Departamento de Engenharia
Eletrónica e Telecomunicações
e de Computadores

Licenciatura em Engenharia Informática e de Computadores
e
Licenciatura em Engenharia Informática, Redes e Telecomunicações

Circuitos aritméticos e Lógicos
(*2º Trabalho de Laboratório*)

Trabalho realizado por:

Nome: Daniel Santos	Nº51701
Nome: Paulo Magalhães	Nº51702

Docente: Pedro Miguens Matutino

Lógica e Sistemas Digitais
2023 / 2024 inverno

22 de novembro de 2023

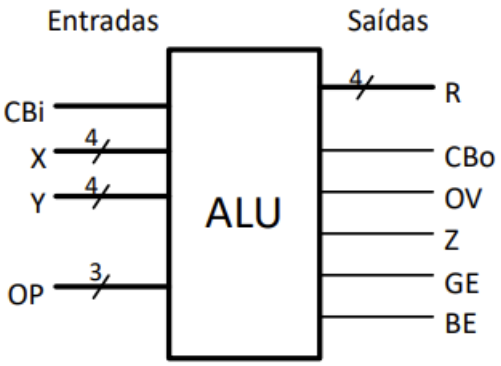
1 Objetivo

O objetivo deste trabalho é descrever um circuito aritmético e lógico (ALU – *Aritmetic and Logic Unit*) com base em VHDL estrutural, simular e implementar o circuito na placa de desenvolvimento DE10-Lite da Intel.

2 Descrição do Circuito a Projetar

Pretende-se implementar uma unidade aritmética que faça as operações aritméticas adição (X + Y + CBi), subtração (X − Y − CBi), incremento (X + CBi) e decremento (X − CBi), as operações de deslocamento (X >> 1 e X >>> 1), e as operações lógicas OR (X | Y) e AND (X & Y), sobre operandos de 4 bits. O resultado tem 4 bits e deve gerar os indicadores (*flags*) de erro e relação *Carry*/*Borrow* (CBo), *Overflow* (OV), *Zero* (Z), *Greater or Equal (*GE) e *Below or Equal* (BE). Para a execução deste projeto, conseguiu-se aproveitar múltiplos componentes feitos em projetos anteriores como o módulo completo que efetua as operações de adição e subtração (*Adder*/*Subtractor*).

As entradas e saídas do sistema, bem como as operações a realizar de acordo com cada combinação da entrada OP, estão representadas na Figura 1.



$OP_{2..0}$	Operação	CBo	OV	Z	GE	BE
0 0 0	X >>> 1	X	–	X	–	–
0 0 1	X >> 1	X	–	X	–	–
0 1 0	X | Y	–	–	X	–	–
0 1 1	X & Y	–	–	X	–	–
1 0 0	$X + CBi$	X	X	X	–	–
1 0 1	$X - CBi$	X	X	X	–	–
1 1 0	$X + Y + CBi$	X	X	X	–	–
1 1 1	$X - Y - CBi$	X	X	X	X	X

Figura 1 – Entradas e saídas da unidade aritmética a desenvolver.

Como referido anteriormente, além do resultado serão geradas as seguintes flags:

- CBo: Representa o *carry* de saída da operação de soma ou o *borrow* de saída da operação de subtração. Nas operações de deslocamento, representa o valor do último bit deslocado de X;
- OV: Fica ativa quando o resultado excede o domínio dos números relativos;
- Z: Fica ativa quando o resultado é igual a zero;

- GE: Fica ativa quando o primeiro operando (X) é maior ou igual do que o segundo (Y + CBi), considerando-se apenas na representação de números relativos;
- BE: Fica ativa quando o primeiro operando (X) é menor ou igual do que o segundo (Y + CBi), considerando-se apenas na representação de números naturais.

Dependendo da operação, o valor de algumas das flags pode não ter significado (o valor que assume não interessa), estando assim representadas pelo carater '-'.

