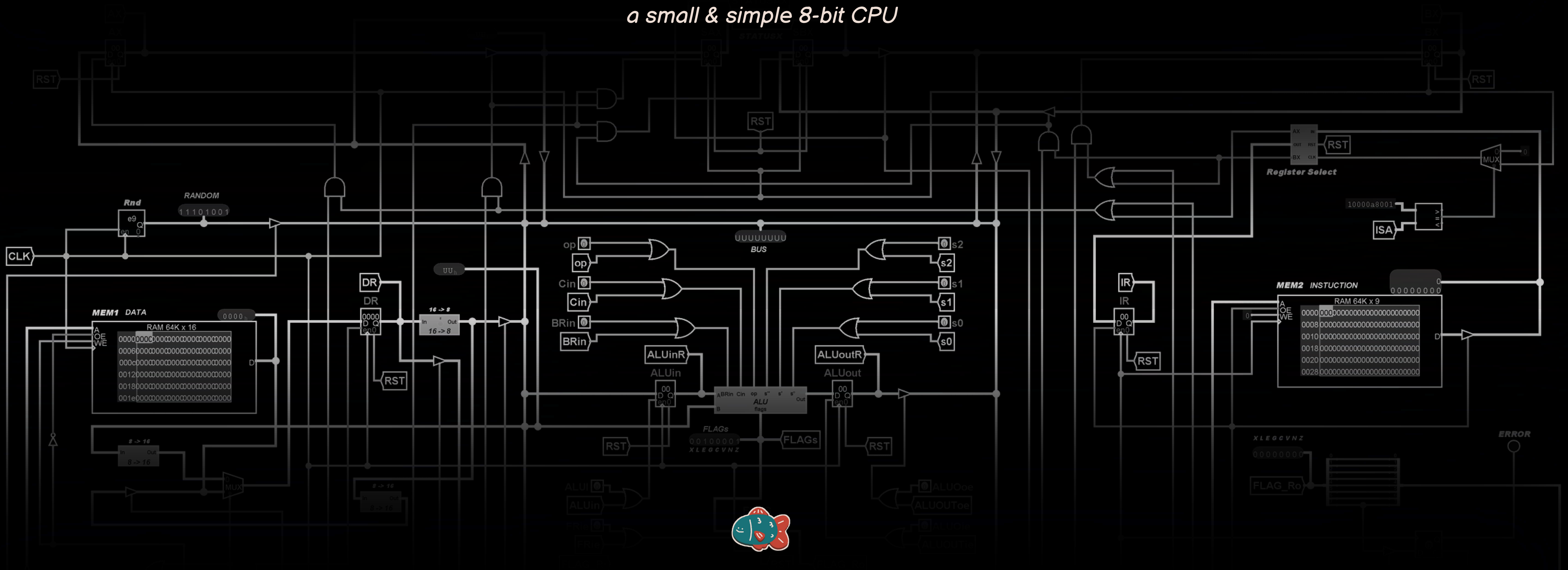
FLiP01
USER MANUAL

a small & simple 8-bit CPU



Pescetti Studio

A manual by Biasolo Riccardo and Croci Lorenzo

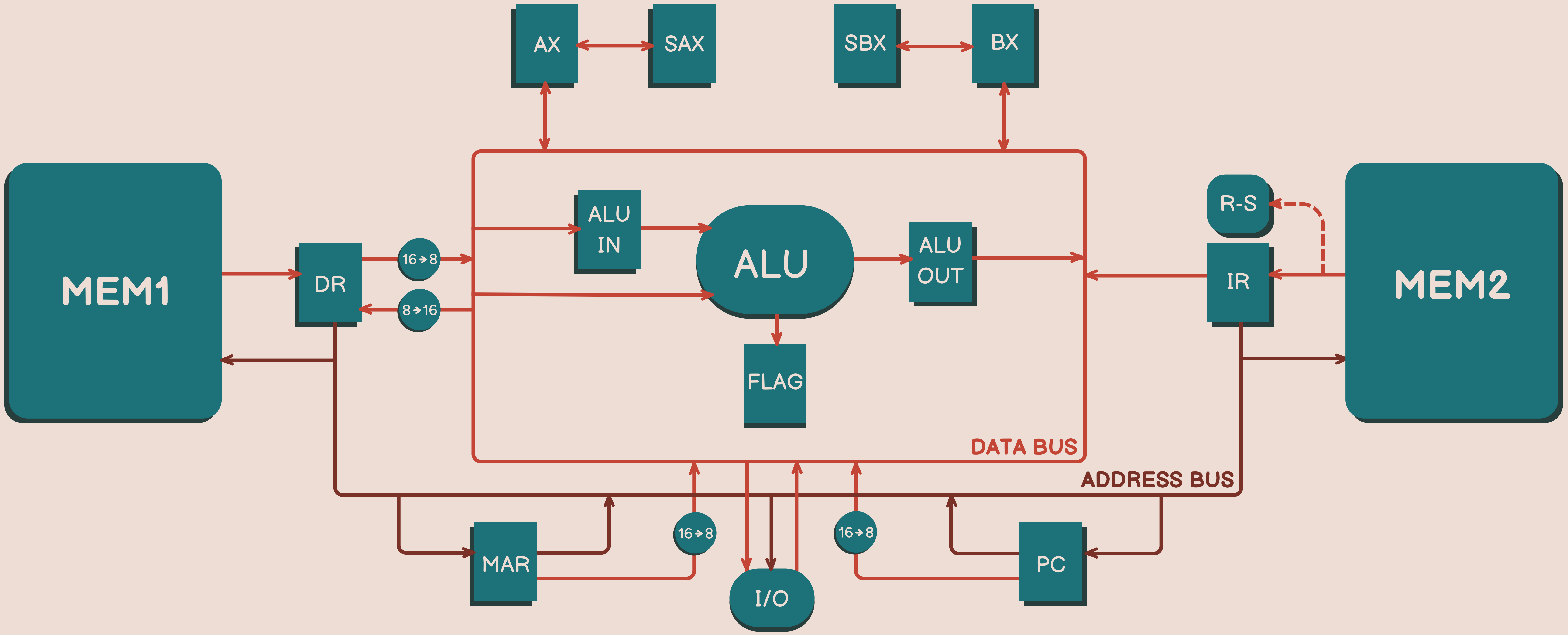
This is an extremely simplified manual designed to make the project accessible to as many people as possible.
To reach such a level of abstraction without getting bogged down in technical details, we've used language that is not always technically precise, though its correctness has been maintained.

