INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

COMPUTAÇÃO FÍSICA

PROJECTO 1

André Fonseca 39758
Daniel Santos 32078
Guilherme Rodrigues 41863

Abril 2017

ÍNDICE

INTRODUÇÃO

Para este projecto pretende-se o desenvolvimento de uma ALU (Arithmetic Logic Unit) que realize as operações aritméticas de soma, subtração, multiplicação e divisão e as operações lógicas AND, OR, NOT e XOR em dois números, ambos a 2 bits.

A selecção de cada operação, aritmética ou lógica, é realizada através de uma palavra de controlo C a 3 bits. Cada operação vai ser identificada por um número binário único e a sua atribuição é dada pela seguinte tabela:

C2	C1	C0	Operation
0	0	0	S = A + B
0	0	1	S = A - B
0	1	0	S = A * B
0	1	1	Q = A / B, R = A % B
1	0	0	F = A . B
1	0	1	F = A + B
1	1	0	G = A/, H = B/
1	1	1	F = A ⊕ B

A ALU será desenvolvida com base no sistema de caixa preta[1] de forma a que o seu utilizador apenas tenha conhecimento do estímulo de entrada e da sua resposta de saída. O seu circuito será desenvolvido com base em uma arquitectura modular. Desta forma, é possível isolar cada funcionalidade ou, neste caso, cada operação tornando cada uma independente das restantes.

[1] https://en.wikipedia.org/wiki/Black_box

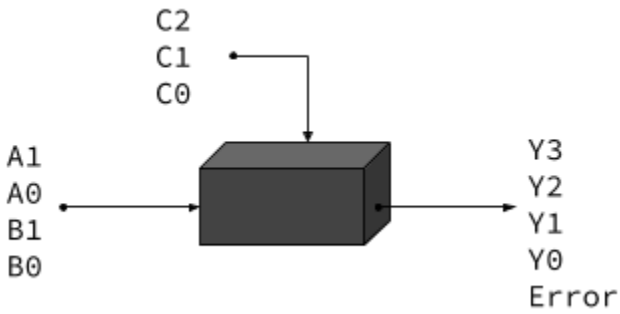
MODELO DE CAIXA PRETA DA ALU

Para incorporar a ALU num sistema de caixa preta é necessário definir qual o estímulo e a resposta ao mesmo.

Os dois números a 2 bits irão corresponder às variáveis de entrada (estímulo): A1, A0, B1, B0. Para que seja possível seleccionar a operação a ser realizada deve-se também considerar uma variável de entrada a 3 bits (bits de seleção): C2, C1, C0.

Em termos de resposta vai ser definida uma variável a 4 bits, visto que esta será a dimensão máxima de todas as operações (dado as variáveis de entrada A e B: Y3, Y2, Y1 e Y0; uma variável de 1 bit para sinalizar algum tipo de erro).

A operação aritmética da divisão apresenta um caso particular em que será necessário assinalar as divisões em que o denominador é igual a 0.



ABORDAGEM MODULAR À ALU

Como existem múltiplas operações estas vão ser divididas e isoladas em módulos, sendo cada módulo independente. Esta abordagem permite modelar de forma isolada cada operação, sendo particularmente vantajosa na resolução e diagnóstico de erros. Os módulos serão posteriormente combinados com as variáveis de selecção de operação, em que apenas será mostrado o resultado da operação seleccionada.

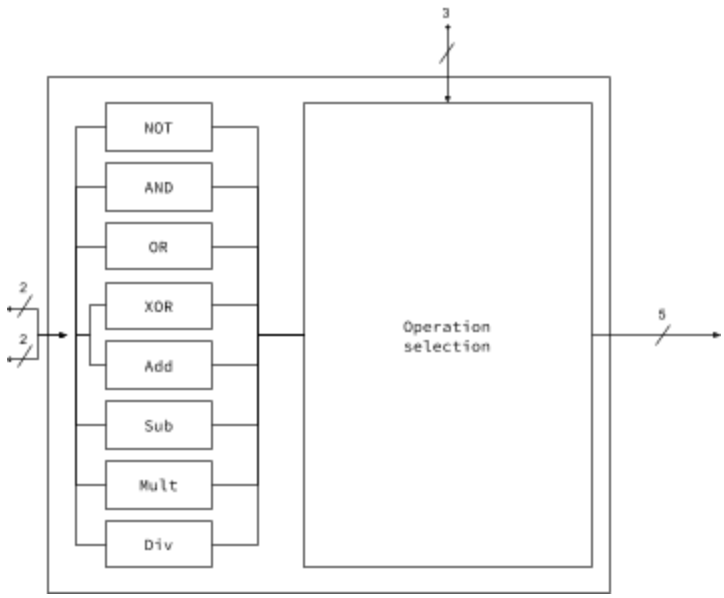


TABELA DE VERDADE E MAPAS DE KARNAUGH

As tabelas de verdade permitem deduzir expressões lógicas pela combinação de valores de entrada e de saída. Para o desenvolvimento da ALU foram calculados os valores de saída de todas as combinações possíveis das variáveis de entrada. Isto é, para o caso A1 = 1, A0 = 0, B1 = 0 e B0 = 1 da operação lógica OR vamos obter o resultado F1 = 0 e F0 = 1.

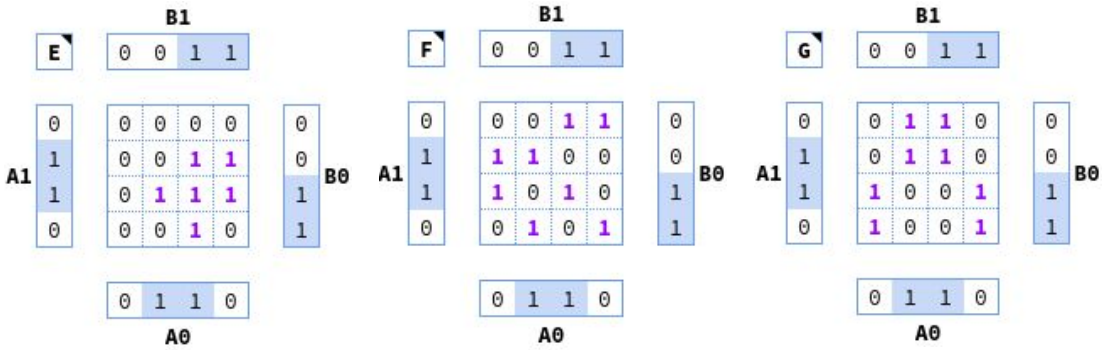
A operação aritmética da divisão apresenta a particularidade em que é necessário assinalar as operações impossíveis, ou seja, quando o denominador da divisão é igual a 0. Para tal, utiliza-se a variável DZ para sinalizar o acontecimento.

