ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Licenciatura em Engenharia Informática e Multimédia (LEIM)

Computação Física – 1819SV

Trabalho prático nº1

Engº Carlos Carvalho

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

5/04/2019

Índice

Índice Figuras, Tabelas e Mapas

Objetivos

Desenhar circuitos combinatórios baseados em funções lógicas. Desenhar circuitos sequenciais utilizando o grafismo ASM e abordagem de encaminhamento de dados. Simular os circuitos projetados no Arduino.

Introdução

Circuitos Combinatórios vs Circuitos Sequenciais:

Circuitos Combinatórios

- Existem três operações lógicas elementares – NOT, AND e OR.
- A cada operação está atribuído um operador.
- A partir destas três operações elementares é possível definir outras operações (NAND, NOR, XOR, XNOR).

Operação NOT-

Símbolo lógico e tabela de verdade:

A	F
0	1
1	0

Tabela 1- Tabela Verdade NOT

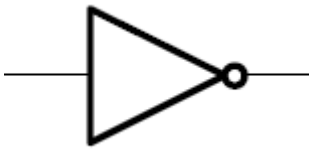


Figura 1 – Símbolo Lógico NOT

Operação AND-

Símbolo lógico e tabela de verdade:

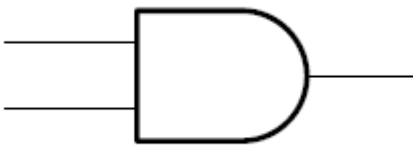


Figura 2 -Símbolo Lógico AND

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

Tabela 2 – Tabela Verdade AND

Operação OR-

Símbolo lógico e tabela de verdade:

Figura 3 – Símbolo Lógico OR

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Tabela 3 – Tabela Verdade OR

Operação XOR-

Símbolo lógico e tabela de verdade:

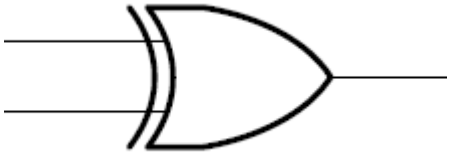


Figura 4 – Símbolo Lógico XOR

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

Tabela 4 – Tabela Verdade XOR

Circuitos Sequenciais

- O valor lógico presente nas saídas depende do valor das entradas e do valor memorizado.
- Dois tipos de células de memória: Latch e Flip-flop.
- 3 tipos de flip-flop "edge-triggered": D, J/K e T.
- São utilizados ASM charts (Algorithmic State Machine) como passo intermediário de um projeto de circuito sequencial.

Flip-flop "J-K edge-triggered" -

Na transição ascendente do CLK, toma em consideração os valores presentes nas entradas J e K. Se o estado presente for 1, apenas K = 1 poderá levá-lo para 0.

Se o estado presente for 0, só J = 1 poderá levá-lo para 1. Quando ambas

as entradas J e K estiverem a 1, inverte o estado da saída.

Tabela de verdade e símbolo lógico:

Q*	Q	J	K
0	0	0	-
0	1	1	-
1	0	-	1
1	1	-	0

Tabela 5 – Tabela Verdade Flip-Flop J – K edge triggerd

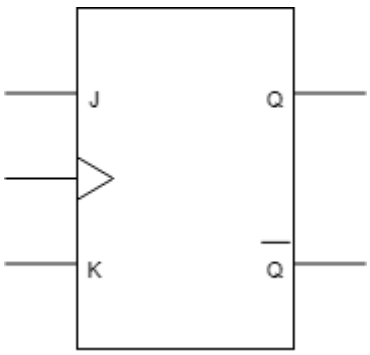


Figura 5-Símbolo Lógico Flip-Flop J – K edge triggered

Flip-flop “T edge-triggered” -

Sempre que T = 1, inverte o estado a cada transição ascendente de CLK.

Com T = 0, permanece no estado anterior, estando insensível ao clock.

Tabela de verdade e símbolo lógico:

