ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

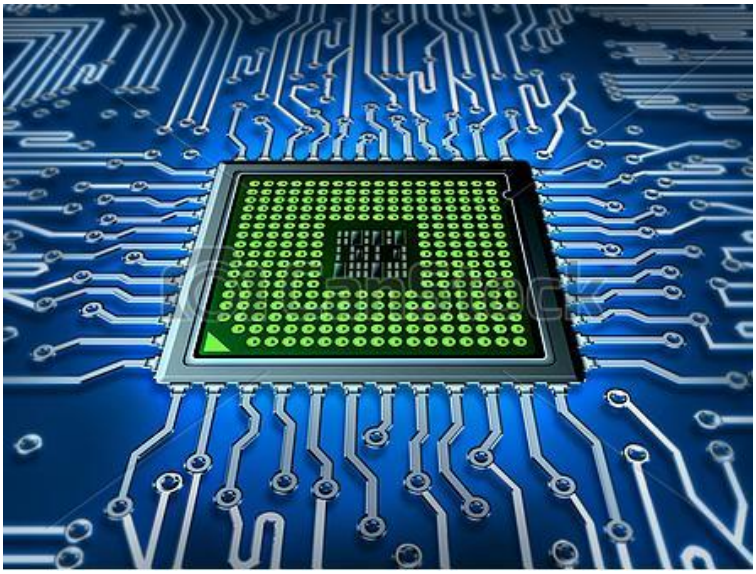
Licenciatura em Engenharia Informática e Multimédia (LEIM)

Computação Física – 1819SV

Trabalho Prático nº2

Engº Carlos Carvalho

Turma LEIM23D



Luís Fonseca	nº 45125
Tiago Oliveira	nº 45144
Rodrigo Correia	nº 45155

10/05/2019

Índice

Índice Figuras e Tabelas

Objetivos/Introdução

O tema deste trabalho consistia na criação de um microprocessador, baseado numa arquitetura Harvard, que fosse capaz de realizar as 12 instruções apresentadas no enunciado. Inicialmente optou-se por realizar a codificação, de cada instrução. Como resultado obteve-se uma codificação de 4 bits, diferenciada, para as 12 instruções.

De seguida, procedemos ao desenho do módulo funcional tendo em mente a técnica de encaminhamento de dados. Após projetar o módulo funcional, passámos ao módulo de controlo onde definimos as suas entradas e saídas.

O último passo antes de passarmos à simulação do microprocessador foi construir a EPROM, sendo neste caso de 16x12 que implementasse o módulo de controlo.

Por fim, foi feita uma implementação em Arduino com futura realização de testes para a verificação do correto funcionamento do microprocessador criado.

Antes de passar para a construção do microprocessador, o grupo definiu um conjunto de objetivos que irá ajudar na construção do microprocessador:

1. Especificar a quantidade de bits de cada um dos registos;

2. Especificar a quantidade de bits dos Address Bus e Data Bus das memórias de código e de dados;

3. Especificar as instruções

4. Codificar as instruções usando o menor número possível de bits;

5. Desenhar o módulo funcional, baseado na técnica de encaminhamento de dados;

6. Especificar as entradas e as saídas do Módulo de Controlo;

7. Realizar a tabela de programação de uma ROM que implementa o Módulo de Controlo;

8. Simular a arquitetura desenhada no Arduino;

9. Verificar a correta operação da arquitetura, realizando pequenos programas de teste que utilizem todas as instruções do CPU.

Desenvolvimento

1– Quantidade de bits – Registos

O primeiro passo para a realização do processador consistia em olhar para as instruções, e retirar a quantidade de bits que cada registo apresentava. Através das instruções fornecidas, é possível retirar a quantidade de bits para cada registo interno, do Address Bus (AB) e do Data Bus(DB), tanto para a memória de código, como para a memória de dados. Estas informações iram ser úteis para a codificação das instruções e no desenho no módulo funcional.

Registo V: 8bits

Registo R: 6bits

Registo A: 8bits

C : Flag Carry(1 bit)

Z : Flag Zero(1bit)

OV : Flag Overflow(1bit)

Rel5 : valor relativo a 5bits

PC : Program Counter (6bits)

Em hardware, um registo é um dispositivo que é implementado à custa de flip-flops tipo D edge-triggered ou D-Latch e que permite registar ou memorizar informação binária (Registo R e A). Um registo bidirecional é desenhado a partir de um registo latch e de portas lógicas tri-state(Registo V).