				Add.			Sub.			Mult.				Div.					AND		OR		XOR		NOT			
A1	**A0**	**B1**	**B0**	**S2**	**S1**	**S0**	**S2**	**S1**	**S0**	**S3**	**S2**	**S1**	**S0**	**DZ**	**Q1**	**Q0**	**R1**	**R0**	**F1**	**F0**	**F1**	**F0**	**F1**	**F0**	**G1**	**G0**	**H1**	**H0**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	x	x	x	x	0	0	0	0	0	0	1	1	1	1
0	0	0	1	0	0	1	1	1	1	0	0	0	0	**0**	0	0	0	0	0	0	0	1	0	1	1	1	1	0
0	0	1	0	0	1	0	1	1	0	0	0	0	0	**0**	0	0	0	0	0	0	1	0	1	0	1	1	0	1
0	0	1	1	0	1	1	1	0	1	0	0	0	0	**0**	0	0	0	0	0	0	1	1	1	1	1	1	0	0
0	1	0	0	0	0	1	0	0	1	0	0	0	0	**1**	x	x	x	x	0	0	0	1	0	1	1	0	1	1
0	1	0	1	0	1	0	0	0	0	0	0	0	1	**0**	0	1	0	1	0	1	0	1	0	0	1	0	1	0
0	1	1	0	0	1	1	1	1	1	0	0	1	0	**0**	0	0	0	1	0	0	1	1	1	1	1	0	0	1
0	1	1	1	1	0	0	1	1	0	0	0	1	1	**0**	0	0	0	1	0	1	1	1	1	0	1	0	0	0
1	0	0	0	0	1	0	0	1	0	0	0	0	0	**1**	x	x	x	x	0	0	1	0	1	0	0	1	1	1
1	0	0	1	0	1	1	0	0	1	0	0	1	0	**0**	1	0	0	0	0	0	1	1	1	1	0	1	1	0
1	0	1	0	1	0	0	0	0	0	0	1	0	0	**0**	0	1	0	0	1	0	1	0	0	0	0	1	0	1
1	0	1	1	1	0	1	1	1	1	0	1	1	0	**0**	0	0	1	0	1	0	1	1	0	1	0	1	0	0
1	1	0	0	0	1	1	0	1	1	0	0	0	0	**1**	x	x	x	x	0	0	1	1	1	1	0	0	1	1
1	1	0	1	1	0	0	0	1	0	0	0	1	1	**0**	1	1	0	0	0	1	1	1	1	0	0	0	1	0
1	1	1	0	1	0	1	0	0	1	0	1	1	0	**0**	0	1	0	1	1	0	1	1	0	1	0	0	0	1
1	1	1	1	1	1	0	0	0	0	1	0	0	1	**0**	0	1	0	0	1	1	1	1	0	0	0	0	0	0

Com a tabela de verdade completa é então necessário deduzir as expressões lógicas mais simplificadas. Para isso, são utilizados mapas de Karnaugh.

Os espaços em branco nos mapas de Karnaugh representam valores “don’t care” - cujo resultado pode ser igual a 0 ou 1, pois o seu significado é irrelevante. São úteis em situações em que seja favorável agrupar um conjunto maior de valores iguais a 1.

Os mapas de Karnaugh podem ser consultados por completo no seguinte anexo.



Add. S2 Add. S1 Add. S0

EXPRESSÕES LÓGICAS

As expressões lógicas sao determinadas pelo agrupamento de valores 1 que estão juntos entre si.

Adição (Add):

S2 = B1 . A1 + A0 . A1 . B0 + A0 . B1 . B0
S1 = B1 . A0/ . A1/ + B1 . A1/ . B0/ + A1 . B0/ . B1/ + A1 . A0/ . B1/ + A0 . B0 . B1/ . A1/ + B1 . A0 . B0 . A1
S0 = A0 . B0/ + B0 . A0/

Subtração (Sub):

S2 = B1 . A1/ + B0 . B1 . A0/ + B0 . A0/ . A1/
S1 = B1 . A1/ . B0/ + B1 . A0 . A1/ + B0 . B1 . A1 . A0/ + A1 . B1/ . B0/ + A1 . A0 . B1/ + B0 . A1/ . B1/ . A0/
S0 = A0 . B0/ + B0 . A0/

Multiplicação (Mult):

S3 = B1 . A0 . B0 . A1
S2 = B1 . A1 . B0/ + B1 . A1 . A0/
S1 = A0 . B1 . B0/ + A1 . B0 . B1/ + B1 . A0 . A1/ + B0 . A1 . A0/
S0 = A0 . B0

Divisão (Div):

DZ = B1/ . B0/
Q1 = A1 . B1/
Q0 = A0 . A1 + A1 . B1 . B0/ + A0 . B0 . B1/
R1 = B0 . B1 . A1. A0/
R0 = A0 . B0 . A1/ + A0 . B1 . B0/

AND:

F1 = A1 . B1
F0 = A0 . B0

AND com portas NAND:

F1 = (A1 ↑ B1) ↑ 1
F0 = (A0 ↑ B0) ↑ 1

OR:

F1 = B1 + A1
F0 = A0 + B0

OR com portas NAND:

F1 = (B1 ↑ 1) ↑ (A1 ↑ 1)
F0 = (A0 ↑ 1) ↑ (B0 ↑ 1)

XOR:

F1 = A1 . B1/ + B1 . A1/
F0 = A0 . B0/ + B0 . A0/

XOR com portas NAND:

F1 = (A1 ↑ (B1 ↑ 1)) ↑ (B1 ↑ (A1 ↑ 1))
F0 = (A0 ↑ (B0 ↑ 1)) ↑ (B0 ↑ (A0 ↑ 1))

NOT:

G1 = A1/
G0 = A0/
H1 = B1/
H0 = B0/

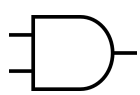
NOT com portas NAND:

G1 = A1 ↑ 1
G0 = A0 ↑ 1
H1 = B1 ↑ 1
H0 = B0 ↑ 1

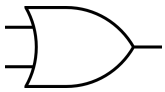
CIRCUITOS COMBINATÓRIOS

Os circuitos combinatórios são representações gráficas de um circuito lógico que mostra as ligações físicas entre as portas lógicas, representadas por um símbolo lógico específico.

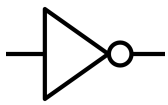
Devido ao elevado número de expressões lógicas obtidas pelos mapas de Karnaugh e à elevada probabilidade de erro foi utilizado um simulador de circuitos lógicos para garantir os resultados apropriados. O simulador permite realizar a representação gráfica do circuito e simular a sua utilização. A simbologia das portas lógicas do simulador é diferente da que é normalmente utilizada:



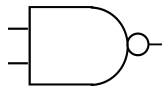
Porta AND



Porta OR



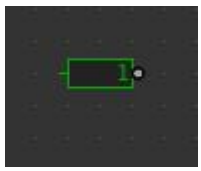
Porta NOT



Porta NAND

Porta AND

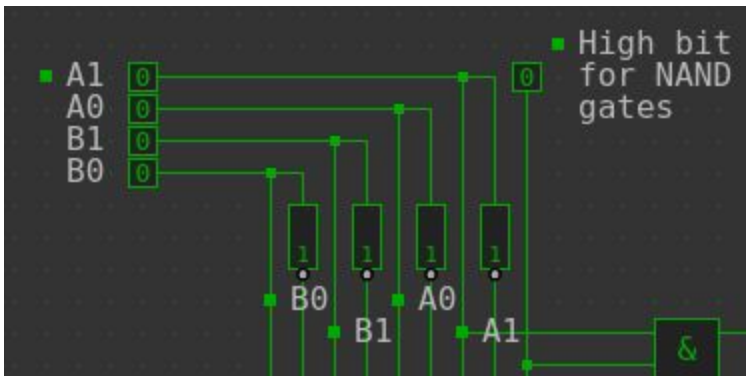
Porta OR



Porta NOT

Porta NAND

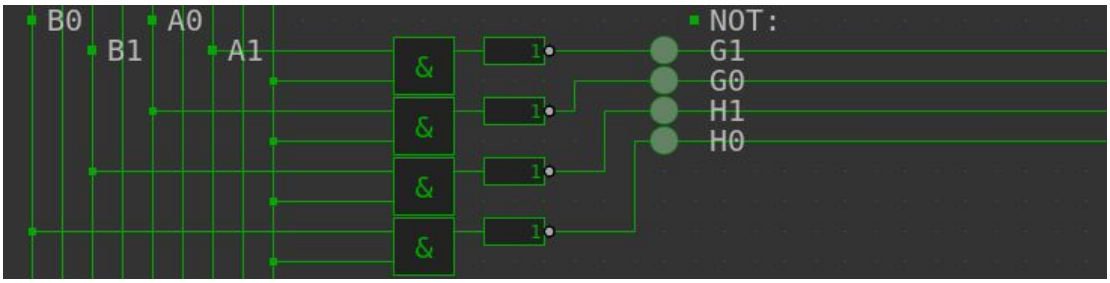
Esta diferença de representação gráfica não afecta de todo a funcionalidade do circuito e/ou as regras das portas lógicas. É unicamente uma maneira diferente de representar as portas lógicas.