3 Projeto do Circuito

3.1 Diagrama de blocos do circuito

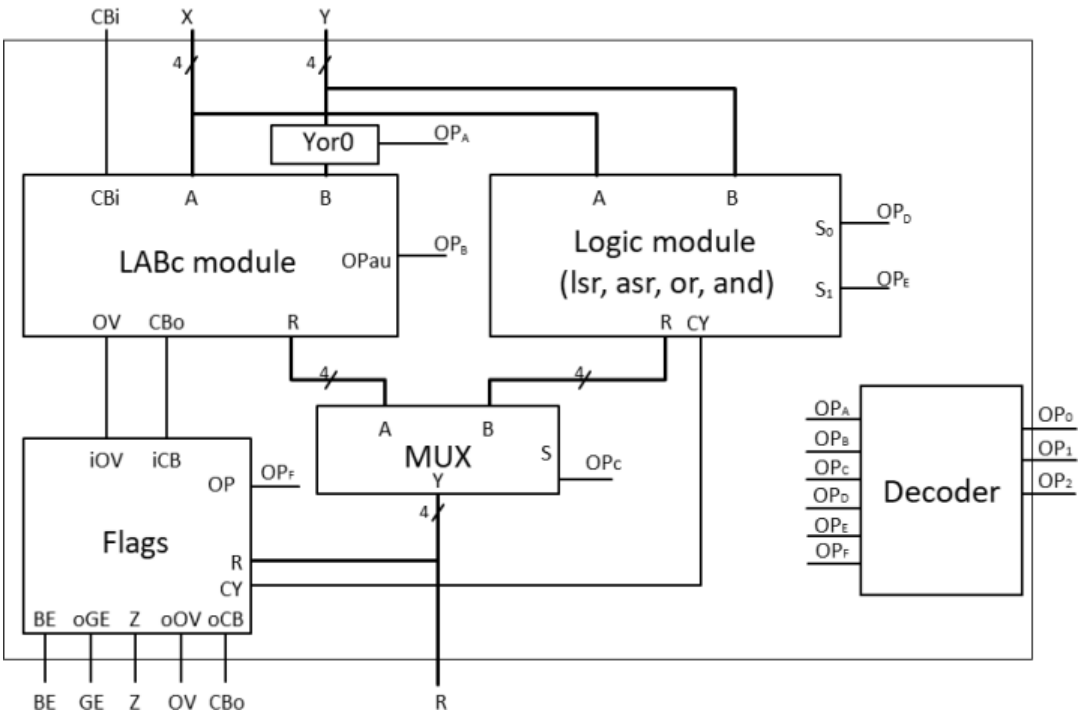


Figura 2 – Diagrama de blocos da ALU

Como é possível visualizar previamente na Figura 2, este circuito é constituído pelos seguintes componentes:

- Yor0 – É responsável por colocar o valor de Y a 0 para efetuar as operações de incremento e decremento;
- *Logic module* – Com base nas entradas OPd e OPe, efetua as operações de deslocamento e as operações lógicas. Também calcula uma *flag* de *carry* para o módulo *flags* do circuito principal;
- MUX – escolhe entre o resultado do *LABc module* e do *Logic module*;

- *Decoder* – Descodifica o valor do OP de 3 bits para calcular 6 novos valores que irão ser usados no circuito para decidir quais as operações a escolher;

- *Flags* – Calcula o valor de todos os indicadores para serem mostrados no final. O significado de cada um deles já foi representado no ponto 2 "Descrição do circuito a projetar".

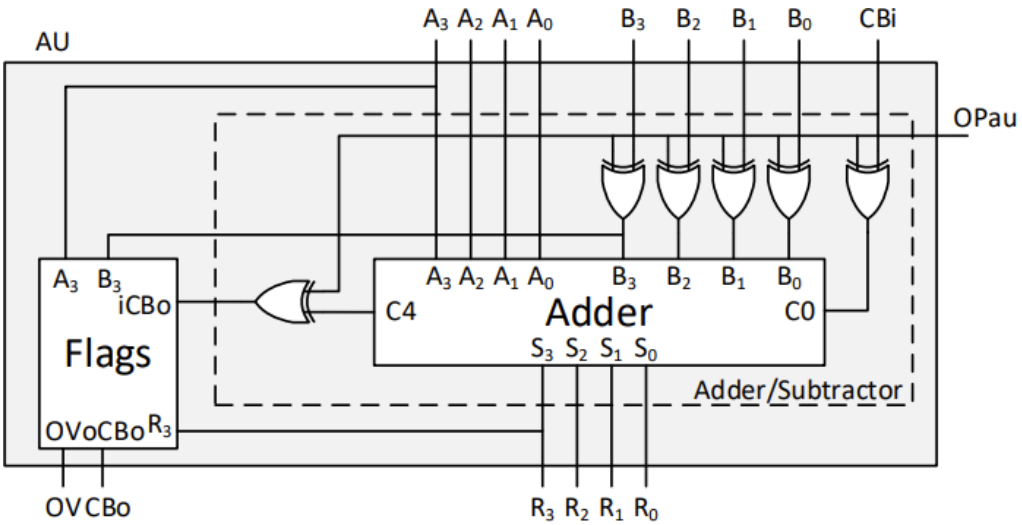


Figura 3 – Diagrama de blocos da unidade aritmética (*LABc module*)

- *LABc module* – É a unidade aritmética que calcula a soma ou a subtração dos 2 operandos, juntamente com o *carry* de entrada. É constituído pelas entradas X e Y, operandos de 4 bits, CBi (*carry*/*borrow* de entrada) e o OPau (indicador se a operação consiste de uma adição ou subtração). A descrição deste módulo também é dividida por diversos blocos como se pode visualizar na Figura 3 e também da seguinte maneira:
    - *Adder*/*Subtractor* – troca o valor de B de modo a efetuar a subtração se o OPau se encontrar ativo e manda o valor dos operandos para o *Adder*;
    - *Adder* – efetua a adição entre os 2 operandos;
    - *Flags* – Gera as *flags* de *overflow* e CBo (*carry*/*borrow out*) para o módulo *flags* do circuito principal. O *carry*/*borrow* que veio não irá sofrer nenhuma alteração, já o *overflow* é calculado verificando o 3º bit dos operandos de entrada A e B e o resultado R;

