INSTITUTO SUPERIOR de ENGENHARIA de LISBOA

Licenciatura em Engenharia Informática e Multimédia

2.º Semestre Letivo 2020/2021

Computação Física 2.º Trabalho Prático

Grupo nº: 7

Alunos:

Duarte Domingues nº 45140

Rui Correa nº 42156

Índice

Índice de figuras

Objetivos

Este trabalho prático tem como objetivo o desenvolvimento dum microprocessador baseado na arquitetura de Harvard. O microprocessador irá ser realizado a partir de dispositivos *hardware* estudados e implementado no trabalho prático anterior, como por exemplo, *multiplexers*, registos e comparadores.

A tabela seguinte ilustra as diferentes instruções que o microprocessador irá executar:

Instrução	Funcionalidade
MOV A,#constante6	A = constante6
MOV A, Rn	A = Rn
MOV Rn, A	Rn = A
CLRC	C = 0
NOT A	A = A/
AND A, Rn	A = A and Rn
OR A, Rn	A = A or Rn
SUBB A, Rn	A = A – Rn - C
ADDC A, Rn	A = A + Rn + C
MOV A, @Rn	A = M(Rn)
MOV @Rn, A	M(Rn) = A
JM rel5	Se (Sinal) PC+= rel5
JP rel5	Se (Sinal/) PC+= rel5
JC rel5	Se (C) PC+= rel5
JNC rel5	Se (C/) PC+= rel5
JZ rel5	Se (Zero) PC+= rel5
JNZ rel5	Se (Zero/) PC+= rel5
JMP end6	PC= end6

Figura 1 - Conjunto de instruções do CPU

Introdução

Um microprocessador é uma máquina *hardware* que executa instruções lidas sequencialmente da memória de código. Uma instrução é uma ação ou conjunto de ações que desencadeiam uma transferência de informação, como por exemplo, uma operação aritmética ou uma operação lógica.

O microprocessador implementado baseia-se na arquitetura de Harvard, neste tipo de arquitetura os dados e os códigos estão em memórias diferentes. O microprocessador poderia também ter sido desenhado para integrar uma arquitetura Von Newman, uma arquitetura em que os dados e códigos estão misturados numa memória física.

A maior vantagem de ter *buses* separados para instruções e para dados é o CPU conseguir aceder instruções e ler / escrever dados simultaneamente.

A memória de código é responsável por guardar as diferentes instruções dos programas e a memória de dados as diferentes variáveis.

Os *buses* são um conjunto de linhas de comunicação que permitem a comunicação entre dispositivos, como o CPU e as memórias, para as diferentes memórias os buses têm as seguintes funcionalidades:

Memória de dados:

- *Data bus* - carrega dados entre a memória de dados e o processador.
- *Address bus* – carrega os endereços dos dados do processador para a memória de dados.

Memória de código:

- *Data bus* – carrega instruções da memória de código para o processador.
- *Address bus* – carrega os endereços das instruções do processador para a memória de código.

O microprocessador irá à parte das memórias e da unidade central de processamento, composta por diferentes registos, ter uma máquina de unidade aritmética e lógica (ALU) e uma unidade de controlo.

A ALU é responsável por realizar as operações aritméticas e lógicas do microprocessador como a soma de dois valores.

A unidade de controlo permite controlar todos os sinais de controlo do microprocessador, controlando o movimento de instruções e dados dentro do sistema.

Desenvolvimento

Quantidade de bits de cada registo

O CPU tem registos internos, que permitem armazenar informação. Os diferentes registos utilizados no microprocessador são implementados à custa de flip-flops do tipo D, tendo uma entrada D e uma saída Q.

Cada registo tem associado um sinal de *enable*, este sinal permite especificar se é possível ou não guardar informação no registo.

Na tabela seguinte pode ser observada a quantidade de bits que os diferentes registos do sistema permitem armazenar.

Registos	Número de bits
Registo RN 0	6 bits
Registo RN 1	6 bits
Registo A	6 bits
Program Counter	6 bits
***Flags* C, Z, S**	1 bit

Figura 2 - Quantidade de bits dos registos

- O registo A tem que ter 6 bits devido à instrução MOV A, #constante6 (Registo A fica com o valor da #constante6), sendo constante6 um número a 6 bits.
- Os registos Rn (R0 e R1) tem que ter 6 bits cada devido à instrução MOV RN, A (O registo Rn fica com o valor do registo A), pois está a ser movido para um dos registos RN um valor a 6 bits.
- Program Counter tem que ter tamanho de 6 bits devido à instrução (PC = *end6*), sendo que nesta instrução está a ser movido um número absoluto a 6 bits para o registo PC.
- As *flags* têm apenas um bit, pois cada uma só pode ter dois valores possíveis (0 ou 1).

Quantidade de bits dos *Address Bus* e *Data Bus* das memórias de dados e de código

Módulo da memória de dados

- *Adress Bus* – 6 bits, tem que ser igual ao conjunto de bits do registo RN.
- *Data Bus* – 6 bits, tem que ser igual ao conjunto de bits do registo A.

A memória de dados tem que ter dimensão 64 (2^6), pois o seu endereçamento está a ser realizado a 6 bits, com 6 bits é possível realizar 64 combinações possíveis. Dentro da memória de dados irão estar presentes valores a 6 bits, pois na memória de dados é guardado o valor presente nos registos A e Rn, sendo estes valores representados a 6 bits.

Módulo da memória de código

- *Address Bus* – 6 bits, tem que ser igual ao conjunto de bits do *Program Counter*.
- *Data Bus* – 9 bits, tem que ser igual ao tamanho da codificação das instruções.

Semelhantemente à memória de dados, a memória de código tem que ter dimensão 64 (2^6), pois o seu endereçamento está a ser realizado a 6 bits. Em contraste à memória de dados, os valores presentes na memória de código vão ser a 9 bits, pois a memória de código guarda as instruções do programa, que foram codificadas a 9 bits.

Na figura seguinte pode-se observar o esquema representativo da arquitetura realizada para o microprocessador baseado na arquitetura de Havard.

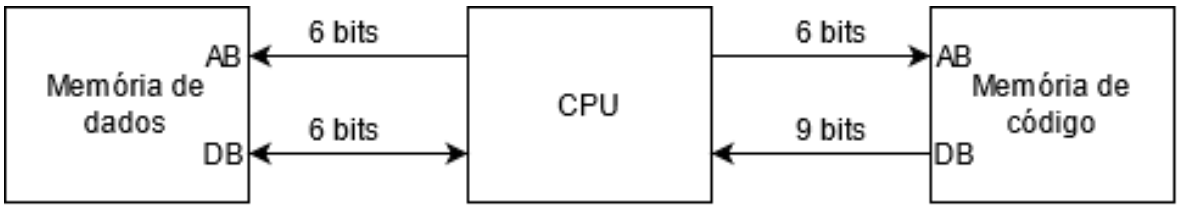

Figura 3 - Arquitetura do microprocessador

Codificação das instruções

Para que o CPU consiga executar diferentes instruções é necessário codificar cada instrução com uma combinação única de bits, de modo, a cada instrução ter uma identificação única no CPU.

As instruções do CPU dividem-se normalmente nas seguintes classes:

- **Instruções de transferência de informação** - Estas instruções são responsáveis pelas operações de leitura e de escrita em registos, contadores, *flags* e a leitura e a alteração de valores em dispositivos exteriores ao CPU, como a memória de dados.
- **Instruções Aritméticas ou lógicas**- Estas instruções são responsáveis por executar as operações aritméticas ou lógicas do CPU, sendo realizadas na ALU.
- **Instruções de Controlo** - Estas instruções servem para alterar a ordem sequencial de execução de instruções no CPU.

As instruções foram codificadas com nove bits, o menor número de bits possível, de forma a reduzir ao máximo o número de bits necessários para o *data bus* da memória de código.

Tentou-se codificar as diferentes instruções com o menor número de bits de distinção entre instruções, de modo, a minimizar o número de bits necessários para o módulo de controlo. Bits de distinção de instruções foram: D8, D7, D6, D5, D0.

Pode-se observar a codificação das instruções na tabela seguinte.

