



PODER EXECUTIVO  
MINISTÉRIO DA EDUCAÇÃO  
UNIVERSIDADE FEDERAL DE RORAIMA  
DEPARTAMENTO DE CIÊNCIA A COMPUTAÇÃO

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

RELATÓRIO DO PROJETO: PROCESSADOR HOLOCRON BATTLE DROID 16 BITS

ALUNOS:

BRUNO CESAR DA SILVA CLAUDINO - 1201424425  
FABIO VITOR DE OLIVEIRA NORONHA - 2201424404

MARÇO DE 2017  
BOA VISTA/RORAIMA



PODER EXECUTIVO  
MINISTÉRIO DA EDUCAÇÃO  
UNIVERSIDADE FEDERAL DE RORAIMA  
DEPARTAMENTO DE CIÊNCIA A COMPUTAÇÃO

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

RELATÓRIO DO PROJETO: PROCESSADOR HOLOCRON BATTLE DROID 16 BITS

MARÇO DE 2017  
BOA VISTA/RORAIMA

## **Resumo**

O presente documento relata alguns pontos importantes do planejamento, implementação e testes do processador Holocron Battle Droid 16 bits. O mesmo foi desenvolvido em forma de trabalho final para a disciplina de arquitetura e organização de computadores (DCC301) na Universidade Federal de Roraima no semestre 2016.6. O Processador em questão foi baseado em arquiteturas conhecidas que serão descritas detalhadamente mais adiante, possui XXX componentes diferentes, num total de XXX componentes e 47 operações possíveis, o que abre um leque de muitas possibilidades para implementação de algoritmos que podem ser executados pelo Holocron Battle Droid 16 bits.

O processador em quest' ao foi implementado usando a linguagem de descrição de hardware VHDL. E os testes executados foram feitos utilizando-se waveforms geradas que simulam o comportamento do hardware descrito, estes mostram um pouco do funcionamento e performance do processador.

# Conteúdo

<b>Lista de Figuras</b>	<b>4</b>
<b>Lista de Tabelas</b>	<b>5</b>
<b>1 Especificações</b>	<b>6</b>
1.1 Plataforma de Desenvolvimento . . . . .	6
1.2 Conjunto de Instruções . . . . .	6
1.2.1 Instruções do tipo R . . . . .	6
1.2.2 Instruções do tipo J . . . . .	7
1.2.3 Instruções do tipo I . . . . .	7
1.2.4 Visão geral das instruções . . . . .	7
1.3 Descrição do Hardware . . . . .	8
1.3.1 Portas lógicas AND, OR e XOR . . . . .	8
1.3.2 Encurtador de Sinal . . . . .	9
1.3.3 Alongadores de Sinal . . . . .	10
1.3.4 Multiplexadores . . . . .	10
1.3.5 Demultiplexadores . . . . .	12
1.3.6 Deslocador para esquerda . . . . .	13
1.3.7 Deslocadores para direita . . . . .	13
1.3.8 Comparador aritmético . . . . .	14
1.3.9 Somador . . . . .	15
1.3.10 Multiplicador . . . . .	16
1.3.11 Divisor . . . . .	16
1.3.12 ULA . . . . .	17
1.3.13 Registrador D Flip-Flop . . . . .	19
1.3.14 PC . . . . .	19
1.3.15 Banco de registradores . . . . .	20
1.3.16 Memória de instruções . . . . .	21
1.3.17 Memória de dados . . . . .	22
1.3.18 Unidade de controle . . . . .	22
1.4 Datapath . . . . .	25
<b>2 Simulações e Testes</b>	<b>26</b>
2.1 Sequência de Fibonacci . . . . .	26
2.1.1 Código . . . . .	26
2.1.2 Waveform . . . . .	27
2.2 Números primos . . . . .	29
2.2.1 Código . . . . .	29
2.2.2 Waveform . . . . .	30
2.3 Bubble Sort . . . . .	30
2.3.1 Código . . . . .	30
2.3.2 Waveform . . . . .	32
<b>3 Considerações Finais</b>	<b>34</b>

