INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

COMPUTAÇÃO FÍSICA

PROJECTO 3

André Fonseca
A39758@alunos.isel.pt

Daniel Santos
A32078@alunos.isel.pt

Guilherme Rodrigues
A41863@alunos.isel.pt

Professor Eng. Carlos Carvalho
cfc@cedet.isel.pt

Junho 2017

ÍNDICE

INTRODUÇÃO **3**

INSTRUÇÕES DO CPU **4**

MÓDULO FUNCIONAL **5**

MÓDULO DE CONTROLO **10**

TABELA EPROM 11

MÓDULO X 13

SIMULAÇÃO EM ARDUÍNO **14**

PROGRAMAS DE TESTE **16**

PROGRAMA 1 16

PROGRAMA 2 16

PROGRAMA 3 16

PROGRAMA 4 16

CONCLUSÃO **17**

BIBLIOGRAFIA **18**

ANEXO – CÓDIGO DE SIMULAÇÃO EM ARDUINO **19**

INTRODUÇÃO

O processador ou CPU[1] é um circuito electrónico dentro de um computador que realiza instruções provenientes de programas de computador. Essas instruções resumem-se a operações aritméticas, lógicas, de controlo e de I/O[2].

Neste projecto pretende-se desenhar de um microprocessador baseado na arquitectura de Harvard que cumpra o seguinte conjunto de instruções:

Instruction	**Functionality**
MOV A, #CONSTANT	A = constant
MOV A, @P	A = M(P)
MOV @P, A	M(P) = A
MOV P, A	P = A
MOV B, A	B = A
ADDC A, B	A = A + B + C
SUBB A, B	A = A - B - C
JC rel5	if (carry) PC += rel5
JZ rel5	if (zero) PC += rel5
JMP end7	PC = end7

Uma arquitectura de computadores define um conjunto de métodos que descreve as funcionalidades, a organização e implementação de um computador, incluindo o seu conjunto de instruções e o desenho da arquitectura dos componentes lógicos e electrónicos. Estas arquitecturas são tipicamente constituídas pela unidade central de processamento e o seu I/O, por uma ou várias unidades de memória e a ALU[3].

A arquitectura Harvard baseia-se na separação dos componentes: a unidade de controlo, memória de código, memória de dados e interface de I/O; sendo estos componentes ligadas através de um bus - sistema de comunicação que transfere dados entre componentes dentro de um computador. Deve-se salientar que os computadores modernos utilizam uma arquitectura de Harvard modificada.

O conjunto de instruções do microprocessador deste projecto irá fazer uso da memória de código para definir as instruções que o CPU permite realizar, a transferência de dados através dos vários bus' das unidade e também a memória de dados para registar dados em memória.

As memórias podem ser acedidas por um address bus que transporta os endereços das posições de memória; e que retornam o seu conteúdo através do data bus.

---

[1] Central Processing unit
[2] Input/Output
[3] Arithmetic-logic unit

INSTRUÇÕES DO CPU

De forma a que o CPU possa realizar um conjunto de instruções é necessário identificá-las. A identificação de instruções é feita através de uma codificação de bits única ao CPU.

A instrução ‘MOV A, #CONSTANT’ permite mover o valor de uma constante a 8 bits para a variável ou registo A. Esta instrução irá precisar de alocar o valor dos dados com a sua identificação. Com base neste princípio foi elaborada a seguinte codificação de instruções:

INSTRUCTIONS CODIFICATION									
Instruction	**Functionality**	**D8**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
MOV A, #CONST	A = CONST	**1**	c7	c6	c5	c4	c3	c2	c1	c0
MOV A, @P	A = M(P)	0	0	**1**	**1**	0	0	0	0	0
MOV @P, A	M(P) = A	0	0	**1**	**1**	0	0	0	0	**1**
MOV P, A	P = A	0	0	**1**	**1**	0	0	0	**1**	0
MOV B, A	B = A	0	0	0	0	0	0	0	0	0
ADDC A, B	A = A + B + Cy	0	0	0	0	0	0	0	0	**1**
SUBB A, B	A = A - B - Cy	0	0	0	0	0	0	0	**1**	0
JC rel5	if (Cy) PC += rel5	0	0	0	**1**	r4	r3	r2	r1	r0
JZ rel5	if (Zero) PC+= rel5	0	0	**1**	0	r4	r3	r2	r1	r0
JMP end7	PC = end7	0	**1**	e6	e5	e4	e3	e2	e1	e0

Todas as instruções têm uma identificação única, como se tratasse de uma assinatura digital que irá ser reconhecida na unidade central do CPU.

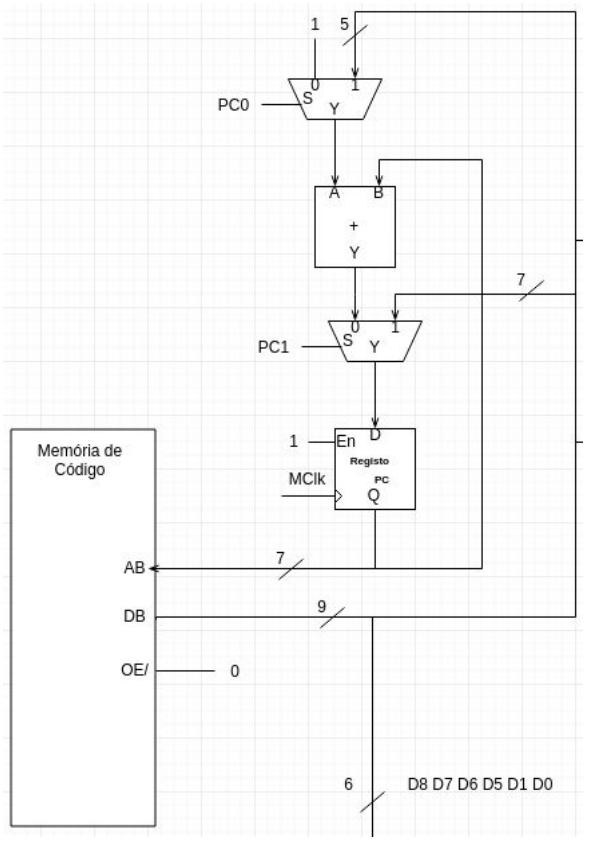
MÓDULO FUNCIONAL

A arquitectura do módulo funcional foi estruturada de acordo com as instruções que o CPU precisa de executar. No geral, as instruções utilizam os seguintes componentes: a memória de código, memória de dados, ALU, o PC[4], os registos A, B e P, o módulo de controlo e módulo complementar, designado de módulo X, que será responsável por afectar os valores das flags de carry e zero.