Bits de entrada A1, A0, B1, B0

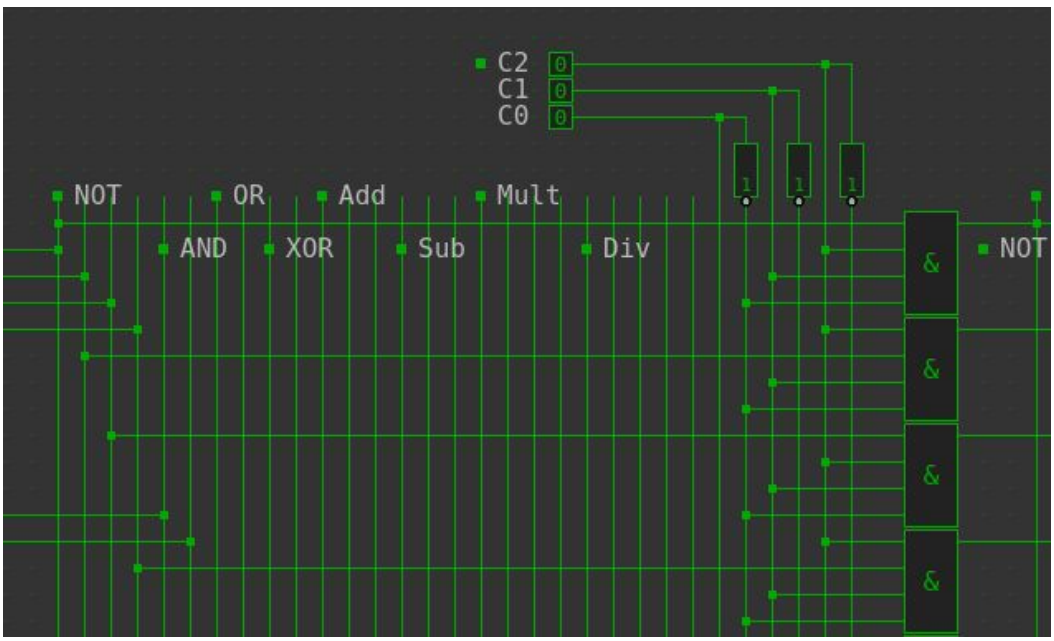
A simulação do circuito pode ser visualizada na íntegra no seguinte link: https://simulator.io/board/uoRFeyr6Ty/11

O circuito inicia-se por definir as variáveis de entrada A1, A0, B1 e B0 que representam os bits a ser utilizados nas operações aritméticas e/ou lógicas. Estas variáveis sao transferidas para a entrada de cada módulo.

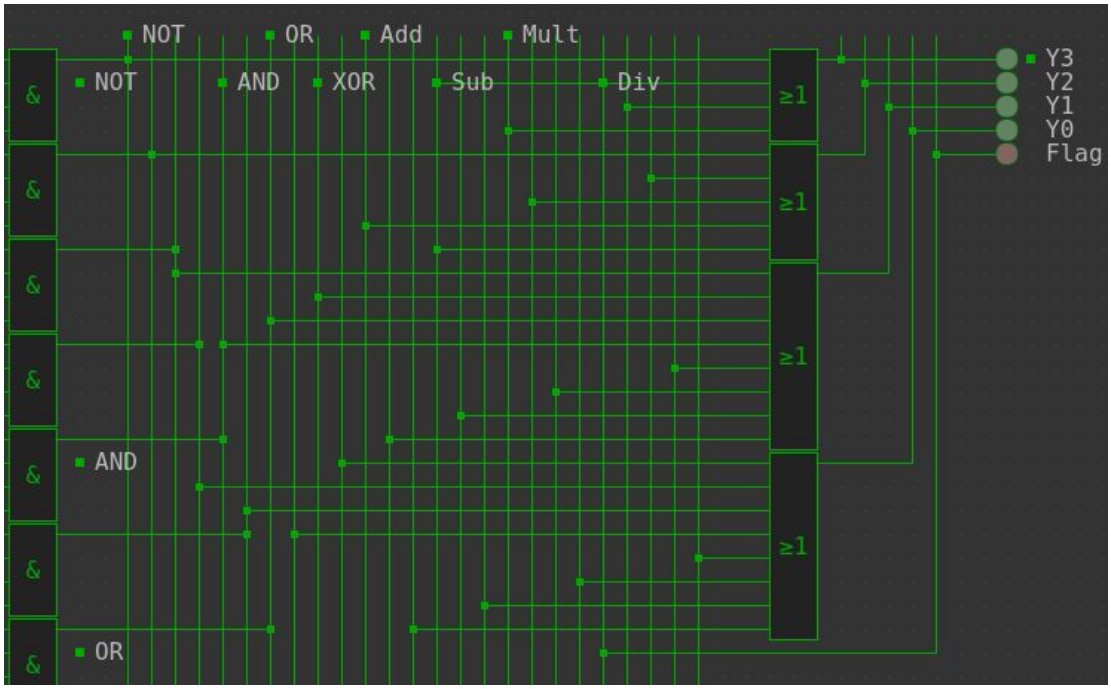


Módulo da operação lógica NOT

Os módulos sao construídos e utilizam as expressões lógicas deduzidas pelos mapas de Karnaugh; podem ser consultados por completo no seguinte anexo.



Bits de entrada C2, C1, C0 para a selecção da operação



Selecção da operação a ser mostrada

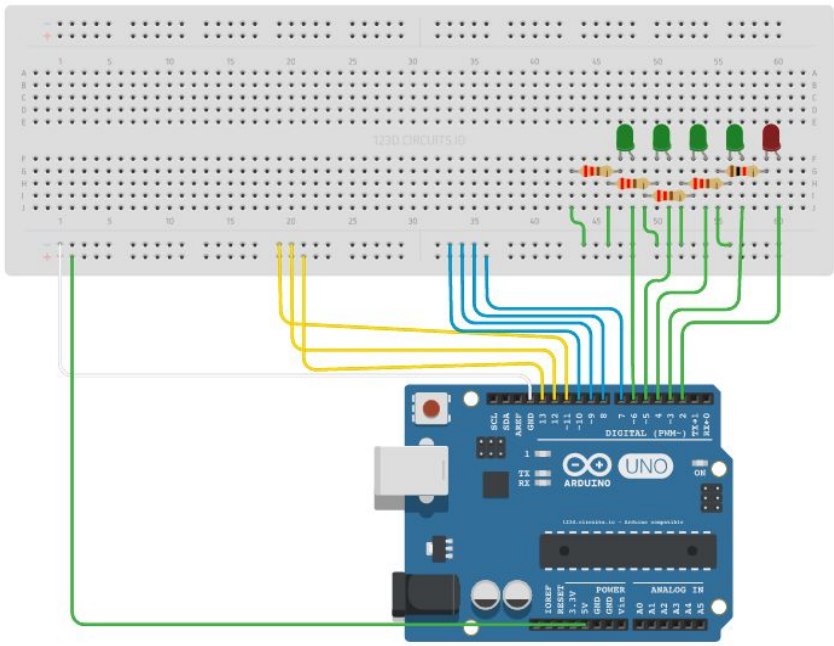
O resultado de cada módulo é encaminhado para o bloco de seleção de operação, onde são realizadas operações lógicas AND com cada variável e cada bit resultante das operações aritméticas e lógicas. Desta forma, consegue-se filtrar apenas a operação seleccionada.

A atribuição final das variáveis Y3, Y2, Y1, Y0 e Flag de erro é realizada por um conjunto de portas lógicas OR com os bits das operações que usem essas variáveis de saída.

SIMULAÇÃO EM ARDUINO

Neste circuito, começamos por montar um cabo aos 5v e outro ao GROUND, ambos ligados aos “power rails” da breadboard nos seus devidos lugares. Em seguida, montaram-se os 5 LEDS visíveis na imagem abaixo: 1 LED vermelho para representar a FLAG e 4 LEDS verdes para representar os bits de saída e um pino digital que liga à maior perna do LED - pois tem polaridade. Quanto aos bits de entrada e de seleção efetuou-se simplesmente a ligação de 7 cabos a 7 pinos digitais do arduino. Para dar um determinado valor booleano (0 ou 1) mudaram-se os cabos nos “power rails”; para o valor 0 ligou-se o cabo ao power rail ligado ao GROUND do arduino e, ainda, para atribuir o valor 1 ligou-se o cabo ao power rail ligado aos 5v do arduino.