Instruções	Parâmetros	D8	D7	D6	D5	D4	D3	D2	D1	D0
MOV A,#constante6	#const6	1	1	1	C5	C4	C3	C2	C1	C0
MOV A, Rn	RN	0	0	0	0	0	RN	0	0	0
MOV Rn, A	RN	0	0	0	0	0	RN	0	0	1
CLRC		0	0	0	1	0	0	0	0	0
NOT A		0	0	0	1	0	0	0	0	1
AND A, Rn	RN	0	0	1	0	0	RN	0	0	0
OR A, Rn	RN	0	0	1	0	0	RN	0	0	1
SUBB A, Rn	RN	0	0	1	1	0	RN	0	0	0
ADDC A, Rn	RN	0	0	1	1	0	RN	0	0	1
MOV A, @Rn	RN	0	1	0	0	0	RN	0	0	0
MOV @Rn, A	RN	0	1	0	0	0	RN	0	0	1
JM rel5	rel5	0	1	0	1	R4	R3	R2	R1	R0
JP rel5	rel5	0	1	1	0	R4	R3	R2	R1	R0
JC rel5	rel5	0	1	1	1	R4	R3	R2	R1	R0
JNC rel5	rel5	1	0	0	0	R4	R3	R2	R1	R0
JZ rel5	rel5	1	0	0	1	R4	R3	R2	R1	R0
JNZ rel5	rel5	1	0	1	0	R4	R3	R2	R1	R0
JMP end6	end6	1	1	0	E5	E4	E3	E2	E1	E0

Figura 4 - Codificação das instruções

Módulo funcional baseado na técnica de encaminhamento de dados

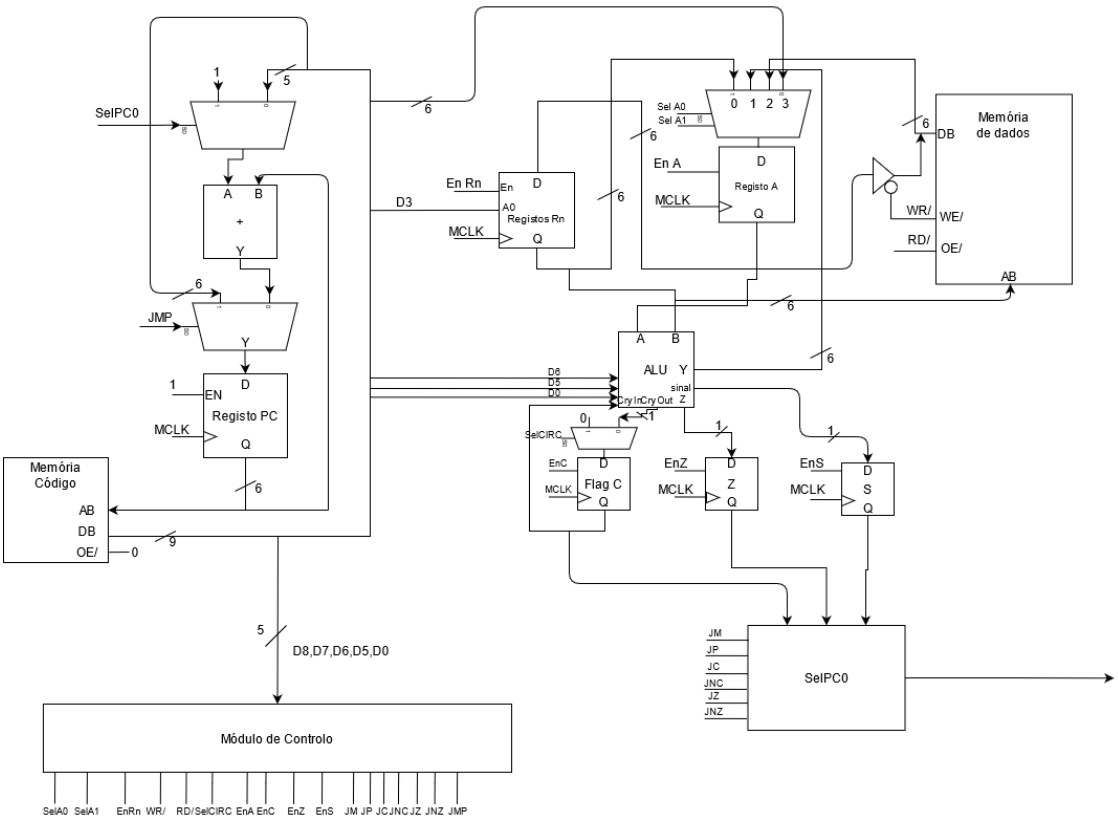

Figura 5 - Módulo funcional

Na figura apresentada acima pode-se observar o módulo funcional do CPU. O módulo funcional é um diagrama de blocos constituído por todos os dispositivos hardware disponíveis no CPU, neste caso são necessários os seguintes componentes:

- Memória de código
- Memória de dados
- Conjunto de registos RN
- Registo A
- ALU
- *Flags* C, Z e S
- *Multiplexers*
- *Demultiplexers*
- Contador
- Módulo de controlo
- Módulo *Program Counter*
- Módulo complementar designado por SelPC0

O módulo funcional foi construído baseado nas diferentes instruções que o CPU tem que realizar, de acordo com a técnica de encaminhamento de dados.

Program Counter

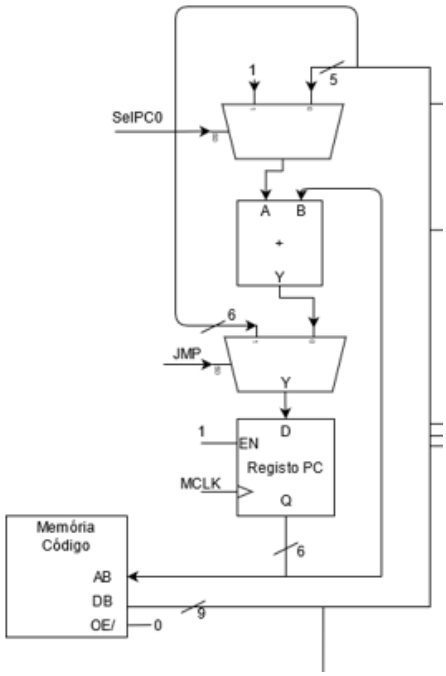

Figura 6 - Módulo Program Counter

O módulo *Program Counter* estabelece o valor do *address* bus da memória de código. O *Program Counter* tem um contador para incrementar ou decrementar as especificações da memória de código e um registo para guardar a posição atual.

A determinação dum novo valor no registo *Program Counter* acontece na transição ascendente do CLCK, sendo que os restantes registos do CPU só vão registar informação na transição descendente do CLCK.

O CPU programado tem um conjunto de instruções do tipo "*jump*" que permitem alterar a execução sequencial das instruções.

De modo a executar a maioria das instruções "*jump*", respetivamente, JM, JP, JC, JNC, JZ e JNZ é necessário um *multiplexer* que segundo um bit seletor SelPC0, realiza a soma do valor atual do registo PC com o valor um, ou então, soma o valor atual do registo PC com um valor rel5. O valor do SelPC0 é definido por um circuito lógico, que é influenciado pelas *flags* C, Z e S e valores de sinais ativos do módulo de controlo.

A instrução JMP, ao contrário das outras instruções "*jump*", atribui um valor absoluto ao registo *Program Counter*, logo irá ser delegado um novo endereço de forma absoluta à memória de código. Para a implementação do JMP, foi necessário um *multiplexer* que segundo um bit seletor JMP, que é uma saída

do módulo de controlo, permite selecionar entre um valor proveniente do somador ou um valor absoluto a 6 bits.

O módulo responsável por original o valor para SelPC0 utilizado no *multiplexer* mencionado anteriormente tem a seguinte representação no módulo funcional e esquema lógico.