A memória de código tem como funcionalidade armazenar todas as instruções que um determinado programa irá cumprir. Estes programas têm a possibilidade de realizar instruções de 'JUMP', 'JZ' ou JC' que especificam posições da memória de código relativas a outras instruções.

As instruções 'JZ' e 'JC' são realizadas através do incremento do PC relativamente ao seu valor actual, enquanto que a instrução 'JUMP' atribui-lhe um novo endereço de forma absoluta.

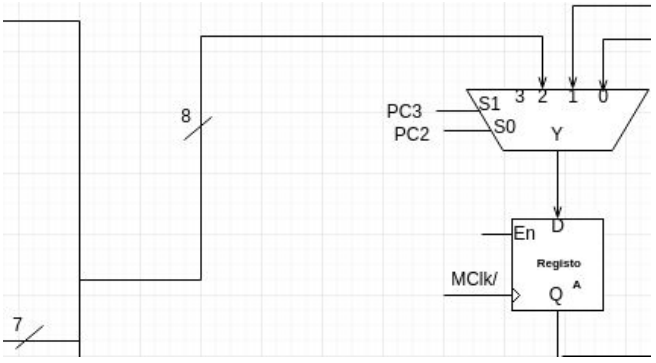
Para tal, é necessário o uso de um multiplexer que permita controlar a selecção entre o endereço presente em PC ou um novo endereço absoluto.



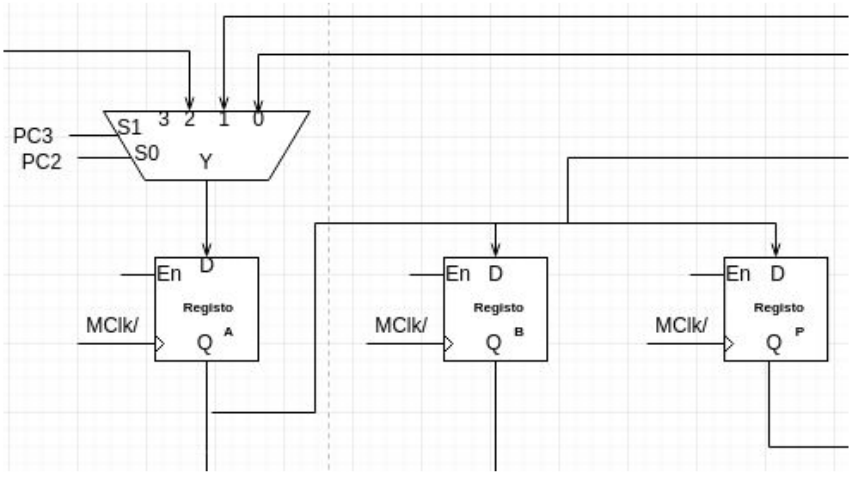
Parte do módulo funcional correspondente à execução das instruções 'JUMP', 'JC' e 'JZ'.

[4] Program Counter

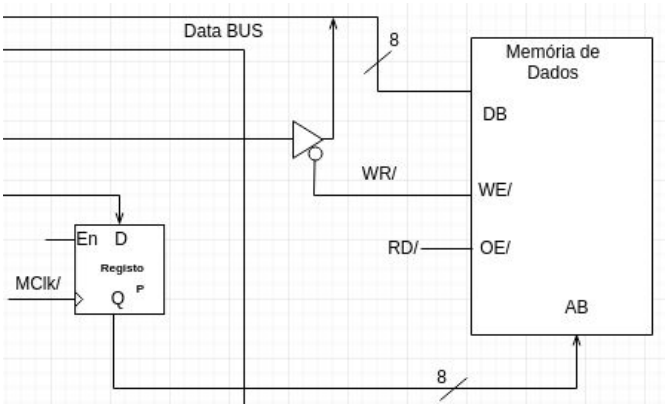
A transferência de dados entre registos é realizada pelas instruções de ‘MOV’ entre os registo A, P e B e para a memória de dados através de o endereço de @P e o valor presente no registo P.



Bifurcação do databus da Memória de Código



Parte do módulo funcional correspondente à execução das instruções ‘MOV’.

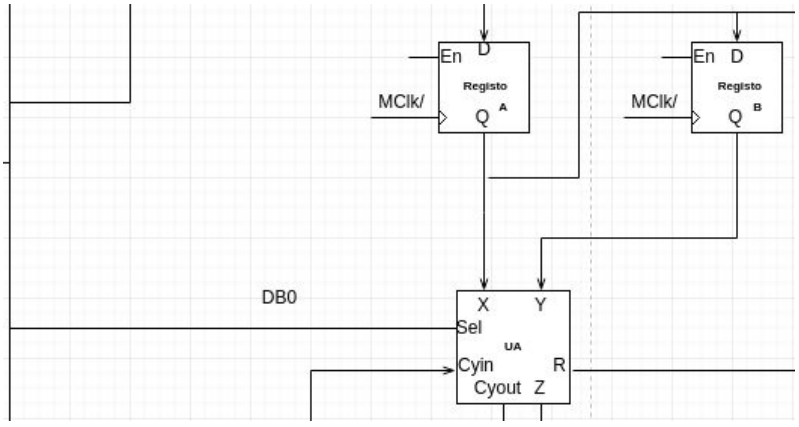


Parte do módulo funcional correspondente à transferência de dados para a memória de dados pelo registo P.