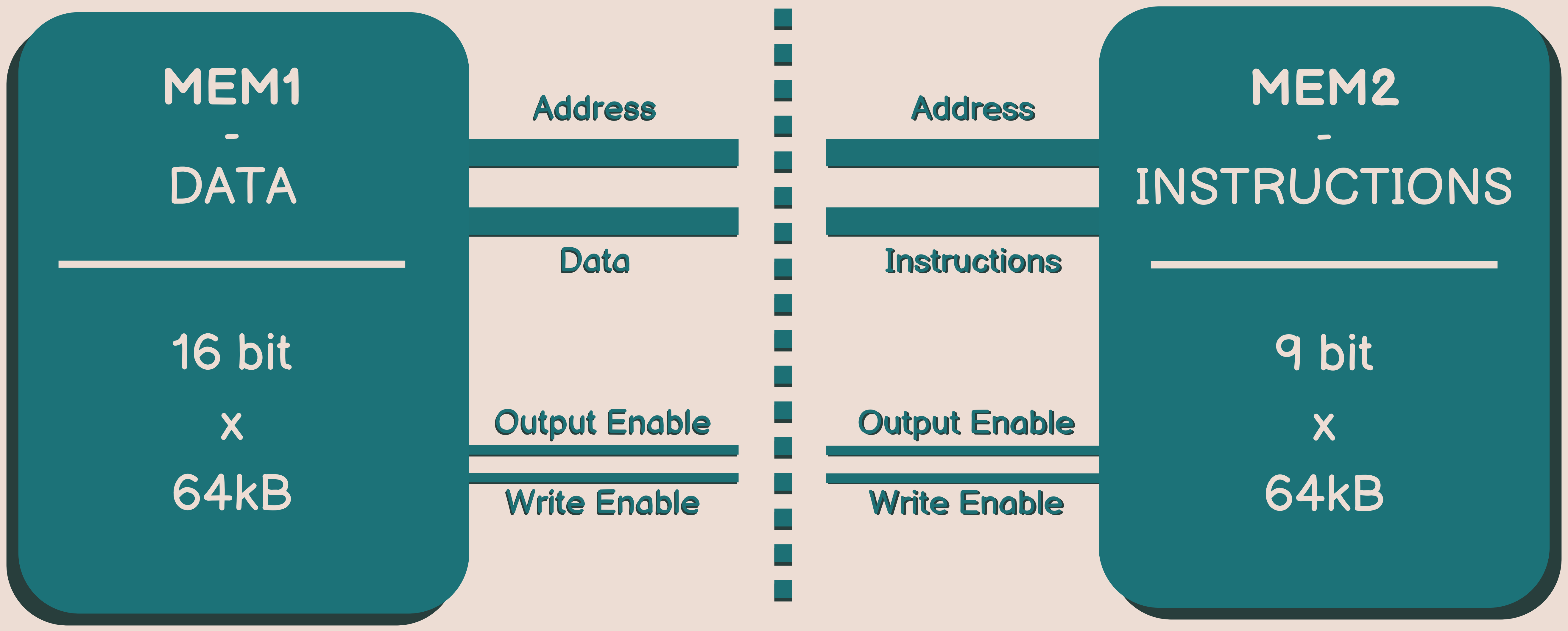
And no, even though it looks similar, the font used is not Comic Sans. I swear.

Index

Index

Flip01, short for *First Level Instructional Processor*, is a small **8-bit CPU** with a **16-bit address** bus.
This means it can handle data ranging from 0 to $2^8 - 1$ (255) and can store up to 2^{16}(65536 bit ~ 64 KB) different pieces of data.
Data and addresses travel on two separate connection networks called the **data bus** and **address bus**, respectively.





Flip01 is based on a **Harvard architecture**, meaning that data and instructions (the actions the processor must execute) are stored in two separate memories: **MEM1 for data** and **MEM2 for instructions**.
MEM1 is an **8-bit memory** with a capacity of 64 KB, while MEM2 is a **9-bit memory** with a capacity of 64 KB.
To move, temporarily store, and possibly modify these data, a component called register is used.

Registers

In very simple terms, a register can be described as a standalone memory cell with **fast read and write capabilities**.
In Flip01, there are **10 main registers** (*+ 1: spoiler*), and their size and connection (data or address bus) are defined by their **specific function**.

AX
8 bit

This is the first of two **general-purpose registers**. This means it has no specific function within the processor but is used to store data for calculations.
Since it only handles data, it is 8 bits in size and is connected solely to the **data bus**.

BX
8 bit

The second and final **general-purpose register**, with the same properties as AX.
Like AX, it is 8 bits and connected only to the **data bus**.

SAX
8 bit

(STATUSAX)

An 8-bit register designed to **save the contents of AX** when needed.
It can only communicate with AX, so it is not directly connected to either the data bus or the address bus.

SBX
8 bit

(STATUSBX)

A mirror register to STATUSAX.
It is also an 8-bit register but **saves the contents of BX** when required.
Like SAX, it is not connected directly to the data or address buses.

DR

16 bit

DATA REGISTER

This register **communicates directly with the memory holding data** (MEM1).
Since it handles both data and addresses, it is a 16-bit register connected to **both the data and address buses**.

It uses converters to adapt the size before input and output:

- On **input**, data is converted **from 8 to 16 bits** by padding with zeros.
- On **output**, data is converted back **from 16 to 8** bits by discarding the first 8 bits.

If any discarded bits contain data, an **error LED** will light up as a preventive measure.

IR

8 bit

INSTRUCTION REGISTER

A mirror register to DR, but it **interfaces with the memory that holds instructions** (MEM2).

Instructions in memory are 9 bits long, divided as follows:

- The first bit specifies which general-purpose register (AX or BX) the instruction refers to: **1 for AX, 0 for BX**.
- The remaining 8 bits contain the **actual instruction**.
  Only the 8 bits representing the instruction are loaded into the IR, so it is an 8-bit register.

ALU IN

8 bit

Inside the processor, there is a unit responsible for performing calculations, called the ALU.
The ALUIN register **takes and stores the first of the two operands for the ALU**.
This is necessary because both operands cannot enter at the same time (they would conflict on the data bus).
This register **holds the first operand** until the second arrives.
The second operand doesn't need to be stored since the ALU processes it immediately upon arrival.
This register is also 8 bits, as it only handles data.

ALU OUT

8 bit

This register **stores the result of the ALU operation**, then passes it to one of the general-purpose registers if needed for future operations.
ALUOUT is also an 8-bit register.

MAR
....
16 bit

MEMORY ADDRESS REGISTER

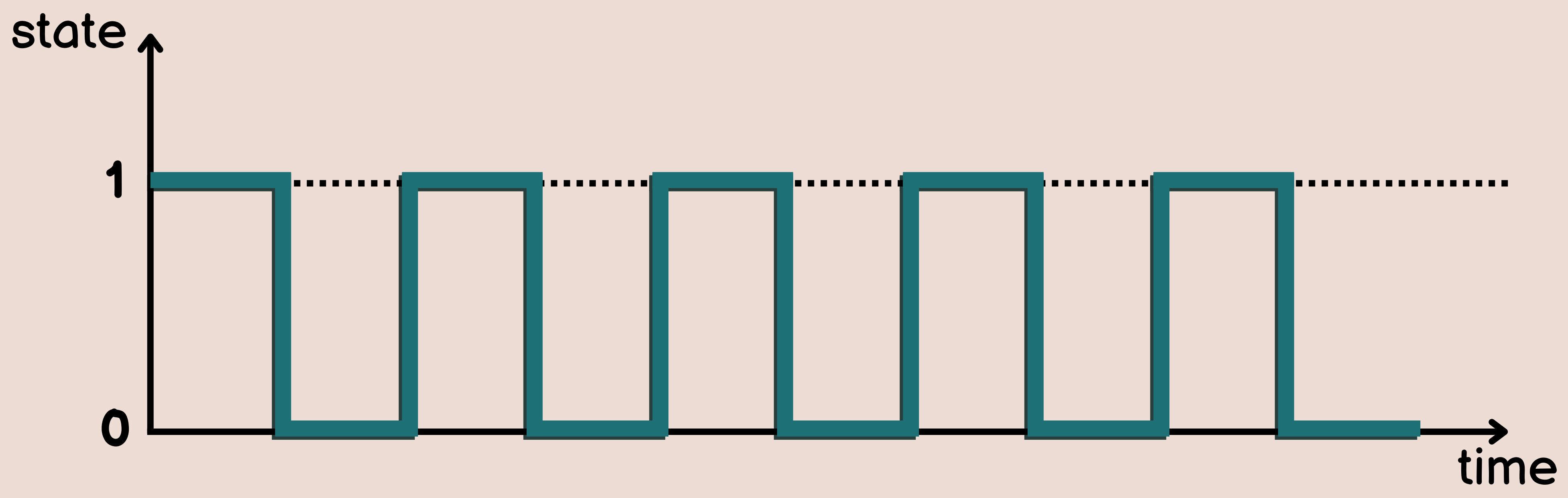
This register **stores the memory address of the data to be read or rewritten**.
It is useful when an instruction refers to reading data from a specific memory location, which is different from the instruction's own location.
MAR helps **retrieve the address of the data, locate it in memory, and either read its value or overwrite it with a new one**.
Since it deals only with addresses, it is a 16-bit register connected to the address bus both for input and output.
It is also connected to the data bus for output, with a **16-to-8-bit converter** (similar to the one in DR) that triggers an error LED if data is lost during conversion.