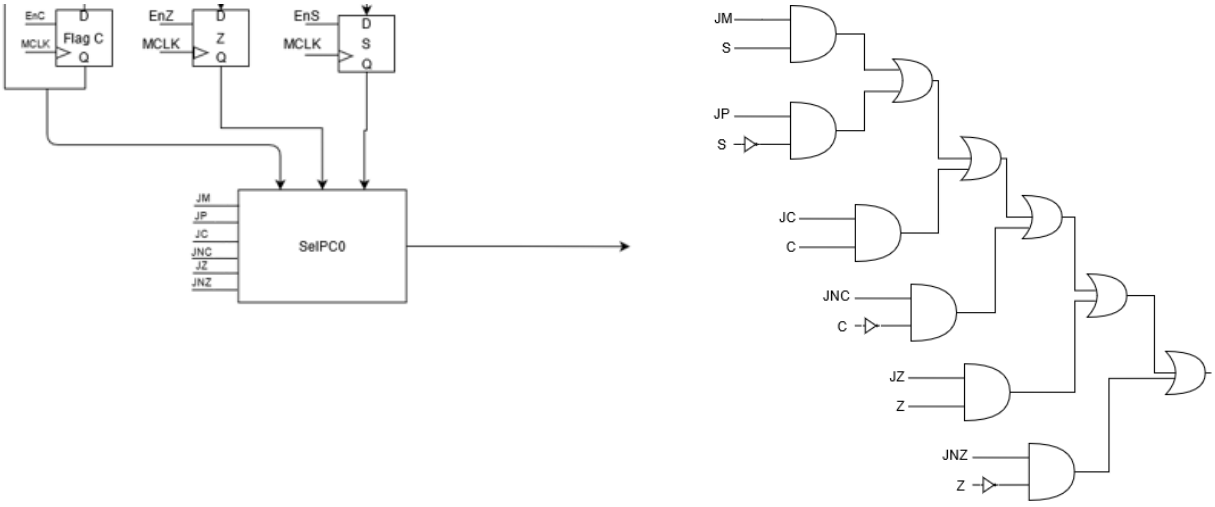

Figura 7 - Módulo SelPC0

Registos

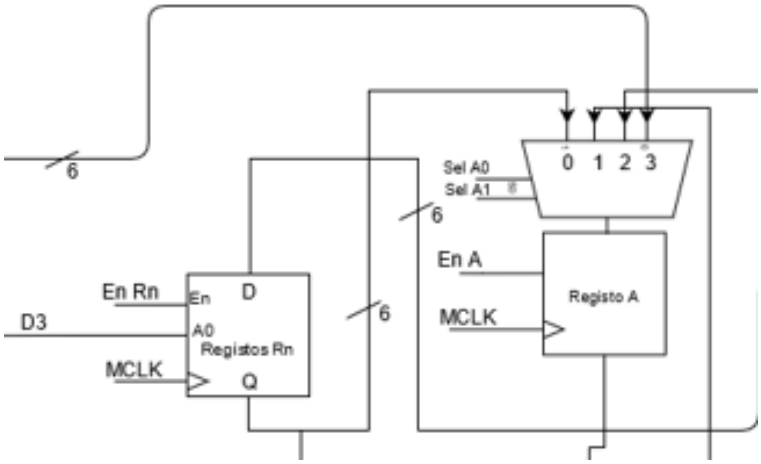

Figura 8 - Conjunto de registos

Para selecionar os valores de entrada para o registo A é necessário um *multiplexer* 4x2. A transferência de dados entre os registos A e o conjunto de registos RN e a leitura e escrita na memória de dados são realizadas a partir de instruções do tipo MOV.

O registo específico (0 ou 1) do conjunto RN é selecionado consoante o valor do bit D3 obtido do *data bus* da memória de código.

Memória de dados

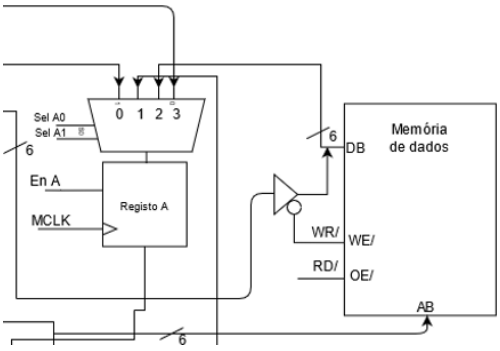

Figura 9 - Escrita e leitura da memória de dados

A memoria de dados permite guardar as diferentes variáveis presentes no sistema, esta memória é do tipo RAM. Memória RAM (*Random Acess Memory*) é um espaço de armazenamento de dados permitindo escrita e leitura.

Na figura acima pode-se observar a parte módulo funcional responsável por leitura e escrita na memória de dados. No caso de se querer escrever na memória de dados o sinal WR/ tem que estar a zero, assim como quando se quer ler da memória de dados o sinal RD/ tem que estar a zero.

ALU

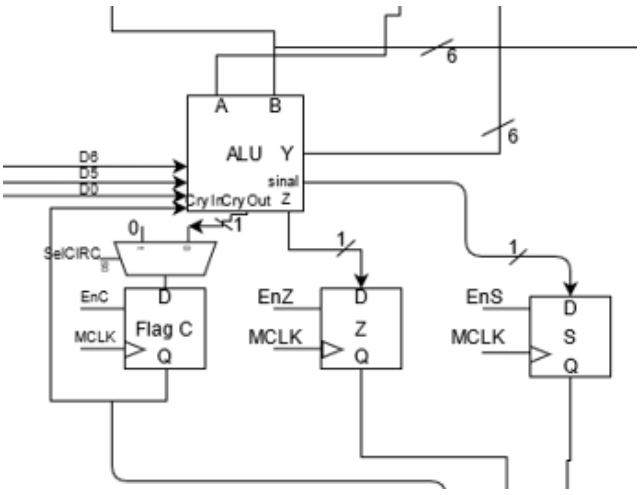

Figura 10 - ALU

A ALU projetada implementa cinco operações, três lógicas e duas aritméticas, necessitando então três bits de seleção para determinar qual operação a ser realizada. A ALU tem duas entradas, a entrada A saída do registo A e a entrada B saída do conjunto de registos RN.

Estão presentes três *flags*, respetivamente:

- *Flag* C - *Flag* de indicador de erro, indica quando o limite de bits possível é excedido.
- *Flag* Z – Fica ativa se o resultado for igual ao zero.
- *Flag* Sinal – *Flag* indicadora de sinal.

Módulo de controlo

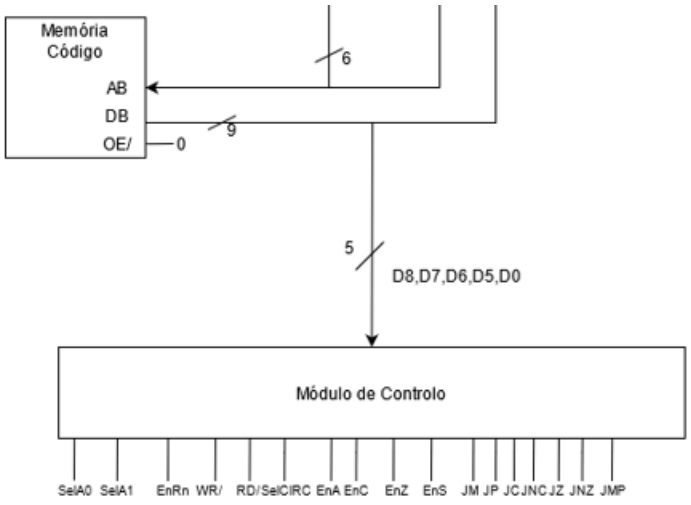

Figura 11 - Módulo de controlo

O módulo de controlo permite a ativação coordenada dos sinais de controlos dos dispositivos hardware presentes no módulo funcional.

O número de bits utilizado no módulo de controlo é resultado do número de bits usado para distinção entre instruções na etapa de codificação das instruções, portanto tem 5 bits (D8, D7, D6, D5, D0). Através destes bits temos a informação dos sinais que necessitam de estar ativos consoante uma instrução especifica.

De seguida vão ser especificados os sinais ativos para as diversas instruções no módulo de controlo, no formato de uma tabela.

Especificação das entradas e saídas do módulo de controlo

O módulo de controlo recebe as instruções dos programas a 9 bits a partir do *data bus* da memória de código. Consoante os bits de distinção das instruções recebidas, são especificados os sinais ativos no módulo de controlo, de forma aos componentes físicos do CPU realizarem os respetivos programas associados às instruções corretamente.

Esta tabela permite associar os bits de instrução com as respetivas instruções.

Instruções	D8	D7	D6	D5	D0	Sinais ativos
MOV A, #constante6	1	1	1	-	-	EnA, SelA0, SelA1
MOV A, Rn	0	0	0	0	0	EnA
MOV Rn, A	0	0	0	0	1	EnRn
CLRC	0	0	0	1	0	SelClrC, EnC
NOT A	0	0	0	1	1	EnA, SelA0 , EnZ, EnS
AND A, Rn	0	0	1	0	0	EnA, SelA0 , EnZ , EnS
OR A, Rn	0	0	1	0	1	EnA, SelA0, EnZ , EnS
SUBB A, Rn	0	0	1	1	0	EnA, SelA0 , EnZ, EnC, EnS
ADDC A, Rn	0	0	1	1	1	EnA, SelA0 , EnZ, EnC, EnS
MOV A, @Rn	0	1	0	0	0	EnA, SelA1 , RD/
MOV @Rn, A	0	1	0	0	1	WR/
JM rel5	0	1	0	1	-	JM
JP rel5	0	1	1	0	-	JP
JC rel5	0	1	1	1	-	JC
JNC rel5	1	0	0	0	-	JNC
JZ rel5	1	0	0	1	-	JZ
JNZ rel5	1	0	1	0	-	JNZ
JMP end6	1	1	0	-	-	JMP

Figura 12 - Tabela de entrada e saídas do módulo de controlo

Tabela EPROM

De seguida à realização da especificação das entradas e saídas do módulo de controlo é necessário definir a tabela EPROM, com a dimensão 32 x 16.