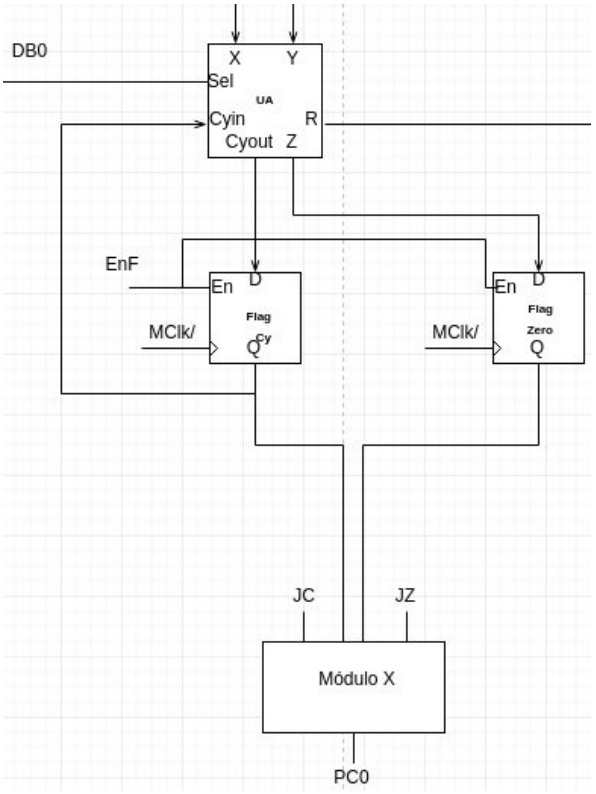
Dada a definição de como as memórias são acedidas, através do address bus e data bus, facilmente se consegue deduzir o número de bits transferidos por cada bus:

- Memória de código:
  - Address bus: 7 bits;
  - Data bus: 9 bits.
- Memória de dados:
  - Address bus: 8 bits;
  - Data bus: 8 bits.

A ALU tem dois valores de entrada, sendo um deles a saída do registo A e o outro a saída do registo B. É apenas necessário um bit de selecção para determinar qual a operação aritmética a realizar, pois apenas podem ser realizadas duas operações: soma e subtração.

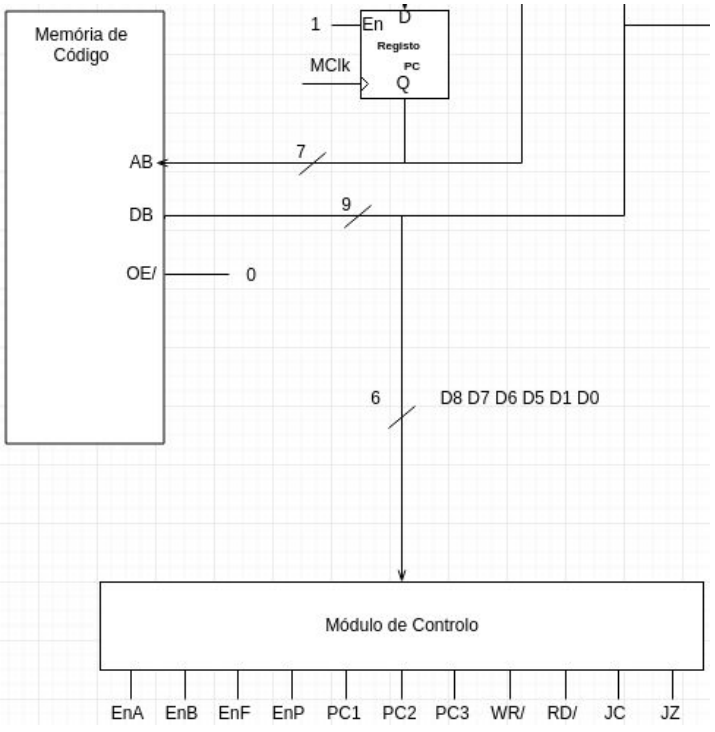


Unidade aritmética (ALU), responsável por realizar as instruções ‘ADDC’ e ‘SUBB’.



Unidade aritmética e registos de flag juntamente com o módulo X

O módulo de controlo tem como entrada os 6 bits únicos que identificam cada instrução e, como resultado ou saída, os bits necessários ao funcionamento correcto do módulo funcional na execução de cada instrução.



O módulo funcional pode ser consultado por completo no seguinte link:
https://www.draw.io/?title=CF%20P03#Uhttps%3A%2F%2Fdrive.google.com%2Fuc%3Fid%3D0B_4l8qw7V90lWk1DUnpJQlpFblk%26export%3Ddownload

MÓDULO DE CONTROLO

O módulo de controlo identifica das instruções a realizar. Esta identificação é feita pelos 6 bits de instrução da codificação de instruções: D8, D7, D6, D5, D1, D0.

Com base nestes bits de entrada vão ser definidos os sinais ou flags de controle de forma a que o módulo funcional active os componentes certos para realizar a instrução correcta.

Relativamente às instruções ‘JC rel 5’ e ‘JZ rel5’ será necessário fazer a verificação da existência de carry ou zero, sinais provenientes da ALU.

CONTROL MODULE										
Instruction	**Functionality**	**D8**	**D7**	**D6**	**D5**	**D1**	**D0**	**Cy**	**Z**	**Signals**
MOV A, #CONST	A = CONST	**1**	c7	c6	c5	c1	c0	-	-	EnA, PC3
MOV A, @P	A = M(P)	0	0	**1**	**1**	0	0	-	-	EnA, PC2, !RD
MOV @P, A	M(P) = A	0	0	**1**	**1**	0	**1**	-	-	!WR
MOV P, A	P = A	0	0	**1**	**1**	**1**	0	-	-	EnP
MOV B, A	B = A	0	0	0	0	0	0	-	-	EnB
ADDC A, B	A = A + B + Cy	0	0	0	0	0	**1**	-	-	EnF, !PC2, !PC3, EnA
SUBB A, B	A = A - B - Cy	0	0	0	0	**1**	0	-	-	EnF, !PC2, !PC3, EnA
JC rel5	if (Cy) PC += rel5	0	0	0	**1**	-	-	1	-	PC0, !PC1
		0	0	0	**1**	-	-	0	-	
JZ rel5	if (Zero) PC+= rel5	0	0	**1**	0	-	-	-	1	PC0, !PC1
		0	0	**1**	0	-	-	-	0	
JMP end7	PC = end7	0	**1**	-	-	-	-	-	-	PC1

Ao traduzir as instruções para referências da memória de código verifica-se que a sua EPROM necessitará de 128 entradas. Para reduzir o seu tamanho irá ser criado um novo módulo para tratar dos sinais correspondentes à existência de carry ou zero nas operações aritméticas da ALU. Este módulo será designado de módulo X.

Todas as tabelas utilizadas na codificação de instruções e EPROM podem ser consultadas na íntegra no seguinte link: https://docs.google.com/spreadsheets/d/1EqnbPua1Y6V_CuXUmJuIQrGFDwrBcG8269CxPkM6V4I/edit?usp=sharing

TABELA EPROM

Para determinar o número de endereços que uma instrução necessita basta verificar a quantidade de “don’t care” dessa instrução (x) e calcular 2^x.

