Trabalho Prático 2

Computação Física

Curso de Licenciatura Informática e Multimédia (LEIM)

Ano Letivo 2017/2018

Data: 15/05/2018

Turma: LEIM 23D

Docente:

Eng. Jorge Pais

Eng. Carlos Carvalho

Grupo: 4

Alunos:

Luis Fonseca (A45125)

Gabriel Diaz(A45133)

Philipp Al-Badavi(A45138)

Índice

1.Introdução……...3

2.Registos da Memória de Dados e Código……..4

3.Especificação das Instruções……..5

4.Codificação das Instruções…….……6

5.Módulo Funcional……..7

6.Tabela do Módulo Funcional…….8

7. EPROM 64x12……10

8.Codigo Arduíno…….....12

9.Conclusão…….28

10.Bibliografia……..…28

1. 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 64x12, 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.

2. Especificação dos registos de uso geral e dos barramentos de endereço e dados para as memórias

Com base nas instruções dadas é possível retirar algumas informações base, como por exemplo o número de bits de cada registo interno ou também, do Adress Bus (AB) e do Data Bus (DB) tanto da memória de Dados como também da memória de Código, que nos serão úteis na codificação das instruções e no desenho do módulo Funcional.

R = 6 bits

V = 8 bits

A = 8 bits

C = Flag Carry (1 bit)

Z = Flag Zero (1 bit)

rel5 = relativo a 5 bits

PC = Program counter (6 bits)

Na Memória de Dados:

DB = 8bits e o AB = 6bits

Na Memória de Código:

DB = 10bits e o AB = 6bits

3. Especificação de 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 memoria do registo R. A 5ª instrução é por sua vez semelhante a 4ª instrução, coloca no valor de posição de memoria no qual se encontra o registo R o valor do registo V. As próximas 4 operações são operações aritméticas, no qual os registos que serão utilizados vão ser o registo A e o registo V, por sua vez na operação de divisão o registo A será utilizado para guardar o resto da divisão inteira dos dois números. As 2 instruções seguintes, JC e JZ irão alterar o valor do *Program Counter* (PC) caso exista *Flag* de *Cy* ou *Flag* de *Z* 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
NAND V, A	V = (V.A)\
ADD V, A	V = V + A
SUBB V, A	V = V - A
DIV V, A	V = V / A ; A = V % A
JC rel5	Se(Cy) PC += rel5
JZ rel5	Se(Z) PC += rel5
JMP end6	PC = end6

