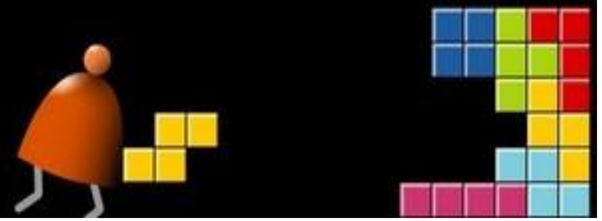


Chapter 2

Boolean Arithmetic

These slides support chapter 2 of the book

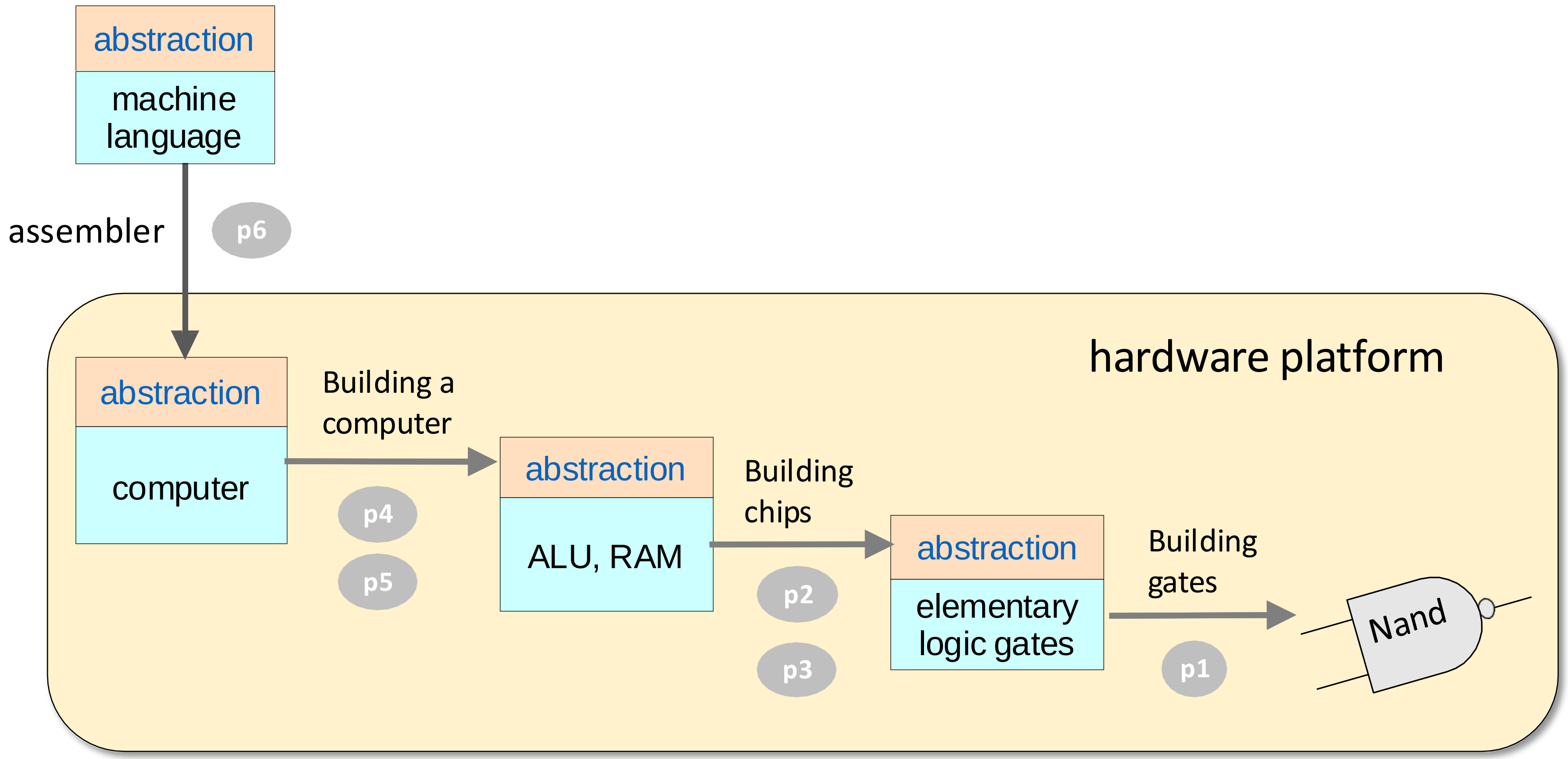
The Elements of Computing Systems

(1st and 2nd editions)