A simulação da ALU no Arduino pode ser visualizada na íntegra no seguinte link: https://circuits.io/circuits/4521482-cf-p01



Simulação do circuito em Arduino

```
void loop() {
   read_inputs();
   calculate();
   write_outputs();
}
```

```
void read_inputs() {
   _A1 = digitalRead(PIN_A1);
   _A0 = digitalRead(PIN_A0);
   _B1 = digitalRead(PIN_B1);
   _B0 = digitalRead(PIN_B0);
```

O Arduino implementa uma função loop() que realiza exatamente o que o nome sugere: a função loop é executada consecutivamente permitindo ao programa responder a pedidos, realizar outro tipo de operações ao longo do tempo e controlar a placa do Arduino. Assim, são executadas as funções de leitura de inputs, a realização de cálculos sobre esses inputs e escrita de resultados (outputs).

```
boolean div_q1() { return _A1 & !_B1; }
boolean div_q0() { return _A0 & _A1 | _A1 & _B1 & !_B0 | _A0 & _B0 & !_B1; }
boolean div_r1() { return _B0 & _B1 & _A1 & !_A0; }
boolean div_r0() { return _A0 & _B0 & !_A1 | _A0 & _B1 & !_B0; }
boolean div_dz() { return !_B1 & !_B0; }
```

Foram definidas as expressões lógicas de cada variável em várias funções. Estas funções começam com o nome da operação fazendo-se seguir o nome da variável e o respectivo bit.

```
boolean op_div(boolean bit) { return bit & !_C2 & _C1 & _C0; }
```

```
void calculate() {
   _Y3 = 0;
   _Y2 = 0;
   _Y1 = 0;
   _Y0 = 0;
   _FLAG = 0;


   _Y3 = op_div(div_q1()) | op_mult(mult_s3()) | op_not(not_g1()) |
op_sub(sub_s2());
   _Y2 = op_div(div_q0()) | op_mult(mult_s2()) | op_not(not_g0()) |
op_sub(sub_s2()) | op_add(add_s2());
   _Y1 = op_div(div_r1()) | op_mult(mult_s1()) | op_not(not_h1()) |
op_sub(sub_s1()) | op_add(add_s1()) | op_xor(xor_f1()) | op_or(or_f1()) |
op_and(and_f1());
   _Y0 = op_div(div_r0()) | op_mult(mult_s0()) | op_not(not_h0()) |
op_sub(sub_s0()) | op_add(add_s0()) | op_xor(xor_f0()) | op_or(or_f0()) |
```

```
op_and(and_f0());
   _FLAG = op_div(div_dz());
}
```

A selecção de cada operação é realizada pelas funções, cujo nome se inicializa por “op”, o nome de operação, sendo todas as funções submetidas a operações lógicas OR; o respectivo resultado é atribuído às variáveis de saídas para ser posteriormente escrito.

O código da ALU pode ser consultado por completo no seguinte link ou em anexo.

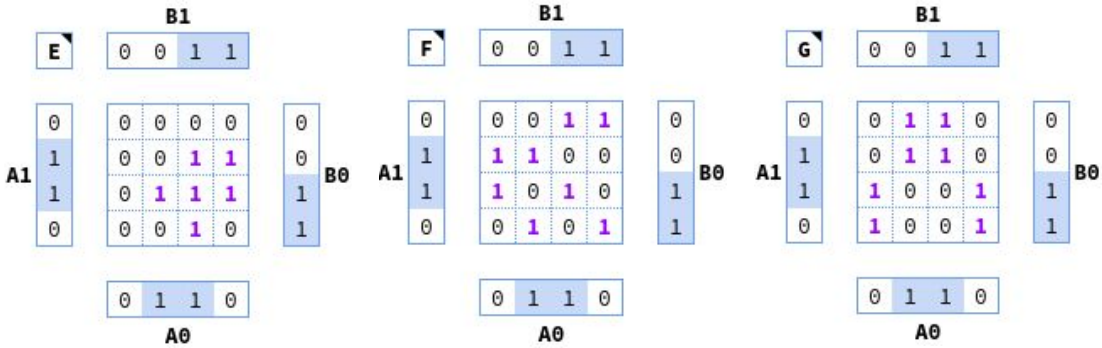
CONCLUSÃO

Um dos pontos mais importantes que é possível concluir passa pela construção de circuitos através de expressões lógicas deduzidas por tabelas de verdade e mapas de Karnaugh e, posteriormente, por um circuito de portas lógicas. Embora apenas tenha sido projectado um conjunto de operações aritméticas e lógicas, facilmente se percebe a possibilidade de implementar o mesmo processo para outros problemas.

A abordagem modular permite a resolução do problema através de uma abordagem incremental e isolada pela funcionalidade de cada módulo. Esta divisão permitiu a resolução de um erro detectado na fase final do projecto em que a operação aritmética da subtração apresentava valores errados. Este problema implicou a produção de uma nova tabela de verdade para esta operação e, por consequência, a realização de novos mapas de Karnaugh e novas expressões. Como esta operação estava isolada das restantes facilmente se procedeu à substituição do circuito no seu módulo.