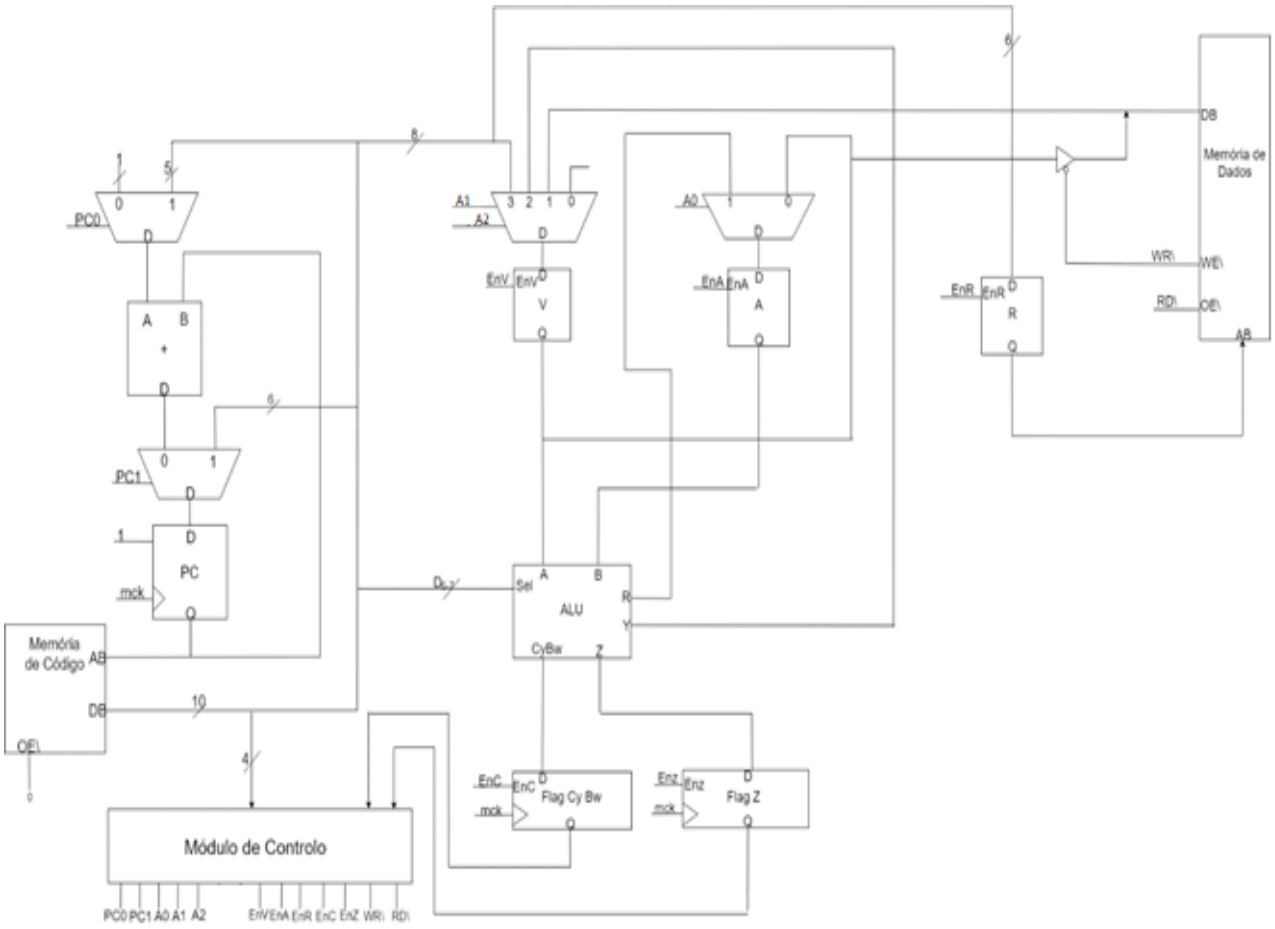
4. Codificação de instruções

Com a tabela da instrução e funcionalidade fornecida, foi feita uma tabela com uma codificação a 10bits, no qual os bits D9, D8, D7 e D6 vão ser usados para distinguir todas as instruções.

Codificação a 10bits											
	Parâmetro	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
MOV V,#const8	Const8	0	1	C7	C6	C5	C4	C3	C2	C1	C0
MOV R,#const6	Const6	1	0	0	0	C5	C4	C3	C2	C1	C0
MOV A,V	-	1	0	0	1	0	0	0	0	0	0
MOV V,@R	-	1	0	1	0	0	0	0	0	0	0
Mov @R,V	-	1	0	0	1	0	0	0	0	0	0
NAND V,A	-	1	1	0	0	0	0	0	0	0	0
ADD V,A	-	1	1	1	1	0	0	0	0	0	0
SUBB V,A	-	1	1	0	0	0	0	0	0	0	0
DIV V,A	-	1	1	0	1	0	0	0	0	0	0
JC rel5	Se(Cy) += rel5	0	0	1	0	0	R4	R3	R2	R1	R0
JZ rel5	Se(Z) += rel5	0	0	0	1	0	R4	R3	R2	R1	R0
JMC end6	PC = end6	0	0	0	0	E5	E4	E3	E2	E1	E0

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), 3 multiplexers 2x1 (que na realidade são 19 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).



6. Módulo de Controlo e tabela com Sinais Ativos

Feito o módulo funcional, passou-se á construção do módulo de controlo, o qual vai ter como entradas, os bits D9, D8, D7, D6 (provenientes do Data Bus da memoria de código), o Cy e o Z (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, A1, A2 (também seletores de multiplexers), os Enables dos registos V, A, R e das *flags* Cy e Z, e o WR\(write) no qual quando ativa (zero lógico) indica que o se está a escrever na memoria de dados e o RD\(read) que quando ativa (zero lógico) indica que se está a ler valores provientes da memoria de dados. Após termos definido as entradas e saídas do modulo de controlo conseguimos representar a ativação dos sinais através da seguinte tabela de verdade.

