Lecture 2

Boolean Arithmetic

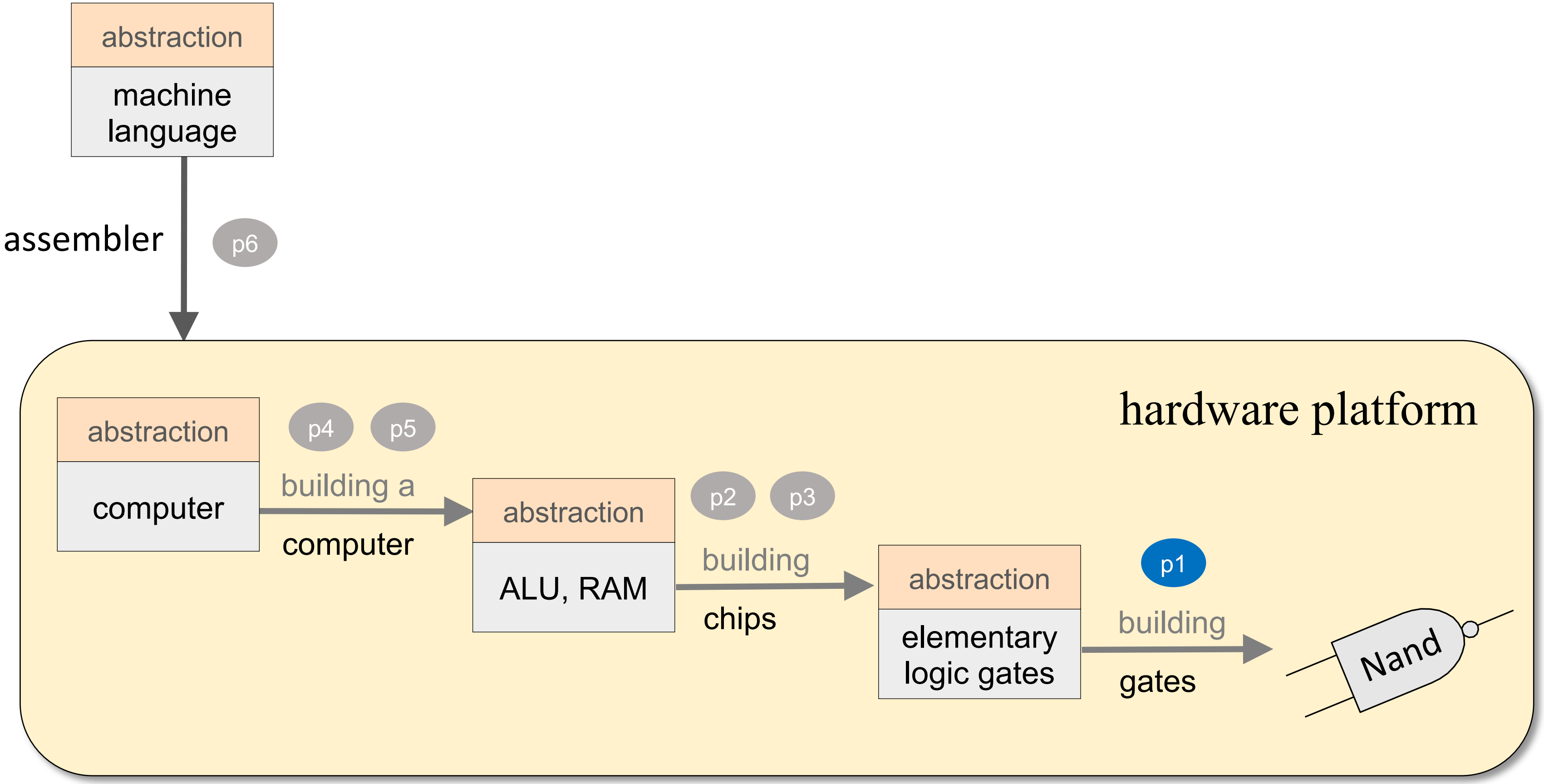
These slides support chapter 2 of the book