2 – Quantidade de bits do Address Bus e do Data Bus

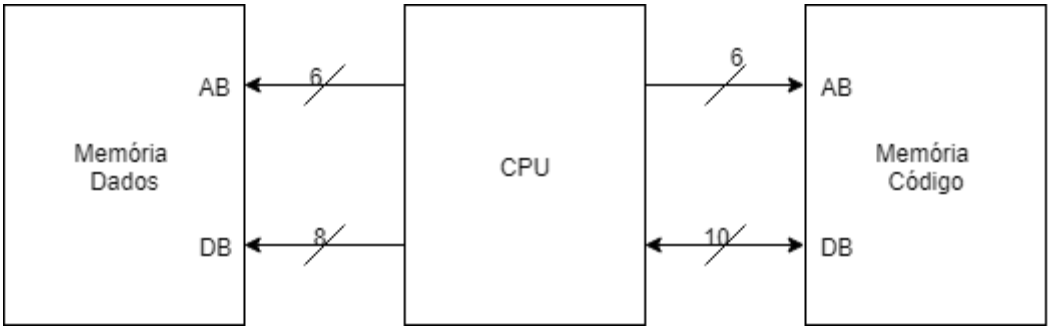


Figura 1 - barramento dos bits entre a Memória de Código e a Memória de Dados

AB – Address Bus
Barramento de Endereços, usado para especificar endereços físicos na memória.

DB – Data Bus
Barramento de Dados, usado para transmitir dados entre os componentes.

Na memória de dados:
DB = 6bits e o AB = 8bits

Na memória de código:
DB = 10bits e o AB = 6bits

A quantidade de registos internos define o número de bits necessários para o endereço de modo a que cada registo tenha um endereço distinto dos outros, este conjunto de bits é designado por barramento de endereços (Address bus).A quantidade de bits de cada registo define a dimensão do conjunto de sinais por onde flui a informação para a RAM, este conjunto de sinais é designado por barramento de dados (data bus).

3– Especificação das instruções

Nas primeiras instruções temos os dois primeiros registos, V e R no qual o registo V vai ter 8 bits e o registo R vai ter 6 bits. A 3ª instrução consiste em colocar no registo A o valor do registo V. A 4ª instrução tem como função de colocar no registo V o valor de posição de memória do registo R. A 5ª instrução é por sua vez semelhante a 4ª instrução, coloca no valor de posição de memória no qual se encontra o registo R o valor do registo V. As próximas 3 operações são operações aritméticas, no qual os registos que serão utilizados vão ser o registo A e o registo V. As 3 instruções seguintes, JNC,JZ e JOV irão alterar o valor do *Program Counter* (PC) caso exista *Flag* de *Not Cy* ou *Flag* de *Z* ou *Flag* de *Overflow* respetivamente. Por último, JMP é realizada quando se pretende realizar uma alteração absoluta do valor do PC.

Instrução	Funcionalidade
MOV V, #const8	V = const8
MOV R, #const6	R = const6
MOV A, V	A = V
MOV V, @R	V = M(R)
MOV @R, V	M(R) = V
NOR V, A	V = (V + A)\
ADC V, A	V = V + A + Cy
SBB V, A	V = V – A - Bw
JNC rel5	Se (!Cy) PC += rel5
JZ rel5	Se (Z) PC += rel5
JOV rel5	Se (OV) PC += rel5
JMP rel6	PC end6

Tabela 1 - Instruções microprocessador

4 - Codificação das instruções

Com a tabela de instruções fornecida, fez-se uma codificação a 10bits, no qual os bits D9,D8,D7,D6 vão ser usados para distinguir todas as instruções. Depois da codificação tratada, foi calculado o valor do Data, este parâmetro irá servir para a implementação no Arduino para cada instrução.

Inst.	Parâm.	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	Data
MOV V,#const8	const 8	0	0	C7	C6	C5	C4	C3	C2	C1	C0	[0h,FFh]
MOV R,#const6	const 6	0	1	0	0	C5	C4	C3	C2	C1	C0	[100h,13Fh]
MOV A,V	-	0	1	0	1	0	0	0	0	0	0	140h
MOV V,@R	-	0	1	1	0	0	0	0	0	0	0	180h
MOV @R,V	-	0	1	1	1	0	0	0	0	0	0	1C0h
NOR V,A	-	1	0	0	0	0	0	0	0	0	0	200h
ADC V,A	-	1	0	0	1	0	0	0	0	0	0	240h
SBB V,A	-	1	0	1	0	0	0	0	0	0	0	280h
JNC rel5	rel5	1	1	0	0	0	R4	R3	R2	R1	R0	[300h,31Fh]
JZ rel5	rel5	1	1	0	1	0	R4	R3	R2	R1	R0	[340h,35Fh]
JOV rel5	rel5	1	1	1	0	0	R4	R3	R2	R1	R0	[380h,39Fh]
JMP end6	end6	1	1	1	1	E5	E4	E3	E2	E1	R0	[3C0h,3FFh]

Tabela 2- codificação a 10 bits

5 – Módulo Funcional

A estrutura aqui apresentada é o módulo funcional do CPU, tendo sido essencial no desenvolver do projeto pois foi a este a quem se recorreu para termos noção com o que estávamos a lidar, assim como a maneira como o CPU funcionava como um todo. Os seus dispositivos base em termos de hardware são o registo e o multiplexer, termos então usado alguns de cada, nomeadamente 4 registos (PC, R, V e A), 2 multiplexers 2x1 (que na realidade são 1 multiplexers para cada um dos bits de para cada uma das operações que necessitam deste tipo de multiplexers) e um multiplexer 4x1 (que na realidade são 8 multiplexers para cada uma das operações que necessitam deste tipo de multiplexers).

