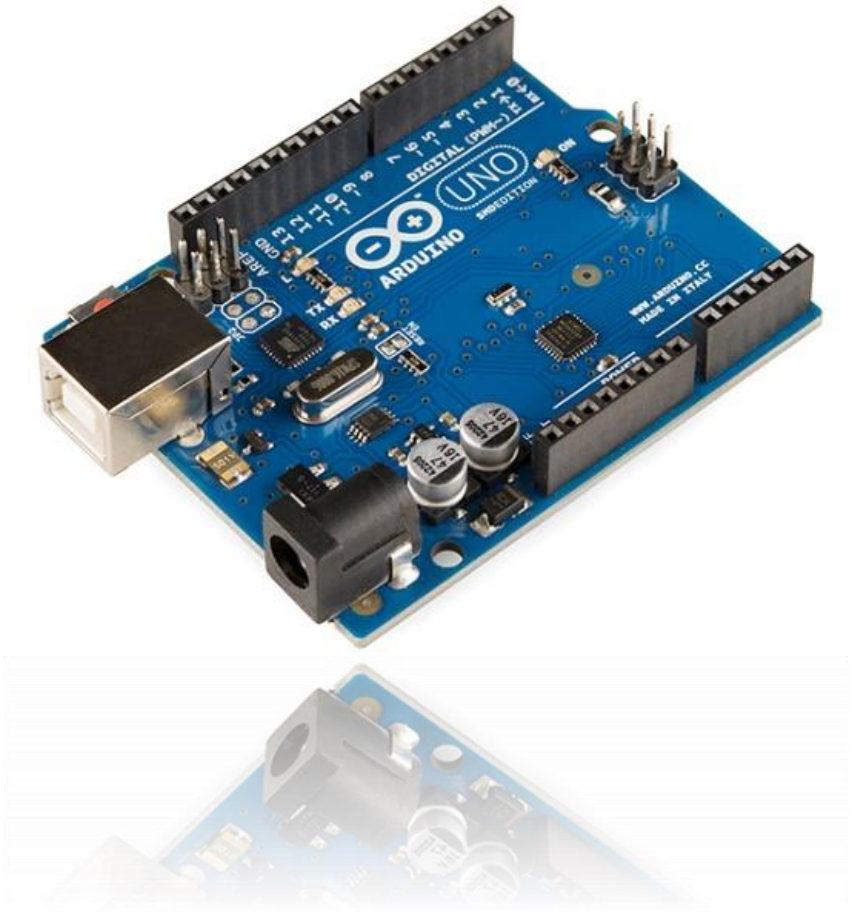
INSTITUTO SUPERIOR de ENGENHARIA de LISBOA

Licenciatura em Engenharia Informática e Multimédia

2º Semestre Lectivo 2014/2015

Computação Física

3º Trabalho Prático



Turma 22D e realizado por:

André Rodrigues – nº39085
André Santos - nº41943
Luís Vital – nº41829

Índice

Introdução 3
1. Módulo Funcional – incluindo bits necessários a todos os componentes do CPU. . 4
2. Entradas e saídas do módulo de controlo. 5
3. Codificação das Instrucções 5
4. Tabela de Programação de EPROM 256x9 6
5. Tabela de Programação de EPROM 64x10 7
6. Material 8
7. Código 9
Conclusão 17
Bibliografia 18

Índice de Figuras

Figura 1 - Módulo funcional 4
Figura 2 - Módulo de Controlo 5
Figura 3 - Esquema da montagem 8

Índice de Tabelas

Tabela 1 - Codificação das Instrucções 5
Tabela 2 - EPROM 256x9 6
Tabela 3 - EPROM 64x10 7

Introdução

Pretende-se desenhar um microprocessador baseado numa arquitectura de *Harvard*, através de código em Arduíno, com um certo conjunto de instrucções.

Para atingir esse objectivo é primeiramente necessário declarar registos internos (R0, R1, A e PC) e *flags Carry* e *Zero* que pertencem ao microprocessador, cuja funcionalidade é guardar dados em memória e os índices dessa memória, para os registos, e mostrar se o valor de A sobrepôs o valor máximo permitido pelo número de *bits* definidos para o microprocessador, no caso da *flag Carry*, ou se o valor de A é zero ou não, no caso da *flag Zero*. Para uma das instrucções, é declarada um número constante de 8 *bits*.

Cada registo necessita de uma certa quantidade de *bits*, de maneira a certificar que a codificação das intrucções nunca seja igual para qualquer alguma. Essa quantidade de *bits* será transferida para as memórias de dados e código que, por sua vez, serão indicadas e utilizadas pelos registos R0 e R1.

