Lecture 2

Boolean Arithmetic

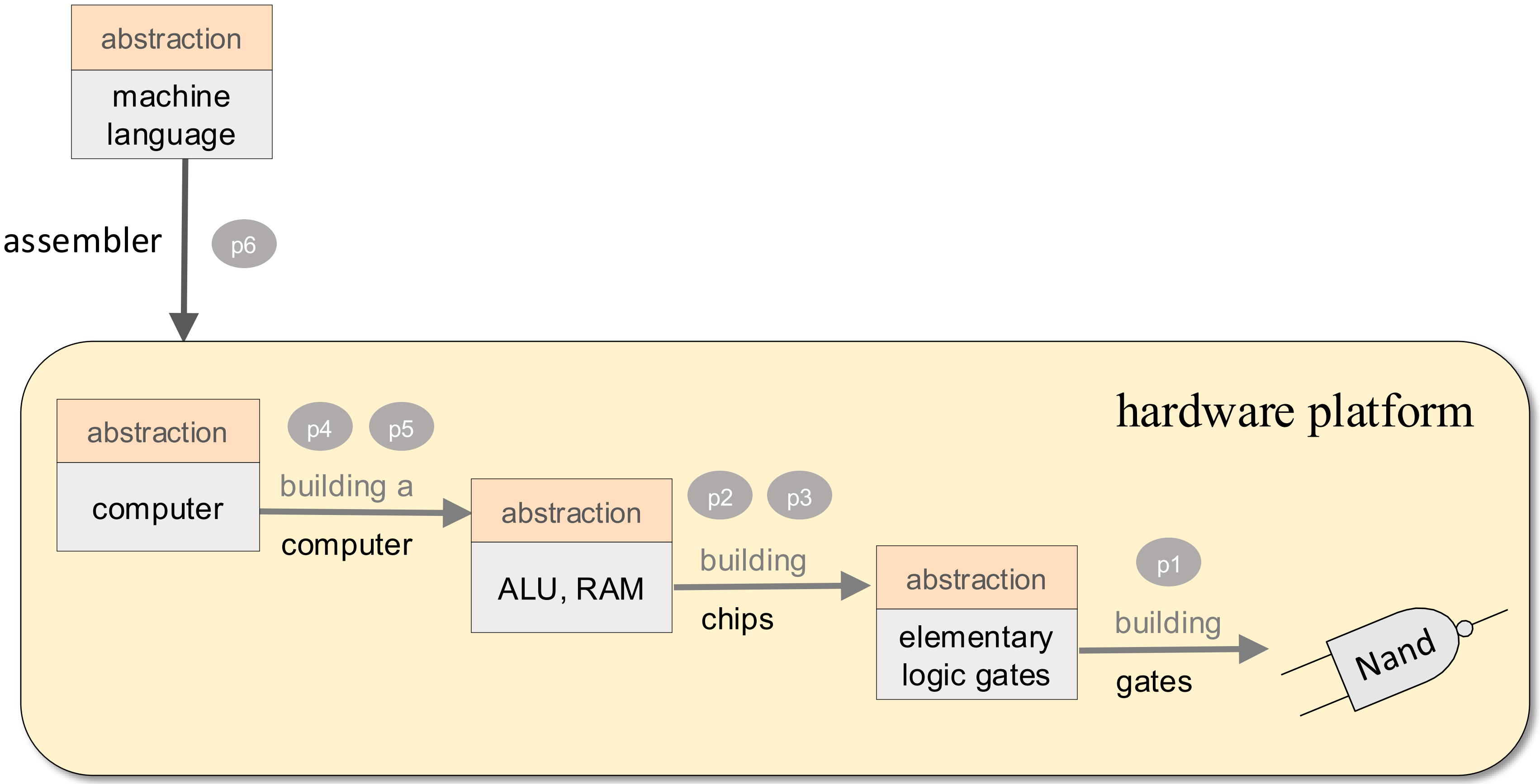
Slide deck for Chapter 2 of the book

The Elements of Computing Systems (2nd edition)

