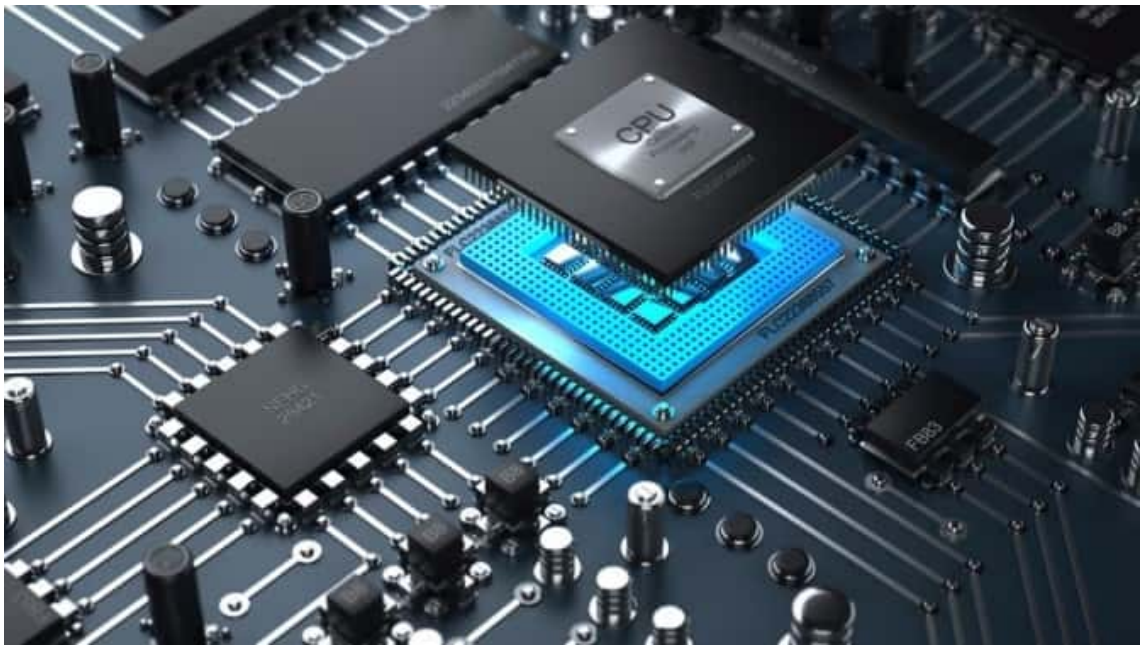
Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e Multimédia

Computação Física - CF - 2122SV

Trabalho Prático 2



Docente Jorge Pais
Realizado por (Grupo 10):
Roman Ishchuk 43498
Pedro Silva 48965
Cláudia Sequeira 49247

23 de maio de 2022

Conteúdo

Lista de Figuras

1 Introdução

Neste trabalho vão ser abordados as seguintes temáticas:

- Microprocessador ou *CPU*(*Central Processing Unit*);
- *RAM* (*Random Acess Memory*);
- e *ROM* (*Read Only Memory*);

Em que o *CPU* é uma máquina de *hardware* que funciona desencadeando uma serie de instruções, guardadas na memória de código (*ROM*).

A *ROM* é um dispositivo onde está guardada a informação necessária para correr o programa. A informação não é perdida quando o sistema é desligado ou seja, os endereços e a informação mantém-se. A *RAM* é constituída por um conjunto de registos bidirecionais, que permitem registar e ir buscar informação.

Para projetar um microprocessador também vão ser abordados temas como o Módulo Funcional, onde estão os nossos registos e a *ALU* (*Arithmetic Logic Unit*) onde vão ser realizadas as operações de lógica e aritméticas. O Módulo de Controlo que controla os sinais necessários para realizar as instruções especificadas.

2 Objetivos

Temos como objetivos projetar um microprocessador A_Rn, baseado numa arquitetura de Harvard. Após projetar o nosso CPU pretendemos fazer a sua simulação de funcionalidade em Arduíno. O nosso microprocessador deve realizar um conjunto de instruções, a especificar no desenvolvimento.

3 Desenvolvimento - Projeção de um CPU

3.1 Especificação das Instruções

O *CPU* ou microprocessador segue a Arquitetura de *Harvard* e tem de cumprir as instruções explicitadas na figura 1.

Instrução	Funcionalidade
MOV Rn, const5	Rn = const5
MOV A, Rn	A = Rn
MOV Rn, A	Rn = A
MOV A, @Rn	A = MD(Rn)
MOV @Rn, A	MD(Rn) = A
CPLF	C = C/; Ov = Ov/; Z = Z/
NOT A	A = A/
AND A, Rn	A = A and Rn
OR A, Rn	A = A or Rn
ADDC A, Rn	A = A + Rn + C
SUBB A, Rn	A = A − Rn − C
JC rel6	Se (C) PC += rel6
JNZ rel6	Se (Z/) PC += rel6
JOV rel6	Se (Ov) PC += rel6
JMP end7	PC = end7

O microprocessador tem o registo de uso geral `A`, um conjunto de registos `Rn` composto pelos registos `R0` e `R1`, o registo de controlo de execução `PC` e as *flags* `C` (*Carry* ou *Borrow*), `Ov` (*Overflow*) e `Z` (Zero). A abreviatura `MD` significa memória de dados. A abreviatura `rel6` representa um valor relativo a 6 bits. A abreviatura `end7` representa um valor absoluto a 7 bits. A abreviatura `const5` representa uma constante a 5 bits.