PC
....
16 bit

PROGRAM COUNTER

This register **holds the address of the next instruction to be executed**.
It ensures that, at each step of the program, the processor knows exactly where to read the next instruction.
This is particularly useful when the execution flow is non-linear.
(foreshadowing jump instructions????)
Like MAR, the PC is a 16-bit register connected to the address bus for both input and output and to the data bus for output, with an **intermediary converter**.

All registers update when there is a state change in a common signal that goes to all components that require it. This signal is called **the clock**, and ideally, it looks like this:



It **oscillates between 0 and 1**, and the instant change from 0 to 1 **updates the values stored in the registers** based on the data circulating in the bus at that exact moment.

If registers are used to temporarily store data, as previously mentioned, another fundamental component called the **ALU** (*Arithmetic Logic Unit*) is used to modify it.

ALU

Every processor has a different ALU in terms of **calculation power** or **data handling capabilities**.
The ALU of Flip01 is designed to take **two 8-bit inputs** and produce a single output of the same size.

Based on the configuration and the combination of **6 binary control signals** (0 or 1), the ALU can perform **7 main operations** and **4 derived ones**.
The 6 control signals are as follows:

BRin Sets the second input of the **ALU to 0**

s0 **s1** **s2** The unique combinations of these signals determine **which of the 7 main operations will be executed**

Cin **op** These signals control **additional bits** for operations that require them

The 7 main operations:

ADD

When given two inputs, A and B, this configuration **outputs the sum of the two**:

A + B

Control signals:

BRin	Cin	op	s0	s1	s2
0	0	0	0	0	0

SUB

With the same inputs, A and B, this configuration **outputs the difference**:

A - B

Control signals:

BRin	Cin	op	s0	s1	s2
0	1	1	0	0	0

AND

For the same inputs, this configuration performs a **bitwise AND operation on A and B**:

A AND B

Following the logical structure of a single-bit AND operator:

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

For 8-bit inputs, the result would be, for example:

01001100
10011101 **AND**

00001100

11010011
01110110 **AND**

01010010

Control signals:

BRin	Cin	op	s0	s1	s2
0	0	0	1	0	0

OR

For the same inputs, this configuration performs a **bitwise OR operation on A and B**:

A OR B

Following the logical structure of a single-bit OR operator:

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

For 8-bit inputs, the result would be, for example:

```
10110100
00110101   OR
-------------
10110101
```

```
00100101
01000100   OR
-------------
01100101
```

Control signals:

BRin	Cin	op	s0	s1	s2
0	0	0	0	1	0

NOT

In this case, only one operand is considered as input (A).
This configuration outputs the **bitwise inverse of the input operand**:

NOT A

Following the logical structure of a NOT operator:

X	NOT X
0	1
1	0

For 8-bit inputs, the result would be, for example:

```
01100100   NOT
___________________
10011011
```

```
01100100   NOT
___________________
10011011
```

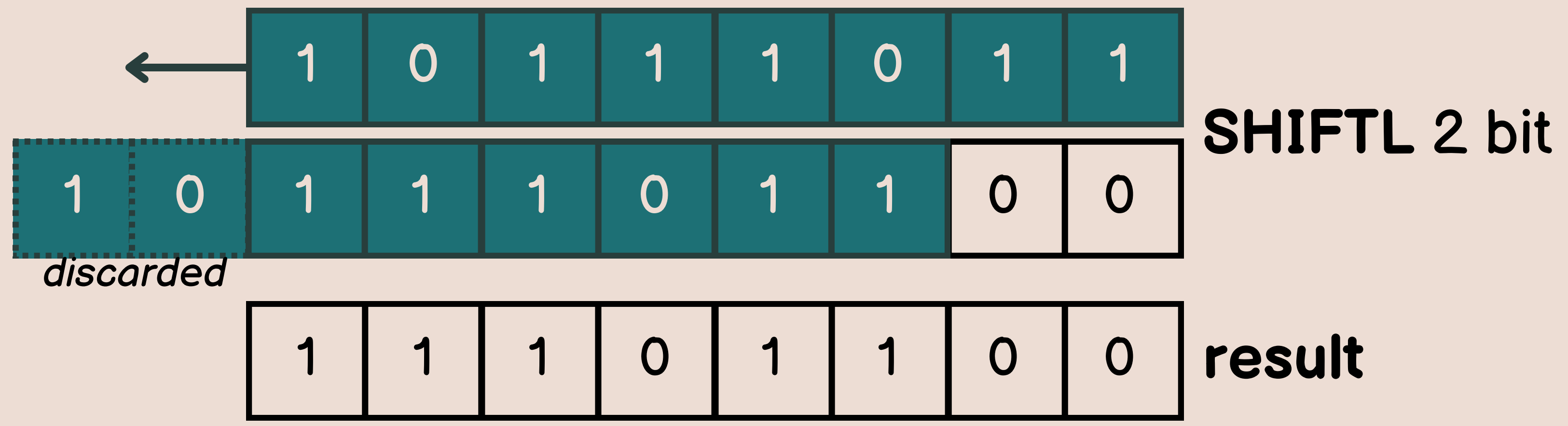
Control signals:

BRin	Cin	op	s0	s1	s2
0	0	1	0	0	0

SHIFTL

This configuration **shifts the bits of operand A to the left by B positions**, keeping the result 8 bits long. The vacated positions on the right are filled with 0 padding.