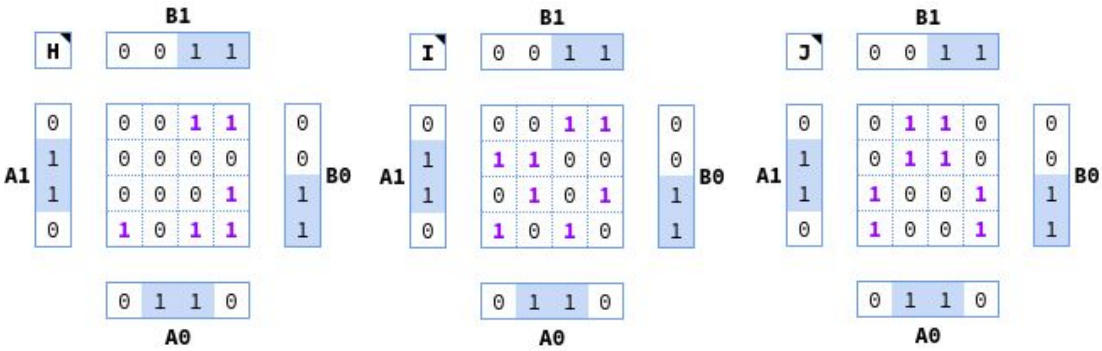
ANEXO - MAPAS DE KARNAUGH

ADIÇÃO



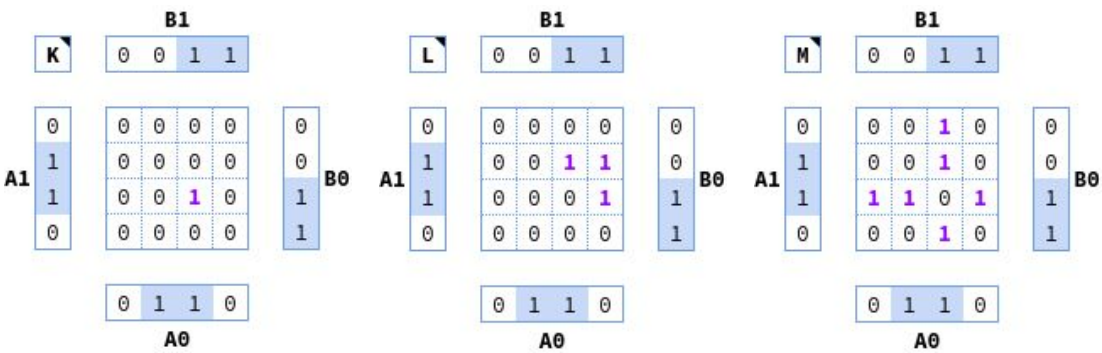
Add. S2 Add. S1 Add. S0

SUBTRACCÃO

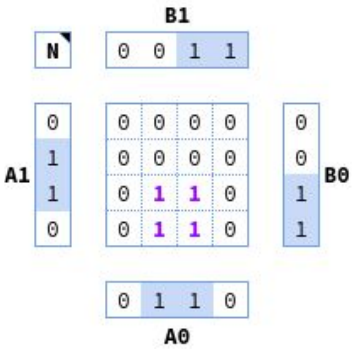


Sub. S2 Sub. S1 Sub. S0

MULTIPLICACÃO

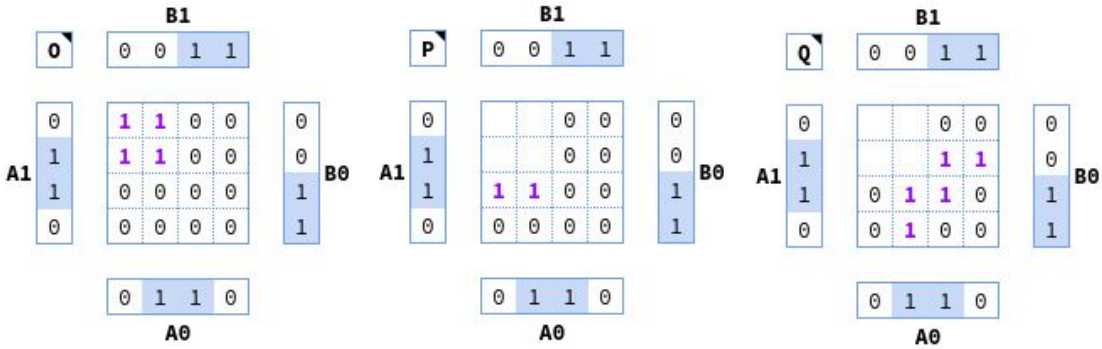


Mult. S3 Mult. S2 Mult. S1

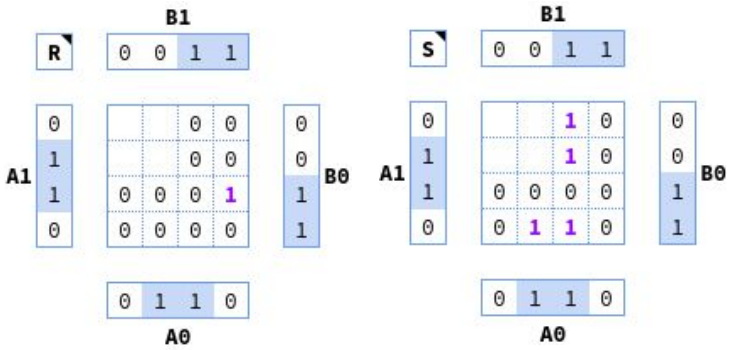


Mult. S0

DIVISÃO

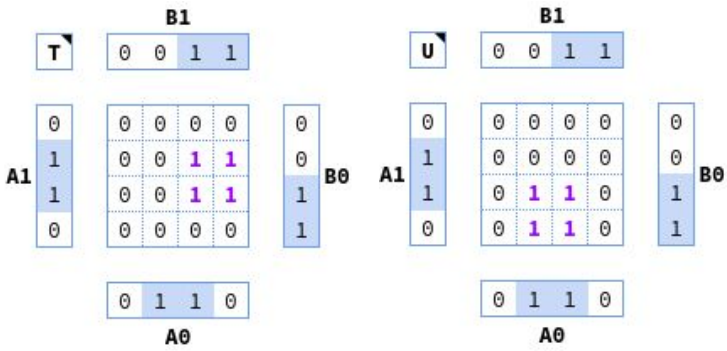


Div. DZ Div. Q1 Div. Q0



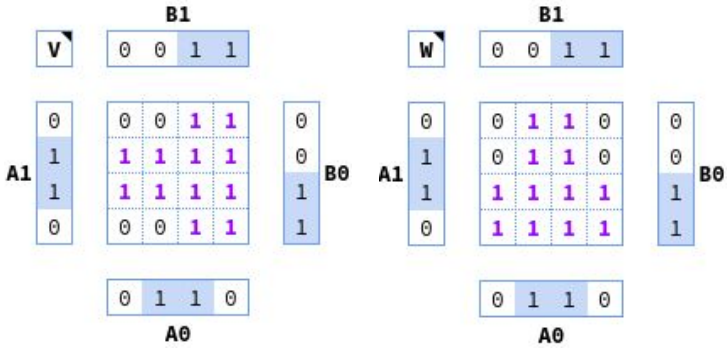
Div. R1 Div. Q0

AND



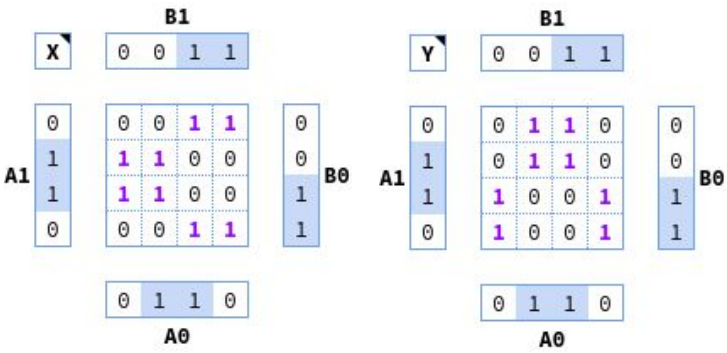
AND F1 AND F0

OR



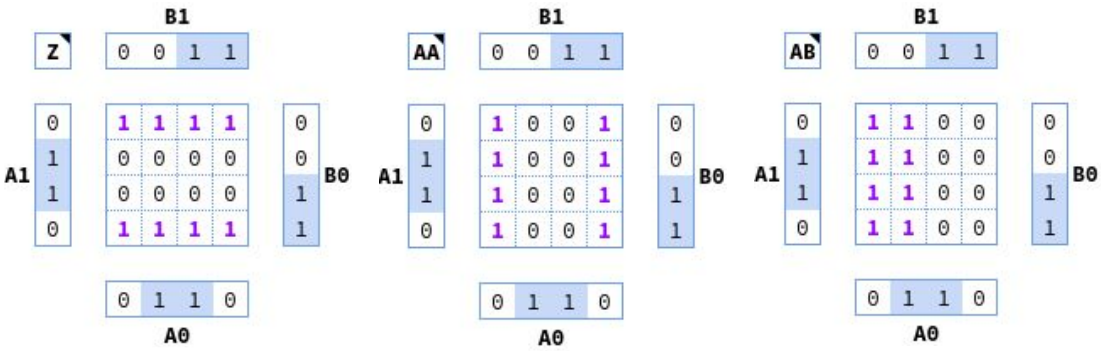
OR F1 OR F0

XOR

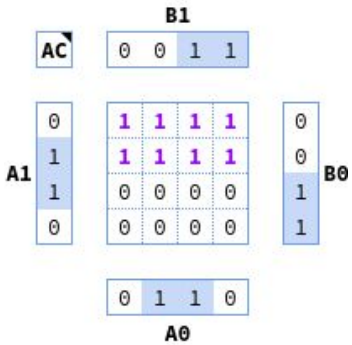


XOR F1 XOR F0

NOT

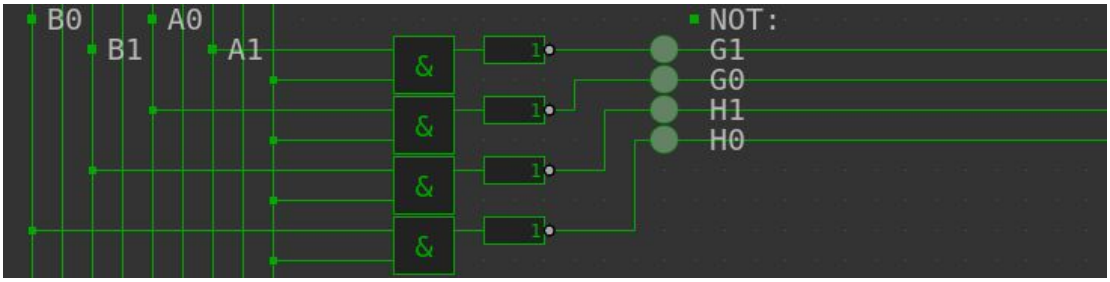


NOT G1 NOT G0 NOT H1

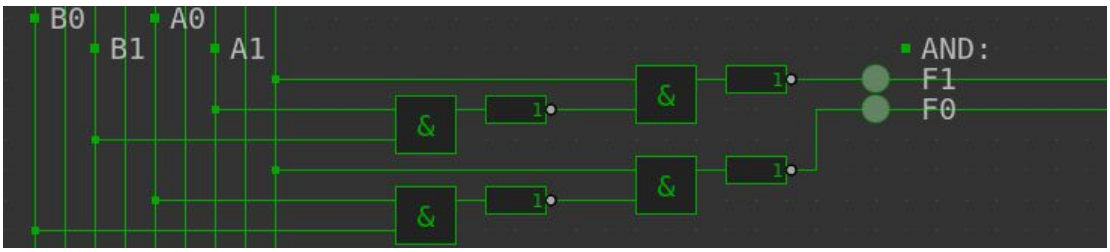


NOT H0

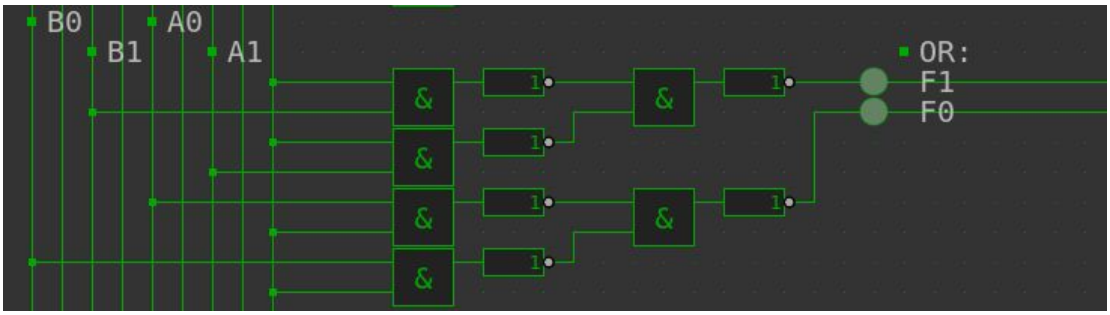
ANEXO - CIRCUITOS COMBINATÓRIOS DE CADA MÓDULO