Q*	Q	T
0	0	0
0	1	1
1	0	1
1	1	0

Tabela 6- Tabela Verdade Flip-Flop T edge triggerd

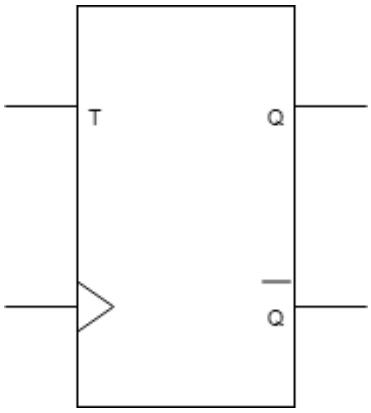


Figura 6 - Símbolo Lógico Flip-Flop J – K edge triggered

ALU

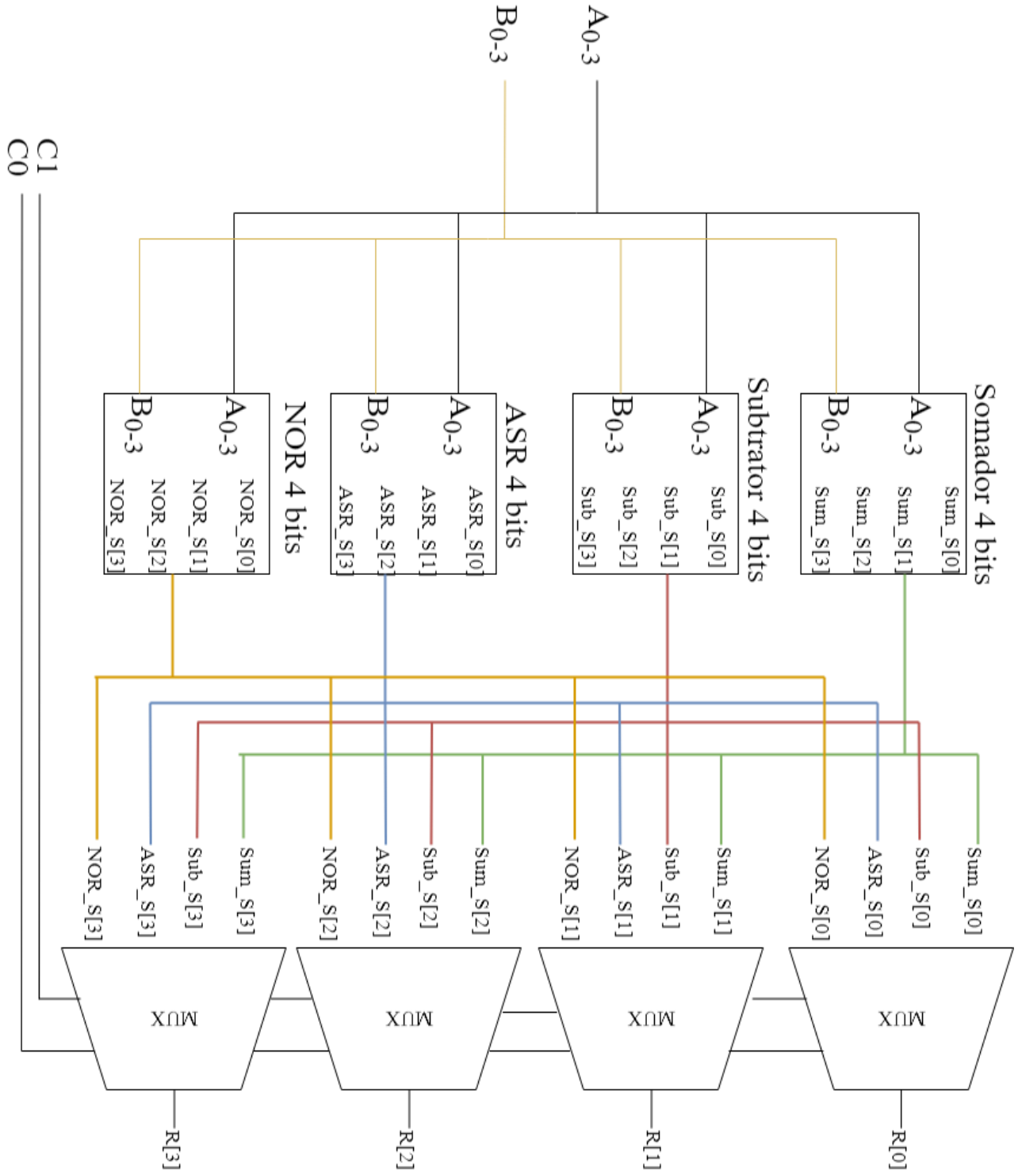


Figura 7-ALU

A – Circuitos Combinatórios

Adição

Operação: R = A + B

Para fazer o somador de dois números a quatro bits, recorreu-se à concatenação de somadores completos a fim de transportar o carry para o peso seguinte, independentemente da ordem e da dimensão do número.

Com as entradas fornecidas, fez-se a tabela de verdade, de seguida um mapa de karnaugh e finalmente obtivemos as expressões para a implementação no Arduino.

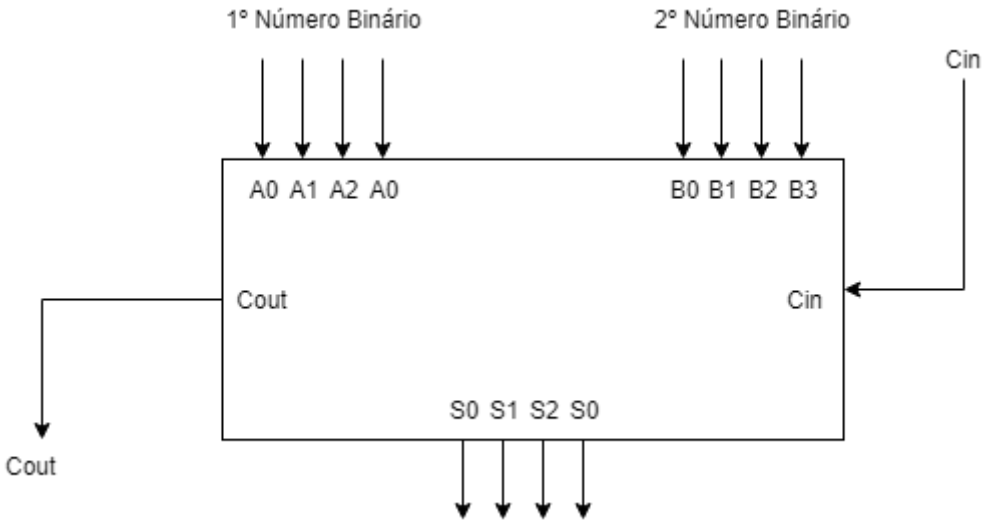


Figura 8- Símbolo lógico do somador de dois números de quatro bits

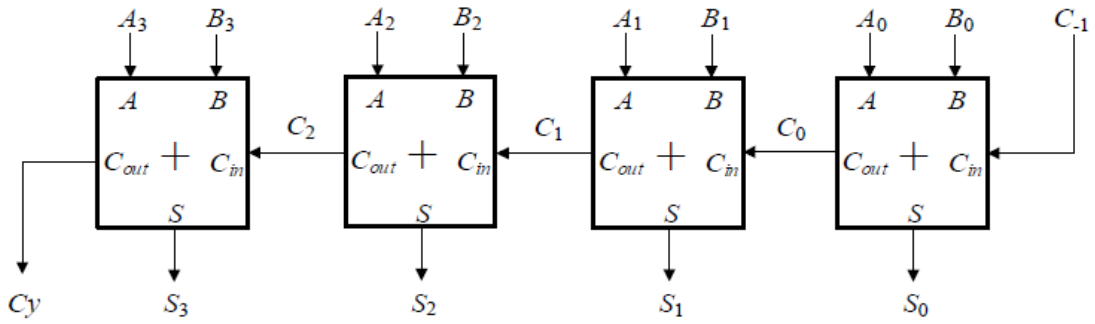


Figura 9-Concatenação de somadores completos

$$Sum = (A \oplus B) \oplus C_{in}$$

Cin \ A,B	00	01	11	10
0	0	1	0	1
1	1	0	1	0

Mapa 1- Mapa Karnaugh Sum

$$C_{out} = A.B + C_{in} . (A \oplus B)$$

Cin \ A,B	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Mapa 2- Mapa Karnaugh Cout

C_{in}	A_n	B_n	C_{out}	S_n
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Tabela 7-Tabela verdade Somador 2 algarismos 1 Bit