A memória do módulo de controlo é do tipo ROM (*Read Only Memory*), ou seja, apenas é possível ler dados desta memória. Em memórias ROM os valores memorizados são sempre os mesmos, até após ser desligada a alimentação.

Nesta tabela são definidos os intervalos dos endereços na memória da EPROM para cada instrução, representados em hexadecimal, e também a interpretação dos valores dos sinais de saída em hexadecimal.

No caso em que os bits de instrução não especificam nenhuma instrução realizada pelo CPU, apenas os sinais WR/ e RD/ ficam ativos.

	D8	D7	D6	D5	d0		SA0	SA1	EnRn	WR/	RD/	SClrC	EnA	EnC	EnZ	JM	JP	JC	JNC	JZ	JNZ	JMP	
INSTRUC…	A4	A3	A2	A1	A0	address	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	Data
MOV A, Rn	0	0	0	0	0	{0h}	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	1A00h
MOV Rn, A	0	0	0	0	1	{1h}	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	3800h
CLRC	0	0	0	1	0	{2h}	0	0	0	1	1	1	0	1	0	0	0	0	0	0	0	0	1D00h
NOT A	0	0	0	1	1	{3h}	1	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0	9A80h
AND A , Rn	0	0	1	0	0	{4h}	1	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0	9A80h
OR A, Rn	0	0	1	0	1	{5h}	1	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0	9A80h
SUBB A, Rn	0	0	1	1	0	{6h}	1	0	0	1	1	0	1	1	1	0	0	0	0	0	0	0	9B80h
ADDC A,RN	0	0	1	1	1	{7h}	1	0	0	1	1	0	1	1	1	0	0	0	0	0	0	0	9B80h
MOV A, @RN	0	1	0	0	0	{8h}	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	5200h
MOV @Rn, A	0	1	0	0	1	{9h}	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0800h
JM rel5	0	1	0	1	-	{Ah, Bh}	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0	1840h
JP rel5	0	1	1	0	-	{CH, DH}	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0	1820h
JC rel5	0	1	1	1	-	{EH, FH}	0	0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	1810h
JNC rel5	1	0	0	0	-	{10H,11H}	0	0	0	1	1	0	0	0	0	0	0	0	1	0	0	0	1808h
JZ rel5	1	0	0	1	-	{12H,13H}	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	1804h
JNZ rel5	1	0	1	0	-	{14H,15H}	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	1802h
JMP end6	1	1	0	-	-	{18H,1BH}	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1801h
MOV A, const6	1	1	1	-	-	{1CH,1FH}	1	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	DA00h
None	1	0	1	1	-	{16H,17H}	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1800h

Figura 13 - Tabela EPROM

Simulação da arquitetura desenhada no arduino

A partir da arquitetura mencionada anteriormente, foi realizada a simulação do microprocessador em arduino, utilizando o simulador TinkerCad.

Foi realizada a seguinte montagem para o sistema projetado, foi utilizado um botão de forma a incrementar o sinal de *CLCK* progressivamente.

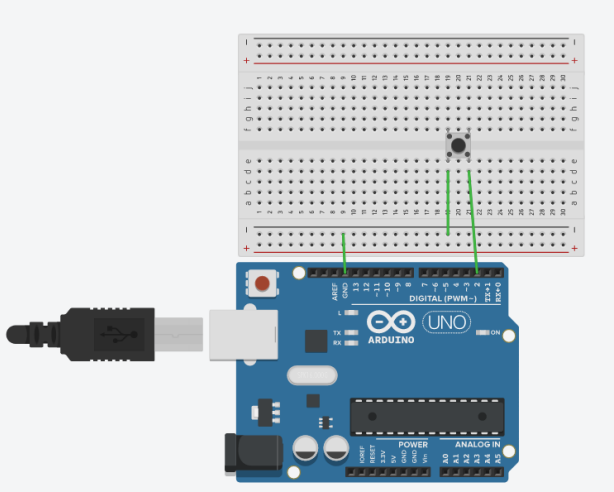

Figura 14 - Montagem realizada na simulação em arduino

A simulação do CPU foi baseada em todo o processo desenvolvido anteriormente, tendo sido realizada uma implementação de todos os componentes do microprocessador com base no modelo funcional projetado.

Um dos principais aspetos que teve que ser tido em atenção na realização simulação do CPU foi a representação dos valores com o número de bits correto. Os valores dos registos Program Counter, por exemplo, são a 6 bits, por isso para representar estes valores utilizou-se o byte, visto que é o data type mais próximo de 6 bits que pode ser utilizado. Portanto para verificar que os valores não ultrapassam os 6 bits é utilizada a seguinte máscara bitwise:

```
//mux1
byte end_six = mem_codigo_db & 0x3F;
```

Foi realizada uma função chamada módulo funcional que implementa todos os conceitos especificados no módulo de controlo. Esta função também é responsável por lidar com a possibilidade de números negativos para o valor rel5.

```
//------------------------- MODELO FUNCIONAL --------------------------

void modulo_funcional(){

  //data-bus da memoria de codigo
  mem_codigo_db = mem_codigo[reg_pc_q];


  //rel5

  //aplicar mascara 11111, para ficar com apenas os 5 bits menos significantes
  byte rel5 = mem_codigo_db & 0x01F;
  // se o numero for negativo, adicionar 1's a frente
  if (bitRead(rel5, 4) == 1){

    rel5 = rel5 | 0x20;

  }

  //defenir EN0 e o EN1
  byte demux_output = DEMUX_2x1(enRn,D3_sinal);
  en_r0 = bitRead(demux_output,0);
  en_r1 = bitRead(demux_output,1);

  //defenir entradas dos registos RN
  reg_rn_r0_d = reg_a_q;
  reg_rn_r1_d = reg_a_q;

  //defenir saida do RN
   byte reg_rn_q = MUX_2x1(D3_sinal, reg_rn_r0_q, reg_rn_r1_q);

  //porta tri_state

  if(!wr){
    mem_dados[reg_rn_q] = reg_a_q;
    mem_dados_db = reg_a_q;
  }

 else {
    mem_dados_db = mem_dados[reg_rn_q];
 }

 byte const_six = mem_codigo_db & 0x3F;

 //multiplexer do registo A
 y_reg_a_mux = MUX_4x1(selA0,selA1,reg_rn_q ,alu_Y ,mem_dados_db , const_six );
 reg_a_d = y_reg_a_mux;

 //ALU
 alu_Y = alu(D6_sinal,D5_sinal,D0_sinal, reg_a_q, reg_rn_q, alu_cryIn);

 //MUX CLRC
 y_clear_mux = MUX_2x1(selClrC, alu_cryOut,0);

 //atribuir o entrada dos registos das flags
 reg_c_d = y_clear_mux;
 reg_z_d = alu_z;
 reg_s_d = alu_sinal;

 //PC
 //Bit sel do mux0
 selPc0 = moduloSelPC0 ();
 //mux0

 y_pc_mux_0 = MUX_2x1(!selPc0, rel5, 1);
 //Serial.print(y_pc_mux_0);

 //adder
 adder_pc = (y_pc_mux_0 + reg_pc_q) & 0x3F;

 //mux1
 byte end_six = mem_codigo_db & 0x3F;
 y_pc_mux_1 = MUX_2x1(jmp,adder_pc,end_six );

 //registo pc
 reg_pc_d = y_pc_mux_1 & 0x3F;  }
```

Programas de teste

De forma a testar o correto funcionamento do microprocessador foram realizados programas de forma a testar as diversas partes dos componentes do CPU através das respetivas instruções

Programa 1

Programa para testar os registos internos do CPU, a leitura e escrita da memória de dados e o segundo *multiplexer* do módulo *Program Counter*.

```
void programa_teste_1(){

  //mov A, 12(0xC)
  mem_codigo[0x00] = 0x1CC;
  //MOV @rn,a          R=0
  mem_codigo[0x01] = 0x81;
  //MOV A, 2
  mem_codigo[0x02] = 0x1C2;
  //MOV A,@RN          R=0
  mem_codigo[0x03] = 0x80;
  //JMP end6 3
  mem_codigo[0x04] = 0x183;

}
```