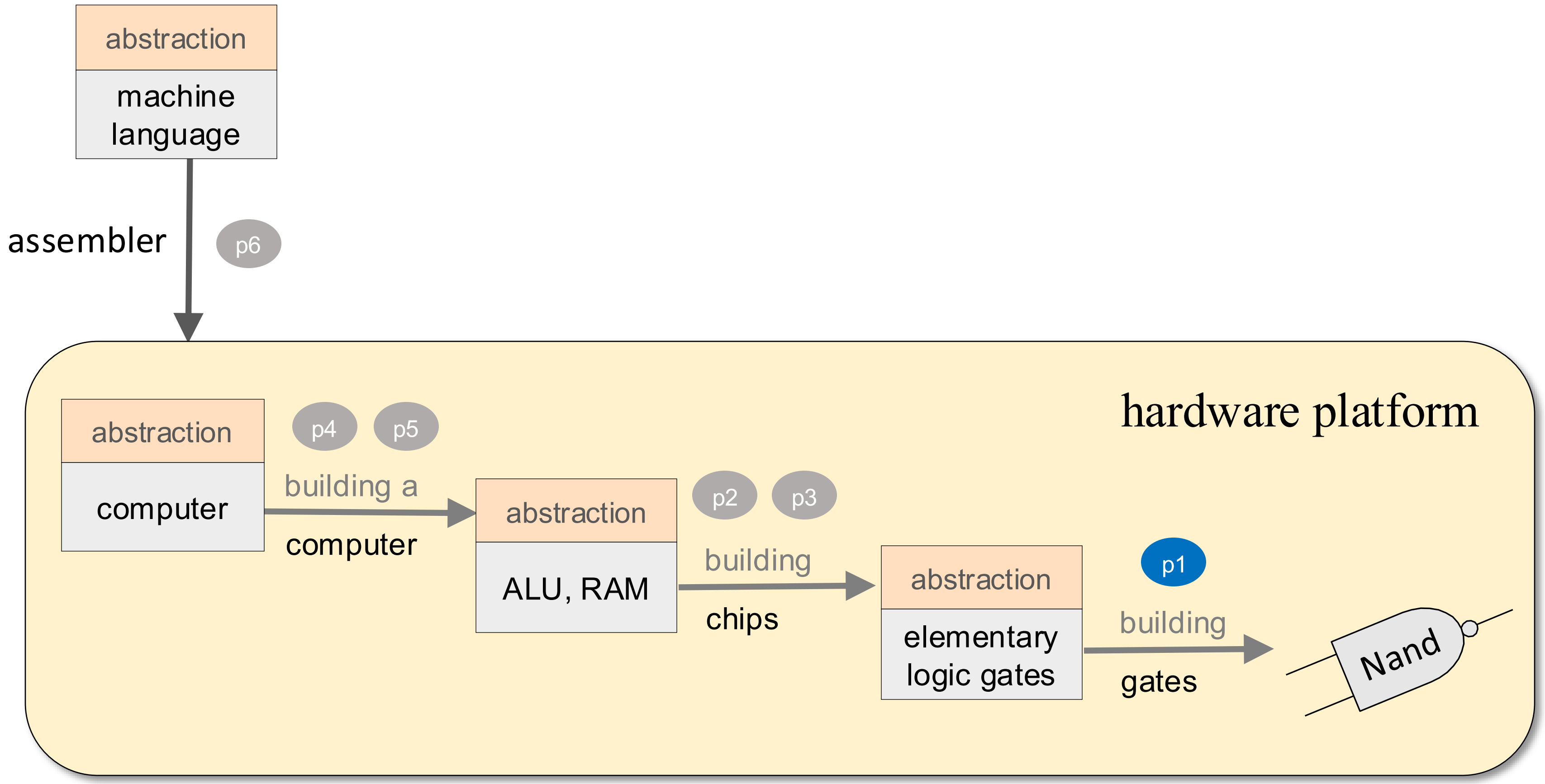
By Noam Nisan and Shimon Schocken

MIT Press

Nand to Tetris Roadmap: Hardware



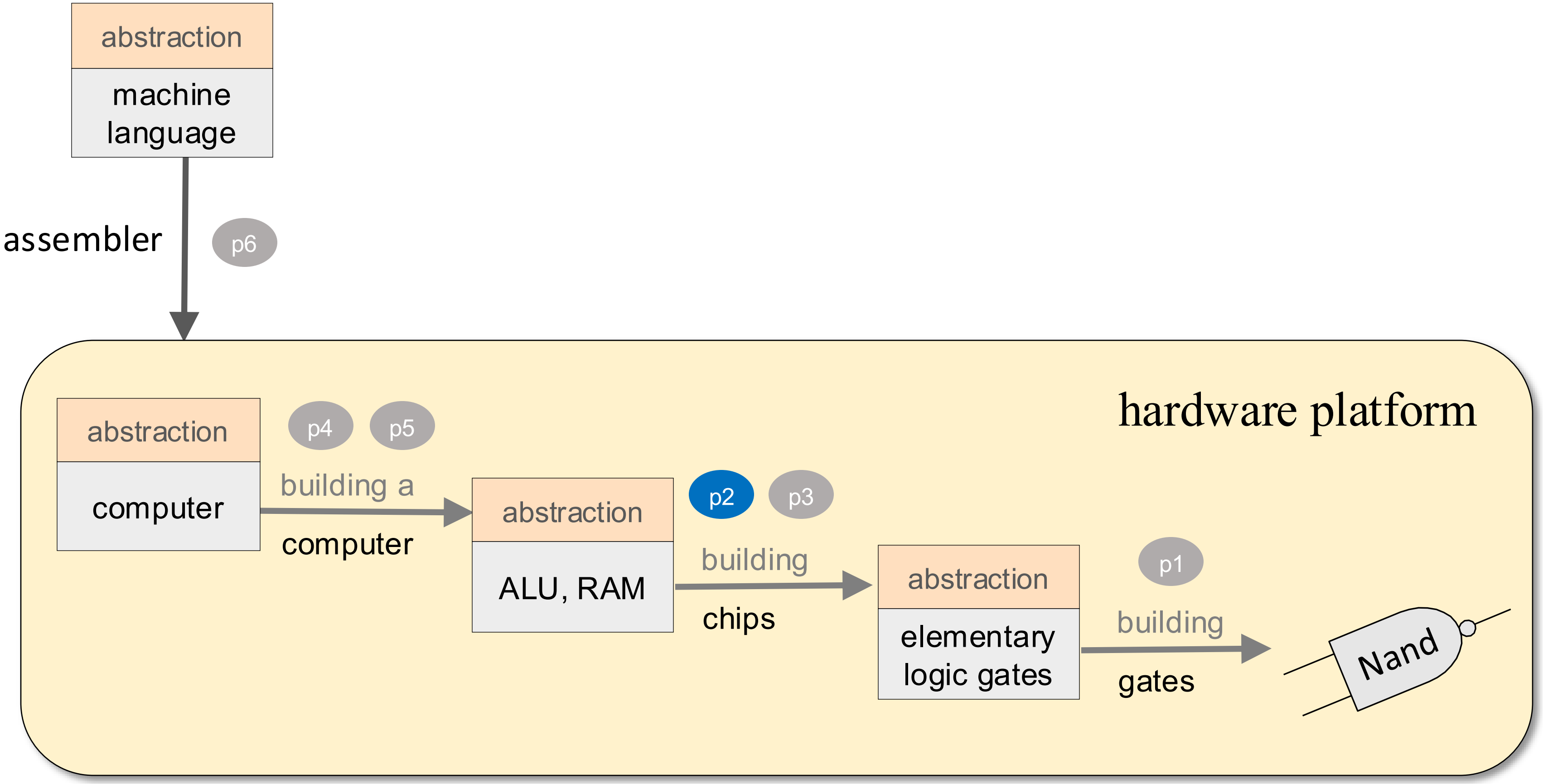
Nand to Tetris Roadmap: Hardware



Project 1

Build 15 elementary logic gates

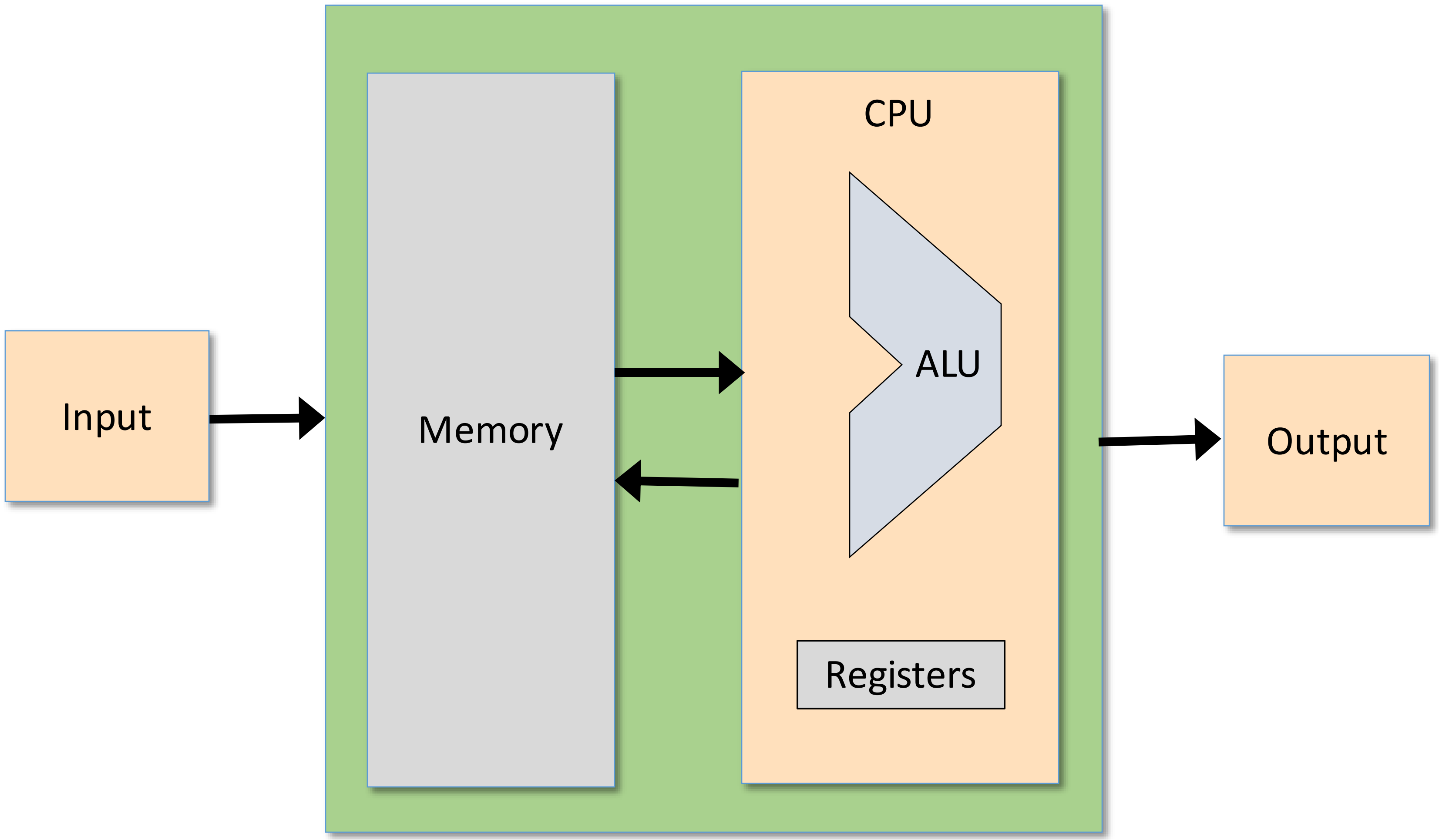
Nand to Tetris Roadmap: Hardware



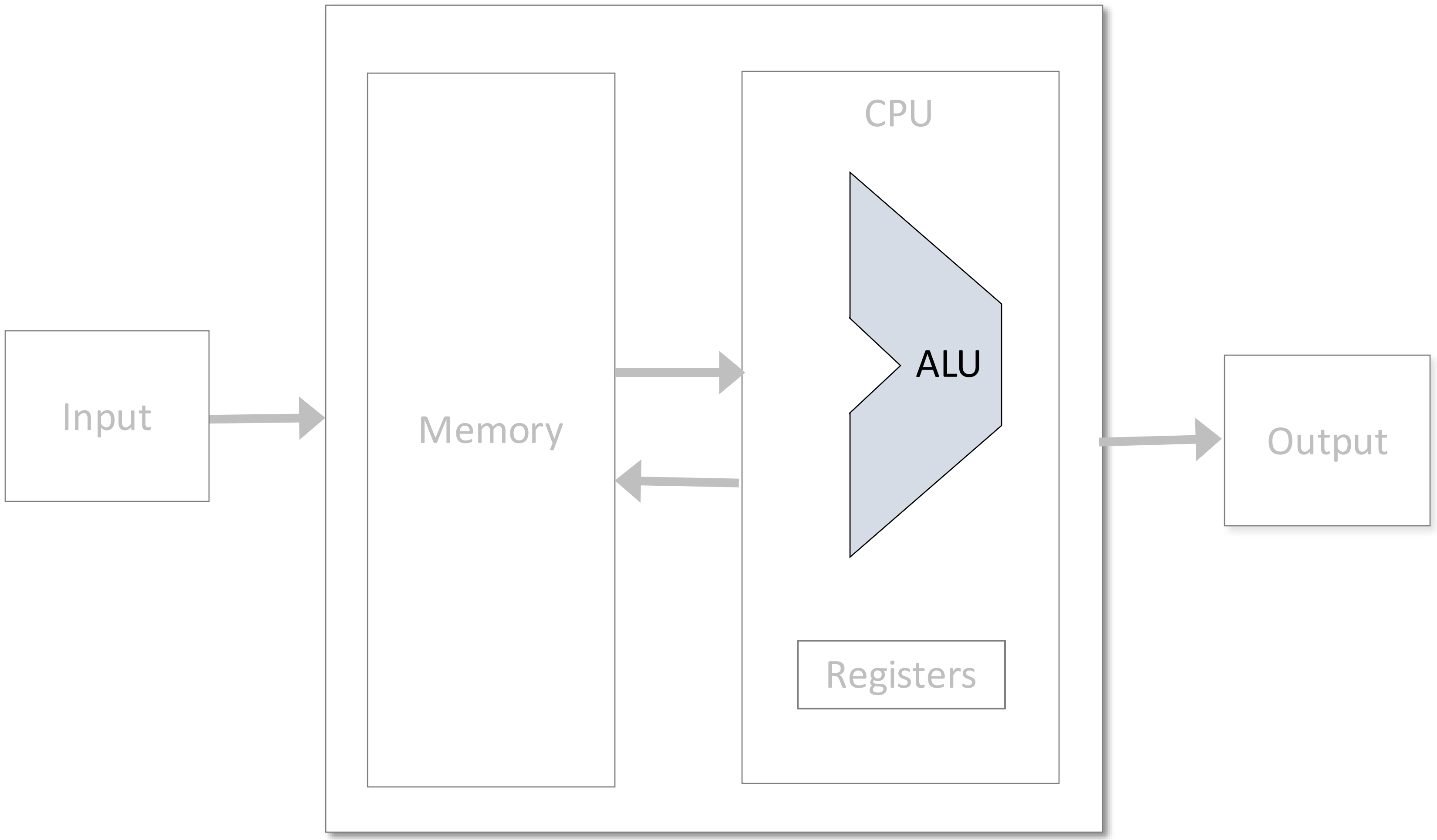
Project 2

Build chips that do arithmetic, leading up to an ALU

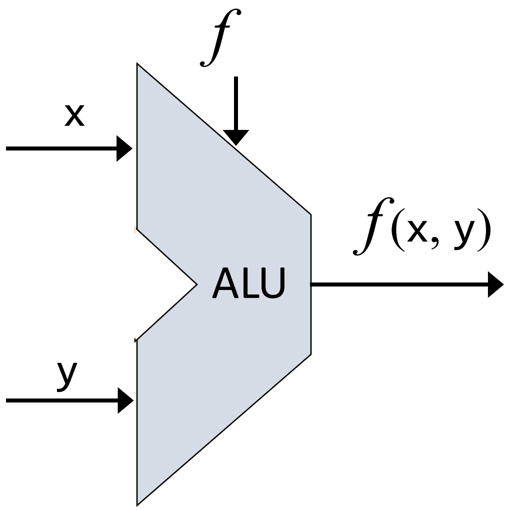
Computer system



Computer system



Arithmetic Logical Unit

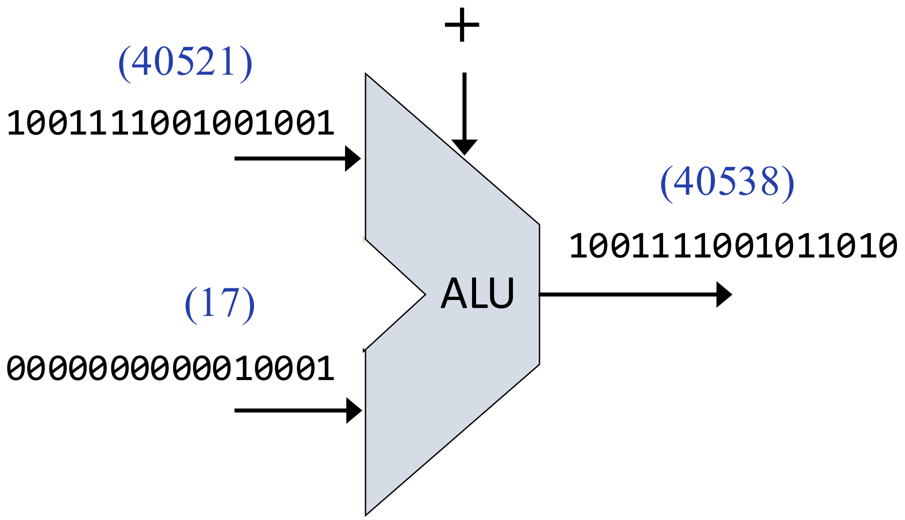


Computes a given function on two n-bit input values, and outputs an n-bit value

ALU functions (f)

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

Arithmetic Logical Unit



Computes a given function on two *n*-bit input values, and outputs an *n*-bit value

<u>ALU functions</u> (*f*)

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

<u>Challenges</u>

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

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

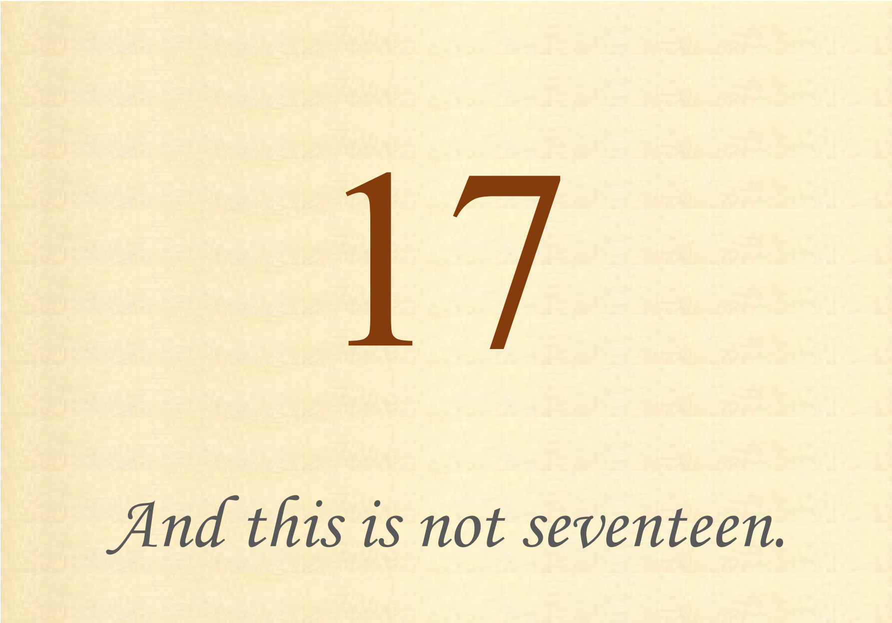
Practice

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

Representation

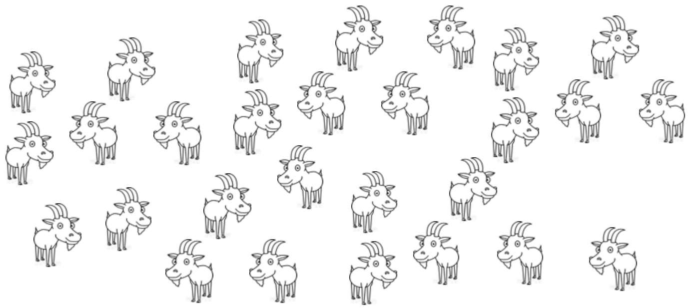
(by René Magritte)

Representation



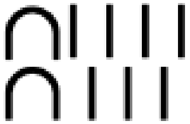
It’s an agreed-upon code (*numeral*)
used for representing the number seventeen.

A brief history of numeral systems



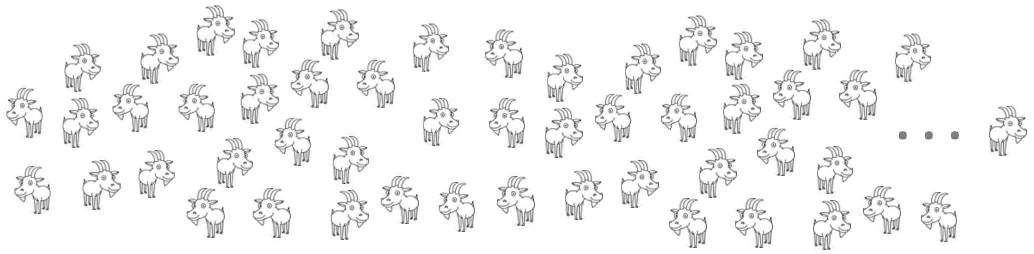
Twenty seven goats

Unary: 

Egyptian: 

Roman: XXVII

A brief history of numeral systems



Six thousands,
five hundreds,
and seven goats

Unary: 

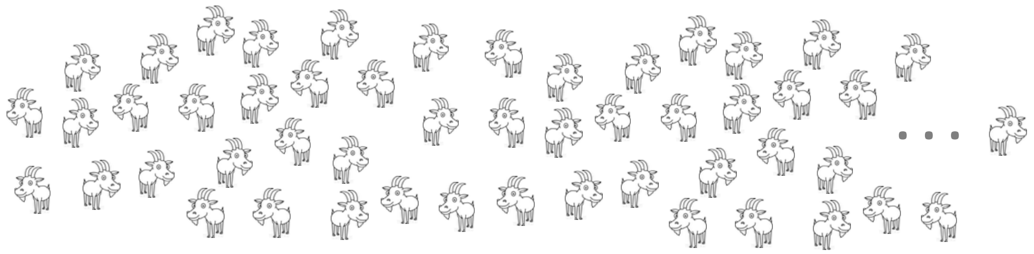
Egyptian: 

Roman: MMMMMMDVII

Old numeral systems:

- Don’t scale
- Cumbersome arithmetic
- Used until about 1,000 years ago
- Hindered the progress of Algebra
  (and commerce, science, technology)

Positional numeral system



Six thousands,
five hundreds,
and seven goats

$$\begin{array}{cccc} 3 & 2 & 1 & 0 \\ 6 & 5 & 0 & 7 \end{array}$$

$$\sum_{0}^{n-1} d_i \cdot 10^i = 6 \cdot 10^3 + 5 \cdot 10^2 + 0 \cdot 10^1 + 7 \cdot 10^0 = 6507$$

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

A crucial innovation, brought to the West from the East around 1200

Positional representation

Digits: A fixed set of symbols, including 0

Base: The number of symbols

Note: The method mentions no specific base.

Numeral: An ordered sequence of digits

Value: The digit at position i (counting from right to left, and starting at 0) encodes how many copies of $base^i$ are added to the value.