The Elements of Computing Systems

By Noam Nisan and Shimon Schocken

MIT Press, 2021

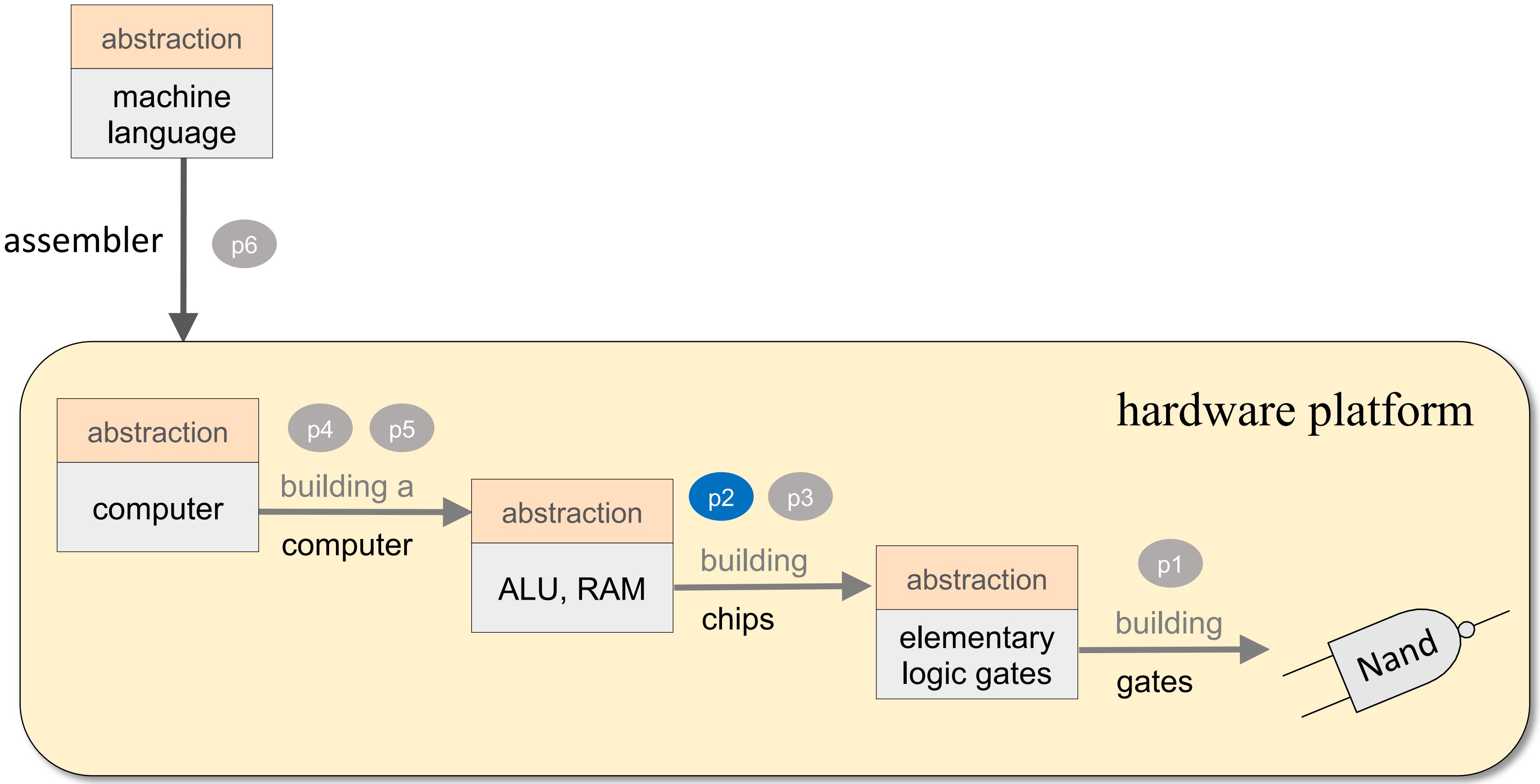
Nand to Tetris Roadmap: Hardware



<u>Project 1</u>

Build 15 elementary logic gates

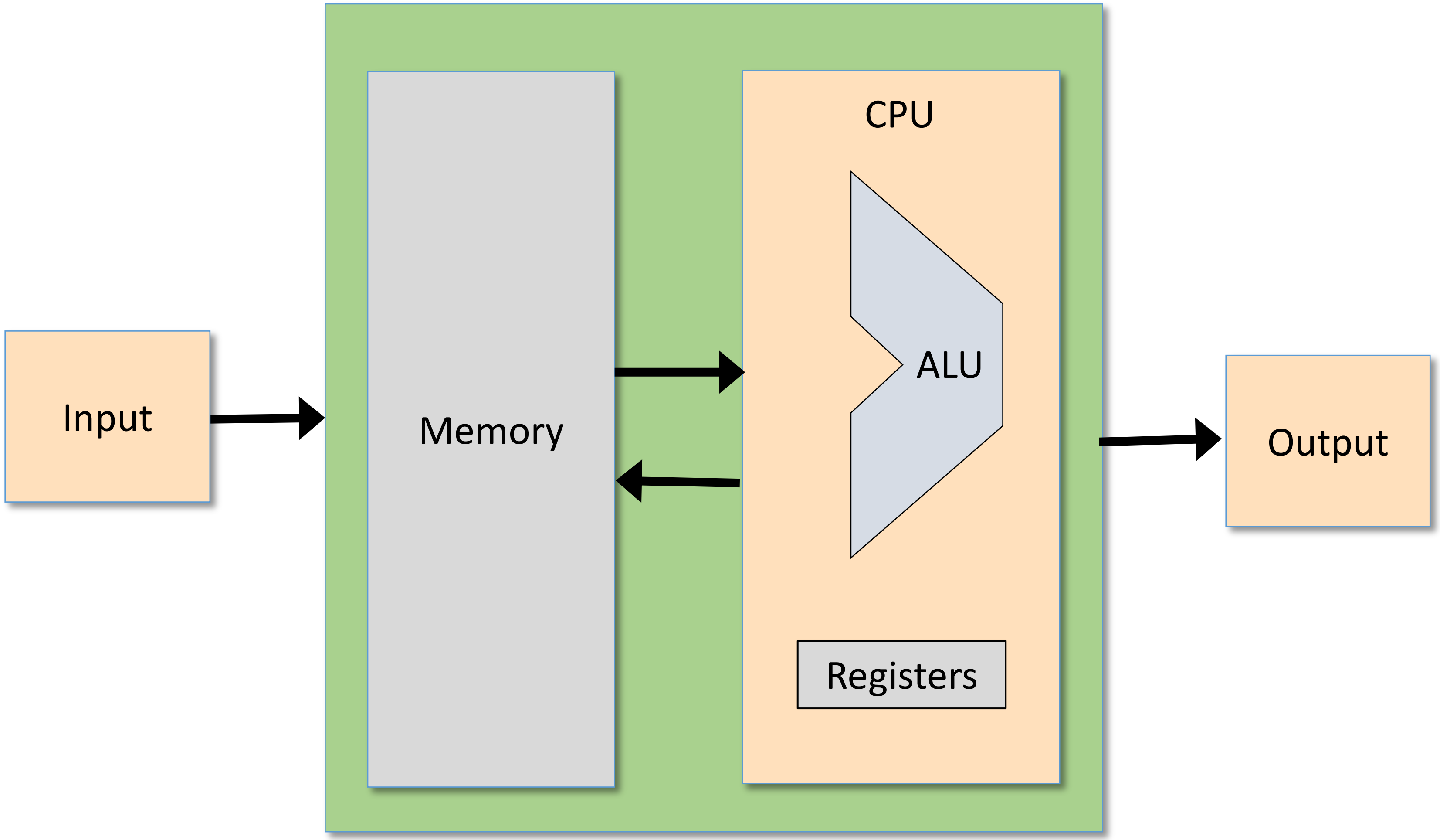
Nand to Tetris Roadmap: Hardware



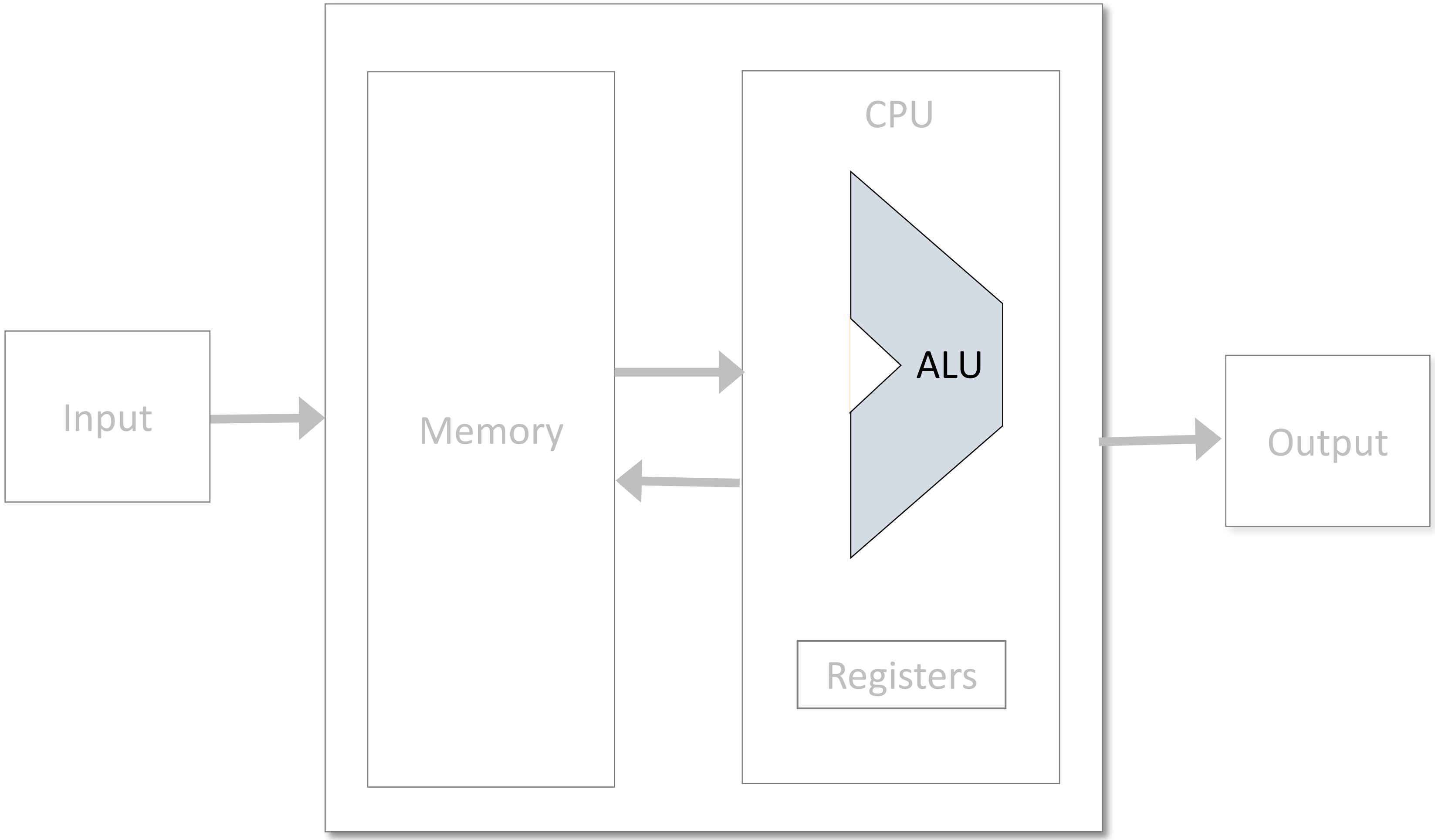
<u>Project 2</u>

Build chips that do arithmetic,
leading up to an ALU

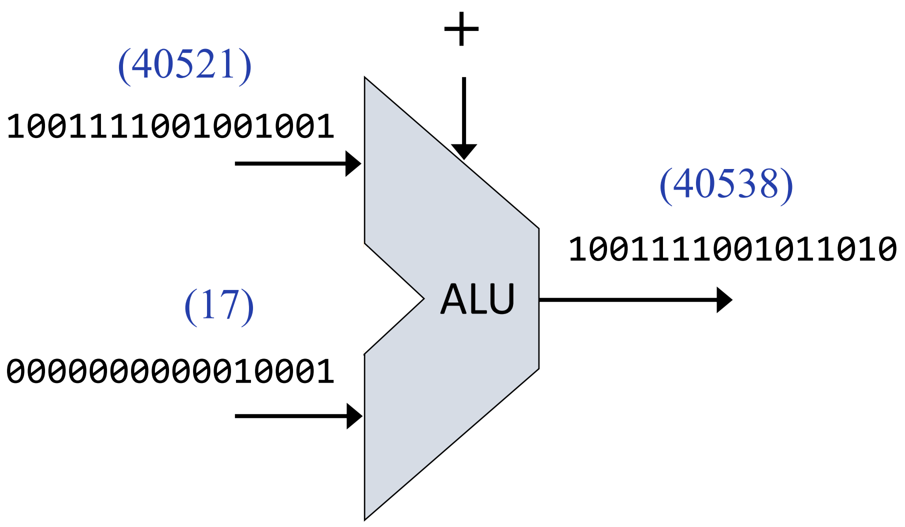
Computer system



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`, ...

Challenges

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

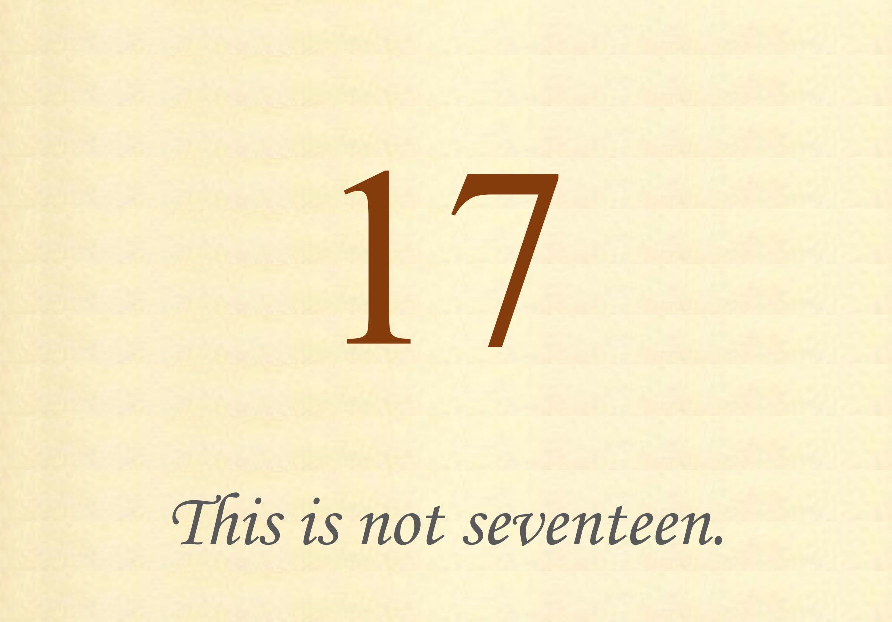
- Arithmetic Logic Unit (ALU)
- Project 2: Chips
- Project 2: Guidelines

Representation

This is not a pipe

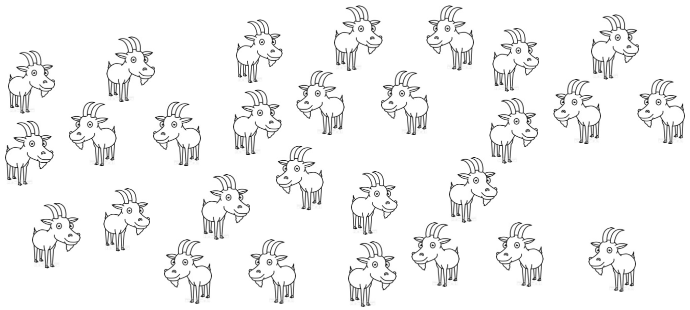
(by René Magritte)

Representation



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

A brief history of numeral systems



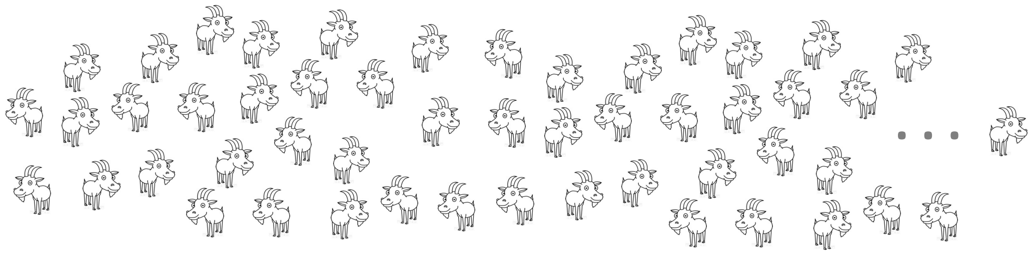
Twenty seven goats

Unary: 

Egyptian: 

Roman: XXVII

A brief history of numeral systems



Six thousands,
five hundreds,
and seven goats

Unary: 