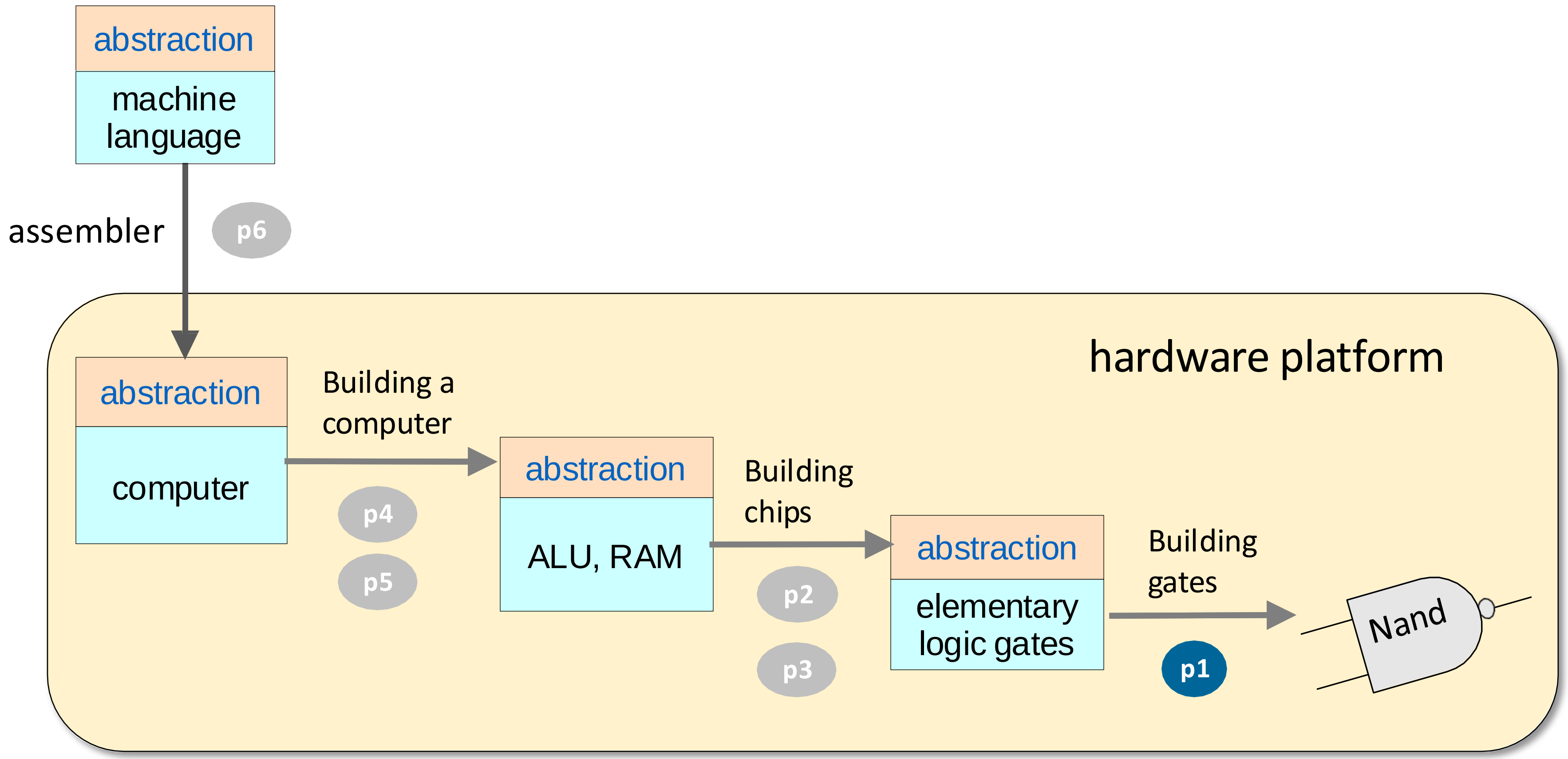
By Noam Nisan and Shimon Schocken

MIT Press

Nand to Tetris Roadmap: Hardware



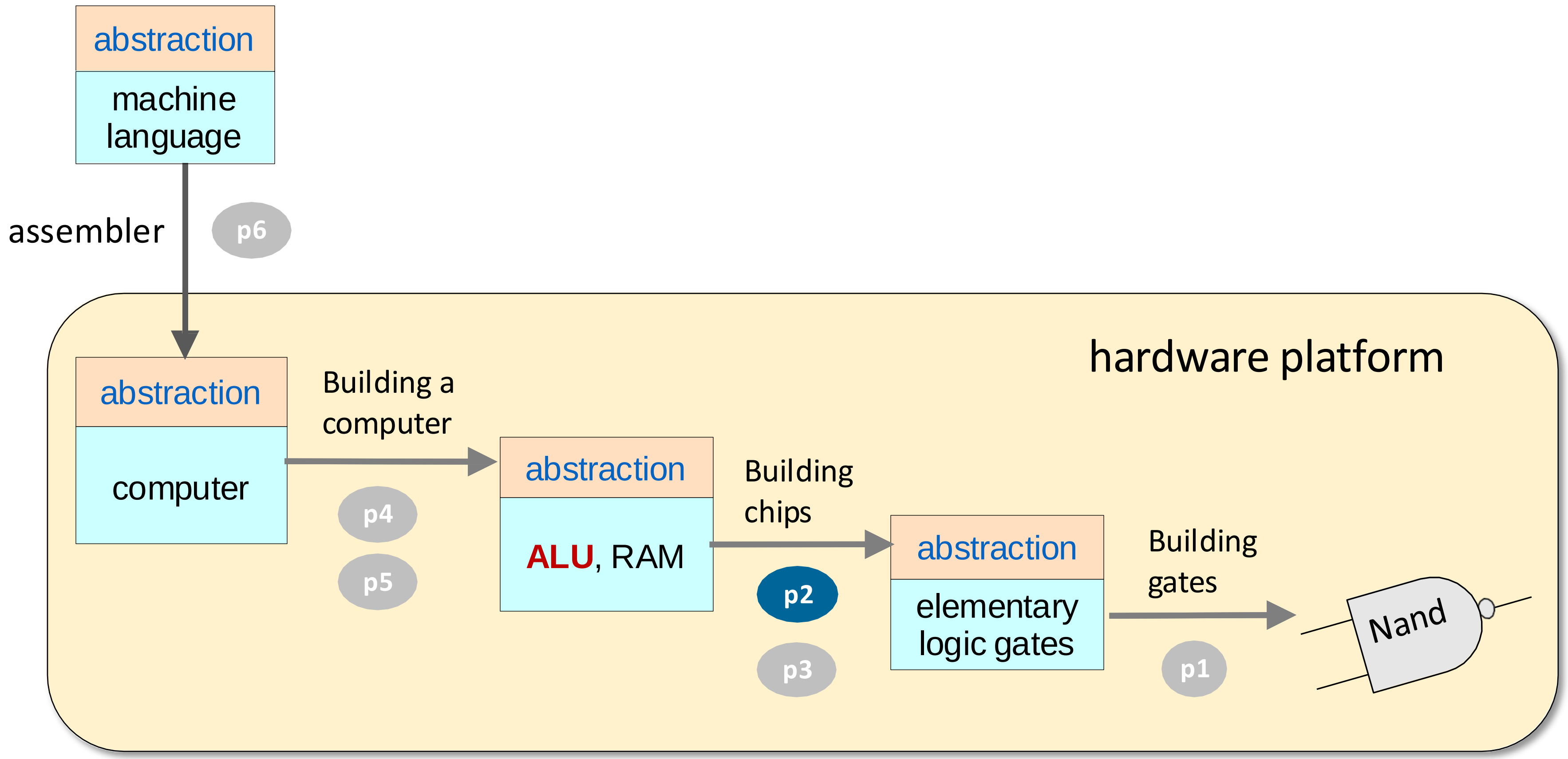
Nand to Tetris Roadmap: Hardware



<u>Project 1</u>

Build 15 elementary logic gates

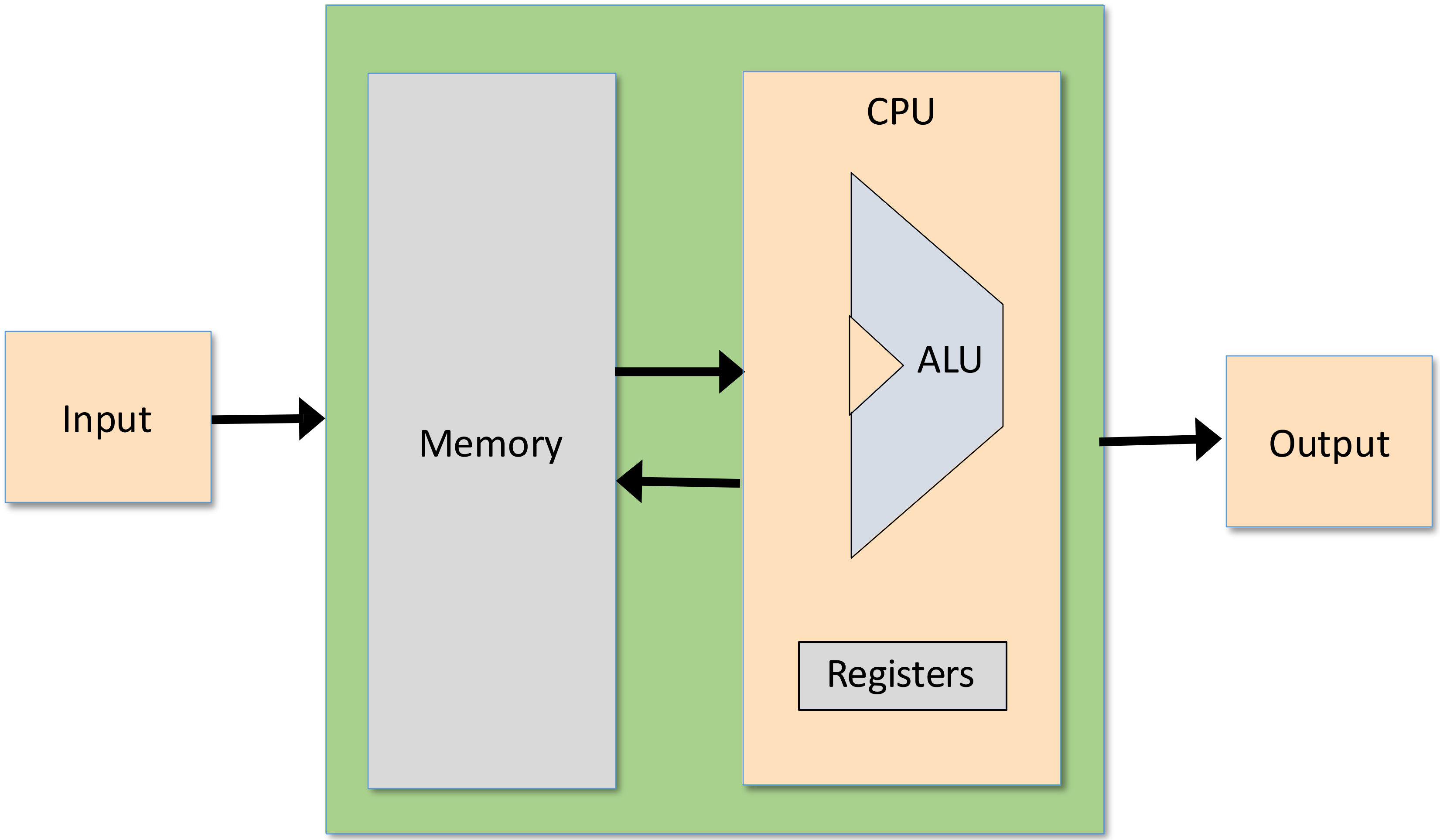
Nand to Tetris Roadmap: Hardware



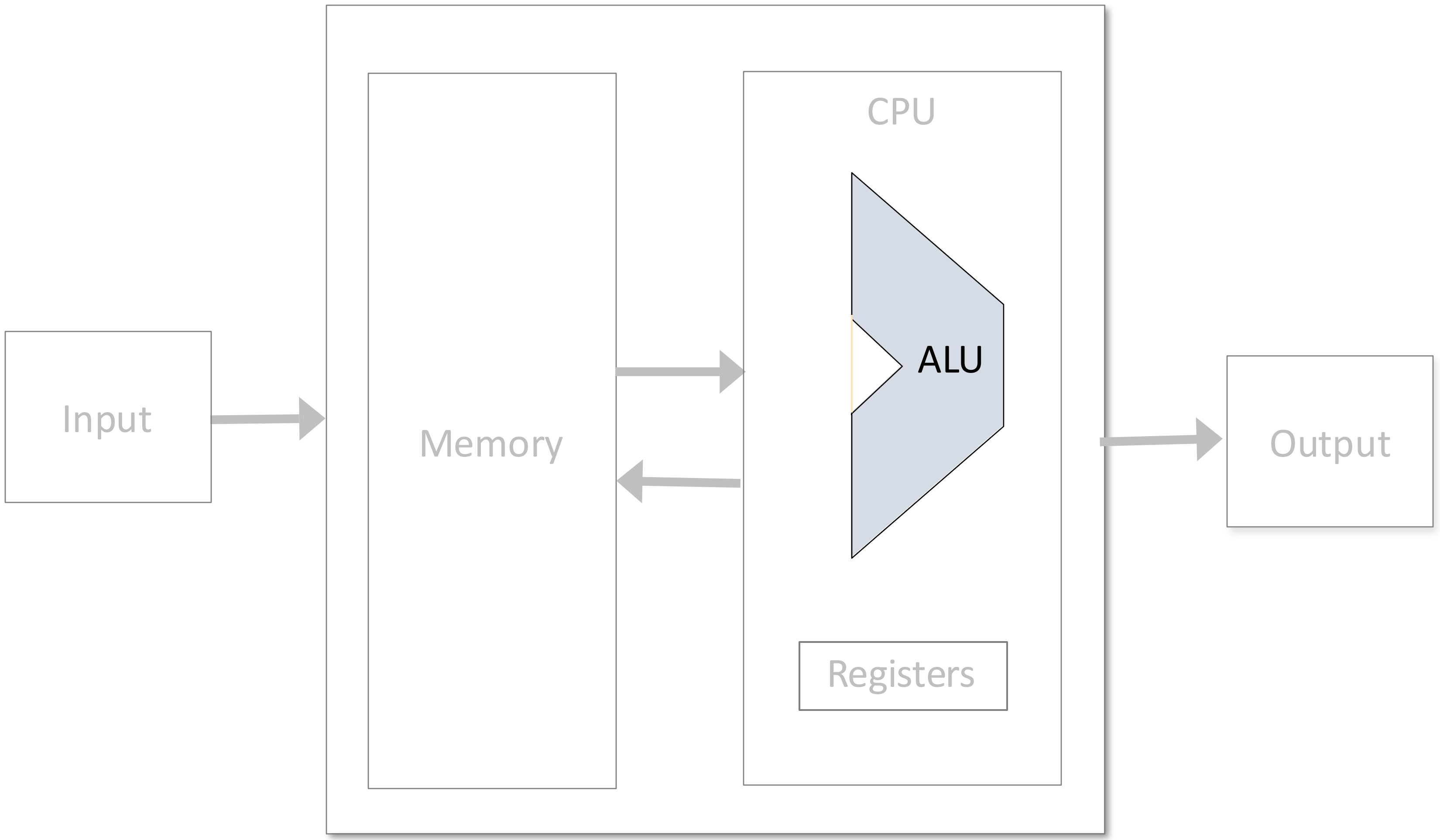
<u>Project 2</u>

Building chips that do arithmetic,
ending up with an ALU

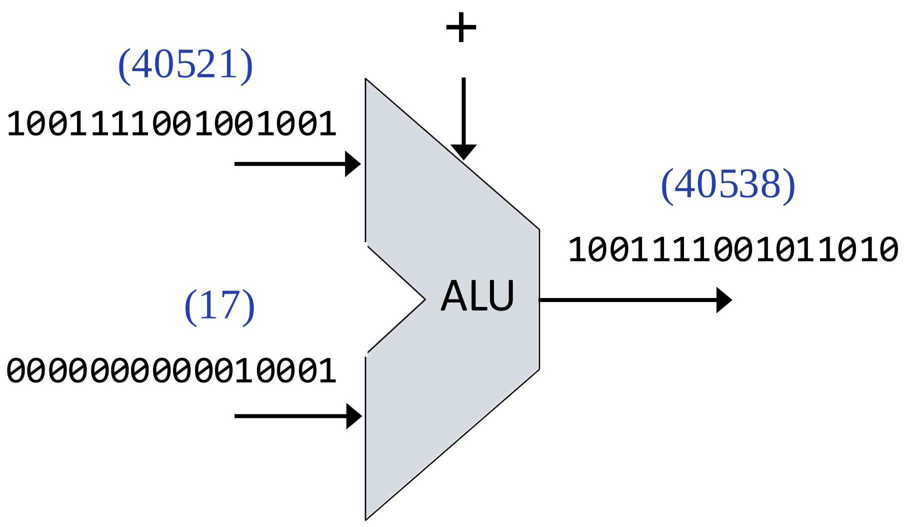
Computer system



Computer system



Arithmetic Logical Unit



The ALU computes a given function on two given *n*-bit values, and outputs an *n*-bit value

ALU functions (*f*)

- Arithmetic: x + y, x – y, x + 1, x – 1, . . .
- Logical: x & y, x | y, !x , . . .

Challenges

- Use `0`’s and `1`’s for representing numbers
- Use logic gates for realizing arithmetic functions.