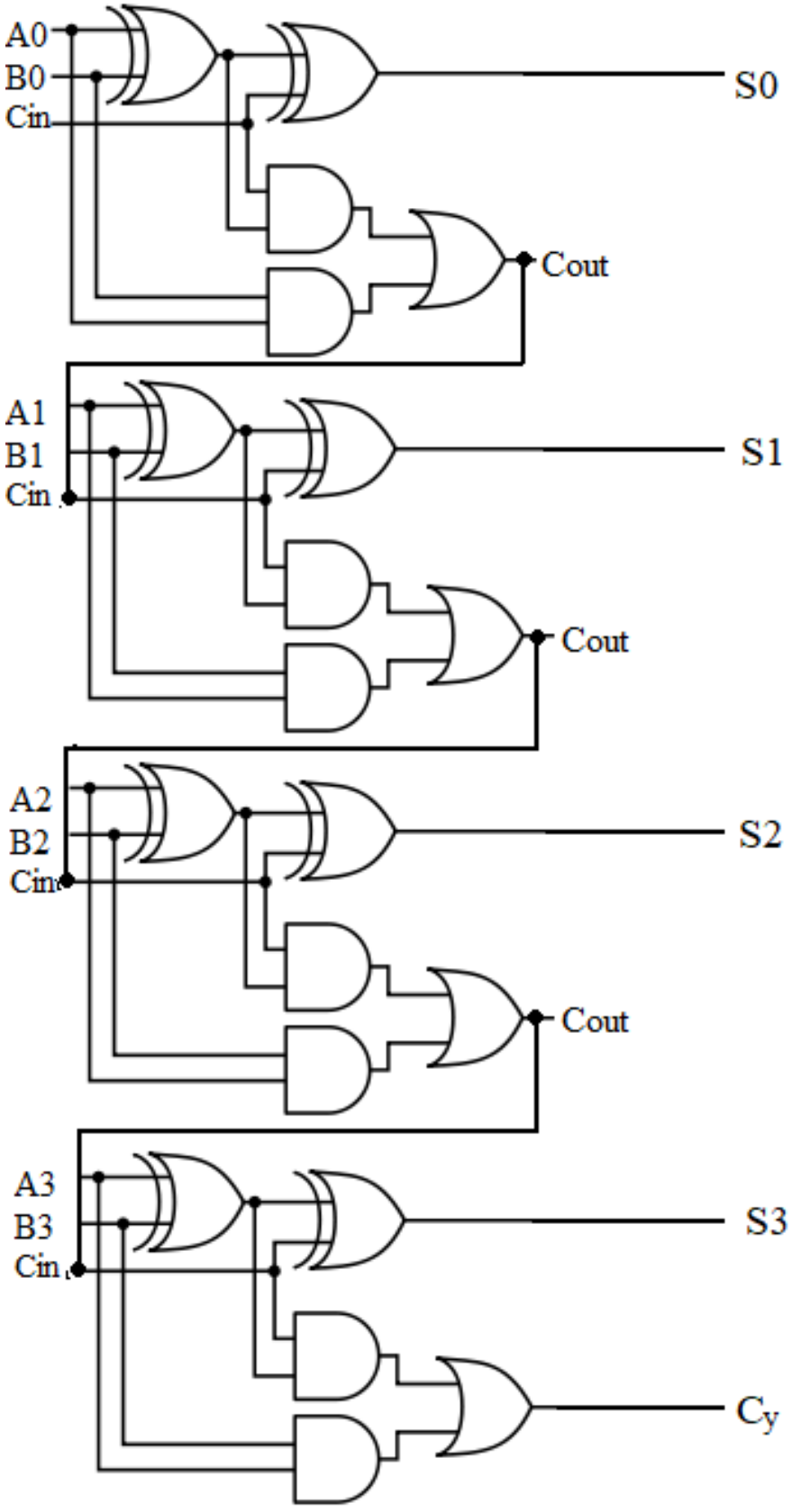


Figura 10- Módulo Somador 2 algarismos 4 bits

Código

```
bool somador_1_bit_S(bool A, bool B, bool CyIN) {
  bool S;
  Cy = CyIN;

  S = A ^ B ^ Cy;

  return S;
}

bool somador_1_bit_C(bool A, bool B, bool CyIN) {
  bool Carry;
  Cy = CyIN;

  Carry = A & B |  A & Cy  | B & Cy;

  return Carry;
}


void moduloSomador(bool A[], bool B[]) {


  Carry_Soma[3] = somador_1_bit_C(A[3], B[3], 0);
  Carry_Soma[2] = somador_1_bit_C(A[2], B[2], Carry_Soma[3]);
  Carry_Soma[1] = somador_1_bit_C(A[1], B[1], Carry_Soma[2]);
  Carry_Soma[0] = somador_1_bit_C(A[0], B[0], Carry_Soma[1]);

  Cy = Carry_Soma[0];

  Sum_S[3]=somador_1_bit_S(A, B,0);
  Sum_S[2]=somador_1_bit_S(A, B,Carry_Soma[3]);
  Sum_S[1]=somador_1_bit_S(A, B,Carry_Soma[2]);
  Sum_S[0]=somador_1_bit_S(A, B,Carry_Soma[1]);

}
```

Subtração

Operação: R = A + B

Para fazer o subtrator de dois números a quatro bits, recorreu-se à soma com o simétrico do subtrativo, negando-se os bits de B e colocando o Carry in (C-1) a 1.

Desta forma, obtém-se a subtração, recorrendo ao mesmo hardware da soma.

