Princípios da Computação

Boolean algebra

Instituto Superior de Engenharia do Porto

Boolean algebra

- Boolean algebra was introduced by George Boole in the 19th century, and is the branch of algebra that establishes the **logic operations** upon **truth values**:
    - False : 0
    - True : 1
- A **boolean variable** can assume only these two values.
- A **logic sentence** is an algebraic expression where logic operations establish a logic relationship between boolean variables and/or constants.
    - The evaluation of a logic sentence produces a truth value.

Boolean operations

Negation (NOT)

- The **negation** is a unary operation where the result is the complement of the input value.
- This is a basic operation.

A	$\bar{A}$
0	**1**
1	**0**

Conjunction (AND)

- The **conjunction** is a binary operation where the result is true only if both input values are true.
- This is a basic operation.

A	B	A . B
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

Disjunction (OR)

- The **disjunction** is a binary operation where the result is false only if both input values are false.
- This is a basic operation.

A	B	A + B
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

Exclusive or (XOR)

- The **exclusive or** is a binary operation, where the result is false only if both input values are equal.
- This is a secondary operation, as it can be composed by basic operations!

A	B	$A \oplus B$
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

isep Instituto Superior de Engenharia do Porto

Operators precedence

- Boolean algebra does not set any precedence between operations AND, OR and XOR.
    - As such, parenthesis should be used to eliminate any ambiguity!
- Be aware that things can get different when using a programming language:
    - SmallTalk evaluates strictly from left-to-right.
    - C defines the order: (1) NOT, (2) AND, (3) XOR, (4) OR.

Bitwise operations

Bitwise logical operations

- Processors are designed to operate on groups of bits of a specific size, known as ***words***.
    - A 64-bit processor is designed to operate one or two 64-bits operands in one single instruction.
    - A 32-bit processor is designed to operate one or two 32-bits operands in one single instruction.
- This means that a logical operation on an ***n***-bit processor will result, in fact, in ***n*** parallel logical operations!

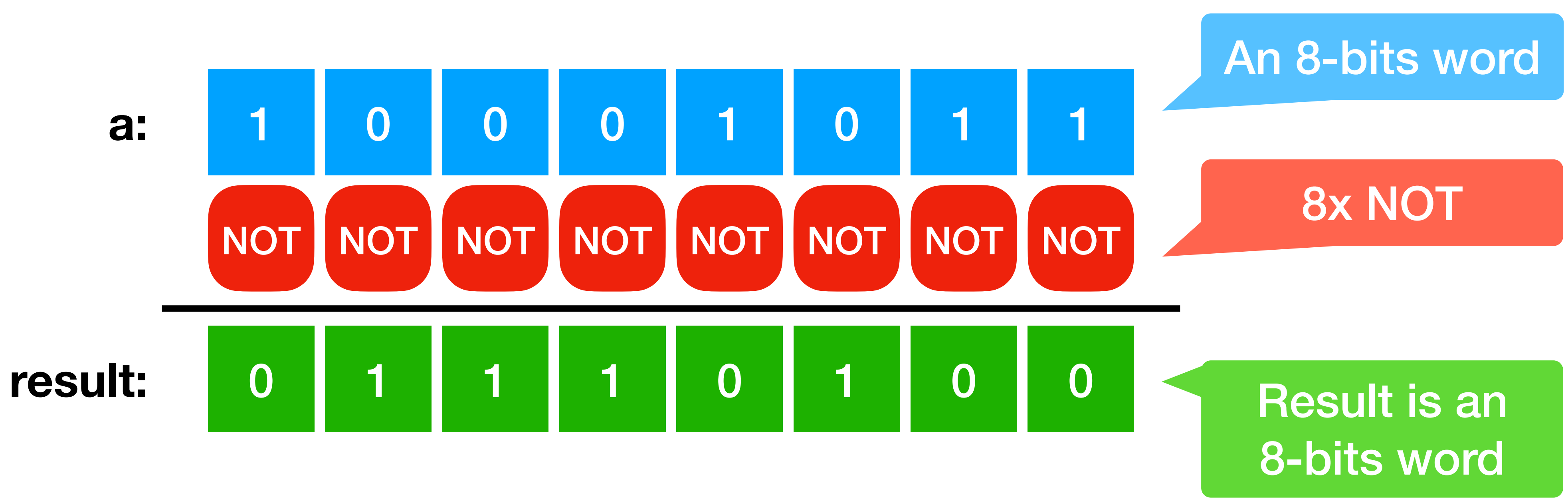
Bitwise operations (an example)