Chapter 2: Boolean Arithmetic

<u>Theory</u>

- Representing numbers
- Binary numbers
- Boolean arithmetic
- Signed numbers

<u>Practice</u>

- Arithmetic Logic Unit (ALU)
- Project 2: Chips
- Project 2: Guidelines

Chapter 2: Boolean Arithmetic

Theory

- Representing numbers
- Binary numbers
- Boolean arithmetic
- Signed numbers

Practice

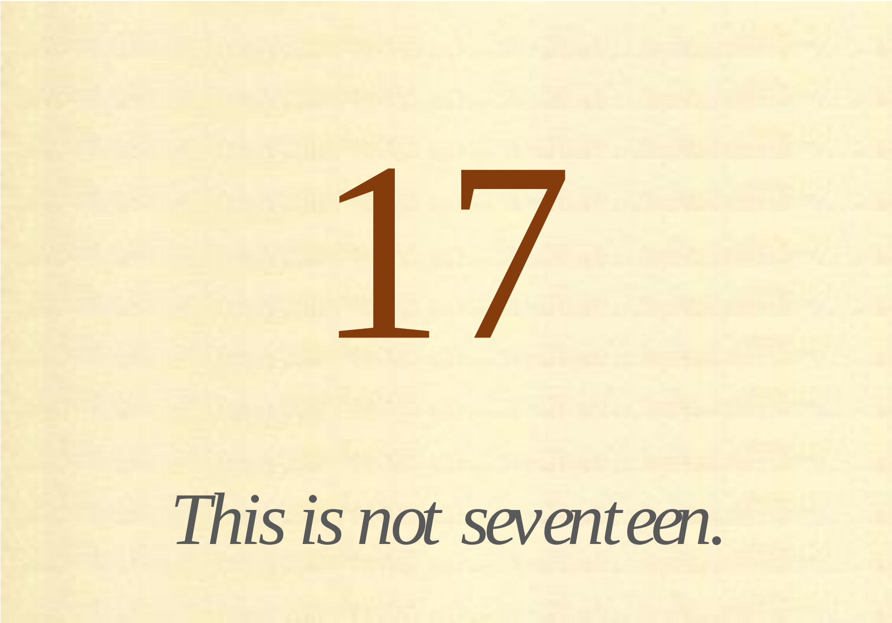
- Arithmetic Logic Unit (ALU)
- Project 2: Chips
- Project 2: Guidelines

Representation

This is not a pipe

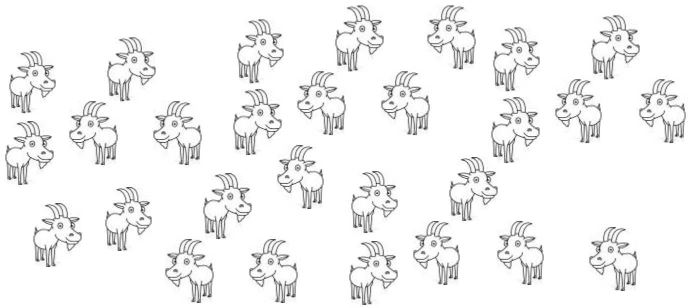
(by René Magritte)

Representation



Rather, it's an agreed-upon code (*numeral*)
that represents the number seventeen.

A brief history of numeral systems



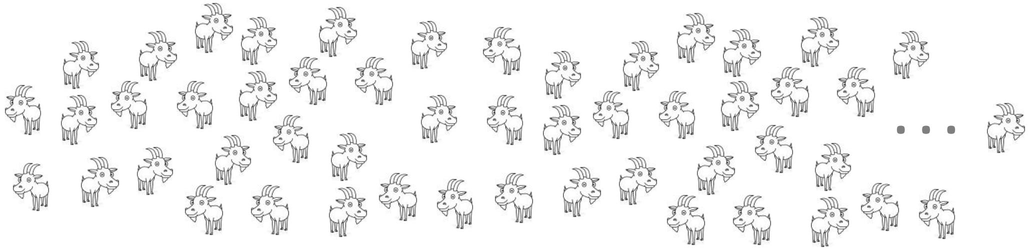
Twenty seven goats

Unary: 

Egyptian: 

Roman: XXVII

A brief history of numeral systems



Six thousands,
five hundreds,
and seven goats

Unary: 

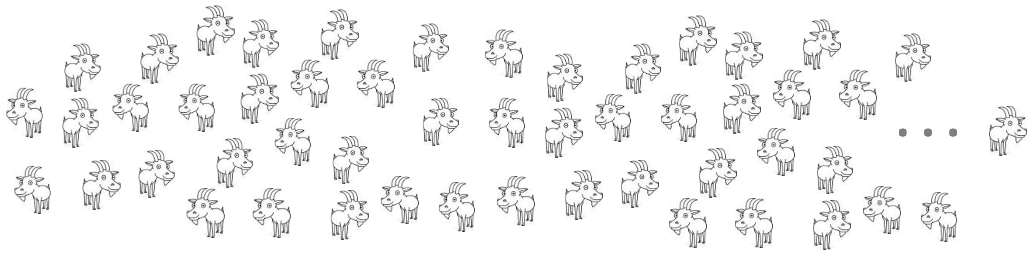
Egyptian: 

Roman: MMMMMMDVII

Old numeral systems:

- Don’t scale
- Cumbersome arithmetic
- Used until about 1000 years ago
- Blocked the progress of Algebra
  (and commerce, science, technology)

Positional numeral system



Six thousands,
five hundreds,
and seven goats

3 2 1 0
6 5 0 7

n#$

! d_i # = 6 #10& + 5 #10² + 0 #10$ + 7 #10⁰ = 6507

10^i

0

Where n is the number of digits in the numeral, and d_i is the digit in position i

A most important innovation, brought to the West from the East around 1200

<u>Positional representation</u>

- *Digits*: A fixed set of symbols, including 0
- *Base*: The number of symbols
- *Numeral*: An ordered sequence of digits
- *Value*: The digit in position i (counting from right to left, and starting at 0) encodes how many copies of *base*^{i} are added to the value.

The method mentions no specific base.

Chapter 2: Boolean Arithmetic

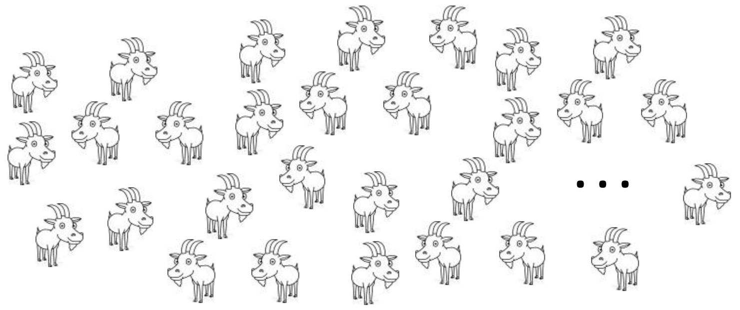
Theory

- ✓ Representing numbers
- → Binary numbers
- Boolean arithmetic
- Representing signed numbers

Practice

- Arithmetic Logic Unit (ALU)
- Project 2: Chips
- Project 2: Guidelines

Positional number system



Seven thousands
and fifty three
goats

3 2 1 0

$7\,0\,5\,3_{10}$

$n \# \$$

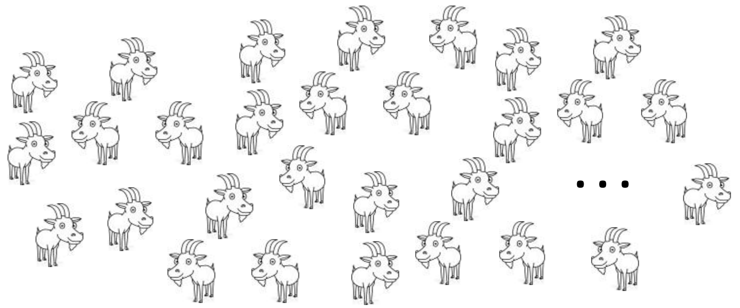
!

10^{i}

0

$d_i \; \# \; = \; 7 \; \# 10^{\&} + 0 \; \# 10^{2} + 5 \; \# 10^{\$} + 3 \; \# 10^{0} \; = \; 7053$

Positional number system



Seven thousands
and fifty three
goats

Decimal (base 10) system:
Human friendly

3 2 1 0

7053_{10}

n#$

!

10^i

0

d_i # = 7 #10^& + 0 #10^2 + 5 #10^$ + 3 #10^0 = 7053

Binary (base 2) system:
Computer friendly

12 11 10 . . . 3 2 1 0

1101110001101_2

. . .

n#$

!

2^i

0

d_i # = 1 #2^$2 + 1 #2^$$ + 0 #2^$0 + ⋯ + 1 #2^0 = 7053

Binary and decimal systems

Binary	Decimal
0	0
1	1
1 0	2
1 1	3
1 0 0	4
1 0 1	5
1 1 0	6
1 1 1	7
1 0 0 0	8
1 0 0 1	9
1 0 1 0	10
1 0 1 1	11
1 1 0 0	12
1 1 0 1	13
...	...

Humans are used to enter and view numbers in base 10;

Computers represent and process numbers in base 2;

Therefore, we need efficient algorithms for converting from one base to the other.

Decimal ⟷ binary conversions

Powers of 2: (aids in calculations)

$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$
$2^8 = 256$
$2^9 = 512$
$2^{10} = 1024$
. . .

Binary to decimal:

$$decimal\,(\overset{5}{1}\overset{4}{1}\overset{3}{0}\overset{2}{1}\overset{1}{0}\overset{0}{1}{}_2) = 2^5 + 2^4 + 2^2 + 2^{\$} = 53_{10}$$

Decimal to binary:

$$binary\,(53_{10}) = 2^5 + 2^4 + 2^2 + 2^{\$} = \overset{5}{1}\overset{4}{1}\overset{3}{0}\overset{2}{1}\overset{1}{0}\overset{0}{1}{}_2$$

Algorithm: What is the largest power of 2 that "fits into" 53? It's $32 = 2^5$. We still have to handle 53 – 32, so, what is the largest power of 2 that fits into 21? It's $16 = 2^4$, and so on.

Practice:

$$decimal\,(1011010_2) = \,?$$

$$binary\,(523_{10}) = \,?$$

Decimal ⟺ binary conversions

Powers of 2: (aids in calculations)

$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

$2^4 = 16$

$2^5 = 32$

$2^6 = 64$

$2^7 = 128$

$2^8 = 256$

$2^9 = 512$

$2^{10} = 1024$

. . .

Binary to decimal:

$$decimal\,(\overset{5}{1}\overset{4}{1}\overset{3}{0}\overset{2}{1}\overset{1}{0}\overset{0}{1}_2) = 2^5 + 2^4 + 2^2 + 2^{\$} = 53_{10}$$

Decimal to binary:

$$binary\,(53_{10}) = 2^5 + 2^4 + 2^2 + 2^{\$} = \overset{5}{1}\overset{4}{1}\overset{3}{0}\overset{2}{1}\overset{1}{0}\overset{0}{1}_2$$

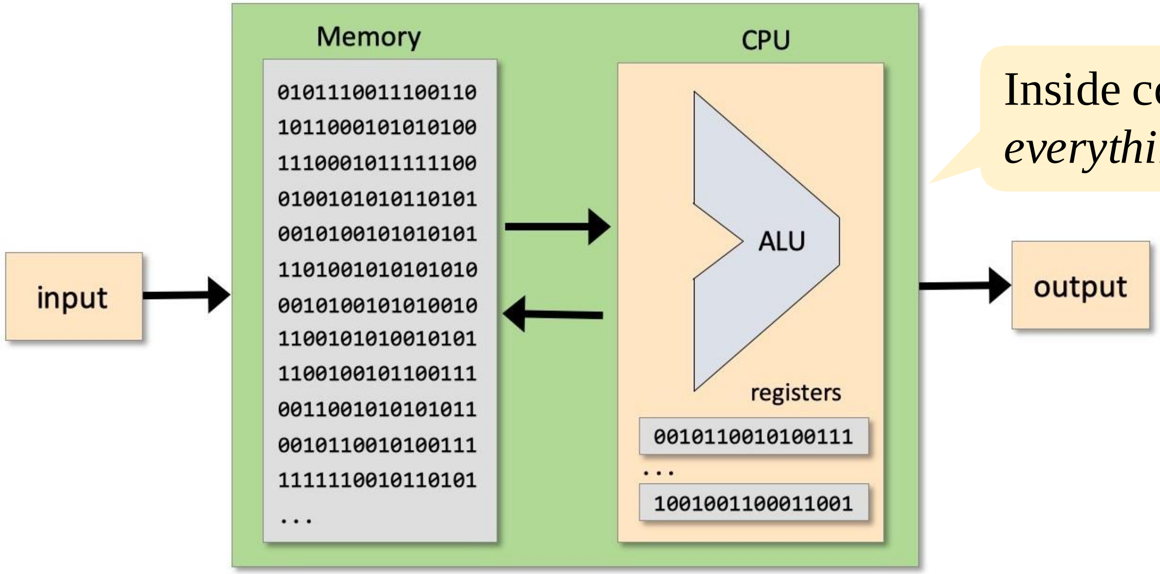
Algorithm: What is the largest power of 2 that “fits into” 53? It’s $32 = 2^5$. We still have to handle 53 – 32, so, what is the largest power of 2 that fits into 21? It’s $16 = 2^4$, and so on.