Para se certificar que a codificação seja a mais simples possível, é obrigatório que as instruções sejam codificadas com o menor número de *bits* possível.

O método de funcionamento do microprocessador é demonstrado através de um módulo funcional, um gráfico que é composto por:

- *Multiplexers*, filtros que, dependendo do valor *booleano* de um registo de *Enable*, tem como saída os dados maleáveis que recebeu como entrada vinda do resto do sistema, ou um valor constante que recebeu como entrada vinda do código em si;
- Uma *ALU*, um módulo que realiza operações aritméticas, utilizando dois valores de entrada vinda do resto do sistema, e que tem como saída o resultado dessas operações e as novas *flags Carry* e *Zero* resultantes das mesmas operações;
- A memória de código, que armazena o código das instrucções em cada um dos seus registos, que por sua vez é enviado através do sistema para realizar essas mesmas instrucções;
- A memória de dados, que armazena valores inteiros em cada registo, podendo esses ser os mesmos utilizados nas operações aritméticas da *ALU*.
- Dois *Flip-Flops D-Latch*, que recebem os valores booleanos das *flags Carry* e *Zero* e, quando o *Master Clock* se encontrar em *RISING*, enviam os mesmos para o módulo de controlo e a *flag Carry* para a *ALU*;
- O módulo de controlo, que, ao receber certos valores de entrada, envia outros valores na saída que irão controlar os vários módulos do resto do sistema.

Após o desenho do módulo funcional, é realizada a programação de uma *ROM*, neste caso uma *EPROM*, que irá conter as codificações de cada instrucção do microprocessador, e essa codificação irá, por sua vez, determinar os valores de saída do módulo de controlo.

Finalmente, será escrito o código na totalidade no Arduíno, certificando-se que é implementada cada instrucção pelo menos uma vez, e de seguida são realizados os testes ao sistema.