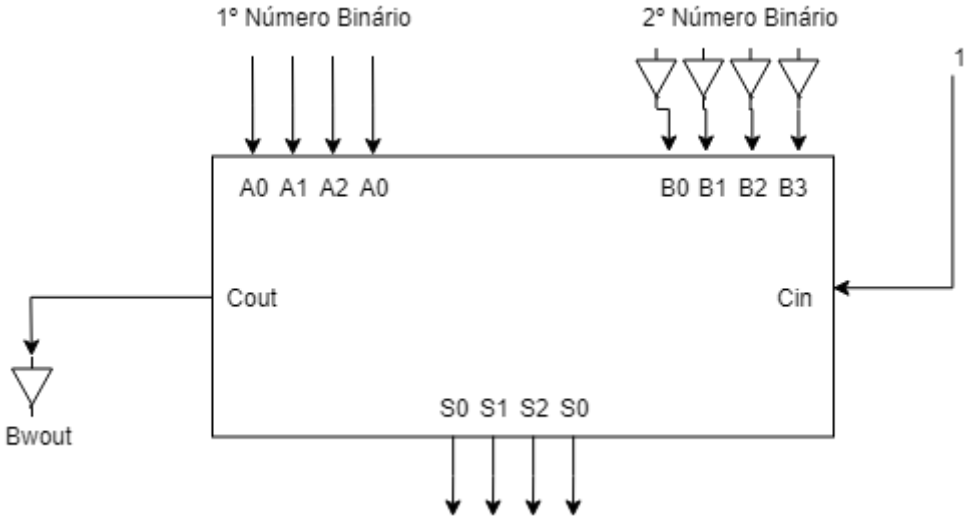


Figura 11 - Símbolo lógico do subtrator de dois números de quatro bits

Código

```
void moduloSubtrator(bool A[], bool B[]) {


  Carry_Sub[3] = somador_1_bit_C(A[3], !B[3], 1);
  Carry_Sub[2] = somador_1_bit_C(A[2], !B[2], Carry_Sub[3]);
  Carry_Sub[1] = somador_1_bit_C(A[1], !B[1], Carry_Sub[2]);
  Carry_Sub[0] = somador_1_bit_C(A[0], !B[0], Carry_Sub[1]);

  Cy = !Carry_Sub[0];

  Sub_S[3]=somador_1_bit_S(A, !B,1);
  Sub_S[2]=somador_1_bit_S(A, !B,Carry_Sub[3]);
  Sub_S[1]=somador_1_bit_S(A, !B,Carry_Sub[2]);
  Sub_S[0]=somador_1_bit_S(A, !B,Carry_Sub[1]);

}
```

NOR

Operação: R = $\overline{A + B}$

Operação sobre 4 variáveis, que só toma o valor 1 quando todas essas variáveis tiverem o valor 0.

Com as entradas fornecidas, fez-se a tabela de verdade, de seguida um mapa de karnaugh e finalmente obtivemos as expressões para a implementação no Arduino.

Expressão Lógica:

S = $\overline{A + B}$

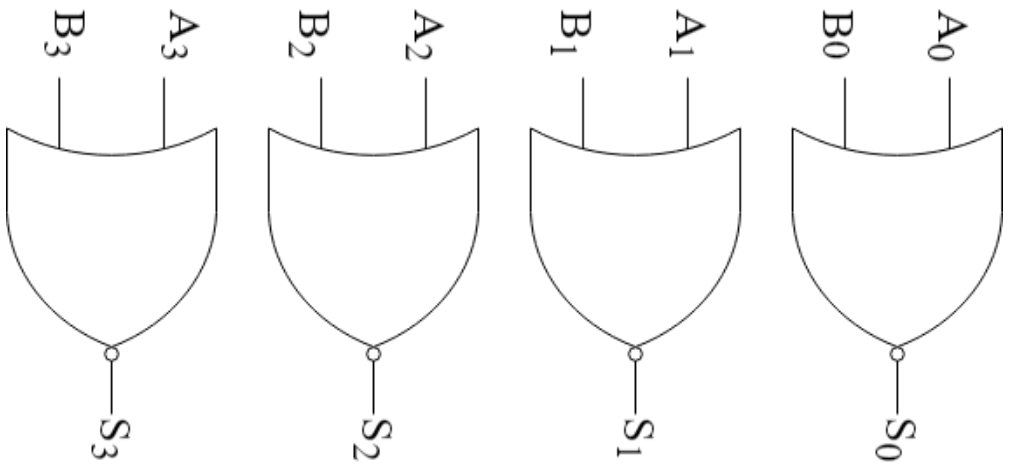


Figura 12- Módulo NOR 2 algarismos 4 bits

	A	
B	1	0
	0	0

Mapa 3- Mapa Karnaugh NOR

A	B	S
0	0	1
0	1	0
1	0	0
1	1	0

Tabela 7- Tabela Verdade NOR

Código

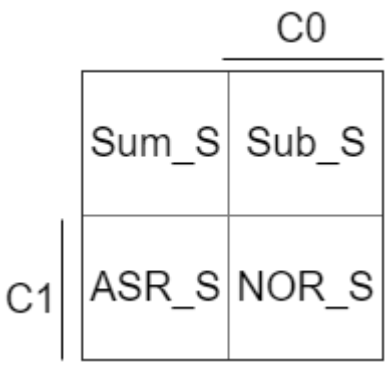
```
bool NOR_1bit(bool A, bool B) {
  bool S;
  S = !(A | B);

  return S;
}

void moduloNOR(bool A[], bool B[]) {

  NOR_S[0]=NOR_1bit(A[0], B[0]);
  NOR_S[1]=NOR_1bit(A[1], B[1]);
  NOR_S[2]=NOR_1bit(A[2], B[2]);
  NOR_S[3]=NOR_1bit(A[3], B[3]);

}
```

MUX

Para se executar apenas uma das quatro operações anteriores, recorre-se a um multiplexer de 4 entradas.

Com as entradas fornecidas, fez-se a tabela de verdade, de seguida um mapa de Karnaugh e finalmente obtivemos as expressões para a implementação no Arduino.

É exemplificado apenas um MUX, mas no Arduino são usados quatro muxs, um para cada bit de cada peso.