Practice:

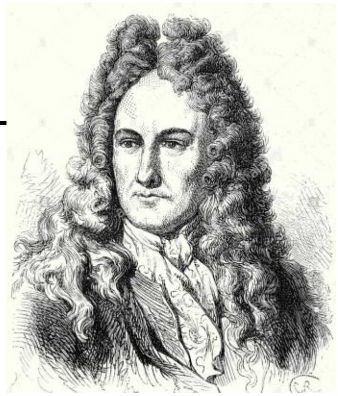
$$decimal\,(1011010_2) = 90_{10}$$

$$binary\,(523_{10}) = 1000001011_2$$

The binary system

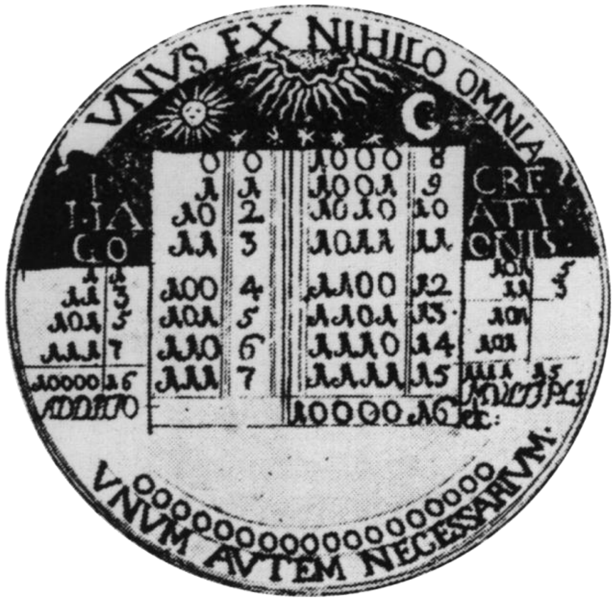


Inside computers, *everything* is binary



G.W. Leibnitz
(1646 – 1716)

Worshipped
binary numbers



<u>Binary numerals are easy to:</u>

- Compare
- Add
- Subtract
- Multiply
- Divide
- ...

- Store
- Transmit
- Verify
- Correct
- Compress
- ...

Chapter 2: Boolean Arithmetic

Theory

- ✓ Representing numbers
- ✓ Binary numbers
- ➡ Boolean arithmetic
- Signed numbers

Practice

- Arithmetic Logic Unit (ALU)
- Project 2: Chips
- Project 2: Guidelines

Boolean arithmetic

We have to figure out efficient ways to perform, *on binary numbers*:

- Addition — We'll implement it using logic gates
- Subtraction — We'll get it for free
- Multiplication
- Division

Based on addition

Addition is the foundation of all arithmetic operations.

Addition

	Binary addition						Decimal addition			
	0	0	1	0			0	1	1	0
+		1	0	1	0	+		7	8	7	5
				1	1				5	6	2
		1	1	0	1			8	4	3	7

Addition

Computers represent integers using a fixed number of bits,
sometimes called “word size”. For example, let’s assume n = 4:

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

Binary addition

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

Another example

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

Overflow

Handling overflow

- Our decision: Ignore it
- As we will soon see, ignoring the overflow bit is not a bug, it’s a feature.

Addition

Word size n = 16, 32, 64, …

	0	…	0	0	0	0	0	1	1	0	1	1	1	0	0	0
	0	…	0	0		0	0	0	1	1	0	1		1	0	1
+	0	…	0	0		0	0	0	0	1	0	1		1	0	0
	0	…	0	0		0	0	1	0	0	1	1		0	0	1

Same addition algorithm for any n

<u>Hardware implementation</u>

We'll build an *Adder* chip that implements this addition algorithm,

Using the chips built in project 1.

How? Later.

<u>Teaching Note</u>

In Nand to Tetris we always separate *abstraction* from *implementation*

First we present the abstraction, leaving the implementation to a later stage in the lecture.

Chapter 2: Boolean Arithmetic

Theory

- ✓ Representing numbers
- ✓ Binary numbers
- ✓ Boolean arithmetic (addition)
- Signed numbers

  $(x + y, -x + y, x + -y, -x + -y)$

Practice

- Arithmetic Logic Unit (ALU)
- Project 2: Chips
- Project 2: Guidelines

Signed integers

- Positive
- 0
- Negative

In most programming languages, the `short`, `int`, and `long` data types use 16, 32, and 64 bits for representing signed integers

Arithmetic operations on signed integers (x `op` y, $-x$ `op` y, x `op` $-y$, $-x$ `op` $-y$, where `op` = {`+`, `-`, `*`, `/`}) are by far what computers do most of the time

Therefore …

Efficient algorithms for handling arithmetic operations on signed integers hold the key to building efficient computers.

Teaching Note: All the algorithms presented in this course can be implemented efficiently in either hardware of software.

Signed integers

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

This particular example: word size is $n = 4$

In general, n bits allow representing all the unsigned integers $0 \ldots 2^n - 1$

<u>What about negative numbers?</u>

We can use half of the code space for representing positive numbers, and the other half for negatives.

Signed integers

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

<u>Representation:</u>

Left-most bit (MSB): Represents the sign, +/-

Remaining bits: Represent a positive integer

<u>Issues</u>

- – 0: Huh?
- $code(x) + code(-x) \neq code(0)$
- The codes are not monotonically increasing
- more complications.

Two's complement

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

The representation

- Assumption: Word size = n bits
- The "two's complement" of x is defined to be $2^n - x$
- The negative of x is coded by the two's complement of x

From decimal to binary:

if $x \geq 0$ return $binary(x)$

else return $binary(2^n - x)$

From binary to decimal:

if MSB = 0 return *decimal*(*bits*)

else return "–" and then (2^n – *decimal*(*bits*))

Two's complement: Addition

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

Compute $x + y$ where x and y are signed

Algorithm: Regular addition, modulo 2^n

$$6 + (-2) = 6 + 14 = 20 \,\%\, 16 = 4 \quad \text{codes } 4$$

$$3 + (-5) = 3 + 11 = 14 \,\%\, 16 = 14 \quad \text{codes } -2$$

$$-2 + (-5) = 14 + 11 = 25 \,\%\, 16 = 9 \quad \text{codes } -7$$

Two's complement: Addition

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

Compute $x + y$ where x and y are signed

Algorithm: Regular addition, modulo 2^n

$$\begin{array}{r} 6 \\ + \\ -2 \end{array} \quad = \quad \begin{array}{r} 6 \\ + \\ 14 \\ \hline 20 \end{array} \; \% \; 16 = 4 \text{ codes } 4$$

Practice:

$$\begin{array}{r} 4 \\ + \\ -7 \end{array} \quad = \quad ?$$

$$\begin{array}{r} -2 \\ + \\ -4 \end{array} \quad = \quad ?$$

Two's complement: Addition

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

Compute $x + y$ where x and y are signed

Algorithm: Regular addition, modulo 2^n

$$6 + (-2) \;=\; 6 + 14$$

20 % 16 = 4 codes 4

Practice:

$$4 + (-7) \;=\; 4 + 9$$

13 % 16 = 13 codes –3

$$-2 + (-4) \;=\; 14 + 12$$

26 % 16 = 10 codes –6

Two’s complement: Addition

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

<u>At the binary level</u> (same algorithm):

$$\begin{array}{r} 6 \\ +\;-2 \end{array} \;=\; \begin{array}{r} 0110 \\ +\;1110 \\ \hline \cancel{1}0100 \end{array} \quad \text{codes } 4$$

Ignoring the overflow bit is the binary equivalent of modulo 2^n

$$\begin{array}{r} 3 \\ +\;-5 \end{array} \;=\; \begin{array}{r} 0011 \\ +\;1011 \\ \hline 1110 \end{array} \quad \text{codes } -2$$

$$\begin{array}{r} -2 \\ +\;-5 \end{array} \;=\; \begin{array}{r} 1110 \\ +\;1011 \\ \hline \cancel{1}1001 \end{array} \quad \text{codes } -7$$

Two's complement: Addition

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

At the binary level (same algorithm):

$$\begin{array}{r} 6 \\ + \; -2 \end{array} \;=\; \begin{array}{r} 0110 \\ + \; 1110 \\ \hline \cancel{1}0100 \end{array} \quad \text{codes } 4$$

More examples:

$$\begin{array}{r} 5 \\ + \; 7 \end{array} \;=\; \begin{array}{r} 0101 \\ + \; 0111 \\ \hline 1100 \end{array} \quad \text{codes } -4 \qquad 5 + 7 = -4 \qquad ???$$

$$\begin{array}{r} -7 \\ + \; -3 \end{array} \;=\; \begin{array}{r} 1001 \\ + \; 1101 \\ \hline \cancel{1}0110 \end{array} \quad \text{codes } 6 \qquad -7 + -3 = 6 \qquad ???$$

Overflow detection

When you add up two positives (negatives) and get a negative (positive) result, you know that you have overflow

Two's complement: Subtraction

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

Compute $x - y$ where x and y are signed

- $x - y$ is the same as $x + (-y)$
- So… convert y and add up the two values
  (we already know how to add up signed numbers)

But … How to convert a number (efficiently)?

Two's complement: Sign conversion

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

<u>Compute $-x$</u> from x

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

$= 1 + (\texttt{1111}) - x$

$= 1 + flippedBits(x)$

<u>Algorithm:</u> To convert *bbb...b*:

Flip all the bits and add 1 to the result

Example: Convert `0010` (2)

```
  1101  (flipped)
+    1
  ----
  1110  (–2)
```

Two's complement: Sign conversion

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

<u>Compute $-x$</u> from x

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

$= 1 + (1111) - x$

$= 1 + flippedBits(x)$

<u>Algorithm:</u> To convert *bbb...b*:

Flip all the bits and add 1 to the result

Practice: Convert `1010` (–6)

Two's complement: Sign conversion

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

Compute $-x$ from x

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

$= 1 + (1111) - x$

$= 1 + flippedBits(x)$

Algorithm: To convert *bbb...b*:

Flip all the bits and add 1 to the result

Practice: Convert `1010` (–6)

```
  0101  (flipped)
+    1
  ----
  0110  (6)
```

But… How to compute $x + 1$ (*efficiently*)?

Two's complement: Add 1

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

Compute $x + 1$ (efficiently)

Given *bbb*...*b*, compute *bbb*...*b* + 1

Algorithm: Flip bits from right to left,
stop when the flipped bit becomes 1

Example: Compute `0101` + 1 (5 + 1)

`0110` (6)

Practice: Compute `0110` + 1 (6 + 1)

Compute `0011` + 1 (3 + 1)

Compute `1000` + 1 (-8 + 1)

Compute `1011` + 1 (-5 + 1)

Two's complement: Recap

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

Observations

- The method represents all the integers in the range $-2^{n-1}, \ldots, -1, 0, 1, \ldots, 2^{n-1}-1$
- $code(x) + code(-x) = code(0)$
- The codes are monotonically increasing
- Arithmetic on signed integers is the same as arithmetic on unsigned integers
- Simple! Elegant! Powerful!