Figura 1: Requisitos do Microprocessador

3.2 Funcionamento da Arquitetura Harvard

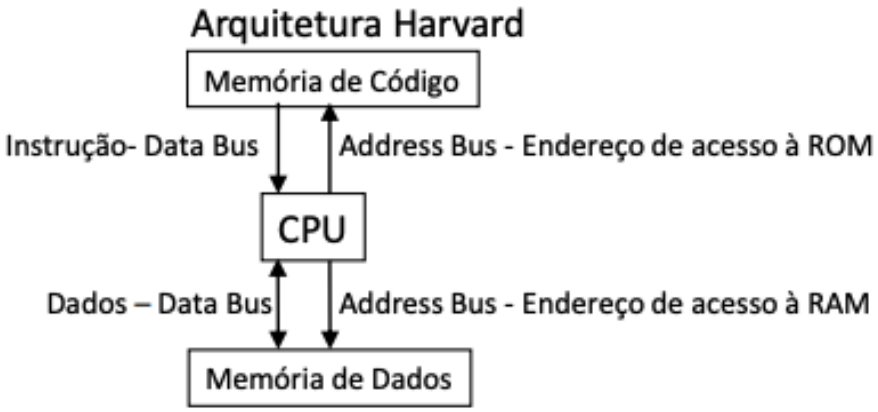


Figura 2: Arquitetura Harvard Fonte:(2) Projeto de um *CPU* Passo a Passo - Jorge Pais

Por cada ciclo do *master clock* o *CPU* cumpre instruções sequencialmente, exceto quando existem *jumps*. As instruções são lidas da Memória de Código (*ROM*), onde o *Data Bus* faz o barramento do conjunto de sinais digitais (ou seja informação) e o *Address Bus* faz o barramento dos endereços das instruções.

A Memória de Dados (*RAM*) guarda as variáveis que estão a ser utilizadas para correr as instruções. É bidirecional, podemos guardar informação ou ir buscar informação à MD.)

3.3 Especificação da quantidade de bits para cada um dos Registos

- Rn — 2 registos (R0, R1) em que ambos são registos de 5 bits;
  O tamanho necessário para os registos Rn é definidos pela instrução MOV Rn, const5 (Rn = const5) em que estamos a atribuir a um dos registos Rn o valor de uma constante a 5 bits.
- A — registo de 5 bits;
  O registo A é definido por 5 bits pela instrução MOV A, Rn (A = Rn) em que o registo A fica com o valor do registo Rn.
- PC — *Program Counter*, registo de 8 bits;
  O *Program Counter* itera entre os endereços das instruções guardadas na *ROM*, no nosso caso optámos por utilizar 8 bits para diferenciar entre as 15 instruções que temos na *ROM*;
- C — *Flag* de *Carry/Borrow*, registo de 1 bit;
- Z — *Flag* de *Zero*, registo de 1 bit;
- Ov — *Flag* de *Overflow*, registo de 1 bit;
- const5 — A constante 5 é um parâmetro com valor a 5 bits;
- rel6 — O valor relativo é definido a 6 bits;
- end7 — Valor definido a 7 bits;

3.4 Especificação da quantidade de bits do *Address Bus* e do *Data Bus*

Após definirmos o tamanho dos registos passamos para a definição do tamanho do barramento de endereço e de dados.

Memória de Dados (MD):

- AB — 5 bits;
  O tamanho do *Address Bus* é definido pelo tamanho da informação que vamos guardar, no nosso caso são os 5 bits dos registos Rn.
- DB — 5 bits;
  O tamanho do *Data Bus* é definido pelo tamanho do registo A, que também são 5 bits;

Memória de Código (MC):

- AB — 9 bits;
  O tamanho do *Address Bus* da memória de código é definido pelo tamanho do PC, no nosso caso são 8 bits e acrescentamos mais um para evitar problemas com os jumps.
- DB — 9 bits;
  O tamanho do *Data Bus* está definido pela quantidade de bits que precisamos para realizar as instruções.

3.5 Codificação das Instruções

Fizemos uma tabela que codifica, de modo a serem diferenciáveis, os endereços das instruções. Encostámos à direita os bits de parâmetro, que são os bits definidos pelo programador que utiliza o *CPU*. A nossa codificação acabou por ser reduzida a 6 bits, que são os nossos bits de dependência.

Instrução	Parâmetros	D8	D7	D6	D5	D4	D3	D2	D1	D0
MOV	Rn, Const 5	0	0	0	C4	C3	C2	C1	C0	Rn
MOV	A, Rn	0	1	0	0	0	0	1	1	Rn
MOV	Rn, A	0	1	0	0	0	1	1	1	Rn
MOV	A, @Rn	0	1	0	0	1	1	1	1	Rn
MOV	@Rn, A	0	1	0	0	1	0	1	1	Rn
CPLF		0	0	1	0	1	1	1	1	1
NOT	A	0	0	1	0	1	0	1	1	1
AND	A, Rn	0	0	1	1	0	0	1	1	Rn
OR	A, Rn	0	0	1	1	0	1	1	1	Rn
ADDC	A, Rn	0	0	1	1	1	0	1	1	Rn
SUBB	A, Rn	0	0	1	1	1	1	1	1	Rn
JC	Rel 6	0	1	1	R5	R4	R3	R2	R1	R0
JNZ	Rel 6	1	0	0	R5	R4	R3	R2	R1	R0
JOv	Rel 6	1	0	1	R5	R4	R3	R2	R1	R0
JMP	End7	1	1	E6	E5	E4	E3	E2	E1	E0

Figura 3: Tabela de Codificação de Instruções

Em seguida criámos uma tabela em que está especificado, para cada instrução, quais são os sinais ativos que vamos precisar. Retiramos os parâmetros que são agora *don't cares*, pois como projetistas de *hardware* são valores que já não nos interessam dado que não são necessários para distinguir entre instruções.