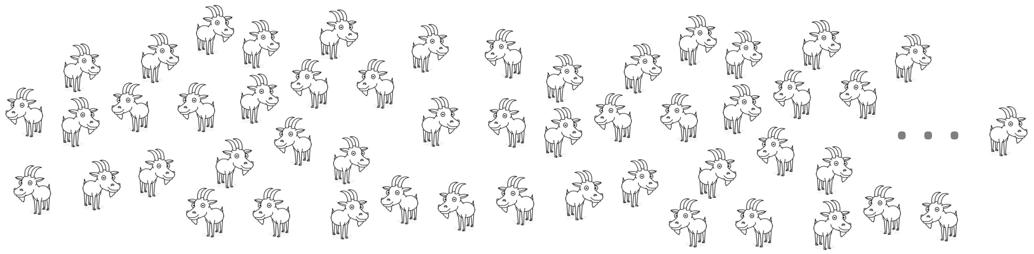
Egyptian: 

Roman: MMMMMMDVII

<u>Old numeral systems</u>:

- 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 is the digit at position i

A most important 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

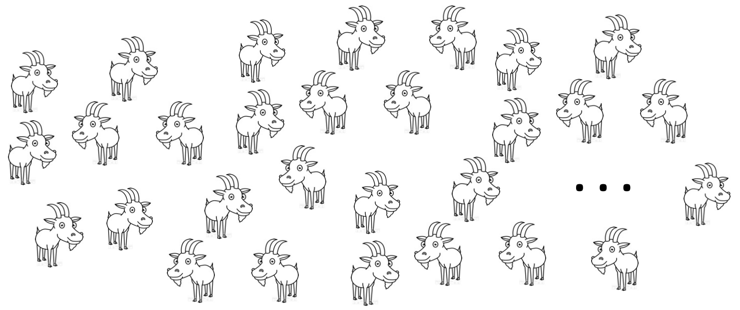
<u>Theory</u>

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

<u>Practice</u>

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

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 enter and view numbers in base 10;

Computers represent and process numbers in base 2;

Therefore, for I/O purposes only, 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:

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

Decimal to binary:

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

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:

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

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

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:

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

Decimal to binary:

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

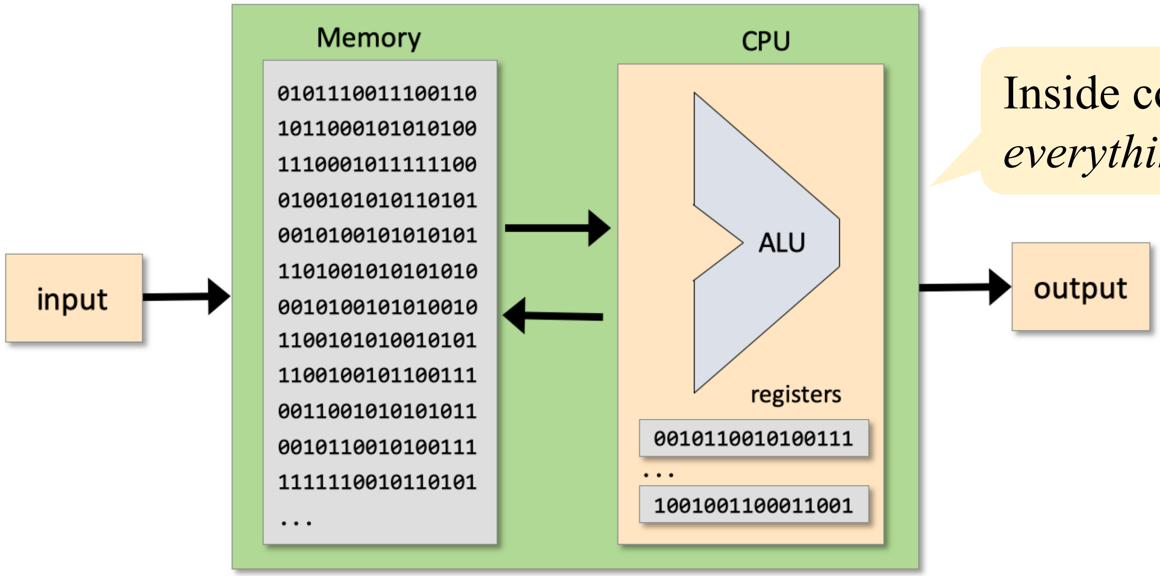
Algorithm: What is the largest power of 2 that “fits into” 53? It’s $2^5 = 32$. We still have to represent 53 – 32, so, what is the largest power of 2 that fits into 21? It’s $2^4 = 16$, and so on.

Practice:

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

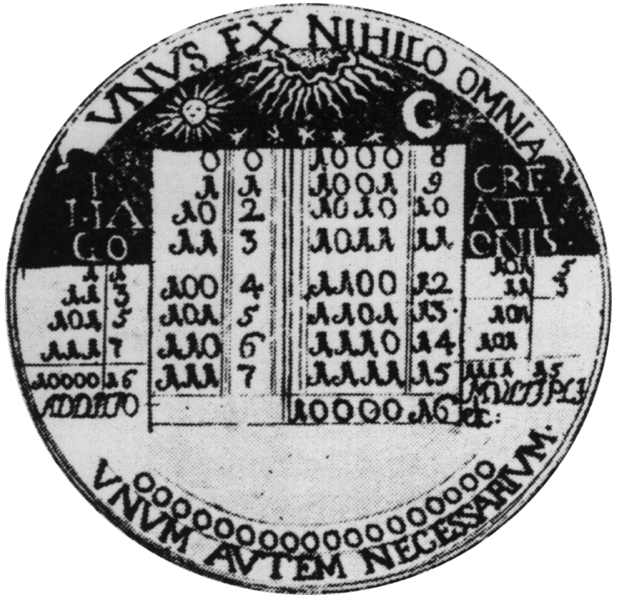
$$binary(523_{10}) = 1000001011_2$$

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 it using addition

Addition is the foundation of all arithmetic.

Addition

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

Binary addition

```
  0 1 1 0
   7 8 7 5
+
     5 6 2
  ---------
   8 4 3 7
```

Decimal addition

Addition

Computers represent integers using a fixed number of bits.
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 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

<u>Handling overflow</u>

- Our approach: Ignore it
- As we'll 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	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

Hardware implementation

We'll build an *Adder* chip that implements this addition algorithm, using the chips built in project 1.

(Later).

Chapter 2: Boolean Arithmetic

Theory

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

Practice

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

Chapter 2: Boolean Arithmetic

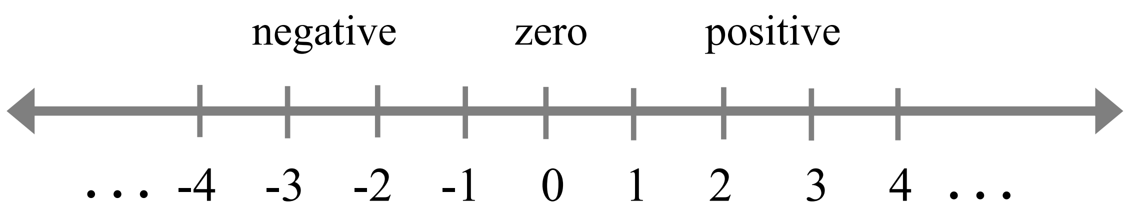
Theory

- ✓ 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



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

Therefore …

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

$$\begin{array}{r} 6 \\ + \ -2 \end{array} \quad = \quad \begin{array}{r} 6 \\ + \ 14 \\ \hline 20 \end{array}$$

20 % 16 = 4 codes 4

Practice:

$$\begin{array}{r} 4 \\ + \ -7 \end{array} \quad = \quad \begin{array}{r} 4 \\ + \ 9 \\ \hline 13 \end{array}$$

13 % 16 = 13 codes –3

$$\begin{array}{r} -2 \\ + \ -4 \end{array} \quad = \quad \begin{array}{r} 14 \\ + \ 12 \\ \hline 26 \end{array}$$

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):

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

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

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

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

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:

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

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

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: 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