Implications for hardware designers

Arithmetic on signed integers can be implemented using ***the same hardware*** used for handling arithmetic of unsigned integers

Chapter 2: Boolean Arithmetic

Theory

- Representing numbers
- Binary numbers
- Boolean arithmetic
- Signed numbers

Practice

- Arithmetic Logic Unit (ALU)
- Project 2: Chips
- Project 2: Guidelines

Chapter 2: Boolean Arithmetic

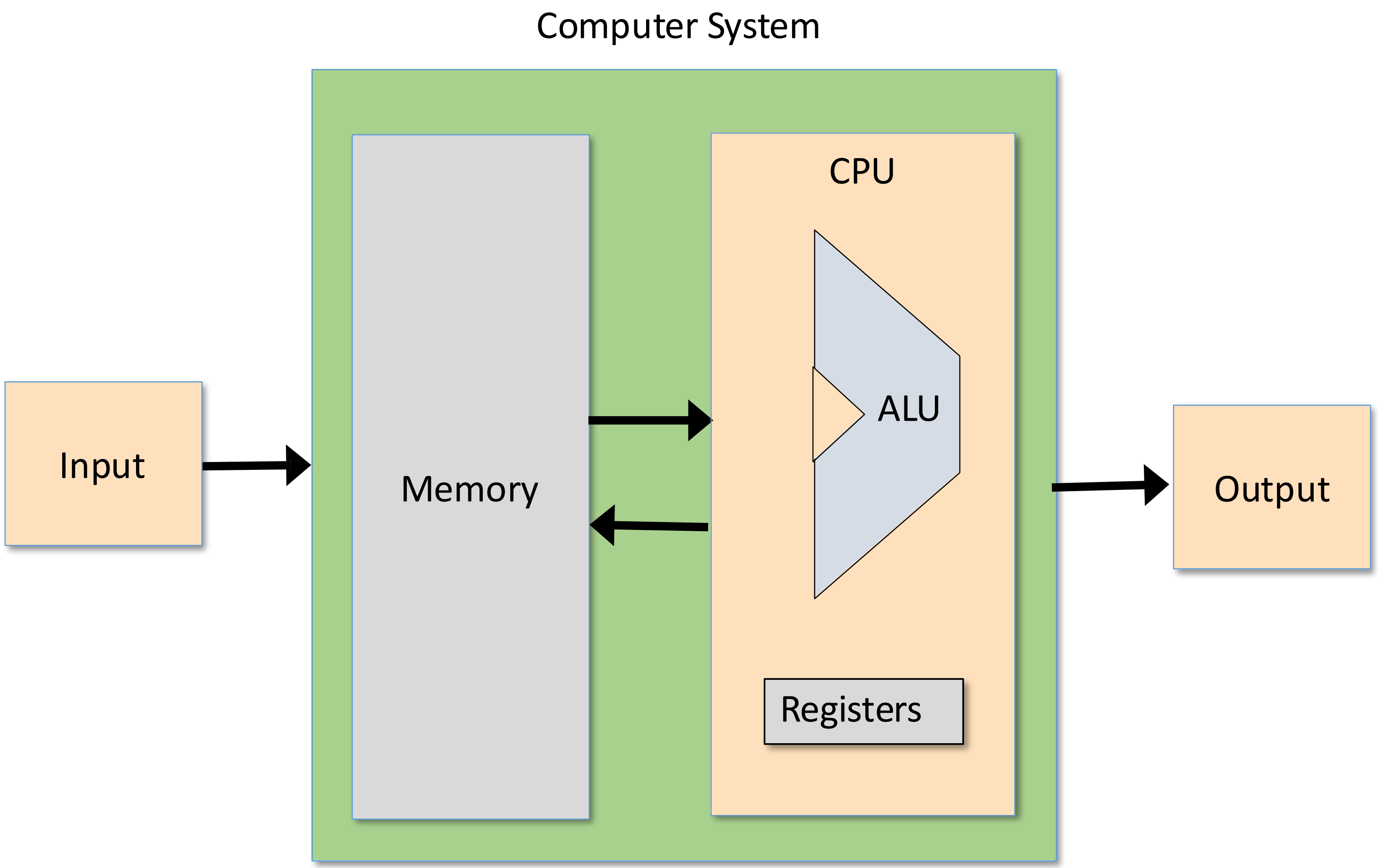
Theory

- Representing numbers
- Binary numbers
- Boolean arithmetic
- Signed numbers

Practice

- Arithmetic Logic Unit (ALU)
- Project 2: Chips
- Project 2: Guidelines

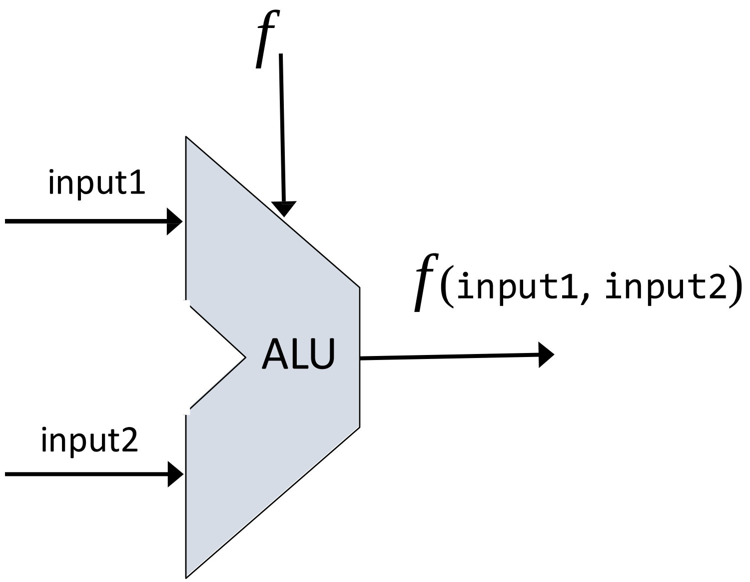
Von Neumann Architecture



The Arithmetic Logical Unit

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

f : one out of a family of pre-defined arithmetic functions (*add*, *subtract*, *multiply*…) and logical functions (*And*, *Or*, *Xor*, …)



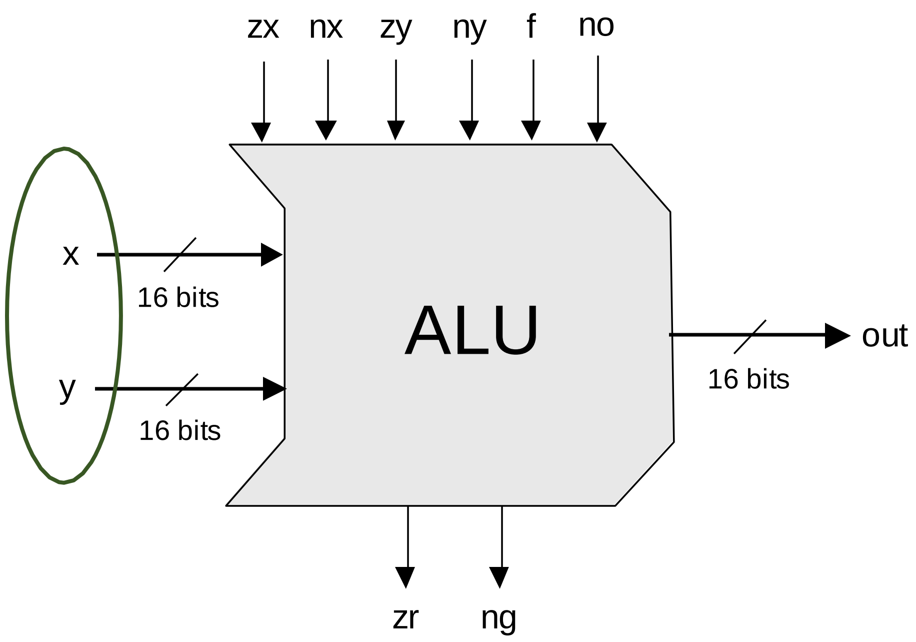
<u>Design issue: Which functions should the ALU perform?</u>

A hardware / software tradeoff: Any function not implemented by the ALU can be implemented later in system software

- Hardware implementations: Faster, and more expensive
- Software implementations: Slower, less expensive

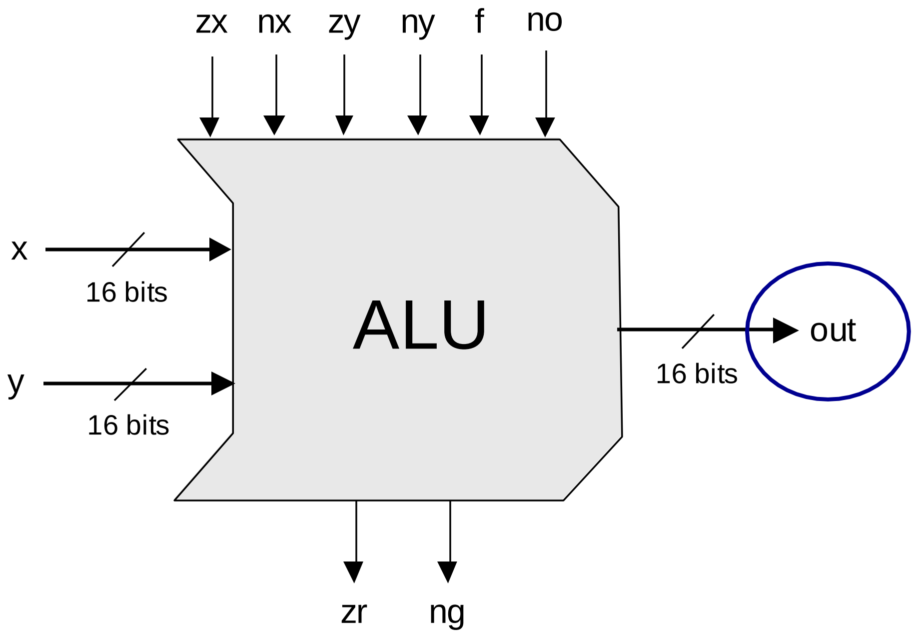
The Hack ALU

- Operates on two 16-bit, two's complement values



The Hack ALU

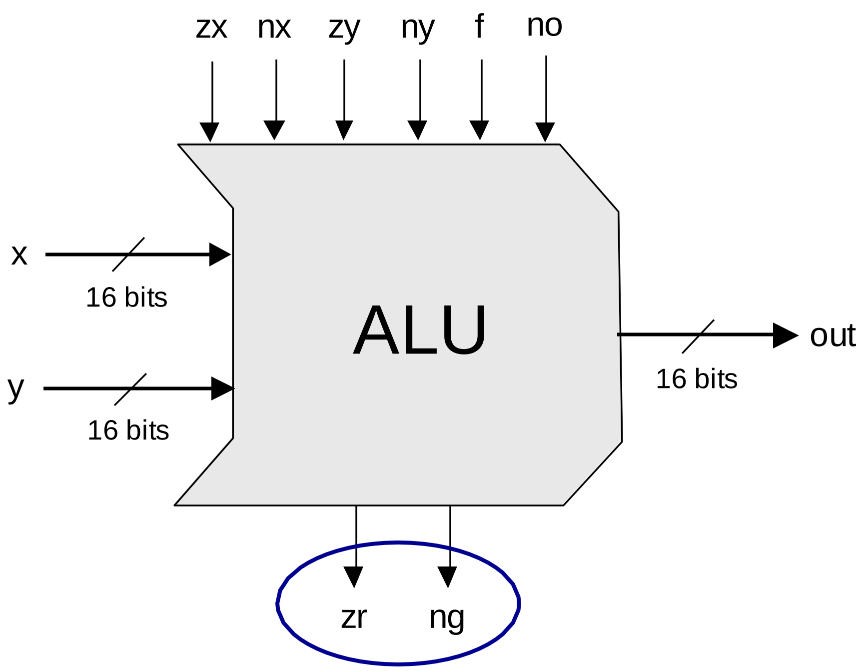
- Operates on two 16-bit, two's complement values
- Outputs a 16-bit, two's complement value