## **Lista de Figuras**

1	Especificações do projeto . . . . .	6
2	Portas lógicas . . . . .	9
3	Vizualização das portas lógicas . . . . .	9
4	Encurtador de sinal de 16 para 1 bit . . . . .	10
5	Alongador de sinal de 4 para 16 bits . . . . .	10
6	Alongador de sinal de 11 para 16 bits . . . . .	10
7	Multiplexador com 2 bits de seleção para 1 bit . . . . .	11
8	Multiplexador com 1 bit de seleção para 4 bits . . . . .	11
9	Multiplexador com 1 bit de seleção para 16 bits . . . . .	11
10	Multiplexador com 3 bits de seleção para 16 bits . . . . .	12
11	Multiplexador com 4 bits de seleção para 16 bits . . . . .	12
12	Demultiplexador com 4 bits de seleção para 1 bit . . . . .	12
13	Demultiplexador com 4 bits de seleção para 16 bits . . . . .	13
14	Deslocador de bit para esquerda . . . . .	13
15	Deslocador lógico de bit para direita . . . . .	14
16	Deslocador aritmético de bit para direita . . . . .	14
17	Comparador Aritmético . . . . .	14
18	Vizualização do comparador . . . . .	15
19	Somador . . . . .	15
20	Vizualização do somador . . . . .	15
21	Multiplicador . . . . .	16
22	Vizualização do multiplicador . . . . .	16
23	Divisor . . . . .	16
24	Vizualização do divisor . . . . .	17
25	ULA . . . . .	17
26	Vizualização da ULA . . . . .	18
27	Registrador D Flip-Flop . . . . .	19
28	Vizualização do registrador D Flip-Flop . . . . .	19
29	Contador do Programa . . . . .	19
30	Vizualização do contador do Programa . . . . .	20
31	Banco de registradores . . . . .	20
32	Vizualização do banco de registradores . . . . .	21
33	Memória de instruções . . . . .	21
34	Memória de dados . . . . .	22
35	Vizualização da memória de dados . . . . .	22
36	Unidade de controle . . . . .	23
37	Vizualização da unidade de controle . . . . .	25
38	Datapath . . . . .	25
39	Waveform Fibonacci 30ns . . . . .	27
40	Waveform Fibonacci 60ns . . . . .	27
41	Waveform Fibonacci 90ns . . . . .	28
42	Waveform Fibonacci 120ns . . . . .	28
43	Waveform Fibonacci 320ns . . . . .	28
44	Waveform Números primos . . . . .	30
45	Waveform Algoritmo bubble sort 30ns . . . . .	32
46	Waveform Algoritmo bubble sort 60ns . . . . .	32
47	Waveform Algoritmo bubble sort 90ns . . . . .	33
48	Waveform Algoritmo bubble sort 120ns . . . . .	33

## **Lista de Tabelas**

1	Instruções do tipo R . . . . .	7
2	Instruções do tipo J . . . . .	7
3	Todos os Opcodes utilizados . . . . .	7
4	Operações da ULA . . . . .	17
5	Flags da unidade de controle . . . . .	23
6	Algoritmo que calcula a sequência de fibonacci . . . . .	26
7	Algoritmo que calcula números primos . . . . .	29
8	Algoritmo de ordenação <i>bubble sort</i> . . . . .	30

# 1 Especificações

O Holocron Battle Droid 16 bits foi desenvolvido para a disciplina de arquitetura e organização de computadores da Universidade Federal de Roraima no semestre 2016.2. O Processador é uniciclo, ou seja, executa uma unica instrução por vez a cada ciclo de clock e as mesmas possuem 16 bits. Ele foi baseado na arquitetura MIPS, uma arquitetura de microprocessadores RISC<sup>1</sup> desenvolvida pela MIPS Computer Systems<sup>2</sup>.

## 1.1 Plataforma de Desenvolvimento

Toda a descrição de Hardware foi feita utilizando a linguagem VHDL(*VHSIC<sup>3</sup> Hardware Description Language*) que é usada para facilitar o design (projeto e concepção) de circuitos digitais.

A IDE utilizada foi a Quartus Prime Lite Edition versão 16.1.0 Build 196 fazendo uma simulação do dispositivo 5CGTFD9E5F35C7 da família Cyclone V. A figura 1 mostra as especificações do projeto.

Figure 1: Especificações do projeto gerado no Quartus

Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	BattleDroid16bits
Top-level Entity Name	Holocron_BattleDroid16bits
Family	Cyclone V
Device	5CGTFD9E5F35C7
Timing Models	Final
Logic utilization (in ALMs)	355 / 113,560 (< 1 %)
Total registers	249
Total pins	536 / 616 (87 %)
Total virtual pins	0
Total block memory bits	0 / 12,492,800 (0 %)
Total DSP Blocks	1 / 342 (< 1 %)
Total HSSI RX PCSS	0 / 12 (0 %)
Total HSSI PMA RX Deserializers	0 / 12 (0 %)
Total HSSI TX PCSS	0 / 12 (0 %)
Total HSSI PMA TX Serializers	0 / 12 (0 %)
Total PLLs	0 / 20 (0 %)
Total DLLs	0 / 4 (0 %)

## 1.2 Conjunto de Instruções

O Processador Holocron Battle Droid 16 bits possui instruções do tipo R e J. O seu banco de registradores possui 16 registradores, são eles chamados \$zero, \$high, \$low, \$bool, \$s0, \$s1, \$s2, \$s3, \$s4, \$s5, \$s6, \$s7, \$t0, \$t1, \$t2 e \$t3. A seguir falarei sobre cada tipo de instrução.