Chapter 2: Boolean Arithmetic

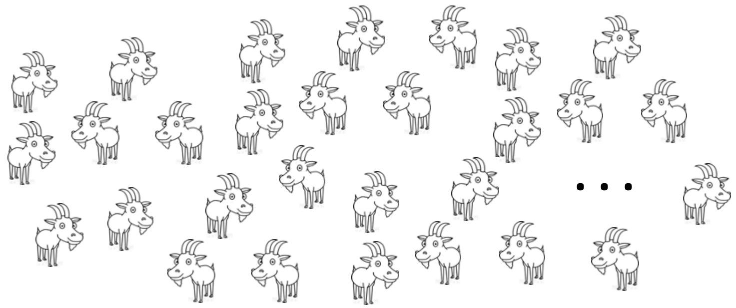
Theory

- Representing numbers
- Binary numbers
- Boolean arithmetic
- Representing signed numbers

Practice

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

Positional number system

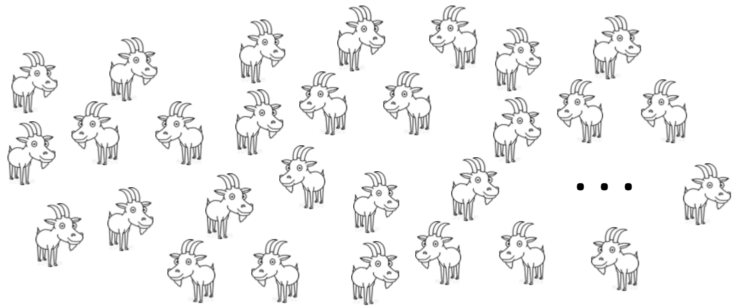


Seven thousands and fifty three goats

$$\begin{array}{cccc} \color{blue}{3} & \color{blue}{2} & \color{blue}{1} & \color{blue}{0} \\ 7 & 0 & 5 & 3_{10} \end{array}$$

$$\sum_{0}^{n-1} d_i \cdot \color{red}{10^i} = 7 \cdot 10^{\color{blue}{3}} + 0 \cdot 10^{\color{blue}{2}} + 5 \cdot 10^{\color{blue}{1}} + 3 \cdot 10^{\color{blue}{0}} = 7053$$

Positional number system



Seven thousands
and fifty three
goats

Decimal (base 10) system:
Human friendly

$$\begin{array}{cccc} 3 & 2 & 1 & 0 \\ 7 & 0 & 5 & 3 \end{array}_{10}$$

$$\sum_{0}^{n-1} d_i \cdot 10^i = 7 \cdot 10^3 + 0 \cdot 10^2 + 5 \cdot 10^1 + 3 \cdot 10^0 = 7053$$

Binary (base 2) system:
Computer friendly

$$\begin{array}{ccccccccccccc} 12 & 11 & 10 & & & \cdots & & & & 3 & 2 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{array}_{2}$$

$$\sum_{0}^{n-1} d_i \cdot 2^i = 1 \cdot 2^{12} + 1 \cdot 2^{11} + 0 \cdot 2^{10} + \cdots + 1 \cdot 2^0 = 7053$$

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 read and write numbers in base 10;

Computers represent and process numbers in base 2;

Therefore, for the sole purpose of interacting with humans, we need algorithms for base conversions.

Decimal ⟷ binary conversions

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:

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

Decimal to binary:

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

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

Practice:

$$\mathit{decimal}\,(1011010_2) = \ ?$$

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

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:

$$\mathit{decimal}\,(\overset{5\,4\,3\,2\,1\,0}{110101}_2) = 2^5 + 2^4 + 2^2 + 2^0 = 53_{10}$$

Decimal to binary:

$$\mathit{binary}\,(53_{10}) = 2^5 + 2^4 + 2^2 + 2^0 = \overset{5\,4\,3\,2\,1\,0}{110101}_2$$

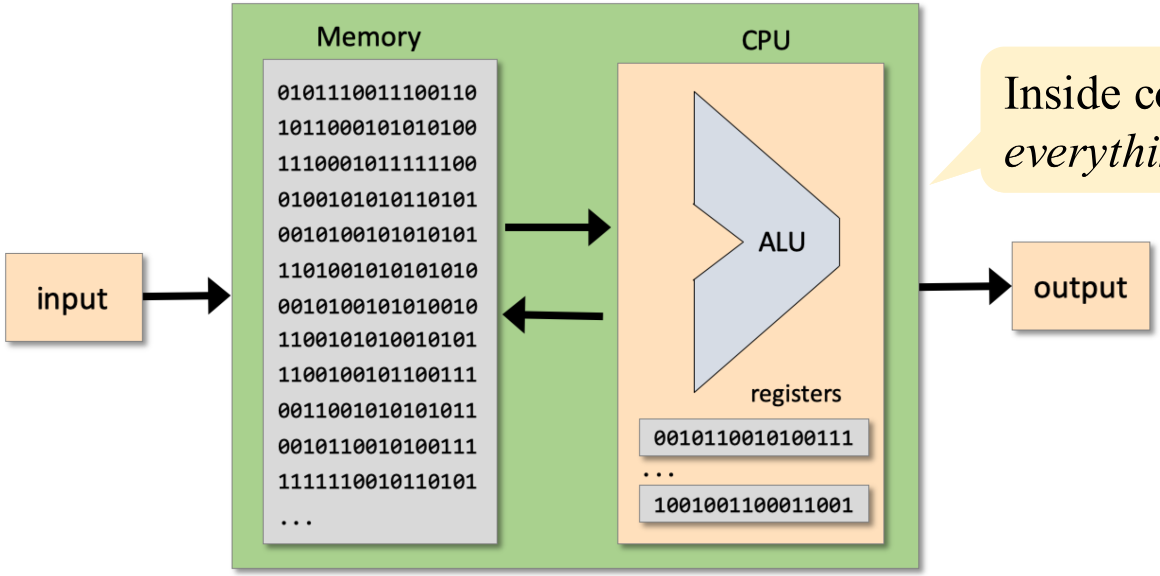
Algorithm: What is the largest power of 2 that "fits into" 53? It's $2^5 = 32$. We still have to represent 53 – 32, so, what is the largest power of 2 that fits into 21? It's $2^4 = 16$, and so on.

Practice:

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

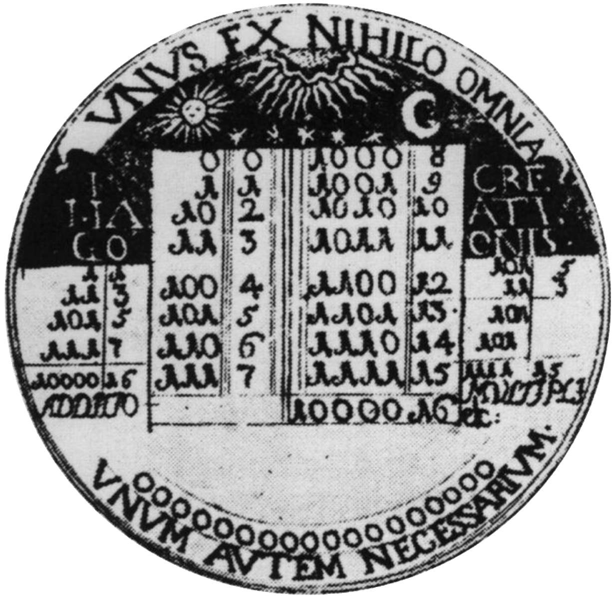
$$\mathit{binary}\,(523_{10}) = 1000001011_2$$

The binary system



Inside computers, *everything* is binary

G.W. Leibnitz
(1646 – 1716)



Leibnitz Medallion, 1697

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

Compare	Verify
Add	Correct
Subtract	Store
Multiply	Transmit
Divide	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
- Subtraction
- Multiplication
- Division

We'll implement addition using logic gates

We'll get it for free

We'll implement them (*efficiently*) using addition

Addition is the foundation of all arithmetic.