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

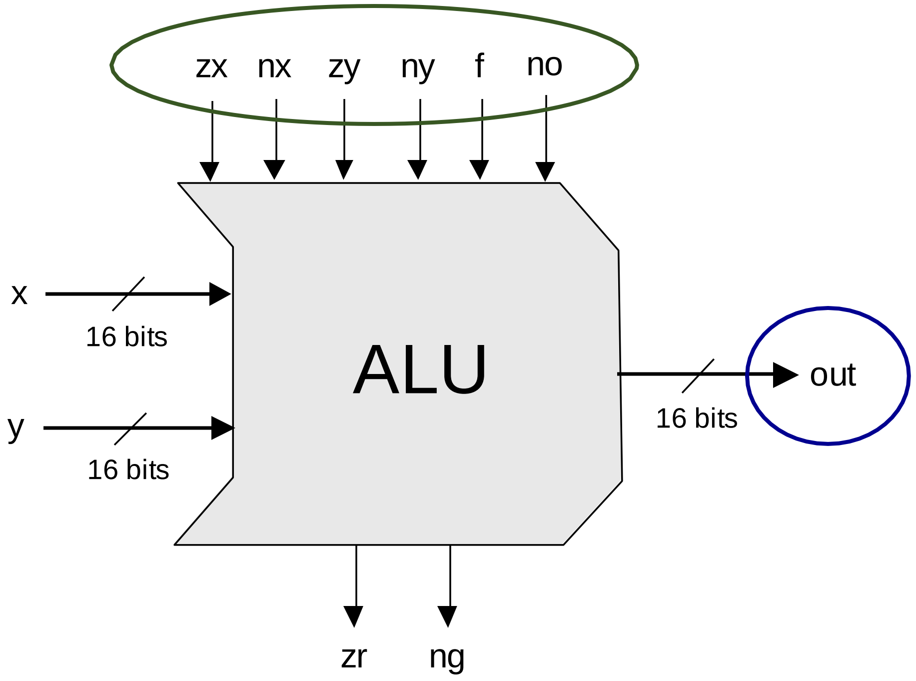
- Operates on two 16-bit, two's complement values
- Outputs a 16-bit, two's complement value
- Also outputs two 1-bit values (later)



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

- Operates on two 16-bit, two's complement values
- Outputs a 16-bit, two's complement value
- Also outputs two 1-bit values (later)
- Which function to compute is set by six 1-bit inputs

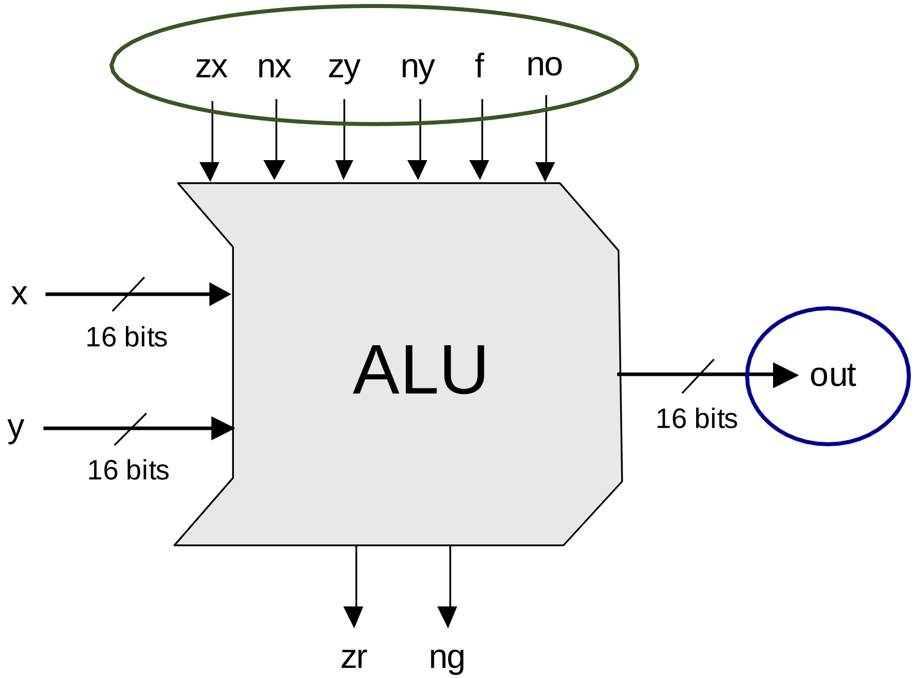


out
0
1
-1
x
y
!x
!y
-x
-y
x+1
y+1
x-1
y-1
x+y
x-y
y-x
x&y
x|y

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.



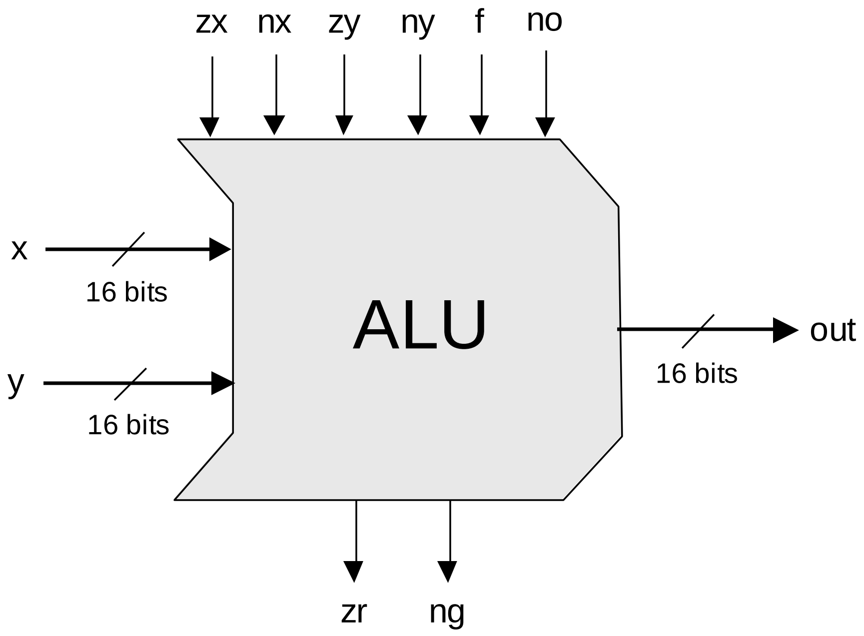
control bits

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

To cause the ALU to compute a function:

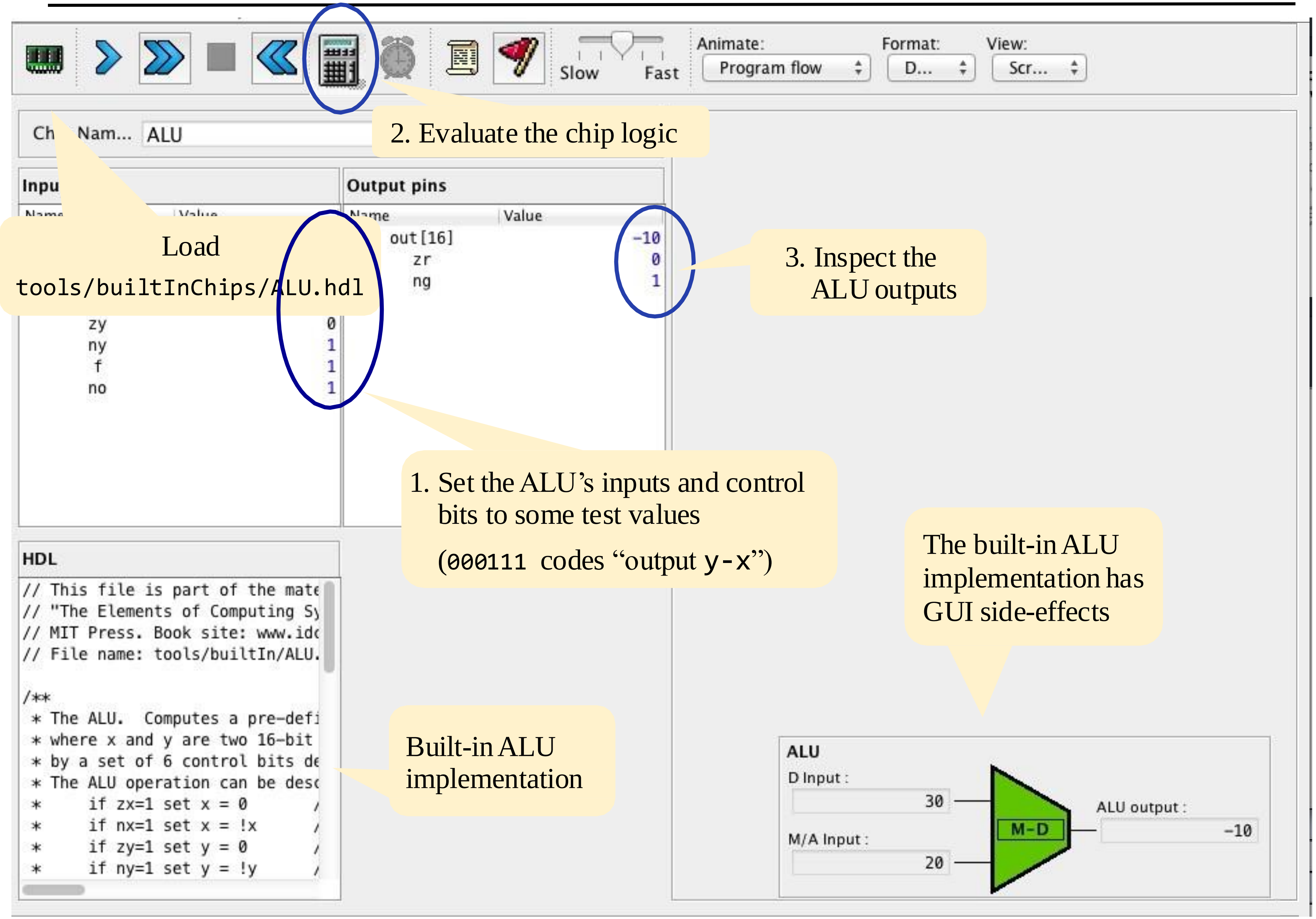
Set the control bits to one of the binary combinations listed in the table.