### 1.2.1 Instruções do tipo R

Abrange instruções de carregamento e gravação de dados na memória primária e instruções baseadas em operações aritméticas, a divisão de bits está descrita na tabela 1 abaixo.

<sup>1</sup>conhecida também como **Computador com um conjunto reduzido de instruções** (*Reduced Instruction Set Computer*), executa um conjunto simples e pequeno de instruções que levam aproximadamente a mesma quantidade de tempo para serem executadas.

<sup>2</sup>Atualmente conhecida como MIPS Technologies, uma das mais conhecidas desenvolvedoras da arquitetura MIPS e uma pioneira no desenvolvimento das CPUs RISC.

<sup>3</sup>*Very High Speed Integrated Circuits*.

Table 1: Formato das instruções do tipo R			
<b>Opcode</b>	<b>Registrador 1</b>	<b>Registrador 2</b>	<b>Funct</b>
15-11 5 bits	10-7 4 bits	6-3 4 bits	2-0 3 bits

### 1.2.2 Instruções do tipo J

Abrange instruções onde é necessário fazer um salto condicional ou não entre os endereços, como um *if* ou um *goTo*, por exemplo. A divisão de bits segue abaixo na tabela 2.

Table 2: Formato das instruções do tipo J	
<b>Opcode</b>	<b>Endereço</b>
15-11 5 bits	10-0 11 bits

### 1.2.3 Instruções do tipo I

Devido à escassez de bits (16 apenas) em cada instrução, o processador Holocron Battle Droid 16 bits não conta com instruções do tipo I da maneira convencional como no MIPS. Os desvios condicionais foram divididos em duas partes: Faz-se a verificação da condição que determinará se o salto será realizado ou não, o resultado é guardado no registrador \$bool, que é um registrador especial para guardar resultados de comparações lógicas, em seguida, usa-se uma instrução do tipo J que recebe o endereço almejado (como mostrado na tabela 2) para saltar condicionalmente. A instrução definirá em quais condições o registrador \$bool deverá estar para que o salto seja efetuado, se ele deve ser *True* ou *False*.

### 1.2.4 Visão geral das instruções

O código de operação de cada instrução (*Opcode*) é igual a 5 bits, logo obtemos um total de 32 opcodes ( $2^5 = 32$ ) e o campo de modificação de função (*Funct*) funciona é composto por 3 bits, para cada Opcode existem 8 Funct's ( $2^3 = 8$ ). Em outras palavras, existem 256 possibilidades de operações diferentes ( $2^3 * 2^5 = 2^8 = 256$ ).

O Holocron Battle Droid 16 bits, entretanto, não implementa todas essas possibilidades, as funções disponíveis estão dispostas na tabela 3 abaixo:

Table 3: Todos os Opcodes implementados pelo Holocron Battle Droid 16 bits

<b>Opcode</b>	<b>Funct</b>	<b>Nome</b>	<b>Formato</b>	<b>Breve descrição</b>
00000	**0	ADD	R	Soma
00000	**1	SUB	R	Subtração
00001	**0	ADDI	R	Soma imediata
00001	**1	SUBI	R	Subtração imediata
00010	*00	AND	R	AND lógico
00010	*10	ANDI	R	AND lógico imediato
00010	*01	NAND	R	NAND lógico
00010	*11	NANDI	R	NAND lógico imediato
00011	*00	OR	R	OR lógico
00011	*10	ORI	R	OR lógico imediato
00011	*01	NOR	R	NOR lógico
00011	*11	NORI	R	NOR lógico imediato
00100	*00	XOR	R	XOR lógico

Continua na próxima página...