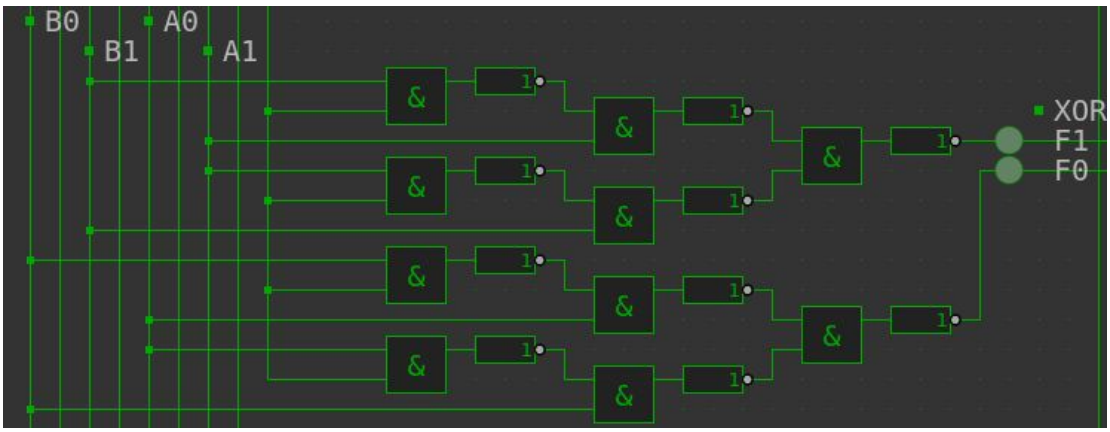
Módulo da operação lógica NOT



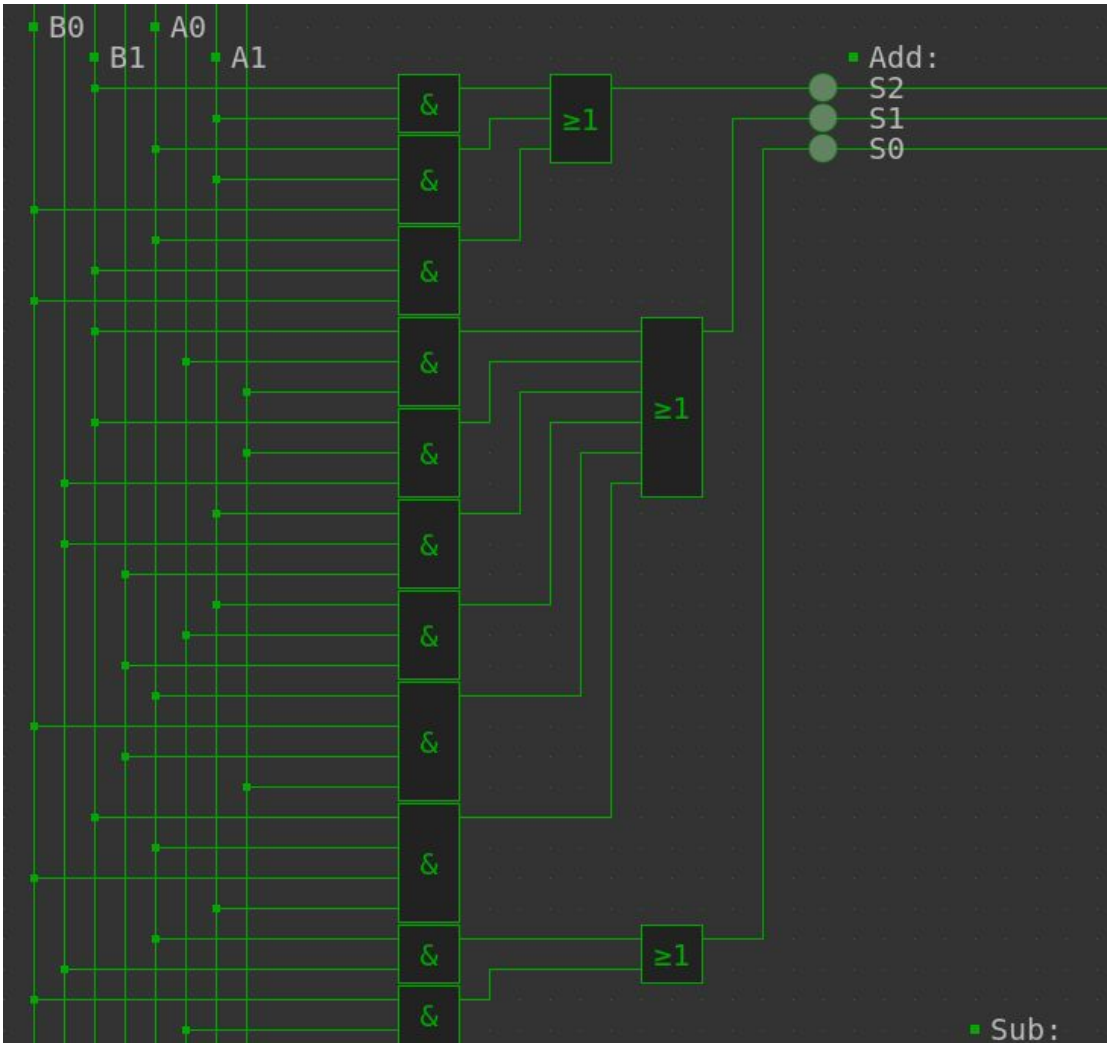
Módulo da operação lógica AND



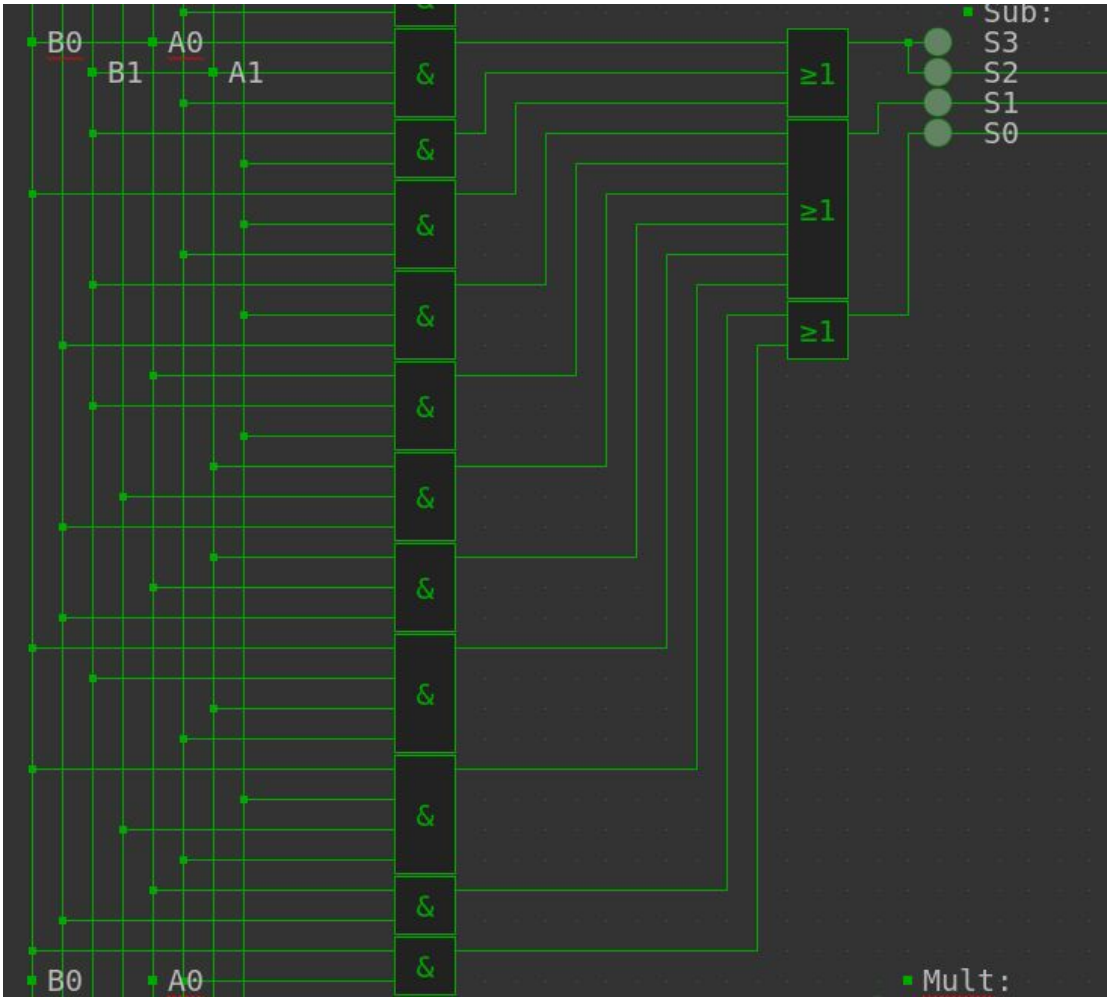
Módulo da operação lógica OR



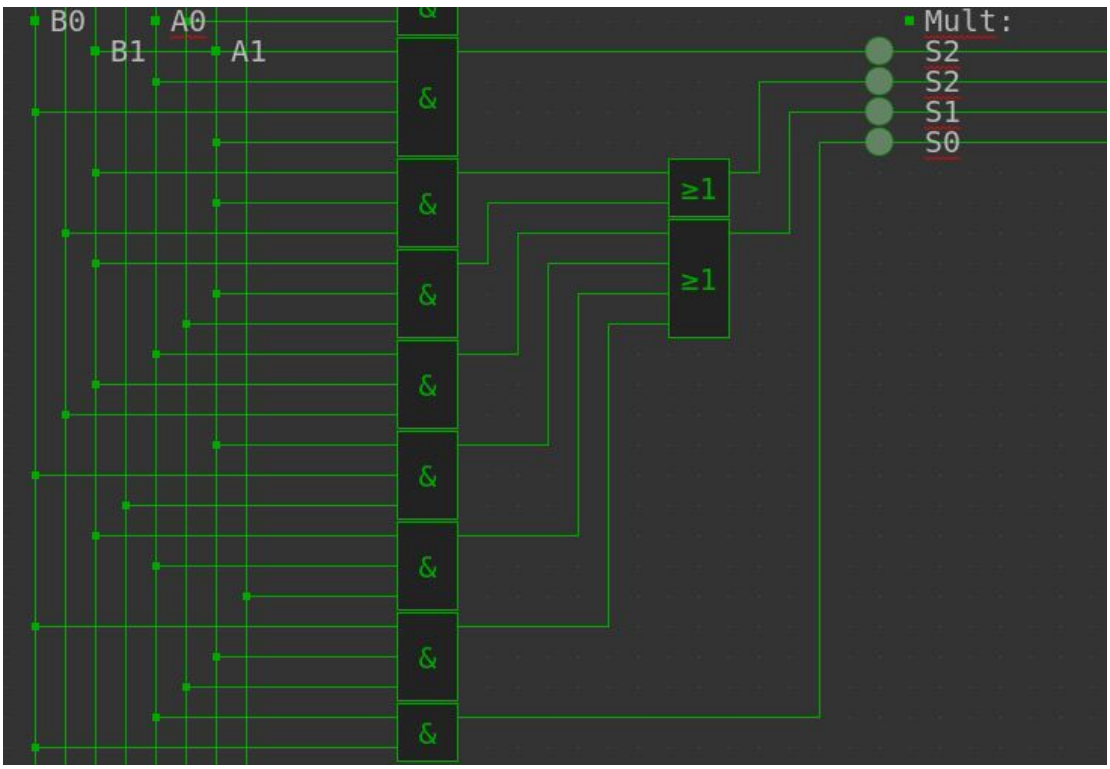
Módulo da operação lógica XOR



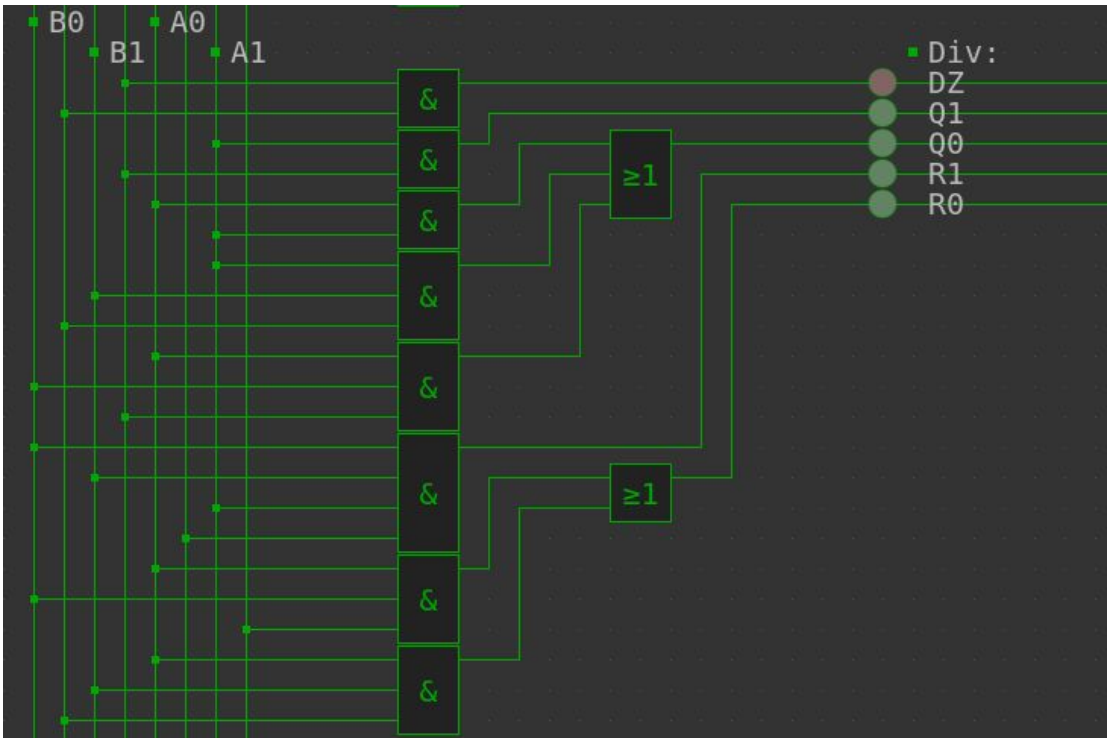
Módulo da operação aritmética adição



Módulo da operação aritmética subtracção



Módulo da operação aritmética multiplicação



Módulo da operação aritmética divisão

ANEXO - CÓDIGO ALU

```
#define PIN_A1 10
#define PIN_A0 9
#define PIN_B1 8
#define PIN_B0 7
#define PIN_C2 11
#define PIN_C1 12
#define PIN_C0 13
#define PIN_Y3 6
#define PIN_Y2 5
#define PIN_Y1 4
#define PIN_Y0 3
#define PIN_FLAG 2

// Input variables
boolean _A1;
boolean _A0;
boolean _B1;
boolean _B0;

// Operation selectors
boolean _C2;
boolean _C1;
boolean _C0;

// Output variables
boolean _Y3;
boolean _Y2;
boolean _Y1;
boolean _Y0;
boolean _FLAG;

void setup() {
   Serial.begin(9600);
   pinMode(PIN_A1, INPUT);
   pinMode(PIN_A0, INPUT);
   pinMode(PIN_B1, INPUT);
   pinMode(PIN_B0, INPUT);
   pinMode(PIN_C2, INPUT);
   pinMode(PIN_C1, INPUT);
   pinMode(PIN_C0, INPUT);
   pinMode(PIN_Y3, OUTPUT);
   pinMode(PIN_Y2, OUTPUT);
   pinMode(PIN_Y1, OUTPUT);
   pinMode(PIN_Y0, OUTPUT);
   pinMode(PIN_FLAG, OUTPUT);
}

void read_inputs() {
   _A1 = digitalRead(PIN_A1);
   _A0 = digitalRead(PIN_A0);
   _B1 = digitalRead(PIN_B1);
   _B0 = digitalRead(PIN_B0);
   _C2 = digitalRead(PIN_C2);
   _C1 = digitalRead(PIN_C1);
   _C0 = digitalRead(PIN_C0);
}


void calculate() {