Instrução	Parâmetros	D8	D7	D6	D5	D4	D3	C	NZ	Ov	Sinais Ativos
MOV	Rn, Const 5	0	0	0	-	-	-	-	-	-	EnAux, EnRn
MOV	A, Rn	0	1	0	0	0	0	-	-	-	A_0, EnA
MOV	Rn, A	0	1	0	0	0	1	-	-	-	EnRn
MOV	A, @Rn	0	1	0	0	1	1	-	-	-	EnA, RD/
MOV	@Rn, A	0	1	0	0	1	0	-	-	-	WR/
CPLF		0	0	1	0	1	1	-	-	-	EnC, EnOv, EnZ
NOT	A	0	0	1	0	1	0	-	-	-	A_1, EnA, EnZ
AND	A, Rn	0	0	1	1	0	0	-	-	-	A_1, EnA, EnZ
OR	A, Rn	0	0	1	1	0	1	-	-	-	A_1, EnA, EnZ
ADDC	A, Rn	0	0	1	1	1	0	-	-	-	A_1, EnC, EnOv, EnZ, EnA
SUBB	A, Rn	0	0	1	1	1	1	-	-	-	A_1, EnC, EnOv, EnZ, EnA
JC	Rel 6	0	1	1	-	-	-	1	-	-	JC
JC	Rel 6	0	1	1	-	-	-	0	-	-	
JNZ	Rel 6	1	0	0	-	-	-	-	0	-	
JNZ	Rel 6	1	0	0	-	-	-	-	1	-	JNZ
JOv	Rel 6	1	0	1	-	-	-	-	-	1	JOV
JOv	Rel 6	1	0	1	-	-	-	-	-	0	
JMP	End7	1	1	-	-	-	-	-	-	-	JMP

Figura 4: Tabela de Codificação de Instruções com os Sinais Ativos

3.6 Módulo Funcional

"O módulo funcional é um diagrama de blocos constituído por todos os dispositivos hardware disponibilizado pelos fabricantes, tais como, multiplexers, demultiplexers, comparadores, codificadores, descodificadores, flip-flops, registos, contadores, memórias, etc." [1]

Para o desenho do nosso Módulo Funcional obedecemos a metodologia de seguir a lista de instruções e desenhar os componentes conforme eram necessários para realizar as mesmas.

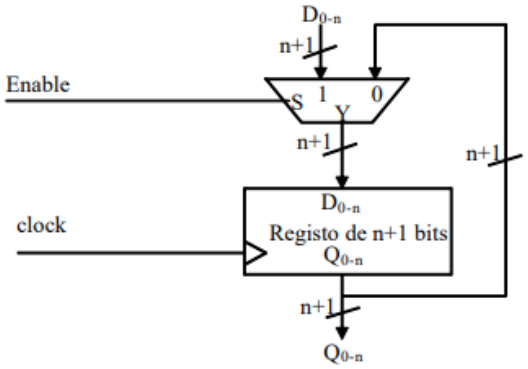


Figura 5: Registo com *enable* de n+1 bits

O registo serve para armazenar a informação que é lhe prestada. Ele só vai armazenar informação nova quando o *Enable* está a 1 e existe um ciclo *clock*. Caso o *Enable* esteja a 0, a informação à saída não muda ou seja não é registada na célula de memória. Os sinais que nos vão permitir fazer registos e utilizar os multiplexers são gerados pelo nosso módulo de controlo.

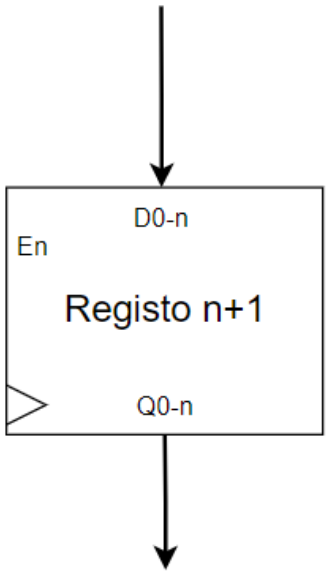


Figura 6: Símbolo lógico do Registo com *enable* de n+1 bits

[1] (1) Folhas de Computação Física - Jorge Pais, 2021/22

Começámos por definir qual deve ser o funcionamento do registo Rn, que por sua vez é um conjunto de 2 registos. A figura 7 demonstra o seu comportamento.

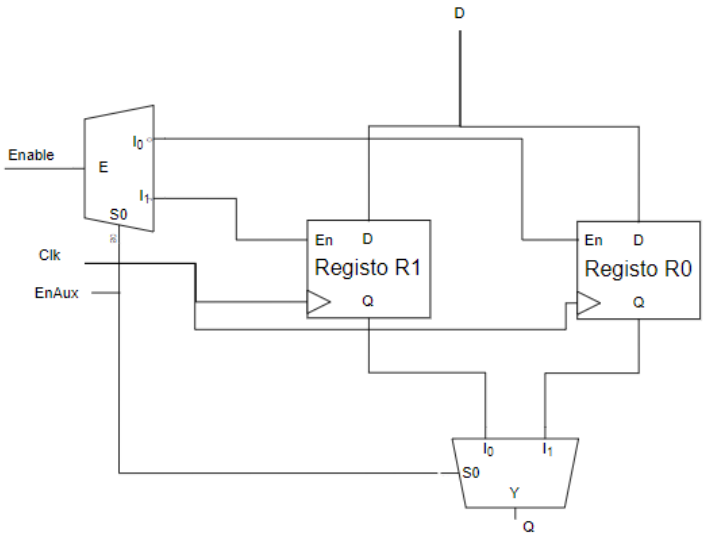


Figura 7: Funcionamento do Registo Rn, que contém 2 Registos

O nosso registo PC vai ser fundamentar pois é ele que nos permite percorrer os endereços da ROM, onde está guardada a informação que precisamos para realizar as instruções do nosso CPU. [2]