Mapa 4- Mapa Karnaugh Mux

C1	C0	R
0	0	Sum_S
0	1	Sub_S
1	0	ASR_S
1	1	NOR_S

Tabela 8 - Tabela Verdade Mux

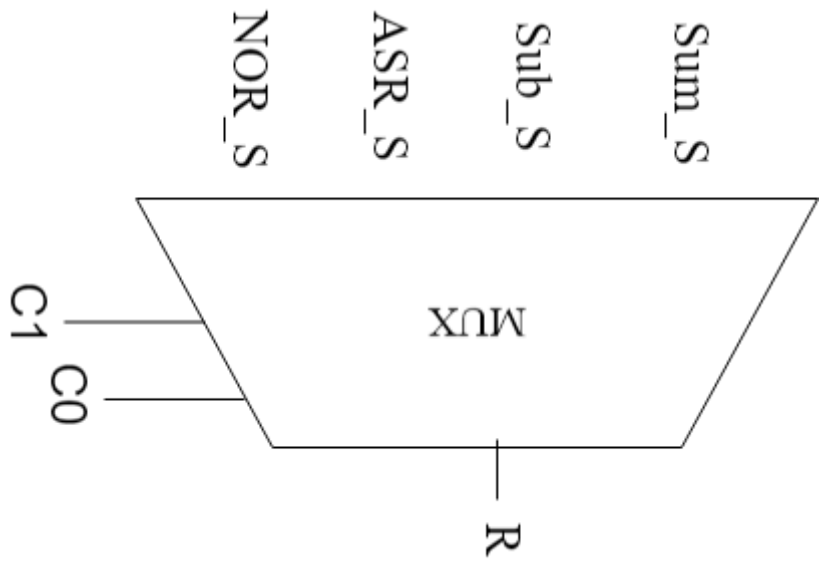


Figura 13- Símbolo lógico do multiplexer usado na ALU para as saídas

Código

```
bool MUX_S3 (bool A, bool B, bool C1, bool C0) {
  return !C1 & !C0 & Sum_S[3]| !C1 & C0 & Sub_S[3] | C1 & C0 &
NOR_S[3];
}

bool MUX_S2 (bool A, bool B, bool C1, bool C0) {
  return !C1 & !C0 & Sum_S[2]| !C1 & C0 & Sub_S[2] | C1 & C0 &
NOR_S[2];
}

bool MUX_S1 (bool A, bool B, bool C1, bool C0) {
  return !C1 & !C0 & Sum_S[1]| !C1 & C0 & Sub_S[1] | C1 & C0 &
NOR_S[1];
}

bool MUX_S0 (bool A, bool B, bool C1, bool C0) {
  return !C1 & !C0 & Sum_S[0]| !C1 & C0 & Sub_S[0] | C1 & C0 &
NOR_S[0];
}
```

B – Circuitos Sequenciais

Os módulos sequenciais distinguem-se dos combinatórios por disporem de uma memória interna, que armazena resultados que irão ser necessários noutras etapas do processo. Não podemos prever diretamente a/s saída/s destes circuitos, visto que as entradas estão passíveis de serem armazenadas em memória e transformadas de acordo os passos que se definirem. Tal como indica o nome, há "sequências" que levam ao/s resultado/s.

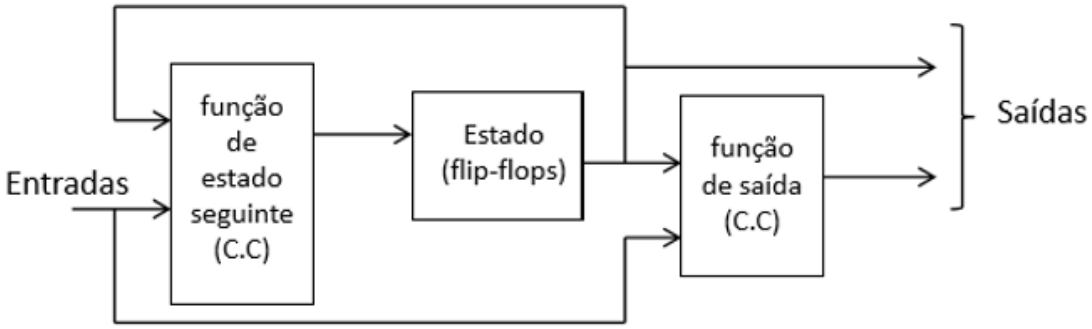


Figura 14 - Modelo composto Moore-Mealey

Contêm *flip-flops*, armazenadores de 1 bit. Um flip-flop tipicamente inclui zero, um ou dois sinais de entrada, um sinal *clock*, por vezes um sinal clear (limpa a saída do valor que lá esteja), e um sinal de saída (ou ainda o complemento do sinal de saída). A mudança no sinal clock faz com que o *flip-flop* mude ou retenha seu sinal de saída, baseado nos valores dos sinais de entrada e na equação característica do *flip-flop*.

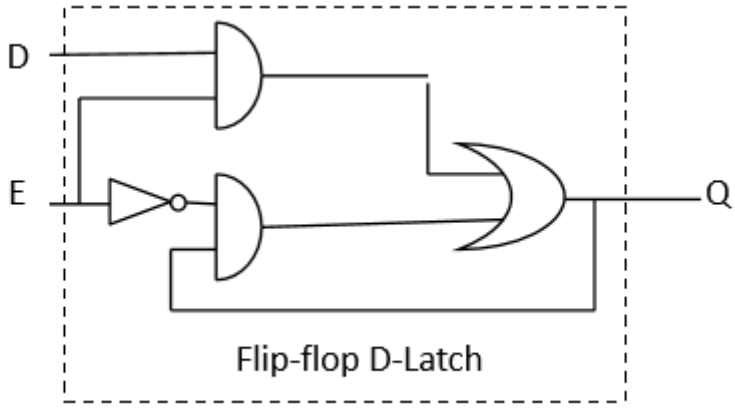


Figura 15- Exemplo de um Flip-Flop

Neste trabalho, o módulo sequencial é o Arithmetic Shift Right (ASR) . Projetámo-lo recorrendo à técnica de encaminhamento de dados e ao grafismo ASM.

Incluímos registos (para armazenar informação nas variáveis usadas), 1 contador que serve para incremento, e 1 Shift Register que é constituído por flip-flops e que servem para fazer shift dos bits (já que baseámos o nosso divisor em subtrações sucessivas, Resto – Valor de B).

O ASR é composto por um Módulo Funcional, que irá executar a operação, e o Módulo de Controlo, que controla o módulo funcional, ligando e desligando determinadas entradas.