   _Y3 = 0;
   _Y2 = 0;
   _Y1 = 0;
   _Y0 = 0;
   _FLAG = 0;


   _Y3 = op_div(div_q1()) | op_mult(mult_s3()) | op_not(not_g1()) | op_sub(sub_s2());
```

```
    _Y2  =  op_div(div_q0())  |  op_mult(mult_s2())  |  op_not(not_g0())  |  op_sub(sub_s2())  |
op_add(add_s2());
    _Y1  =  op_div(div_r1())  |  op_mult(mult_s1())  |  op_not(not_h1())  |  op_sub(sub_s1())  |
op_add(add_s1()) | op_xor(xor_f1()) | op_or(or_f1()) | op_and(and_f1());
    _Y0  =  op_div(div_r0())  |  op_mult(mult_s0())  |  op_not(not_h0())  |  op_sub(sub_s0())  |
op_add(add_s0()) | op_xor(xor_f0()) | op_or(or_f0()) | op_and(and_f0());
   _FLAG = op_div(div_dz());
}


boolean nand(boolean A, boolean B) { return !(A & B); }


boolean op_not(boolean bit) { return bit & _C2 & _C1 & !_C0; }
boolean op_div(boolean bit) { return bit & !_C2 & _C1 & _C0; }
boolean op_mult(boolean bit) { return bit & !_C2 & _C1 & !_C0; }
boolean op_add(boolean bit) { return bit & !_C2 & !_C1 & !_C0; }
boolean op_sub(boolean bit) { return bit & !_C2 & !_C1 & _C0; }
boolean op_xor(boolean bit) { return bit & _C2 & _C1 & _C0; }
boolean op_or(boolean bit) { return bit & _C2 & !_C1 & _C0; }
boolean op_and(boolean bit) { return bit & _C2 & !_C1 & !_C0; }


boolean not_g1() { return nand(_A1, 1); }
boolean not_g0() { return nand(_A0, 1); }
boolean not_h1() { return nand(_B1, 1); }
boolean not_h0() { return nand(_B0, 1); }


boolean div_q1() { return _A1 & !_B1; }
boolean div_q0() { return _A0 & _A1 | _A1 & _B1 & !_B0 | _A0 & _B0 & !_B1; }
boolean div_r1() { return _B0 & _B1 & _A1 & !_A0; }