3.2 Funções lógicas dos circuitos;

- OPA => OP(1);

	OP0	OP0/OP1	OP1	
-	-	-	-	
0	0	1	1	OP2

- OPB => OP(0);

	OP0	OP0/OP1	OP1	
-	-	-	-	
0	1	1	0	OP2

- OPC => OP(2);

	OP0	OP0/OP1	OP1	
-	-	-	-	
1	1	1	1	OP2

- OPD => OP(1);

	OP0	OP0/OP1	OP1	
0	0	1	1	
-	-	-	-	OP2

- OPE => OP(0);

	OP0	OP0/OP1	OP1	
0	1	1	0	
-	-	-	-	OP2

- OPF => OP(2);

	OP0	OP0/OP1	OP1	
-	-	-	-	
1	1	1	1	OP2

- LSR

Cy = 1

- ASR

Cy = 1

3.3 Simplificação lógica

- Decoder (não há simplificação possível):
  - OPa <= OP(1);
  - OPb <= OP(0);
  - OPc <= OP(2);
  - OPd <= OP(1);
  - OPe <= OP(0);
  - OPf <= OP(2);

3.4 Descrição VHDL

3.4.1 Tlab2.VHD

```
library ieee;
use ieee.std_logic_1164.all;
entity tlab2 is
      port (
            X, Y: std_logic_vector(3 downto 0);
            OP: std_logic_vector(2  downto  0);
            R: out std_logic_vector(3 downto 0);

            CBi: std_logic;
            CBo, OV, Z, GE, BE: out std_logic
      );
end tlab2;
architecture tlab2_logic of tlab2 is
component au is
      port (
            A, B: std_logic_vector(3 downto 0);
            R: out std_logic_vector(3 downto 0);

            CBi, OPau: std_logic;
            CBo, OV: out std_logic
      );
end component;
component logic_module is
      port (
            A, B: std_logic_vector(3 downto 0);
            R: out std_logic_vector(3 downto 0);

            S0, S1: std_logic;
            Cy: out std_logic
      );
end component;
component mux4 is
      port (
                  A, B: std_logic_vector(3 downto 0);
                  S: std_logic;
         R: out std_logic_vector(3 downto 0)
       );
end component;
component decoder is
      port (
            OP: std_logic_vector (2 downto 0);
            OPa, OPb, OPc, OPd, OPe, OPf: out std_logic
```

```
	);
end component;
component Yor0 is
	port (
		Y: std_logic_vector(3 downto 0);
		R: out std_logic_vector(3 downto 0);
		OPa: std_logic
	);
end component;
component flags_main is
	port (
		R: std_logic_vector(3 downto 0);
		iOV, iCB, OP, CY: std_logic;

		BE, oGE, Z, oOV, oCB: out std_logic
	);
end component;
signal OPaS, OPbS, OPcS, OPdS, OPeS, OPfS: std_logic;
signal inAu: std_logic_vector(3 downto 0);
signal outAu, outLogic: std_logic_vector(3 downto 0);
signal auOV, auCBo, logicCY: std_logic;
signal outMUX: std_logic_vector(3 downto 0);
begin
	R <= outMUX;
	Decode: decoder port map(
		OP => OP,
		OPa => OPaS,
		OPb => OPbS,
		OPc => OPcS,
		OPd => OPdS,
		OPe => OPeS,
		OPf => OPfS
	);

	U1_B: Yor0 port map(
		Y => Y,
		OPa => OPaS,
		R => inAu
	);
	U1: au port map(
		A => X,
		B => inAu,
		CBi => CBi,
		OPau => OPbS,
		R => outAu,
		OV => auOV,
		CBo => auCBo
	);

	U2: logic_module port map(
		A => X,
		B => Y,
		S0 => OPdS,
		S1 => OPeS,
		R => outLogic,
		CY => logicCY
	);
```

```
	U3: mux4 port map(
		A => outAu,
		B => outLogic,
		R  =>  outMUX,
		S => OPcS
	);

	U4: flags_main port map(
		iOV => auOV,
		iCB => auCBo,
		OP  =>  OPfS,
		R => outMUX,
		CY => logicCY,
		BE => BE,
		oGE => GE,
		Z => Z,
		oOV => OV,
		oCB => CBo
	);
end architecture;
```

O bloco TLAB2, sendo a entidade de topo do projeto, recebe 2 operandos de 4 bits (X e Y), um operando de 3 bits (OP) e um operando de um bit (CBi). Como já foi descrito, criámos diversos componentes, onde tivemos a oportunidade de aproveitar múltiplos componentes feitos em projetos passados o que ajudou no tempo despendido a fazer a descrição VHDL. Quando se refere à AU (*Aritmetic Unit*), está-se a referir ao *LABc Module* (é uma unidade aritmética, mas como foi desenvolvida no LABc, no projeto é denominado de *LABc Module*).