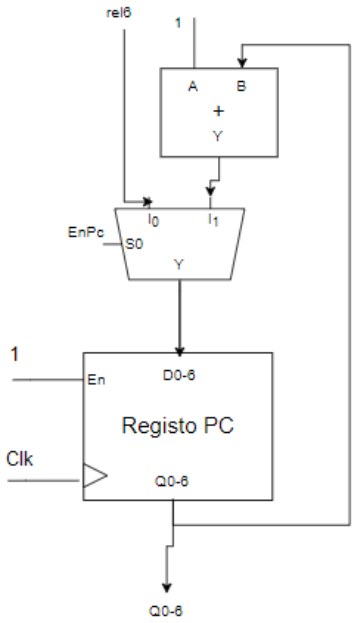


Figura 8: Funcionamento do *Program Counter*

[2]Projeto de um *CPU* Passo a Passo - Jorge Pais, 2021/22

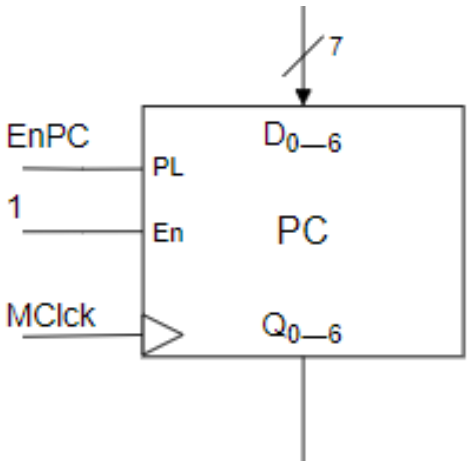


Figura 9: Símbolo lógico do contador crescente com enable

Após definirmos a simbologia para os nossos registos desenhamos os componentes todos que pertencem ao Módulo Funcional. Como podemos averiguar na figura 10, temos as 2 memórias uma de cada lado. Temos o registo A, o conjunto de registos Rn, o contador PC, os registos das *flags*, 2 *multiplexers*, a ALU e finalmente o módulo de controlo.

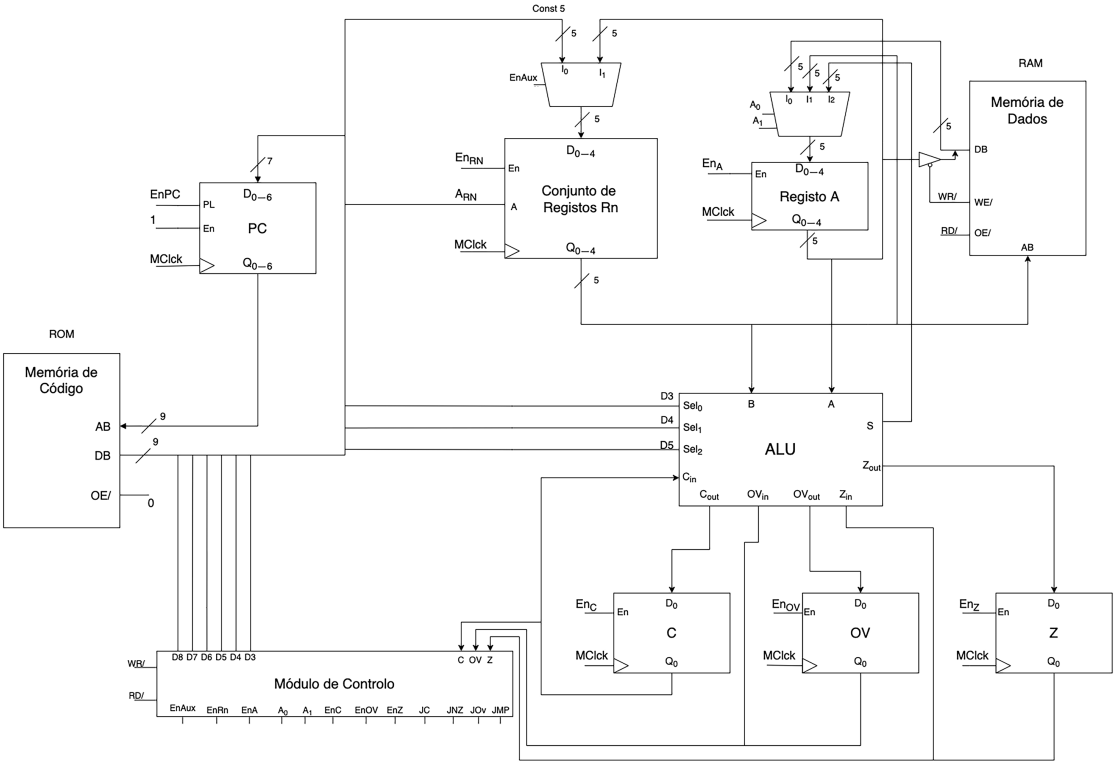


Figura 10: Módulo Funcional

3.7 Módulo de Controlo

"O módulo de controlo é um circuito combinatório (...) que aciona os dispositivos existentes no módulo funcional. Para tal, o módulo de controlo tem como entradas, sinais vindos do módulo funcional, e como saídas, sinais que comandam os dispositivos constituintes do módulo funcional." [3]

O nosso módulo de controlo vai ter como entradas os bits de dependência D8-3 (estes são os que diferem as instruções), e também as saídas dos registos *Carry, Overflow* e Zero. As saídas são todos os sinais ativos falados na secção 3.5.