O intervalo de endereços de uma instrução é definido através da conversão para hexadecimal do menor número binário possível e do maior número binário possível da codificação desta instrução.

Por exemplo, para a instrução ‘MOV B, A’:

- D8 D7 D6 D5 D1 D0 Cy Z
1. 0 0 0 0 0 0 - -

O menor e maior números binários possíveis da sua codificação para esta instrução:

2. 0 0 0 0 0 0 0 0 = 0x0
3. 0 0 0 0 0 0 1 1 = 0x3

Assim, o intervalo de endereços será: [0x0 0x3].

As instruções ‘JC rel5’ e ‘JZ rel5’ apresentam a particularidade de terem um bit fixo entre ou à direita dos “don’t care”, assim não terão um intervalo sequencial de endereços, mas sim um conjunto de endereços.

EPROM 128x10										
Inst.	**D8**	**D7**	**D6**	**D5**	**D1**	**D0**	**Cy**	**Z**	**Quant.**	**Address**
MOV B, A	0	0	0	0	0	0	-	-	4	0x0 0x3
ADDC A, B	0	0	0	0	0	1	-	-	4	0x4 0x7
SUBB A, B	0	0	0	0	1	0	-	-	4	0x8 0xB
JC rel5	0	0	0	1	-	-	0	-	8	0x10, 0x11, 0x14, 0x15, 0x18, 0x19, 0x1C, 0x1D
	0	0	0	1	-	-	1	-	8	0x12, 0x13, 0x16, 0x17, 0x1A, 0x1B, 0x1E, 0x1F
JZ rel5	0	0	1	0	-	-	-	0	8	0x20, 0x22, 0x24, 0x26, 0x28, 0x2A, 0x2C, 0x2E
	0	0	1	0	-	-	-	1	8	0x21, 0x23, 0x25, 0x27, 0x29, 0x2B, 0x2D, 0x2F
MOV A, @P	0	0	1	1	0	0	-	-	4	0x2C 0x2F
MOV @P, A	0	0	1	1	0	1	-	-	4	0x30 0x33
MOV P, A	0	0	1	1	1	0	-	-	4	0x34 0x37
JMP end7	0	1	-	-	-	-	-	-	64	0x38 0x77
MOV A, #CONST	1	c7	c6	c5	c1	c0	-	-	128	0x78 0xF7

Ao se proceder às transferências das flags de entrada de carry e zero para o módulo X o tamanho da EPROM diminui para 32 entradas, o que se revela ser uma alteração bastante significativa. Para tal se suceder, apenas é necessário ter duas novas flags de saída que indiquem a utilização da instrução de ‘JC rel5’ ‘JZ rel5’.

EPROM 32x11

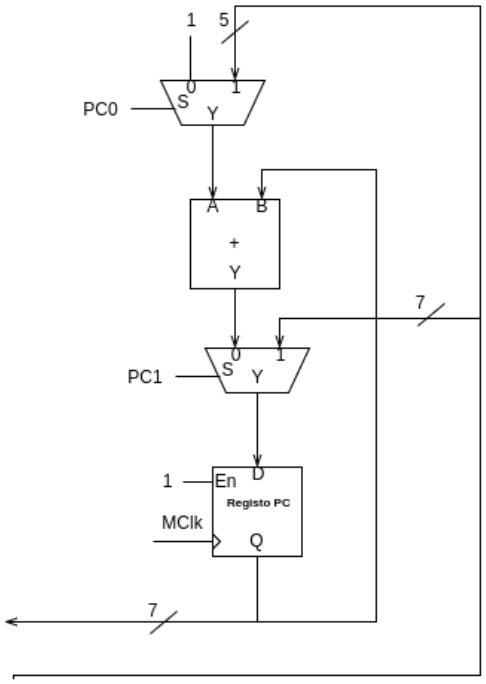
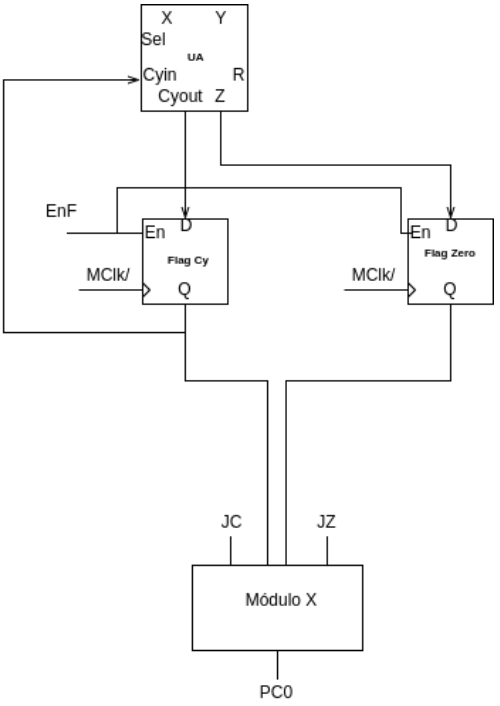
Instruction	D8	D7	D6	D5	D1	D0	Quant.	Address	
MOV B, A	0	0	0	0	0	0	1	0x0	0x0
ADDC A, B	0	0	0	0	0	1	1	0x1	0x1
SUBB A, B	0	0	0	0	1	0	1	0x2	0x2
JC rel5	0	0	0	1	-	-	4	0x4	0x7
JZ rel5	0	0	1	0	-	-	4	0x8	0xB
MOV A, @P	0	0	1	1	0	0	1	0xC	0xC
MOV @P, A	0	0	1	1	0	1	1	0xD	0xD
MOV P, A	0	0	1	1	1	0	1	0xE	0xE
JMP end7	0	1	-	-	-	-	16	0x10	0x1F
MOV A, #CONST	1	c7	c6	c5	c1	c0	32	0x20	0x3F

Flags

Address		EnA	EnB	EnF	EnP	PC1	PC2	PC3	!WR	!RD	JC	JZ	Data
0x0	0x0	0	1	0	0	0	0	0	1	1	0	0	0x20C
0x1	0x1	1	0	1	0	0	0	0	1	1	0	0	0x50C
0x2	0x2	1	0	1	0	0	0	0	1	1	0	0	0x50C
0x4	0x7	0	0	0	0	0	0	0	1	1	1	0	0x00E
0x8	0xB	0	0	0	0	0	0	0	1	1	0	1	0x00D
0xC	0xC	1	0	0	0	0	1	0	1	0	0	0	0x428
0xD	0xD	0	0	0	0	0	0	0	0	1	0	0	0x004
0xE	0xE	0	0	0	1	0	0	0	1	1	0	0	0x08C
0x10	0x1F	0	0	0	0	1	0	0	1	1	0	0	0x04C
0x20	0x3F	1	0	0	0	0	0	1	1	1	0	0	0x41C

MÓDULO X

O módulo X é um pequeno circuito combinatório que implementa a expressão lógica: **PC0 = JC & Cy | JZ & Z.** O seu resultado é transferido para o início dos componentes do program counter do módulo funcional.