3.4.2 Yor0.VHD

```
library ieee;
use ieee.std_logic_1164.all;
entity Yor0 is
	port (
		Y: std_logic_vector(3 downto 0);
		R: out std_logic_vector(3 downto 0);
		OPa: std_logic
	);
end Yor0;
architecture Yor0_logic of Yor0 is
signal OPa_vector: std_logic_vector(3 downto 0);
begin
	OPa_vector(3 downto 0) <= (others => OPa);
	R <= OPa_vector and Y;
end architecture;
```

Yor0 – Tem o objetivo de colocar o operando B a 0 caso estejamos a falar de operações de incremento/decremento usando o valor do OPa.

3.4.3 OR.VHD & AND.VHD

```
library ieee;
use ieee.std_logic_1164.all;
entity logicor is
      port (
            A, B: std_logic_vector(3 downto 0);
            R: out std_logic_vector(3 downto 0)
      );
end logicor;
architecture logicor_logic of logicor is
begin
   R <= A or B;
end architecture;


library ieee;
use ieee.std_logic_1164.all;
entity logicand is
      port (
            A, B: std_logic_vector(3 downto 0);
            R: out std_logic_vector(3 downto 0)
      );
end logicand;
architecture logicand_logic of logicand is
begin
   R <= A and B;
end architecture;
```

OR e AND – Comparam os operandos A e B de modo a efetuar as operações lógicas no *Logic Module*.

3.4.4 LSR.VHD

```
library ieee;
use ieee.std_logic_1164.all;
entity logiclsr is
      port (
            A: std_logic_vector(3 downto 0);
            R: out std_logic_vector(3 downto 0);
            Cy: out std_logic
      );
end logiclsr;
architecture logiclsr_logic of logiclsr is
begin
      Cy <= A(0);
      R(0) <= A(1);
      R(1) <= A(2);
      R(2) <= A(3);
      R(3) <= '0';
end architecture;
```

LSR (*Logical Shift Right*) – Responsável por fazer uma das operações de deslocamento onde desloca todos os bits uma casa para a direita, guarda o bit 0 no *carry* de saída e coloca no 3º bit o valor 0.

3.4.5 ASR.VHD

```
library ieee;
use ieee.std_logic_1164.all;
entity logicasr is
      port (
            A: std_logic_vector(3 downto 0);
            R: out std_logic_vector(3 downto 0);
            Cy: out std_logic
      );
end logicasr;
architecture logicasr_logic of logicasr is
begin
      Cy <= A(0);
      R(0) <= A(1);
      R(1) <= A(2);
      R(2) <= A(3);
      R(3) <= A(3);
end architecture;
```

ASR (*Aritmetic Shift Right*) – Responsável por fazer a outra operação de deslocamento que faz a mesma operação que o LSR (*Logical Shift Right*) mas no 3º bit mantém o valor.

3.4.6 logic_module.VHD

```
library ieee;
use ieee.std_logic_1164.all;
entity logic_module is
      port (
            A, B: std_logic_vector(3 downto 0);
            R: out std_logic_vector(3 downto 0);

            S0, S1: std_logic;
            Cy: out std_logic
      );
end logic_module;
architecture logic_module_logic of logic_module is
component logiclsr is
      port (
                  A: std_logic_vector(3 downto 0);
         R: out std_logic_vector(3 downto 0);
                  Cy: out std_logic
       );
end component;
component logicasr is
      port (
                  A: std_logic_vector(3 downto 0);
         R: out std_logic_vector(3 downto 0);
                  Cy: out std_logic
       );
```

```
end component;
component logicor is
      port (
                  A, B: std_logic_vector(3 downto 0);
         R: out std_logic_vector(3 downto 0)
       );
end component;
component logicand is
      port (
                  A, B: std_logic_vector(3 downto 0);
         R: out std_logic_vector(3 downto 0)
       );
end component;
signal outLSR, outASR, outOR, outAND: std_logic_vector(3 downto 0);
signal outCyLSR, outCyASR: std_logic;
signal s0_vector, s1_vector: std_logic_vector(3 downto 0);
begin
      s0_vector(3 downto 0) <= (others => S0);
      s1_vector(3 downto 0) <= (others => S1);
      R <= (not s0_vector and not s1_vector and outLSR) or (not s0_vector and
s1_vector and outASR) or (s0_vector and not s1_vector and outOR) or (s0_vector
and s1_vector and outAND);
      Cy <= (not S0 and not S1 and outCyLSR) or (not S0 and S1 and outCyASR);
      U1: logiclsr port map(
            A => A,
            R => outLSR,
            Cy => outCyLSR
      );

      U2: logicasr port map(
            A => A,
            R => outASR,
            Cy => outCyASR
      );

      U3: logicor port map(
            A => A,
            B => B,
            R => outOR
      );

      U4: logicand port map(
            A => A,
            B => B,
            R => outAND
      );
end architecture;
```

Logic Module – Recebe o valor do S0 e do S1 que vem do valor de OPd e do OPe onde será decidido entre 2 bits qual das 4 operações a realizar e faz também todas as ligações entre estes 4 módulos de forma a conseguir pegar no resultado e no *carry* (no caso das operações de deslocamento) para enviar no final ao MUX e às *Flags*.