0x1CC = 111001100

0X81 = 10000001

0X1C2 = 111000010

0X80 = 10000000

0X183 = 110000011

Programa 2

Programa para testar o correto funcionamento da ALU e do módulo SelPCO.

```
void programa_teste_2(){

  //MOV A, 8(0x8)
  mem_codigo[0x00] = 0x1C8;
  //MOV RN, A       R=0
  mem_codigo[0x01] = 0x01;
  //AND A, RN       R=0
  mem_codigo[0x02] = 0x40;
  //NOT A, RN       R=0
  mem_codigo[0x03] = 0x21;
  //OR A, RN        R=0
  mem_codigo[0x04] = 0x41;
  //ADD A, RN       R=0
  mem_codigo[0x05] = 0x61;
  //CLR C
  mem_codigo[0x06] = 0x20;
  //SUBB A, RN      R=0
  mem_codigo[0x07] = 0x60;
  //JC
  mem_codigo[0x08] = 0xE0;
  //JZ
  mem_codigo[0x09] = 0x120;
  //JM  010110100
  mem_codigo[0xA] = 0xB4;

}
```

0x1C8 = 111001000

0X01 = 000000001

0X40 = 001000000

0X21 = 000100001

0X41 = 001000001

0X61 = 001100001

0X20 = 000100000

0X60 = 011000000

0XE0 = 011100000

0X120 = 100100000

0XB4 = 10110100

Programa 3

Programa para testar o correto funcionamento do registo 1 do conjunto RN, teste das instruções do tipo JMP e da instrução vazia.

```
//MOV A 15(0xF)
mem_codigo[0x00] = 0x1CF;
//MOV RN,A      R=1
mem_codigo[0x01] = 0x09;
//MOV @RN, A    R=1
mem_codigo[0x02] = 0x89;
//MOV A 20
mem_codigo[0x03] = 0x1CC;
//add A, RN      R=1
mem_codigo[0x04] = 0x69;
//MOV A, @RN     R=1
mem_codigo[0x05] = 0x88;
//JNC 2
mem_codigo[0x06] = 0x102;
//JNZ 1
mem_codigo[0x07] = 0x141;
//JP 1
mem_codigo[0x08] = 0xC1;
 //NONE
mem_codigo[0x09] = 0x160;
//JMP 9
mem_codigo[0xA] = 0x189;
```

0x1CF = 111001111

0X09 = 000001001

0X89 = 010001001

0X1CC = 111001100

0X69 = 001101001

0X88 = 010001000

0X102 = 100000010

0X141 = 101000001

0XC1 = 011000001

0X160 = 101100000

0X189 = 110001001

Programa 4

Programa para testar o correto funcionamento da instrução **mov rn, a**.

```
void programa_teste_4(){

   //mov A, 12(0xC)
  mem_codigo[0x00] = 0x1CC;
  //mov rn, a    R=0
  mem_codigo[0x01] = 0x01;
  //mov a, 2 (0x2)
  mem_codigo[0x02] = 0x1C2;
  //
  //mov a,rn
  mem_codigo[0x03] = 0x0;

}
```

0X1CC = 111001100

0X01 = 000000001

0X1C2 = 101100000

0X0 = 111000010

Programa adição de dois valores introduzidos pelo utilizador

Realizou-se um programa que permite receber o input do utilizador, o utilizador escolhe dois valores e é realizada a soma destes dois valores, e uma instrução “jump if carry”.

```
void programa_teste_adicao() {

  int val1 = 0;
  int val2 = 0;

  Serial.print("Primeiro Valor: ");

  bool waiting = true;
  bool waiting_2 = true;

  while(waiting){

    val1 = Serial.readString().toInt();
    if (val1 !=0 ){
      Serial.print(val1);
      waiting = false;
    }
}
Serial.println();
Serial.print("Segundo Valor: ");

 while(waiting_2){

  val2 = Serial.readString().toInt();

  if (val2 !=0 ){
    Serial.print(val2);
     waiting_2 = false;
  }
}
Serial.println();

//obter 6 primeiros bits
val1 = val1 & 0x3F;
// juntar 111000000 com o valor introduzido a 6 bits
val1 = val1 | 0x1C0;


val2 = val2 & 0x3F;
val2 = val2 | 0x1C0;


//MOV A , VAL1
mem_codigo[0x00] = val1;
//MOV RN, A     R=0
mem_codigo[0x01] = 0x01;
//MOV A, VAL2
mem_codigo[0x02] = val2;
//ADD VAL1, VAL2
mem_codigo[0x03] = 0x61;
//JP CARRY
mem_codigo[0x04] = 0xE0;
}
```

Código em anexo

```

#define mem_codigo_dim 64 //2^6
#define mem_dados_dim 64  //2^6
#define eprom_dim 32      //2^5

#define debounce_time  200
unsigned long now0,ago0; //timestamp debounce MLCK
unsigned long now1, ago1; //timestamp debounce MCLK negativo

//arrays das memorias

//arr memoria codigo
word mem_codigo[mem_codigo_dim];
word mem_codigo_db;

//arr memoria dados
byte mem_dados[mem_dados_dim];
byte mem_dados_db;

//arr eprom
word eprom[eprom_dim];


//---------- ROGRAM COUNTER ----------------------------

//saida do primeiro mux do pc
byte y_pc_mux_0;
//bit selector do mux
bool selPc0;

//adder
byte adder_pc;

        _
//saida do segundo mux do pc
byte y_pc_mux_1;

//registo PC
byte reg_pc_d;
byte reg_pc_q;


//---------- Registos ----------------------------------------

//registos RN

//R0
byte reg_rn_r0_d;
byte reg_rn_r0_q;
bool en_r0;

//R1
byte reg_rn_r1_d;
byte reg_rn_r1_q;
bool en_r1;

//registo A

//mux
byte y_reg_a_mux;

//
byte reg_a_d;
byte reg_a_q;

//registo C

bool reg_c_d;
bool reg_c_q;

//registo Z
bool reg_z_d;
bool reg_z_q;

//registo S
bool reg_s_d;
bool reg_s_q;

//---------- ALU ----------------------------------------

//saidas da ALU
byte alu_Y;
bool alu_cryIn;
bool alu_cryOut;
bool alu_z;
bool alu_sinal;

//mux clr_c
bool y_clear_mux;

//-----------------SAIDAS MODULO CONTROLO-----------------

bool selA0;
bool selA1;
bool enRn;
bool wr;
bool rd;
bool selClrC;
bool enA;
```

```
bool enC;
bool enZ;
bool enS;
bool jm;
bool jp;
bool jc;
bool jnc;
bool jz;
bool jnz;
bool jmp;
//bit de seleção de registo
bool D3_sinal;
//bit de seleção de operacao da ALU
bool D0_sinal;
bool D5_sinal;
bool D6_sinal;



void setup(){

  Serial.begin(9600);
  wr=true;
  rd=true;
  pinMode(2,INPUT_PULLUP);
  preencher_mem_dados_random();
  preencherEPROM();
  //preencher_mem_codigo_random();
  mostrar_menu();
  programa_teste();
  //programa_teste_1();
  //programa_teste_2();

  // programa_teste_3();



  //interrupts realizam atualizacao de registos e flags do CPU
  attachInterrupt(digitalPinToInterrupt(2), MCLKneg, FALLING);
  interrupts();


}

void loop(){

    modulo_funcional();
    modulo_controlo();
    handleUserInput();

}

void preencher_aux(byte add_inicial, byte add_final, word data, word rom[]){

  for ( int i =add_inicial; i<= add_final; i++){
    rom[i] = data;
  }
}

void preencherEPROM(){

  eprom[0x00] = 0x1A00;
  eprom[0x01] = 0x3800;
  eprom[0x02] = 0x1D00;
  eprom[0x03] = 0x9A80;
  eprom[0x04] = 0x9A80;
  eprom[0x05] = 0x9A80;
  eprom[0x06] = 0x9B80;

  eprom[0x07] = 0x9B80;
  eprom[0x08] = 0x5200;
  eprom[0x09] = 0x0800;
  preencher_aux(0xA,0xB,0x1840, eprom);
  preencher_aux(0xC,0xD,0x1820, eprom);
  preencher_aux(0xE,0xF,0x1810, eprom);
  preencher_aux(0x10,0x11,0x1808, eprom);
  preencher_aux(0x12,0x13,0x1804, eprom);
  preencher_aux(0x14,0x15,0x1802, eprom);
  preencher_aux(0x18,0x1B,0x1801, eprom);
  preencher_aux(0x1C,0x1F,0xDA00, eprom);
  preencher_aux(0x16,0x17,0x1800, eprom);

}



void preencher_mem_dados_random(){

  for (int i=0;i < mem_dados_dim; i++){

    //mem dados tem valores a 6 bits
    mem_dados[i] = random(64);
  }

}
void preencher_mem_codigo_random(){


   for (int i=0;i < mem_codigo_dim; i++){
        //mem_codigo tem valores  a 9 bits
        mem_codigo[i] = random(512);
  }

}