Componentes do program counter	Componentes da ALU e módulo X
	

SIMULAÇÃO EM ARDUÍNO

A programação para este projecto é análoga aos projectos anteriores. A sua estrutura foi definida com base no modelo conceptual de cada módulo presente no módulo funcional documentado no presente documento, de tal forma será apenas abordado algumas operações que não foram realizadas nos projectos anteriores.

A simulação completa do projecto pode ser visualizada no seguinte link: https://circuits.io/circuits/5125845-cf-p03

O restante codigo pode ser consultado em anexo.

A operação de shift permite deslocar uma certa quantidade de bits *n* vezes para um lado, esquerdo ou direito.

```
1.      input_address = (
2.              (input_address & 0x100) >> 3 |
3.              (input_address & 0x080) >> 3 |
4.              (input_address & 0x040) >> 3 |
5.              (input_address & 0x020) >> 3|
6.              input_address & 0x002 |
7.              input_address & 0x001
8.      );
```

Selecção dos bits D8 D7 D6 D5 D1 D0 que dão entrada no módulo de controlo

```
1.  boolean read_bit(word bits, byte n) {
2.      // Shift n positions to the right
3.      bits = bits >> n;
4.      // Filter the last bit of the right
5.      bits = bits & 0x01;
6.      return bits == 0x01;
7.  }
```

Função que permite fazer a leitura de um bit

A função do módulo funcional executa passo a passo os conceitos definidos sobre o módulo funcional. É também realizada a transformação de endereços quando se pretende realizar as instruções ‘JC’ e ‘JZ’ com números negativos.

```
1.  void functional_module() {
2.      // Code memory
3.      code_memory_block(pc_reg_q0);
```

```
4.
5.      // Data memory
6.      data_memory_block(p_reg_q0, read, write);
7.
8.      // Transformation for negative numbers
9.      byte x = code_memory_db & 0x01F;
10.
11.     if (read_bit(x, 4)) {
12.         x |= 0x0E0;
13.     }
14.
15.     // Program counter
16.     // Mask bits D4, D3, D2, D1, D0
17.     pc_mux_y0 = MUX_2x1(pc0_enable, 1, x);
18.     pc_add_y0 = add(pc_mux_y0, pc_reg_q0);
19.     pc_mux_y1 = MUX_2x1(pc1_enable, pc_add_y0, code_memory_db & 0x07F);
20.     pc_reg_d0 = pc_mux_y1;
21.
22.     // A, B, P registers
23.     // Mask bits D7, D6, D5, D4, D3, D2, D1, D0
24.     a_mux_y0 = MUX_4x1(pc3_enable, pc2_enable, 0, code_memory_db &
    0x0FF, data_memory_db, alu_r);
25.
26.     a_reg_d0 = a_mux_y0;
27.     b_reg_d0 = a_reg_q0;
28.     p_reg_d0 = a_reg_q0;
29.     z_reg_d0 = alu_z;
30.     cy_reg_d0 = alu_c;
31.
32.     // ALU
33.     alu_r = alu((code_memory_db & 0x01) == 1, a_reg_q0, b_reg_q0,
    alu_c);
34.
35.     // X module
36.     pc0_enable = x_module(jump_carry, cy_reg_q0, jump_zero, z_reg_q0);
37. }
```

Função do módulo funcional

PROGRAMAS DE TESTE

De forma a testar todas as instruções do CPU foram realizados 4 pequenos programas. Cada programa tem o objectivo de testar pequenas partes dos componentes do CPU através das respectivas instruções.

PROGRAMA 1

Código de simulação	Instruções	Codificações
476. code_memory[0x00] = 0x10A;	MOV A, 10	1 0000 1010 = 0x10A
477. code_memory[0x01] = 0x061;	MOV @P, A	0 0110 0001 = 0x061
478. code_memory[0x02] = 0x100;	MOV A, 0	1 0000 0000 = 0x100
479. code_memory[0x03] = 0x060;	MOV A, @P	0 0110 0000 = 0x060
480. code_memory[0x04] = 0x084;	JMP 0x04	0 1000 0100 = 0x084

PROGRAMA 2

Código de simulação	Instruções	Codificações
488. code_memory[0x00] = 0x10A;	MOV A, 10	1 0000 1010 = 0x10A
489. code_memory[0x01] = 0x062;	MOV P, A	0 0110 0010 = 0x062
490. code_memory[0x02] = 0x000;	MOV B, A	0 0000 0000 = 0x000
491. code_memory[0x03] = 0x083;	JMP 0x03	0 1000 0011 = 0x083

PROGRAMA 3

Código de simulação	Instruções	Codificações
500. code_memory[0x00] = 0x10A;	MOV A, 10	1 0000 1010 = 0x10A
501. code_memory[0x01] = 0x000;	MOV B, A	0 0000 0000 = 0x000
502. code_memory[0x02] = 0x002;	SUBB A, B	0 0000 0000 = 0x002
503. code_memory[0x03] = 0x045;	JZ 0x05	0 0100 0101 = 0x045
504. code_memory[0x08] = 0x088;	JMP 0x08	0 1000 1000 = 0x088

PROGRAMA 4

Código de simulação	Instruções	Codificações
514. code_memory[0x00] = 0x1F5;	MOV A, 245	1 1111 0101 = 0x1F5
515. code_memory[0x01] = 0x000;	MOB B, A	0 0000 0000 = 0x000
516. code_memory[0x02] = 0x1F5;	MOV A, 245	1 1111 0101 = 0x1F5
517. code_memory[0x03] = 0x001;	ADDC A, B	0 0000 0001 = 0x001
518. code_memory[0x04] = 0x025;	JC 0x05	0 0010 0101 = 0x025
519. code_memory[0x09] = 0x089;	JMP 0x09	0 1000 1001 = 0x089

CONCLUSÃO

A elaboração do presente projecto permitiu transmitir conhecimentos de um nível mais baixo que geralmente é abstraído do utilizador comum, no entanto estes sao conceitos fundamentais no funcionamento de um CPU.