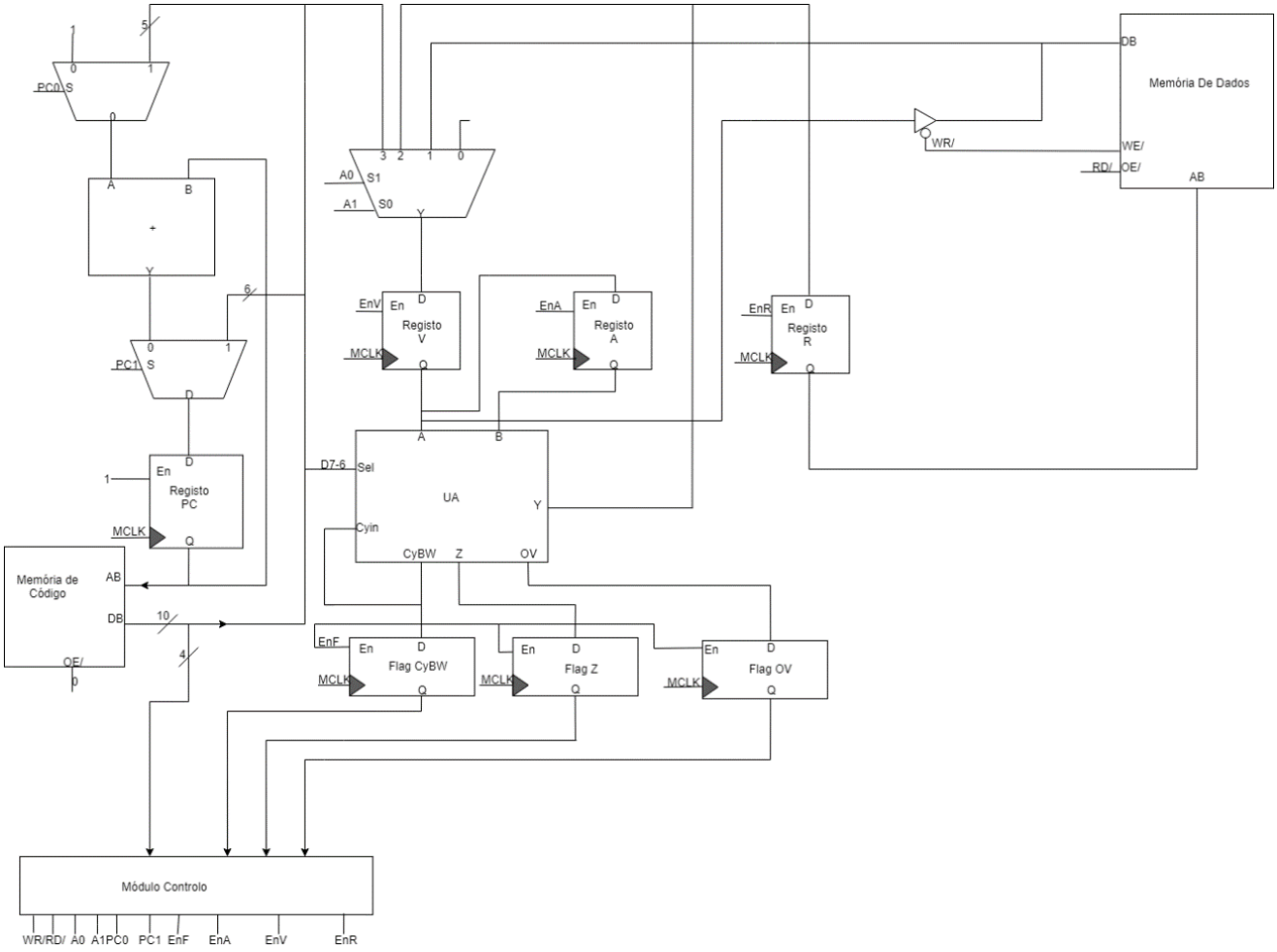


Figura 2 - módulo funcional

6 – Módulo de Controlo e tabela com os sinais ativos

Feito o módulo funcional, passou-se à construção do módulo de controlo, o qual vai ter como entradas, os bits D9, D8, D7, D6 (provenientes do Data Bus da memória de código), o CyBw,Z e o OV (provenientes das suas respetivas *flags*). Já para as saídas temos os dois seletores para os multiplexers que faram as operações sobre o Program Counter, os bits de saída A0 e A1 (também seletores de multiplexers), os Enables dos registos V, A, R e das *flags* Not Cy, Z, OV e o WR\(write) no qual quando ativa (zero lógico) indica se está a escrever na memória de dados e o RD\(read) que quando ativa (zero lógico) indica que se está a ler valores provenientes da memória de dados. Após termos definido as entradas e saídas do módulo de controlo conseguimos representar a ativação dos sinais através da seguinte tabela de verdade.

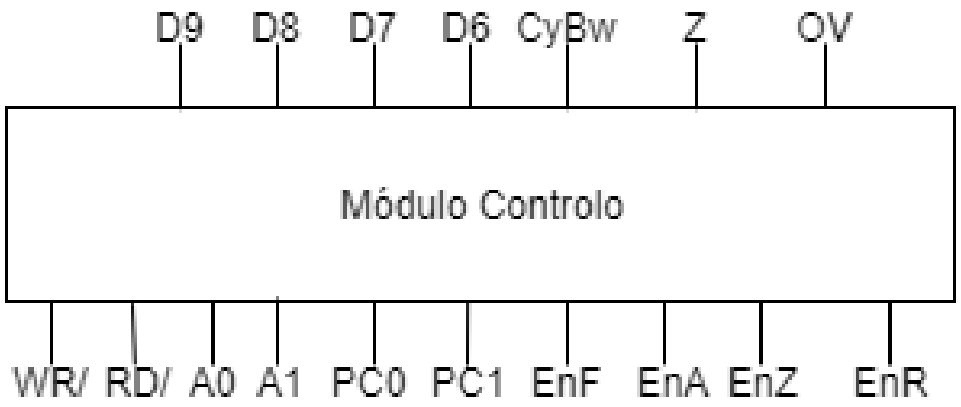


Figura 3- módulo de controlo

- **Tabela com os sinais ativos do módulo de controlo**

Instrução	D9	D8	D7	D6	CyBw	Z	OV	Sinais Ativos
MOV V , #const8	0	0	-	-	-	-	-	EnV , A0 , A1
MOV R , #const6	0	1	0	0	-	-	-	EnR
MOV A, V	0	1	0	1	-	-	-	EnA
MOV V , @R	0	1	1	0	-	-	-	EnV , RD\ ,A1
MOV @R , V	0	1	1	1	-	-	-	WR\
NOR V , A	1	0	0	0	-	-	-	EnV, A0
ADD V , A	1	0	0	1	-	-	-	EnV, EnF, A0
SUBB V ,A	1	0	1	0	-	-	-	EnV , EnF, A0
JNC rel5	1	1	0	0	0	-	-	PC0
JNC rel5	1	1	0	0	1	-	-	X
JZ rel5	1	1	0	1	-	0	-	PC0
JZ rel5	1	1	0	1	-	1	-	X
JOV rel5	1	1	1	0	-	-	0	PC0
JOV rel5	1	1	1	0	-	-	1	
JMP end6	1	1	1	1	-	-	-	PC1

Tabela 3- Tabela com Sinais Ativos

7 – EPROM 16x12

Com a tabela feita dos sinais ativos das diferentes instruções, prosseguiu-se para o último passo antes da implementação do CPU no Arduino, calcular o valor da data e do address. O cálculo do address foi feito à custa da codificação feita nos sinais de entrada, já a data foi feita através dos sinais de saída, ambos provenientes do módulo de controlo. Foi necessário o cálculo dos mesmo na base hexadecimal. A construção da respetiva tabela foi feita à custa de uma EPROM (*Erasable programable read only memory*) no qual, consiste num chip de memória que retém os dados quando o aparelho é desligado.

Nota: Devido ao comprimento da tabela EPROM feita, teve de ser divida em duas partes, uma para as entradas, outra para as saídas.