3.4.7 au.VHD

```
library ieee;
use ieee.std_logic_1164.all;
entity au is
      port (
            A, B: std_logic_vector(3 downto 0);
            R: out std_logic_vector(3 downto 0);

            CBi, OPau: std_logic;
            CBo, OV: out std_logic
      );
end au;
architecture au_logic of au is
component adder_sub is
      port (
            A, B: std_logic_vector(3 downto 0);
            S: out std_logic_vector(3 downto 0);

            Ci, OPau: std_logic;
            Co, b3: out std_logic
      );
end component;
component flags is
      port (
            A3, B3, iCBo, R3: std_logic;
            CBo, OV: out std_logic
      );
end component;
signal b_3, iCBo: std_logic;
signal s_out: std_logic_vector(3 downto 0);

begin
       R <= s_out;
    U1: adder_sub port map(
            A => A,
            B => B,
            OPau => OPau,
            Ci => CBi,
            Co => iCBo,
            b3 => b_3,
            S => s_out
      );

      U2: flags port map(
            A3 => A(3),
            B3 => b_3,
            iCBo => iCBo,
            R3 => s_out(3),
            OV => OV,
            CBo => CBo
      );
end architecture;
```

AU (*Aritmetic Unit*) – Responsável por fazer as operações de adição/subtração e deincremento/decremento. É escolhido se irá acontecer as operações de adição/incremento ou subtração/decremento com base no valor de OPb. Faz a ligação entre o *Adder/Subtractor* de forma a obtero resultado que irá ser enviado para o MUX e também com o módulo *Flags* interno que irá calcular algumas*flags*.

3.4.8 adder_sub.VHD

```
library ieee;
use ieee.std_logic_1164.all;
entity adder_sub is
	port (
		A, B: std_logic_vector(3 downto 0);
		S: out std_logic_vector(3 downto 0);

		Ci, OPau: std_logic;
		Co, b3: out std_logic
	);
end adder_sub;
architecture adder_sub_logic of adder_sub is
component adder is
	port (
		A, B: std_logic_vector(3 downto 0);
		S: out std_logic_vector(3 downto 0);

		Ci: std_logic;
		Co: out std_logic
	);
end component;
signal bt, opau_vector: std_logic_vector(3 downto 0);
signal Cit, Cot: std_logic;
begin
	opau_vector(3 downto 0) <= (others => OPau);
	bt <= B xor opau_vector;

	Cit <= Ci xor OPau;
	Co <= Cot xor OPau;

	b3 <= B(3) xor Opau;
	U1: adder port map(
		A => A,
		B => bt,
		Ci => Cit,
		Co => Cot,
		S => S
	);
end architecture;
```

Adder/Subtractor – Trata de enviar o valor dos operandos A e B (que poderá estar transposto dependendo da operação escolhida) juntamente com o valor do *carry* de entrada e o *carry* de saída (também poderão estar transpostos de acordo com a operação escolhida) para o módulo *Adder*.

3.4.9 adder.VHD

```
library ieee;
use ieee.std_logic_1164.all;
entity adder is
      port (
            A, B: std_logic_vector(3 downto 0);
            S: out std_logic_vector(3 downto 0);

            Ci: std_logic;
            Co: out std_logic
      );
end adder;
architecture adder_logic of adder is
component fa is
      port (
            A, B, Ci: std_logic;
            S, Co: out std_logic
      );
end component;
signal carry: std_logic_vector(3 downto 1);
begin
    U1: fa port map(
            A => A(0),
            B => B(0),
            Ci => Ci,
            S => S(0),
            Co => carry(1)
      );

      U2: fa port map(
            A => A(1),
            B => B(1),
            Ci => carry(1),
            S => S(1),
            Co => carry(2)
      );

      U3: fa port map(
            A => A(2),
            B => B(2),
            Ci => carry(2),
            S => S(2),
            Co => carry(3)
      );

      U4: fa port map(
            A => A(3),
```

```
            B => B(3),
            Ci => carry(3),
            S => S(3),
            Co => Co
      );
end architecture;
```

Adder – Composto por 4 *Full-Adder*, onde irá enviar cada bit de ambos os operandos para cada um e irá enviar por ordem os *carry's* de entrada e saída. No final irá se conseguir obter a soma de todos os 4 bits juntamente com o *carry* de saída.