1. Módulo Funcional – incluindo bits necessários a todos os componentes do CPU.

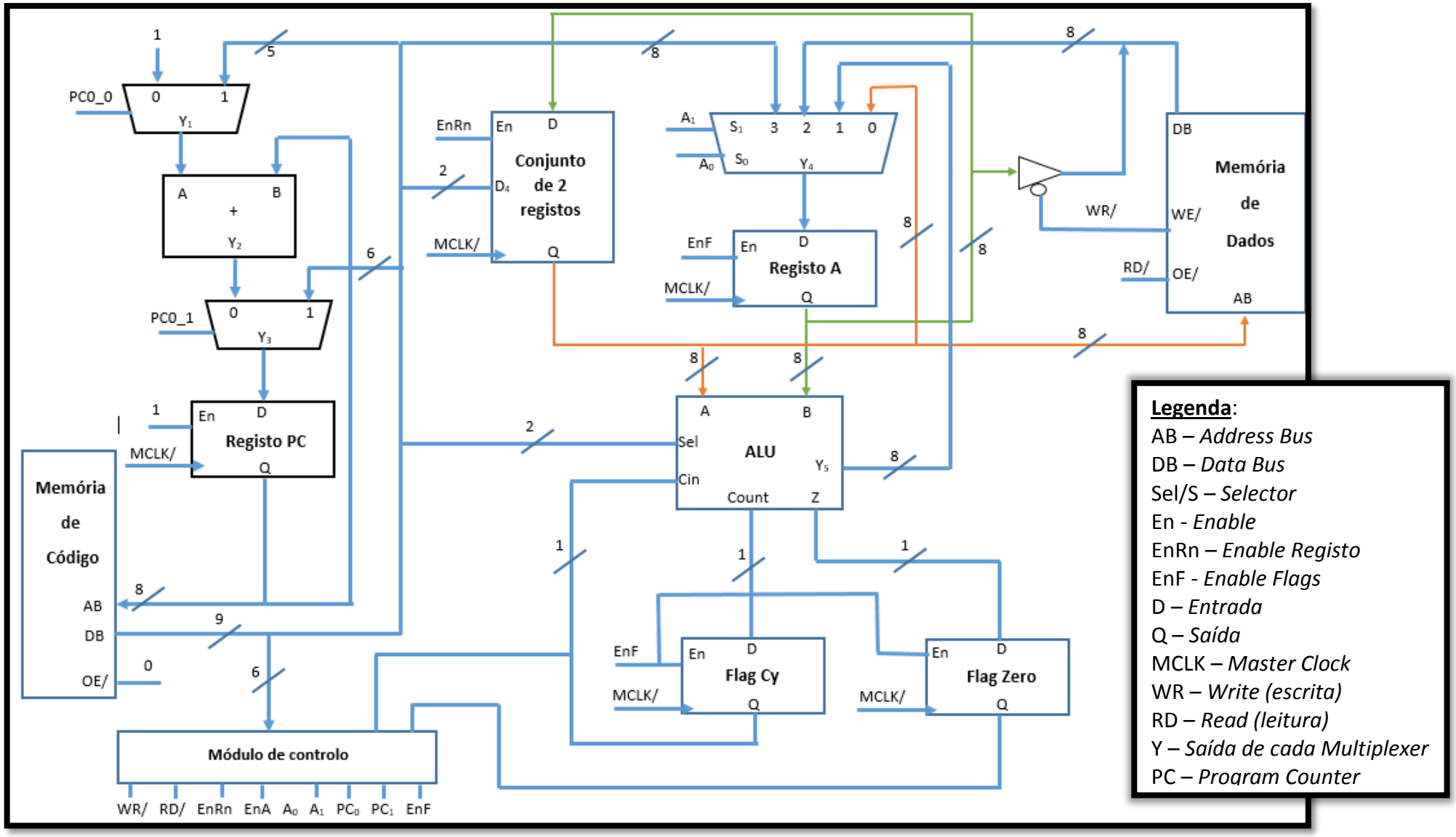


Figura 1 - Módulo funcional

2. Entradas e saídas do módulo de controlo.

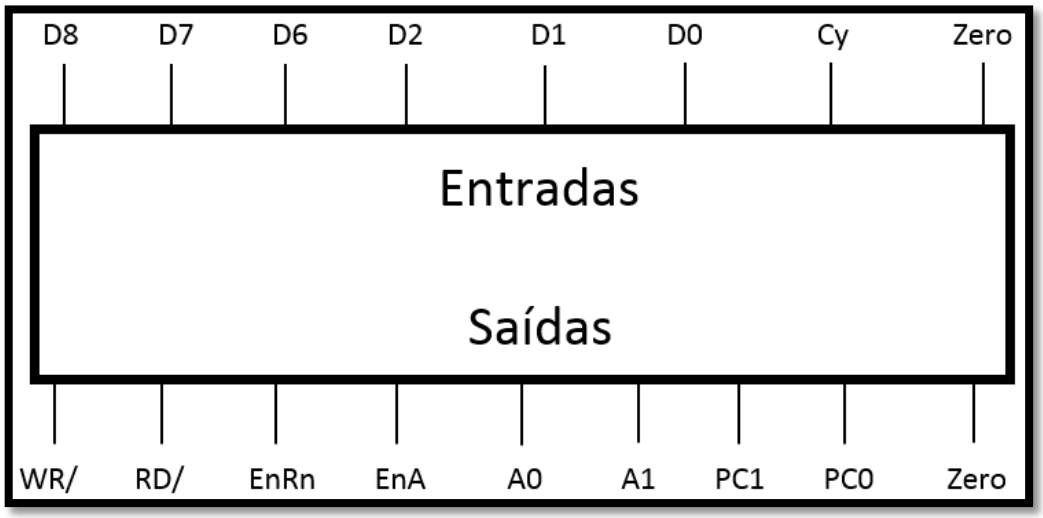


Figura 2 - Módulo de Controlo

3. Codificação das Instrucções

			Codificação								
Instrucção		**Parâmetros**	**D8**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
MOV	**A, #Constante8**	*A*	1	C7	C6	C5	C4	C3	C2	C1	C0
MOV	**A, Rn**	*A*	0	1	1	1	Rn	0	0	1	0
MOV	**Rn, A**	*Rn*	0	1	1	1	Rn	0	0	1	1
NOT	**A**	*A*	0	1	1	1	Rn	0	1	0	1
AND	**A, Rn**	*A*	0	1	1	1	Rn	0	1	1	0
OR	**A, Rn**	*A*	0	1	1	1	Rn	0	1	1	1
ADDC	**A, Rn**	*A*	0	1	1	1	Rn	0	1	0	0
MOV	**A, @Rn**	*A*	0	1	1	1	Rn	0	0	0	0
MOV	**@Rn, A**	*@Rn*	0	1	1	1	Rn	0	0	0	1
JC	**rel5**	*Rel5*	0	0	0	1	r4	r3	r2	r1	r0
JNZ	**rel5**	*Rel5*	0	0	1	1	r4	r3	r2	r1	r0
JMP	**end6**	*End6*	0	1	0	e5	e4	e3	e2	e1	e0