For example:



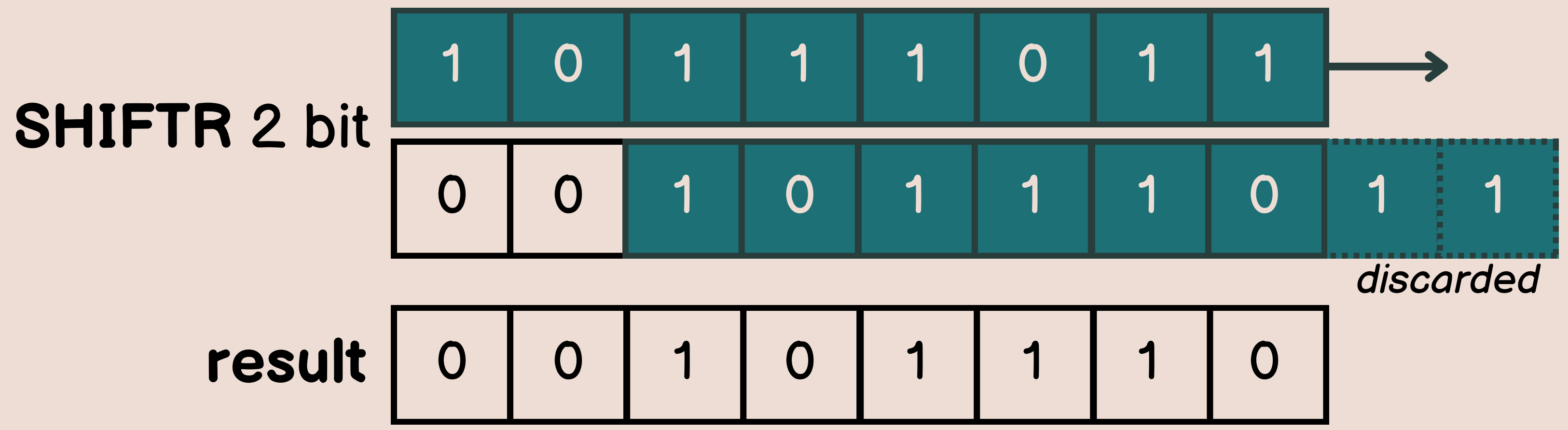
Control signals:

BRin	Cin	op	s0	s1	s2
0	1	1	1	1	0

SHIFTR

This configuration **shifts the bits of operand A to the right by B positions**, keeping the result 8 bits long. The vacated positions on the left are filled with 0 padding, regardless of the leftmost bit (logical shift).

For example:



Control signals:

BRin	Cin	op	s0	s1	s2
0	0	0	0	0	1

By manipulating the ALU's inputs, you can derive **four additional operations**:

INC

The first input to the ALU remains unchanged, while the second is forced to 1.
Setting the ALU to the add configuration gives A + 1, **incrementing the value by one unit**.

Control signals:

BRin	Cin	op	s0	s1	s2
1	1	0	0	0	0

DEC

Similarly, the first input remains unchanged, while the second is forced to 1.
However, setting the ALU to the sub configuration gives A - 1, **decrementing the value by one unit**.

Control signals:

BRin	Cin	op	s0	s1	s2
0	0	1	0	0	0

0

Regardless of the input, this operation **outputs the value 0**.
Given input A, forcing the second input to 0 and performing an AND operation results in 0, no matter the value of A.

Control signals:

BRin	Cin	op	s0	s1	s2
1	1	0	0	0	0

A

Finally, you can **output the same value as the first input without modifying it**.
Given input A, forcing the second input to 0 and performing a sub operation results in A:

A - 0 = A.

Control signals:

BRin	Cin	op	s0	s1	s2
0	0	1	0	0	0

Connected to the ALU there is an **additional, critical register**:

FLAG Register

This is the third and final register that communicates with the ALU.
It's an 8-bit register where **each bit indicates a specific aspect of the most recent operation**:

Flags (1/2)

shift error

X

This bit is set if **the number of bits to be shifted exceeds 8**, surpassing the maximum length of the operand involved in the shift operation. It can be ignored if the executed operation is of a different type.

A < B

L

This bit is set if **operand A is less than operand B** in absolute value.

A = B

E

This bit is set if **operand A is equal to operand B** in absolute value.

A > B

G

This bit is set if **operand A is greater than operand B** in absolute value.

Flags (2/2)

overflow

V

This bit is set if **the result exceeds the 8-bit limit**, even though both operands are 8 bits. This indicates that an extra bit would have been required for the result to be correct.

carry

C

This bit is set if there was a carry during the operation, meaning **the most significant bit had to generate a carry to the previous bit**.

zero

Z

This bit is set if **the result of the operation is zero**.

negative

N

This bit is set if **the result of the operation is negative** for any reason.

Instructions

Instructions represent the full range of actions the processor can perform. Flip01 has **39 instructions**, each identified by a **unique operation code** (*op-code*) in hexadecimal format.

Instructions consist of **micro-instructions**, which are movements of data between registers or arithmetic and logical operations.

For example, the addR instruction, which adds the contents of the two general-purpose registers AX and BX, is made up of the following micro-instructions:

- **AX → ALUA** The content of the **AX register is moved to the ALUA input**, preparing it for the ALU operation.
- **ALUA + BX → ALUOUT** BX is provided as the second input to the ALU, allowing the sum to be calculated. **The result is stored in ALUOUT**.
- **ALUOUT → AX** **The result saved in ALUA is transferred back to the AX register**, replacing its previous content, so it can be used in subsequent instructions.

If two micro-instructions operate on different buses (data and address) and are sequential, **they can be executed in the same clock cycle**, as no data will conflict.

In MEM1 memory cells, it's possible to assign a **unique name to each cell,** allowing access to it during program execution through read and write actions.

To assign a name to a memory cell, the following syntax is used:

[name] = [value]

The = operator symbolizes an **assignment operation**.

For the remainder of the manual, this type of memory cell will be referred to as a

Variable

adopting standard terminology.

In textual form, this can be expressed as:

"The cell named [name] will initially contain the value [value] at the start of the program."

The **address of the memory cell**, or its location within MEM1, **is assigned procedurally**, typically as the first available cell after those reserved for program parameters.

The value indicated represents only the initial assignment to the cell, and to modify its content later, you must use the CPU instructions.

Currently, the instructions available on Flip01 can be grouped into **five categories**:

1. Direct addressing instructions
2. Single-operand instructions
3. Zero-operand instructions
4. Immediate instructions
5. Jump instructions

A long list of all possible instructions follows

Direct Addressing Instructions

These instructions consist of **two parameters**: the first is the **general-purpose register** (AX or BX) they refer to, and the second is the memory address containing the value to be considered.
In Flip01's high-level syntax, the memory address specified as the second parameter is associated with the **unique name of a variable**.

These instructions involve memory access for reading the data.

Instruction syntax: Instruction [register] [variable]

Direct Addressing Instruction

load

load [register] [variable]

This instruction copies the value of the variable specified in the *[variable]* parameter into the register indicated in the *[register]* parameter.

op-code

0x02

Microinstructions:

- MEM1 [variable] ➔ DR
- DR ➔ [register]

The **load** instruction requires 2 clock cycles to execute.

Usage example

```
var1 = 5
var2 = 3
load AX var1
load BX var2
...
```

store

store [register] [variable]

This instruction copies the value in the *[register]* parameter into the memory cell assigned to the variable specified in the *[variable]* parameter.

op-code

0x04

Microinstructions:

- [register] ➔ DR
- DR ➔ MEM1 [variable]

The **store** instruction requires 2 clock cycles to execute.

Usage example

```
var = 2
load AX var
...
store AX var
...
```

Direct Addressing Instruction

add

add [register] [variable]

This instruction adds the value associated with the *[variable]* parameter to the value stored in the *[register]* parameter. The result is saved in the specified register.

