

Lecture 2

Boolean Arithmetic

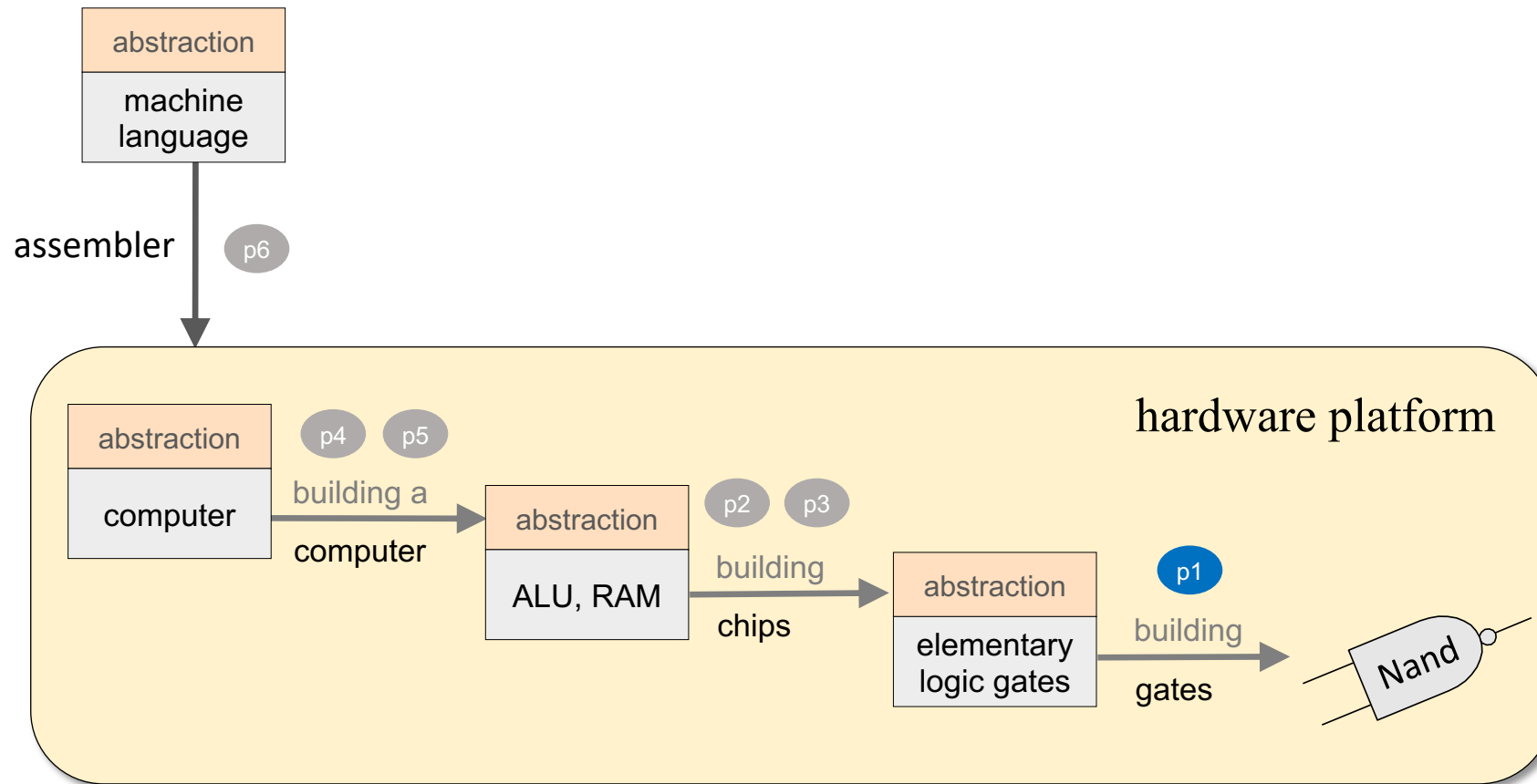
These slides support chapter 2 of the book