- **Tabela com os sinais de entrada do modulo de controlo e os sinais ativos.**

Instrução	D9	D8	D7	D6	Cy	Z	Sinais Activos
MOV V,#const8	0	1	-	-	-	-	EnV,A1,A2
MOV R,#const6	1	0	0	0	-	-	EnR
MOV A,V	1	0	0	1	-	-	EnA
MOV V,@R	1	0	1	0	-	-	EnV,RD\,A1
MOV @R,V	1	0	1	1	-	-	WR\
NAND V,A	1	1	0	0	-	-	EnV
ADD V,A	1	1	0	1	-	-	EnV,EnC,A2
SUBB V,A	1	1	1	0	-	-	EnV,EnC,A2
DIV V,A	1	1	1	1	-	-	EnV,EnC,A2
JC rel5	0	0	0	0	0	-	
JC rel5	0	0	0	0	1	-	PC0
JZ rel5	0	0	0	1	-	0	
JZ rel5	0	0	0	1	-	1	PC0
JMC end6	0	0	1	0	-	-	PC1

7. Tabela EPROM 64x12

Com a tabela feita dos sinais ativos das diferentes instruções, prossegui-se para o último passo antes da implementação do CPU no Arduíno, calcular o valor da data e do address. O cálculo do address foi feita á custa da codificação feita nos sinais de entrada do, já a data foi feita através dos sinais de saída, ambos provenientes do modulo de controlo. Foi necessário o cálculo dos mesmo na base hexadecimal.

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 entradas e o valor do address**

	D9	D8	D7	D6	Cy	Z	Address
JC	0	0	0	0	1	-	[2,3]
JZ	0	0	0	1	-	1	[5,7]
JMP	0	0	1	0	-	-	[8,9, a,b]
	0	0	1	1	-	-	[c,d,e,f]
MOV V,#const8	0	1	-	-	-	-	[10,1f]
MOV R,#const6	1	0	0	0	-	-	[20,23]
MOV A,V	1	0	0	1	-	-	[24,27]
MOV V,@R	1	0	1	0	-	-	[28,26]
MOV @R,V	1	0	1	1	-	-	[2c,2f]
NAND V,A	1	1	0	0	-	-	[30,33]
ADD V,A	1	1	0	1	-	-	[34,37]
SUBB V,A	1	1	1	0	-	-	[38,36
DIV V,A	1	1	1	1	-	-	[3c,3f]

- **Tabela com as saídas e o valor da data**

Enc	EnZ	WR\	RD\	EnN	EnR	EnA	PC0	PC1	A0	A1	A2	Data
1	0	1	1	0	0	0	1	0	0	0	0	B10h
0	1	1	1	0	0	0	1	0	0	0	0	710h
0	0	1	1	0	0	0	0	1	0	0	0	308h
0	0	1	1	0	0	0	0	0	0	0	0	300h
0	0	1	1	0	0	0	0	0	0	1	1	383h
0	0	1	1	0	1	0	0	0	0	0	0	340h
0	0	1	1	0	0	1	0	0	0	0	0	320h
0	0	0	1	1	0	0	0	0	0	1	0	182h
0	0	1	0	0	0	0	0	0	0	0	0	200h
0	0	1	1	1	0	0	0	0	0	0	1	381h
1	0	1	1	1	0	0	0	0	0	0	1	B81h
1	0	1	1	1	0	0	0	0	0	0	1	B81h
0	1	1	1	1	0	1	0	0	1	0	1	7A5h

