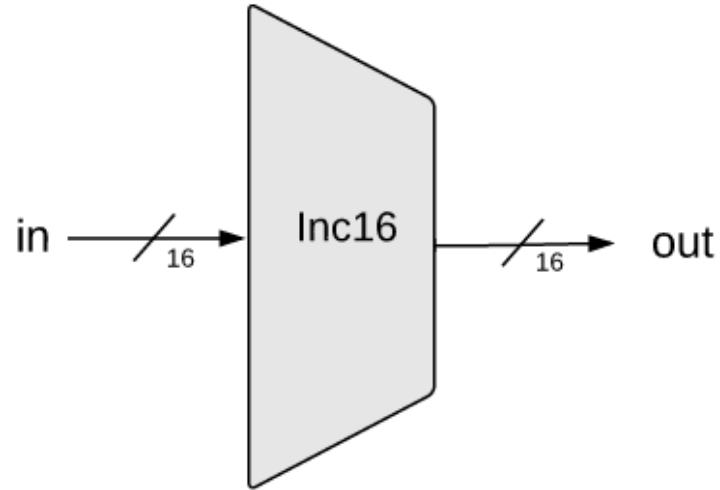
Add16.hdl

```
/*
 * Adds two 16-bit, two's-complement values.
 * The most-significant carry bit is ignored.
 */

CHIP Add16 {
    IN a[16], b[16];
    OUT out[16];

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

16-bit incrementor



Implementation tip

The single-bit 0 and 1 values are represented in HDL as false and true.

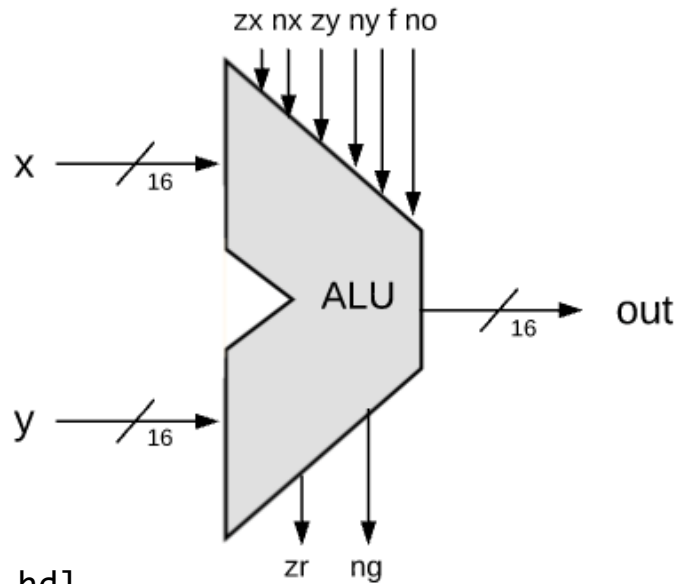
Inc16.hdl

```
/*
 * Outputs in + 1.
 * The most-significant carry bit is ignored.
 */

CHIP Inc16 {
    IN in[16];
    OUT out[16];

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


ALU



Implementation tips

- ❑ Building blocks: Add16, and some gates built in project 1
- ❑ Can be built with ~20 lines of HDL code

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
...
```

Project 2 Resources

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

Home

Prerequisites

Syllabus

Course

Book

Software

Terms

Papers

Talks

Cool Stuff

About

Team

Q&A

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 |

All the necessary project 2 files are available in:
nand2tetris / projects / 02

Slide 114

More resources

- HDL Survival Guide
- Hardware Simulator Tutorial
- nand2tetris Q&A forum



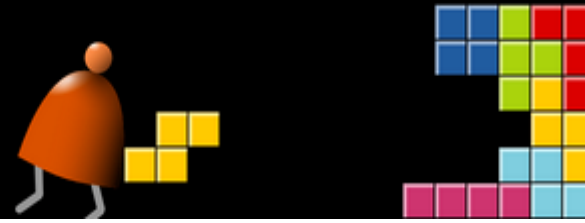
All available in: www.nand2tetris.org

Best practice advice

- Try to implement the chips in the given order
- If you don't implement some of the chips required in project 2, you can still use them as chip-parts in other chips. Just rename the given stub-files; this will cause the simulator to use the built-in versions of these chips
- You can invent new, “helper chips”; however, this is not required: you can build any chip using previously-built chips only
- Strive to use as few chip-parts as possible
- You will have to use chips implemented in Project 1
- For efficiency and consistency's sake, use their built-in versions rather than your own implementation.

Chapter 2: Boolean arithmetic

- ✓ Binary numbers
- ✓ Binary addition
- ✓ Negative numbers
- ✓ Arithmetic Logic Unit
- ✓ Project 2 overview



Chapter 2

Boolean Arithmetic

These slides support chapter 2 of the book

The Elements of Computing Systems

By Noam Nisan and Shimon Schocken

MIT Press