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
```

Binary addition

Decimal addition

Addition

Computers represent integers using a fixed number of bits.
For example, assume $n = 4$:

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

Binary addition

	0	0	0	1
+	0	1	0	1

Another example

Addition

Computers represent integers using a fixed number of bits.
For example, assume $n = 4$:

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

Binary addition

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

Another example

Addition

Computers represent integers using a fixed number of bits.
For example, assume $n = 4$:

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

Binary addition

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

Another example

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

Another example

Addition

Computers represent integers using a fixed number of bits.
For example, assume n = 4:

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

Binary addition

0	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

Another example

Overflow

Handling overflow

- Our approach: Ignore it
- As we’ll soon see, ignoring overflow is a sensible practice.

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

(Later).

Chapter 2: Boolean Arithmetic

<u>Theory</u>

✓ Representing numbers

✓ Binary numbers

✓ Boolean arithmetic (addition)

- Signed numbers

<u>Practice</u>

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

Chapter 2: Boolean Arithmetic

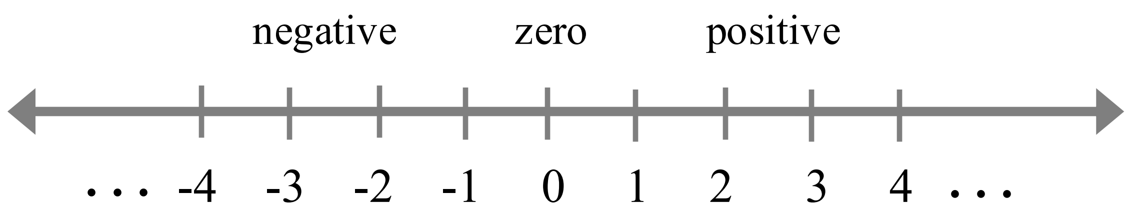
Theory

- ✓ Representing numbers
- ✓ Binary numbers
- ✓ Boolean arithmetic (addition)
- ➔ Signed numbers
  ($x + y$, $–x + y$, $x + –y$, $–x + –y$)

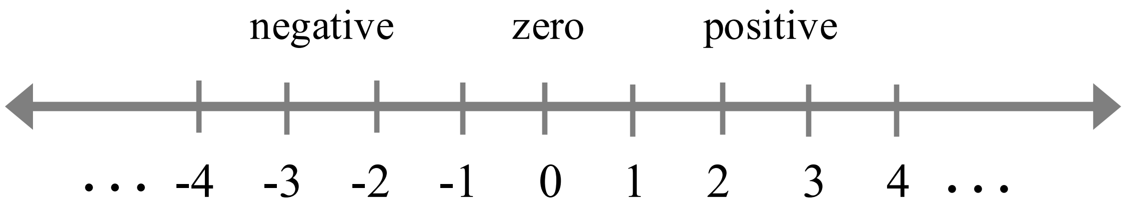
Practice

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

Signed integers



Signed integers



In high-level languages, signed integers are typically represented using the data types `short`, `int`, and `long` (16, 32, and 64 bits)

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

<u>Therefore</u> …

Efficient algorithms for handling arithmetic operations on signed integers are essential for building efficient computers.

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: $n = 4$

In general, n bits allow representing 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

Representation:

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

Remaining bits: Represent a non-negative integer

Issues

- – 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

Representation (using 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 "–" followed by (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

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

$$\begin{array}{r} 3 \\ + \; -5 \end{array} \quad = \quad \begin{array}{r} 3 \\ + \; 11 \\ \hline 14 \end{array} \; \% \; 16 = 14 \text{ codes } -2$$

$$\begin{array}{r} -2 \\ + \; -5 \end{array} \quad = \quad \begin{array}{r} 14 \\ + \; 11 \\ \hline 25 \end{array} \; \% \; 16 = 9 \text{ codes } -7$$