3.4.10 fa.VHD

```
library ieee;
use ieee.std_logic_1164.all;
entity fa is
      port (
            A, B, Ci: std_logic;
            S, Co: out std_logic
      );
end fa;
architecture fa_logic of fa is
component ha is
      port (
            A, B: std_logic;
            S, Co: out std_logic
      );
end component;
signal addAB: std_logic;
signal carry: std_logic_vector(1 downto 0);
begin
    U1: ha port map(
            A => A,
            B => B,
            S  =>  addAB,
            Co => carry(0)
      );

      U2: ha port map(
            A => addAB,
            B => Ci,
            S => S,
            Co => carry(1)
      );

      Co <= carry(0) or carry(1);
end architecture;
```

FA (*Full-Adder*) – Composto por 2 *Half-Adder*, onde acaba por permitir a realização da soma com o *carry* de entrada.

3.4.11 ha.VHD

```
library ieee;
use ieee.std_logic_1164.all;
entity ha is
      port (
            A, B: std_logic;
            S, Co: out std_logic
      );
end ha;
architecture ha_logic of ha is
begin
    S <= A xor B;
       Co <= A and B;
end architecture;
```

HA (*Half-Adder*) – Efetua a soma entre 2 operandos de 1 bit usando a operação XOR e calcula o *carry* de saída se ambos os valores estiverem ativos.

3.4.12 flags.VHD

```
library ieee;
use ieee.std_logic_1164.all;
entity flags is
      port (
            A3, B3, iCBo, R3: std_logic;
            CBo, OV: out std_logic
      );
end flags;
architecture flags_logic of flags is
begin
    CBo <= iCBo;
       OV <= (A3 and B3 and not R3) or (not A3 and not B3 and R3);
end architecture;
```

Flags AU (*Flags* da *Aritmetic Unit*) – Calcula as *flags* de *carry*/*borrow* e de *overflow* com base no resultado, nos operandos de entrada e no *carry*/*borrow* de entrada.

3.4.13 mux4.VHD

```
library ieee;
use ieee.std_logic_1164.all;
entity mux4 is
    port (
                A, B: in std_logic_vector(3 downto 0);
                S: in std_logic;
          R: out std_logic_vector(3 downto 0)
       );
end mux4;
architecture mux4_logic of mux4 is
signal sel: std_logic_vector(3 downto 0);
begin
      sel(3 downto 0) <= (others => S);
      R <= (sel and A) or (not sel and B);
end architecture;
```

MUX – Escolhe com base no valor do OPc se irá mostrar como resultado o que veio da AU ou do Logic Module.

3.4.14 Flags_main.VHD

```
library ieee;
use ieee.std_logic_1164.all;
entity flags_main is
      port (
            R: std_logic_vector(3 downto 0);
            iOV, iCB, OP, CY: std_logic;

            BE, oGE, Z, oOV, oCB: out std_logic
      );
end flags_main;
architecture flags_main_logic of flags_main is
signal zero: std_logic;
begin
      zero <= not (R(0) or R(1) or R(2) or R(3));

      Z <= zero;
      oGE <= (not zero and not R(3) and not iOV) or (R(3) and iOV) or (zero and
not iOV);
      BE <= iCB or zero;
      oOV <= iOV;
      oCB <= (not OP and CY) or (OP and ICB);
end architecture;
```

Flags – Calcula diversas flags, como já foram referidas as condições usando o overflow e o carry/borrow da AU, usado o carry do Logic Module e usando o valor do OPf que faz a distinção entre as operações da unidade aritmética ou entre as de deslocamento/lógicas.

3.4.15 decoder.VHD

```
library ieee;
use ieee.std_logic_1164.all;
entity decoder is
      port (
            OP: std_logic_vector (2 downto 0);
            OPa, OPb, OPc, OPd, OPe, OPf: out std_logic
      );
end decoder;
architecture decoder_logic of decoder is
-- 0 0 0, OPd = 0 OPe = 0
-- 0 0 1, OPd = 0 OPe = 1
-- 0 1 0 , OPd = 1 OPe = 0
-- 0 1 1 , OPd = 1 OPe = 1
begin
      OPa <= OP(1);
      OPb <= OP(0);
      OPc <= OP(2);
      OPd <= OP(1);
      OPe <= OP(0);
      OPf <= OP(2);
end architecture;
```

Decoder – Um dos módulos mais importantes pois é o que faz a distinção com base no OP de 3 bits quais vai ser as operações a realizar. Como já foi falado ao longo da descrição de cada módulo estes valores têm todos as seguintes funções:

- Opa – Distingue as operações de adição/subtração com as de incremento/decremento;
- OPb – Distingue as operações de adição/incremento com as de subtração/decremento;
- OPc – Escolhe entre as operações da unidade aritmética ou entre as lógicas/deslocamento;
- OPd e OPe- São 2 bits que escolhem entre as 4 operações de lógica/deslocamento;
- OPf – Mesma função que o OPc mas que é usado de forma ao módulo *Flags* conseguir saber se deve usar o *carry*/*borrow* da unidade aritmética ou o *carry* do *Logic Module*.