[register] = [register] + [variable]

op-code

0x06

Microinstructions:

- MEM1 [variable] ➔ DR
  [register] ➔ ALUA
- DR + ALUA ➔ ALUOUT
- ALUOUT ➔ [register]

The **add** instruction requires 3 clock cycles to execute.

Usage example

```
var1 = 10
var2 = 5
load AX var1
add AX var2
...
```

Direct Addressing Instruction

sub

sub [register] [variable]

This instruction subtracts the value associated with the *[variable]* parameter from the value stored in the *[register]* parameter. The result is saved in the specified register.

[register] = [register] - [variable]

op-code

0x09

Microinstructions:

- MEM1 [variable] → DR
  [register] → ALUA
- DR - ALUA → ALUOUT
- ALUOUT → [register]

The **sub** instruction requires 3 clock cycles to execute.

Usage example

```
var1 = 7
var2 = 2
load AX var1
sub AX var2
...
```

Direct Addressing Instruction

and

and [register] [variable]

This instruction performs a bitwise AND operation between the value stored in the *[register]* and the value associated with the *[variable]*. The result is saved in the specified register.

[register] = [register] AND [variable]

op-code

0x0C

Microinstructions:

- MEM1 [variable] → DR
  [register] → ALUA
- DR AND ALUA → ALUOUT
- ALUOUT → [register]

The **and** instruction requires 3 clock cycles to execute.

Usage example

```
var1 = 7
var2 = 2
load AX var1
and AX var2
...
```

Direct Addressing Instruction

or

or [register] [variable]

This instruction performs a bitwise OR operation between the value stored in the *[register]* and the value associated with the *[variable]*. The result is saved in the specified register.

[register] = [register] OR [variable]

op-code

0x0F

Microinstructions:

- MEM1 [variable] → DR
  [register] → ALUA
- DR OR ALUA → ALUOUT
- ALUOUT → [register]

The **or** instruction requires 3 clock cycles to execute.

Usage example

```
var1 = 7
var2 = 2
load BX var1
or BX var2
...
```

Direct Addressing Instruction

cmp

cmp [register] [variable]

The CMP instruction compares the value in the register specified in the *[register]* field with the value of the variable indicated in the *[variable]* field.
The flags are updated upon completion of this operation.

op-code

0x30

Microinstructions:

- MEM1 [variable] → DR
  [register] → ALUA
- ALUA - DR → ALUOUT

The **cmp** instruction requires 2 clock cycles to execute.

Usage example

```
var1 = 7
var2 = 2
load AX var1
cmp AX var2
...
```

Single-Operand Instructions

These instructions involve **only one parameter**,
usually the general-purpose register (AX or BX) they refer to.

Instruction syntax: Instruction [register]

not

not [register]

This instruction inverts the bits of the value stored in the *[register]*. The result is saved in the specified register.

[register] = NOT [register]

op-code

0x12

Microinstructions:

- DR → ALUA
- ALUA - [register] → ALUOUT
- ALUOUT → [register]

The **not** instruction requires 3 clock cycles to execute.

Usage example

```
var1 = 7
load AX var1
not AX
...
```

Single-Operand Instruction

neg

neg [register]

This instruction negates the value stored in the *[register]*. The result is saved in the specified register.

[register] = NEG [register]

op-code

0x15

Microinstructions:

- DR → ALUA
- ALUA - [register] + 1 → ALUOUT
- ALUOUT → [register]

The **neg** instruction requires 3 clock cycles to execute.

Usage example

```
var1 = 7
load BX var1
neg BX
...
```

Single-Operand Instruction

inc

inc [register]

This instruction increments the value stored in the *[register]* by 1. The result is saved in the specified register.

[register] = [register] + 1

op-code

0x24

Microinstructions:

- [register] → ALUA
- ALUA + 1 → ALUOUT
- ALUOUT → [register]

The **inc** instruction requires 3 clock cycles to execute.

Usage example

```
var1 = 8
load BX var1
inc BX
...
```

Single-Operand Instruction

dec

dec [register]

This instruction decrements the value stored in the *[register]* by 1. The result is saved in the specified register.

[register] = [register] - 1

op-code

0x27

Microinstructions:

- [register] → ALUA
- ALUA - 1 → ALUOUT
- ALUOUT → [register]

The **dec** instruction requires 3 clock cycles to execute.

Usage example

```
var = 1
load AX var
dec AX
...
```

rnd

rnd [register]

This instruction stores a random value between 0 and 255 in the *[register]*.

op-code

0x37

Microinstructions:

- RND ➔ [register]

The **save** instruction requires 1 clock cycle to execute.

Usage example

```
...
rnd AX
add AX var
...
```

save

save [register]

This instruction copies the value from the *[register]* into the corresponding status register.

op-code

0x5C

Microinstructions:

- [register] → STATUS[register]

The **save** instruction requires 1 clock cycle to execute.

Usage example

```
...
load AX var
...
save AX
...
```

read

read [register]

This instruction copies the value from the corresponding status register into the *[register]*.

op-code

0x5D

Microinstructions:

STATUS[register] ➔ [register]

The **read** instruction requires 1 clock cycle to execute.

Usage example

```
...
load AX var
save AX
...
read AX
...
```

Zero-Operand Instructions

These instructions **do not take any parameters**.
They primarily act on the processor state or perform operations between the two general-purpose registers, **without needing to specify a target**.

Instruction syntax: Instruction

Zero-Operand Instruction

addR

addR

This instruction adds the values contained in the two general-purpose registers (AX and BX). The result is stored in register AX.

AX = AX + BX

op-code

0x18

Microinstructions:

- AX → ALUA
- ALUA + BX → ALUOUT
- ALUOUT → AX

The **addR** instruction requires 3 clock cycles to execute.

Usage example

```
load AX var1
load BX var2
addR
...
```

Zero-Operand Instruction

subR

subR

This instruction subtracts the value in register BX from the value in register AX. The result is stored in register AX.

AX = AX - BX

op-code

0x1E

Microinstructions:

- AX → ALUA
- ALUA - BX → ALUOUT
- ALUOUT → AX

The **subR** instruction requires 3 clock cycles to execute.

Usage example

```
load AX var1
load BX var2
subR
...
```

Zero-Operand Instruction

andR

andR

This instruction performs a bitwise AND operation between the values stored in the general-purpose registers. The result is stored in register AX.

AX = AX AND BX

op-code

0x21

Microinstructions:

- AX ➔ ALUA
- ALUA AND BX ➔ ALUOUT
- ALUOUT ➔ AX

The **andR** instruction requires 3 clock cycles to execute.

Usage example

```
load AX var1
load BX var2
andR
...
```

Zero-Operand Instruction

orR

orR

This instruction performs a bitwise OR operation between the values stored in the general-purpose registers. The result is stored in register AX.

AX = AX OR BX

op-code

0x1B

Microinstructions:

- AX → ALUA
- ALUA OR BX → ALUOUT
- ALUOUT → AX

The **orR** instruction requires 3 clock cycles to execute.

Usage example

```
load AX var1
load BX var2
orR
...
```

Zero-Operand Instruction

cmpR

cmpR [register] [variable]

This instruction compares the values stored in the general-purpose registers. The comparison is performed by subtracting the value in register BX from the value in register AX, updating the Flag register accordingly, and discarding the result.

op-code

0x32

Microinstructions:

- AX → ALUA
- ALUA - BX → ALUOUT

The **cmpR** instruction requires 2 clock cycles to execute.