```

```
205
206 //COMPONENTES
207
208 //MUX 2X1
209 byte MUX_2x1(boolean S, byte A, byte B){
210   return S ? B : A;
211 }
212
213 //DEMUX 2X1
214 byte DEMUX_2x1 (bool D, bool S0){
215   return D << S0;
216 }
217
218 //REGISTO C/ ENABLE
219 byte registoComEnable(bool en, byte D, byte Q){
220   return MUX_2x1(en,Q,D);
221 }
222
223 //SOMADOR
224 byte somador(byte A, byte B){
225   return A+B;
226 }
227
228 //MUX 4X1
229 byte MUX_4x1(boolean S0, boolean S1, byte in_0, byte in_1, byte in_2, byte in_3){
230   if (!S0 & !S1){
231     return in_0;
232   }
233   if (S0 & !S1){
234     return in_1;
235   }
236   if (!S0 & S1){
237     return in_2;
238   }
239   if (S0 & S1){
240     return in_3;
241   }
242 }
```

```
243
244
245 //----------------------------- modulo controlo -----------------------------
246 void modulo_controlo(){
247
248   word input_db = mem_codigo_db;
249
250    //bits de selecao de RN e de operacao da ALU
251    D3_sinal =  bitRead(input_db,3);
252    D0_sinal =  bitRead(input_db,0);
253    D5_sinal =  bitRead(input_db,5);
254    D6_sinal =  bitRead(input_db,6);
255
256
257   //bits significantes D8,D7,D6,D5,D0
258   //111100001 = 0x1E1
259   word input_masked = input_db & 0x1E1; //aplicar mascara
260
261   //transformar D8,D7,D6,D5,D0 em A4,A3,A2,A1,A0
262   word inputShifted = ((bitRead(input_masked,8) <<4)
263                         | (bitRead(input_masked,7) <<3)
264                        | (bitRead(input_masked,6) <<2)
265                        | (bitRead(input_masked,5) <<1)
266                        | (bitRead(input_masked,0) <<0));
267
268
269   word data = eprom[inputShifted];
270
271   //ler bits data da eprom, atribuir valores aos sinais de saida
272    selA0 = bitRead(data, 15);
273    selA1 = bitRead(data, 14);
274    enRn  = bitRead(data, 13);
275    wr    = bitRead(data, 12);
276    rd    = bitRead(data, 11);
277    selClrC = bitRead(data, 10);
278    enA  = bitRead(data, 9);
```

```
279    enC  = bitRead(data, 8);
280    enZ  = bitRead(data, 7);
281    jm = bitRead(data, 6);
282    jp = bitRead(data, 5);
283    jc = bitRead(data, 4);
284    jnc = bitRead(data, 3);
285    jz = bitRead(data, 2);
286    jnz = bitRead(data, 1);
287    jmp = bitRead(data, 0);
288
289
290 }
291
292
293
294
295 //-------------------------- moduloSelPC0 ----------------------------------------
296
297 bool moduloSelPC0 (){
298
299
300
```

```
  return jm & (reg_s_q) | jp & (!reg_s_q) |

        jc & (reg_c_q) |  jnc & (!reg_c_q)

        | jz & (reg_z_q)  | jnz & (!reg_z_q);

}


//---------------------------- ALU -------------------------------------------

//clrc
bool clrc(bool c){

 c = 0;

}

//not
byte not_a(byte a){
  return ~a;
}

//and
byte and_a(byte a, byte rn){

 return a & rn;
}

//or
byte or_a(byte a, byte rn){
  return a  | rn;
}

//SUBB
byte subb(byte a, byte rn, byte cy) {
  return (a - rn - cy) ;
}
```

```
 }

 //ADDC
byte addc(byte a, byte rn, byte cy) {



  return (a + rn + cy) ;
}


byte alu(bool sel2, bool sel1, bool sel0, byte a, byte rn, bool cy){

  byte output;
  //NOT A
  if (!sel2 & sel1 & sel0){
    output = not_a(a) & 0x3F;

    alu_z = ( output==0);
    alu_sinal = bitRead(output,5);

  }

  //AND A
  if (sel2 & !sel1 & !sel0){
    output = and_a(a,rn)& 0x3F;
    alu_z = ( output==0);
    alu_sinal =  bitRead(output,5);

  }

   //OR A
  if (sel2 & !sel1 & sel0){
    output = or_a(a,rn)& 0x3F;
    alu_z = ( output==0);
    alu_sinal =  bitRead(output,5);

  }