Compute $-x$ from x

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

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

$= 1 + flippedBits(x)$

Algorithm: 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

Compute $-x$ from x

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

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

$= 1 + flippedBits(x)$

Algorithm: 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)
```

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

- Using n bits, 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
- Addition / subtraction / conversion are $O(n)$
- 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

<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

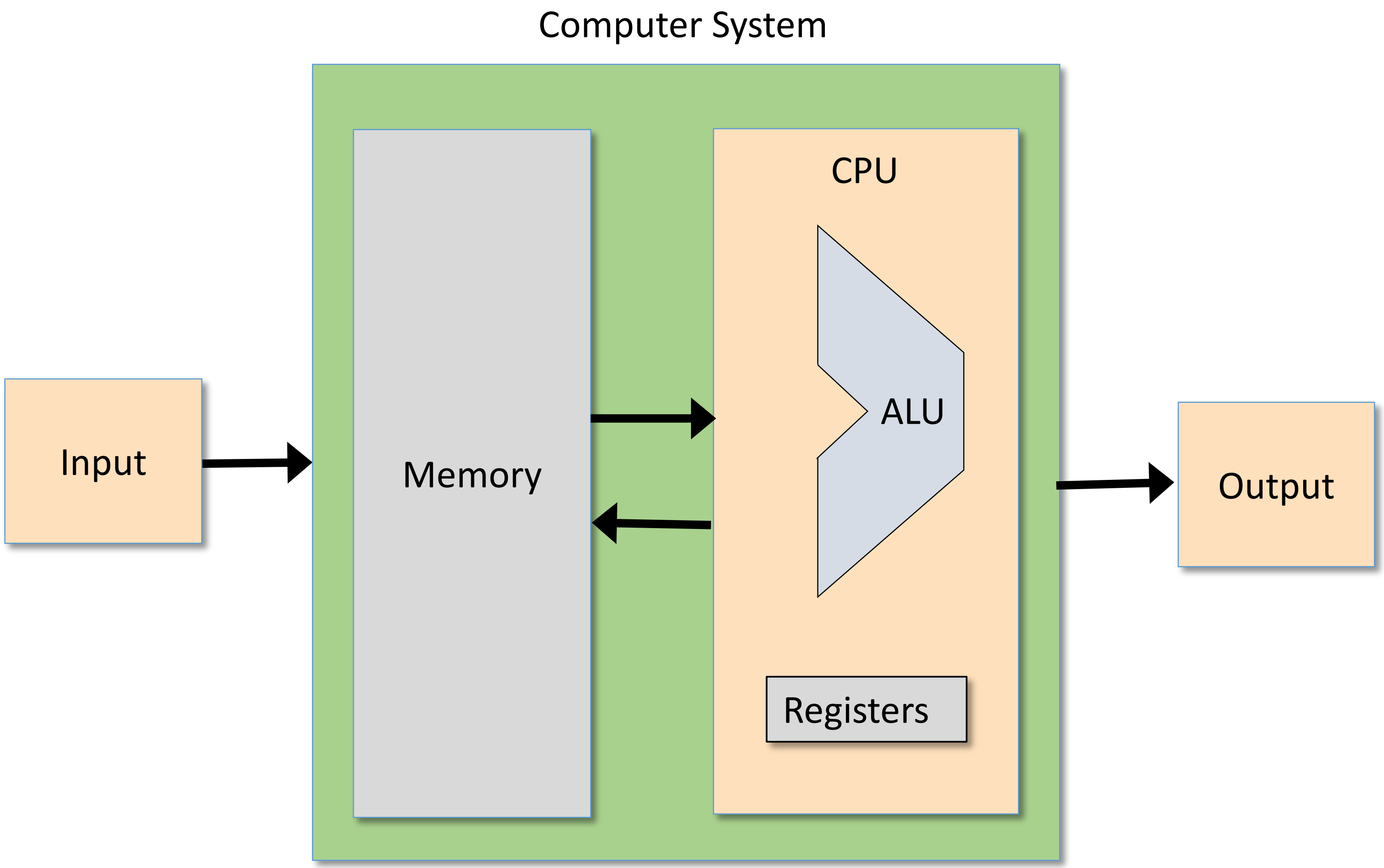
Theory

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

Practice

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

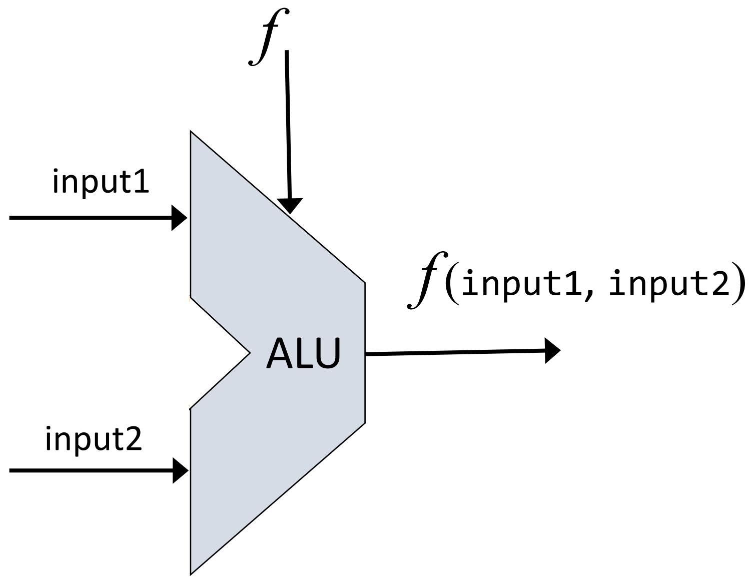
Von Neumann Architecture