- **Tabela com os sinais de entrada da EPROM**

	D9	**D8**	**D7**	**D6**	
Instruções	**A3**	**A2**	**A1**	**A0**	**Endereços**
MOV V,#const8	0	0	-	-	[0, 3]
MOV R,#const6	0	1	0	0	4
MOV A,V	0	1	0	1	5
MOV V,@R	0	1	1	0	6
MOV @R,V	0	1	1	1	7
NOR V,A	1	0	0	0	8
ADC V,A	1	0	0	1	9
SBB V,A	1	0	1	0	A
JNC rel5	1	1	0	0	C
JZ rel5	1	1	0	1	D
JOV rel5	1	1	1	0	E
JMP end6	1	1	1	1	F

Tabela 4 – Entradas e Endereços

- **Tabela com os sinais de saída da EPROM**

EnF	WR/	RD/	EnV	EnA	EnR	A0	A1	JMP	JNC	JZ	JOV	
D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	DATA
0	1	1	1	0	0	1	1	0	0	0	0	730h
0	1	1	0	0	1	0	0	0	0	0	0	840h
0	1	1	0	1	0	0	0	0	0	0	0	680h
0	0	1	1	0	0	0	1	0	0	0	0	320h
0	1	0	0	0	0	0	0	0	0	0	0	400h
0	1	1	1	0	0	1	0	0	0	0	0	720h
1	1	1	1	0	0	1	0	0	0	0	0	F20h
1	1	1	0	0	0	1	0	0	0	0	0	E20h
0	1	1	0	0	0	0	0	0	1	0	0	604h
0	1	1	0	0	0	0	0	0	0	1	0	602h
0	1	1	0	0	0	0	0	0	0	0	1	601h
0	1	1	0	0	0	0	0	1	0	0	0	608h

Tabela 5 - Saídas e Data

O sinal PC0 obteve-se a partir dos sinais gerados na EPROM designados por JZ, JNC, JOV e das flags de Zero, Cy e Ov do CPU através da seguinte expressão lógica:

PC0 = JNC. $\overline{Carry}$ + JZ.Zero + JOV.Overflow

O sinal PC1 obtém-se a partir do JMP:

PC1 = JMP

Esquema de ligações do Arduíno

Antes de passar para os testes para gerar as instruções do CPU, o grupo realizou um diagrama de ligações no Arduino. Associando ao botão utilizamos a resistência interna do Arduino, configurando o pino como INPUT_PULLUP. Este botão tem como função realizar o movimento ascendente de um sinal clock, e atualizando as instruções quando pressionado.