Tabela 1 - Codificação das Instrucções

4. Tabela de Programação de EPROM 256x9

		D8	D7	D6	D2	D1	D0	Cy	Zero		EnF	WR/	RD/	EnA	A1	A0	PC1	PC0	EnRn	
Instruções		**A8**	**A7**	**A6**	**A4**	**A3**	**A2**	**A1**	**A0**	**Address**	**D8**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**	**DATA**
MOV	**A, #constante8**	1	-	-	-	-	-	-	-	[80h, FFh]	0	1	1	1	1	1	0	0	0	[F8h]
JMP		0	1	0	-	-	-	-	-	[40h, 5Fh]	0	1	1	0	0	0	1	0	0	[C4h]
JNZ		0	0	1	-	-	-	-	0	[20h, 22h, 24h, 26h, 28h, 2Ah, 2Ch, 2Eh, 30h, 32h, 34h, 36h, 38h, 3Ah, 3Ch, 3Eh]	0	1	1	0	0	0	0	1	0	[C2h]
JNZ		0	0	1	-	-	-	-	1	[21h, 23h, 25h, 27h, 29h, 2Bh, 2Dh, 2Fh, 31h, 33h, 35h, 37h, 39h, 3Bh, 3Dh, 3Fh]	0	1	1	0	0	0	0	0	0	[C0h]
JC		0	0	0	-	-	-	0	-	[00h, 01h, 04h, 05h, 08h, 09h, 0Ch, 0Dh, 10h, 11h, 14h, 15h, 18h, 19h, 1Ch, 1Dh]	0	1	1	0	0	0	0	0	0	[C0h]
JC		0	0	0	-	-	-	1	-	[02h, 03h, 06h, 07h, 0Ah, 0Bh, 0Eh, 0Fh, 12h, 13h, 16h, 17h, 1Ah, 1Bh, 1Eh, 1Fh]	0	1	1	0	0	0	0	1	0	[C2h]
MOV	**A, Rn**	0	1	1	0	1	0	-	-	[68h, 6Bh]	0	1	1	1	0	0	0	0	0	[E0h]
MOV	**Rn, A**	0	1	1	0	1	1	-	-	[6Ch, 6Fh]	0	1	1	0	0	0	0	0	1	[C1h]
NOT	**A**	0	1	1	1	0	1	-	-	[74h, 77h]	1	1	1	1	0	1	0	0	0	[1E8h]
AND	**A, Rn**	0	1	1	1	1	0	-	-	[78h, 7Bh]	1	1	1	1	0	1	0	0	0	[1E8h]
OR	**A, Rn**	0	1	1	1	1	1	-	-	[7Ch, 7Fh]	1	1	1	1	0	1	0	0	0	[1E8h]
ADDC	**A, Rn**	0	1	1	1	0	0	-	-	[70h, 73h]	1	1	1	1	0	1	0	0	0	[1E8h]
MOV	**A, @Rn**	0	1	1	0	0	0	-	-	[60h, 63h]	0	1	0	1	1	0	0	0	0	[B0h]
MOV	**@Rn, A**	0	1	1	0	0	1	-	-	[64h, 67h]	0	0	1	0	0	0	0	0	0	[40h]

Tabela 2 - EPROM 256x9

5. Tabela de Programação de EPROM 64x10

		D8	D7	D6	D2	D1	D0		EnF	WR/	RD/	EnA	A1	A0	JMP	JNZ	JC	EnRn	
Instruções		**A6**	**A5**	**A4**	**A3**	**A1**	**A0**	**Address**	**D9**	**D8**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**	**DATA**
MOV	**A, #constante8**	1	-	-	-	-	-	[20h, 3Fh]	0	1	1	1	1	1	0	0	0	0	[1F0h]
JMP		0	1	0	-	-	-	[10h, 17h]	0	1	1	0	0	0	1	0	0	0	[188h]
JNZ		0	0	1	-	-	-	[08h, 0Fh]	0	1	1	0	0	0	0	1	0	0	[184h]
JC		0	0	0	-	-	-	[00h, 07h]	0	1	1	0	0	0	0	0	1	0	[182h]
MOV	**A, Rn**	0	1	1	0	1	0	[1Ah]	0	1	1	1	0	0	0	0	0	0	[1C0h]
MOV	**Rn, A**	0	1	1	0	1	1	[1Bh]	0	1	1	0	0	0	0	0	0	1	[181h]
NOT	**A**	0	1	1	1	0	1	[1Dh]	1	1	1	1	0	1	0	0	0	0	[3D0h]
AND	**A, Rn**	0	1	1	1	1	0	[1Eh]	1	1	1	1	0	1	0	0	0	0	[3D0h]
OR	**A, Rn**	0	1	1	1	1	1	[1Fh]	1	1	1	1	0	1	0	0	0	0	[3D0h]
ADDC	**A, Rn**	0	1	1	1	0	0	[1Ch]	1	1	1	1	0	1	0	0	0	0	[3D0h]
MOV	**A, @Rn**	0	1	1	0	0	0	[18h]	0	1	0	1	1	0	0	0	0	0	[160h]
MOV	**@Rn, A**	0	1	1	0	0	1	[19h]	0	0	1	0	0	0	0	0	0	0	[080h]