Definir as entradas e as saídas

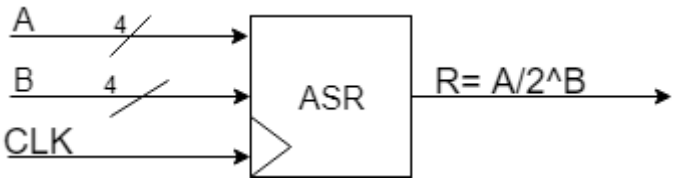


Figura 16 - ASR com as entradas e saída

Algoritmo da divisão

```
for (X = A, Y = B, Y != 0){
        Y--;
        X>>1;
}
```

Tipos de Hardware

X: Registo (servindo para registar o resultado)

Y: Registo (registo de iteração)

A: Entrada (operando)

B: Entrada (operando)

Diagrama de blocos do módulo funcional

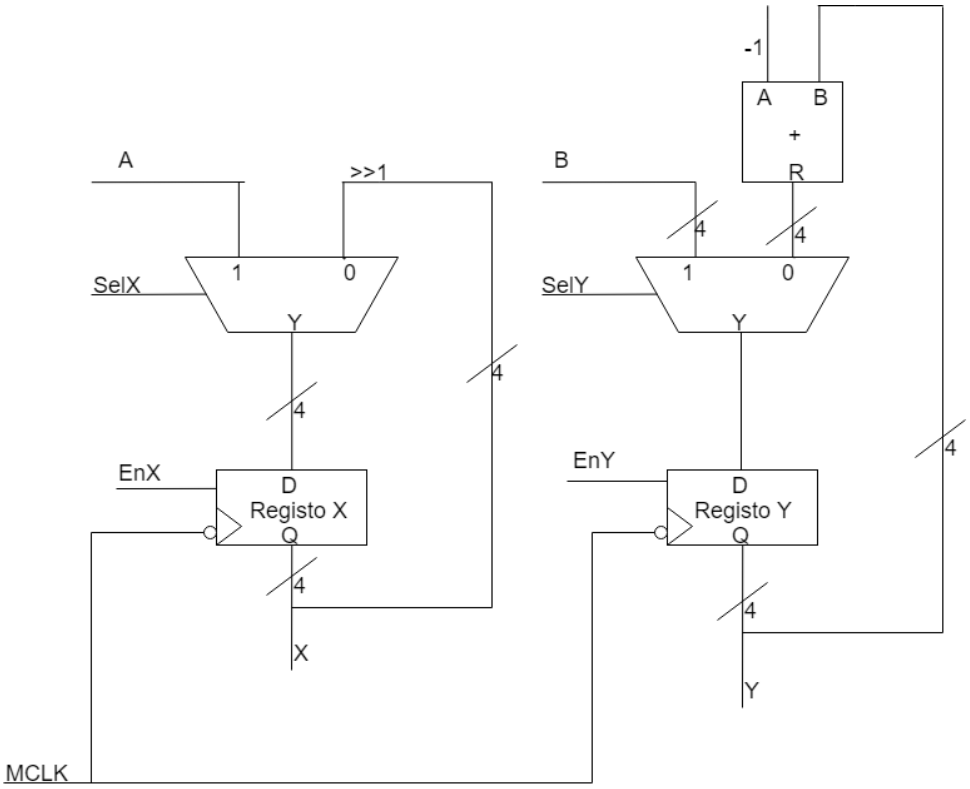


Figura 17 - Módulo funcional do ASR

Especificar as entradas e saídas do módulo de controlo

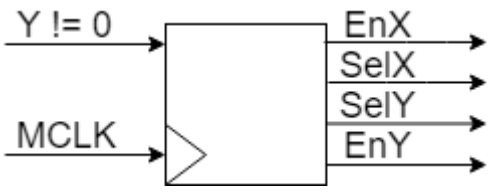


Figura 18 - Entradas e saídas do módulo de controlo

ASM – Módulo de Controlo

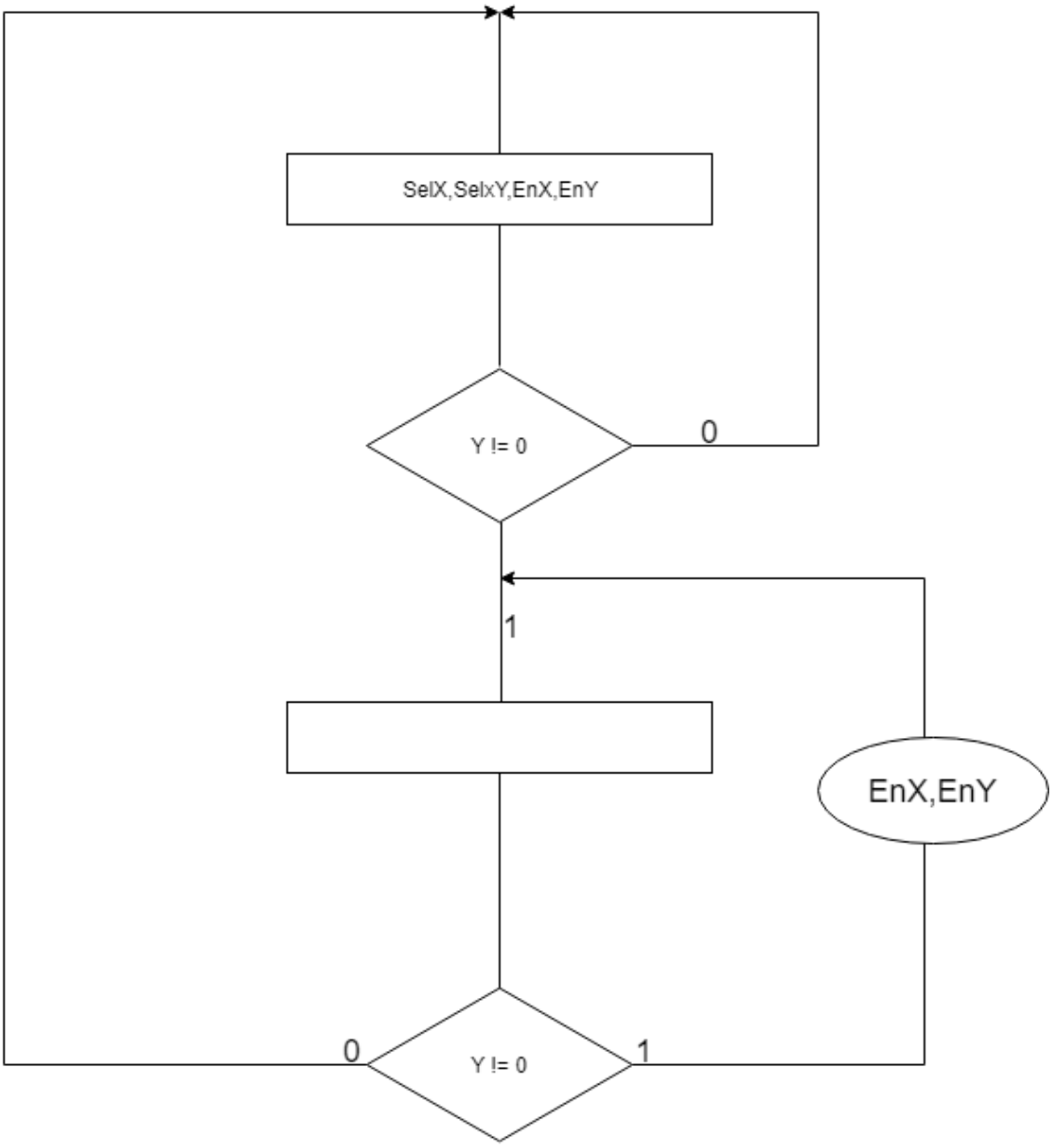


Figura 19 – ASM

Escolher o tipo de célula de Memória

Para o desenvolvimento do circuito sequencial, foi escolhido o flip-flop do tipo D, sendo que as suas respetivas tabelas serão:

Para a transição de estados:

Q*	Q	D
0	0	0
0	1	1
1	0	0
1	1	1

Tabela 9- Transição de estados

Com clock associado:

CLK	D	Q
↑	0	0
↑	1	1
0	-	Q*
1	-	Q*
↓	-	Q*

Tabela 10- Transição de estados (com clock)

Esquema Mealy-Moore

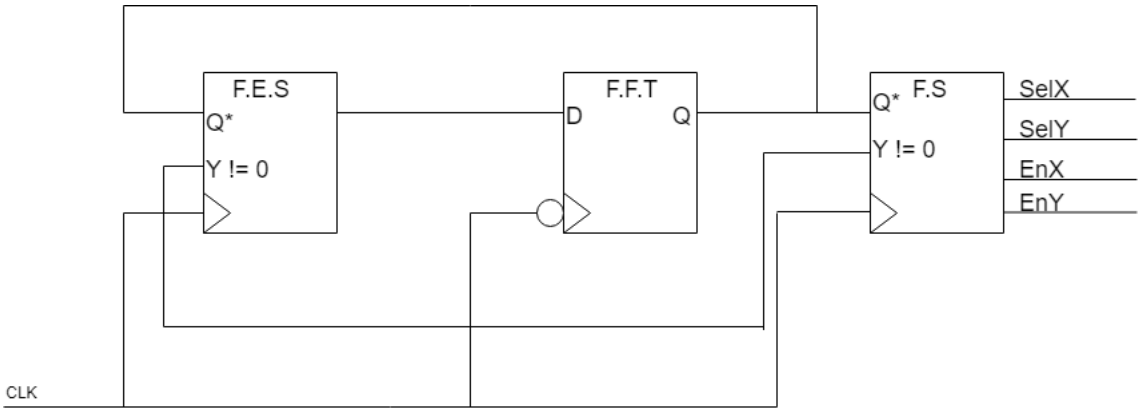
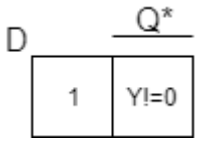


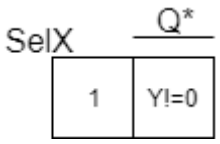
Figura 20 - Esquema mealy-moore com duas entradas na F.E.S

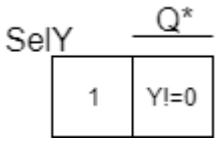
Tabela de Verdade e Mapa de Karnaugh

Y! = 0	Q*	SelX	EnX	SelY	EnY	Q	D
0	0	1	1	1	1	1	1
0	1	0	0	0	0	0	0
1	0	1	1	1	1	1	1
1	1	0	1	0	1	1	1

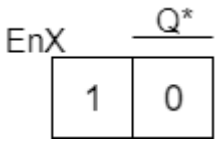
Tabela 11

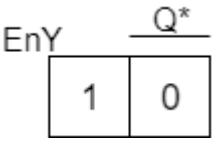






D = Q*/ + (Y! = 0) SelX= Q*/ + (Y != 0) SelY= Q*/ + (Y != 0)