- Assume an 8-bit processor.
- Assume unary logical operation: **NOT a**

a:	1	0	0	0	1	0	1	1
	NOT	NOT	NOT	NOT	NOT	NOT	NOT	NOT
result:	0	1	1	1	0	1	0	0

Bitwise operations (an example)

- Assume an 8-bit processor.
- Assume unary logical operation: NOT a



isep Instituto Superior de Engenharia do Porto

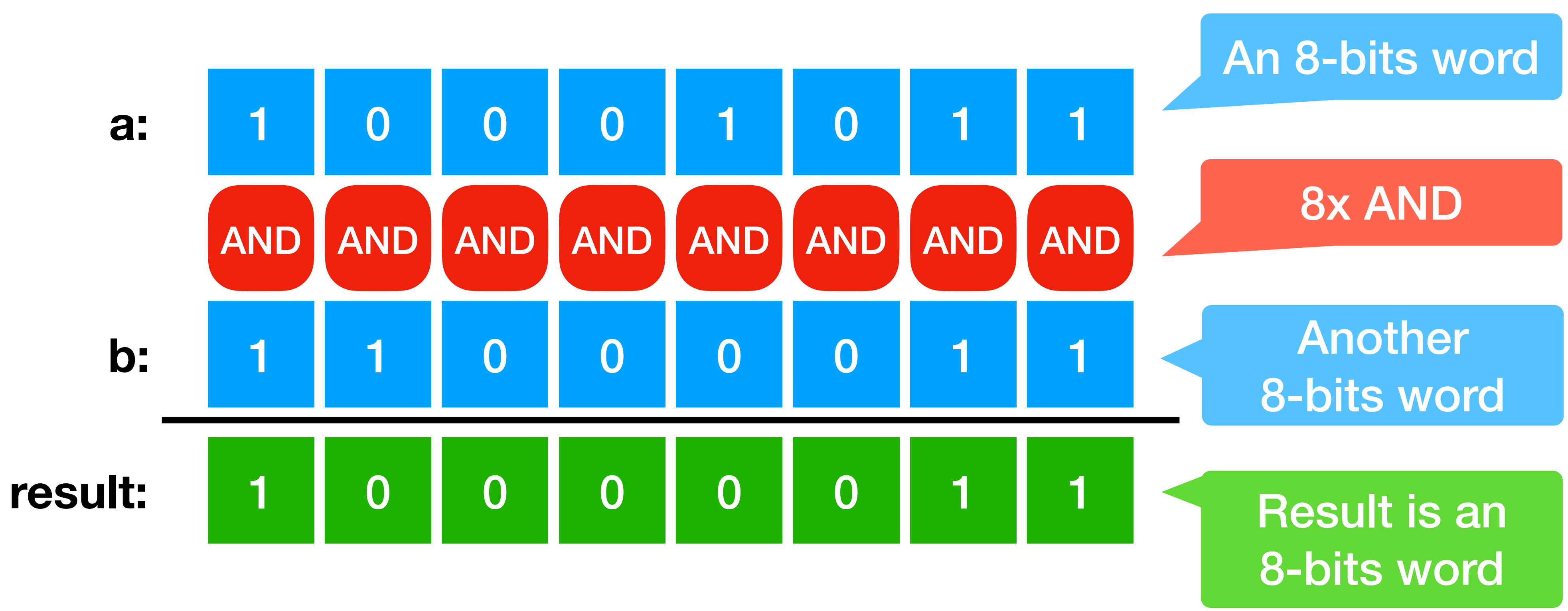
Bitwise operations (an example)

- Assume an 8-bit processor.
- Assume binary logical operation: **a AND b**

a:	1	0	0	0	1	0	1	1
	AND	AND	AND	AND	AND	AND	AND	AND
b:	1	1	0	0	0	0	1	1
result:	1	0	0	0	0	0	1	1

Bitwise operations (an example)

- Assume an 8-bit processor.
- Assume binary logical operation: a AND b



isep Instituto Superior de Engenharia do Porto

Logical masking

Logical masking

- Processors perform logical operations on fixed-size bit strings.

- This ability is quite useful to transform a word in selected bits, by using:

  - a **bit mask**, and

  - an appropriate **bitwise operation**.