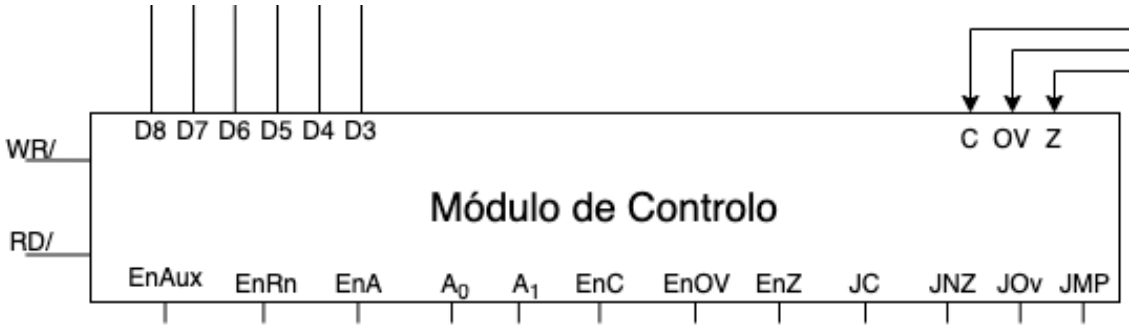


Figura 11: Módulo de Controlo

Nós podemos dividir o módulo em dois módulos diferentes. Um módulo direcionado para gerar os sinais PC (implementado com um circuito combinatório) e o outro para gerar o resto dos sinais. Ao separarmos os dois vamos diminuir a dimensão da nossa tabela de programação e consequentemente o tamanho da ROM do módulo de controlo. Vamos gerar o sinal EnPC com os valores de C, Ov e Z.

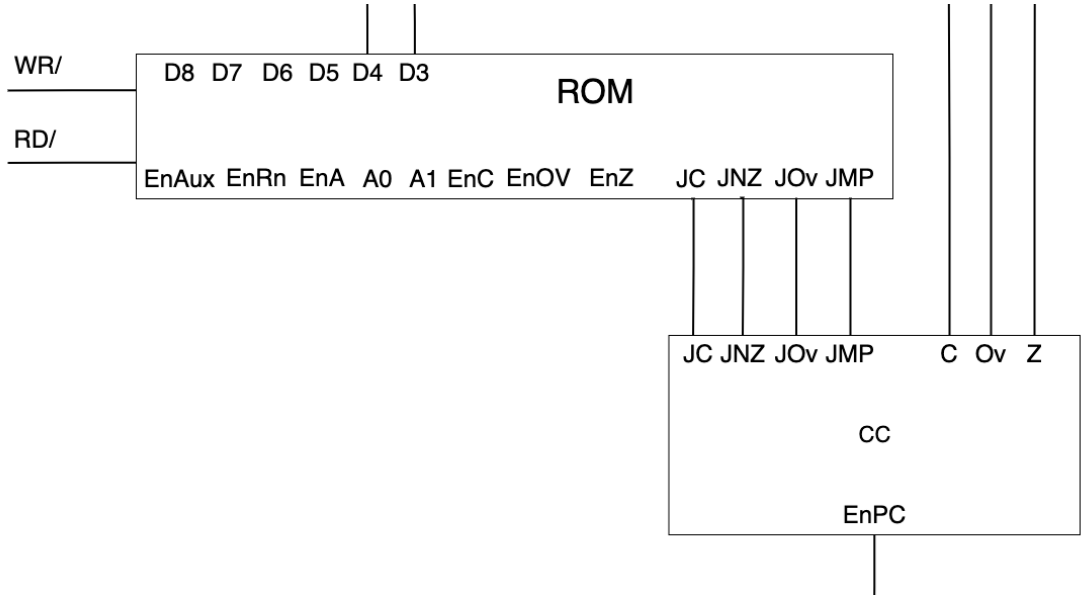


Figura 12: Módulo de controlo

Obtendo a expressão lógica do EnPC:

$$EnPC = JNZ \;\&\&\; QZ/ \;||\; JC \;\&\&\; QC \;||\; JOV \;\&\&\; QOV \;||\; JMP$$

[3](1) Folhas de Computação Física - Jorge Pais, 2021/22

3.8 Programação da *ROM* que implementa o Módulo de Controlo

Para conseguirmos simular o *CPU* no Arduino, temos que criar uma tabela de programação da *ROM* que implementa o módulo de controlo, ou seja, uma tabela que tenha as entradas e saídas do MC. Começámos por construir a tabela mais simplificada, já com os endereços em hexadecimal da saída.

D8	D7	D6	D5	D4	D3	Sinais Ativos	EnC	EnZ	EnOv	WR/	RD/	EnA	A_0	A_1	EnRn	JC	JNZ	JOv	JMP	EnAux	Hex
0	0	0	-	-	-	EnAux, EnRn	0	0	0	1	1	0	0	0	1	0	0	0	0	1	621h
0	1	0	0	0	0	A_0, EnA	0	0	0	1	1	1	1	0	0	0	0	0	0	0	780h
0	1	0	0	0	1	EnRn	0	0	0	1	1	0	0	0	1	0	0	0	0	0	620h
0	1	0	0	1	1	EnA, RD/	0	0	0	1	0	1	0	0	0	0	0	0	0	0	500h
0	1	0	0	1	0	WR/	0	0	0	0	1	0	0	0	0	0	0	0	0	0	200h
0	0	1	0	1	1	EnC, EnOv, EnZ	1	1	1	1	1	0	0	0	0	0	0	0	0	0	3E00h
0	0	1	0	1	0	A_1, EnA, EnZ	0	1	0	1	1	1	0	1	0	0	0	0	0	0	1740h
0	0	1	1	0	0	A_1, EnA, EnZ	0	1	0	1	1	1	0	1	0	0	0	0	0	0	1740h
0	0	1	1	0	1	A_1, EnA, EnZ	0	1	0	1	1	1	0	1	0	0	0	0	0	0	1740h
0	0	1	1	1	0	A_1, EnC, EnOv, EnZ, EnA	1	1	1	1	1	1	0	1	0	0	0	0	0	0	3F40h
0	0	1	1	1	1	A_1, EnC, EnOv, EnZ, EnA	1	1	1	1	1	1	0	1	0	0	0	0	0	0	3F40h
0	1	1	-	-	-	JC	0	0	0	1	1	0	0	0	0	1	0	0	0	0	610h
1	0	0	-	-	-	JNZ	0	0	0	1	1	0	0	0	0	0	1	0	0	0	608h
1	0	1	-	-	-	JOV	0	0	0	1	1	0	0	0	0	0	0	1	0	0	604h
1	1	-	-	-	-	JMP	0	0	0	1	1	0	0	0	0	0	0	0	1	0	602h