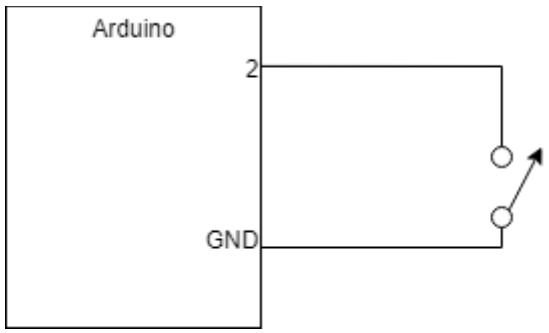


Figura 4- esquema de ligação entre um arduino e um botão

Código Arduino

```
// SAÍDAS
bool RD, WR, EnA, EnV, EnR, EnF, R1, R0, JMP, JNC, JZ, JOV, PC_0,
PC_1;

//Codificacao
bool D9, D8, D7, D6, D5, D4, D3, D2, D1, D0;

//variaveis instrucoes
byte relativo3, relativo2, relativo1, relativo0, end6, Constante8,
Constante6;
int relativo5, relativo4;

//entradas e saidas registos/flags
volatile byte  Q_V, Q_R, Q_A, Q_PC, Q_Ov, Q_Cy, Q_Z;
byte D_V, D_R, D_A, D_PC, D_Ov, D_Cy, D_Z, QALU, AUX1, AUX0;;

//variaveis clock
int tempoClock = 250;
unsigned long tN_Clock, tB_Clock, tN_MClock, tB_MClock;

word memCodigo[64];
byte memDados[128];
word moduloControlo[16];


void setup() {

  Serial.begin(9600);
  pinMode(2, INPUT_PULLUP);
  EPROM();
  Instrucoes();
  attachInterrupt(digitalPinToInterrupt(2), MclkN, RISING);
  interrupts();

}

void loop() {

  MF();
  visualizacoes();

}
```

```
void visualizacoes() {

  if (Serial.available()) {
    char recebido = Serial.read();

    switch (recebido) {
      case '1':
        registos();
        break;
      case '2':
        variaveis();
        break;
      case '3':
        saidas();
        break;
      case '4':
        memoriaDados();
        break;
      case '5':
        memoriaCodigo();
        break;
    }
  }
}

void registos() {

  Serial.println();
  Serial.println("Registos");
  Serial.println("");
  Serial.print("V: "); Serial.println(Q_V , HEX);
  Serial.print("R: "); Serial.println(Q_R , HEX);
  Serial.print("A: "); Serial.println(Q_A , HEX);
  Serial.print("PC: "); Serial.println(Q_PC , HEX);
  Serial.println(" ");

}
```

```
void saidas() {
  Serial.println();
  Serial.println("Saídas:");
  Serial.println("");

  Serial.print("EnR: " );
  Serial.println(EnR);
  Serial.print("EnV: " );
  Serial.println(EnV);
  Serial.print("EnA: " );
  Serial.println(EnA);
  Serial.print("EnF: " );
  Serial.println(EnF);
  Serial.print("R1: " );
  Serial.println(R1);
  Serial.print("R0: " );
  Serial.println(R0);
  Serial.print("RD: " );
  Serial.println(RD);
  Serial.print("WR: " );
  Serial.println(WR);
  Serial.print("JMP: " );
  Serial.println(JMP);
  Serial.print("JNC: " );
  Serial.println(JNC);
  Serial.print("JZ: " );
  Serial.println(JZ);
  Serial.print("JOV: " );
  Serial.println(JOV);

  Serial.println(" ");
  Serial.println();
}

void variaveis() {
  Serial.println();
  Serial.println("Variaveis:");
  Serial.println("");

  Serial.print("Const8: " ); Serial.println(Constante8 , HEX);
  Serial.print("Const6: " ); Serial.println(Constante6 , HEX);
  Serial.print("Rel5: " ); Serial.println(relativo5 , HEX);
  Serial.print("End6: " ); Serial.println(end6 , HEX);

  Serial.println(" ");
  Serial.println();
}
```

```
void memoriaCodigo() {

  Serial.println();
  Serial.println("memoriaCodigo:");
  Serial.println("");

  for (int i = 0; i < sizeof(moduloControlo); i++) {
    Serial.print(i, HEX);
    Serial.print(" : 0x");