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

$$\begin{array}{r} 6 \\ + \; -2 \end{array} \quad = \quad \begin{array}{r} 6 \\ + \; 14 \\ \hline \end{array}$$

20 % 16 = 4 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

$$\begin{array}{r} 6 \\ + \;\; -2 \end{array} \quad = \quad \begin{array}{r} 6 \\ + \;\; 14 \\ \hline \end{array}$$

20 % 16 = 4 codes 4

Practice:

$$\begin{array}{r} 4 \\ + \;\; -7 \end{array} \quad = \quad \begin{array}{r} 4 \\ + \;\; 9 \\ \hline \end{array}$$

13 % 16 = 13 codes –3

$$\begin{array}{r} -2 \\ + \;\; -4 \end{array} \quad = \quad \begin{array}{r} 14 \\ + \;\; 12 \\ \hline \end{array}$$

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):

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

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

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

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

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):

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

More examples:

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

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

Overflow detection

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

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

- Using n bits, the method represents the integers in the range $-2^{n-1}, \ldots, 2^{n-1}-1$
- $code(x) + code(-x) = code(\mathtt{0})$
- $code(x)$ is monotonically increasing with x
- Arithmetic on signed integers is the same as arithmetic on unsigned integers
- Addition / subtraction / negation are $O(n)$, but practically $O(1)$ since n is fixed
- Simple! Elegant! Efficient!

<u>Implications for hardware designers</u>

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

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

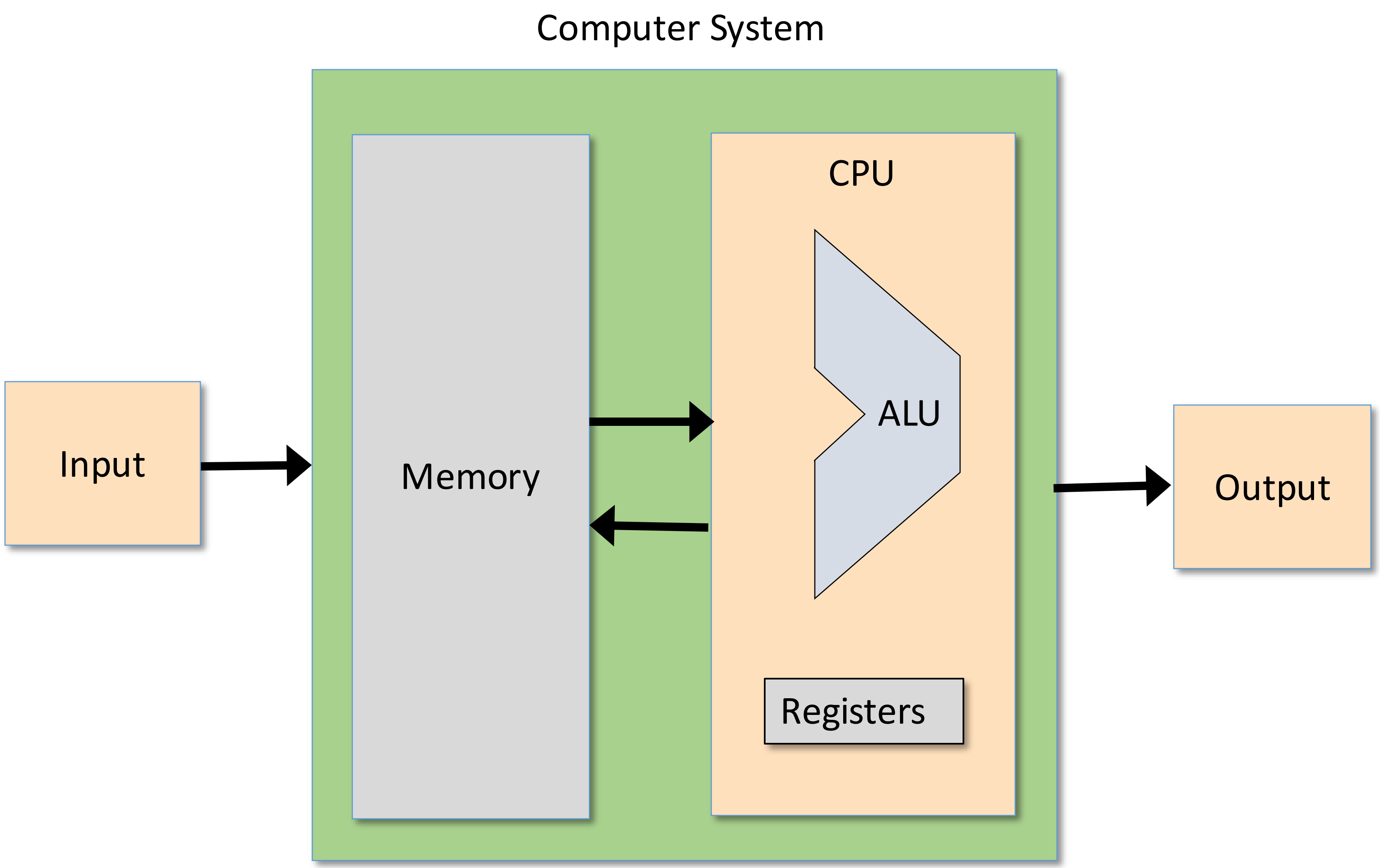
Theory

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

Practice

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

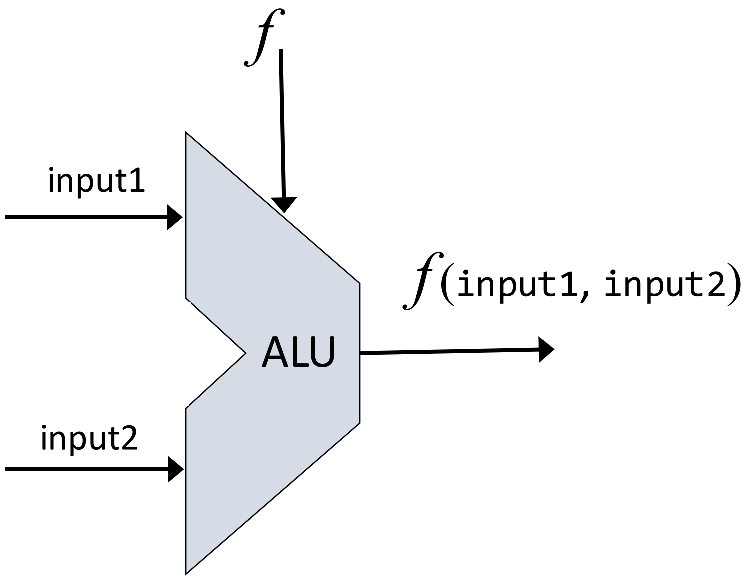
Von Neumann Architecture



The Arithmetic Logical Unit

The ALU computes a given function on two given 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*, …)

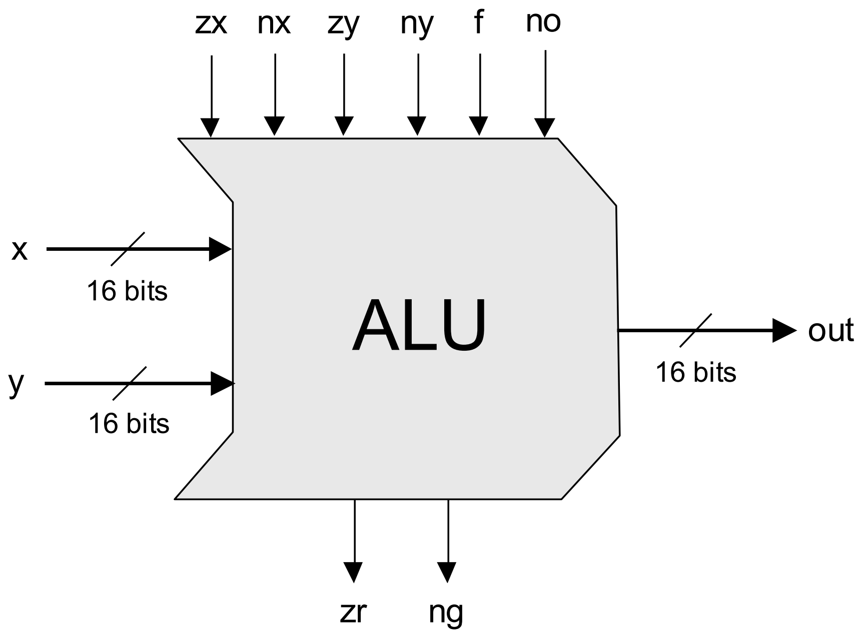


<u>Design issue: Which functions should the ALU compute?</u>

A hardware / software tradeoff:
Functions not implemented by the ALU can be implemented later by software

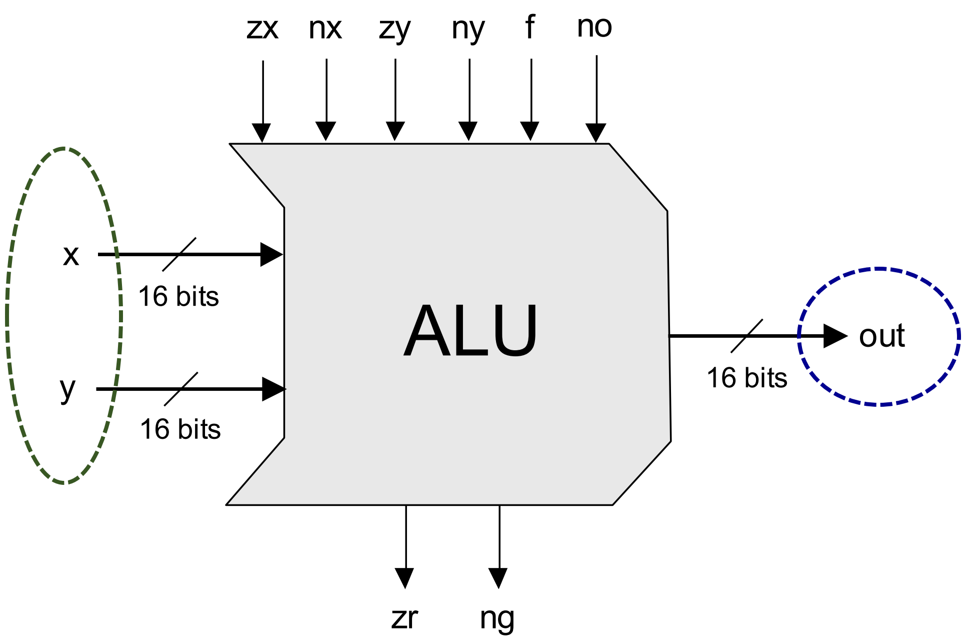
- Hardware implementations: Faster, more expensive
- Software implementations: Slower, less expensive.

The Hack ALU (abstraction)



The Hack ALU (abstraction)

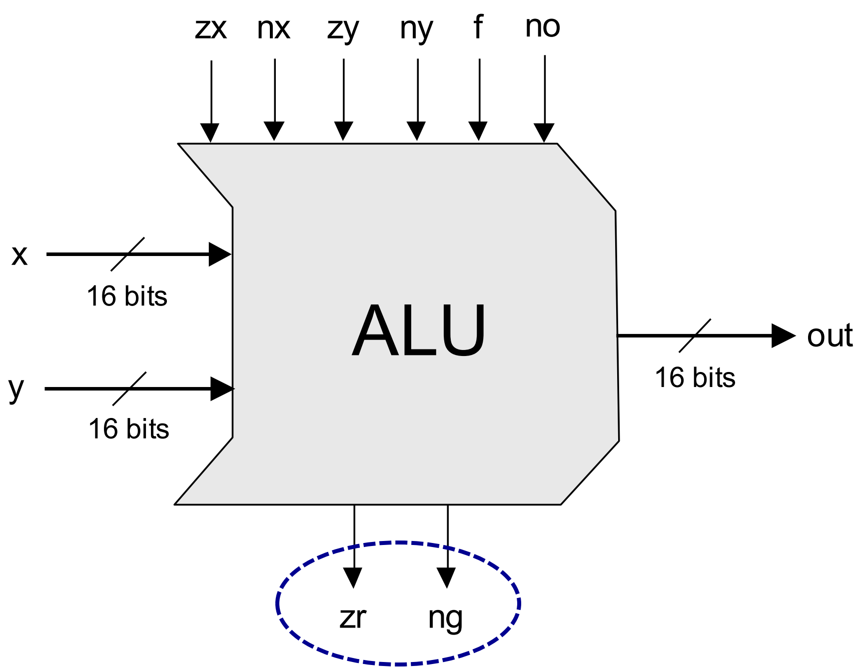
- Operates on two 16-bit, two's complement values
- Computes one of 18 functions, listed on the right



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 (abstraction)

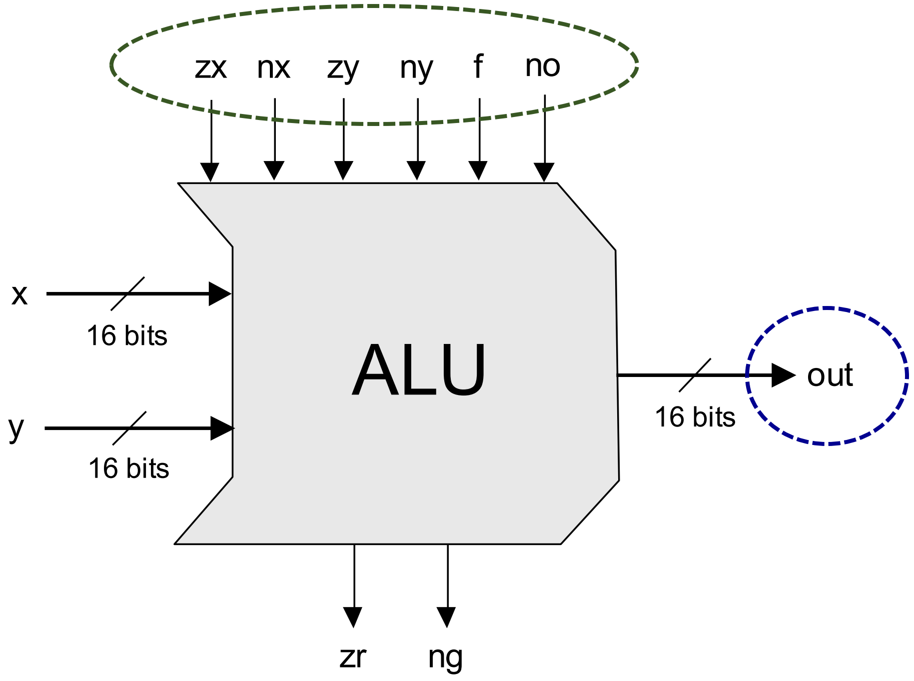
- Operates on two 16-bit, two’s complement values