The Elements of Computing Systems

By Noam Nisan and Shimon Schocken

MIT Press, 2021

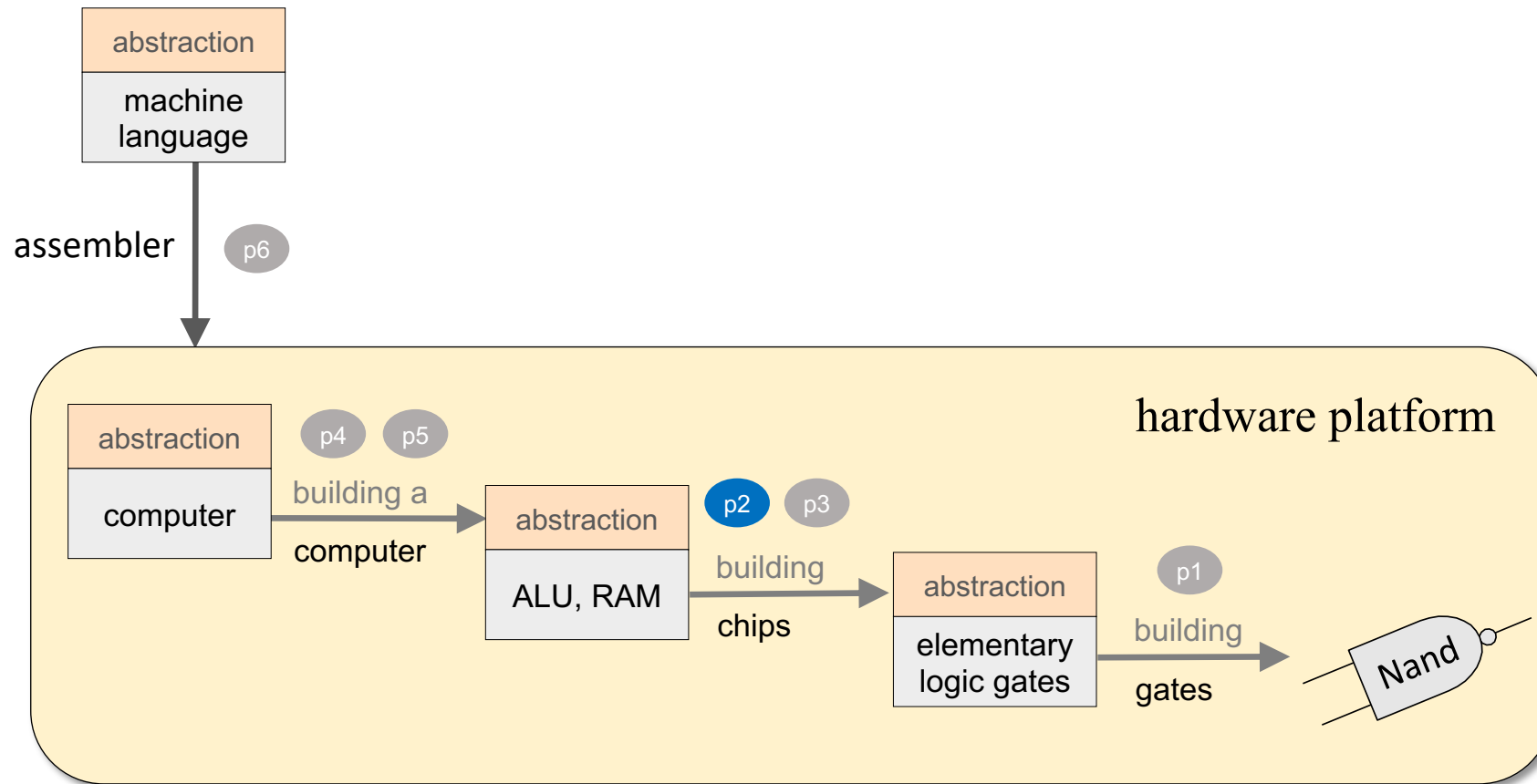
Nand to Tetris Roadmap: Hardware



Project 1

Build 15 elementary logic gates

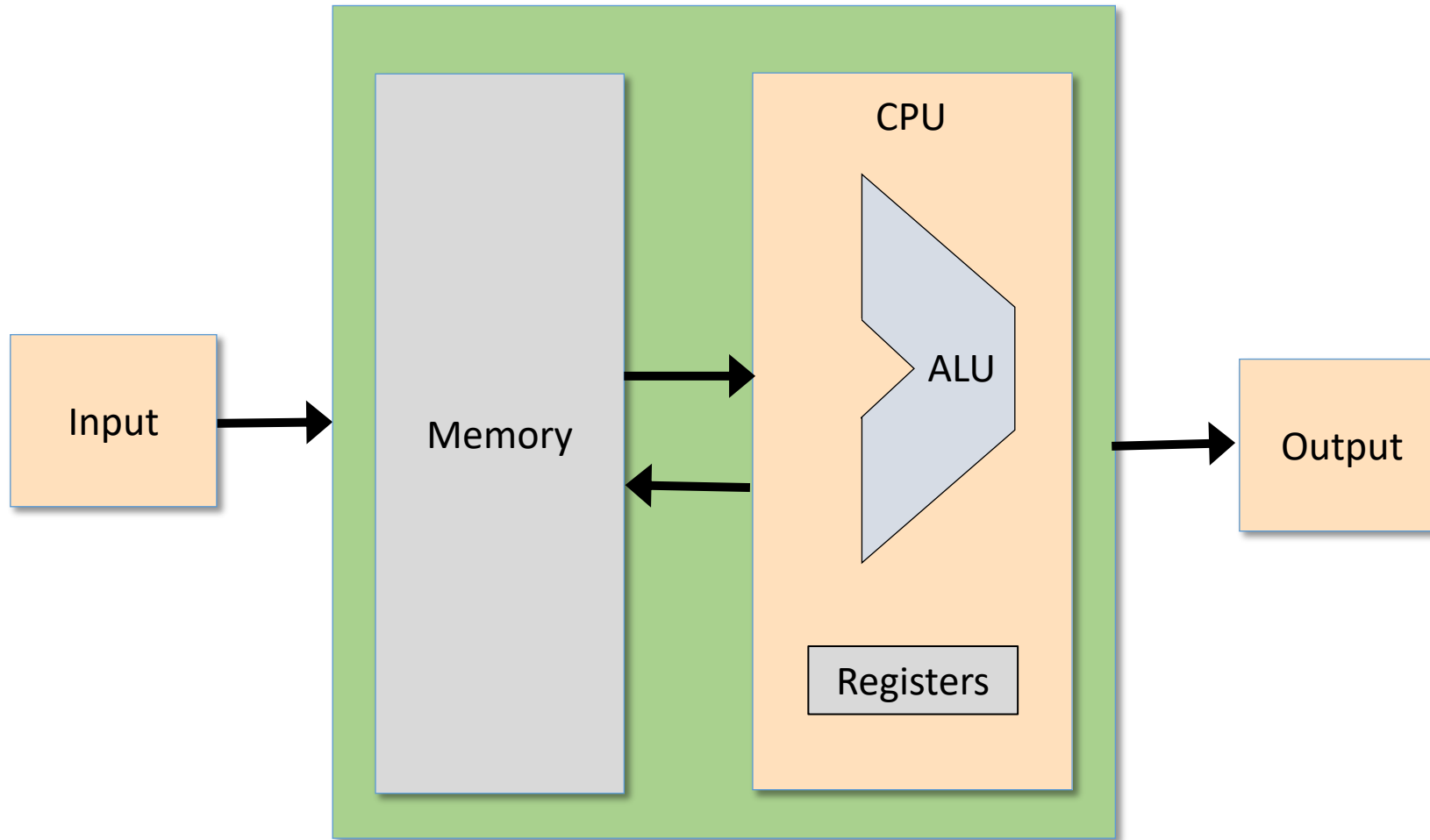
Nand to Tetris Roadmap: Hardware



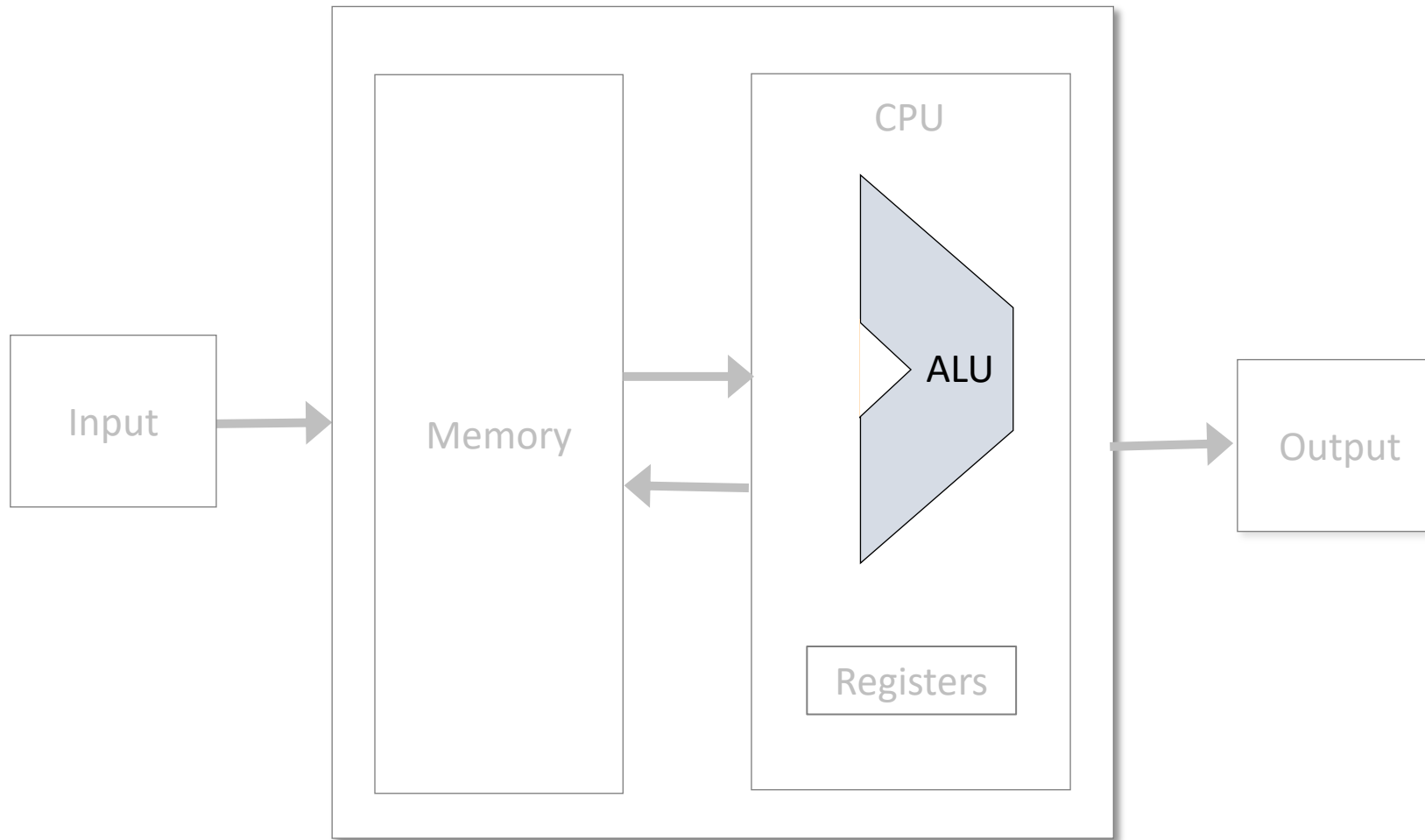
Project 2

Build chips that do arithmetic,
leading up to an ALU

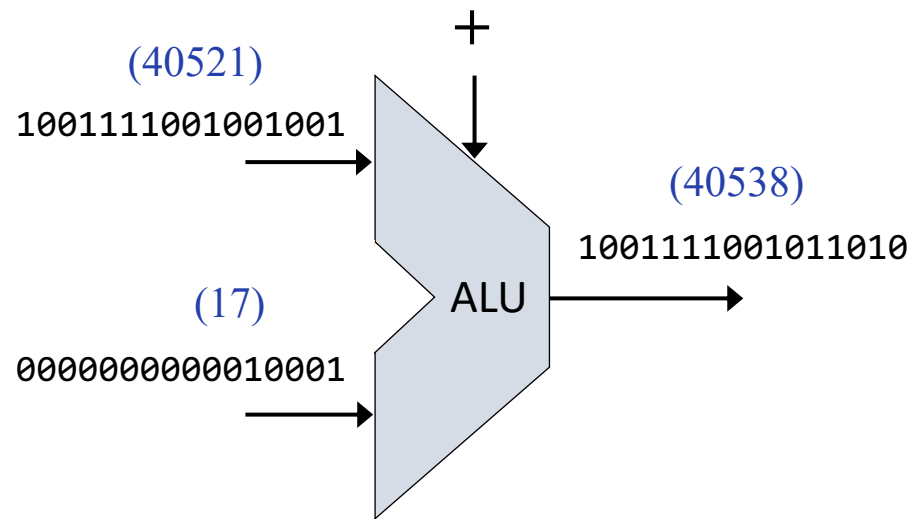
Computer system



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 , $!x$, ...

Challenges

- 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



This is not a pipe
(by René Magritte)

Representation

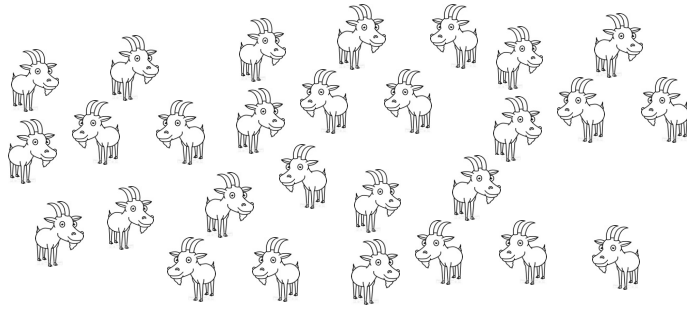


17

This is not seventeen.


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

A brief history of numeral systems



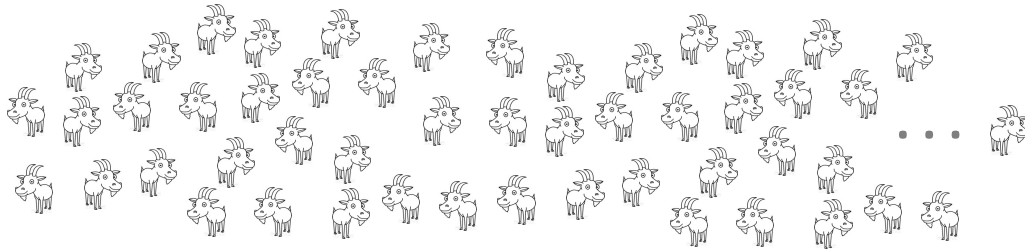
Twenty seven
goats

Unary: 

Egyptian: 


Roman: XXVII

A brief history of numeral systems



Six thousands,
five hundreds,
and seven goats

Unary: 

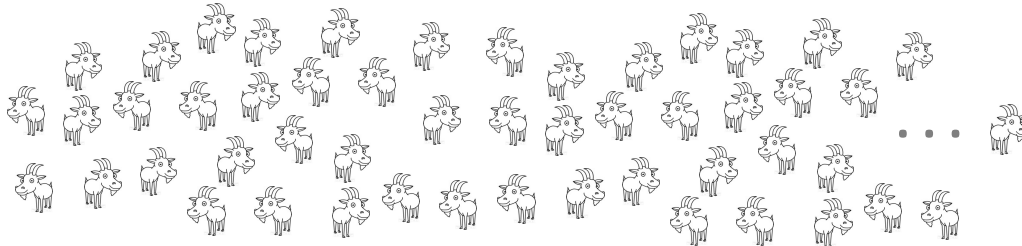
Egyptian: 

Roman: MMMMMMDVII

Old numeral systems:

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

Positional numeral system



Six thousands,
five hundreds,
and seven goats

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

The diagram shows the digits 6, 5, 0, 7 of the number 6507. Above each digit is its position index: 3 for 6, 2 for 5, 1 for 0, and 0 for 7. Lines connect each digit to its corresponding term in the summation formula below.

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

Positional representation

Digits: A fixed set of symbols, including 0

Base: The number of symbols

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.

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

Note: The method mentions no specific base.