```

```
  //SUBB A, Rn
    if (sel2 & sel1 & !sel0){
        byte outputAux = subb(a,rn, cy);
        alu_cryOut = ((int) a - rn - cy) > 64 | ((int) a - rn - cy) < 0 ;
        output = outputAux & 0x3F;
        alu_z = ( output==0);
        alu_sinal = bitRead(output,5);

  }

  //ADDC A, Rn
  if (sel2 & sel1 & sel0){
    byte outputAux = addc(a,rn, cy);
    alu_cryOut = ((int) a + rn + cy) > 64  | ((int) a + rn + cy) < 0;
    output = outputAux & 0x3F;
    alu_z = ( output==0);
    alu_sinal =  bitRead(output,5);

  }

  return output;

}
```

```
//------------------------- MODELO FUNCIONAL --------------------------

void modulo_funcional(){

  //data-bus da memoria de codigo
  mem_codigo_db = mem_codigo[reg_pc_q];


  //rel5

  //aplicar mascara 11111, para ficar com apenas os 5 bits menos significantes
  byte rel5 = mem_codigo_db & 0x01F;
  // se o numero for negativo, adicionar 1's a frente
  if (bitRead(rel5, 4) == 1){

    rel5 = rel5 | 0x20;

  }

  //defenir EN0 e o EN1
  byte demux_output = DEMUX_2x1(enRn,D3_sinal);
  en_r0 = bitRead(demux_output,0);
  en_r1 = bitRead(demux_output,1);

  //defenir entradas dos registos RN
  reg_rn_r0_d = reg_a_q;
  reg_rn_r1_d = reg_a_q;

  //defenir saida do RN
   byte reg_rn_q = MUX_2x1(D3_sinal, reg_rn_r0_q, reg_rn_r1_q);

  //porta tri_state

  if(!wr){
    mem_dados[reg_rn_q] = reg_a_q;

    mem_dados_db = reg_a_q;
  }

  else {
    mem_dados_db = mem_dados[reg_rn_q];
  }

  byte const_six = mem_codigo_db & 0x3F;

  //multiplexer do registo A
  y_reg_a_mux = MUX_4x1(selA0,selA1,reg_rn_q ,alu_Y ,mem_dados_db , const_six );
  reg_a_d = y_reg_a_mux;

  //ALU
  alu_Y = alu(D6_sinal,D5_sinal,D0_sinal, reg_a_q, reg_rn_q, alu_cryIn);

  //MUX CLRC
  y_clear_mux = MUX_2x1(selClrC, alu_cryOut,0);

  //atribuir o entrada dos registos das flags
  reg_c_d = y_clear_mux;
  reg_z_d = alu_z;
  reg_s_d = alu_sinal;

  //PC
  //Bit sel do mux0
  selPc0 = moduloSelPC0 ();
  //mux0

  y_pc_mux_0 = MUX_2x1(!selPc0, rel5, 1);
  //Serial.print(y_pc_mux_0);

  //adder
  adder_pc = (y_pc_mux_0 + reg_pc_q) & 0x3F;

  //mux1
```

```
474   byte end_six = mem_codigo_db & 0x3F;
475   y_pc_mux_1 = MUX_2x1(jmp,adder_pc,end_six );
476
477   //registo pc
478   reg_pc_d = y_pc_mux_1 & 0x3F;  }
479
480
481 }
482
483
484 void MCLK(){
485
486    long now0 = millis();
487    if (now0 - ago0 > debounce_time){
488
489      //atualizar Registo Program counter
490      reg_pc_q = registoComEnable(1,reg_pc_d,reg_pc_q);
491      attachInterrupt(digitalPinToInterrupt(2), MCLKneg, FALLING);
492
493
494  }
495     ago0 = now0;
496
497 }
498
499 void MCLKneg(){
```

```
500
501
502   now1 = millis();
503   if (now1 - ago1 > debounce_time){
504
505     //Atualizar registos que não o do program counter
506
507     //registos RN
508     reg_rn_r0_q = registoComEnable(en_r0, reg_rn_r0_d, reg_rn_r0_q);
509     reg_rn_r1_q = registoComEnable(en_r1, reg_rn_r1_d, reg_rn_r1_q);
510
511     //registo A
512     reg_a_q = registoComEnable(enA, reg_a_d, reg_a_q);
513
514     //registos flags
515     reg_z_q = registoComEnable(enZ,reg_z_d, reg_z_q );
516
517     reg_c_q = registoComEnable(enC,reg_c_d, reg_c_q );
518     alu_cryIn = reg_c_q;
519
520     reg_s_q = registoComEnable(enZ,reg_s_d, reg_s_q );
521
522     attachInterrupt(digitalPinToInterrupt(2), MCLK, RISING);
523
524   }
525   ago1 = now1;
526
527 }
528
529
530 void teste_mod_funcional(){
531
532   word db = 0x157;
533   byte rel5 = db & 0x01F;
534
535   if (bitRead(rel5, 4) == 1){

536     rel5 = rel5 |  0x0E0;
537   }
538
539 }
540
541
542 void mostrar_menu(){
543
544
545   Serial.println("MENU");
546   Serial.println("r: Mostrar Registos, c: Mostrar memoria de codigo, d: Mostrar memoria de dados, e: Mostrar EPROM, s: Mostrar Sinais,o: Mostrar instrucoes");
547   Serial.println();
548
549
550 }
551
552
553
554
555
556 void mostrar_eprom(){
557
558   Serial.print("----------TABELA EPROM----------");
559   Serial.println(" ");
560   Serial.println(" ");
561   Serial.print("ENDERECO | ");
562   Serial.print(" DATA");
563   Serial.println(" ");
564
565   for (int i =0; i < eprom_dim; i++){
566      Serial.print("0x");
567      Serial.print(i, HEX);
568     if (i<=0xF){
569      Serial.print("      |  0x");
570     }
571     else{
572      Serial.print("     |  0x");
573      }
574      Serial.print(eprom[i], HEX);
```

```
575
576       Serial.println("");
577
578     }
579   }
580
581
582   void mostrar_mem_dados(){
583
584     Serial.print("----------Memoria de dados--------------");
585     Serial.println(" ");
586     Serial.println(" ");
587     Serial.print("ENDERECO    | ");
588     Serial.print(" INSTRUCAO");
589     Serial.println(" ");
590
591     int c =0;
592
593
594     for (byte i=0;i < mem_dados_dim; i++){
595
596       if (c==0){
597
598        Serial.print("[0x");
599        Serial.print(i,HEX);
600        Serial.print(" - ");
```

```
601         Serial.print("0x");
602         Serial.print(i+8,HEX);
603         if(i==0){
604         Serial.print("]     ");
605         }
606         else if(i==8){
607           Serial.print("]   ");
608         }
609         else{
610            Serial.print("]  ");
611         }
612       }
613       Serial.print("0x");
614       Serial.print(mem_dados[i],HEX);
615
616       Serial.print(" ");
617       c++;
618       if (c == 8){
619           Serial.println();
620           c = 0;
621         }
622     }
623     Serial.println();
624
625   }
626
627   void mostrar_mem_codigo(){
628
629     Serial.print("----------Memoria de Codigo--------------");
630     Serial.println(" ");
631     Serial.println(" ");
632     Serial.print("ENDERECO    | ");
633     Serial.print(" INSTRUCAO");
634     Serial.println(" ");
635
636     int c=0;
637
638     for (byte i=0;i < mem_codigo_dim; i++){
639
```

```
640     if (c==0){
641
642       Serial.print("[0x");
643       Serial.print(i,HEX);
644       Serial.print(" - ");
645       Serial.print("0x");
646       Serial.print(i+8,HEX);
647       if(i==0){
648       Serial.print("]    ");
649       }
650       else if(i==8){
651         Serial.print("]   ");
652       }
653       else{
654          Serial.print("]  ");
655       }
656     }
657     Serial.print("0x");
658     Serial.print(mem_codigo[i],HEX);
659
660     Serial.print(" ");
661     c++;
662     if (c == 8){
663         Serial.println();
664         c = 0;
665       }
666   }
667   Serial.println();
668 }
669
670
671
672
673 void mostrar_registos(){
674
675   Serial.println("----------Registos----------------");
676   Serial.println(" ");
677
678    Serial.print("[PC : 0x");
679    Serial.print(reg_pc_q, HEX);
680    Serial.print(" | R0 : 0x");
681    Serial.print(reg_rn_r0_q, HEX);
682    Serial.print(" | R1 : 0x");
683    Serial.print(reg_rn_r1_q, HEX);
684    Serial.print(" | RA : 0x");
685    Serial.print(reg_a_q, HEX);
686    Serial.print(" | RC : 0x");
687    Serial.print(reg_c_q, HEX);
688    Serial.print(" | RZ : 0x");
689    Serial.print(reg_z_q, HEX);
690    Serial.print(" | RS : 0x");
691    Serial.print(reg_s_q, HEX);
692    Serial.println(" ] ");
693    Serial.println();
694
695 }
696
697
698 void mostrar_sinais(){
699
700
```

```
    Serial.println("-----------Modulo de controlo----------------");
    Serial.println("");
    Serial.print("[selA0 : ");
    Serial.print(selA0, BIN);
    Serial.print(" | selA1 : ");
    Serial.print(selA1, BIN);
    Serial.print(" | enRn : ");
    Serial.print(enRn, BIN);
    Serial.print(" | /wr : ");
    Serial.print(wr, BIN);
    Serial.print(" | /rd : ");
    Serial.print(rd, BIN);
    Serial.print(" | selClrC : ");
    Serial.print(selClrC, BIN);
    Serial.print(" | enA : ");
    Serial.print(enA, BIN);
    Serial.print(" | enC : ");
    Serial.print(enC, BIN);
    Serial.print(" | enZ : ");
    Serial.print(enZ, BIN);
    Serial.print(" | enS : ");
    Serial.print(enZ, BIN);
    Serial.print(" | jm : ");
    Serial.print(jm, BIN);
    Serial.print(" | jp : ");
    Serial.print(jp, BIN);
    Serial.print(" | jc : ");
    Serial.print(jc, BIN);
    Serial.print(" | jnc : ");
    Serial.print(jnc, BIN);
    Serial.print(" | jz : ");
    Serial.print(jz, BIN);
    Serial.print(" | jnz : ");
    Serial.print(jnz, BIN);
    Serial.print(" | jmp : ");
    Serial.print(jmp, BIN);
    Serial.println(" ] ");
    Serial.println();



}

void mostrar_instrucoes(){

  word instrucao = mem_codigo[reg_pc_q];

  Serial.println("-----------Instrucao----------------");
  Serial.print("instrucao: 0x");
  Serial.print(instrucao, HEX);
  Serial.print(" ");

  //bits significantes D8,D7,D6,D5,D0
  bool bit_D8 = bitRead(instrucao, 8);
  bool bit_D7 = bitRead(instrucao, 7);
  bool bit_D6 = bitRead(instrucao, 6);
  bool bit_D5 = bitRead(instrucao, 5);
  bool bit_D0 = bitRead(instrucao, 0);

  if ( bit_D8 & bit_D7 & bit_D6){
    Serial.print("MOV A, #constante6");
  }

  else if (!bit_D8 & !bit_D7 & !bit_D6 & !bit_D5 & !bit_D0){
    Serial.print("MOV A, Rn");
  }
   else if (!bit_D8 & !bit_D7 & !bit_D6 & !bit_D5 & bit_D0){
    Serial.print("MOV Rn, A");
  }
  else if (!bit_D8 & !bit_D7 & !bit_D6 & bit_D5 & !bit_D0){
    Serial.print("CLRC");
  }
  else if (!bit_D8 & !bit_D7 & !bit_D6 & bit_D5 & bit_D0){
    Serial.print("NOT A");
  }
  else if (!bit_D8 & !bit_D7 & bit_D6 & !bit_D5 & !bit_D0){
    Serial.print("AND A, Rn");
  }

  else if (!bit_D8 & !bit_D7 & bit_D6 & !bit_D5 & bit_D0){
    Serial.print("OR A, Rn");
  }
   else if (!bit_D8 & !bit_D7 & bit_D6 & bit_D5 & !bit_D0){
    Serial.print("SUBB A, Rn");
  }
  else if (!bit_D8 & !bit_D7 & bit_D6 & bit_D5 & bit_D0){
    Serial.print("ADDC A, Rn");
  }
  else if (!bit_D8 & bit_D7 & !bit_D6 & !bit_D5 & !bit_D0){
    Serial.print("MOV A, @Rn");
  }
  else if (!bit_D8 & bit_D7 & !bit_D6 & !bit_D5 & bit_D0){
    Serial.print("MOV @Rn, A");
  }
```

```
  else if (!bit_D8 & bit_D7 & !bit_D6 & bit_D5 ){
    Serial.print("JM rel5");
  }
  else if (!bit_D8 & bit_D7 & bit_D6 & !bit_D5 ){
    Serial.print("JP rel5");
  }
  else if (!bit_D8 & bit_D7 & bit_D6 & bit_D5 ){
    Serial.print("JC rel5");
  }
  else if (bit_D8 & !bit_D7 & !bit_D6 & !bit_D5 ){
    Serial.print("JNC rel5");
  }
  else if (bit_D8 & !bit_D7 & !bit_D6 & bit_D5 ){
    Serial.print("JZ rel5");
  }
  else if (bit_D8 & !bit_D7 & bit_D6 & !bit_D5 ){
    Serial.print("JNZ rel5");
  }
   else if (bit_D8 & bit_D7  & !bit_D6 ){
    Serial.print("JMP end6");
  }
  else if (bit_D8 & !bit_D7  & bit_D6 & bit_D5 ){
    Serial.print("None");
  }



   Serial.println();



}
void handleUserInput(){

   if (Serial.available() > 0) {
    // read the incoming byte:


     switch(Serial.read()){
      case 'r': mostrar_registos();
       break;
      case 'c': mostrar_mem_codigo();
       break;
       case 'd': mostrar_mem_dados();
       break;
       case 'e': mostrar_eprom();
       break;
       case 's': mostrar_sinais();
       break;
       case 'o':mostrar_instrucoes();
       break;

     }

   }

}


void programa_teste_adicao() {

  int val1 = 0;
  int val2 = 0;

  Serial.print("Primeiro Valor: ");

  bool waiting = true;
  bool waiting_2 = true;

  while(waiting){

    val1 = Serial.readString().toInt();
    if (val1 !=0 ){
      Serial.print(val1);

      waiting = false;
    }
}
Serial.println();
Serial.print("Segundo Valor: ");

 while(waiting_2){

  val2 = Serial.readString().toInt();

  if (val2 !=0 ){
    Serial.print(val2);
     waiting_2 = false;
  }
}
Serial.println();

//obter 6 primeiros bits
val1 = val1 & 0x3F;
// juntar 111000000 com o valor introduzido a 6 bits
val1 = val1 | 0x1C0;

val2 = val2 & 0x3F;
val2 = val2 | 0x1C0;
```

```
892
893
894 //MOV A , VAL1
895 mem_codigo[0x00] = val1;
896 //MOV RN, A     R=0
897 mem_codigo[0x01] = 0x01;
898 //MOV A, VAL2
899 mem_codigo[0x02] = val2;
900 //ADD VAL1, VAL2
901 mem_codigo[0x03] = 0x61;
902 //JP CARRY
903 mem_codigo[0x04] = 0xE0;
904 }
905 /*
906
907 Instruções  D8   D7 D6  D5  D0
908 MOV A, #c6  1   1   1   -   -
909 MOV A, Rn   0   0   0   0   0
910 MOV Rn, A   0   0   0   0   1
911 CLRC    0   0   0   1   0
912 NOT A   0   0   0   1   1
913 AND A, Rn   0   0   1   0   0
914 OR A, Rn    0   0   1   0   1
915 SUBB A, Rn  0   0   1   1   0
916 ADDC A, Rn  0   0   1   1   1
917 MOV A, @Rn  0   1   0   0   0
918 MOV @Rn, A  0   1   0   0   1
919 JM rel5 0   1   0   1   -
920 JP rel5 0   1   1   0   -
921 JC rel5 0   1   1   1   -
922 JNC rel5    1   0   0   0   -
923 JZ rel5 1   0   0   1   -
924 JNZ rel5    1   0   1   0   -
925 JMP end6    1   1   0   -   -
926 */
927
928
929
930 //fazer programinha de teste
```

```
931
932
933 void programa_teste_1(){
934
935   //mov A, 12(0xC)
936   mem_codigo[0x00] = 0x1CC;
937   //MOV @rn,a         R=0
938   mem_codigo[0x01] = 0x81;
939   //MOV A, 2
940   mem_codigo[0x02] = 0x1C2;
941   //MOV A,@RN         R=0
942   mem_codigo[0x03] = 0x80;
943   //JMP end6 3
944   mem_codigo[0x04] = 0x183;
945
946 }
947
948 void programa_teste_2(){
949
950   //MOV A, 8(0x8)
951   mem_codigo[0x00] = 0x1C8;
952   //MOV RN, A       R=0
953   mem_codigo[0x01] = 0x01;
954   //AND A, RN       R=0
955   mem_codigo[0x02] = 0x40;
956   //NOT A, RN       R=0
957   mem_codigo[0x03] = 0x21;
958   //OR A, RN        R=0
959   mem_codigo[0x04] = 0x41;
960   //ADD A, RN       R=0
961   mem_codigo[0x05] = 0x61;
962   //CLR C
963   mem_codigo[0x06] = 0x20;
964   //SUBB A, RN      R=0
965   mem_codigo[0x07] = 0x60;
966   //JC
967   mem_codigo[0x08] = 0xE0;
```

```
   //JZ
   mem_codigo[0x09] = 0x120;
   //JM  010110100
   mem_codigo[0xA] = 0xB4;

}

void programa_teste_3(){

    //MOV A 15(0xF)
    mem_codigo[0x00] = 0x1CF;
    //MOV RN,A     R=1
    mem_codigo[0x01] = 0x09;
    //MOV @RN, A   R=1
    mem_codigo[0x02] = 0x89;
    //MOV A 20
    mem_codigo[0x03] = 0x1CC;
    //add A, RN     R=1
    mem_codigo[0x04] = 0x69;
    //MOV A, @RN    R=1
    mem_codigo[0x05] = 0x88;
    //JNC 2
    mem_codigo[0x06] = 0x102;
    //JNZ 1
    mem_codigo[0x07] = 0x141;
    //JP 1
    mem_codigo[0x08] = 0xC1;
     //NONE
    mem_codigo[0x09] = 0x160;
    //JMP 9
    mem_codigo[0xA] = 0x189;



}
```

```
void programa_teste_4(){

    //mov A, 12(0xC)
   mem_codigo[0x00] = 0x1CC;
   //mov rn, a   R=0
   mem_codigo[0x01] = 0x01;
   //mov a, 2 (0x2)
   mem_codigo[0x02] = 0x1C2;
   //
   //mov a,rn
   mem_codigo[0x03] = 0x0;

}
```