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*, …)



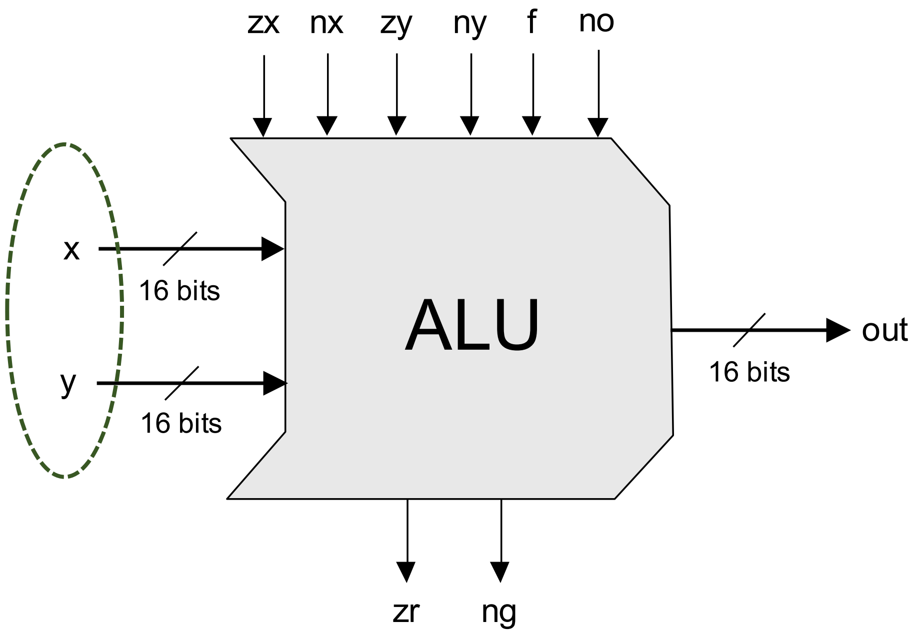
<u>Design issue: Which functions should the ALU perform?</u>

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.

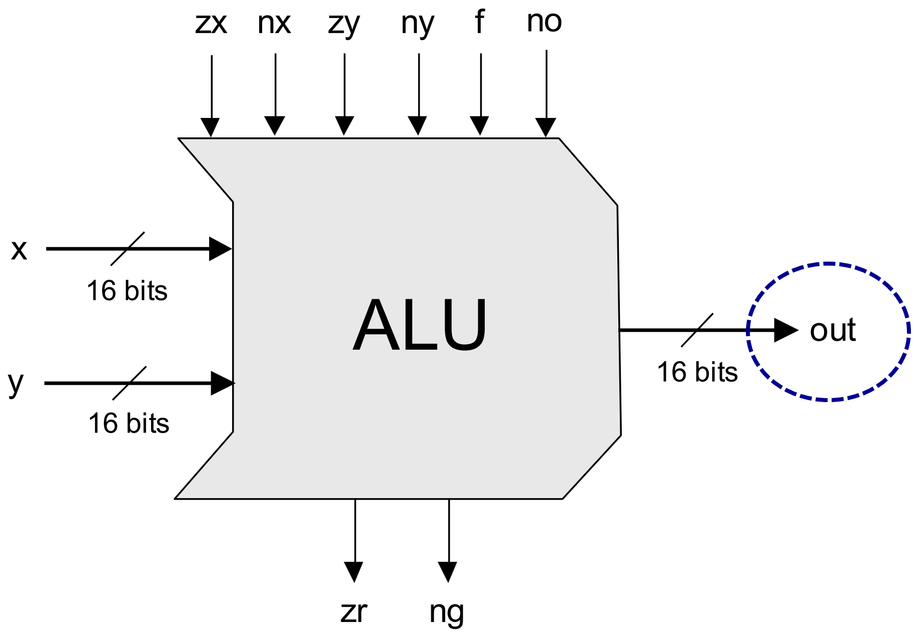
The Hack ALU

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



The Hack ALU

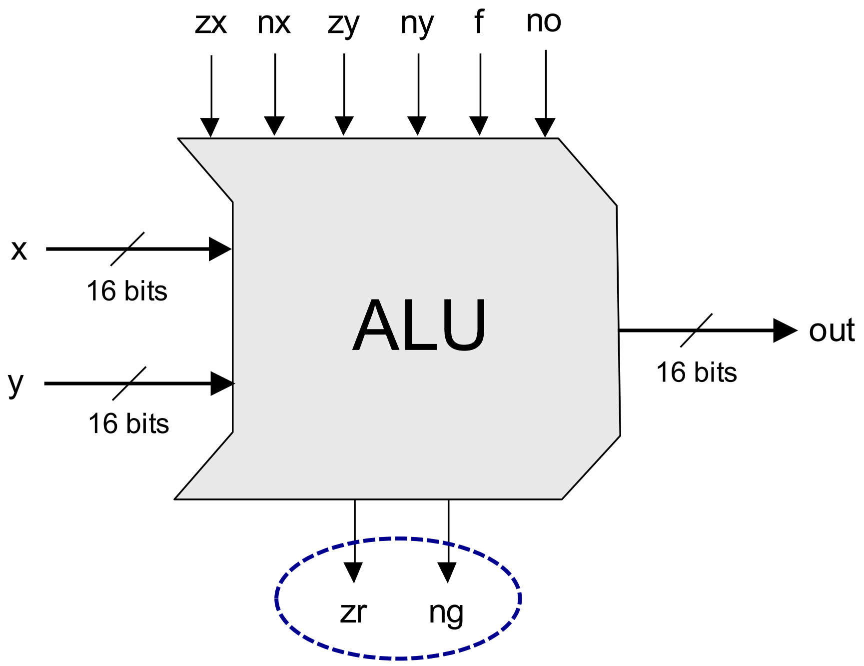
- Operates on two 16-bit, two's complement values
- Outputs a 16-bit, two's complement value



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

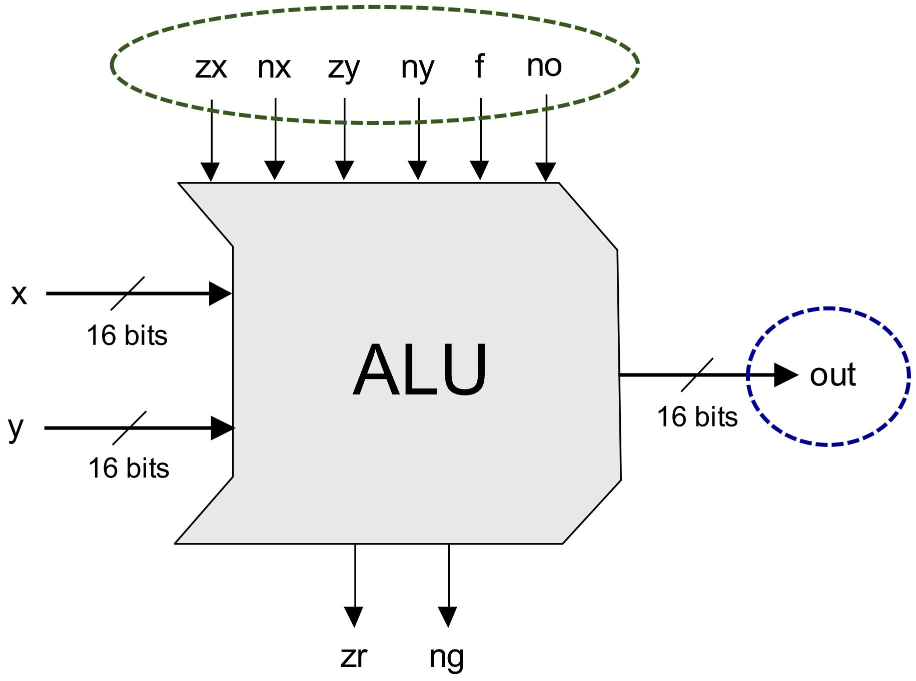
- Operates on two 16-bit, two's complement values
- Outputs a 16-bit, two's complement value
- Also outputs two 1-bit values (later)



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

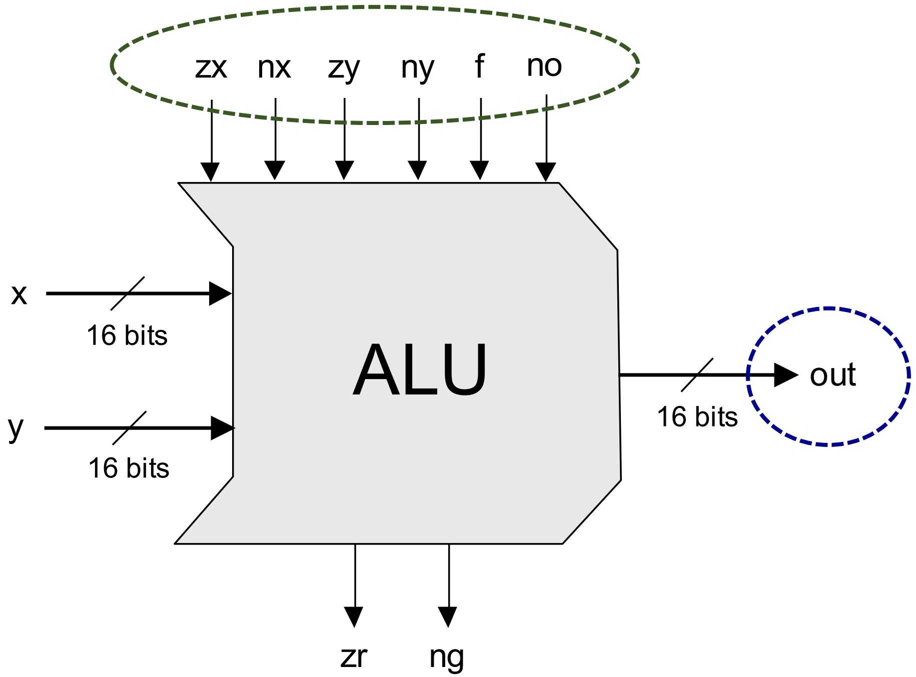


out
0
1
-1
x
y
!x
!y
-x
-y
x+1
y+1
x-1
y-1
x+y
x-y
y-x
x&y
x|y

The Hack ALU

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