- Computes one of 18 functions, listed on the right
- Also outputs two 1-bit values (later)



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 (abstraction)

- Operates on two 16-bit, two's complement values
- Computes one of 18 functions, listed on the right
- Also outputs two 1-bit values (later)
- Which function to compute is set by six 1-bit inputs

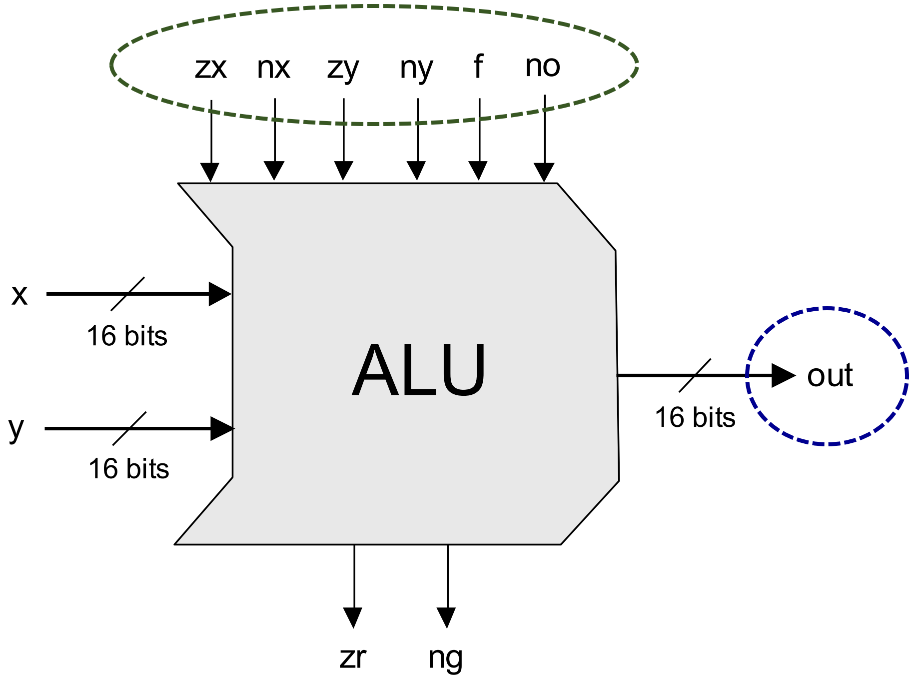


out
0
1
-1
x
y
!x
!y
-x
-y
x+1
y+1
x-1
y-1
x+y
x-y
y-x
x&y
x|y

The Hack ALU (abstraction)

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

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



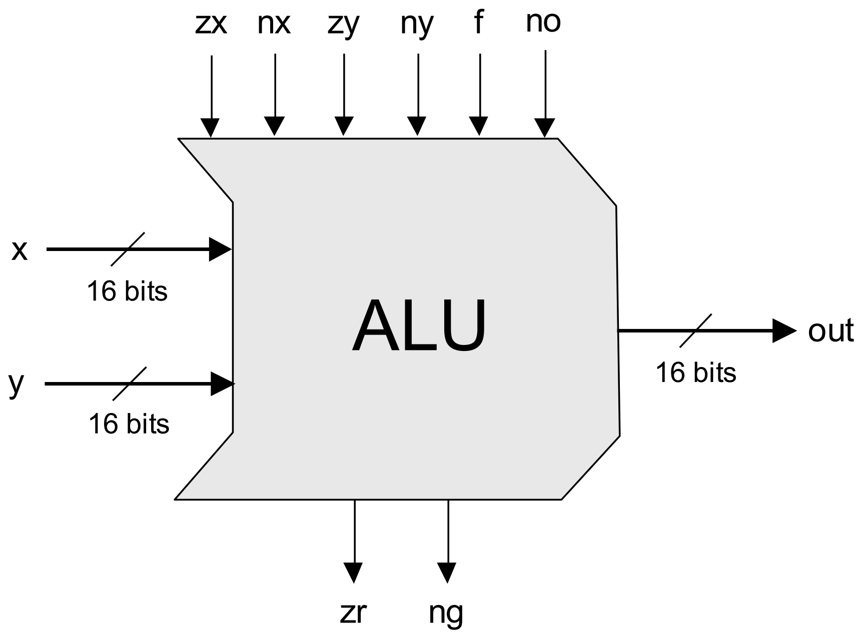
control bits

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

To cause the ALU to compute a function:

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



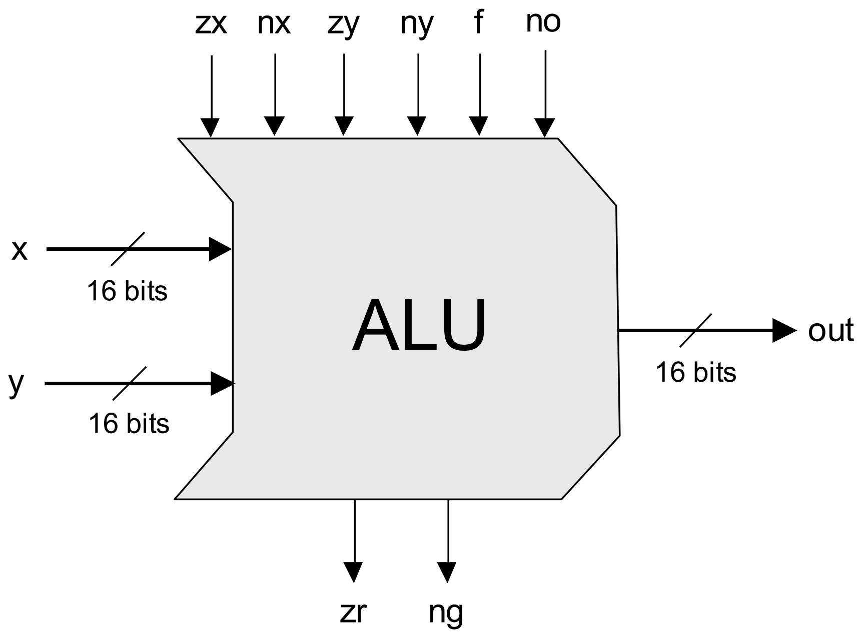
control bits

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:

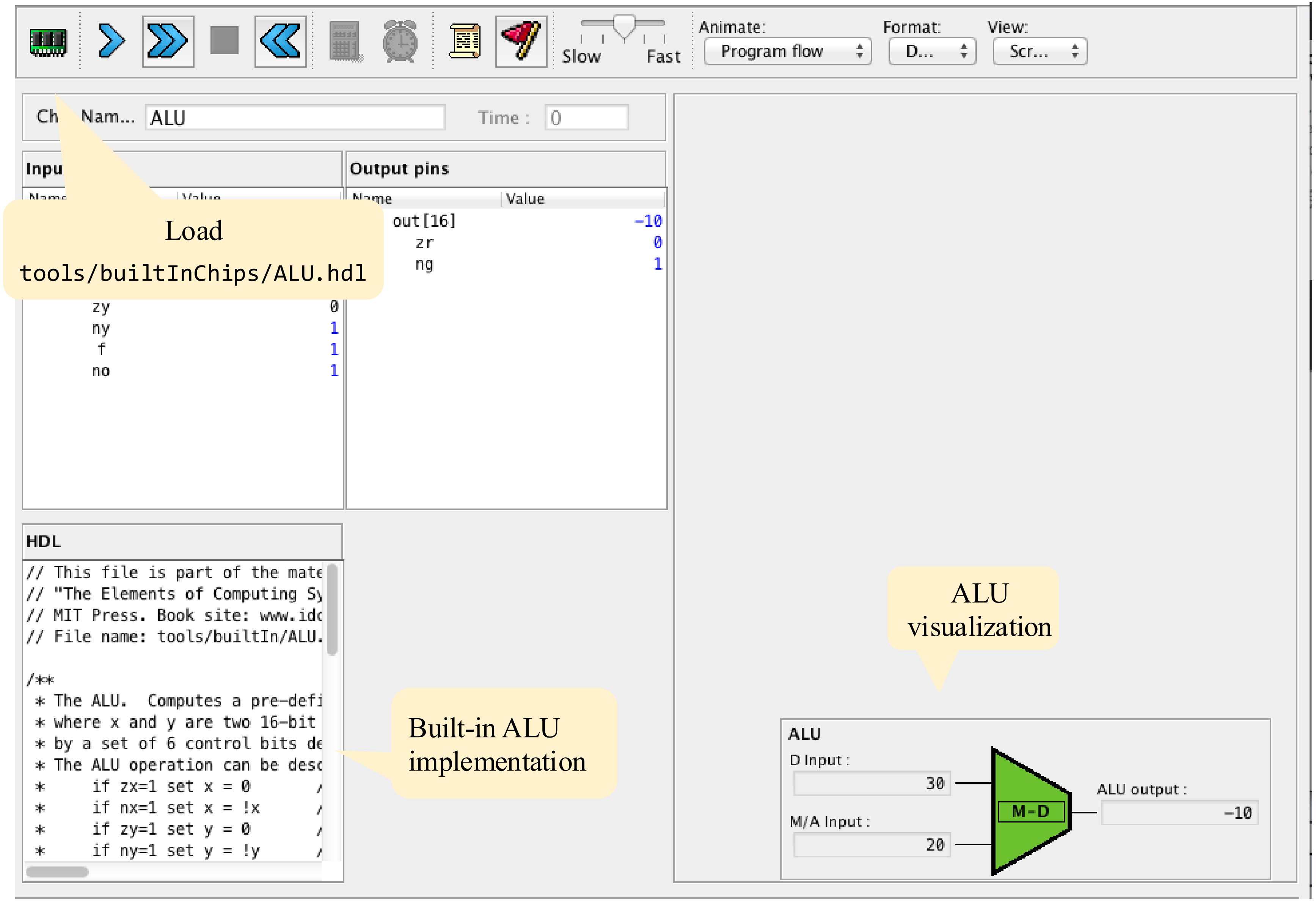
Set the control bits to one of the binary combinations listed in the table.



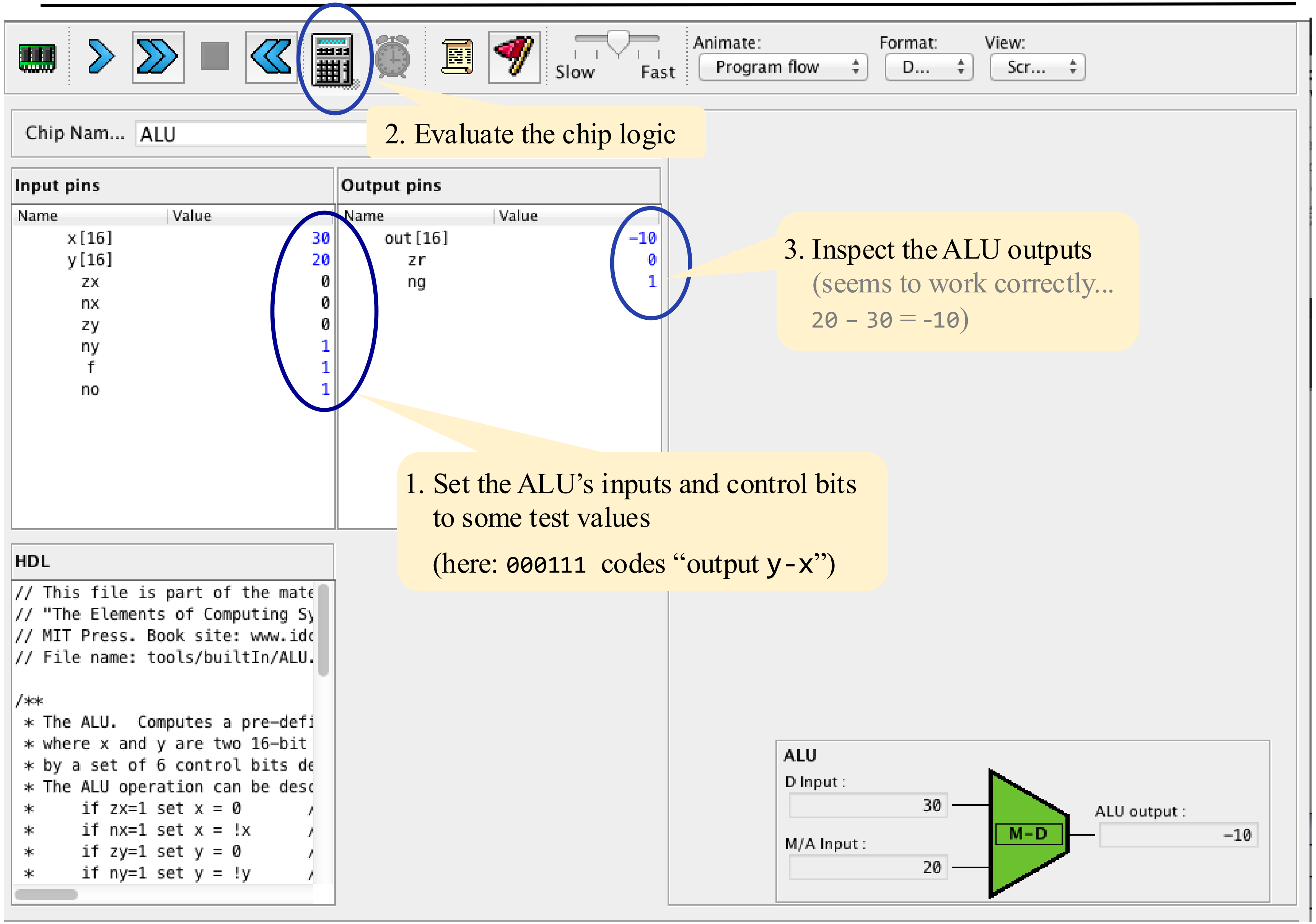
control bits

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



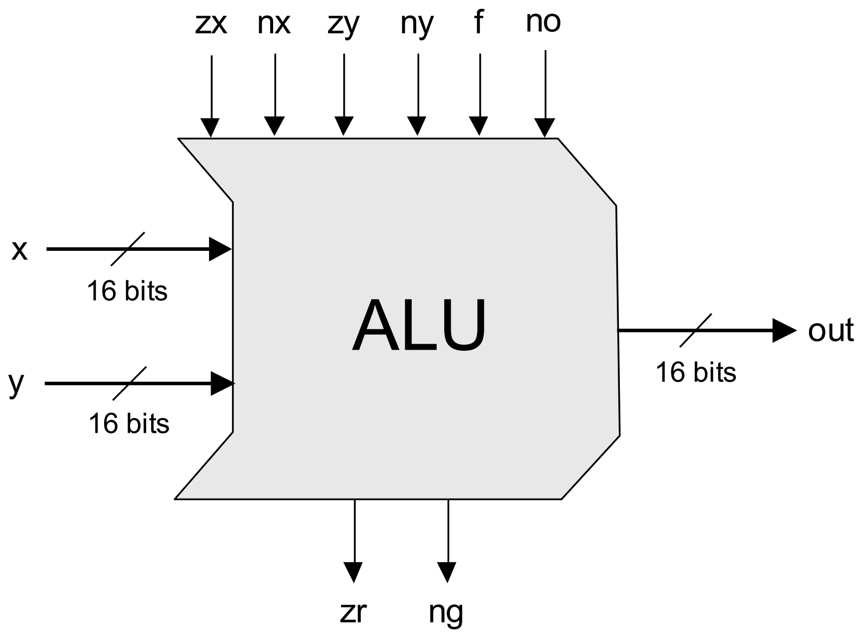
The Hack ALU in action: Compute `y-x`



The Hack ALU in action

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

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



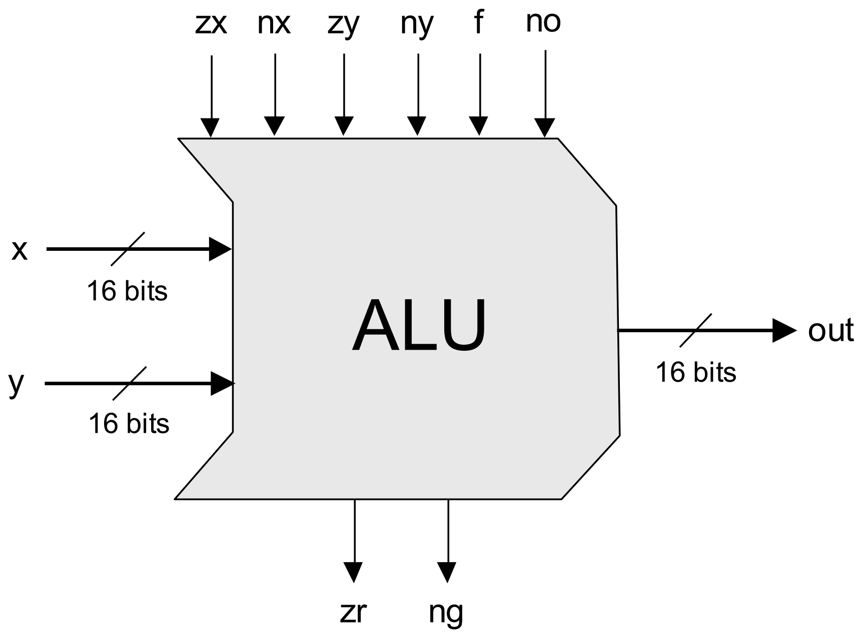
control bits

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`

To cause the ALU to compute a function:

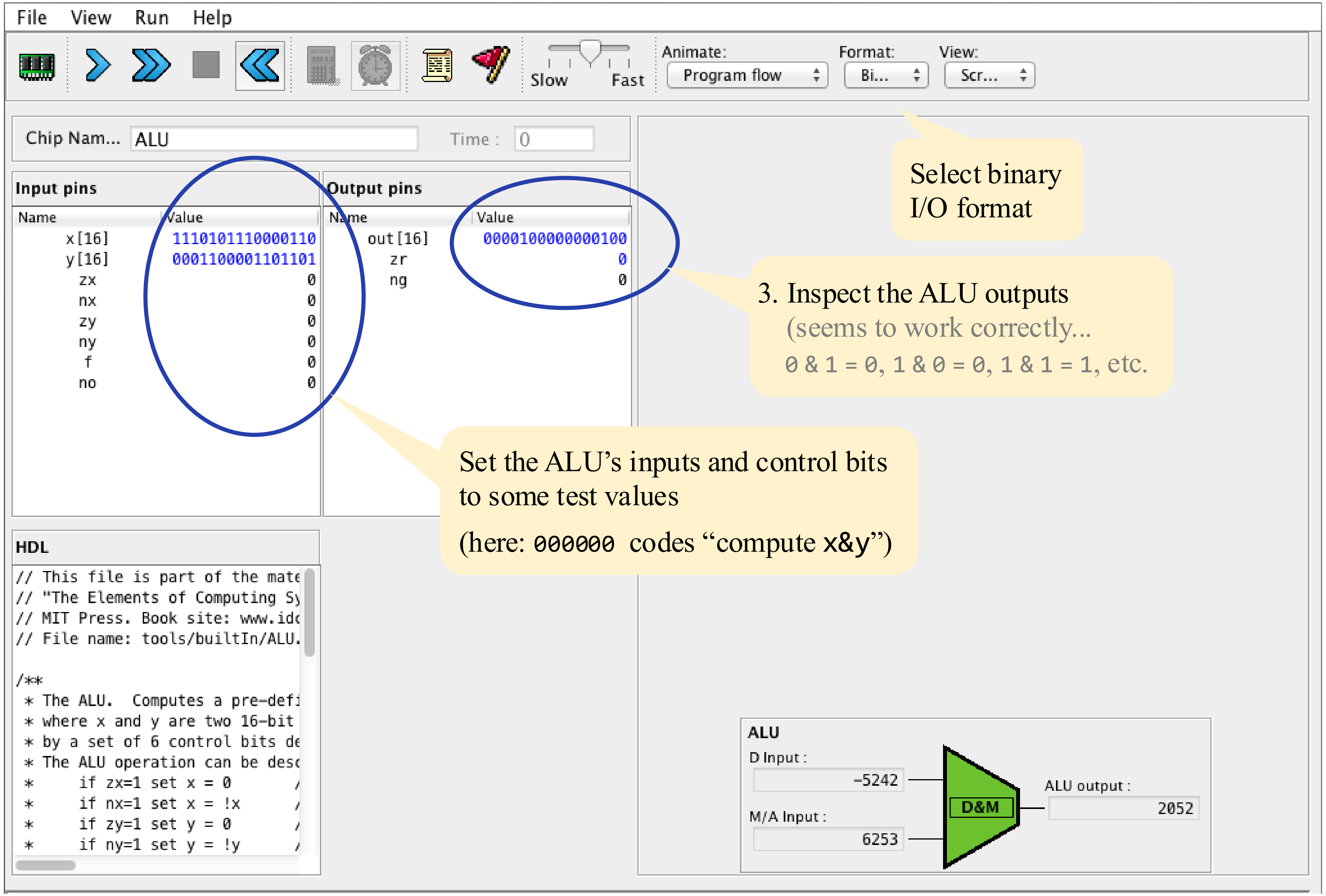
Set the control bits to one of the binary combinations listed in the table.



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 `x & y`



Select binary I/O format

3. Inspect the ALU outputs
(seems to work correctly...
`0 & 1 = 0`, `1 & 0 = 0`, `1 & 1 = 1`, etc.

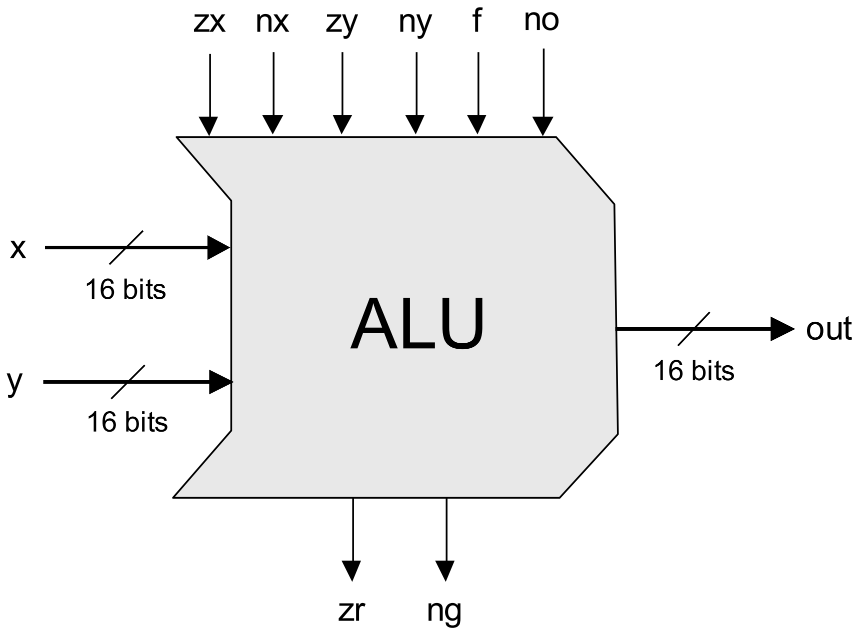
Set the ALU's inputs and control bits to some test values

(here: `000000` codes "compute `x&y`")

The Hack ALU operation: How it Works

So far we discussed the ALU *abstraction*;

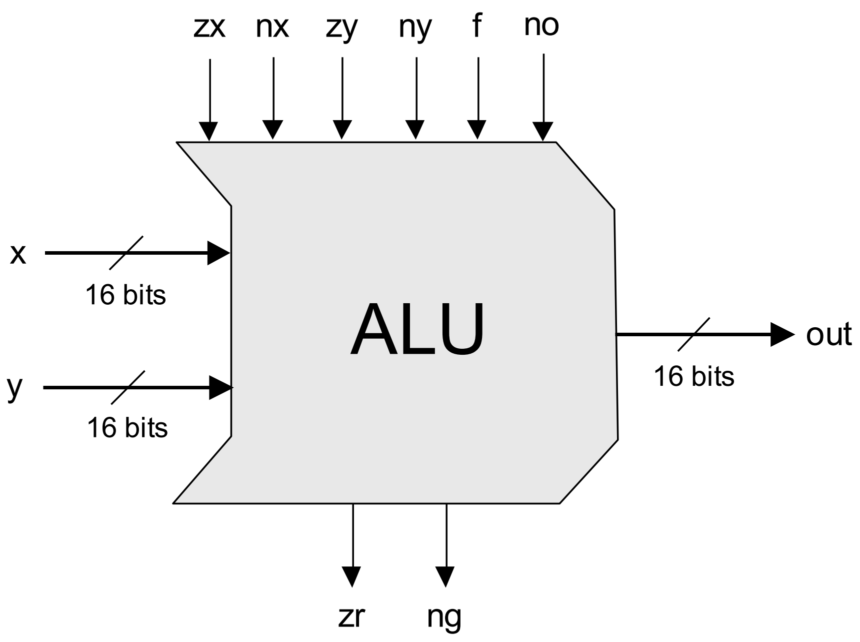
We'll now discuss the ALU *implementation*.



The Hack ALU operation

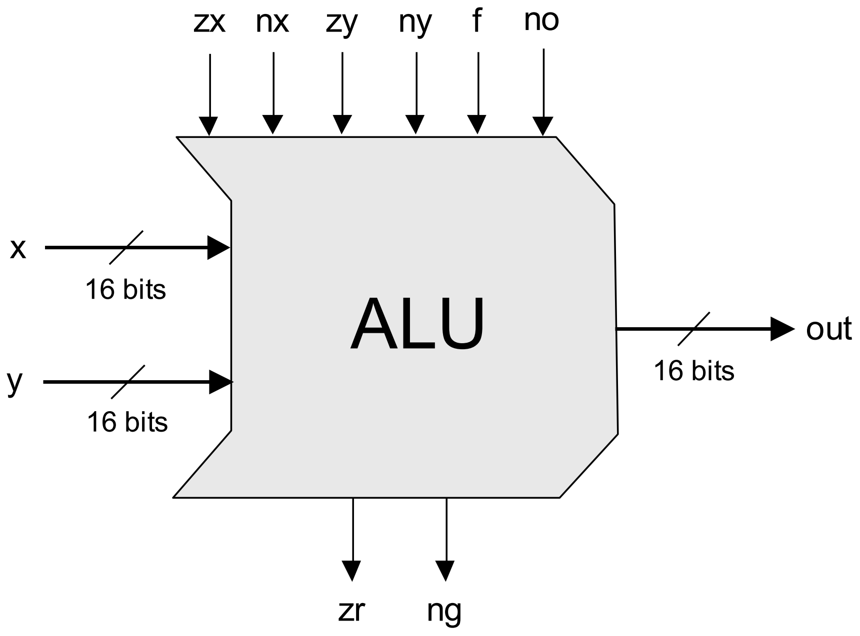
pre-setting
the x input

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



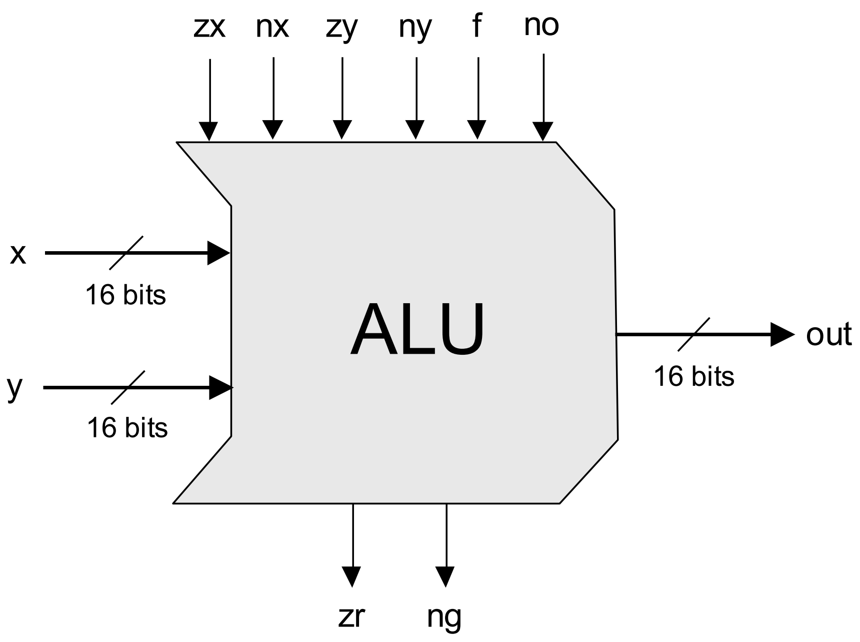
The Hack ALU operation

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



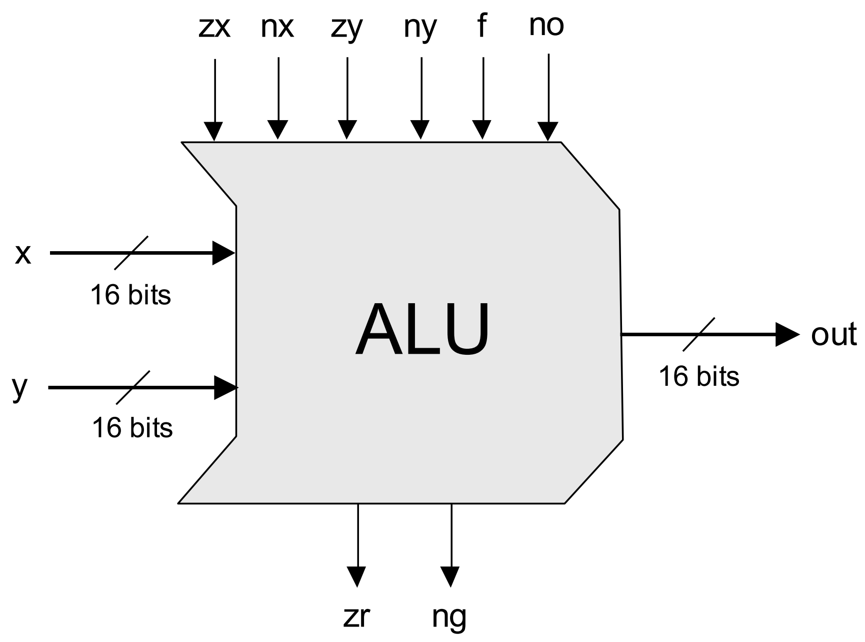
The Hack ALU operation

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



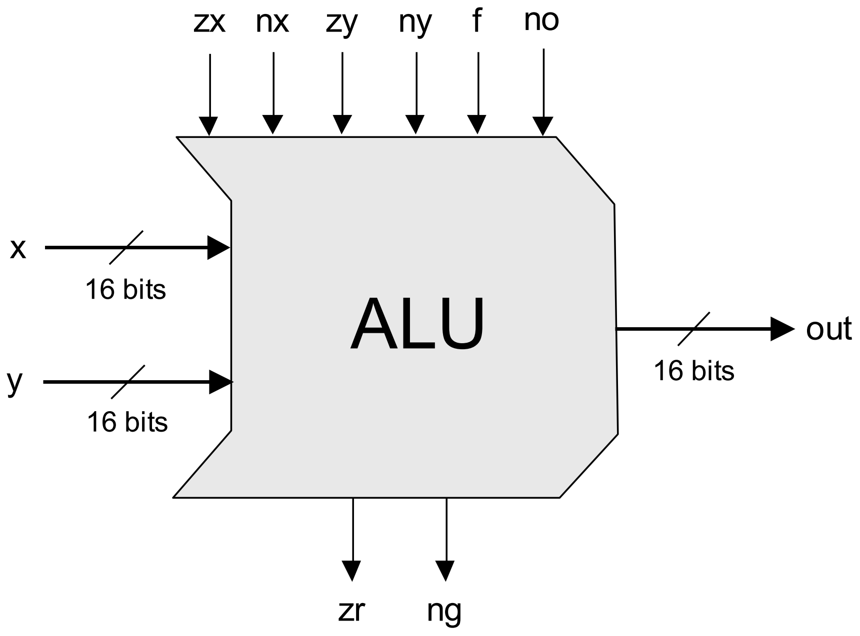
The Hack ALU operation

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



The Hack ALU operation

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



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

How does the magic work?

The Hack ALU operation (assuming 4-bit x and y values)

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

Some useful algebraic insights:

```
-x = !(x + 1111)
```

Example:

```
   0100  = 4
+
   1111
  -----
! 10011
  -----
   1100  = -4
```

Notation

- `+` algebraic plus
- `–` algebraic minus
- `!` bitwise Not
- `&` bitwise And
- `1` overflow bit
- `1111` $2^n – 1$

The Hack ALU operation

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

Some useful algebraic insights:

```
x + 1 = !(!x + 1111)
```

Example:

```
!   1001  = -7
   ------
    0110
+
    1111
   ------
!  10101
   ------
    1010  = -6
```

Notation

- `+` algebraic plus
- `-` algebraic minus
- `!` bitwise Not
- `&` bitwise And
- `1` overflow bit
- `1111` $2^n - 1$

The Hack ALU operation

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

Some useful algebraic insights:

```
x - y = !(!x + y)
```

Example:

```
   0010  =  2
-
   1011  = -5
   ----
   ????

!  0010
   ----
   1101
+
   1011
   ----
! 11000
   ----
   0111  =  7
```

Notation

+ algebraic plus

– algebraic minus

! bitwise Not

& bitwise And

1 overflow bit

1111 $2^n - 1$

The Hack ALU operation

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

Some useful algebraic insights:

```
x = x & 1111
```

Example:

```
   0101  =  5
&  1111
   ----
   0101  =  5
```

Notation

- `+` algebraic plus
- `-` algebraic minus
- `!` bitwise Not
- `&` bitwise And
- `1` overflow bit
- `1111` $2^n - 1$

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

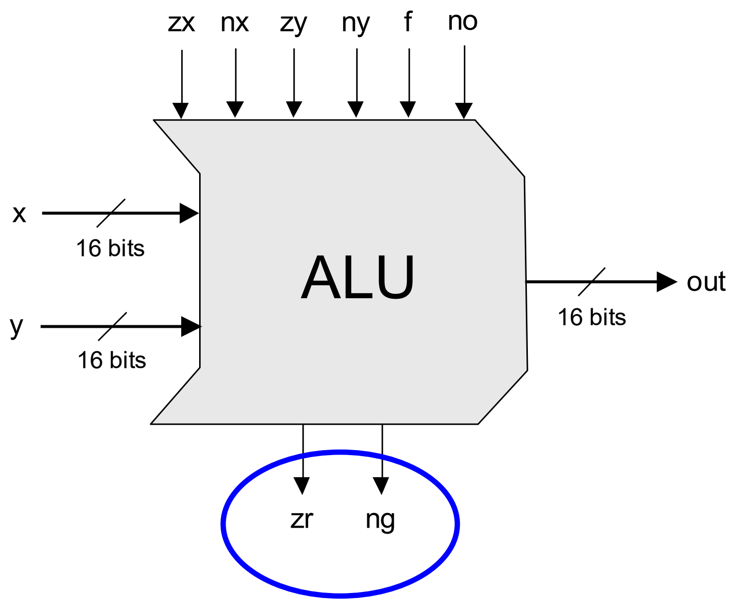
Our ALU logic is based on:

```
 x = x & 1111
-x = !(x + 1111)
 x + 1 = !(!x + 1111)
 x - y = !(!x + y)
```

And a few more similar insights

The Hack ALU operation

<u>One more detail</u>




```
zr  =  ((out == 0), 1, 0)

ng  =  ((out < 0), 1, 0)
```

The `zr` and `ng` output bits will come into play when we'll build the computer's CPU, 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

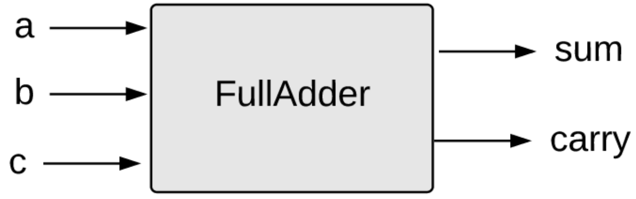
HalfAdder.hdl

```
/** Computes the sum of two bits. */
CHIP HalfAdder {
    IN a, b;
    OUT sum, carry;

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

<u>Implementation tip</u>

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

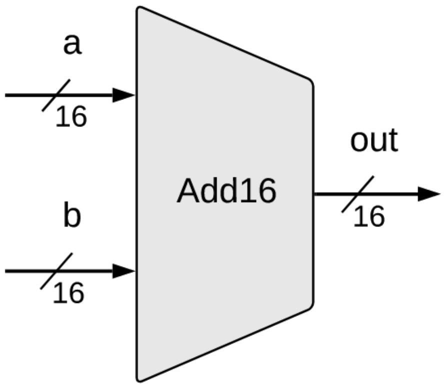
`FullAdder.hdl`

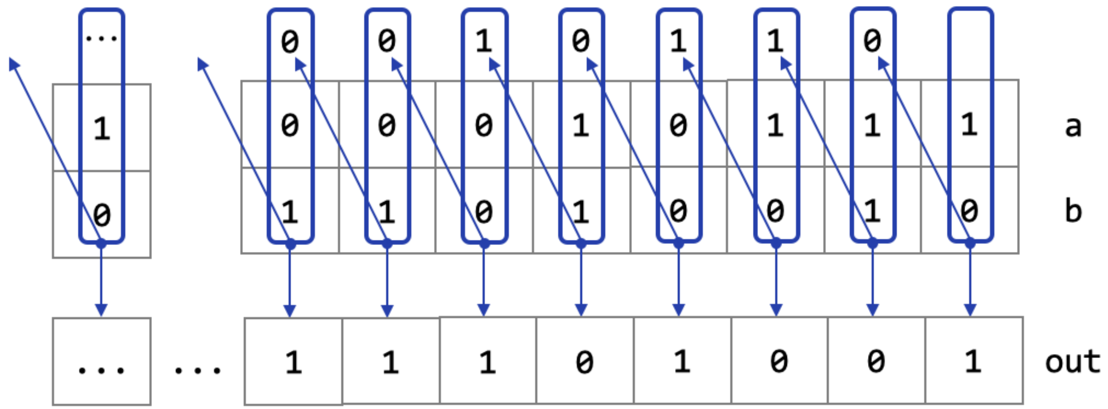
```
/** Computes the sum of three bits. */
CHIP FullAdder {
    IN a, b, c;
    OUT sum, carry;

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

Implementation tip

Can be built from two half-adders.

16-bit adder



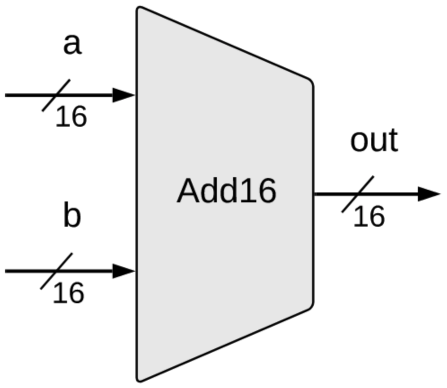


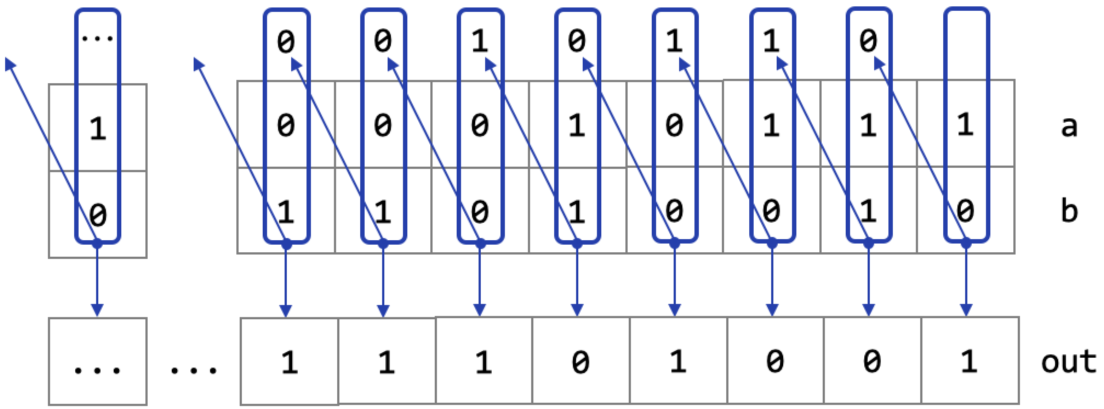
Add16.hdl

```
/* 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 computed in parallel
- The carry propagations are computed sequentially
- How does it end up working?
  Wait for the next lecture / chapter.

16-bit adder





Add16.hdl

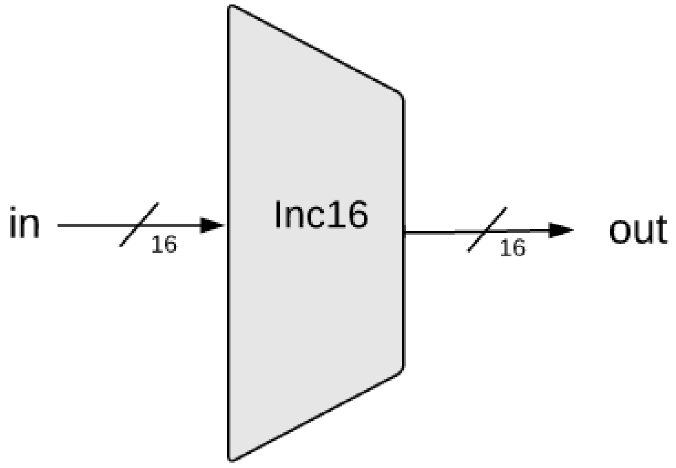
```
/* Adds two 16-bit, two's-complement values.
   The most-significant carry bit is ignored. */
CHIP Add16 {
    IN a[16], b[16];
    OUT out[16];
    PARTS:
    // Put you code here:
}
```

<u>Implementation tip</u>

To set a pin `x` to `0` (or `1`) in HDL,

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

16-bit incrementor



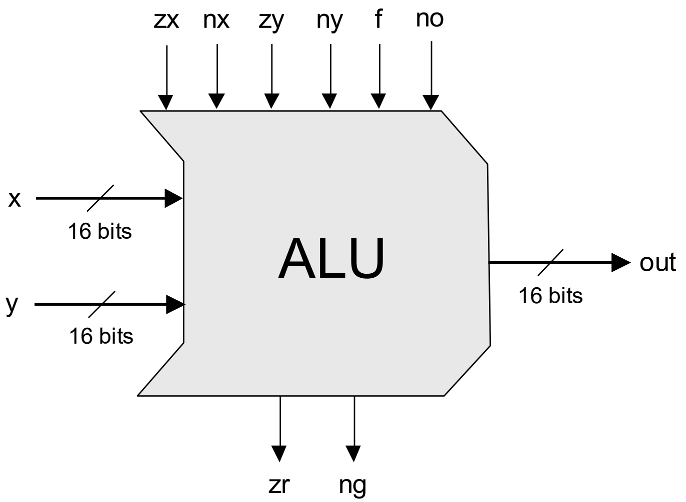
Inc16.hdl

```
/** Outputs in + 1. */
CHIP Inc16 {
    IN in[16];
    OUT out[16];

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

Implementation tips

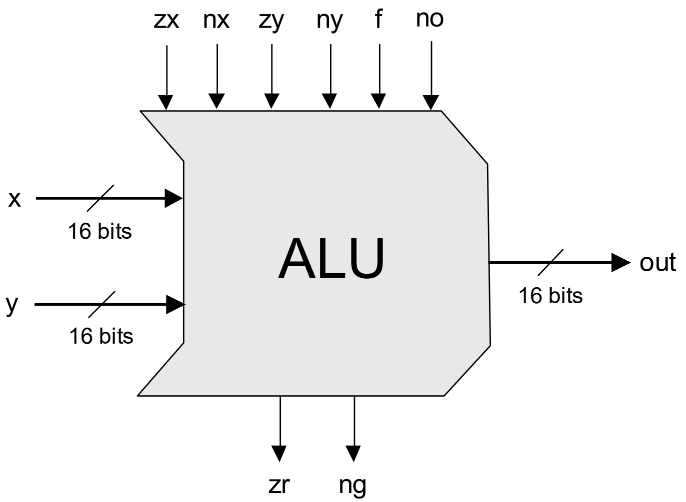
- Can be built using an `Add16` chip-part
- To set a bus-subset `x[i..j]` to `00…0` (or to `11…1`) in HDL, use: `x[i..j] = false` (or `x[i..j] = true`)

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

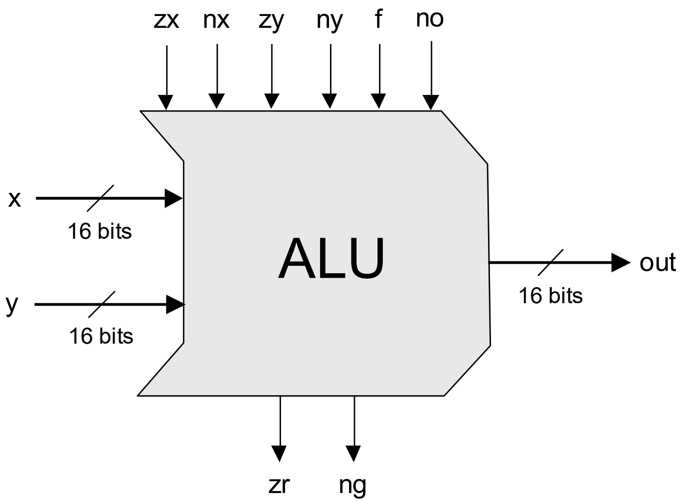


pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	**nx**	**zy**	**ny**	**f**	**no**	**out**
`if zx then x=0`	`if nx then x=!x`	`if zy then y=0`	`if ny then y=!y`	`if f then out=x+y else out=x&y`	`if no then out=!out`	`out(x,y)=`

ALU.hdl