Um ponto igualmente importante e’ a elaboração da aquitectura do CPU. As decisões tomadas na sua elaboração sao decisivas para a execução adequada de todas as instruções. Após a verificação e validação da arquitectura a concepção do código do CPU torna-se trivial, sendo a funcionalidade de cada componente traduzida de uma forma directa por funções de código.

BIBLIOGRAFIA

A. Computer’s CPU
   https://en.wikipedia.org/wiki/Central_processing_unit
B. Computer architecture
   https://en.wikipedia.org/wiki/Computer_architecture
C. Computer bus
   https://en.wikipedia.org/wiki/Bus_(computing)
D. Harvard architecture
   https://en.wikipedia.org/wiki/Harvard_architecture
E. EPROM
   https://en.wikipedia.org/wiki/EPROM

ANEXO – CÓDIGO DE SIMULAÇÃO EM ARDUINO

```
// EPROM: There are a total of 0x3F addresses so the
// proper size is 2^4 = 64
#define EPROM_SIZE 64

// Code memory: The JUMP instruction can have a max jump of
// 0 1111 1111 = 0x0FF = 255 so the proper size is 2^8 = 256
#define CODE_MEMORY_SIZE 256

// Data memory: Max bit in Address Bus is 9 so 2^9 = 512
#define DATA_MEMORY_SIZE 512

#define DEBOUNCE_TIME 200

// Arithemtic unit
byte alu_r;
byte alu_c;
byte alu_z;

// Program counter
byte pc_mux_y0;
word pc_add_y0;
word pc_mux_y1;
word pc_reg_d0;
volatile word pc_reg_q0;
boolean pc0_enable;

// Registers
byte a_mux_y0;
byte a_reg_d0;
byte a_reg_q0;
byte b_reg_d0;
volatile byte b_reg_q0;
byte p_reg_d0;
volatile byte p_reg_q0;
byte z_reg_d0;
byte z_reg_q0;
byte cy_reg_d0;
volatile byte cy_reg_q0;

// Control module
boolean a_enable;
boolean b_enable;
boolean f_enable;
boolean p_enable;

boolean pc1_enable;
boolean pc2_enable;
boolean pc3_enable;
boolean write;
boolean read;
boolean jump_carry;
boolean jump_zero;