Chapter 2: Boolean Arithmetic

Theory



Representing numbers



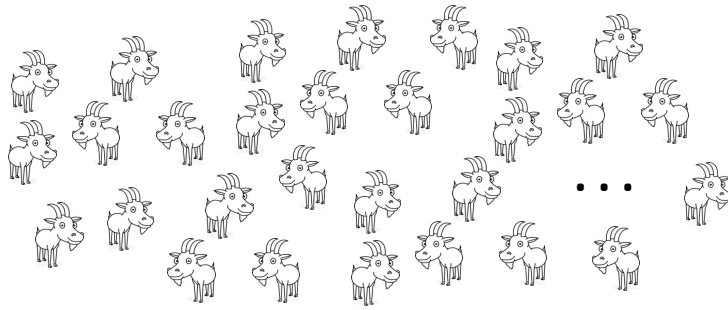
Binary numbers

- 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

Decimal (base 10) system:
Human friendly

3 2 1 0
7 0 5 3₁₀

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

Binary (base 2) system:
Computer friendly

12 11 10 ... 3 2 1 0
1 1 0 1 1 1 0 0 0 1 1 0 1₂

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

Binary and decimal systems

<u>Binary</u>	<u>Decimal</u>
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:

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

Decimal to binary:

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

Algorithm: What is the largest power of 2 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:

$$\text{decimal } (1011010_2) = ?$$

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

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:

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

Decimal to binary:

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

Algorithm: What is the largest power of 2 that “fits into” 53? It’s $2^5 = 32$.

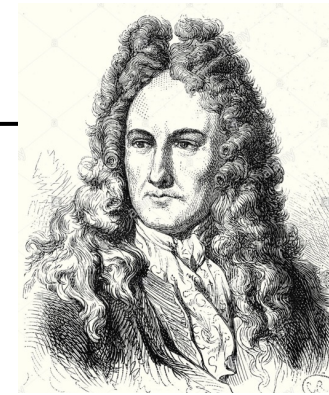
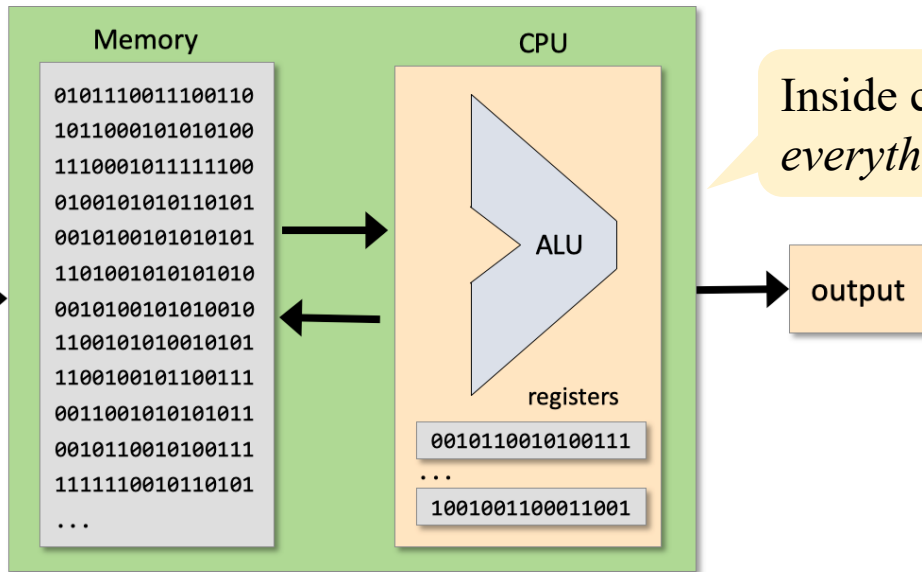
We still have to represent $53 - 32$, so, what is the largest power of 2 that fits into 21? It’s $2^4 = 16$, and so on.

Practice:

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

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

The binary system



G.W. Leibnitz
(1646 – 1716)

Binary numerals are easy to:

- | | |
|----------|----------|
| Compare | Verify |
| Add | Correct |
| Subtract | Store |
| Multiply | Transmit |
| Divide | Compress |
| ... | ... |



Leibnitz Medallion, 1697

Chapter 2: Boolean Arithmetic

Theory

✓ Representing numbers

✓ Binary numbers

➡ Boolean arithmetic

- Signed numbers

Practice

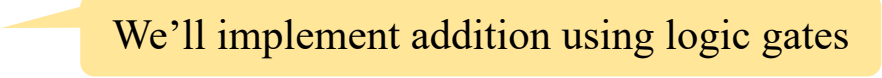
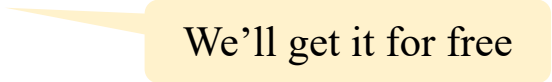
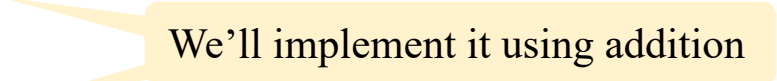
• Arithmetic Logic Unit (ALU)

• Project 2: Chips

• Project 2: Guidelines

Boolean arithmetic

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

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

Addition is the foundation of all arithmetic.

Addition

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

Binary addition

$$\begin{array}{rcccc} 0 & 1 & 1 & 0 \\ + & 7 & 8 & 7 & 5 \\ & & 5 & 6 & 2 \\ \hline 8 & 4 & 3 & 7 \end{array}$$

Decimal addition

Addition

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

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

Binary addition

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

Another example

$$\begin{array}{r} \textcolor{red}{1} \ 1 \ 1 \ 0 \\ + \begin{array}{|c|c|c|c|} \hline 0 & 1 & 1 & 1 \\ \hline 1 & 1 & 1 & 0 \\ \hline \end{array} \\ \hline \textcolor{red}{1} \begin{array}{|c|c|c|c|} \hline 0 & 1 & 0 & 1 \\ \hline \end{array} \end{array}$$

Another example

Overflow

Handling overflow

- Our approach: Ignore it
- As we'll soon see, ignoring the overflow bit is not a bug, it's a feature.

Addition

Word size $n = 16, 32, 64, \dots$

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

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)



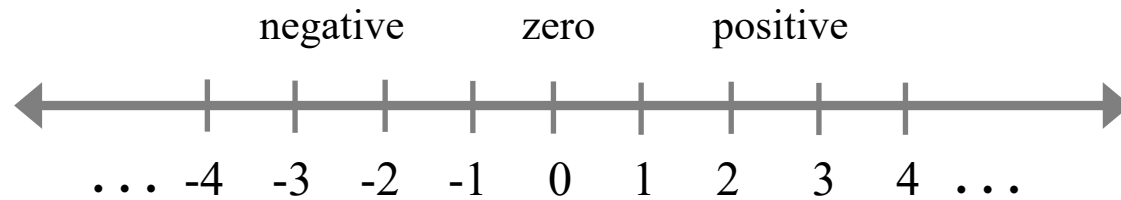
Signed numbers

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

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 ($x \text{ op } y$, $-x \text{ op } y$, $x \text{ op } -y$, $-x \text{ op } -y$, where $\text{op} = +, -, *, /$) 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 $0 \dots 2^n - 1$

What about negative numbers?

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

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

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 ($2^n - \text{decimal}(\text{bits})$)

Two's complement: Addition

code(x)	x		<u>Compute $x + y$ where x and y are signed</u>	
0000	0	0	Algorithm: Regular addition, modulo 2^n	
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

$$\begin{array}{rcl}
 + 6 & = & + 6 \\
 -2 & & \underline{14} \\
 & & 20 \% 16 = 4 \text{ codes } 4
 \end{array}$$

$$\begin{array}{rcl}
 + 3 & = & + 3 \\
 -5 & & \underline{11} \\
 & & 14 \% 16 = 14 \text{ codes } -2
 \end{array}$$

$$\begin{array}{rcl}
 -2 & & 14 \\
 + -5 & = & + \underline{11} \\
 & & 25 \% 16 = 9 \text{ codes } -7
 \end{array}$$

Two's complement: Addition

code(x)	x
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

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

Algorithm: Regular addition, modulo 2^n

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

Practice:

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

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

Two's complement: Addition

code(x)	x		Compute $x + y$ where x and y are signed
0000	0	0	Algorithm: Regular addition, modulo 2^n
0001	1	1	
0010	2	2	
0011	3	3	$\begin{array}{r} + 6 \\ -2 \end{array} = \begin{array}{r} + 6 \\ 14 \end{array}$
0100	4	4	$20 \% 16 = 4 \text{ codes } 4$
0101	5	5	
0110	6	6	Practice:
0111	7	7	
1000	8	-8	
1001	9	-7	$\begin{array}{r} + 4 \\ -7 \end{array} = \begin{array}{r} + 4 \\ 9 \end{array}$
1010	10	-6	$13 \% 16 = 13 \text{ codes } -3$
1011	11	-5	
1100	12	-4	
1101	13	-3	$\begin{array}{r} -2 \\ + -4 \end{array} = \begin{array}{r} 14 \\ + 12 \end{array}$
1110	14	-2	$26 \% 16 = 10 \text{ codes } -6$
1111	15	-1	

Two's complement: Addition

code(x)	x
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

At the binary level (same algorithm):

$$\begin{array}{rcl}
 +6 & = & +0110 \\
 -2 & & \underline{+1110} \\
 & & \text{codes } 4
 \end{array}$$

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

$$\begin{array}{rcl}
 +3 & = & +0011 \\
 -5 & & \underline{+1011} \\
 & & 1110 \text{ codes } -2
 \end{array}$$

$$\begin{array}{rcl}
 -2 & & +1110 \\
 + -5 & = & \underline{+1011} \\
 & & \text{codes } -7
 \end{array}$$

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):

$$\begin{array}{r}
 6 \\
 + \\
 \hline
 -2 \\
 + \\
 \hline
 10100 \quad \text{codes } 4
 \end{array}$$

More examples:

$$\begin{array}{r}
 5 \\
 + \\
 \hline
 7 \\
 + \\
 \hline
 1100 \quad \text{codes } -4 \quad ???
 \end{array}$$

$$\begin{array}{r}
 -7 \\
 + \\
 \hline
 -3 \\
 + \\
 \hline
 10110 \quad \text{codes } 6 \quad ???
 \end{array}$$

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

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

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

Example: Convert 0010 (2)

$$\begin{array}{r}
 1101 \text{ (flipped)} \\
 + \quad 1 \\
 \hline
 1110 \text{ (-2)}
 \end{array}$$