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



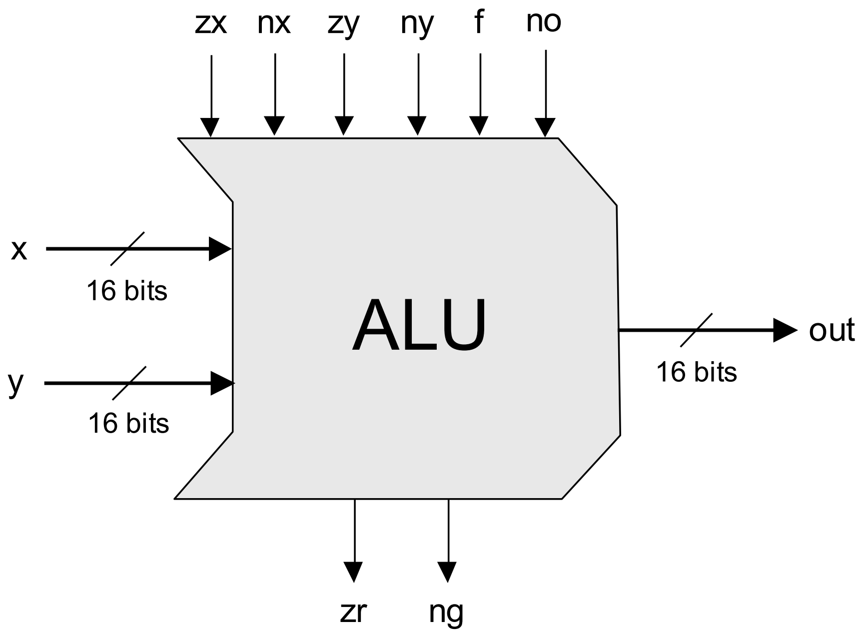
control bits

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 `y-x`

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

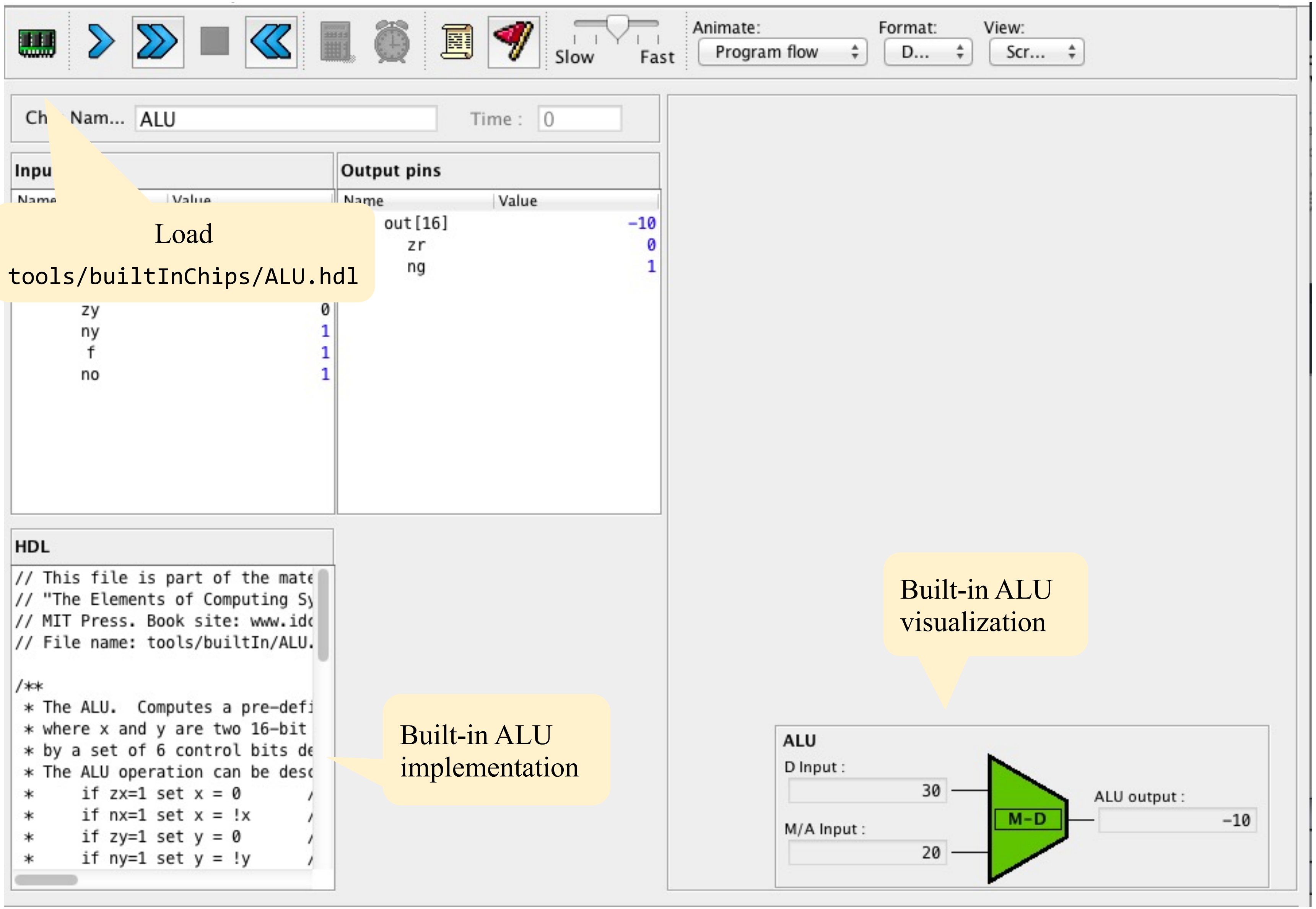
Set the control bits to one of the binary combinations listed in the table.



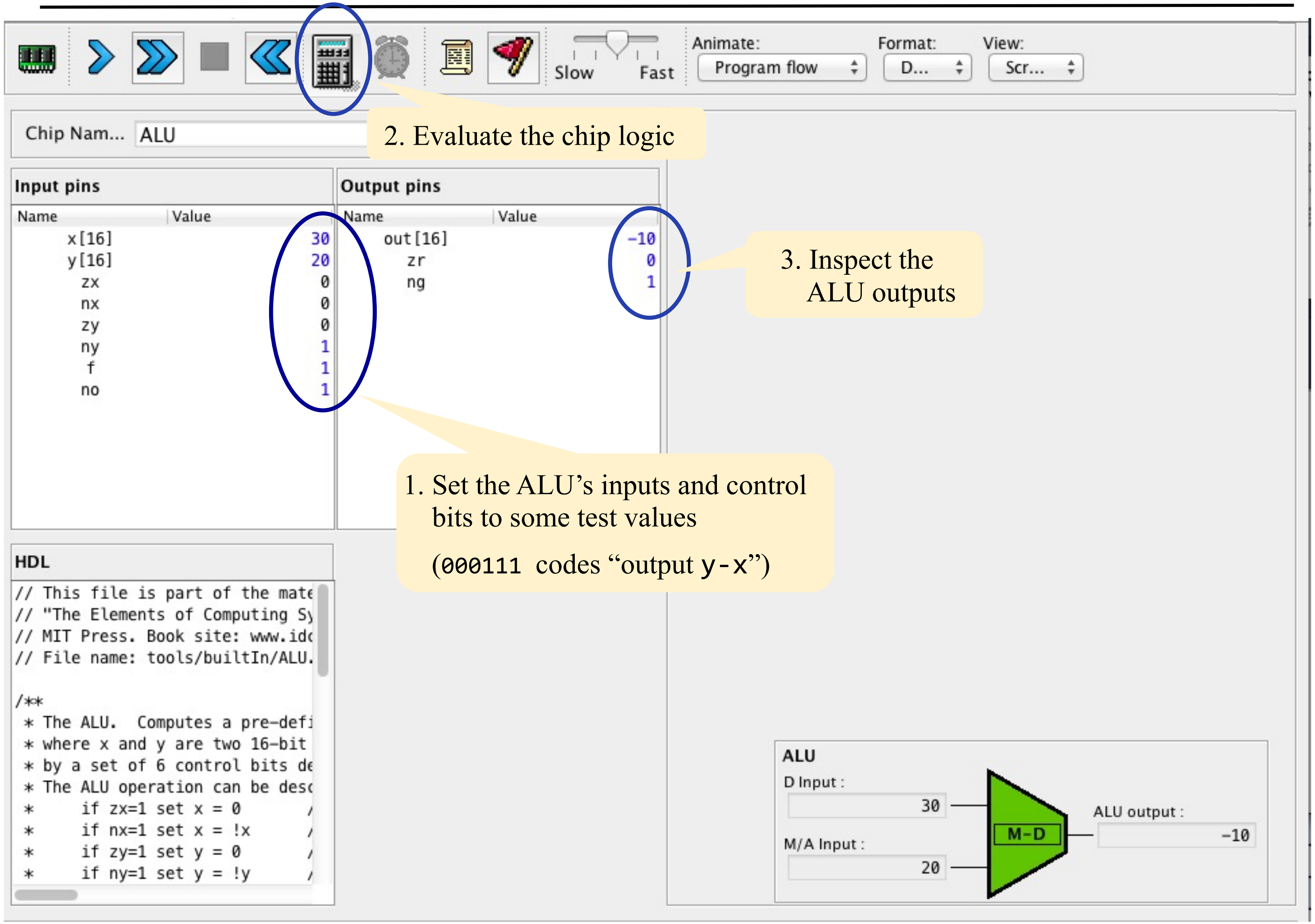
control bits

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 `y-x`



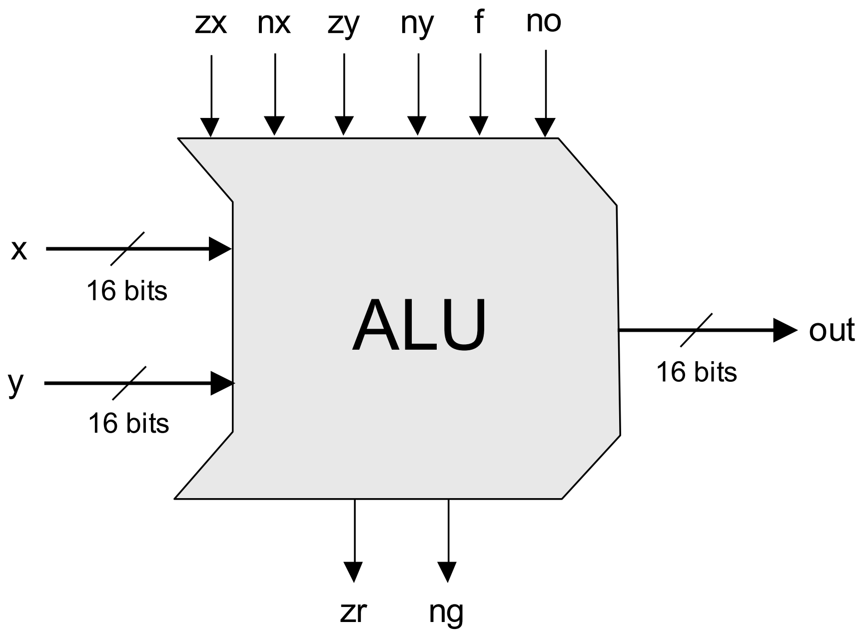
The Hack ALU in action: Compute `y-x`



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

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

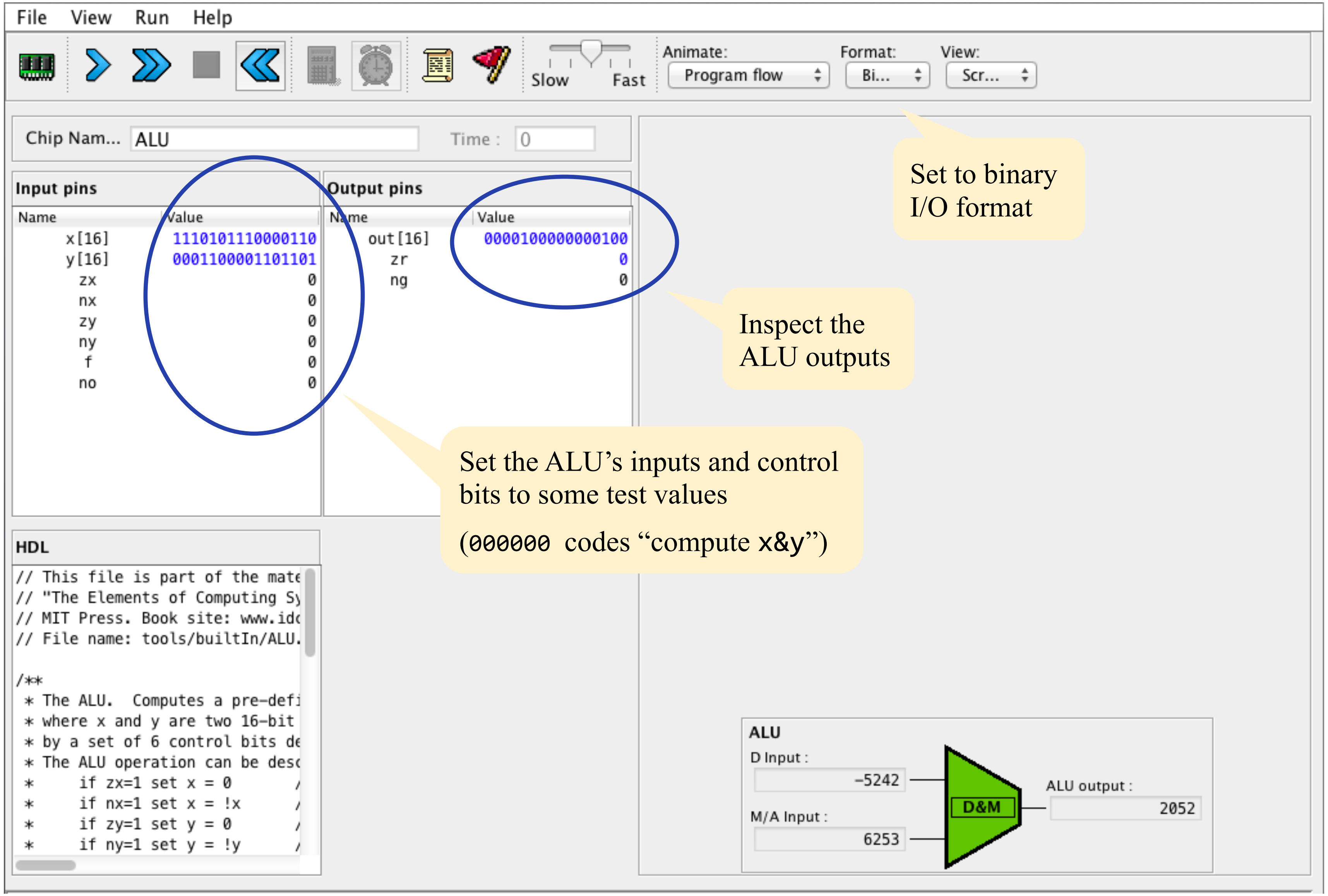
Set the control bits to one of the binary combinations listed in the table.



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

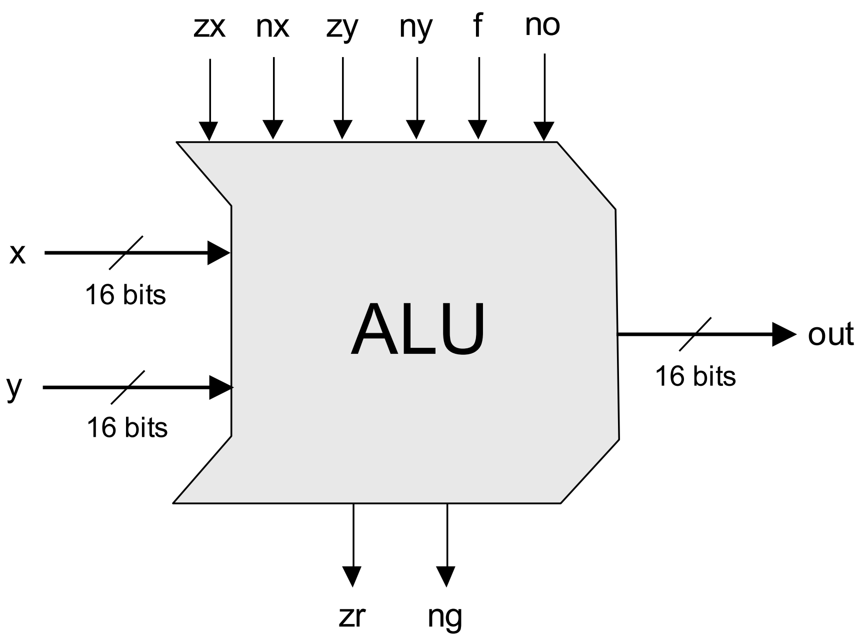
Inspect the ALU outputs

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 (don't care)
```