    Serial.println(moduloControlo[i], HEX);
  }

  Serial.println(" ");
  Serial.println();
}


void memoriaDados() {
  Serial.println();
  Serial.println("memoriaDados:");
  Serial.println("");

  for (int i = 0; i < sizeof(memDados); i++) {
    Serial.print(i, HEX);
    Serial.print(" : 0x");
    Serial.println(memDados[i], HEX);
  }

  Serial.println(" ");
  Serial.println();
}

void Mclk() {

  tN_Clock = millis();
  if ( (tN_Clock - tB_Clock) > tempoClock) {
    Q_PC = D_PC;

    Serial.print("Instrução: "); Serial.println(memCodigo[Q_PC], HEX);
    Serial.print("Registo PC: "); Serial.println(Q_PC);

    attachInterrupt(digitalPinToInterrupt(2), MclkN, FALLING);
  }

  tB_Clock = tN_Clock;
}
```

```
void MclkN() {

  tN_MClock = millis();
  if (tN_MClock - tB_MClock > tempoClock) {

    Q_R   = reg(Q_R, D_R, EnR);
    Q_V   = reg(Q_V, D_V, EnV);
    Q_A   = reg(Q_A, D_A, EnA);
    Q_Z  = reg(Q_Z, D_Z, EnF);
    Q_Cy = reg(Q_Cy, D_Cy, EnF);
    Q_Ov = reg(Q_Ov, D_Ov, EnF);

    Serial.println();
    Serial.print("V = ");
    Serial.println(Q_V, HEX);
    Serial.print("R = ");
    Serial.println(Q_R, HEX);
    Serial.print("A = ");
    Serial.println(Q_A, HEX);
    Serial.print("@R = ");
    Serial.println(memDados[Q_R], HEX);
    Serial.println("FLAGS");
    Serial.print("Z = ");
    Serial.println(Q_Z);
    Serial.print("Cy/Bw = ");
    Serial.println(Q_Cy);
    Serial.print("Ov = ");
    Serial.println(Q_Ov);
    Serial.println();
    Serial.println();
    attachInterrupt(digitalPinToInterrupt(2), Mclk, RISING);
  }
}


byte reg(byte A, byte B, bool Enable) {

  switch (Enable) {
    case 0:
      return A;
    case 1:
      return B;
  }
}


byte somador(byte A, byte B) {

  return A + B;
}
```

```
byte ALU(byte A, byte B, boolean Q_Cy, bool S1, bool S0, byte &D_Ov,
byte &D_Z, byte &D_Cy) {

  byte AU_AUX;
  int Sum_AUX, Sub_AUX;

  switch (S1 << 1 | S0) {

    case  B00:

      AU_AUX = ~(A | B);

      D_Z = (AU_AUX == 0) ? 1 : 0;

      break;

    case B01:

      Sum_AUX = A + B + Q_Cy;

      D_Cy = ( Sum_AUX > 255) ? 1 : 0;

      AU_AUX = (byte) Sum_AUX & 0xFF;

      D_Z = (AU_AUX == 0) ? 1 : 0;

      D_Ov = ((A < 0 && B < 0 && Sum_AUX > 0) || (A > 0 && B > 0 &&
Sum_AUX < 0)) ? 1 : 0;

      break;

    case B10:

      Sub_AUX = A - B - Q_Cy;

      D_Cy = (Sub_AUX < 0) ? 1 : 0;

      AU_AUX = (byte) Sub_AUX & 0xFF;

      D_Z = (AU_AUX == 0) ? 1 : 0;

      D_Ov = ((A < 0 && B < 0 && Sub_AUX > 0) || (A > 0 && B > 0 &&
Sub_AUX < 0)) ? 1 : 0;

      break;

  }
  return AU_AUX;

}
```

```
void MF() {

  int memCodigoI = 0;

  memCodigoI |= ((bitRead(memCodigo[Q_PC], 9) << 3) |
(bitRead(memCodigo[Q_PC], 8) << 2) | (bitRead(memCodigo[Q_PC], 7) <<
1) |
                 (bitRead(memCodigo[Q_PC], 6) << 0));

  int rel5F = B00011111 & memCodigo[Q_PC];

  relativo4 = (bitRead(rel5F, 4) * - 16);
  relativo3 = (bitRead(rel5F, 3) * 8);
  relativo2 = (bitRead(rel5F, 2) * 4);
  relativo1 = (bitRead(rel5F, 1) * 2);