Two's complement: Sign conversion

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

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)

Two's complement: Sign conversion

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

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)

$$\begin{array}{r}
 0101 \text{ (flipped)} \\
 + \quad 1 \\
 \hline
 0110 \text{ (6)}
 \end{array}$$

Two's complement: Recap

code(x)	x	Observations
0000	0	<ul style="list-style-type: none">Using n bits, the method represents all the integers in the range $-2^{n-1}, \dots, -1, 0, 1, \dots, 2^{n-1} - 1$$code(x) + code(-x) = code(0)$The codes are monotonically increasingArithmetic on signed integers is the same as arithmetic on unsigned integersAddition / subtraction / conversion are $O(n)$Simple! Elegant! Powerful!
0001	1	
0010	2	
0011	3	
0100	4	
0101	5	
0110	6	
0111	7	
1000	8	<u>Implications for hardware designers</u> Arithmetic on signed integers can be implemented using <i>the same hardware</i> used for handling arithmetic of unsigned integers
1001	9	
1010	10	
1011	11	
1100	12	
1101	13	
1110	14	
1111	15	

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

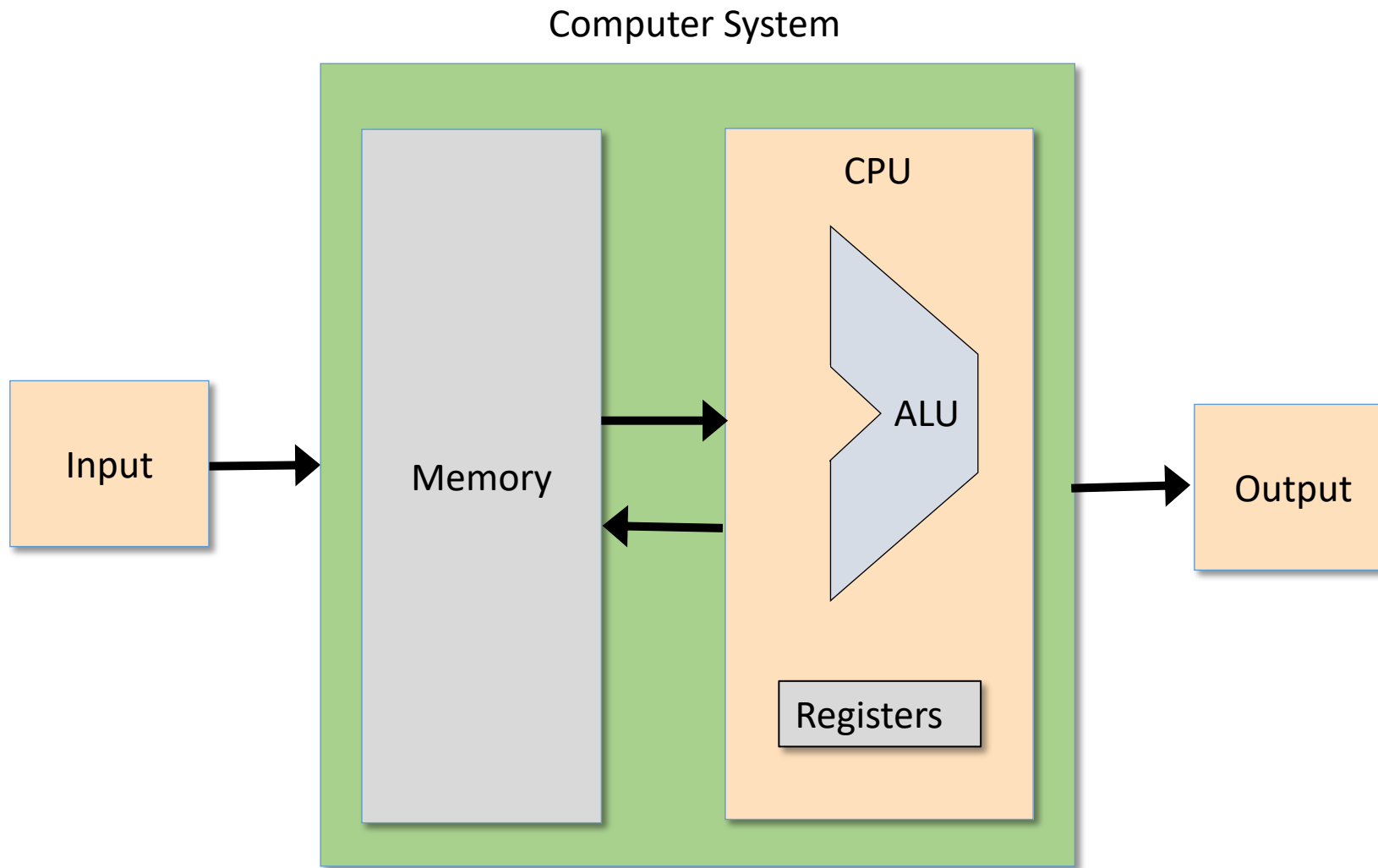
Practice



Arithmetic Logic Unit (ALU)

- Project 2: Chips
- Project 2: Guidelines

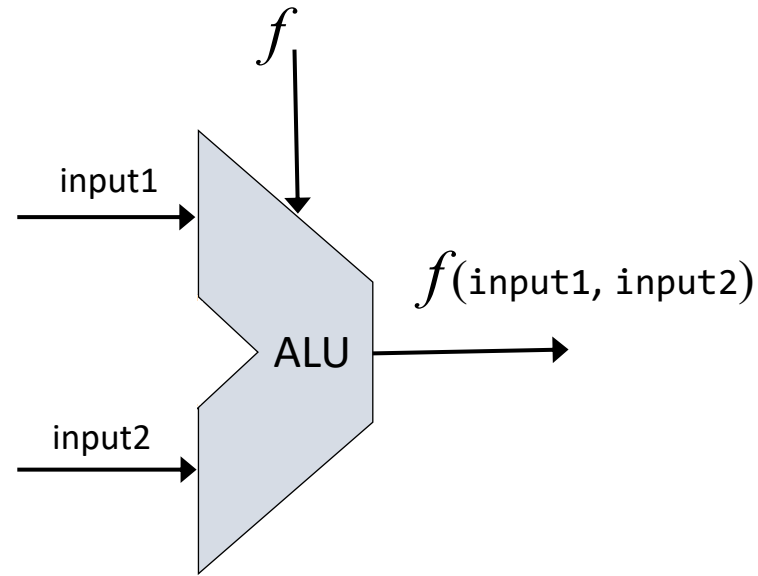
Von Neumann Architecture



The Arithmetic Logical Unit

The ALU computes a given function on 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, ...*)



Design issue: Which functions should the ALU perform?

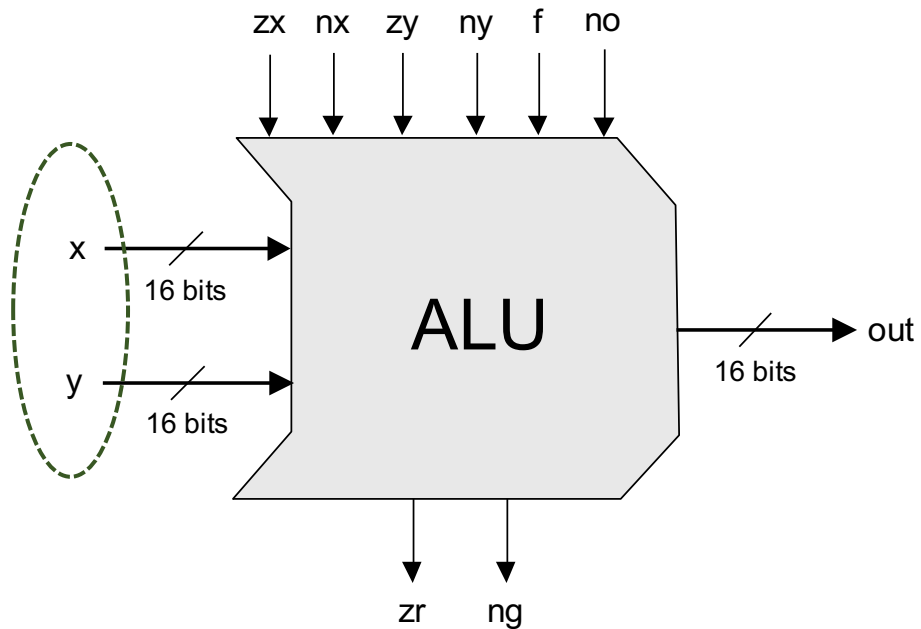
A hardware / software tradeoff:

Functions not implemented by the ALU can be implemented later by software

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

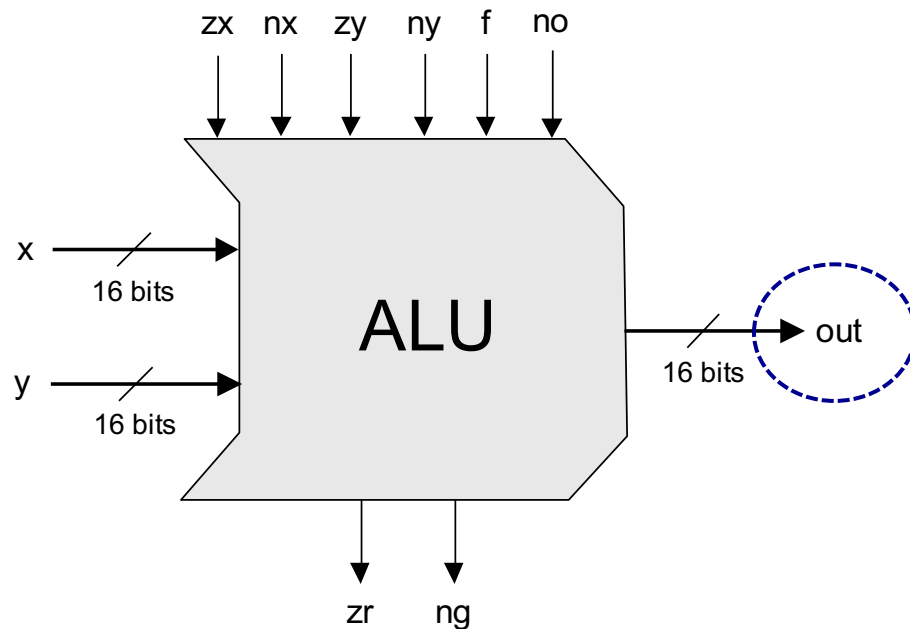
The Hack ALU

- 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

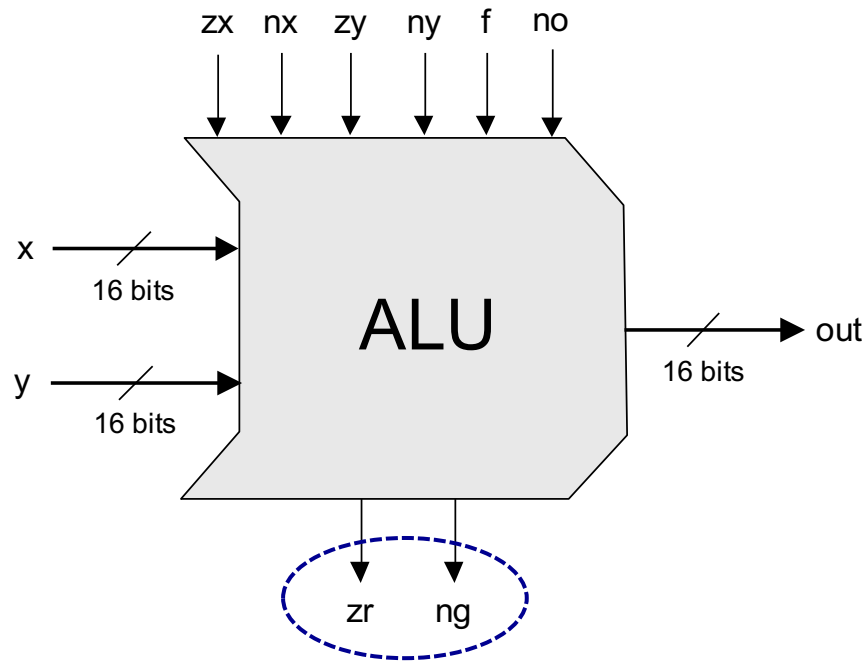


out

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

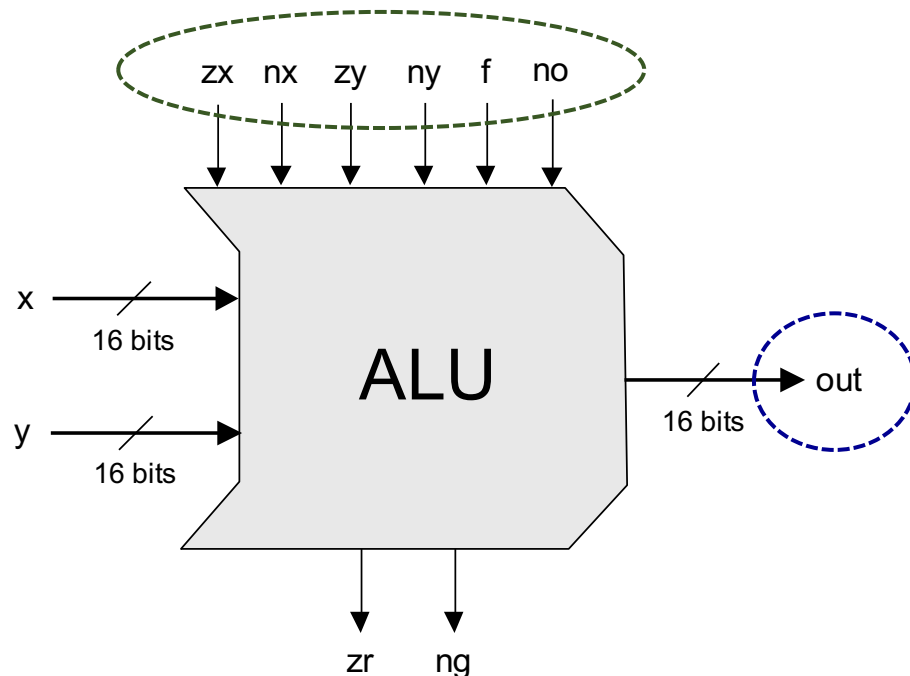
- Operates on two 16-bit, two's complement values
- Outputs a 16-bit, two's complement value
- Also outputs two 1-bit values (later)



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

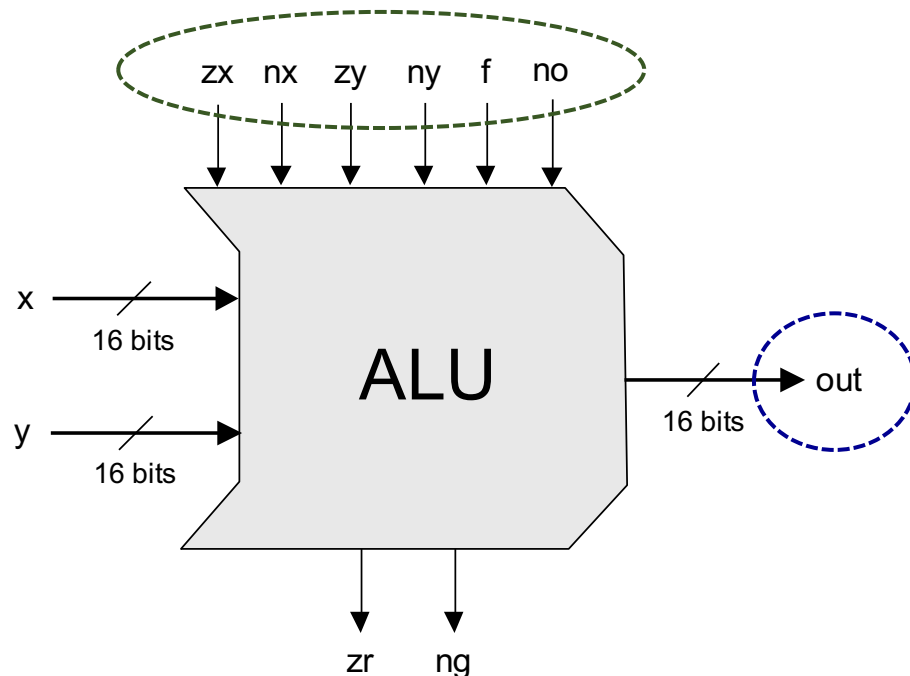


out
0
1
-1
x
y
!x
!y
-x
-y
x+1
y+1
x-1
y-1
x+y
x-y
y-x
x&y
x y

The Hack ALU

To cause the ALU to compute a function:

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

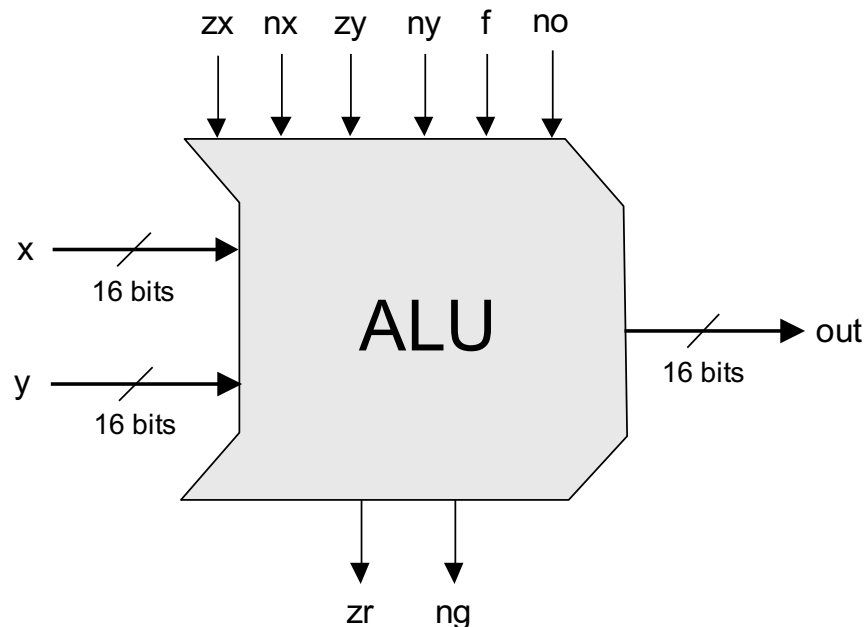


control bits						
zx	nx	zy	ny	f	no	out
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	<i>x</i>
1	1	0	0	0	0	<i>y</i>
0	0	1	1	0	1	! <i>x</i>
1	1	0	0	0	1	! <i>y</i>
0	0	1	1	1	1	- <i>x</i>
1	1	0	0	1	1	- <i>y</i>
0	1	1	1	1	1	<i>x</i> +1
1	1	0	1	1	1	<i>y</i> +1
0	0	1	1	1	0	<i>x</i> -1
1	1	0	0	1	0	<i>y</i> -1
0	0	0	0	1	0	<i>x</i> + <i>y</i>
0	1	0	0	1	1	<i>x</i> - <i>y</i>
0	0	0	1	1	1	<i>y</i> - <i>x</i>
0	0	0	0	0	0	<i>x</i> & <i>y</i>
0	1	0	1	0	1	<i>x</i> <i>y</i>

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 screenshot shows the Nand2Tetris IDE interface. At the top is a toolbar with icons for running, stepping, and other simulation controls. Below the toolbar, the 'Ch Nam...' field is set to 'ALU' and the 'Time' is 0. The 'Input' table shows control signals: 'zy' (0), 'ny' (1), 'f' (1), and 'no' (1). The 'Output pins' table shows: 'out[16]' (-10), 'zr' (0), and 'ng' (1). A yellow callout bubble points to the 'Load' button and the file path 'tools/builtInChips/ALU.hdl'. The 'HDL' section displays the source code for the ALU chip, which implements the operation $y-x$ based on control signals. A yellow callout bubble points to the HDL code with the text 'Built-in ALU implementation'. The right side of the IDE shows a visualization of the ALU chip. It has two inputs: 'D Input' (30) and 'M/A Input' (20). The output is 'ALU output' (-10). A green trapezoidal symbol labeled 'M-D' represents the ALU operation. A yellow callout bubble points to this visualization with the text 'Built-in ALU visualization'.