Following pre-setting:

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

Compute and post-set:

```
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			
0	0	1	1			
1	1	0	0			
0	0	1	1			
1	1	0	0			
0	0	1	1			
1	1	0	0			
0	1	1	1			
1	1	0	1			
0	0	1	1			
1	1	0	0			
0	0	0	0			
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
```

Compute and post-set:

```
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
```

Compute and post-set:

```
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

Example: compute `y-1`

```
x:    0 1 0 1 (don't care)
y:    0 1 1 0 (6)
```

Following pre-setting:

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

Compute and post-set:

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

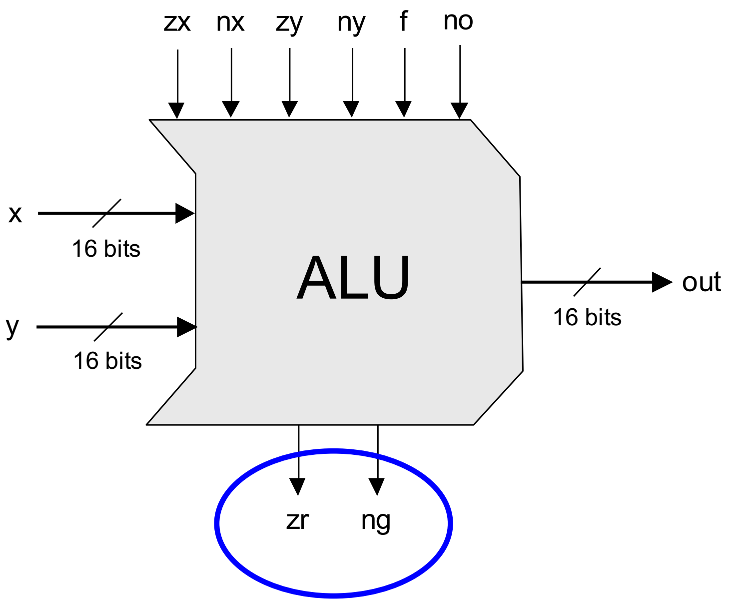
Practice:

See if you get

`0 1 0 1` (5)

The Hack ALU operation

One more detail:



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

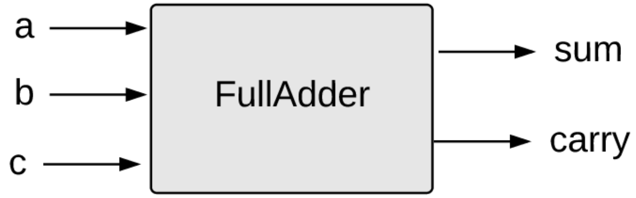
HalfAdder.hdl

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

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

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

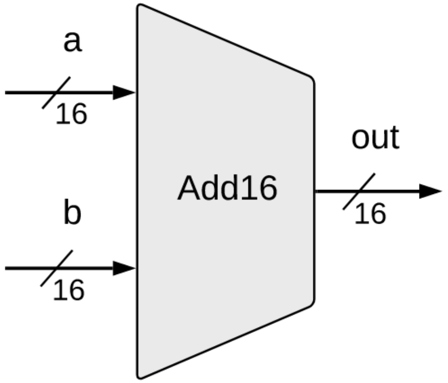
FullAdder.hdl

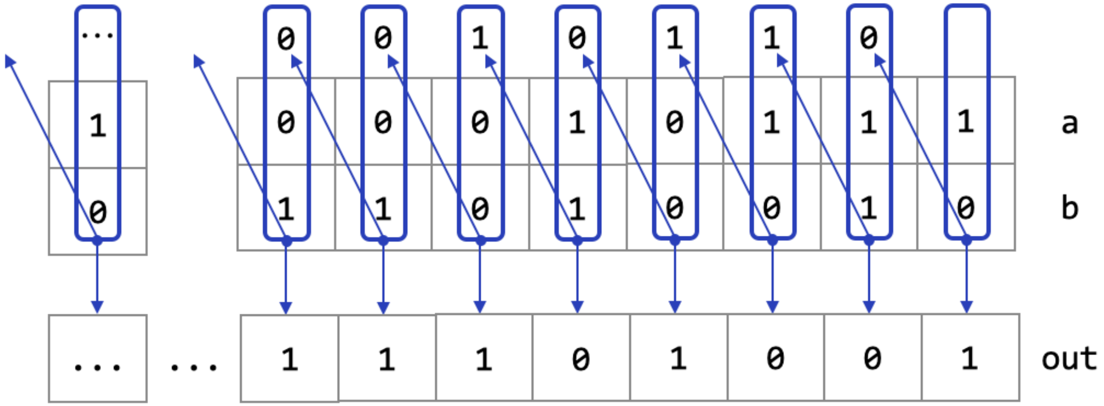
```
/** Computes the sum of three bits. */
CHIP FullAdder {
    IN a, b, c;
    OUT sum, carry;

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

<u>Implementation tip</u>

Can be built from two half-adders.

16-bit adder



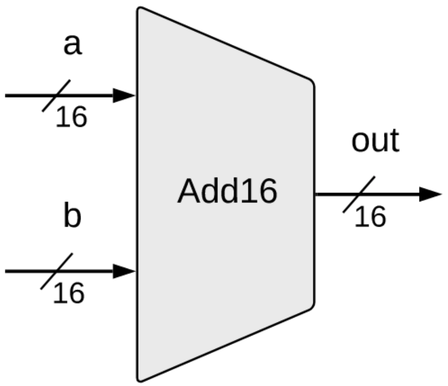


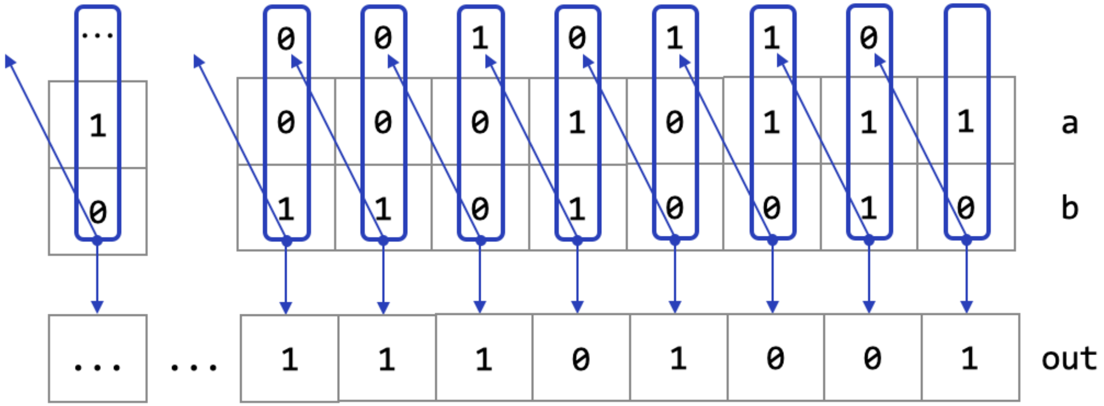
Add16.hdl

```
/* 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 chapter / lecture 3.

16-bit adder





Add16.hdl

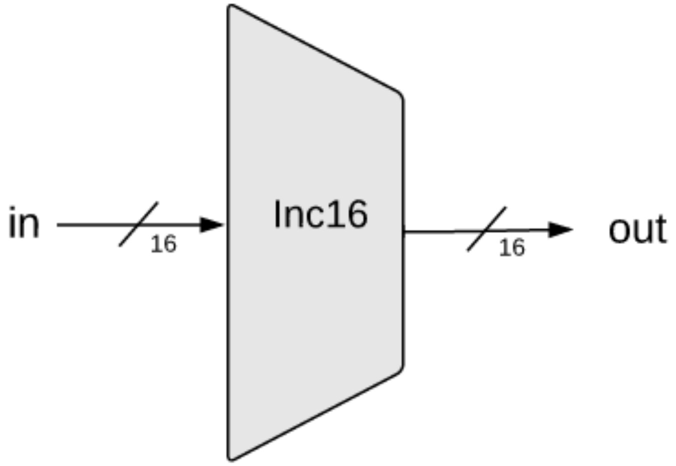
```
/* Adds two 16-bit, two's-complement values.
   The most-significant carry bit is ignored. */

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



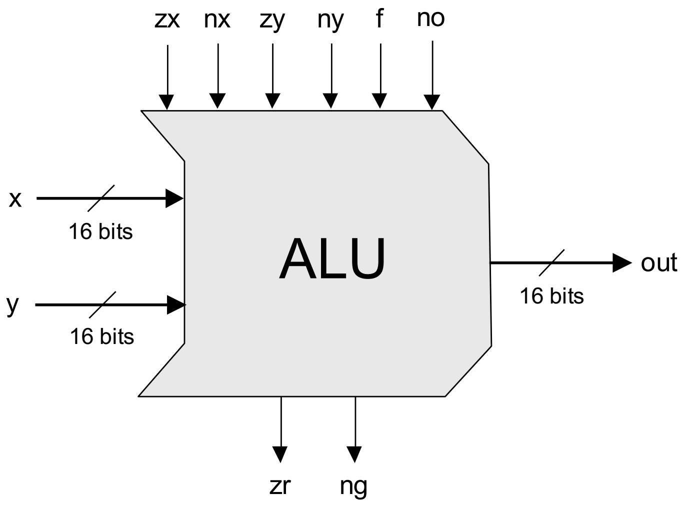
Inc16.hdl

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

<u>Implementation tip</u>

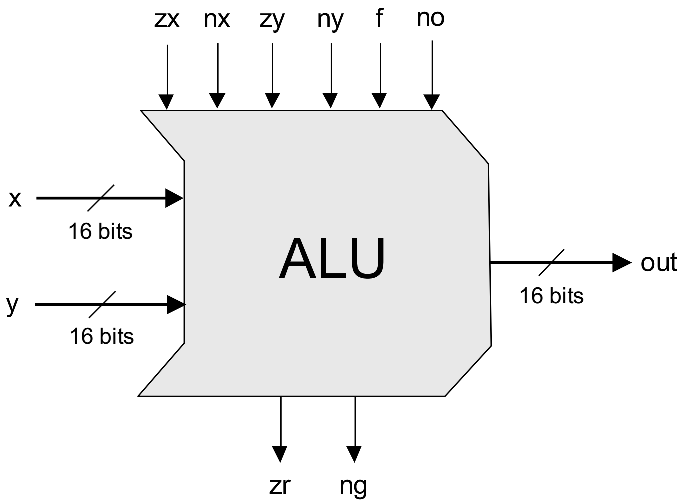
To set a bus-subset `x[i..j]` to `00…0` (or to `11…1`) in HDL,
use: `x[i..j] = false` (or `x[i..j] = true`)

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

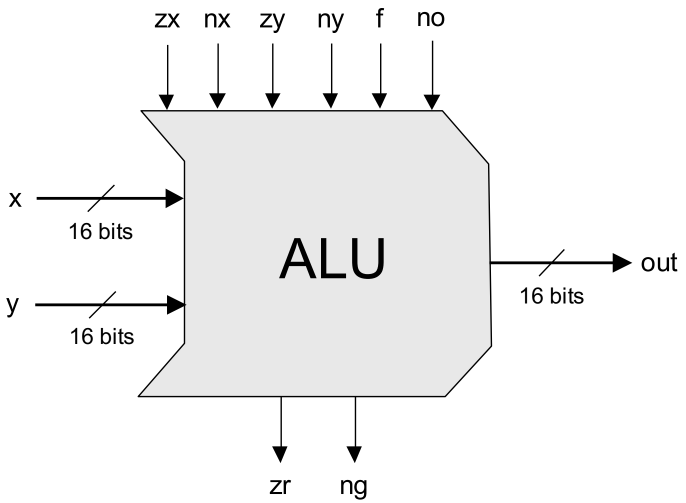


pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	**nx**	**zy**	**ny**	**f**	**no**	**out**
`if zx then x=0`	`if nx then x=!x`	`if zy then y=0`	`if ny then y=!y`	`if f then out=x+y else out=x&y`	`if no then out=!out`	`out(x,y)=`