Tabela 3 - EPROM 64x10

6. Material

- ✓ Arduíno Uno
- ✓ Breadboard
- ✓ Fios de cobre
- ✓ Botão
- ✓ Resistência de 1KΩ

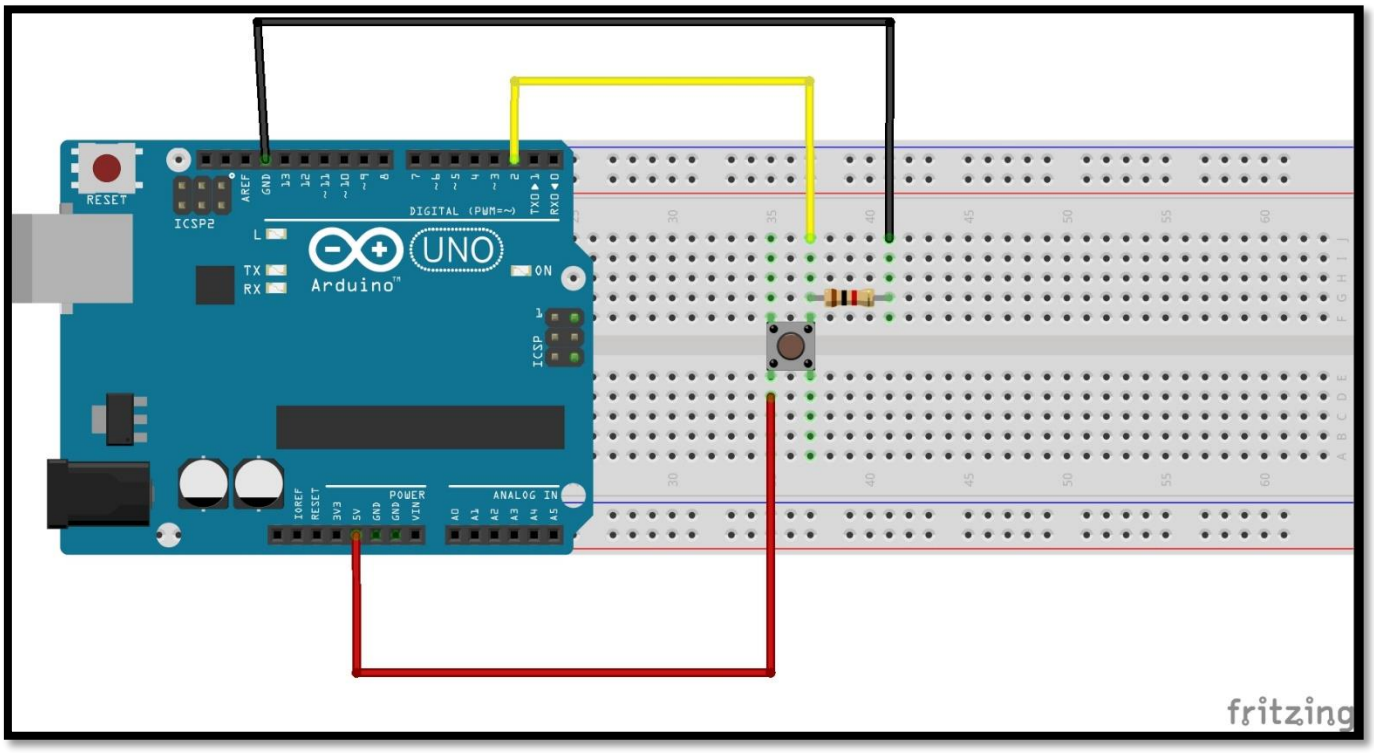


Figura 3 - Esquema da montagem

7. Código

```
#define DEBOUNCETIME 500
unsigned long LastInterrupt;

word MemDados[256];
word MemCodigo[64];
word ModControlo[128];

boolean Zero = 0;
boolean CarryOut = 0;

byte Y1 = 0;
byte Y2 = 0;
byte Y3 = 0;
byte Y4 = 0;
byte Y5 = 0;
byte DB_MemDados = 0;

boolean EnR0, EnR1, EnF, WR, RD, EnA, SA1, SA0, JMP, JC, JNZ, EnRn, QFZ, PC0_1,
PC0_2;

byte DFC, DFZ, DRA;

byte QPC, DPC, QRn, QR1, DRn, QR0, QRA, QFC;

boolean D8 = bitRead(MemCodigo[QPC], 8);
boolean D7 = bitRead(MemCodigo[QPC], 7);
boolean D6 = bitRead(MemCodigo[QPC], 6);
boolean D5 = bitRead(MemCodigo[QPC], 5);
boolean D4 = bitRead(MemCodigo[QPC], 4);
boolean D3 = bitRead(MemCodigo[QPC], 3);
boolean D2 = bitRead(MemCodigo[QPC], 2);
boolean D1 = bitRead(MemCodigo[QPC], 1);
boolean D0 = bitRead(MemCodigo[QPC], 0);

void setup() {
    Serial.begin(9600);
    QPC = 0;
    EPROM();
    Instruccoes();
    attachInterrupt(0, MCLK, RISING);
    interrupts();
}
```

```
void MCLK(){
    if(millis() - LastInterrupt < DEBOUNCETIME ){
        QPC= DPC;
        if(EnRn){
            if(D4){
                QR1= DRn;
            }
            else{
                QR0= DRn;
            }
        }
        if(EnA){
                QRA= DRA;
        }
        if(EnF){
                QFC= DFC;
                QFZ= DFZ;
        }
        PrintRegistos();
        PrintSinais();
     }

    LastInterrupt = millis();
}

byte MUX_2x1(boolean S, byte In0, byte In1){
    if(!S)
        return In0;
    return In1;
}

byte MUX_4x1(boolean S1, boolean S0, byte In0, byte In1, byte In2, byte In3){

    if(!S0 && !S1)
        return In0;

    else if(S0 && !S1)
        return In1;

    else if(!S0 && S1)
        return In2;

    else
        return In3;
}
```

```
byte ALU(byte Select, byte In0, byte In1, byte CarryIn){

    word help;

    if (Select == 0b100){

     //ADDC
        help = In0 + In1 + CarryIn;
        if(help & 0b100000000){
            CarryOut = 1;
        }
        else {
            CarryOut = 0;
        }
        Y5 = (byte)help;
    }

    else if (Select == 0b101){
        //NOT
        Y5 = ~In1;
    }

    else if (Select == 0b110){
        //AND
        Y5 = (In0 & In1);
    }