```
/** The ALU */
```

ALU

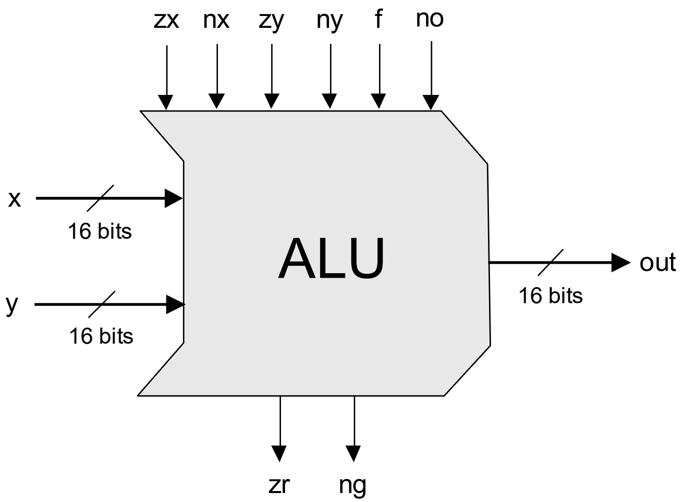


pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	**nx**	**zy**	**ny**	**f**	**no**	**out**
`if zx then x=0`	`if nx then x=!x`	`if zy then y=0`	`if ny then y=!y`	`if f then out=x+y else out=x&y`	`if no then out=!out`	`out(x,y)=`

ALU.hdl

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

ALU



Implementation tips

We need logic for:

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

ALU.hdl

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

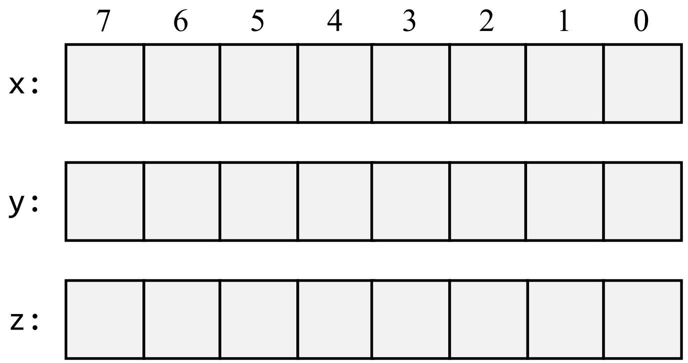
Implementation strategy

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

Useful bus tips

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

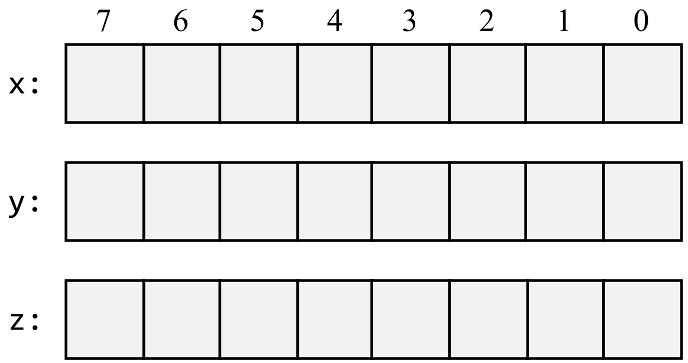
```
...
// Suppose that x, y, z are 8-bit bus-pins:
```



Useful bus tips

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

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



Useful 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`

Useful bus tips

Sub-bussing:

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

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

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

Useful bus tips

Sub-bussing:

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

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

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

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

Useful 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*

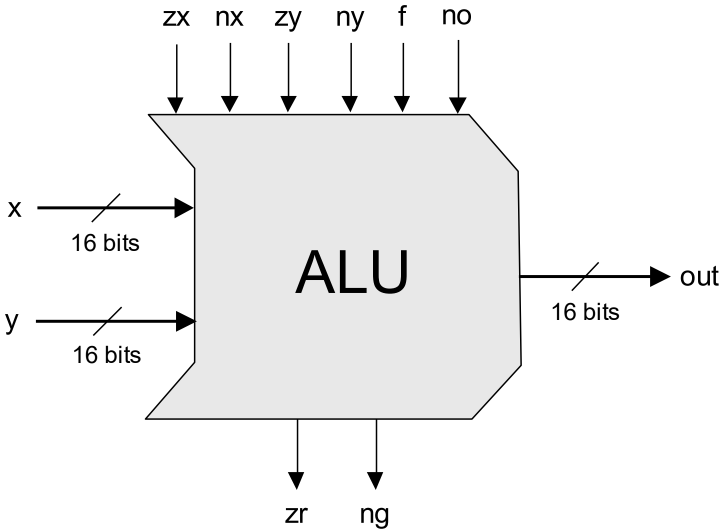
```
/* 16-bit adder */
CHIP Add16 {
    IN a[16], b[16];
    OUT out[16];
    PARTS:
    ...
}
```

```
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



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

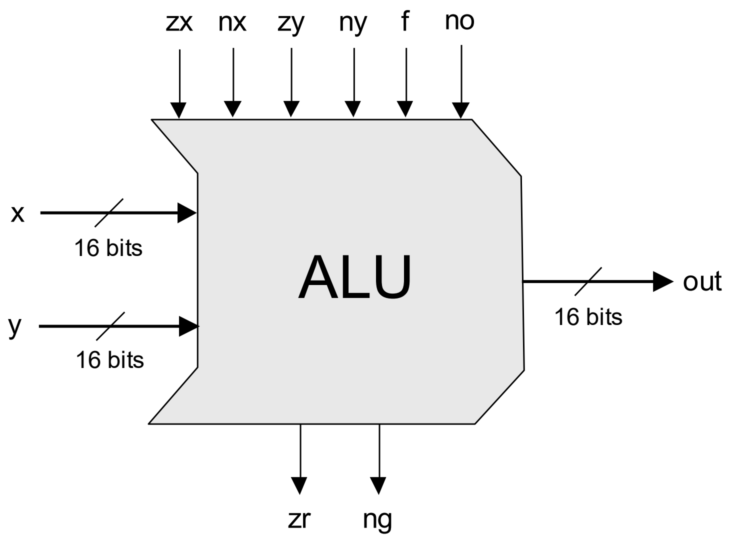
To implement the ALU logic:

We need to know how to...

- Implement "if bit == `0`/`1`" conditions
- Set a 16-bit value to `0000000000000000`
- Set a 16-bit value to `1111111111111111`
- Negate a 16-bit value (bitwise)
- Compute `Add` and `And` on two 16-bit values

All simple operations

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

Given: The chips built in Project 1

Goal: Build the chips:

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

Project 2 Guidelines

Best practice advice (same as project 1)

- 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

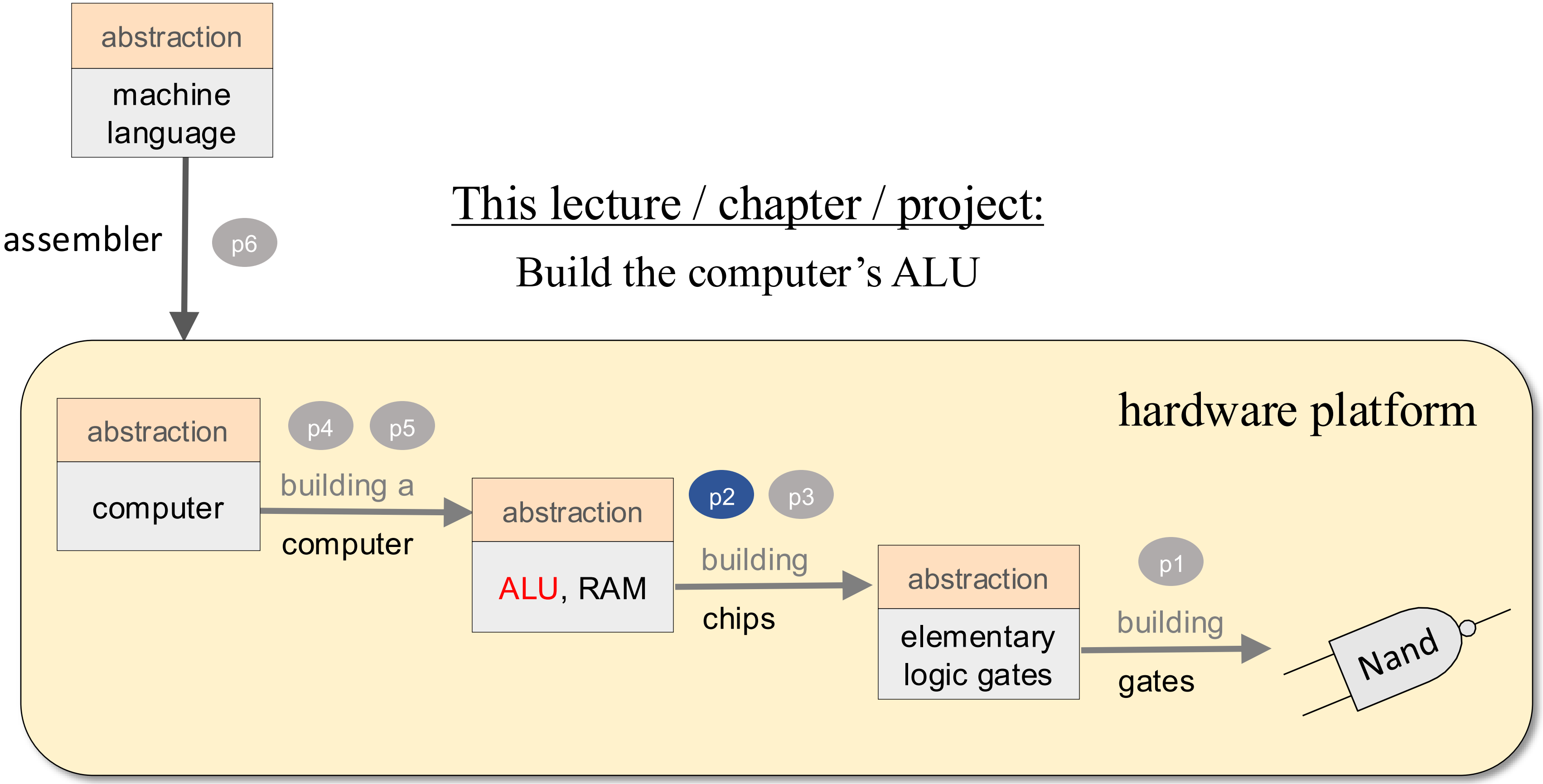
Best practice advice

- 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, we use *built-in* chip-part implementations.

That's It!

Go Do Project 2!

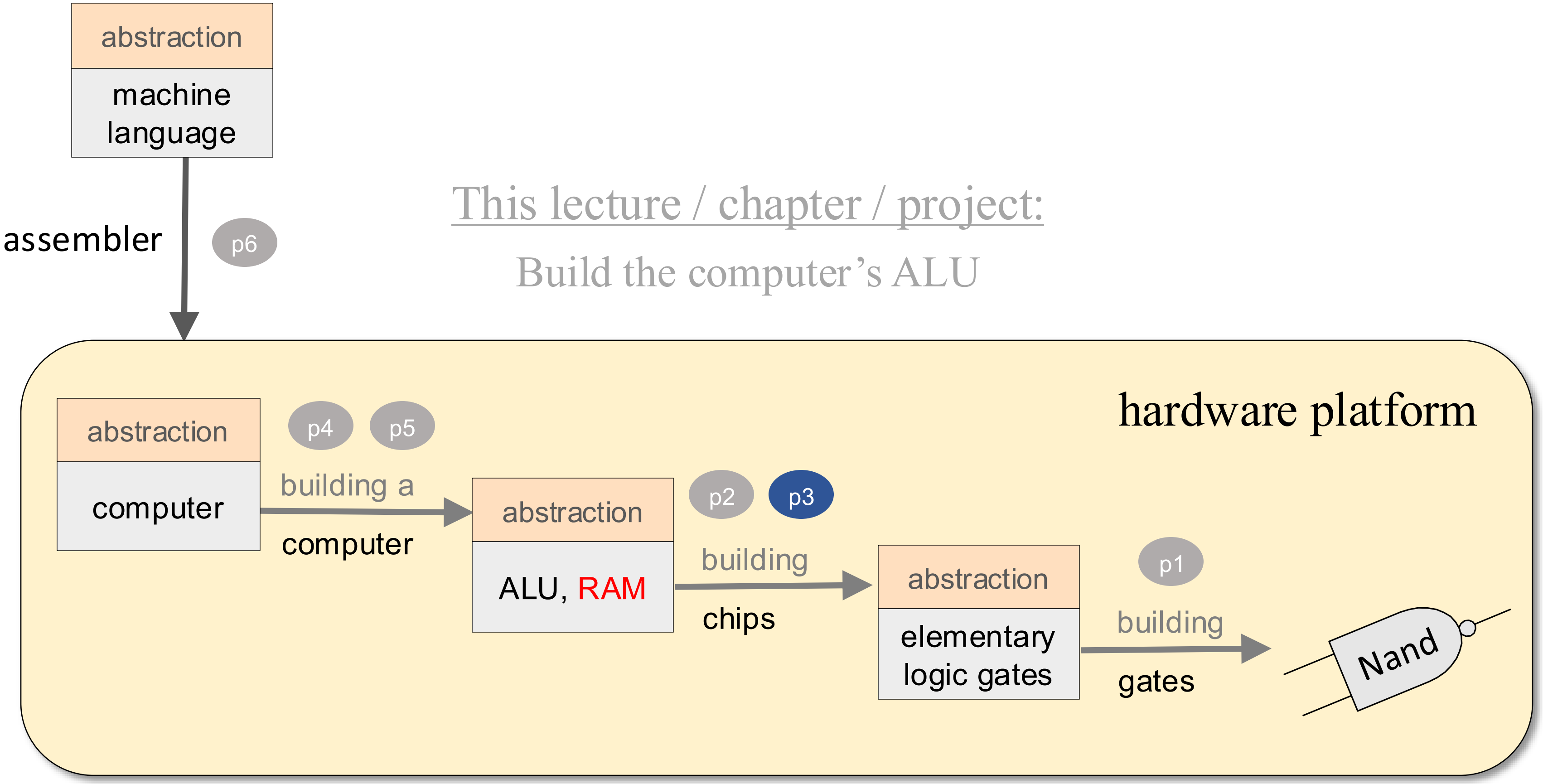
What’s next?



This lecture / chapter / project:

Build the computer’s ALU

What's next?



Next lecture / chapter / project:

Build the computer's RAM