Ch Nam... ALU Time : 0

Input		Output pins	
Name	Value	Name	Value
zy	0	out[16]	-10
ny	1	zr	0
f	1	ng	1
no	1		

Load
tools/builtInChips/ALU.hdl

HDL

```
// This file is part of the material for "The Elements of Computing Systems"
// by Noam Nisan and Shimon Schocken. Copyright 2005 MIT Press.
// MIT Press. Book site: www.nand2tetris.org
// File name: tools/builtIn/ALU.hdl

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

Built-in ALU implementation

Built-in ALU visualization

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

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

The screenshot shows the Nand2Tetris IDE interface. At the top, a toolbar contains icons for simulation control, with a calculator icon circled in blue. Below the toolbar, the 'Chip Nam...' field is set to 'ALU'. The main workspace is divided into three sections: 'Input pins', 'Output pins', and 'HDL'.

Input pins table:

Name	Value
x[16]	30
y[16]	20
zx	0
nx	0
zy	0
ny	1
f	1
no	1

Output pins table:

Name	Value
out[16]	-10
zr	0
ng	1

HDL code:

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

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

ALU Diagram:

The diagram shows the ALU chip with two inputs: 'D Input' (30) and 'M/A Input' (20). These inputs are connected to a green trapezoidal block labeled 'M-D'. The output of this block is labeled 'ALU output' and shows the value -10.

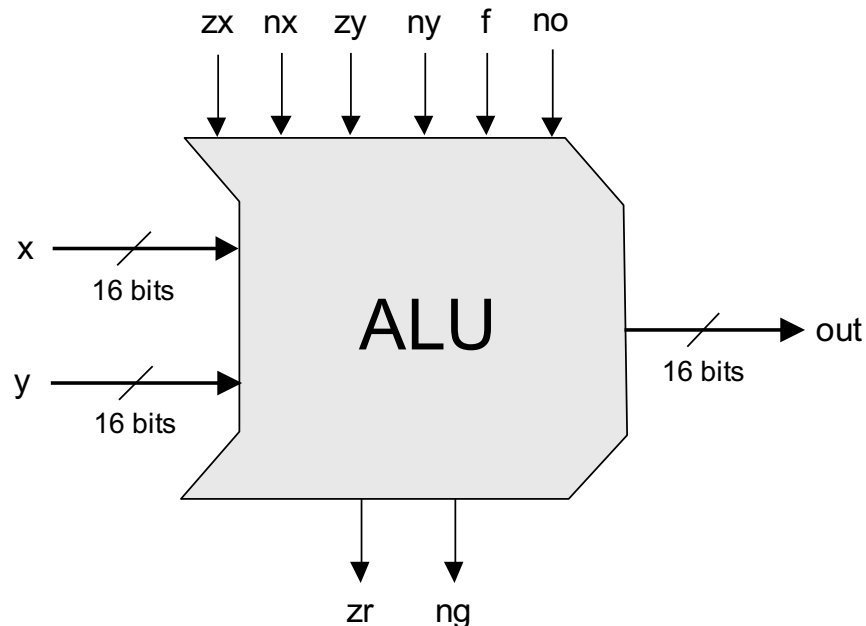
Annotations:

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

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 $x \& y$

The screenshot shows the Hack ALU simulator interface. The top menu bar includes File, View, Run, and Help. Below the menu is a toolbar with icons for running, pausing, and stepping through the simulation, along with a speed slider (Slow to Fast) and dropdowns for Animate (Program flow), Format (Binary), and View (Screen).

The main window is divided into several sections:

- Chip Name:** ALU, Time: 0
- Input pins:** A table with columns Name and Value. The values for x[16] and y[16] are circled in blue.
- Output pins:** A table with columns Name and Value. The values for out[16], zr, and ng are circled in blue.
- HDL:** A text area containing Verilog code for the ALU.
- Logic Diagram:** A diagram showing the ALU block with D Input (-5242), M/A Input (6253), and ALU output (2052).

Annotations with yellow callouts provide additional information:

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

Input pins table:

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

Output pins table:

Name	Value
out[16]	0000100000000100
zr	0
ng	0

HDL code:

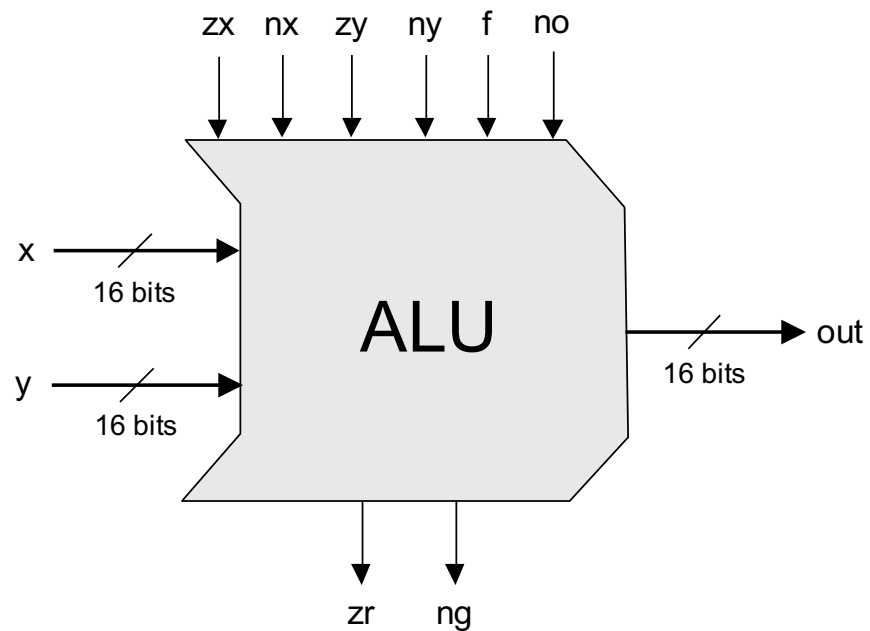
```
// This file is part of the material for the book "The Elements of Computing Systems" by Noam Nisan and Shimon Schocken. MIT Press. Book site: www.nand2tetris.org. File name: tools/builtIn/ALU.  
  
/**  
 * The ALU. Computes a pre-defined operation on two 16-bit integers x and y.  
 * where x and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

Logic Diagram:

ALU block with D Input: -5242, M/A Input: 6253, and ALU output: 2052. The block is labeled D&M.

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	0	0	0	0	-x
1	1	0	0	0	0	-y
0	1	0	0	0	0	x+1
1	1	0	0	0	0	y+1
0	0	0	1	0	0	x-1
1	1	0	1	0	0	y-1
0	0	0	1	0	0	x+y
0	1	0	1	0	0	x-y
0	0	0	0	0	0	y-x
0	0	0	0	0	0	x&y
0	1	0	0	0	0	x y

Example: compute !x

x: 1 1 0 0

y: 1 0 1 1 (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	1	0	1
0	0	1	1	1	1	0
1	1	0	0	1	0	0
0	0	1	1	1	1	1
1	1	0	0	1	0	0
0	0	1	1	1	1	1
1	1	0	0	1	0	0
0	1	1	1	1	1	1
1	1	0	1	1	0	1
0	0	1	1	1	1	1
1	1	0	0	1	0	1
0	0	0	0	0	1	1
0	1	0	0	1	1	y
0	0	0	1	1	1	x-y
0	0	0	0	0	0	y-x
0	1	0	1	0	1	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)

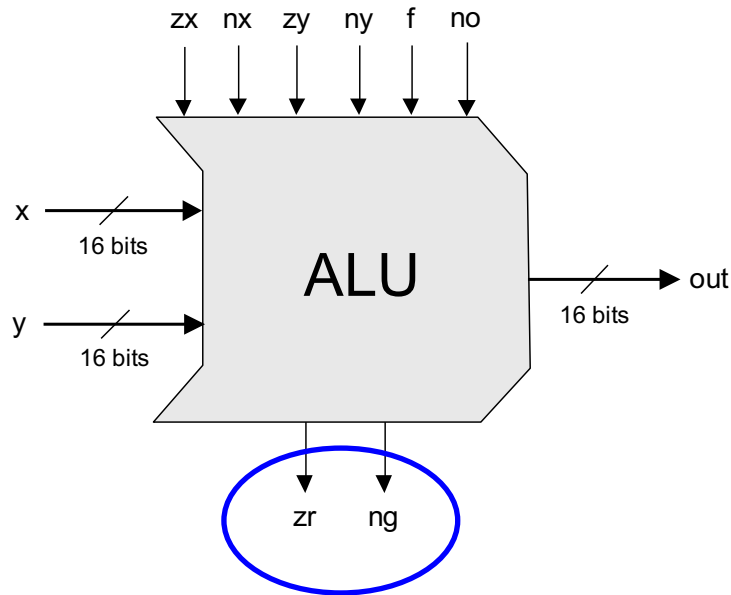
Practice:

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

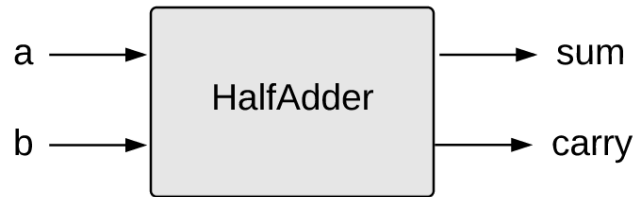
Project 2

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

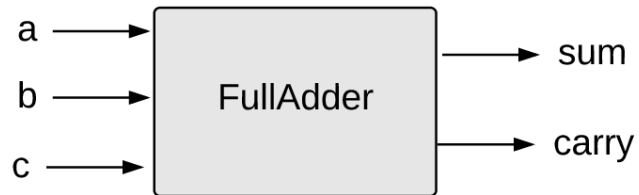
HalfAdder.hdl

```
/** 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

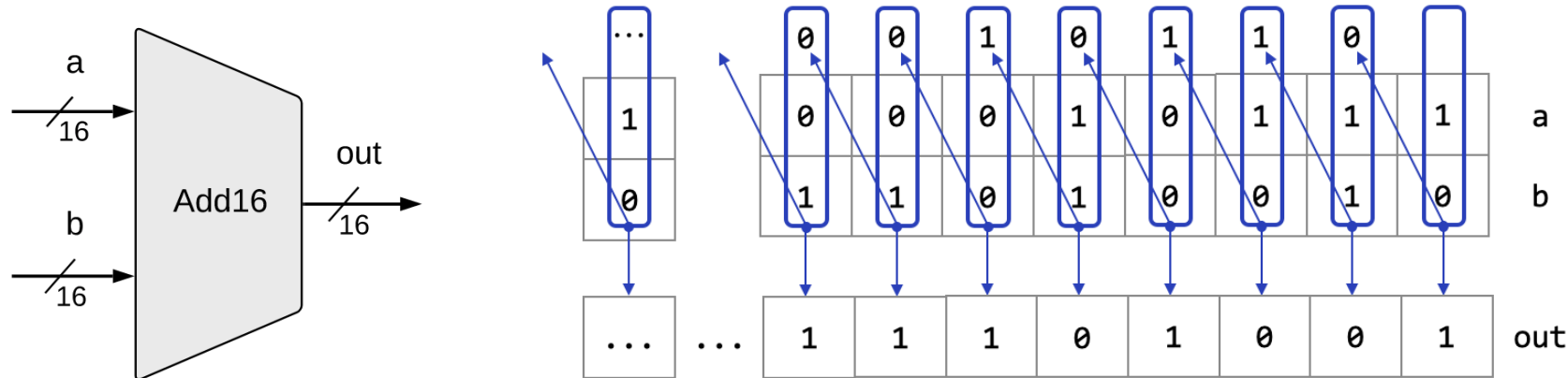
FullAdder.hdl

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

Implementation tip

Can be built from two half-adders.

16-bit adder



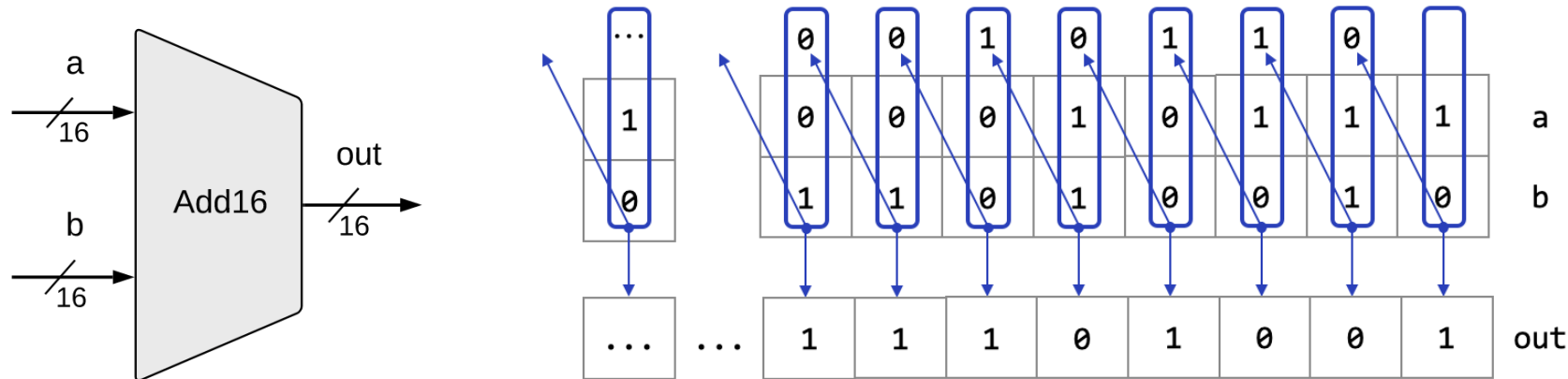
Add16.hdl

```
/* 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 your 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



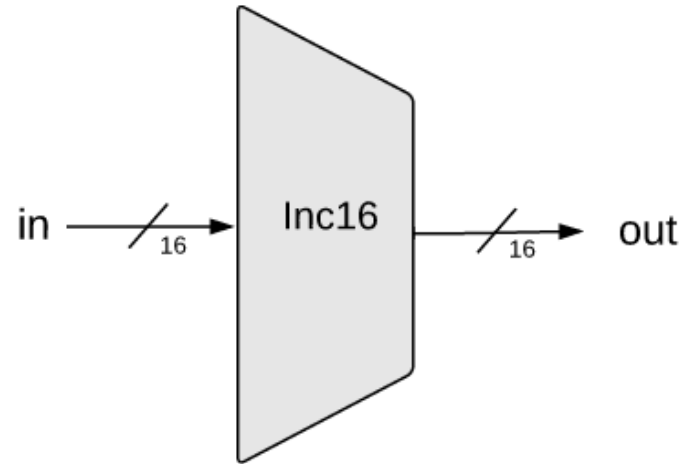
Add16.hdl

```
/* 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:
}
```

Implementation tip

To set a pin x to 0 (or 1) in HDL,
use: $x = \text{false}$ (or $x = \text{true}$)

16-bit incrementor



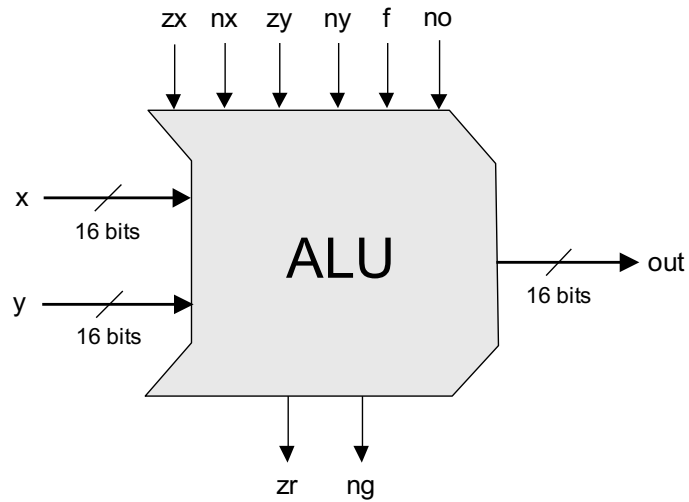
Inc16.hdl

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

Implementation tip

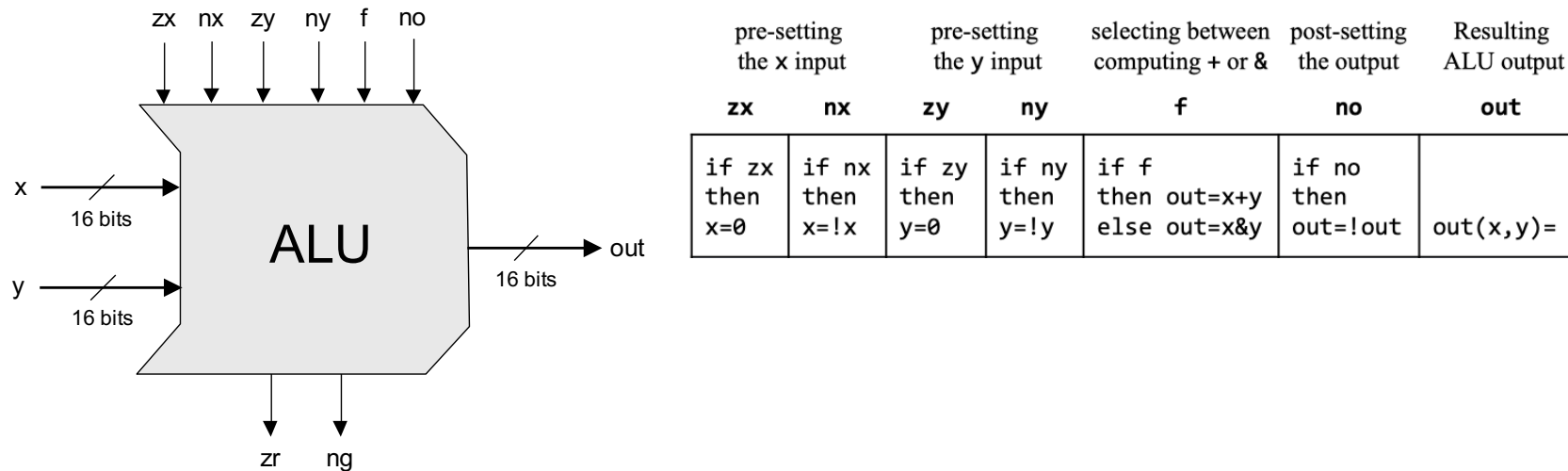
To set a bus-subset $x[i..j]$ to $00\dots0$ (or to $11\dots1$) in HDL,
use: $x[i..j] = \text{false}$ (or $x[i..j] = \text{true}$)

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



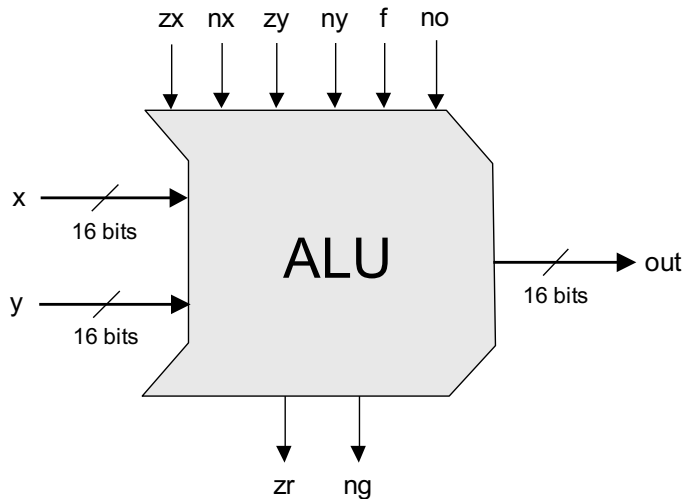
ALU.hdl

```

/** 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 And on two 16-bit values

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:

```
chipPart(..., x=true, y=false, z[0..2]=true, z[6..7]=true);
```

...

We can assign values to sub-buses

	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 */
```