    else if (Select == 0b111){
        //OR
        Y5 = (In0 | In1);
    }

    if (Y5 == 0) {
        Zero = 1;
    }

    else {
        Zero = 0;
    }

    return Y5;
}
```

```
void ExtSinal(){

    byte I = MemCodigo[QPC];
    byte r = I & 0x1F;
    if (D4)
        r = r | 0xE0; // Nº Negativo
    else
        r = r & 0x1F; // Nº Positivo
}

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

void RegistoRn(byte D4){

    if(EnRn == 1){
        if (D4 == 1) {
            EnR1 = 1;
            EnR0 = 0;
        }

        else {
            EnR1 = 0;
            EnR0 = 1;
        }
    }
    else {
        EnR0 = 0;
        EnR1 = 0;
    }
}

void EPROM(){
    for(byte b=0x20 ; b <=0x3F ; b++){
        ModControlo[b]= 0x1F0;
    }

    for(byte b=0x10 ; b <=0x17 ; b++){
        ModControlo[b]= 0x188;
    }

    for(byte b=0x08 ; b <=0x0F ; b++){
        ModControlo[b]= 0x184;
    }
```

```
    for(byte b=0x00 ; b <=0x07 ; b++){
        ModControlo[b]= 0x182;
    }

    ModControlo[0x1A]= 0x1C0;
    ModControlo[0x1B]= 0x181;
    ModControlo[0x1D]= 0x3D0;
    ModControlo[0x1E]= 0x3D0;
    ModControlo[0x1F]= 0x3D0;
    ModControlo[0x1C]= 0x3D0;
    ModControlo[0x18]= 0x160;
    ModControlo[0x19]= 0x080;
}

void Instruccoes(){
    MemDados[0] = 25;
    MemDados[1] = 5;

    // Armazenar o valor da Memória de Dados no índice 1

    MemCodigo[0] = 0b100000111; // MOV A, const8 => A = const8 = 0
    MemCodigo[1] = 0b011100011; // MOV R0, A => R0 = A = 0
    MemCodigo[2] = 0b011100000; // MOV A, @R0 => A = @R0 = 25
    MemCodigo[3] = 0b011110011; // MOV R1, A => R1 = A = 25

    // Armazenar o valor da Memória de Dados no índice 2

    MemCodigo[4] = 0b100000001; // MOV A, const8 => A = const8 = 1
    MemCodigo[5] = 0b011100011; // MOV R0, A => R0 = A = 1
    MemCodigo[6] = 0b011100000; // MOV A, @R0 => A = @R0 = 5

    // Realizar a soma

    MemCodigo[7] = 0b011110100; // ADDC A, R1 => A = A + R1 = 25 + 5 = 30

   // Mover o resultado da soma para a Memória de Dados

    MemCodigo[8] = 0b011110011; //MOV R1, A => R1 = A = 30
    MemCodigo[9] = 0b100000010; //MOV A, const8 => A = 2
    MemCodigo[10] = 0b011100011; //MOV R0, A => R0 = A = 2
    MemCodigo[11] = 0b011110010; //MOV A, R1 => A = R1 = 30
    MemCodigo[12] = 0b011100001; //MOV @R0, A -> @R0 = A <=> @2 = A = 30
    MemCodigo[13] = 0b001100110; //JNZ 6; QPC = 19
```