EnX = Q*/ EnY = Q*/

Código Completo

```
bool A[]={0,0,1,0}; //0010 //entrada A que contem 4bits
bool B[]={0,0,1,0}; //0010 //entrada B que contem 4bits
bool Carry_Soma[4];
bool Carry_Sub[4];
bool Sum_S[4];
bool Sub_S[4];
bool NOR_S[4];
bool R[4]; //saida que contem os valores dos 4 bits, mais o Cy e o Borrow
bool Cy;
bool Of;
bool C1=0;
bool C0=1;
int a_0 = 4;
int a_1 = 5;
int a_2 = 6;
int a_3 = 7;
int b_0 = 8;
int b_1 = 9;
int b_2 = 10;
int b_3 = 11;
int S0 = 19;
int S1 = 18;
int S2 = 17;
int S3 = 16;

void setup() {
  // put your setup code here, to run once:
  //FLAGS
  Serial.begin(9600);
  pinMode(3, OUTPUT);
  pinMode(14, OUTPUT);
  pinMode(15, OUTPUT);
  //BITS CONTROLO
  pinMode(12, INPUT);
  pinMode(13, INPUT);
  //SAIDA
  pinMode(S0, OUTPUT);
  pinMode(S1, OUTPUT);
  pinMode(S2, OUTPUT);
  pinMode(S3, OUTPUT);
  //ENTRADA_A
  pinMode(a_0, INPUT);
  pinMode(a_1, INPUT);
  pinMode(a_2, INPUT);
  pinMode(a_3, INPUT);
  //ENTRADA_B
  pinMode(b_0, INPUT);
  pinMode(b_1, INPUT);
  pinMode(b_2, INPUT);
  pinMode(b_3, INPUT);

}

void loop() {
  // put your main code here, to run repeatedly:
  escreverEntradas();
  ALU(A, B, C1, C0);
  escreverSaidas();
```

```
//----------------------ENTRADAS--------------------------------

void escreverEntradas(){
  C1=digitalRead(12);
  C0=digitalRead(13);
  A[0]=digitalRead(a_0);
  A[1]=digitalRead(a_1);
  A[2]=digitalRead(a_2);
  A[3]=digitalRead(a_3);
  B[0]=digitalRead(b_0);
  B[1]=digitalRead(b_1);
  B[2]=digitalRead(b_2);
  B[3]=digitalRead(b_3);
}
//----------------------------MÓDULOS-------------------------
bool somador_1_bit_S(bool A, bool B, bool CyIN) {
  bool S;
  Cy = CyIN;

  S = A ^ B ^ Cy;

  return S;
}

bool somador_1_bit_C(bool A, bool B, bool CyIN) {
  bool Carry;
  Cy = CyIN;

  Carry = A & B |  A & Cy  | B & Cy;

  return Carry;
}


void moduloSomador(bool A[], bool B[]) {


  Carry_Soma[3] = somador_1_bit_C(A[3], B[3], 0);
  Carry_Soma[2] = somador_1_bit_C(A[2], B[2], Carry_Soma[3]);
  Carry_Soma[1] = somador_1_bit_C(A[1], B[1], Carry_Soma[2]);
  Carry_Soma[0] = somador_1_bit_C(A[0], B[0], Carry_Soma[1]);

  Cy = Carry_Soma[0];

  Sum_S[3]=somador_1_bit_S(A, B,0);
  Sum_S[2]=somador_1_bit_S(A, B,Carry_Soma[3]);
  Sum_S[1]=somador_1_bit_S(A, B,Carry_Soma[2]);
  Sum_S[0]=somador_1_bit_S(A, B,Carry_Soma[1]);

}

void moduloSubtrator(bool A[], bool B[]) {


  Carry_Sub[3] = somador_1_bit_C(A[3], !B[3], 1);
  Carry_Sub[2] = somador_1_bit_C(A[2], !B[2], Carry_Sub[3]);
  Carry_Sub[1] = somador_1_bit_C(A[1], !B[1], Carry_Sub[2]);
  Carry_Sub[0] = somador_1_bit_C(A[0], !B[0], Carry_Sub[1]);

  Cy = !Carry_Sub[0];

  Sub_S[3]=somador_1_bit_S(A, !B,1);
  Sub_S[2]=somador_1_bit_S(A, !B,Carry_Sub[3]);
  Sub_S[1]=somador_1_bit_S(A, !B,Carry_Sub[2]);
  Sub_S[0]=somador_1_bit_S(A, !B,Carry_Sub[1]);
}
```

```
bool NOR_1bit(bool A, bool B) {
  bool S;
  S = !(A | B);

  return S;
}

void moduloNOR(bool A[], bool B[]) {

  NOR_S[0]=NOR_1bit(A[0], B[0]);
  NOR_S[1]=NOR_1bit(A[1], B[1]);
  NOR_S[2]=NOR_1bit(A[2], B[2]);
  NOR_S[3]=NOR_1bit(A[3], B[3]);

}

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


void ALU(bool A[], bool B[], bool C1, bool C0) {
  moduloSomador(A, B);
  moduloSubtrator(A, B);
  moduloNOR(A, B);
  R[0]=MUX_S0(A[0], B[0], C1, C0);
  R[1]=MUX_S1(A[1], B[1], C1, C0);
  R[2]=MUX_S2(A[2], B[2], C1, C0);
  R[3]=MUX_S3(A[3], B[3], C1, C0);

}

bool MUX_S3 (bool A, bool B, bool C1, bool C0) {
  return !C1 & !C0 & Sum_S[3]| !C1 & C0 & Sub_S[3] | C1 & C0 & NOR_S[3];
}

bool MUX_S2 (bool A, bool B, bool C1, bool C0) {
  return !C1 & !C0 & Sum_S[2]| !C1 & C0 & Sub_S[2] | C1 & C0 & NOR_S[2];
}

bool MUX_S1 (bool A, bool B, bool C1, bool C0) {
  return !C1 & !C0 & Sum_S[1]| !C1 & C0 & Sub_S[1] | C1 & C0 & NOR_S[1];
}

bool MUX_S0 (bool A, bool B, bool C1, bool C0) {
  return !C1 & !C0 & Sum_S[0]| !C1 & C0 & Sub_S[0] | C1 & C0 & NOR_S[0];
}

//----------------------SAÍDAS---------------------------------
void escreverSaidas() {

  MUX_Saidas(C1,C0);

}
```

```
void MUX_Saidas (bool C1, bool C0) {
  switch (C1 << 1 | C0) {
    case B00:
      mostrarResultado();
      flagZero(R);
      mostrarCarry();
      mostrarOverflow();
      break;
    case B01:
      mostrarResultado();
      flagZero(R);
      mostrarBorrow();
      mostrarOverflow_sub();
      break;
    case B11:
      mostrarResultado();
      flagZero(R);
      break;
  }
}

//----------------------RESULTADOS-E-FLAGS---------------------------------

void mostrarResultado() {

  digitalWrite(S0, R[0]);
  digitalWrite(S1, R[1]);
  digitalWrite(S2, R[2]);
  digitalWrite(S3, R[3]);

}


void mostrarCarry() {
  digitalWrite(15, Carry_Soma[0]);
}

void mostrarBorrow() {
  digitalWrite(15, !Carry_Sub[0]);
}


void mostrarOverflow() {
  Of = XOR(Carry_Soma[1], Carry_Soma[0]);
  digitalWrite(14, Of);
}

void mostrarOverflow_sub() {
  Of = XOR(Carry_Sub[1], Carry_Sub[0]);
  digitalWrite(14, Of);
}

void flagZero(bool S[]) {
  digitalWrite(3, NOR_flag(S));
}


//----------------------FUNÇÕES-AUXILIARES---------------------------------
bool NOR_flag(bool S[]) {

  return !(S[0] | S[1] | S[2] | S[3]);
}

bool XOR(bool A, bool B) {
  return (A & !B) | (B & !A);
}
```

Conclusões

Com este trabalho foi permitido uma profunda abordagem em relação a este tópico, como a elaboração de mapas de Karnaugh, modelos “caixa-preta”, circuitos combinatórios e sequenciais e a sua implementação em Arduino.

Este trabalho foi uma boa ‘rampa de lançamento’ para os trabalhos futuros e nas aulas que fomos tendo entretanto, foi fácil perceber que bastantes aspetos utilizados neste projeto serão aplicados em projetos futuros.

Este trabalho introduziu-nos primeiramente aos números binários e às operações de soma e subtração, depois às portas lógicas elementares e às suas derivadas, e depois aos circuitos sequenciais, incluindo os flip-flops, ASM’s, módulos de controlo e funcionais.

Bibliografia

Moodle – Folhas de CF (complementares)

Moodle – Folhas de Computação Física