8. Implementação no Arduino

```
boolean imprimir;
int adress;

//SAIDAS MODULO CONTROLO
boolean ENC,ENZ,ENV,ENR,ENA;
boolean WR,RD,SPC0,SPC1,JMP,JZ,JC,SA0,SA1,SA2;
boolean AD0,AD1,AD2,AD3,AD4,AD5; //ADRESS



int memCodigo[256]; //64
int memDados[128] = {0,0,0}; //4096 = 128*16

int rel5, end6;
int const8, const6;
int DBC,ABC;
int DBD,ABD;
int DPC,QPC,DV,QV,DCY,DZ,QCY,QZ,DA,QA,DR,QR;
int DSum, ALUY, ALUR;
boolean Pclock;
unsigned long tempoClk;
const unsigned long TFILTRO = 250;

const int EPROM_MC[64] =
{0x300,0x300,0xB10,0xB10,0x710,0x710,0x308,0x308,0x308
```

```
,0x308,0x0300,0x300,0x300,0x300,0x383,0x383,0x383,0x383
,0x383,0x383,0x383,0x383,
0x383,0x383,0x383,0x383,0x383,0x383,0x383,0x383,0x340,0
x340,0x340,0x340,0x340,0x320,0x320,0x320,0x320,0x182,0x
182,0x182,0x182,0x200,0x200,0x200,0x200,
0x381,0x381,0x381,0x381,0xB81,0xB81,0xB81,0xB81,0xB81
,0xB81,0xB81,0xB81,0x7A5,0x7A5,0x7A5,0x7A5};


void setup(){
  Serial.begin(9600);
  pinMode(2,OUTPUT);
  Pclock = digitalRead(2);
  tempoClk = millis();
  attachInterrupt(0,Clock,RISING);
  imprimir = true;
  reset();
  loadprogram();
  interrupts();
}
void Clock(){

  if(Pclock == HIGH){
    if(millis() - tempoClk >= TFILTRO){
      tempoClk = millis();
      //reset();
      MFSequencial();
```

```
      imprimir = true;
    }
  }
  else{
    Pclock = digitalRead(2);
  }
}

void loop(){
  //MCComb();
  //MFComb();



  if(imprimir){
    imprimirRegistos();
    MCComb();
    MFComb();
    imprimir = false;
   }
 }


void reset(){
  QPC = 0;
```

```
  QV = 0;
  QR = 0;
  QA = 0;
  QCY = 0;
  QZ = 0;

 }

void loadprogram(){
  memCodigo[0] = 0x1FF;//MOV V, #255
  memCodigo[1] = 0x205;//MOV R, #5
  memCodigo[2] = 0x2C0; //MOV @R, V
  memCodigo[3] = 0x100; //MOV V, #0
  memCodigo[4] = 0x280; //MOV V, @R = 255

  memCodigo[5] = 0x105; //MOV V, #5

  memCodigo[6] = 0x240;//MOV A, V

  memCodigo[7] = 0x100; //MOV V, #0
  memCodigo[8] = 0x3C0; //DIV 0, 5
  memCodigo[9] = 0x07F; //JZ 31
  memCodigo[40] = 0x1FF; //MOV V, #255
  memCodigo[41] = 0x240; // MOV A,V
  memCodigo[42] = 0x340; //ADD, 255 + 255
```

```
  memCodigo[43] = 0x005; //JC 5
  memCodigo[51] = 0x380; //SUB, 510 - 255
  memCodigo[52] = 0x105; //MOV V, #5
  memCodigo[53] = 0x240; // MOV A,V
  memCodigo[54] = 0x1FF; //MOV V, #255
  memCodigo[55] = 0x3C0; //DIV 255, 5 (para trocar flag de
Z)

  //memCodigo[56] = 0x300; // NAND
  /*
  memCodigo[55] = 0x101;//MOV V, #1
  memCodigo[56] = 0x240;//MOV A, V
  memCodigo[57] = 0x201;//MOV R, #1
  */



  memCodigo[60] = 0x80; //JMP end 0
}

 void MCComb(){

  //ADRESSES
  AD0 = QZ;
  AD1 = QCY;
```

```
  AD2 = (memCodigo[QPC] & 0x040) >> 5;  //6 bit do indice 0x040h(64)
  AD3 = (memCodigo[QPC] & 0x080) >> 6;  //7 bit do indice 0x080h(128)
  AD4 = (memCodigo[QPC] & 0x100) >> 7;  //8 bit do indice 0x100h(256)
  AD5 = (memCodigo[QPC] & 0x200) >> 8;  //9 bit do incide 0x200h(512)
  adress = AD0;
  adress = (AD1 << 1) | adress;
  adress = (AD2 << 2) | adress;
  adress = (AD3 << 3) | adress;
  adress = (AD4 << 4) | adress;
  adress = (AD5 << 5) | adress;



  //DATA
  ENC = (EPROM_MC[adress] & 0x800) >> 11;
  ENZ = (EPROM_MC[adress] & 0x400) >> 10;
  WR = (EPROM_MC[adress] & 0x200) >> 9;
  RD = (EPROM_MC[adress] & 0x100) >> 8;
  ENV = (EPROM_MC[adress] & 0x080) >> 7;