<b>Opcode</b>	<b>Funct</b>	<b>Nome</b>	<b>Formato</b>	<b>Breve descrição</b>
00100	*10	XORI	R	XOR lógico imediato
00100	*01	XNOR	R	XNOR lógico
00100	*11	XNORI	R	XNOR lógico imediato
00101	***	SRA	R	Shift Aritmético pra direita
00110	***	SRL	R	Shift Lógico pra direita
00111	***	SLLA	R	Shift para esquerda
01000	**0	EQU	R	Comparador igual
01000	**1	EQUI	R	Comparador igual imediato
01001	**0	DIF	R	Comparador diferente
01001	**1	DIFI	R	Comparador diferente imediato
01010	**0	SMA	R	Comparador menor que
01010	**1	SMAI	R	Comparador menor que imediato
01011	**0	SMEQ	R	Comparador menor que ou igual a
01011	**1	SMEQI	R	Comparador menor que ou igual a imediato
01100	**0	GRT	R	Comparador maior que
01100	**1	GRTI	R	Comparador maior que imediato
01101	**0	GREQ	R	Comparador maior que ou igual a
01101	**1	GREQI	R	Comparador maior que ou igual a imediato
01110	**0	MULT	R	Multiplicação
01110	**1	MULTI	R	Multiplicação imediata
01111	**0	DIV	R	Divisão
01111	**1	DIVI	R	Divisão imediata
10000	***	JR	I	Salto para endereço no registrador
10001	***	JIM	I	Salto para endereço imediato
10010	***	JRBT	I	Salto para endereço no registrador se \$bool = <i>True</i>
10011	***	JRBF	I	Salto para endereço no registrador se \$bool = <i>False</i>
10100	***	JIMBT	I	Salto para endereço imediato se \$bool = <i>True</i>
10101	***	JIMBF	I	Salto para endereço imediato se \$bool = <i>False</i>
10110	***	LDR	R	Carrega um dado do endereço no registrador
10111	***	LDI	R	Carrega um dado do endereço imediato
11000	***	STR	R	Grava um dado no endereço no registrador
11001	***	STI	R	Grava um dado no endereço imediado
11010	**0	MOVE	R	Move um dado para o registrador
11010	**1	MOVI	R	Move um dado imediato para o registrador

### 1.3 Descrição do Hardware

O processador Holocron Battle Droid 16 bits possui 27 componentes diferentes descritos em Hardware. Segue a descrição de cada um.

#### 1.3.1 Portas lógicas AND, OR e XOR

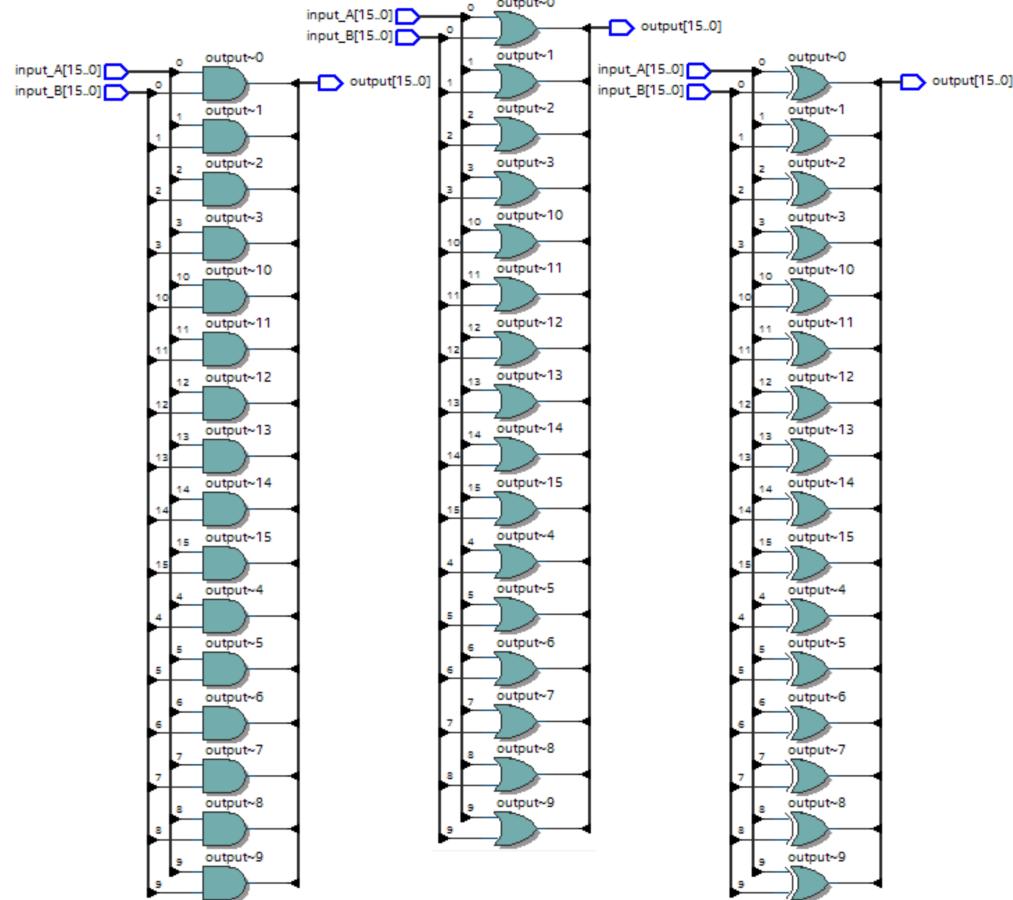
Foram implementadas as portas lógicas simples AND, OR e XOR adaptadas para entradas de 16 bits. Todas têm a descrição muito parecida, por esta razão a figura 2 mostra apenas a descrição da porta AND, o que difere uma da outra são o nome do componente e, obviamente, a implementação do comportamento das mesmas. A visualização, gerada automaticamente pelo Quartus, está na figura 3, respectivamente: AND, OR e XOR.