ALU.hdl

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

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*

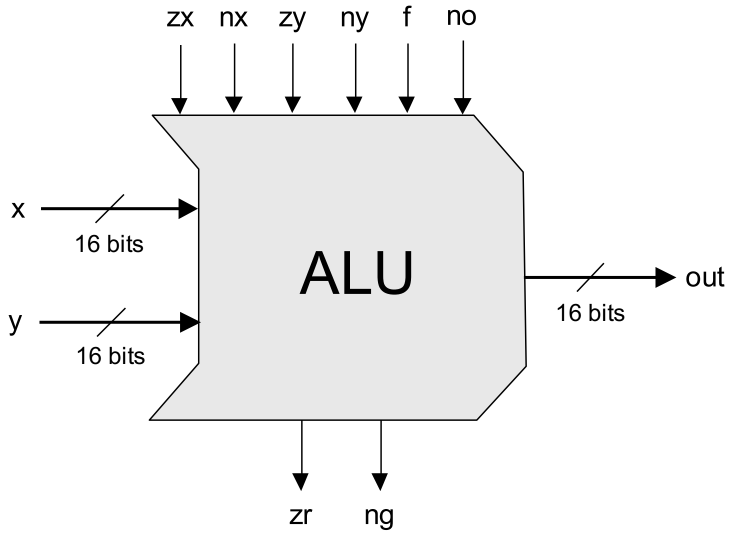
```
/* 16-bit adder */
CHIP Add16 {
    IN a[16], b[16];
    OUT out[16];
    PARTS:
    ...
}
```

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

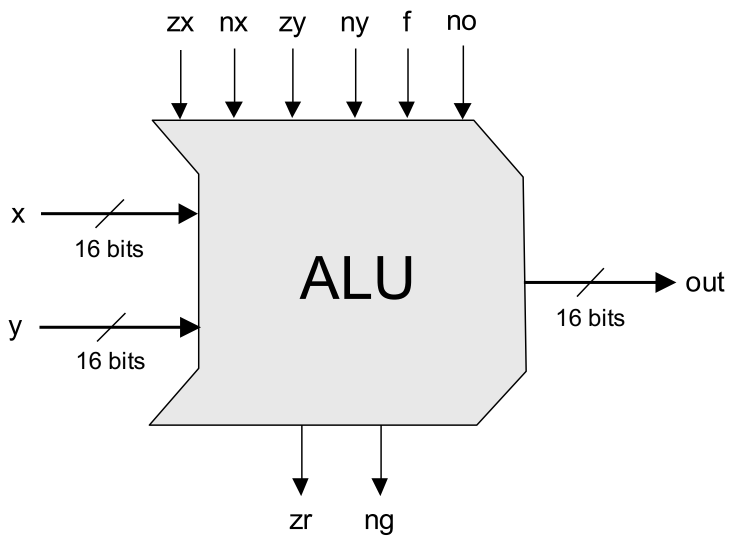
<u>To implement the ALU logic</u>:

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`

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;

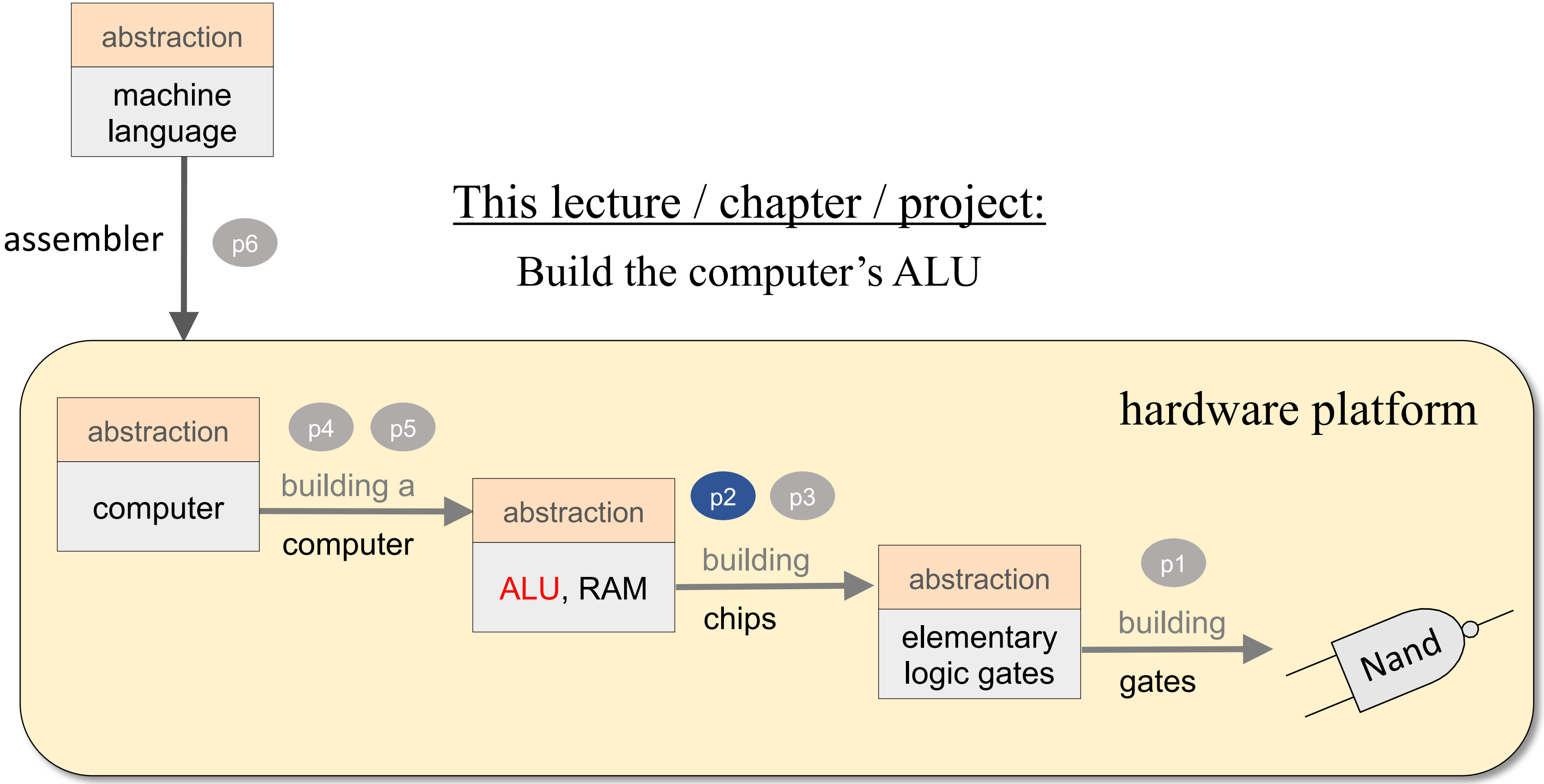
  For efficiency and consistency's sake, use their built-in versions, rather than your own HDL implementations

  Simple rule: Don't add any HDL files to the project 2 folder.

That's It!

Go Do Project 2!

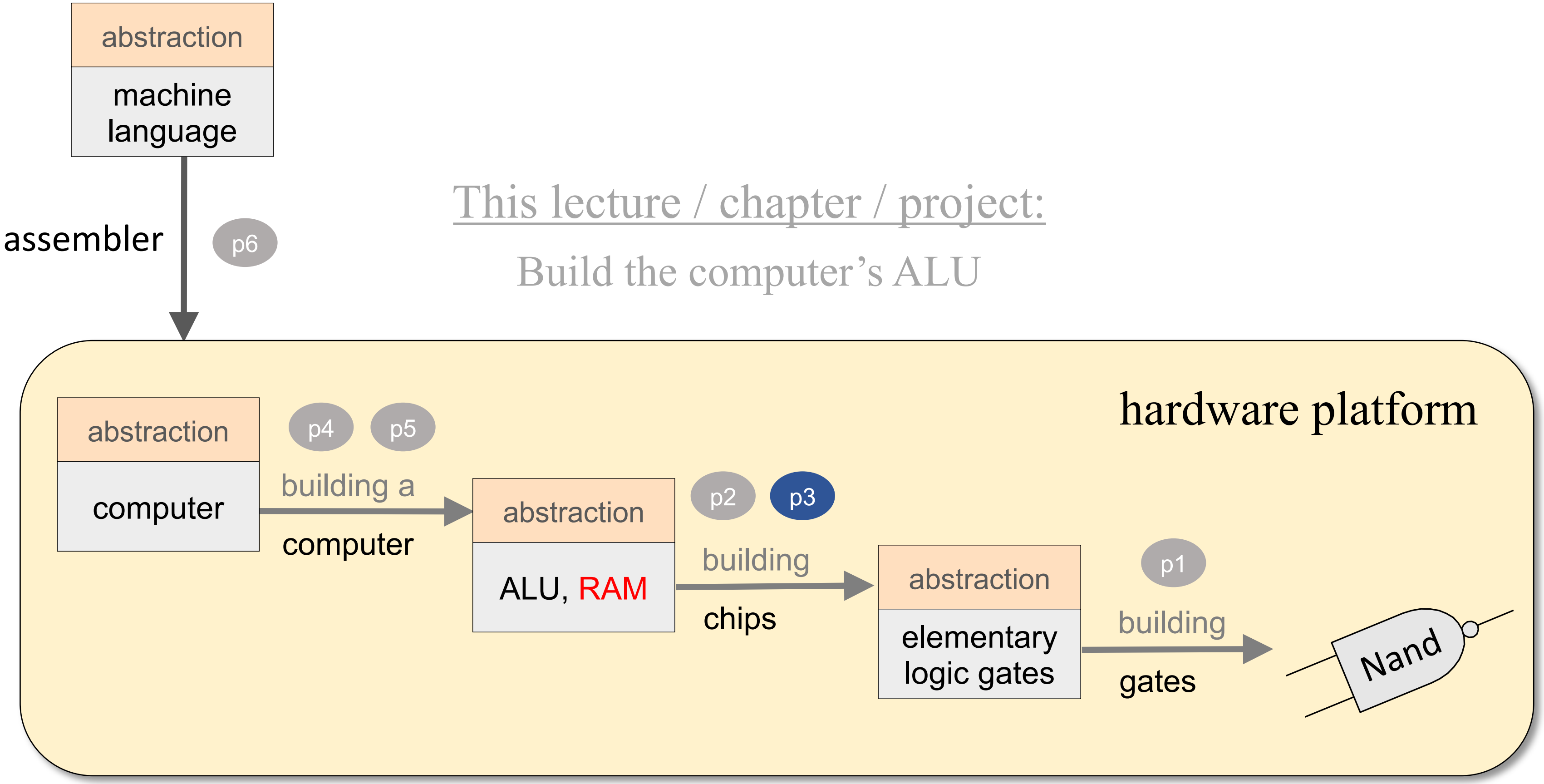
What's next?



<u>This lecture / chapter / project:</u>

Build the computer's ALU

What’s next?



Next lecture / chapter / project:

Build the computer’s RAM