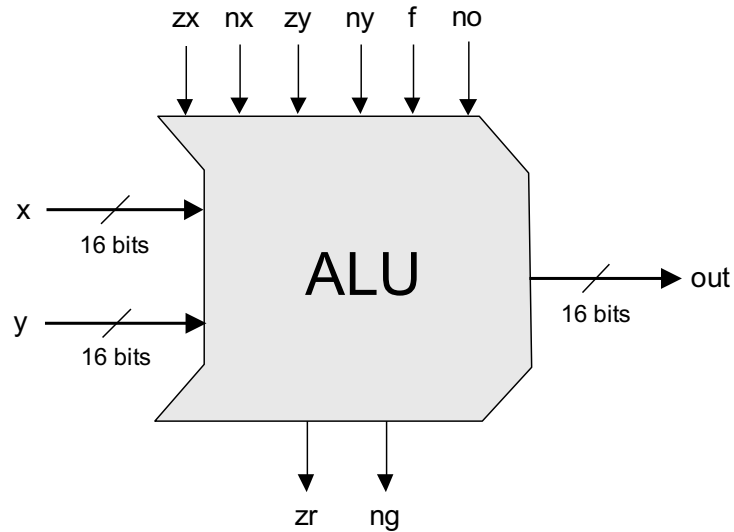
```
CHIP Add16 {  
  IN a[16], b[16];  
  OUT out[16];  
  
  PARTS:  
  ...  
}
```

```
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

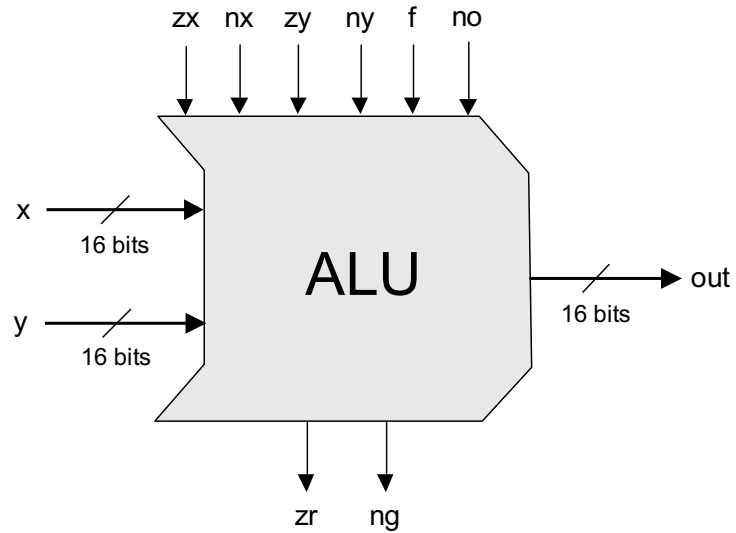
To implement the ALU logic:

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

Best practice advice (same as project 1)

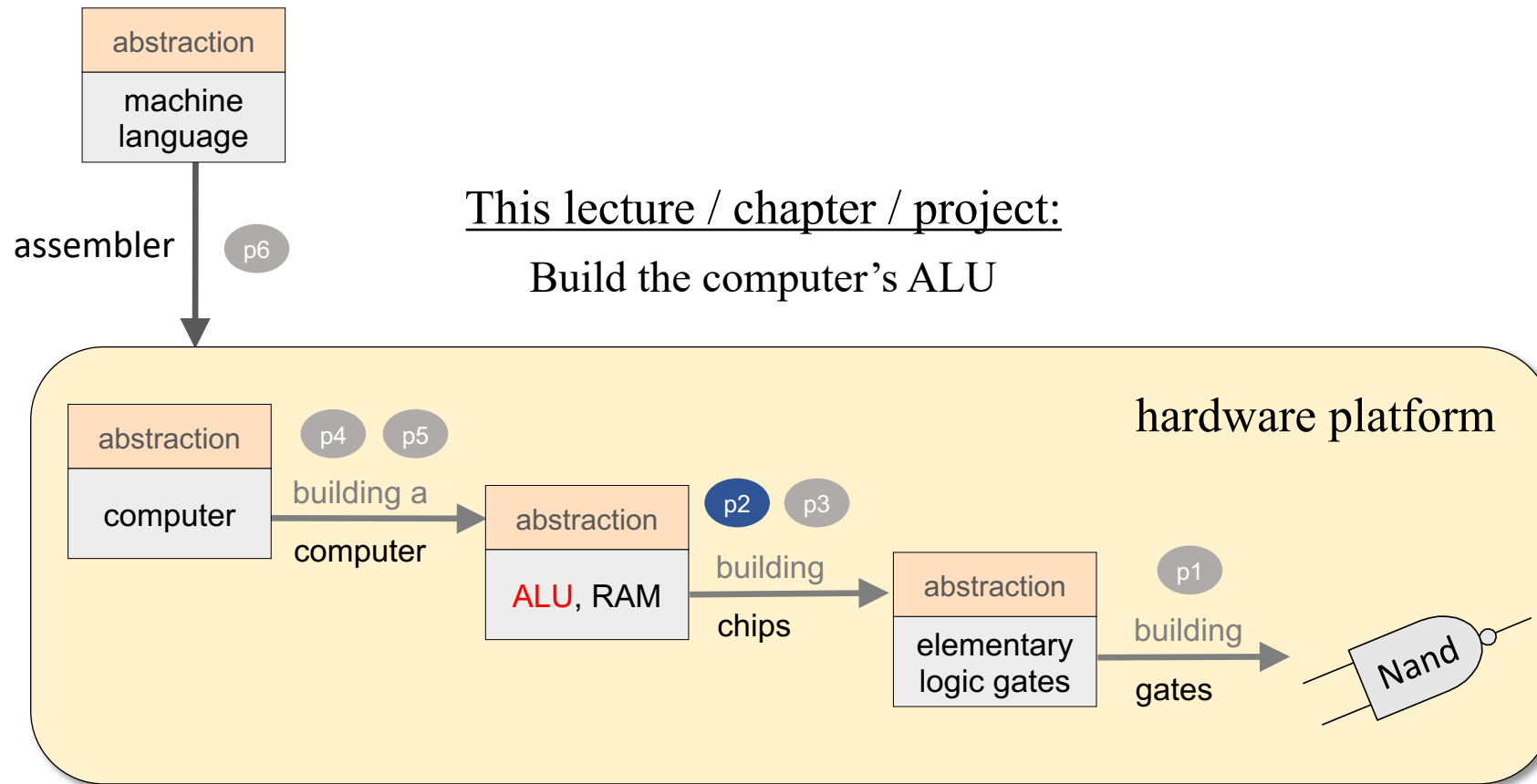
- Implement the chips in the order in which they appear in the project guidelines
- If you don't implement some chips, you can still use their built-in implementations
- No need for “helper chips”: Implement / use only the chips we specified
- In each chip definition, strive to use as few chip-parts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

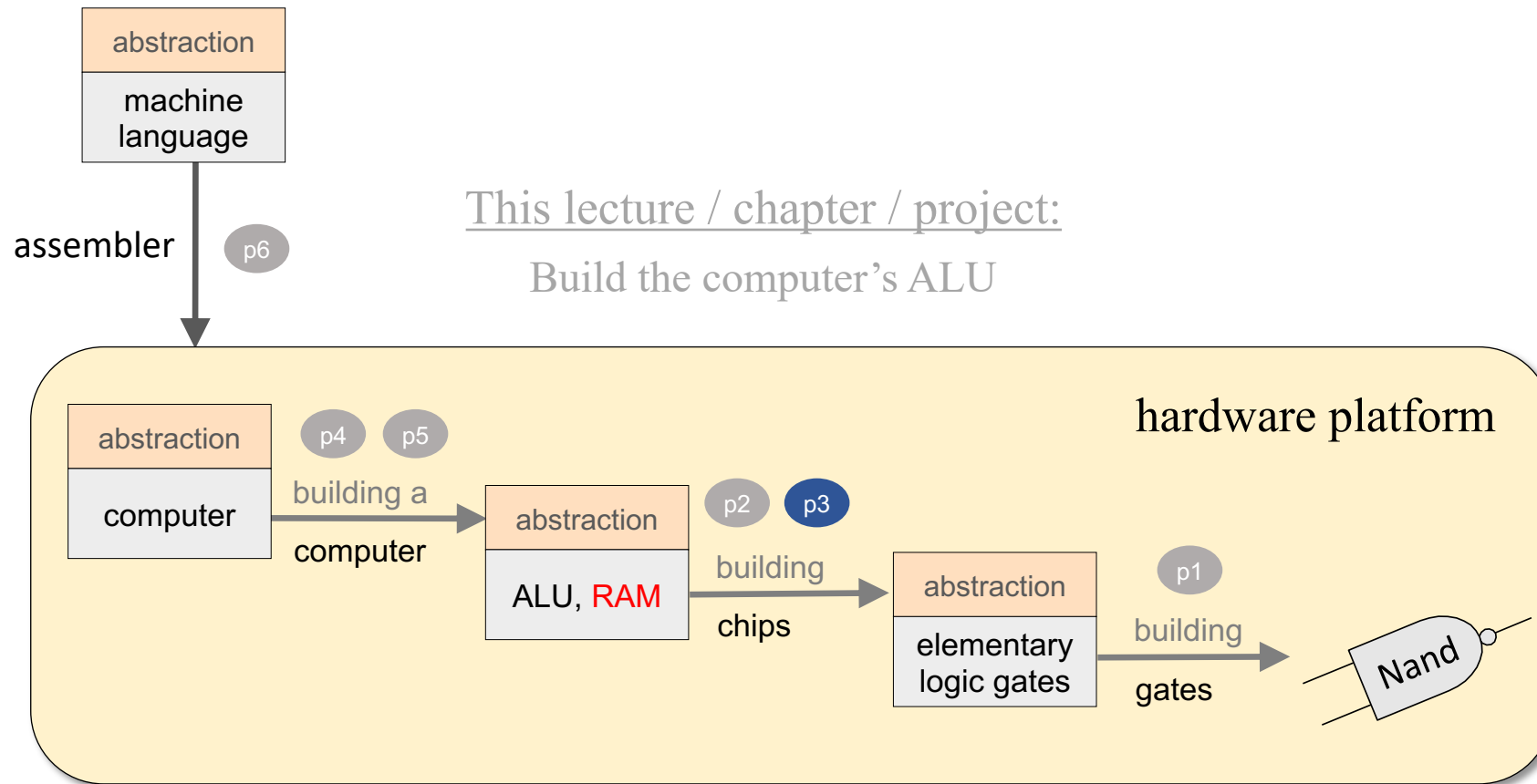
That's It!
Go Do Project 2!

What's next?



This lecture / chapter / project:
Build the computer's ALU

What's next?



Next lecture / chapter / project:

Build the computer's RAM