Figura 13: Tabela *ROM* 1

Logo de seguida, efetuamos os cálculos para os endereços das entradas do módulo de controlo:

A5	A4	A3	A2	A1	A0	Gama de	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	Hex
D8	D7	D6	D5	D4	D3	Endereços (HEX)	EnC	EnZ	EnOv	WR/	RD/	EnA	A_0	A_1	EnRn	JC	JNZ	JOv	JMP	EnAux	Hex
0	0	0	-	-	-	[0, 7]	0	0	0	1	1	0	0	0	1	0	0	0	0	1	621h
0	1	0	0	0	0	10	0	0	0	1	1	1	1	0	0	0	0	0	0	0	780h
0	1	0	0	0	1	11	0	0	0	1	1	0	0	0	1	0	0	0	0	0	620h
0	1	0	0	1	1	13	0	0	0	1	0	1	0	0	0	0	0	0	0	0	500h
0	1	0	0	1	0	12	0	0	0	0	1	0	0	0	0	0	0	0	0	0	200h
0	0	1	0	1	1	B	1	1	1	1	1	0	0	0	0	0	0	0	0	0	3E00h
0	0	1	0	1	0	A	0	1	0	1	1	1	0	1	0	0	0	0	0	0	1740h
0	0	1	1	0	0	C	0	1	0	1	1	1	0	1	0	0	0	0	0	0	1740h
0	0	1	1	0	1	D	0	1	0	1	1	1	0	1	0	0	0	0	0	0	1740h
0	0	1	1	1	0	E	1	1	1	1	1	0	0	1	0	0	0	0	0	0	3F40h
0	0	1	1	1	1	F	1	1	1	1	1	0	0	1	0	0	0	0	0	0	3F40h
0	1	1	-	-	-	[18, 1F]	0	0	0	1	1	0	0	0	0	1	0	0	0	0	610h
1	0	0	-	-	-	[20, 27]	0	0	0	1	1	0	0	0	0	0	1	0	0	0	608h
1	0	1	-	-	-	[28, 2F]	0	0	0	1	1	0	0	0	0	0	0	1	0	0	604h
1	1	-	-	-	-	[30, 3F]	0	0	0	1	1	0	0	0	0	0	0	0	1	0	602h
						[8, 9]	0	0	0	1	1	0	0	0	0	0	0	0	0	0	600h

Figura 14: Tabela *ROM* 2

Com esta última tabela, vamos poder utilizá-la no nosso código, "transformando"a num *array* de hexadecimais. A ordem será a primeira gama de endereços (entradas), e o conteúdo será os endereços das saídas.

Observação: Podemos averiguar, que não foi utilizado a gama de endereços [8,9], mas para o código funcionar bem, não podíamos deixar de lado, então codificamos a de tal forma, que não ativa nenhum sinal.

4 Conclusões

O objetivo deste trabalho era o de projetar um *CPU* e, durante o processo, compreender o seu modo de funcionamento. Ao desenvolvermos o projeto passo a passo ganhámos conhecimento sobre como funciona a arquitetura de Harvard e a relação entre *CPU*, *RAM* e *ROM*. Aprendemos sobre os conceitos Módulo Funcional e Módulo de Controlo. Após projetar o *CPU* foi nos pedido que simulássemos o seu funcionamento, utilizando o microprocessador Arduino.

O desenvolvimento deste projeto foi difícil, devido à complexidade de um *CPU*, mas achamos que foi bem sucedido. A simulação foi um processo trabalhoso. Traduzir as tabelas, desenhos do modulo funcional e modulo controlo para código demorou mas felizmente quando os bugs foram corrigidos o funcionamento foi o esperado. Conseguimos utilizar as nossas instruções para realizar 4 pequenos programas.