  relativo0 = (bitRead(rel5F, 0) * 1);

  D9 = bitRead(memCodigo[Q_PC],  9);
  D8 = bitRead(memCodigo[Q_PC],  8);
  D7 = bitRead(memCodigo[Q_PC],  7);
  D6 = bitRead(memCodigo[Q_PC],  6);
  D5 = bitRead(memCodigo[Q_PC],  5);
  D4 = bitRead(memCodigo[Q_PC],  4);
  D3 = bitRead(memCodigo[Q_PC],  3);
  D2 = bitRead(memCodigo[Q_PC],  2);
  D1 = bitRead(memCodigo[Q_PC],  1);
  D0 = bitRead(memCodigo[Q_PC],  0);

  JOV  = bitRead(moduloControlo[memCodigoI],   11);
  JZ   = bitRead(moduloControlo[memCodigoI],   10);
  JNC  = bitRead(moduloControlo[memCodigoI],   9);
  JMP  = bitRead(moduloControlo[memCodigoI],   8);
  R1   = bitRead(moduloControlo[memCodigoI],   7);
  R0   = bitRead(moduloControlo[memCodigoI],   6);
  EnR  = bitRead(moduloControlo[memCodigoI],   5);
  EnA  = bitRead(moduloControlo[memCodigoI],   4);
  EnV  = bitRead(moduloControlo[memCodigoI],   3);
  RD   = bitRead(moduloControlo[memCodigoI],   2);
  WR   = bitRead(moduloControlo[memCodigoI],   1);
  EnF  = bitRead(moduloControlo[memCodigoI],   0);

  PC_0 = ( (JNC & !Q_Cy) | (JZ & Q_Z) | (JOV & Q_Ov));
  PC_1 = JMP;

  relativo5 = relativo4 + relativo3 + relativo2 + relativo1 +
relativo0;

  end6 = B00111111 & memCodigo[Q_PC];

  AUX0 = mux_2x1(1, relativo5, PC_0);

  AUX1 = somador(AUX0, Q_PC);

  D_PC = mux_2x1(AUX1, end6, JMP);

  D_R = end6;
```

```
  Constante8 = B11111111 & memCodigo[Q_PC];

  Constante6 = B00111111 & memCodigo[Q_PC];

  QALU = ALU(Q_V, Q_A, Q_Cy, D7, D6, D_Ov, D_Z, D_Cy);

  D_V = mux_4x1(0, Constante8, memDados[Q_R], QALU, R1, R0);

  D_A = Q_V;

  memDados[Q_R] = !WR ? Q_V : memDados[Q_R];

}


void Instrucoes() {

  memCodigo[0X00] = 0xFE; //MOV V, CONST8
  memCodigo[0X01] = 0x108; //MOV R, CONST6
  memCodigo[0X02] = 0x303; // JNC REL5
  memCodigo[0X02] = 0x2B1; // JZ REL6
  memCodigo[0X02] = 0x2DA; // Jov REL6
  memCodigo[0X02] = 0x3E9; // JMP REL6
  memCodigo[0X02] = 0x140; //MOV A, V
  memCodigo[0X03] = 0x180; //MOV V, @R
  memCodigo[0X04] = 0x7D; //MOV V, CONST8
  memCodigo[0X05] = 0x1C0; //MOV @R, V
  memCodigo[0X06] = 0x240; // ADD V, A
  memCodigo[0X06] = 0x280; // SUBB V, A
  memCodigo[0X07] = 0x140; //MOV A, V
  memCodigo[0X08] = 0x280; //NOR V, A

}

byte mux_2x1(byte A, byte B, bool S) {

  switch (S) {
    case 0:
      return A;
    case 1:
      return B;
  }
}


byte mux_4x1(byte A, byte B, byte C, byte D, bool S1, bool S0) {

  switch (S1 << 1 | S0) {
    case B11:
      return A;
    case B10:
      return B;
    case B01:
      return C;
    case B00:
      return D;
  }
```

```
}

void EPROM() {

  int i;

  moduloControlo[0x04] = 0x840;
  moduloControlo[0x05] = 0x680;
  moduloControlo[0x06] = 0x320;
  moduloControlo[0x07] = 0x400;
  moduloControlo[0x08] = 0x720;
  moduloControlo[0x09] = 0xF20;
  moduloControlo[0x10] = 0xD20;
  moduloControlo[0x11] = 0x604;
  moduloControlo[0x12] = 0x602;
  moduloControlo[0x13] = 0x601;
  moduloControlo[0x14] = 0x608;

  for (i = 0x00; i <= 0x03; i++) {
    moduloControlo[i] = 0x730;
  }
}
```