  ENR = (EPROM_MC[adress] & 0x040) >> 6;
```

```
  ENA = (EPROM_MC[adress] & 0x020) >> 5;
  SPC0 = (EPROM_MC[adress] & 0x010) >> 4;
  SPC1 = (EPROM_MC[adress] & 0x008) >> 3;
  SA0 = (EPROM_MC[adress] & 0x004) >> 2;
  SA1 = (EPROM_MC[adress] & 0x002) >> 1;
  SA2 = (EPROM_MC[adress] & 0x001);

}
void MFComb(){

 if(!WR){
   QV = memDados[QR];
  }
 if(!RD){
  memDados[QR]= QV;
  }

 DBC = memCodigo[QPC];
 ABC = QPC;
 DBD = memDados[QR];
 ABD = QR;
```

```
//constante 8

byte c0 = memCodigo[QPC] & 0x001;
byte c1 = (memCodigo[QPC] & 0x002) >> 1;
byte c2 = (memCodigo[QPC] & 0x004) >> 2;
byte c3 = (memCodigo[QPC] & 0x008) >> 3;
byte c4 = (memCodigo[QPC] & 0x010) >> 4;
byte c5 = (memCodigo[QPC] & 0x020) >> 5;
byte c6 = (memCodigo[QPC] & 0x040) >> 6;
byte c7 = (memCodigo[QPC] & 0x080) >> 7;
const8 = c0;
const8 = (c1 << 1) | const8;
const8 = (c2 << 2) | const8;
const8 = (c3 << 3) | const8;
const8 = (c4 << 4) | const8;
const8 = (c5 << 5) | const8;
const8 = (c6 << 6) | const8;
const8 = (c7 << 7) | const8;



//constante 6
byte C0 = memCodigo[QPC] & 0x001;
byte C1 = (memCodigo[QPC] & 0x002) >> 1;
byte C2 = (memCodigo[QPC] & 0x004) >> 2;
```

```
byte C3 = (memCodigo[QPC] & 0x008) >> 3;
byte C4 = (memCodigo[QPC] & 0x010) >> 4;
byte C5 = (memCodigo[QPC] & 0x020) >> 5;
const6 = C0;
const6 = (C1 << 1) | const6;
const6 = (C2 << 2) | const6;
const6 = (C3 << 3) | const6;
const6 = (C4 << 4) | const6;
const6 = (C5 << 5) | const6;

//REGISTOS

DSum = soma(ABC,Mux2x1(SPC0,1,DBC & 0x01F)); //
JC/JZ rel5
DPC = Mux2x1(SPC1,DSum,(DBC & 0x03F)); //JMP end6

int D6 = (memCodigo[QPC] & 0x040) >> 6;  //Bits controlo
ALU
int D7 = (memCodigo[QPC] & 0x080) >> 7;

ALUY = Mux4x2(D7,D6,nand(QV,QA),(QV+QA),(QV-
QA),(QV/QA));

DV = Mux4x2(SA2,SA1,0,DBD,ALUY,const8);

DR = const6;
```

```
 DA = Mux2x1(SA0,QV,(QV%QA));
 //FLAGS

  if(ALUY == (QV+QA)){
    DCY = CySoma(QV,QA);
  }
  else if (ALUY == (QV-QA)){
    DCY = BwSub(QV,QA);
  }
    else{
    DCY = 0;
  }

  DZ = FZero(QV,QA);




}


void MFSequencial(){
    QPC = Registo(1,QPC,DPC);
    QV = Registo(ENV,QV,DV);
    QR = Registo(ENR,QR,DR);
```

```
    QA = Registo(ENA,QA,DA);
    QCY = Registo(ENC,QCY,DCY);
    QZ = Registo(ENZ,QZ,DZ);

}



void imprimirRegistos(){


    Serial.println("---REGISTOS---");
    Serial.print("QPC=");Serial.print(QPC);Serial.print("
");Serial.print( adress >= 2 & adress <= 3 ? "JC " : adress >= 4
& adress <= 7? "JZ " :
    adress >= 8 & adress <= 11? "JMP " : adress>=16 &
adress<=31 ? "MOV V, #const8 " : adress >=32 &  adress
<=35? "MOV R, #const6 " : adress >= 36 & adress <=39 ?
"MOV A, V "
    : adress >= 40 & adress <= 43? "MOV V, @R "  : adress >=
44 & adress <= 47? "MOV @R, V " : adress >= 48 & adress
<= 51 ? "NAND " : adress >= 52 & adress <= 55 ? "ADD "
    : adress >= 56 & adress <= 59? "SUB " : adress >= 60 &
adress <= 63? "DIV " : adress >= 0 & adress <= 1 ? "Nenhuma
operação executada " : 0);
    if(adress >= 48 & adress <= 63){
      if(QV==-1){
        QV = 0;
        Serial.print(QV);
```

```
      }
      Serial.print(QV);
      Serial.print(" ");
      Serial.println(QA);
    }
    else if(adress >= 2 & adress <= 7){
      Serial.println(DBC & 0x01F);
    }
    else if(adress >= 8 & adress <= 11){
      Serial.println(DBC & 0x03F);