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

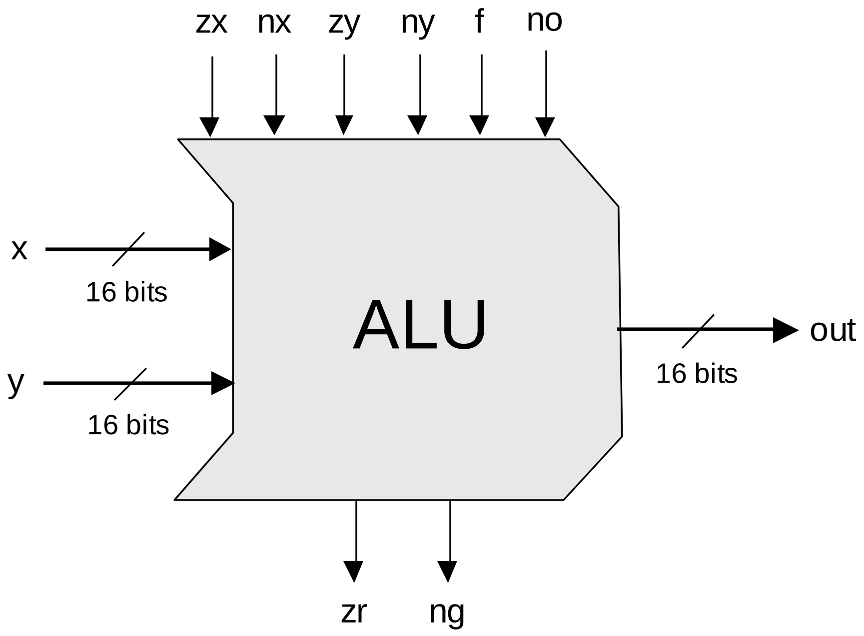
The Hack ALU in action: Compute `y-x`



The Hack ALU in action: Compute `x & y`

<u>To cause the ALU to compute a function:</u>

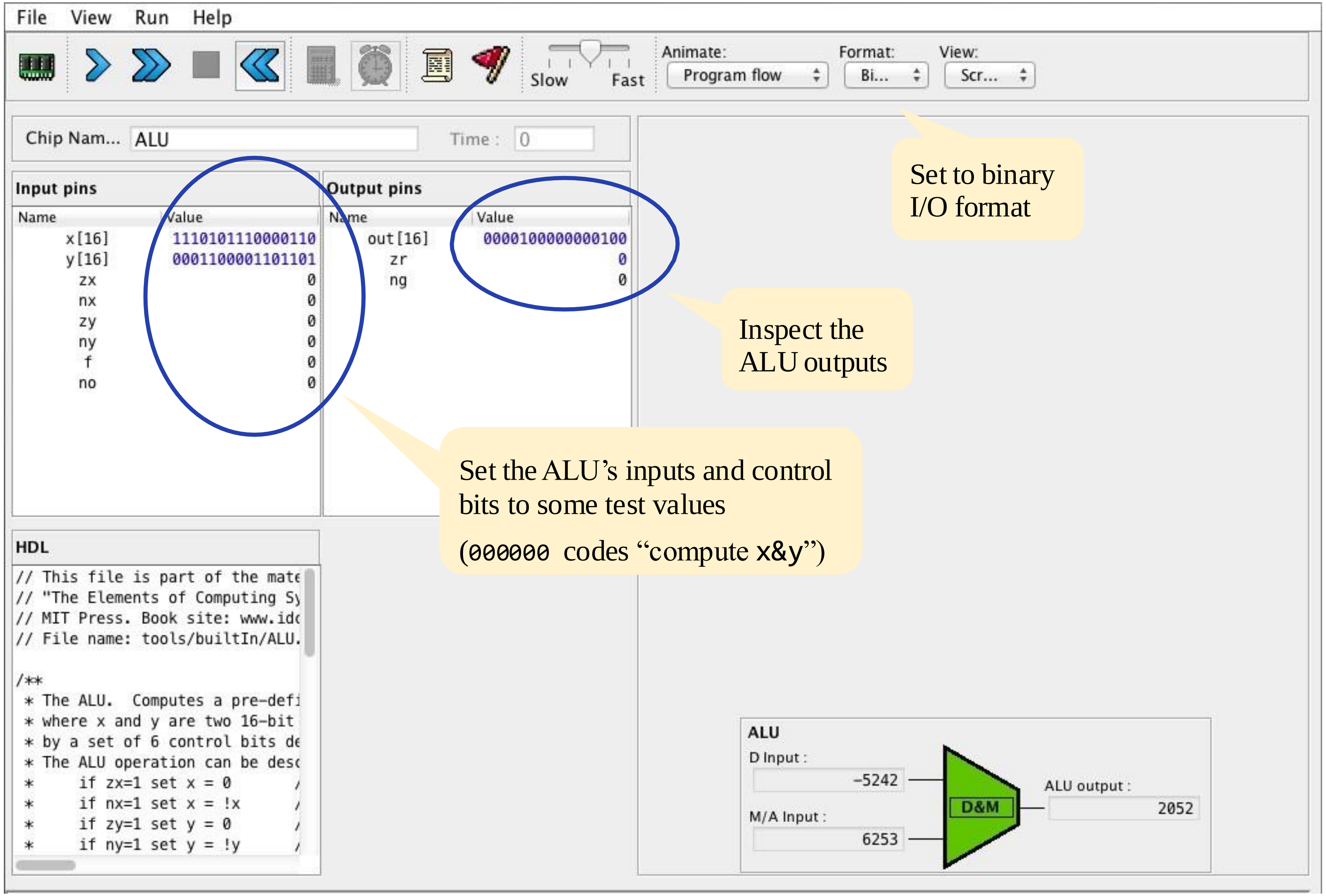
Set the control bits to one of the binary combinations listed in the table.



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 `x & y`



Set to binary I/O format

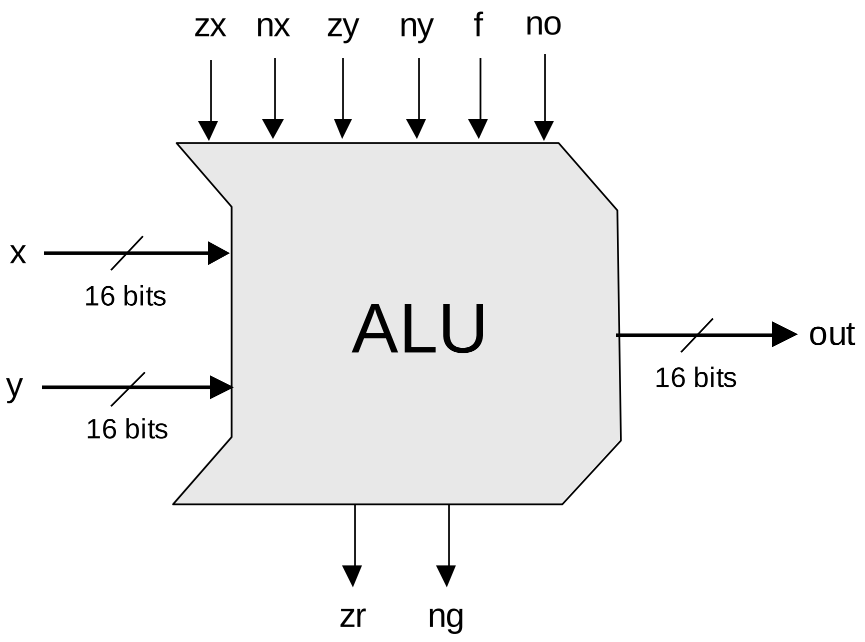
Inspect the ALU outputs

Set the ALU's inputs and control bits to some test values

(`000000` codes "compute `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)=`



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

The Hack ALU operation: 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	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

<u>Example</u>: 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)
```

The Hack ALU operation: 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	1	1	0			1
0	0	1	1			
1	1	0	0			
0	0	1	1			x
1	1	0	0			y
0	0	1	1			x
1	1	0	0			y
0	1	1	1			1
1	1	0	1			1
0	0	1	1			1
1	1	0	0			1
0	0	0	0			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 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-se tting:

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

The Hack ALU operation: Compute 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
```

Practice:

See if you get

0 1 1 1 (bitwise Or)

The Hack ALU operation: Compute 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	
1	0	1	0	1	0	
1	1	1	1	1	1	
1	1	1	0	1	0	
0	0	1	1	0	0	
1	1	0	0	0	0	
0	0	1	1	0	1	
1	1	0	0	0	1	
0	0	1	1	1	1	
1	1	0	0	1	1	
0	1	1	1	1	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

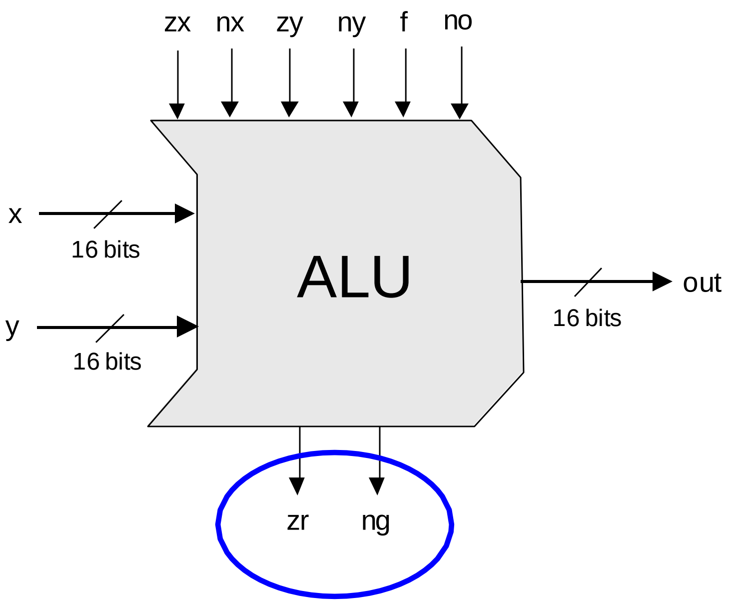
Practice:

See if you get

0 1 0 1 (5)

The Hack ALU operation

One more detail:



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

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

The `zr` and `ng` output bits will come into play when we'll build the complete CPU architecture, later in the course.

Chapter 2: Boolean Arithmetic

Theory

- Representing numbers
- Binary numbers
- Boolean arithmetic
- Signed numbers

Practice

- ✓ Arithmetic Logic Unit (ALU)
- → Project 2: Chips
- Project 2: Guidelines

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

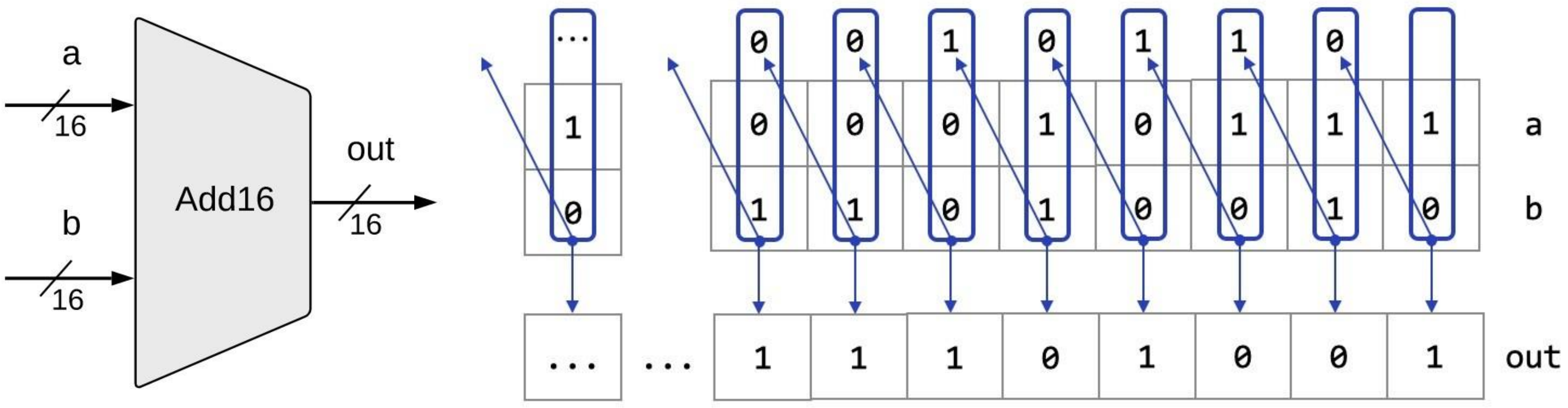
FullAdder.hdl

```
/** 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:
}
```

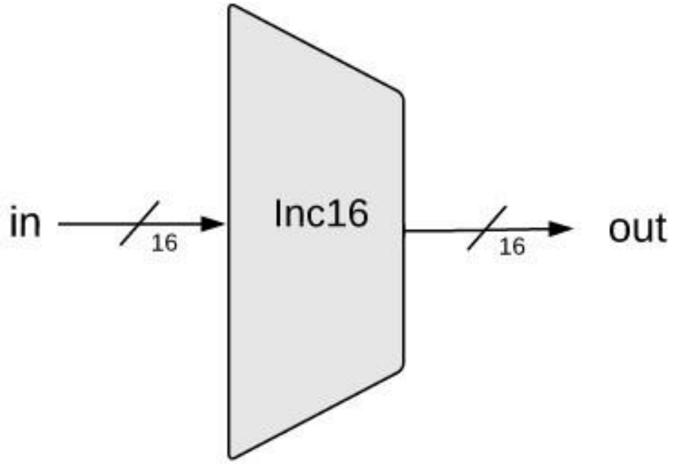
- The bitwise additions are done in parallel
- The carry propagation is sequential
- Yet… it works fine, as is.
  How? Stay tuned for chapter 3.