Usage example

```
load AX var1
load BX var2
cmpR
...
```

Zero-Operand Instruction

flip

flip

This instruction swaps the values stored in the two general-purpose registers (AX and BX).

op-code

0x34

Microinstructions:

- AX → ALUA
- BX → AX
  ALUA → ALUOUT

The **flip** instruction requires 2 clock cycles to execute.

Usage example

```
load AX var1
load BX var2
flip
...
```

Zero-Operand Instruction

pause

pause

This instruction halts the execution of the program.

op-code

0x38

Microinstructions:

- CLKDIS = 1
- (wait)

The **pause** instruction requires 2 clock cycles to execute.

Usage example

```
...
addR
store BX var3
pause
```

Immediate Instructions

These instructions consist of **two parameters**: the first is the **general-purpose register** (AX or BX) they refer to, and the second is the **constant value** to be considered.

Unlike direct instructions, these instructions do not require memory access to read the data.

Instruction syntax: Instruction [register] [value]

Immediate Instruction

load$

load$ [register] [value]

This instruction copies the *[value]* parameter into the *[register]*. It is a load instruction.

op-code

0x59

Microinstructions:

- DR → [register]

The **load$** instruction requires 1 clock cycle to execute.

Usage example

```
...
load$ AX 3
...
```

Immediate Instruction

add$

add$ [register] [value]

This instruction adds the *[value]* parameter to the value stored in the *[register]*. The result is saved in the specified register.

[register] = [register] + [value]

op-code

0x50

Microinstructions:

- [register] ➔ ALUA
- ALUA + DR ➔ ALUOUT
- ALUOUT ➔ [register]

The **add$** instruction requires 3 clock cycles to execute.

Usage example

```
load AX var1
...
add$ AX 3
...
```

Immediate Instruction

sub$

sub$ [register] [value]

This instruction subtracts the *[value]* parameter from the value stored in the *[register]*. The result is saved in the specified register.

[register] = [register] - [value]

op-code

0x4D

Microinstructions:

- [register] → ALUA
- ALUA - DR → ALUOUT
- ALUOUT → [register]

The **sub$** instruction requires 3 clock cycles to execute.

Usage example

```
load AX var1
...
sub$ AX 3
...
```

Immediate Instruction

and$

and$ [register] [value]

This instruction performs a bitwise AND operation between the value stored in the *[register]* and the *[value]* parameter. The result is saved in the specified register.

[register] = [register] AND [value]

op-code

0x53

Microinstructions:

- [register] → ALUA
- ALUA AND DR → ALUOUT
- ALUOUT → [register]

The **and$** instruction requires 3 clock cycles to execute.

Usage example

```
load AX var1
...
and$ AX 3
...
```

Immediate Instruction

or$

or$ [register] [value]

This instruction performs a bitwise OR operation between the value stored in the *[register]* and the *[value]* parameter. The result is saved in the specified register.

[register] = [register] OR [value]

op-code

0x56

Microinstructions:

- [register] → ALUA
- ALUA OR DR → ALUOUT
- ALUOUT → [register]

The **or$** instruction requires 3 clock cycles to execute.

Usage example

```
load AX var1
...
or$ AX 3
...
```

Immediate Instruction

shiftl$

shiftl$ [register] [value]

This instruction performs a logical left shift on the value stored in the *[register]* by the number of bits specified in the *[value]* parameter. The result is saved in the specified register.

[register] = [register] shiftl [value]

op-code

0x2A

Microinstructions:

- [register] ➔ ALUA
- ALUA SHIFTL DR ➔ ALUOUT
- ALUOUT ➔ [register]

The **shiftl$** instruction requires 3 clock cycles to execute.

Usage example

```
var = 1
load AX var
shiftl$ AX 1
...
```

Immediate Instruction

shiftr$

shiftr$ [register] [value]

This instruction performs a logical right shift on the value stored in the *[register]* by the number of bits specified in the *[value]* parameter.
The result is saved in the specified register.

[register] = [register] shiftr [value]

op-code

0x2D

Microinstructions:

- [register] → ALUA
- ALUA SHIFTR DR → ALUOUT
- ALUOUT → [register]

The **shiftr$** instruction requires 3 clock cycles to execute.

Usage example

```
var = 1
load AX var
shiftR$ AX 1
...
```

Immediate Instruction

cmp$

cmp$ [register] [value]

This instruction compares the value stored in the *[register]* with the *[value]* parameter. The comparison is performed by subtracting the [value] from the register's value, updating the Flag register accordingly, and discarding the result.

op-code

0x5A

Microinstructions:

- [register] → ALUA
- ALUA OR DR → ALUOUT

The **cmp$** instruction requires 2 clock cycles to execute.

Usage example

```
load AX var1
...
cmp$ AX 3
...
```

Jump Instructions

These instructions **interrupt the linear execution of the program** to execute code segments identified by **labels**.

Instruction syntax: Instruction [label]

jmp

jmp [label]

This instruction unconditionally jumps to the specified *[label]*.

op-code

0x4C

Microinstructions:

● DR → PC

The **jmp** instruction requires 1 clock cycle to execute.

Usage example

```
...
cmp AX var1
jmp labelTest
...
labelTest:
...
```

jc

jc [label]

This instruction jumps to the specified *[label]* if the **carry bit (C)** is set to 1 by the previous instruction.

op-code

0x3A

Microinstructions:

- DR → PC

The **jc** instruction requires 1 clock cycle to execute.

Usage example

```
...
jc labelTest
...
labelTest:
...
```

jv

jv [label]

This instruction jumps to the specified *[label]* if the **overflow bit (V)** is set to 1 by the previous instruction.

op-code

0x3C

Microinstructions:

● DR → PC

The **jv** instruction requires 1 clock cycle to execute.

Usage example

```
...
jv labelTest
...
labelTest:
...
```

jn

jn [label]

This instruction jumps to the specified [label] if the **negative bit (N)** is set to 1 by the previous instruction.

op-code

0x3E

Microinstructions:

- DR → PC

The **jn** instruction requires 1 clock cycle to execute.

Usage example

```
...
jn labelTest
...
labelTest:
...
```

jz

jz [label]

This instruction jumps to the specified *[label]* if the **zero bit (Z)** is set to 1 by the previous instruction.

op-code

0x40

Microinstructions:

- DR → PC

The **jz** instruction requires 1 clock cycle to execute.

Usage example

```
...
jz labelTest
...
labelTest:
...
```

je

jc [label]

⚠
This instruction must immediately follow a compare instruction (cmp, cmpR, cmp$).

It jumps to the specified *[label]* if and only if the operands specified by the previous compare instruction are equal; otherwise, the program continues normally.

op-code

0x42

Microinstructions:

- DR ➔ PC

The **je** instruction requires 1 clock cycle to execute.

Usage example

```
...
cmp AX var1
je labelTest
...
labelTest:
...
```

jg

jg [label]

⚠
This instruction must immediately follow a compare instruction (cmp, cmpR, cmp$).

It jumps to the specified *[label]* if and only if the first operand in the previous compare instruction is greater than the second operand; otherwise, the program continues normally.

op-code

0x44

Microinstructions:

- DR → PC

The **jg** instruction requires 1 clock cycle to execute.