3.5 Simulação

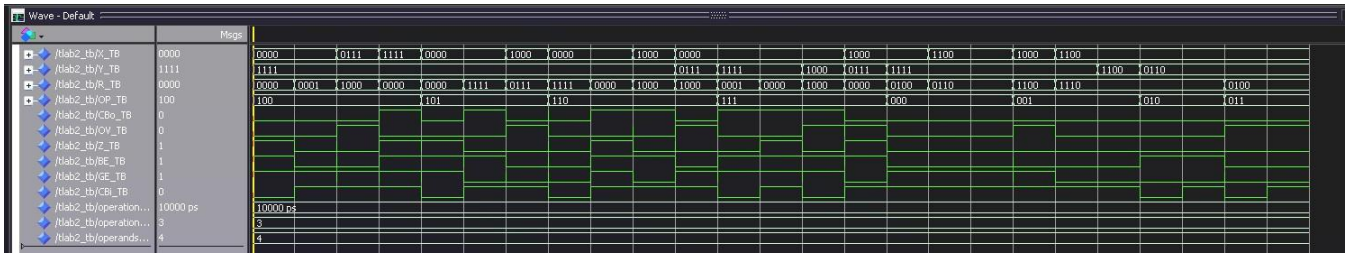


Figura 4 – Simulação do projeto

Além de ter sido executado o ficheiro de simulação fornecido pelo docente, como se pode ver pela Figura 4, os resultados foram verificados e depois também foi executado outro ficheiro que o mesmo tinha onde automaticamente colocava os valores e validava logo de forma a saber facilmente se o projeto estava correto.

4 Teste do Circuito

4.1 Atribuição de pinos

Entradas:

- X [3..0] – SW [3..0] | PIN_C12, PIN_D12, PIN_C11, PIN_C10;
- Y [3..0] – SW [7..4] | PIN_A14, PIN_A13, PIN_B12, PIN_A12;
- OP [1..0] – SW [9..8] | PIN_F15, PIN_B14;
- OP [2] – BTN [0] | PIN_B8;
- CBi – BTN [1] | PIN_A7;

Saídas:

- R [3..0] – LED [3..0] | PIN_B10, PIN_A10, PIN_A9, PIN_A8;
- CBo – LED [4] | PIN_D13;
- OV – LED [5] | PIN_C13;
- Z – LED [6] | PIN_E14;
- GE – LED [7] | PIN_D14;
- BE – LED [8] | PIN_A11;

4.2 Resultados do teste (tabela com entradas/saídas do circuito)

OP	CBi	X(2)	X(N)	X(Z)	Y(2)	Y(N)	Y(Z)	R(2)	R(N)	R(Z)	CBo	OV	Z	GE	BE
100'	**0**	**0000'**	0	0	**1111'**	15	-1	0000'	0	0	0	0	1	-	-
100'	**1**	**0000'**	0	0	**1111'**	15	-1'	0001'	1	+1'	0	0	0	-	-
100'	**1**	**0010'**	2	+2'	**0101'**	5	+5'	**1000'**	8	-8	0	0	0	-	-
100'	**1**	**1111'**	15	-1	**1010'**	10	-6	0000'	0	0	1	0	1	-	-
101'	**1**	**0000'**	0	0	**1111'**	15	-1'	1111'	15	-1	1	0	0	-	-
101'	**1**	**0001'**	1	+1'	**0101'**	5	+5'	0000'	0	0	0	0	1	-	-
101'	**1**	1000'	8	-8	**1010'**	10	-6	0111'	**7**	+7'	0	1	0	-	-
110'	**0**	0001'	1	+1'	**1110'**	14	-2'	1111'	15	**-1'**	0	0	0	-	-
110'	**1**	**0001'**	1	+1'	1111'	15	-1	0001'	1	+1'	1	0	**1**	-	-
110'	**1**	**1000'**	8	-8	1111'	15	-1	**1000'**	8	-8'	1	0	0	-	-
110'	**1**	**0010'**	2	+2'	**0101'**	5	+5'	1000'	8	-8'	0	1	0	-	-
111'	**0**	**0011'**	3	+3'	**0101'**	5'	+5'	1110'	14	-2'	1	0	0	0	1
111'	**0**	**1101'**	13	-3	1011'	11	-5	0010'	**2**	+2'	0	0	0	1	0
111'	**1**	**0110'**	6	+6'	**0110'**	6	+6'	1111'	15	-1'	1	0	0	0	1
111'	**1**	0111'	7	+7'	**0111'**	7	+7'	0000'	0	0	0	0	1	**1**	**1**
000'	**1**	**0001'**	1	+1'	**1100'**	12	-4	0000'	0	0	1	-	1	-	-
000'	**0**	**1101'**	13	-3	**1001'**	9	-7	0111'	7	+7'	1	-	0	-	-
001'	**0**	**1010'**	10	-6	**1111'**	15	-1	1101'	13	-3'	0	-	0	-	-
010'	**0**	**1100'**	12	-4	**0110'**	6	+6'	1110'	**14**	-2	-	-	0	-	-
011'	**1**	**1100**	12	-4	**0110'**	6	+6'	0100'	**4**	+4'	-	-	0	-	-