boolean div_r0() { return _A0 & _B0 & !_A1 | _A0 & _B1 & !_B0; }
boolean div_dz() { return !_B1 & !_B0; }


boolean mult_s3() { return _B1 & _A0 & _B0 & _A1; }
boolean mult_s2() { return _B1 & _A1 & !_B0 | _B1 & _A1 & !_A0; }
boolean mult_s1() { return _A0 & _B1 & !_B0 | _A1 & _B0 & !_B1 | _B1 & _A0 & !_A1 | _B0 & _A1
& !_A0; }
boolean mult_s0() { return _A0 & _B0; }


boolean add_s2() { return _B1 & _A1 | _A0 & _A1 & _B0 | _A0 & _B1 & _B0; }
boolean add_s1() { return _B1 & !_A0 & !_A1 | _B1 & !_A1 & !_B0 | _A1 & !_B0 & !_B1 | _A1 &
!_A0 & !_B1 | _A0 & _B0 & !_B1 & !_A1 | _B1 & _A0 & _B0 & _A1; }
boolean add_s0() { return _A0 & !_B0 | _B0 & !_A0; }


boolean sub_s2() { return _B1 & !_A1 | _B0 & _B1 & !_A0 | _B0 & !_A0 & !_A1; }
boolean sub_s1() { return _B1 & !_A1 & !_B0 | _B1 & _A0 & !_A1 | _B0 & _B1 & _A1 & !_A0 | _A1
& !_B1 & !_B0 | _A1 & _A0 & !_B1 | _B0 & !_A1 & !_B1 & !_A0; }
boolean sub_s0() { return _A0 & !_B0 | _B0 & !_A0; }


boolean xor_f1() { return nand(nand(_A1, nand(_B1, 1)), nand(_B1, nand(_A1, 1))); }
boolean xor_f0() { return nand(nand(_A0, nand(_B0, 1)), nand(_B0, nand(_A0, 1))); }


boolean or_f1() { return nand(nand(_B1, 1), nand(_A1, 1)); }
boolean or_f0() { return nand(nand(_A0, 1), nand(_B0, 1)); }


boolean and_f1() { return nand(nand(_A1, _B1), 1); }
boolean and_f0() { return nand(nand(_A0, _B0), 1); }


void write_outputs() {
   digitalWrite(PIN_Y3, _Y3);
   digitalWrite(PIN_Y2, _Y2);
   digitalWrite(PIN_Y1, _Y1);
   digitalWrite(PIN_Y0, _Y0);
   digitalWrite(PIN_FLAG, _FLAG);
```

```
}


void loop() {
   read_inputs();
   calculate();
   write_outputs();
}
```