Usage example

```
...
cmp AX var1
jg labelTest
...
labelTest:
...
```

Jump Instruction

jl

jg [label]

⚠
This instruction must immediately follow a compare instruction (cmp, cmpR, cmp$).

It jumps to the specified *[label]* if and only if the first operand in the previous compare instruction is less than the second operand; otherwise, the program continues normally.

op-code
0x46

Microinstructions:

- DR → PC

The **jl** instruction requires 1 clock cycle to execute.

Usage example

```
...
cmp AX var1
jl labelTest
...
labelTest:
...
```

jle

jle [label]

⚠
This instruction must immediately follow a compare instruction (cmp, cmpR, cmp$).

It jumps to the specified *[label]* if and only if the first operand in the previous compare instruction is less than or equal to the second operand; otherwise, the program continues normally.

op-code

0x48

Microinstructions:

- DR → PC

The **jle** instruction requires 1 clock cycle to execute.

Usage example

```
...
cmp AX var1
jle labelTest
...
labelTest:
...
```

jge

jge [label]

⚠
This instruction must immediately follow a compare instruction (cmp, cmpR, cmp$).

It jumps to the specified *[label]* if and only if the first operand in the previous compare instruction is greater than or equal to the second operand; otherwise, the program continues normally.

op-code

0x4A

Microinstructions:

● DR → PC

The **jge** instruction requires 1 clock cycle to execute.

Usage example

```
...
cmp AX var1
jge labelTest
...
labelTest:
...
```

A processor must also be capable of **receiving data from external sources**, processing it, and **displaying it to the user**.
To accomplish this, Flip01 utilizes an **external circuit called the Input & Output Manager**, or simply the

I/O Manager

This circuit is responsible for reading from and writing to **64 unique I/O devices**.
Split into **32 input devices** and **32 output devices**, each port has an identifier ranging from 0 to 31.

During the program, the user will decide which port to connect their devices to and assign them an identifier number accordingly.

In the demo version of Flip01 on Logisim, both in the "*Flip01_Full*" circuit and the "*Playground*" version (which features a compact processor design to facilitate external connections), the I/O Management module **already has devices connected in specific positions**, with ports already assigned.

Input:

- Port **I00** - Keyboard Data [7 bits]
- Port **I01** - Keyboard Ready: Indicates if the keyboard buffer contains a character [1 bit]
- Port **I23** - BTN0: A push-button *(1/8)* [1 bit]
- Port **I24** - BTN1: A push-button *(2/8)* [1 bit]
- Port **I25** - BTN2: A push-button *(3/8)* [1 bit]
- Port **I26** - BTN3: A push-button *(4/8)* [1 bit]
- Port **I27** - BTN4: A push-button *(5/8)* [1 bit]
- Port **I28** - BTN5: A push-button *(6/8)* [1 bit]
- Port **I29** - BTN6: A push-button *(7/8)* [1 bit]
- Port **I30** - BTN7: A push-button *(8/8)* [1 bit]

The **21 input** ports from **I02 to I22** are unassigned and can be freely used by the user.

Output:

- Port **O00** - TTY Data (Teletypewriter, capable of displaying text messages) [7 bits]
- Port **O01** - RGB Video X Coordinate [8 bits]
- Port **O02** - RGB Video Y Coordinate [8 bits]
- Port **O03** - RGB Video Write Enable: When set to 1, it writes to the cursor position defined by X and Y [1 bit]
- Port **O04** - LED State 0: Turns on if it receives 1, and stays on until it receives 2, and vice versa *(1/4)*[2 bits]
- Port **O05** - LED State 1: Turns on if it receives 1, and stays on until it receives 2, and vice versa *(2/4)*[2 bits]
- Port **O06** - LED State 2: Turns on if it receives 1, and stays on until it receives 2, and vice versa *(3/4)*[2 bits]
- Port **O07** - LED State 3: Turns on if it receives 1, and stays on until it receives 2, and vice versa *(4/4)*[2 bits]

Output:

- Port **O23** - LED Matrix 8x8 Row 0 *(1/8)*[8 bits]
- Port **O24** - LED Matrix 8x8 Row 1 *(2/8)*[8 bits]
- Port **O25** - LED Matrix 8x8 Row 2 *(3/8)*[8 bits]
- Port **O26** - LED Matrix 8x8 Row 3 *(4/8)*[8 bits]
- Port **O27** - LED Matrix 8x8 Row 4 *(5/8)*[8 bits]
- Port **O28** - LED Matrix 8x8 Row 5 *(6/8)*[8 bits]
- Port **O29** - LED Matrix 8x8 Row 6 *(7/8)*[8 bits]
- Port **O30** - LED Matrix 8x8 Row 7 *(8/8)*[8 bits]
- Port **O31** - Reset Bit: When set to 1, it resets the connected device. If multiple devices are in use, it might be helpful to assign different reset bits to reset devices at different times.

The **15 output** ports from **O08 to O22** are unassigned and can be freely used by the user.

If you need to change the reference ports for certain components, you will have to **update the corresponding port** entered as a constant within the module.

The port inside the module must match the selected port.

This step is also crucial if **additional components of the same type are added**.

The devices that require this change are:

The 8x8 LED matrix controller

Each row corresponds to an output, meaning **each port must be synchronized with the I/O Manager.**

The RGB video controller

The X and Y coordinate inputs correspond to **two ports that also need to be synchronized with the I/O Manager**.

I/O Instructions

These instructions **handle the communication between external devices and the CPU**.
They require **two parameters**: the first is the register that will be affected by the instruction, while the second is the name of the device that will interact with that register.

Instruction syntax: Instruction [register][device]

input

input [register] [device]

This instruction reads the value transmitted from the input device *[device]* connected to the port *[port number]* and copies it into the specified register *[register]*.

Each input device must be declared using the syntax:

> [device] [port number]

op-code

0x5E

Microinstructions:

- DR > I/O Manager
  [device] > [register]

The **input** instruction requires 1 clock cycle to execute.

Usage example

```
> InputDevice 3
...
input AX inputDevice
...
pause
```

output

output [register] [device]

This instruction reads the value stored in the specified register *[register]* and transmits it to the output device *[device]* connected to the port *[port number]*.

Each output device must be declared using the syntax:

< [device] [port number]

op-code

0x5F

Microinstructions:

- DR > I/O Manager
  [register] > [device]

The **output** instruction requires 1 clock cycle to execute.

Usage example

```
< OutputDevice 5
...
output AX OutputDevice
...
pause
```

Writing programs and manually replacing op-codes, hexadecimal values, and addresses can be tiresome, so we have created a small **assembler**, which acts as a **translator from high-level language**.

We believe that the UI and UX are quite intuitive, but it may be helpful to explicitly outline **all possible errors**, their **corresponding codes**, and **how to resolve them:**

Errors (1/5)

- **E - 000:** The file you are trying to open and read using the designated button may be corrupted or unreachable. Please check that the file is intact and in the correct format (.txt).
- **E - 001:** The code you have written is too large for Flip01!
  Try to shorten it and clean it up by using calls to labels for repeated code segments.
- **E - 002:** The instruction you have written is not among those supported by Flip01.
  Double-check that the syntax is correct and remember that the assembler is case-sensitive: instructions must be written in lowercase (except for the R in immediate instructions).