PROGRAMA DE TESTE

```
    // Forçar Carry = 1

    MemCodigo[19] = 0b111111111; // MOV A, const8 => A = 255
    MemCodigo[20] = 0b011100011; // MOV R0, A => R0 = A = 255
    MemCodigo[21] = 0b100000001; // MOV A, const8 => A = 1
    MemCodigo[22] = 0b011100100; // ADDC A, R0 => A = A + R0 = 1 + 255 = 256;
Carry = 1

    MemCodigo[23] = 0b000100111; // JC 7; QPC = 30

    // HALT!
    MemCodigo[25] = 0b010011001; // JMP 0 => HALT!

    // Teste a NOT
    MemCodigo[30] = 0b011100101; // NOT A => A = -A = 254

    // Teste a AND
    MemCodigo[31] = 0b011110110; // AND A, R1 => A = A & R1 = 254 & 30 = 30

    MemCodigo[32] = 0b101011000; // MOV A, const8 => A = 88

    // Teste a OR
    MemCodigo[33] = 0b011110111; // OR A, R1 => A = A | R1 = 88 | 30 = 94

    // Teste a Zero
    MemCodigo[34] = 0b100000000; // MOV A, const8 => A = 0
    MemCodigo[35] = 0b011100011; // MOV R0, A => R0 = A = 0
    MemCodigo[36] = 0b011100100; // ADDC A, R0 => A = A + R0 = 0 + 0 = 0
    MemCodigo[37] = 0b001100010; // JNZ 2 => QPC = 39

    // Jump para trás
    MemCodigo[38] = 0b011110111; // ADDC A, R1 => A = A + R1 = 0 + 30 = 30
    MemCodigo[39] = 0b001110010; // JNZ -14 => QPC = 25
}
```

```
void calcVarComb(){

    // leitura de bits da Memoria de Codigo

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

     // Modulo de Controlo
     byte Idx = D8 << 5 | D7 << 4 | D6 << 3 | D2 << 2 | D1 << 1 | D0;
    EnF     = bitRead(ModControlo[Idx], 9);
    WR      = bitRead(ModControlo[Idx], 8);
    RD      = bitRead(ModControlo[Idx], 7);
    EnA     = bitRead(ModControlo[Idx], 6);
    SA1     = bitRead(ModControlo[Idx], 5);
    SA0     = bitRead(ModControlo[Idx], 4);
    JMP     = bitRead(ModControlo[Idx], 3);
    JNZ     = bitRead(ModControlo[Idx], 2);
    JC = bitRead(ModControlo[Idx], 1);
    EnRn = bitRead(ModControlo[Idx], 0);
    DFC= CarryOut;
    DFZ= Zero;
    // se pc0_1 der mal faco = JMP
    PC0_2= JNZ && !QFZ || JC && QFC;
    PC0_1= !JMP;

    RegistoRn(D4);

    DRn= QRA;

    QRn= MUX_2x1(D4, QR0, QR1);

    if(!WR){
        MemDados[QRn] = QRA;
    }
    if(!RD){
        DB_MemDados = MemDados[QRn];
    }

    byte Y5 = ALU((D2 << 2 | D1 << 1 | D0), QRn, QRA, QFC);
```

```
DRA= MUX_4x1(SA1, SA0, QRn, Y5, DB_MemDados, MemCodigo[QPC] & 0xFF);

    byte rel5 = MemCodigo[QPC] & 0x1F;
    if (bitRead(rel5, 4)) rel5 |= 0xE0;

    DPC= MUX_2x1(PC0_1, MemCodigo[QPC] & 0x3F, soma(MUX_2x1(PC0_2, 1,
(rel5) | ((MemCodigo[QPC] & 0x10) << 1 )), QPC));

}

void loop(){
    calcVarComb();
}

void PrintRegistos(){
    Serial.println ("Registo A: " + (String) QRA + "; ");
    Serial.println ("Program Counter: " + (String) QPC + "; ");
    Serial.println ("Registo 0: " + (String) QR0 + "; ");
    Serial.println ("Registo 1: " + (String) QR1 + "; ");
    Serial.println ("Carry: " + (String) CarryOut     + "; ");
    Serial.println ("Zero: " + (String) Zero     + "; ");
    Serial.println ("Constante 1: " + (String)MemDados[0] + "; ");
    Serial.println ("Constante 2: " + (String)MemDados[1] + "; ");
    Serial.println ("Resultado: " + (String)MemDados[2] + "; ");
}

void PrintSinais() {
    Serial.print      ("Address: ");
    Serial.print      (D8 << 5 | D7 << 4 | D6 << 3 | D2 << 2 | D1 << 1 | D0, HEX);
    Serial.println ("; ");
    Serial.print      ("Data: ");
    Serial.print       (ModControlo[D8 << 5 | D7 << 4 | D6 << 3 | D2 << 2 | D1 << 1 |
D0], HEX);
    Serial.println ("; ");
    Serial.println ("Selector 1 do Registo A: " + (String)SA0 + "; ");
    Serial.println ("Selector 2 do Registo A: " + (String)SA0 + "; ");
    Serial.println ("Bit posição 5: " + (String)D4 + "; ");
    Serial.println ("Enable do Registo A: " + (String)EnA + "; ");
    Serial.println ("Enable do Registo Rn: " + (String)EnRn + "; ");
    Serial.println ("Enable da Flags Carry e Zero: " + (String)EnF + "; ");
    Serial.println ("Selector 1 do Program Counter: " + (String)PC0_1 + "; ");
    Serial.println ("Selector 2 do Program Counter: " + (String)PC0_2 + "; ");
    Serial.println ("Condição do Jump: " + (String)JMP + "; ");
    Serial.println ("Condição do Jump if Carry: " + (String)JC + "; ");
    Serial.println ("Condição do Jump if Not Zero: " + (String)JNZ + "; ");
    Serial.println ("-------------------------------------------------");
}
```