5 Bibliografia

1. Folhas de Computação Física - Jorge Pais, 2021/22
2. Projeto de um *CPU* Passo a Passo - Jorge Pais, 2021/22

6 Código Arduíno

```
int RAM[256];
int ROM[128];

#define tempoRuido 200

const int ROM_MC[64]= {0x621, 0x621, 0x621, 0x621, 0x621, 0x621, 0x621, 0x621,
                       0x600, 0x600,0x1740,0x3E00,0x1740,0x1740,0x3F40,0x3F40,
                       0x780, 0x620, 0x200, 0x500, 0x600, 0x600, 0x600, 0x600,
                       0x610, 0x610, 0x610, 0x610, 0x610, 0x610, 0x610, 0x610,
                       0x608, 0x608, 0x608, 0x608, 0x608, 0x608, 0x608, 0x608,
                       0x604, 0x604, 0x604, 0x604, 0x604, 0x604, 0x604, 0x604,
                       0x602, 0x602, 0x602, 0x602, 0x602, 0x602, 0x602, 0x602,
                       0x602, 0x602, 0x602, 0x602, 0x602, 0x602, 0x602, 0x602};


int DPC, QPC = 0;     // registo PC
int DRn[2], QRn[2], ARn;  // conjunto de 2 registos
int DA, QA;               // registo A
int const5, rel6, end7;
int S;                    // Saída da ALU
int addressMC;
long unsigned int t0, t;

bool DC, QC;    // flag Cy ou Borrow
bool DZ, QZ;    // flag zero
bool DOV, QOV;    // flag overflow
bool EnAux, EnPC, EnRn, EnA, EnA0, EnA1, EnC,EnOV,EnZ,JC,JNZ, JOV, JMP, WR, RD;
bool clk;
bool d3, d4, d5, d6, d7, d8;


void setup(){
  Serial.begin(9600);
  attachInterrupt(0, clock, RISING);  //d2
  t=millis();

  program();    // TESTE DO ÚLTIMO SLIDE DO MOODLE

  //program2(); // SUBTRATOR

  //program3(); // TESTE A OPERAÇÕES LÓGICAS

  //program4(); // TESTE AOS JUMPS

}
```

```
void loop(){
  circuitoSequencial();
  circuitoCombinatorio();
  moduloControlo();
  escreverSaidas();
}

void circuitoSequencial(){
  if (clk){
    //Registo A
    registos5bits(EnA, DA, &QA);

    //Conjunto de registo RN
    ARn = ROM[QPC] & 0x01;
    registos5bits(EnRn, DRn[ARn], &QRn[ARn]);

    // Contador PC
    contador7bits(EnPC, DPC, &QPC);

    // Carry
    registo1bit(EnC, DC, &QC);

    // OV
    registo1bit(EnOV, DOV, &QOV);

    // Zero
    registo1bit(EnZ, DZ, &QZ);

    clk=false;
  }
}

void circuitoCombinatorio(){
    // definição das variáveis
    // bits e0 -> e6 (d0 - d6)
    end7   = ROM[QPC] & 0x07F;

    // bits c0 -> c4 (d1 - d5)
    const5 = ROM[QPC] & 0x03E;

    // bits r0 -> r5  (d0 - d5)
    rel6   = ROM[QPC] & 0x03F;

    //distinção entre R0 e R1
    ARn = ROM[QPC] & 0x01 == 0x01;

    //bits que distinguem entre as operação da ALU
    d3 = ROM[QPC] & 0x008;
    d4 = ROM[QPC] & 0x010;
    d5 = ROM[QPC] & 0x020;
```

```
    // Mux do Registo Rn
    mux2x1(EnAux, QA, const5, &DRn[ARn]);

    // Mux do Registo A
    mux3x1(EnA, EnA1, EnA0, S, QRn[ARn], RAM[QRn[ARn]], &DA);

    //bits que distinguem entre as operação da ALU
    d6 = ROM[QPC] & 0x040;
    d7 = ROM[QPC] & 0x080;
    d8 = ROM[QPC] & 0x100;

    // Arithmetic Logic Unit
    ALU(d3 , d4 , d5 , QC, QOV, QZ, QA, QRn[ARn], &DC, &DOV, &DZ, &S);

    // Registo na RAM (@Rn, A)
    registos8bits(!WR, QA, &RAM[QRn[ARn]]);

}
//Simulação do Sinal EnPC
void circuitoCombinatorioMC(){
    EnPC = JNZ && !QZ || JC && QC|| JOV && QOV || JMP;
}

void validarSinaisMC(){

//address dos bits dependentes/bits de entrada no Modulo de Controlo

    addressMC = ((ROM[QPC] & 0x1F8) >> 3); // D8 -> D3

    EnZ  = (ROM_MC[addressMC] & 0x1000) == 0x1000;
    EnC  = (ROM_MC[addressMC] & 0x2000) == 0x2000;
    EnOV = (ROM_MC[addressMC] & 0x0800) == 0x0800;
    WR   = (ROM_MC[addressMC] & 0x0400) == 0x0400;
    RD   = (ROM_MC[addressMC] & 0x0200) == 0x0200;
    EnA  = (ROM_MC[addressMC] & 0x0100) == 0x0100;
    EnA0 = (ROM_MC[addressMC] & 0x080) == 0x080;
    EnA1 = (ROM_MC[addressMC] & 0x040) == 0x040;
    EnRn = (ROM_MC[addressMC] & 0x020) == 0x020;
    JC   = (ROM_MC[addressMC] & 0x010) == 0x010;
    JNZ  = (ROM_MC[addressMC] & 0x08) == 0x08;
    JOV  = (ROM_MC[addressMC] & 0x04) == 0x04;
    JMP  = (ROM_MC[addressMC] & 0x02) == 0x02;
    EnAux= (ROM_MC[addressMC] & 0x01) == 0x01;
}
```

```
void moduloControlo(){
  validarSinaisMC();
  circuitoCombinatorioMC();
}

// Simulação do CLOCK
void clock(){
  if(filtroRuido()) clk=true;
}

bool filtroRuido(){
  t0 = t;
  t = millis();
  return((t - t0) > tempoRuido);
}

// Mux RegistoRn
void mux2x1(bool EnAux, int i1, int i0, int *y){
  if (EnAux) *y=i0;
  else *y=i1;
}

// Registos A e Rn
void registos5bits(bool enable, int D, int *Q){
  if (enable) *Q=D & 0x1F;
}

void registos8bits(bool enable, int D, int*Q){
  if (enable) *Q=D & 0xFF;
}

// Mux Registo A
void mux3x1(bool enable, bool a1, bool a0, int i2, int i1, int i0, int *y){
  if (a0 && !a1) *y=i1;
  else if (!a0 && !a1) *y=i0;