```

```
// Blocks
word eprom[EPROM_SIZE];

word code_memory[CODE_MEMORY_SIZE];
word code_memory_db;

byte data_memory[DATA_MEMORY_SIZE];
byte data_memory_db;

// Clock
long time_pos;
long time_neg;

void setup() {
    Serial.begin(9600);
    initialize();
    attachInterrupt(digitalPinToInterrupt(2), MCLK_negative, FALLING);
    interrupts();
}

void initialize() {
    fill_eprom();
    fill_data_memory();
    program_3();
    print_instruction();
    read = true;
    write = true;
}

void loop() {
    functional_module();
    control_module();

    if (Serial.available()) {
        read_input();
    }
}

void read_input() {
    switch (Serial.read()) {
        case 'C':
            return print_control_module();
        case 'F':
            return print_functional_module();
        case 'M':
            return print_memory();
        }
    }

    void print_control_module() {
        Serial.print(" > C: ");
        Serial.print(" EnA ");
        Serial.print(a_enable, HEX);
        Serial.print(" EnB ");
        Serial.print(b_enable, HEX);
        Serial.print(" EnF ");
        Serial.print(f_enable, HEX);
```

```
111.        Serial.print(" EnP ");
112.        Serial.print(p_enable, HEX);
113.        Serial.print(" PC1 ");
114.        Serial.print(pc1_enable, HEX);
115.        Serial.print(" PC2 ");
116.        Serial.print(pc2_enable, HEX);
117.        Serial.print(" PC3 ");
118.        Serial.print(pc3_enable, HEX);
119.        Serial.print(" /WR ");
120.        Serial.print(write, HEX);
121.        Serial.print(" /RD ");
122.        Serial.print(read, HEX);
123.        Serial.print(" JC ");
124.        Serial.print(jump_carry, HEX);
125.        Serial.print(" JZ ");
126.        Serial.print(jump_zero, HEX);
127.        Serial.println();
128.    }
129.
130.    void print_functional_module() {
131.
132.    }
133.
134.    void print_memory() {
135.        Serial.print(" > Reg: ");
136.        Serial.print(" PC_Q 0x");
137.        Serial.print(pc_reg_q0, HEX);
138.        Serial.print(" A_Q 0x");
139.        Serial.print(a_reg_q0, HEX);
140.        Serial.print(" B_Q 0x");
141.        Serial.print(b_reg_q0, HEX);
142.        Serial.print(" P_Q 0x");
143.        Serial.print(p_reg_q0, HEX);
144.        Serial.print(" Cy_Q 0x");
145.        Serial.print(cy_reg_q0, HEX);
146.        Serial.print(" Z_Q 0x");
147.        Serial.print(z_reg_q0, HEX);
148.        Serial.println();
149.    }
150.
151.    void print_instruction() {
152.        print_memory();
153.        print_control_module();
154.
155.        word instruction = code_memory[pc_reg_q0];
156.
157.        Serial.print("0x0");
158.        Serial.print(pc_reg_q0);
159.        Serial.print(" ");
160.
161.        boolean D8 = read_bit(instruction, 8);
162.        boolean D7 = read_bit(instruction, 7);
163.        boolean D6 = read_bit(instruction, 6);
164.        boolean D5 = read_bit(instruction, 5);
165.        boolean D1 = read_bit(instruction, 1);
166.        boolean D0 = read_bit(instruction, 0);
167.
```

```
168.        if (D8) {
169.            Serial.print("MOV A, #CONST");
170.        }
171.        if (!D8 & D7) {
172.            Serial.print("JMP end7");
173.        }
174.        if (!D8 & !D7 & D6 & D5 & D1 & !D0) {
175.            Serial.print("MOV P, A");
176.        }
177.        if (!D8 & !D7 & D6 & D5 & !D1 & D0) {
178.            Serial.print("MOV @P, A");
179.        }
180.        if (!D8 & !D7 & D6 & D5 & !D1 & !D0) {
181.            Serial.print("MOV A, @P");
182.        }
183.        if (!D8 & !D7 & D6 & !D5) {
184.            Serial.print("JZ rel5");
185.        }
186.        if (!D8 & !D7 & !D6 & D5) {
187.            Serial.print("JC rel5");
188.        }
189.        if (!D8 & !D7 & !D6 & !D5 & D1) {
190.            Serial.print("SUBB A, B");
191.        }
192.        if (!D8 & !D7 & !D6 & !D5 & !D1 & D0) {
193.            Serial.print("ADDC A, B");
194.        }
195.        if (!D0 & !D1 & !D5 & !D6 & !D7 & !D8) {
196.            Serial.print("MOV B, A");
197.        }
198.
199.        Serial.print(" > ");
200.        Serial.print(instruction, BIN);
201.        Serial.print(" = ");
202.        Serial.print("0x0");
203.        Serial.print(instruction, HEX);
204.        Serial.println();
205.    }
206.
207.    /** * * * * * * * * * * * * * * * * * * * * * * * * * * *
208.     * Modules
209.     ** * * * * * * * * * * * * * * * * * * * * * * * * * * */
210.
211.    byte MUX_2x1(boolean S, byte A, byte B) {
212.        return S ? B : A;
213.    }
214.
215.    byte MUX_4x1(boolean S1, boolean S0, byte M3, byte M2, byte M1, byte
    M0) {
216.        if (!S1 & !S0) {
217.            return M0;
218.        }
219.        if (!S1 & S0) {
220.            return M1;
221.        }
222.        if (S1 & !S0) {
223.            return M2;
```

```
        }
        if (S1 & S0) {
            return M3;
        }
    }

    byte add(byte A, byte B) {
        return A + B;
    }

    byte sub(byte A, byte B) {
        return A - B;
    }

    byte register_memory(boolean E, byte D, byte Q) {
        return E ? D : Q;
    }

    // Arithmetic and Logic Unit
    byte alu(boolean Sel, byte X, byte Y, byte carry_in) {
        byte result;

        if (Sel) {
            byte aux = add(X, Y);
            result = add(aux, carry_in);
            alu_z = (byte) (X + Y + carry_in) == 0;
            alu_c = (int) (X + Y + carry_in) > 255;
        }
        else {
            byte aux = sub(X, Y);
            result = sub(aux, carry_in);
            alu_z = (byte) (X - Y - carry_in) == 0;
            alu_c = (int) (X - Y - carry_in) > 255;
        }

        return result;
    }

    boolean x_module(boolean jc, boolean c, boolean jz, boolean z) {
        return jc & c | jz & z;
    }

    void code_memory_block(word address_bus) {
        code_memory_db = code_memory[address_bus];
    }

    void data_memory_block(byte address_bus, boolean output_enable, boolean
write_enable) {
        if (!write_enable) {
            data_memory_db = a_reg_q0;
            data_memory[address_bus] = a_reg_q0;
        }

        if (!output_enable) {
            data_memory_db = data_memory[address_bus];
        }
    }
```

```
280.
281.    void functional_module() {
282.        // Code memory
283.        code_memory_block(pc_reg_q0);
284.
285.        // Data memory
286.        data_memory_block(p_reg_q0, read, write);
287.
288.        // Program counter
289.        // Mask bits D4, D3, D2, D1, D0
290.        byte x = code_memory_db & 0x01F;
291.
292.        if (read_bit(x, 4)) {
293.            x |= 0x0E0;
294.        }
295.
296.        pc_mux_y0 = MUX_2x1(pc0_enable, 1, x);
297.        pc_add_y0 = add(pc_mux_y0, pc_reg_q0);
298.        pc_mux_y1 = MUX_2x1(pc1_enable, pc_add_y0, code_memory_db & 0x07F);
299.        pc_reg_d0 = pc_mux_y1;
300.
301.        // A, B, P registers
302.        // Mask bits D7, D6, D5, D4, D3, D2, D1, D0
303.          a_mux_y0 = MUX_4x1(pc3_enable, pc2_enable, 0, code_memory_db &
    0x0FF, data_memory_db, alu_r);
304.
305.        a_reg_d0 = a_mux_y0;
306.        b_reg_d0 = a_reg_q0;
307.        p_reg_d0 = a_reg_q0;
308.        z_reg_d0 = alu_z;
309.        cy_reg_d0 = alu_c;
310.
311.        // ALU