Conclusões

A partir da realização deste trabalho prático foi possível desenvolver diferentes competências, respetivamente:

- Reconhecimento das vantagens e desvantagens associadas a diferentes métodos de arquitetura do CPU, respetivamente arquitetura de Harvard e arquitetura Von Newman.
- Distinção e utilização de memórias do tipo RAM e ROM.
- Codificação de diferentes instruções com o menor número de bits possível.
- Distinção de diferentes instruções de CPU em conjuntos específicos.
- Abordagem modular para implementar diferentes módulos físicos.
- Interação de diferentes componentes de hardware com base na técnica de encaminhamento de dados.
- Projeção de circuitos físicos em arduino com múltiplos componentes a funcionar a um ritmo de *Clock*.
- Técnicas de programação baseadas em operações bit-a-bit como a utilização de *bitmasks.*

Neste trabalho prático foi possível criar um microprocessador capaz de realizar diversas instruções a partir duma abordagem modular e sequencial de etapas. No desenvolvimento do microprocessador começou-se por efetuar uma etapa de projeção. Na fase de projeção do CPU foi especifica a quantidade de bits para os diferentes componentes, codificação das diferentes instruções e o desenho do módulo funcional do microprocessador. Por fim foi realizada uma etapa de implementação, a partir da simulação do CPU no arduino.

O correto funcionamento do microprocessador foi comprovado a partir dum conjunto de programas de teste que permitem validar a correta implementação do microprocessador testando todas as instruções possíveis de ser realizadas.

Bibliografia

- Folhas de Computação Física de 2020/2021, da autoria do Professor Jorge Pais, em formato PDF.
- Folhas de Computação Física de 2020/2021, da autoria do Professor Carlos Carvalho, em formato PDF.
- Harvard Arquitecture. Disponível em https://www.geeksforgeeks.org/harvard-architecture/.
- Arquitetura Harvard. Disponível em https://pt.wikipedia.org/wiki/Arquitetura_Harvard.