    }
    else if((adress >= 16 & adress <= 31) || (adress >= 40 &
adress <= 43)){
      Serial.println(QV);
    }
    else if(adress >= 32 & adress <= 35){
      Serial.println(QR);
    }
    else if(adress >= 36 & adress <= 39){
      Serial.println(QA);
    }
    else if(adress >= 44 & adress <= 47){
      Serial.println(DBD);
    }
    else if(adress >= 0 & adress <= 1){
```

```
      Serial.println(" ");
    }
    Serial.print("QV=");Serial.println(adress >= 60 && adress
<= 63 && QV == -1 ? QV = 0 : QV);
    Serial.print("QR=");Serial.println(QR);
    Serial.print("QA=");Serial.println(QA);
    Serial.print("ENV ");Serial.print("ENR ");Serial.print("ENA
");Serial.print("PC0 ");Serial.print("PC1 ");Serial.print("SA0
");Serial.print("SA1 ");Serial.print("SA2 ");
    Serial.println("");
    Serial.print(" ");Serial.print(ENV);Serial.print("
");Serial.print(ENR);Serial.print("
");Serial.print(ENA);Serial.print("
");Serial.print(SPC0);Serial.print("   ");
    Serial.print(SPC1);Serial.print("   ");
    Serial.print(SA0);Serial.print("
");Serial.print(SA1);Serial.print("   ");
Serial.print(SA2);Serial.print("  ");
    Serial.println("");
    Serial.print("DBD=");Serial.print(DBD);Serial.print("
|ABD=");Serial.print(ABD);Serial.print("
|DBC=");Serial.print(DBC);
    Serial.print(" |ABC=");Serial.print(ABC);Serial.print("
|JC=");Serial.print(QCY);Serial.print(" |JZ=");Serial.print(QZ);
    Serial.println("");
```

```
    Serial.print("Memoria de dados 5 primeiros
");Serial.print(memDados[0]);Serial.print("
");Serial.print(memDados[1]);Serial.print("
");Serial.print(memDados[2]);Serial.print("
");Serial.print(memDados[3]);Serial.print("
");Serial.print(memDados[4]);Serial.print("
");Serial.println(memDados[5]);


}


int Registo(boolean EN,int Q,int D){
 if(EN){
  return D;
 }
 else{
 return Q;
 }
}
 byte Mux2x1(boolean Sel, byte E0, byte E1){
 if(Sel){
  return E1;
 }
 else{
  return E0;
 }
}
```

```
int Mux4x2(boolean S1, boolean S0, int A, int B, int C, int
D){ //USAR NA ALU
 switch (S1 << 1 | S0){
 case B00:
 return A;
 case B01:
 return B;
 case B10:
 return C;
 case B11:
 return D;
 }
}

int soma(int A, int B){
 return A + B ;
}




int CySoma(int A, int B){
  if (A+B > 255){
    return 1;
  }
```

```
  return 0;
}

int BwSub(int A, int B){
  if(B > A){
    return 1;
  }
  return 0;
}

int nand(int A, int B){
  int s = (~(A & B)) & 0x0FF;
  return s;
}

int FZero(int A, int B){
  int s = (~(A & B)) & 0x0FF;
  if(A == 0 || B == 0 || s == 0){
    return 1;
  }
  return 0;
}
```

9. Conclusão

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 à cerca dos conceitos de Data Bus assim como o de Adress 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. A maioria das instruções implementadas demonstravam um funcionamento correto.

Resumindo, os objetivos foram quase todos alcançados. Não foi possível concluir completamente o trabalho, pois apesar de ter sido realizada a função que implementa o NAND, não foi possível juntar a instrução às funcionalidades do CPU. Apesar disto foi possível adquirir conhecimento acerca de toda a nova matéria lecionada.

10. Bibliografia

Pais, Eng. Jorge, PowerPoint Computação Física, pp.36-50

Carvalho, Eng. Carlos, PowerPoint Computação Física, pp. 200-207