Implementation note

If you need to set a pin `x` to `0` (or `1`) in HDL,

use: `x = false` (or `x = true`)

16-bit incrementor



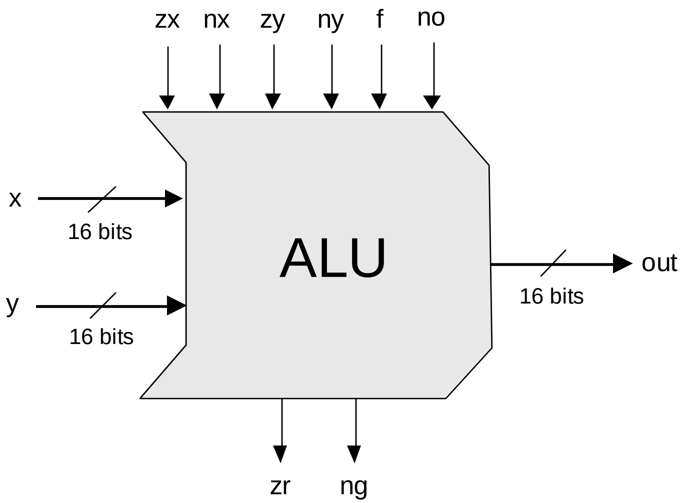
Inc16.hdl

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

<u>Implementation</u>:

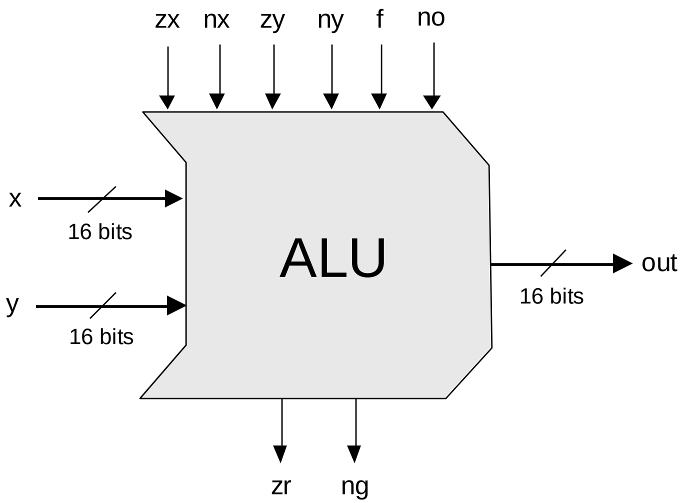
Simple.

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

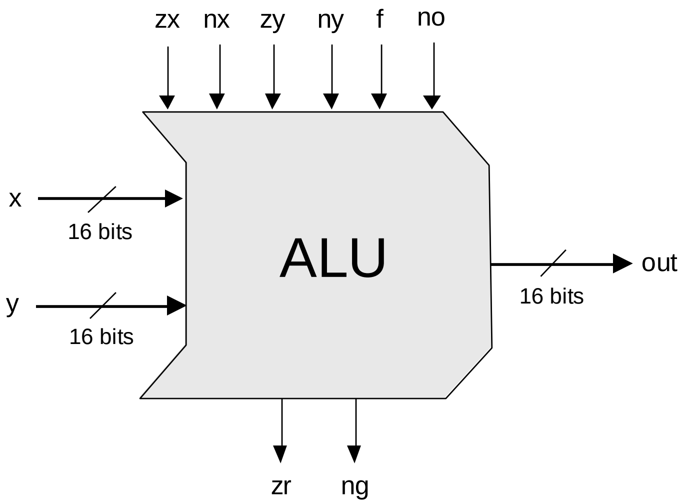


pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	**nx**	**zy**	**ny**	**f**	**no**	**out**
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	out(x,y)=

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



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 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 `Or` on two 16-bit values

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:

We can assign values to sub-buses

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

	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*

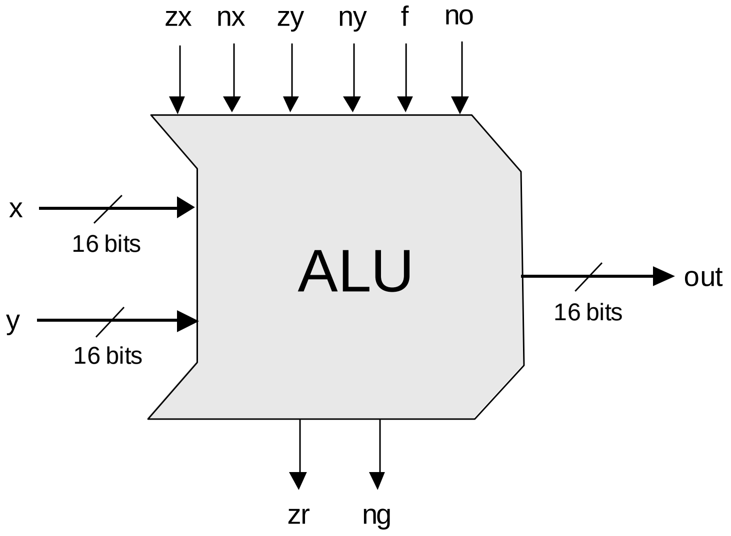
```
/* 16-bit adder */
CHIP Add16 {
    IN a[16], b[16];
    OUT out[16];

    PARTS:
    ...
}
```

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

ALU: Recap



The Hack ALU is:

- Simple
- Elegant

“Simplicity is the ultimate sophistication.”

— Leonardo da Vinci

Chapter 2: Boolean Arithmetic

Theory

- Representing numbers
- Binary numbers
- Boolean arithmetic
- Signed numbers

Practice

✓ Arithmetic Logic Unit (ALU)

✓ Project 2: Chips

➡ Project 2: Guidelines

Project 2

<u>Given:</u> All the chips built in Project 1

<u>Goal:</u> Build the chips:

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

Guidelines: www.nand2tetris.org/project02

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

Resources

Project 2 folder (`.hdl`, `.tst`, `.cmp` files):
`nand2tetris/projects/02`

Tools

- Text editor (for completing the given `.hdl` stub-files)
- Hardware simulator: `nand2tetris/tools`

Guides

- Hardware Simulator Tutorial
- HDL Guide
- Hack Chip Set API

Chip interfaces: Hack chip set API

Open the API in a window, and copy-paste chip signatures into your HDL code, as needed

```
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= ,a
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= ,o
```

```
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= );
```

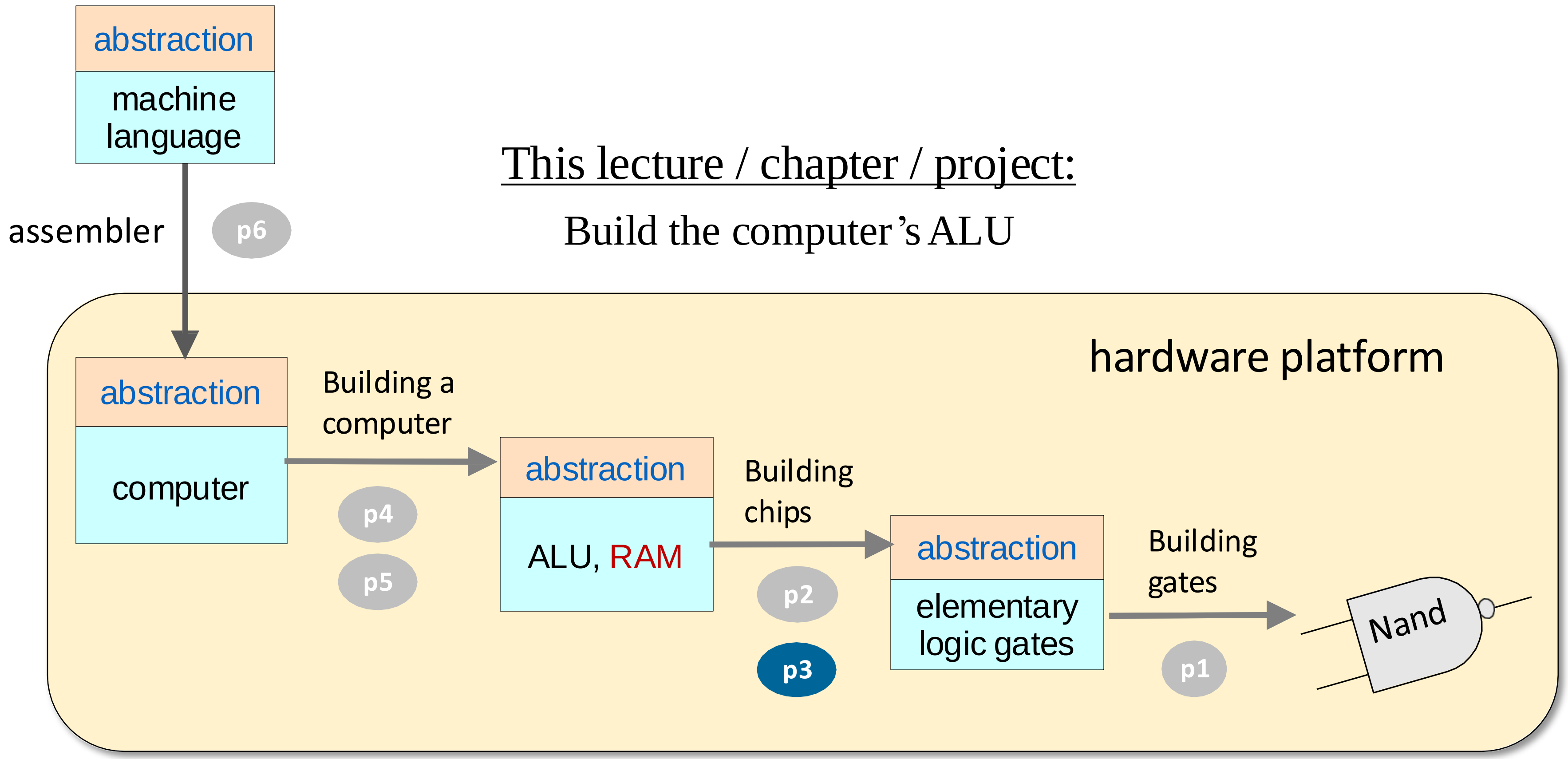
Best practice advice

- Implement the chips in the order in which they appear in the project guidelines
- If you don't implement some chips, you can still use their built-in implementations
- No need for "helper chips": Implement / use only the chips we specified
- In each chip definition, 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 HDL implementations.

That's It!

Go Do Project 2!

What’s next?



This lecture / chapter / project:

Build the computer’s ALU

Next lecture / chapter / project:

Build the computer’s RAM