Conclusão

O objectivo inicial foi atingido com sucesso, tendo sido verificada a funcionalidade de cada instrucção no microprocessador, sem se detectar quaisquer anomalias no final do período de testes.

Após a realização deste projecto, é possível confirmar que existiu um progresso muito satisfatório na aprendizagem da construção deste tipo de sistemas, sendo agora possível desenhar um microprocessador que realize operações aritméticas e lógicas simples com relativa facilidade.

Durante a realização deste trabalho, foi encontrado um obstáculo que foi progressivamente ultrapassados: a falta de progressão do *Program Counter* durante o período de testes, que foi resolvida através duma breve correcção na *EPROM*. Não existiram quaisquer outros problemas de peso elevado.

Bibliografia

- http://pt.wikipedia.org/wiki/EPROM
- http://www.tutorialspoint.com/assembly_programming/
- http://pt.wikipedia.org/wiki/Assembly
- http://pt.wikipedia.org/wiki/Unidade_lógica_e_aritmética
- http://whatis.techtarget.com/definition/arithmetic-logic-unit-ALU
- http://pt.wikipedia.org/wiki/Conjunção_lógica
- http://study.com/academy/lesson/arithmetic-logic-unit-alu-definition-design-function.html
- https://www.youtube.com/watch?v=iD1msbNFEEk
- http://pt.wikipedia.org/wiki/Negação
- http://pt.wikipedia.org/wiki/Disjunção_lógica
- http://pt.wikipedia.org/wiki/Disjunção_exclusiva
- http://www.electronics-tutorials.ws/combination/comb_2.html
- http://en.wikipedia.org/wiki/Status_register