312.          alu_r = alu((code_memory_db & 0x01) == 1, a_reg_q0, b_reg_q0,
    alu_c);
313.
314.        // X module
315.        pc0_enable = x_module(jump_carry, cy_reg_q0, jump_zero, z_reg_q0);
316.    }
317.
318.    void control_module() {
319.        // Control module input are the bits:
320.        // D8 D7 D6 D5 !D4 !D3 !D2 D1 D0 = 1 1110 0011 = 0x1E3
321.        int input_address = code_memory_db & 0x1E3;
322.
323.        // Form a new address with just
324.        // D8 D7 D6 D5 D1 D0 bits
325.        // just like in EPROM table.
326.        input_address = (
327.                (input_address & 0x100) >> 3 |
328.                (input_address & 0x080) >> 3 |
329.                (input_address & 0x040) >> 3 |
330.                (input_address & 0x020) >> 3|
331.                input_address & 0x002 |
332.                input_address & 0x001
333.        );
334.
```

```
335.        // input_address = (
336.        //         input_address & 0x1E0 >> 3 |
337.        //         input_address & 0x002 |
338.        //         input_address & 0x001
339.        // );
340.
341.        // Read the data at the eprom input address
342.        int data = eprom[input_address];
343.
344.        // Read individual bits of data
345.        a_enable = read_bit(data, 10);
346.        b_enable = read_bit(data, 9);
347.        f_enable = read_bit(data, 8);
348.        p_enable = read_bit(data, 7);
349.        pc1_enable = read_bit(data, 6);
350.        pc2_enable = read_bit(data, 5);
351.        pc3_enable = read_bit(data, 4);
352.        write = read_bit(data, 3);
353.        read = read_bit(data, 2);
354.        jump_carry = read_bit(data, 1);
355.        jump_zero = read_bit(data, 0);
356.    }
357.
358.    boolean read_bit(word bits, byte n) {
359.        // Shift n positions to the right
360.        bits = bits >> n;
361.        // Filter the last bit of the right
362.        bits = bits & 0x01;
363.        return bits == 0x01;
364.    }
365.
366.    void fill_eprom() {
367.        fill(eprom, 0x00, 0x00, 0x20C); // MOV B, A
368.        fill(eprom, 0x01, 0x01, 0x50C); // ADDC A, B
369.        fill(eprom, 0x02, 0x02, 0x50C); // SUBB A, B
370.        fill(eprom, 0x04, 0x07, 0x00E); // JC rel5
371.        fill(eprom, 0x08, 0x0B, 0x00D); // JZ rel5
372.        fill(eprom, 0x0C, 0x0C, 0x428); // MOV A, @P
373.        fill(eprom, 0x0D, 0x0D, 0x004); // MOV @P, A
374.        fill(eprom, 0x0E, 0x0E, 0x08C); // MOV P, A
375.        fill(eprom, 0x10, 0x1F, 0x04C); // JMP end7
376.        fill(eprom, 0x20, 0x3F, 0x41C); // MOV A, #CONST
377.    }
378.
379.    void fill(word array[], word from, word to, word value) {
380.        for (int i = from; i <= to; i++) {
381.            array[i] = value;
382.        }
383.    }
384.
385.    void fill_data_memory() {
386.        for (int i = 0x00; i < DATA_MEMORY_SIZE; i++) {
387.            data_memory[i] = random(0x00, 0x0FF);
388.        }
389.    }
390.
391.    /** * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
 * Clock
 ** * * * * * * * * * * * * * * * * * * * * * * * * * * * */

void MCLK_positive() {
    long time = millis();

    if (time - time_pos < DEBOUNCE_TIME) {
        return;
    }

    if (true) {
        pc_reg_q0 = pc_reg_d0;
    }

    print_instruction();

    attachInterrupt(digitalPinToInterrupt(2), MCLK_negative, FALLING);
    time_pos = time;
}

void MCLK_negative() {
    long time = millis();

    if (time - time_neg < DEBOUNCE_TIME) {
        return;
    }

    a_reg_q0 = register_memory(a_enable, a_mux_y0, a_reg_q0);
    b_reg_q0 = register_memory(b_enable, a_reg_q0, b_reg_q0);
    p_reg_q0 = register_memory(p_enable, a_reg_q0, p_reg_q0);
    z_reg_q0 = register_memory(f_enable, z_reg_d0, z_reg_q0);
    cy_reg_q0 = register_memory(f_enable, cy_reg_d0, cy_reg_q0);

    attachInterrupt(digitalPinToInterrupt(2), MCLK_positive, RISING);
    time_neg = time;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Programs
 ** * * * * * * * * * * * * * * * * * * * * * * * * * * * */

void code_memory_test() {
    // Instructions from codification table

    // Arbitrary parameters values for test purposes:
    // Dashes (-) are the instructions bits
    // #CONST           = - 1110 1011 = 0x0EB = 235
    // rel5             = - --00 0101 = 0x005 = 005
    // end7             = - -000 1001 = 0x009 = 009

    // end7             = - -000 1000 = 0x008 = 008

    // Instructions codification with parameters values:
    // MOV A, #CONST    > 1 1110 1011 = 0x1EB
    // JC rel5          > 0 0010 0101 = 0x025
    // JZ rel5          > 0 0100 0101 = 0x045
    // JMP end7         > 0 1000 1001 = 0x089
```

```
449.
450.        // JMP end7         > 0 1000 1000 = 0x088
451.
452.        // The last instruction is a JMP to the same address: HALT
453.
454.        // Instructions:
455.        code_memory[0x00] = 0x1EB; // MOV A, #CONST
456.        code_memory[0x01] = 0x061; // MOV @P, A
457.        code_memory[0x02] = 0x060; // MOV A, @P
458.        code_memory[0x03] = 0x062; // MOV P, A
459.        code_memory[0x04] = 0x000; // MOV B, A
460.        code_memory[0x05] = 0x001; // ADDC A, B
461.        code_memory[0x06] = 0x002; // SUBB A, B
462.        code_memory[0x07] = 0x025; // JC rel5
463.        code_memory[0x08] = 0x045; // JZ rel5
464.        code_memory[0x09] = 0x089; // JMP end7
465.        // Jumps to go back
466.        code_memory[0x0C] = 0x088; // JMP end7
467.        code_memory[0x0D] = 0x089; // JMP end7
468.    }
469.
470.    void program_1() {
471.        // MOV A, 10    = 1 0000 1010 = 0x10A
472.        // MOV @P, A
473.        // MOV A, 0     = 1 0000 0000 = 0x100
474.        // MOV A, @P
475.        // JMP 0x04     = 0 1000 0100 = 0x084
476.        code_memory[0x00] = 0x10A;
477.        code_memory[0x01] = 0x061;
478.        code_memory[0x02] = 0x100;
479.        code_memory[0x03] = 0x060;
480.        code_memory[0x04] = 0x084;
481.    }
482.
483.    void program_2() {
484.        // MOV A, 10    = 1 0000 1010 = 0x10A
485.        // MOV P, A
486.        // MOV B, A
487.        // JMP 0x03     = 0 1000 0011 = 0x083
488.        code_memory[0x00] = 0x10A;
489.        code_memory[0x01] = 0x062;
490.        code_memory[0x02] = 0x000;
491.        code_memory[0x03] = 0x083;
492.    }
493.
494.    void program_3() {
495.        // MOV A, 10    = 1 0000 1010 = 0x10A
496.        // MOV B, A
497.        // SUBB A, B
498.        // JZ 0x05      = 0 0100 0101 = 0x045
499.        // JMP 0x08     = 0 1000 1000 = 0x088
500.        code_memory[0x00] = 0x10A;
501.        code_memory[0x01] = 0x000;
502.        code_memory[0x02] = 0x002;
503.        code_memory[0x03] = 0x045;
504.        code_memory[0x08] = 0x088;
505.    }
```

```
506.
507.    void program_4() {
508.        // MOV A, 245   = 1 1111 0101 = 0x1F5
509.        // MOV B, A
510.        // MOV A, 245   = 1 1111 0101 = 0x1F5
511.        // ADDC A, B
512.        // JC 0x05      = 0 0010 0101 = 0x025
513.        // JMP 0x09     = 0 1000 1001 = 0x089
514.        code_memory[0x00] = 0x1F5;
515.        code_memory[0x01] = 0x000;
516.        code_memory[0x02] = 0x1F5;
517.        code_memory[0x03] = 0x001;
518.        code_memory[0x04] = 0x025;
519.        code_memory[0x09] = 0x089;
520.    }
```