Cenário de Testes

Depois do código implementado, passou-se para um cenário de testes, nos prints que se seguem, o valor da instrução aparece em hexadecimal. Visto que surgiram alguns problemas, nos quais quando uma das instruções era apresentada na consola, os registos não apresentavam valores, e caso apresentassem era o valor de 0, e também a ordem pela qual as instruções apareciam não correspondiam à ordem que foi fornecida no enunciado. Por isso é apresentado alguns prints de algumas instruções.

```
Instrução: 108
Registo PC: 1

V = 0
R = 0
A = 0
@R = 0
FLAGS
Z = 0
Cy/Bw = 0
Ov = 0
```

Figura 5 - instrução MOV R, CONST6 com o endereço 0x108 (em hexadecimal)

```
Instrução: 140
Registo PC: 2

V = 0
R = 0
A = 0
@R = 0
FLAGS
Z = 0
Cy/Bw = 0
Ov = 0
```

Figura 6 - instrução MOV A,V com o endereço 0x140 (em hexadecimal)

```
Instrução: 180
Registo PC: 3

V = 0
R = 0
A = 0
@R = 0
FLAGS
Z = 0
Cy/Bw = 0
Ov = 0
```

Figura 7 - instrução MOV V, @R com o endereço 0x180 (em hexadecimal)

```
Instrução: FE
Registo PC: 4

V = 0
R = 0
A = 0
@R = 0
FLAGS
Z = 0
Cy/Bw = 0
Ov = 0
```

Figura 8 -instrução MOV V, CONST8 com o endereço 0xFE (em hexadecimal)

```
Instrução: 1C0
Registo PC: 5

V = 0
R = 0
A = 0
@R = 0
FLAGS
Z = 0
Cy/Bw = 0
Ov = 0
```

Figura 9 - instrução MOV @R, V com o endereço 0x1C0 (em hexadecimal)

```
Instrução: 280
Registo PC: 6

V = 0
R = 0
A = 0
@R = 0
FLAGS
Z = 0
Cy/Bw = 0
Ov = 0
```

Figura 10 - instrução SUBB V, A com o endereço 0x280 (em hexadecimal)

```
Instrução: 140
Registo PC: 7

V = 0
R = 0
A = 0
@R = 0
FLAGS
Z = 0
Cy/Bw = 0
Ov = 0
```

Figura 11 - instrução MOV A, V com o endereço 0x140 (em hexadecimal)

```
Instrução: 280
Registo PC: 8

V = 0
R = 0
A = 0
@R = 0
FLAGS
Z = 0
Cy/Bw = 0
Ov = 0
```

Figura 12 - Instrução NOR A,V com o endereço 0x280 (em hexadecimal)

Conclusões

Com a realização deste trabalho laboratorial conseguimos entender melhor o funcionamento dos CPU's, uma vez que implementamos uma versão reduzida do mesmo. Através da resolução do problema proposto atingimos uma maior compreensão acerca dos conceitos de Data Bus assim como o de Address Bus, e de muitos outros, cada um deles essencial para compreender o funcionamento interno de um CPU.

O grupo prosseguiu todos os passos do enunciado, o que possibilitou a implementação do microprocessador em Arduino, mas apesar de o grupo ter conseguido o procedimento de chegar à implementação do Arduino, fazer a codificação, o desenho do modulo funcional e do módulo de controlo, tabela de sinais ativos e uma EPROM, não foi possível realizar todas as instruções.

Resumindo, os objetivos foram quase todos alcançados. Não foi possível concluir completamente o trabalho, pois muitas das instruções que foram implementadas não correspondiam à ordem que foi fornecida da tabela, mas, apesar das dificuldades e dos problemas encontrados, foi possível adquirir conhecimento acerca de toda a nova matéria lecionada.

Bibliografia

Moodle – Folhas de CF (complementares)

Moodle – Folhas de Computação Física