Masking bits to 1 ("setting" bits)

- Bitmask
  - **1** : where the bits should be 1,
  - **0** : where the bits should stay unchanged.
- Bitwise operation : **OR**

Masking bits to 1 (example)

- Let us assume an arbitrary 8-bit string X.
- Turn the 4 most significant bits (i.e. bits 7 to 4) to 1, leaving the 4 least significant bits (i.e. bits 3 to 0) unchanged.
- Solution:
  - Bitmask : 11110000
  - Bitwise operation : OR

Masking bits to 1 (example)

- Let us assume that X = 10101010, then…

```
       X: 10101010
 Bitmask: 11110000

  Result: 11111010
```

Masking bits to 0 ("clearing" bits)

- Bitmask
  - **0** : where the bits should be 0,
  - **1** : where the bits should stay unchanged.
- Bitwise operation : **AND**

Masking bits to 0 (example)

- Let us assume an arbitrary 8-bit string X.
- Turn the 4 most significant bits (i.e. bits 7 to 4) to 0, leaving the 4 least significant bits (i.e. bits 3 to 0) unchanged.
- Solution:
  - Bitmask : 00001111
  - Bitwise operation : AND

Masking bits to 0 (example)

- Let us assume that X = 10101010, then…

```
      X: 10101010
Bitmask: 00001111

 Result: 00001010
```

Flipping bit values

- Bitmask
  - **1** : where the bits should be flipped,
  - **0** : where the bits should remain unchanged.
- Bitwise operation : **XOR**

Flipping bit values (example)

- Let us assume an arbitrary 8-bit string X.
- Flip the 4 higher bits (i.e. bits 7 to 4), leaving the 4 lower bits (i.e. bits 3 to 0) unchanged.
- Solution:
  - Bitmask : 11110000
  - Bitwise operation : XOR

Toggling bit values (example)

- Let us assume that X = 10101010, then…

```
       X: 10101010
 Bitmask: 11110000

  Result: 01011010
```

Comparing two numbers

- Bitmask : the number to be compared with.
- Bitwise operation : **XOR**
- If both numbers are equal, the result is **zero**.

Comparing two numbers (example)

- Let us assume an arbitrary 8-bit string X.
- Determine if X equals 10101010
- Solution:
    - Bitmask : 10101010
    - Bitwise operation : XOR

Comparing two numbers (example)

- Let us assume that X = 10101010, then…

```
      X: 10101010
Bitmask: 10101010

 Result: 00000000
```

Bit shifts

Bit shifts

- Processors can move the bits stored in a register either to the left or the right.
- The size of the displacement is usually an operand of the operation:
  - "*shift left 3 bits*"
  - "*shift right 5 bits*"

Bit shifts

- When the word in a register is shifted *N* bits…
  - … *N* bits are shifted out of the register on one side, and
  - … *N* bits are shifted in to the register on the other side.

```
0 1 1 0 1 1 0 0        "shift right 2 bits"

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

Bits "shifted in" **Bits "shifted out"**

Logical Left Shift (SLL)

- This operation shifts the word in the register *N* bits to the left.
- Shifts in ZEROS on the right side.
- Equivalent to multiply original word by 2^N.

```
0 0 0 0 1 1 0 0   SLL  2

0 0 1 1 0 0 0 0
```

Zeros "shifted in"

Logical Right Shift (SRL)

- This operation shifts the word in the register *N* bits to the right.

- Shifts in ZEROS on the left side.

```
1 1 0 0 1 1 0 0   SRL  2

0 0 1 1 0 0 1 1
```

Zeros "shifted in"

Arithmetic Right Shift (SRA)

- This operation shifts the word in the register *N* bits to the right.
- Shifts in copies of the most significant digit on the left side.
  - Thus, it maintains the sign of numbers in two's complement.

```
1 1 0 0 1 1 0 0   SRA  2

1 1 1 1 0 0 1 1
```

Ones "shifted in", because the most significant digit is 1.

Arithmetic Right Shift (SRA)

- Right shifting *N* bits is equivalent to divide by 2^N…
  - … but only for positive numbers!!!!
  - **Correct** for even negative numbers.
  - **Incorrect** for odd negative numbers!
    - The operation result is -1 from the correct result.
    - Example: **`–25 SRA 2`** results in -13; the correct result is -12.

Exercises

Exercises

- Assume the 8-bit variable `x` with initial value `10001011`.
- Select the appropriate bitwise operator and bit mask, for the following sequence of operations. Determine the end result after all operations are performed.

    1. Set the 3 least significant bits to zero.
    2. Set the 2 most significant bits to one.
    3. Flip bits 3 and 5.

Solution:

- **x:** `10001011 (initial value).`

1. Set the 3 least significant bits to zero: bitwise operator: `AND` — bitmask: `11111000`
    - Result: `10001011 AND 11111000 = 10001000`
2. Set the 2 most significant bits to one: bitwise operator: `OR` — bitmask: `11000000`
    - Result: `10001000 OR 11000000 = 11001000`
3. Flip bits 3 and 5: bitwise operator: `XOR` — bitmask: `00101000`
    - Result: `11001000 XOR 00101000 = 11100000`

isep Instituto Superior de Engenharia do Porto