Errors (2/5)

- **E - 003:** The instruction does not have a declared variable as its second argument.
  This argument must not be a number.
- **E - 004:** The instruction is missing some required parameters.
  Refer to the previous chapter to check how many and which arguments are needed for each instruction.
- **E - 005:** This instruction requires a general-purpose register (AX or BX) as the first argument.
  The parameter written in the first position is likely incorrect or undefined.
- **E - 006:** The instruction has more parameters than required.
  Refer to the previous chapter to check how many and which arguments are needed for each instruction.
- **E - 007:** The argument associated with this direct instruction is too large!
  Remember that acceptable values range from 0 to 255.
- **E - 008:** The argument associated with the direct instruction is not an integer.
  It is likely that an alphanumeric value was entered or that this parameter has not been defined.

Errors (3/5)

- **E - 009:** The variable has not been declared at the beginning of the program.
  Ensure that the variable name is spelled correctly and that the name written in the declaration matches the one used in the instruction.
  Remember that the assembler is case-sensitive, and the declaration must follow the structure:

  [variable name] = [numeric value]

- **E - 010:** The value associated with the variable is not an integer.
  An alphanumeric value may have been entered, or this parameter has not been defined.

- **E - 011:** This variable has been defined multiple times within the code.
  Remember that to change the value of a memory cell, you must use processor instructions and not assignment operations; the latter are only for declarations.

- **E - 012:** The value associated with this variable is too large! Acceptable values range from 0 to 255.

- **E - 013:** The label has been called but not declared.
  Each label must be defined with the syntax [label:].
  Check for any spelling errors or issues related to the case-sensitivity of the assembler.

Errors (4/5)

- **E - 014:** This label is defined multiple times within the code.
  A label can have only one definition, and two labels cannot share the same name.
- **E - 015:** You have declared too many input devices.
  Flip01 supports up to 32 different input devices.
- **E - 016:** You have declared too many output devices.
  Flip01 supports up to 32 different output devices.
- **E - 017:** The name assigned to this variable is a number. Variable names must contain at least one letter.
- **E - 018:** You have declared a device as input but then used it as output.
  Ensure that the declaration and usage are consistent (> input).
- **E - 019:** You have declared a device as output but then used it as input.
  Ensure that the declaration and usage are consistent (< output).
- **E - 020:** You are using a device that you have not declared.
  Ensure that the name matches the declaration and remember that Flip01 is case-sensitive.

Errors (5/5)

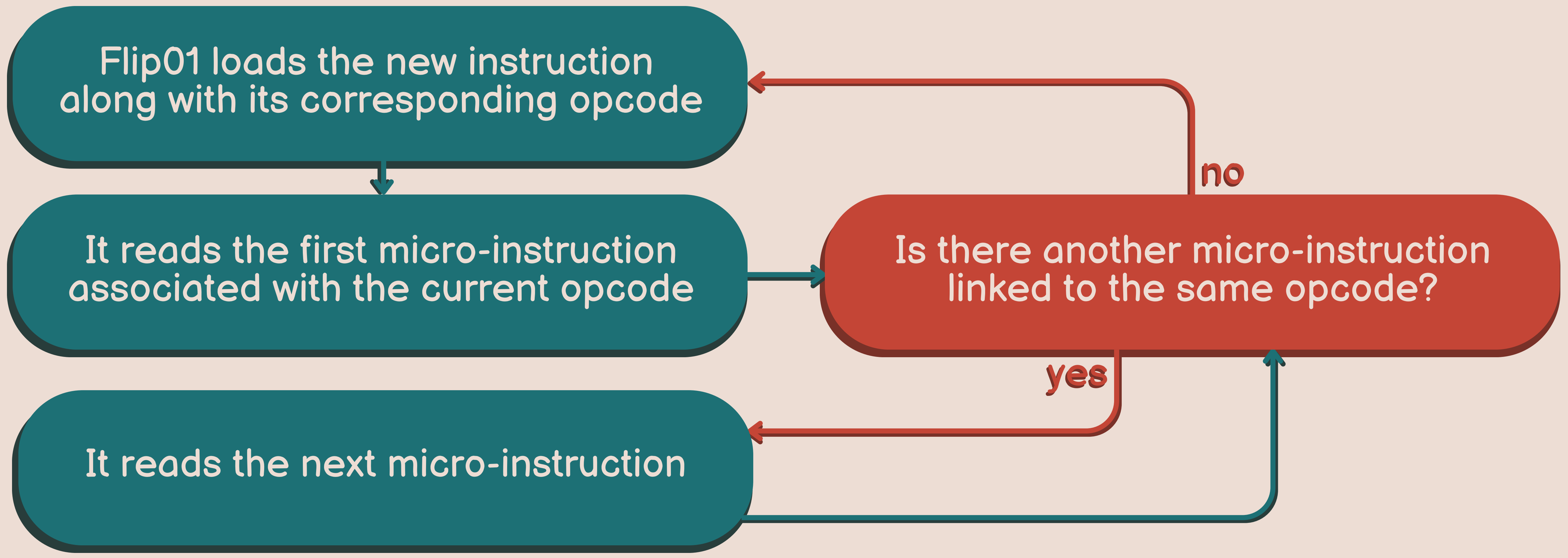
- **E - 021:** The name assigned to this label is a number.
  Label names must contain at least one letter.
- **E - 022:** The name assigned to this device is a number.
  I/O device names must contain at least one letter.
- **E - 023:** You have not specified which device to use for this input or output operation.
- **E - 024:** The name you have chosen for this I/O device has already been used for another device.
- **E - 025:** The port number you have chosen for this device has already been assigned to another device.
  To avoid conflicts, use different ports for different devices.
- **E - 026:** The field that should contain the port number for the I/O device does not contain an integer.
  Please check the format.
- **E - 027:** The port number assigned to this device is not a valid integer.
  Remember that port numbers must be between 0 and 31.

Micro Program Counter

The micro program counter is a section of Flip01 that, upon receiving the op-code of the currently active function, **sequentially produces all the control signals** that regulate the state of each component of the circuit and define the operation of the micro-instructions associated with that instruction.

To achieve this, the following steps are executed in sequence:

1. The micro program counter **reads the first micro-instruction** from a third memory (MEM3) at the address corresponding to the op-code of the instruction.
2. If there is another micro-instruction linked to that instruction, it executes it; otherwise, the address returns to 0, executing the fetch instruction, which retrieves the next instruction to be executed. **This process is repeated until the described condition becomes true**.


Flip01 loads the new instruction along with its corresponding opcode
It reads the first micro-instruction associated with the current opcode
Is there another micro-instruction linked to the same opcode?
no
yes
It reads the next micro-instruction

The memory accessed by the micro program counter (MEM3) is non-volatile, and **the set of control signal instructions is referred to as the ISA** (*Instruction Set Architecture*).

Flip01 has been in development for a long time, perhaps too long for what it is, and yet it is still far from being a complete project.
There are still many possible implementations to explore, which is why we have made it open source.

Now Flip01 is in your hands; use your creativity, play with it, break it, and push beyond every limit we have designed.
Have fun.

-Pescetti Studio

(Riccardo Biasolo e Lorenzo Croci)

If you find errors, inaccuracies, or typos, write to us!
Send us an email at

pescettistudio@gmail.com

with [bug] at the beginning of the subject line.