  else if (!a0 && a1) *y=i2;
}
```

```
// ALU
void ALU(bool d3, bool d4, bool d5, bool cyin, bool ovin, bool zin,
         int a, int b, bool *cyout, bool *ovout, bool *zero, int *s){

  if (d3 && d4 && !d5){  //CPLF
    *cyout=!cyin;
    *ovout=!ovin;
    *zero= !zin;
  }
  else{   //Intructions:
    if (!d3 && d4 && !d5) *s = !a & 0x1F;             //NOT
    if (!d3 && !d4 && d5) *s = (a & b) & 0x1F;        //AND
    if (d3 && !d4 && d5) *s = (a | b) & 0x1F;         //OR
    if (!d3 && d4 && d5)  *s= (a + b + cyin) & 0x1F;  //ADD
    if (d3 && d4 && d5) *s = (a - b - cyin) & 0x1F;   //SUBB
  }
  //Flags
  if (*s == 0 ) *zero = 1;
  else *zero = 0;
  if (*s > 15 ) *ovout = 1;
  else *ovout = 0;
  *cyout = (*s & 0x20) == 0x20;
}

//Somador rel6 do PC
void somador7bits(int A, int B, int *Y){
  *Y = A + B;
}

// PC
void contador7bits(bool enable, int D, int *Q){
  if (enable){
    if(JMP) *Q = end7 & 0xFF;
    else if (JC | JNZ | JOV) *Q = (*Q + rel6) & 0xFF;

  } else *Q = (*Q + 1) & 0xFF;

}

// Resgistos do Carry, OV e Zero
void registo1bit(bool enable, bool D, bool *Q){
  if (enable) *Q=D;
}
```

```
void program(){
 // Programa que testa o acesso à Memoria de Dados (RAM)
 // e testa a função aritmética da adição

  RAM[2] = 5; // X
  RAM[4] = 0x1F; // Y
  RAM[6] = 0; // Z

  ROM[0]  = 0x002; // MOV R0, 2               | Rn[0] = const5 (=2)
  ROM[1]  = 0x09E; // MOV A, @R0 (A = X)      | A = MD(Rn[0])
  ROM[2]  = 0x08F; // MOV R1, A (R1 = X)      | Rn[1] = A
  ROM[3]  = 0x000; // MOV R0, 0               | Rn[0] = const5 (=0)
  ROM[4]  = 0x086; // MOV A, R0 (A = R0 = 0). | A = Rn[0]
  ROM[5]  = 0x076; // ADDC A, R0 (CY = 0)     | A = A + Rn[0] + C
  ROM[6]  = 0x004; // MOV R0, 4               | Rn[0] = const5 (=4)
  ROM[7]  = 0x09E; // MOV A, @R0 (A = Y)      | A = Rn[0]
  ROM[8]  = 0x077; // ADDC A, R1 (A = X+Y+0 ) | A = A + Rn[1] + C
  ROM[9]  = 0x006; // MOV R0, 6               | Rn[0] = const5 (=2)
  ROM[10] = 0x096; // MOV @R0, A (Z = X+Y)    | MD(Rn) = A
  ROM[11] = 0x180; // JMP 0                   | PC = end7

}
void program2(){
// Programa que testa o funcionamento das funções aritméticas do CPU,
//em particular testa a subtração entre dois números

  ROM[0]  = 0x00A; // MOV R0, 10           | Rn[0] = const5 (=10)
  ROM[1]  = 0x086; // MOV A, R0 (A=R0=10)  | A = Rn[0]
  ROM[2]  = 0x004; // MOV R0, 4            | Rn[0] = const5 (=4)
  ROM[3]  = 0x07E; // SUBB A, Rn1          | A = A { Rn[1] { C (C=0)
  ROM[4]  = 0x006; // MOV  R0, 6           | Rn[0] = const5 (=6)
  ROM[5]  = 0x07E; // SUBB A, Rn1          | A = A { Rn[1] { C (C=0)
  ROM[6]  = 0X10A; // JNZ  Rel 6           | Se (Z/) PC += rel6 (Rel6= 10)
  ROM[16] = 0x180; // JMP 0                | PC = end7

}

void program3(){
//Programa que testa o funcionamento das operações lógicas da ALU

  ROM[0] = 0x002; // MOV R0, 2     | Rn[0] = const5 (=2)
  ROM[1] = 0x086; // MOV A, R0     | A = Rn[0]
  ROM[2] = 0X06E; // OR  A, Rn     | A = A or Rn
  ROM[3] = 0x086; // MOV A, R0     | A = Rn[0]
  ROM[4] = 0X066; // AND A, Rn     | A = A and Rn
  ROM[5] = 0x057; // NOT A         | A = A/
}
```

```
void program4(){
//Programa para testar o funcionamento dos jumps do CPU

  ROM[0]  = 0x004; // MOV R0, 4     | Rn[0] = const5 (=4)
  ROM[1]  = 0x086; // MOV A, R0     | A = Rn[0]
  ROM[2]  = 0x009; // MOV R1, 8     | Rn[1] = const5 (=8)
  ROM[3]  = 0x077; // ADDC A, R1    | A = A + Rn[1] + C
  ROM[4]  = 0x104; // JNZ 4         | Se (Z/) PC += rel6
  ROM[5]  = 0x002; // MOV R0, 0     | NÃO EXECUTA
  ROM[8]  = 0x01E; // MOV R0, 0     | Rn[0] = const5 (=0)
  ROM[9]  = 0x086; // MOV A, R0     | A = Rn[0]
  ROM[10] = 0x01F; // MOV R1, 30    | Rn[1] = const5 (=30)
  ROM[11] = 0x077; // ADDC A, R1    | A = A + Rn[1] + C
  ROM[12] = 0x0C4; // JC 4          | Se (C) PC += rel6
  ROM[16] = 0x145; // JOV 5         | Se (Ov) PC += rel6
  ROM[21] = 0x05F; //CPLF           | C=C/;Ov=Ov/;Z=Z/
  ROM[22] = 0X180; // JMP 0         | PC = end7
}

void escreverSaidas(){
//As saídas só são escritas na consola se houver um clock

  if(clk){
    Serial.print("DBROM: 0x");Serial.print(ROM[QPC],16);
    Serial.print("; ROM_MC: 0x");Serial.println(ROM_MC[addressMC] ,16);

    String s = "QPC: " + String(QPC) +"; DRn[0]: " + String(DRn[0])+";
    QRn[0]:" + String(QRn[0]) +"; DRn[1]: " + String(DRn[1])+";
    QRn[1]: " + String(QRn[1]) +"; DA: " + String(DA) + " ;
    QA: " + String(QA)+ " ; Cyout: " + String(QC)+ " ;
    Ov: " + String(QOV)+ " ; Zero: " + String(QZ) + " ;
    S: " + String(S) + " ; RAM[QRn["+String(ARn)+"]]: " + String(RAM[QRn[ARn]]);

    Serial.print(s);
    Serial.println();
  }
}
```