Figura 4 – Tabela de resultados teóricos

OP	CBi	X(2)	X(N)	X(Z)	Y(2)	Y(N)	Y(Z)	R(2)	R(N)	R(Z)	CBo	OV	Z	GE	BE
100'	**0**	**0000'**	0	0	**1111'**	15	-1	0000'	0	0	0	0	1	-	-
100'	**1**	**0000'**	0	0	**1111'**	15	-1'	0001'	1	+1'	0	0	0	-	-
100'	**1**	**0010'**	2	+2'	**0101'**	5	+5'	**1000'**	8	-8	0	0	0	-	-
100'	**1**	**1111'**	15	-1	**1010'**	10	-6	0000'	0	0	1	0	1	-	-
101'	**1**	**0000'**	0	0	**1111'**	15	-1'	1111'	15	-1	1	0	0	-	-
101'	**1**	**0001'**	1	+1'	**0101'**	5	+5'	0000'	0	0	0	0	1	-	-
101'	**1**	1000'	8	-8	**1010'**	10	-6	0111'	**7**	+7'	0	1	0	-	-
110'	**0**	0001'	1	+1'	**1110'**	14	-2'	1111'	15	**-1'**	0	0	0	-	-
110'	**1**	**0001'**	1	+1'	1111'	15	-1	0001'	1	+1'	1	0	**1**	-	-
110'	**1**	**1000'**	8	-8	1111'	15	-1	**1000'**	8	-8'	1	0	0	-	-
110'	**1**	**0010'**	2	+2'	**0101'**	5	+5'	1000'	8	-8'	0	1	0	-	-
111'	**0**	**0011'**	3	+3'	**0101'**	5'	+5'	1110'	14	-2'	1	0	0	0	1
111'	**0**	**1101'**	13	-3	1011'	11	-5	0010'	**2**	+2'	0	0	0	1	0
111'	**1**	**0110'**	6	+6'	**0110'**	6	+6'	1111'	15	-1'	1	0	0	0	1
111'	**1**	0111'	7	+7'	**0111'**	7	+7'	0000'	0	0	0	0	1	**1**	**1**
000'	**1**	**0001'**	1	+1'	**1100'**	12	-4	0000'	0	0	1	-	1	-	-
000'	**0**	**1101'**	13	-3	**1001'**	9	-7	0111'	7	+7'	1	-	0	-	-
001'	**0**	**1010'**	10	-6	**1111'**	15	-1	1101'	13	-3'	0	-	0	-	-
010'	**0**	**1100'**	12	-4	**0110'**	6	+6'	1110'	**14**	-2	-	-	0	-	-
011'	**1**	**1100**	12	-4	**0110'**	6	+6'	0100'	**4**	+4'	-	-	0	-	-

Figura 5 – Tabela com resultados experimentais

5 Conclusões

Este circuito consistiu na realização de uma unidade lógica e aritmética, também denominada por ALU (*Aritmetic and Logic Unit*). Esta unidade permite fazer operações de adição, subtração, incremento e decremento em operandos de 4 bits e também operações lógicas e de deslocamento como o OR, AND, LSR (*Logical Shift Right*) e ASR (*Aritmetic Shift Right*). Felizmente muitos dos componentes já estavam descritos em VHDL, só foi necessário fazer a lógica das operações de deslocamento, a lógica de um *Decoder* com uma entrada de 3 bits e um componente de *flags* mais completo. Para efetuar a lógica do *Decoder* recorremos aos mapas de *Karnaugh*, de modo a poder separar os 3 bits em vários bits para poderem ser usados em todo o circuito de modo a termos a operação pretendida de acordo com o enunciado.

Após isto, abordámos uma lógica de fora para dentro, onde fomos fazendo a descrição hardware dos vários blocos, mas sem qualquer lógica, permitindo apenas declarar as portas de entrada e saída e também fazer a ligação entre tudo. Por fim, fomos descrevendo a lógica em VHDL de cada módulo para ficar tudo funcional. Já com o projeto todo descrito, passámos à fase de simulação onde usámos um ficheiro de testes fornecido de forma a confirmar todos os valores, onde tivemos alguns erros no *LABc module* e nas *Flags* da ALU, mas fomos corrigindo aos poucos com o auxílio do docente.

Por fim, atribuímos os pinos necessários a todas as portas de acordo com o enunciado e testámos na placa FPGA onde foi validado pelo docente. Este projeto foi todo desenvolvido no software Intel Quartus Prime Lite 20.1 e a FPGA usada foi a MAX10 DE10 - Lite.