Figure 2: Declaração da porta lógica AND

```
entity LogicalAND_2x16 is
  port(
    input_A,
    input_B : in std_logic_vector(15 downto 0);
    output : out std_logic_vector(15 downto 0)
  );
end LogicalAND_2x16;
```

- **input\_A** e **input\_B**- Sinais de entrada em 16 bits.
- **output** - Sinal de saída após a comparação bit a bit.

Figure 3: Vizualização das portas lógicas AND, OR e XOR



### 1.3.2 Encurtador de Sinal

Este encurtador de sinal foi implementado especificamente para verificação de resultados lógicos, ele checa a existência de, pelo menos, um bit 1 para que a saída seja 1, caso contrário a saída é 0. Ele está descrito como mostra a figura 4.

Figure 4: Declaração de um encurtador de sinal de 16 para 1 bit

```
entity SignalShorter_16x1 is
  port(
    input : in std_logic_vector(15 downto 0);
    output : out std_logic
  );
end SignalShorter_16x1;
```

- **input** - Sinal de entrada em 16 bits.
- **output** - Sinal de saída em 1 bit.

### 1.3.3 Alongadores de Sinal

Existem dois alongadores de sinal, um que recebe 4 bits e outro que recebe 11 bits, ambos transformam essa entrada em uma saída de 16 bits. O alongador de 4 para 16 bits preenche os bits extras com 0's, já o de 11 para 16 bits, replica o bit menos significativo nas posições subsequentes. A declaração desses alongadores é mostrada, nas figuras 5 e 6.

Figure 5: Declaração de um alongador de sinal de 4 para 16 bits

```
entity SignalExtender_4x16 is
  port(
    input : in std_logic_vector(3 downto 0);
    output : out std_logic_vector(15 downto 0)
  );
end SignalExtender_4x16
```

Figure 6: Declaração de um alongador de sinal de 11 para 16 bits

```
entity SignalExtender_11x16 is
  port(
    input : in std_logic_vector(10 downto 0);
    output : out std_logic_vector(15 downto 0)
  );
end SignalExtender_11x16
```

- **input** - Sinal de entrada em 4 / 11 bits.
- **output** - Sinal de saída em 16 bits.

### 1.3.4 Multiplexadores

Durante a construção do Holocron Battle Droid 16 bits foram utilizados 5 tipos de multiplexadores. A declaração, bem como a descrição dos pinos de entrada e saída, seguem nas figuras 7, 8, 9, 10 e 11.

Figure 7: Declaração de um mutiplexador com 2 bits de seleção para 1 bit

```
entity Multiplexer_2x1 is
  port(
    selector : in  std_logic_vector(1 downto 0);
    input_A,
    input_B,
    input_C,
    input_D : in  std_logic;
    output   : out std_logic
  );
end Multiplexer_2x1;
```

- **selector** - Seletor de saída, um valor entre 00 e 11 que define qual das entradas será a saída.
- **input\_A** a **input\_D** - Entradas de dados de 1 bit.
- **output** - Sinal de saída contendo o valor de uma das entradas de acordo com o **selector**.

Figure 8: Declaração de um mutiplexador com 1 bit de seleção para 4 bits

```
entity Multiplexer_1x4 is
  port(
    selector : in  std_logic;
    input_A,
    input_B : in  std_logic_vector(3 downto 0);
    output   : out std_logic_vector(3 downto 0)
  );
end Multiplexer_1x4;
```

- **selector** - Seletor de saída que define qual das entradas será a saída.
- **input\_A** e **input\_B** - Entradas de dados de 4 bits.
- **output** - Sinal de saída contendo o valor de uma das entradas de acordo com o **selector**.

Figure 9: Declaração de um mutiplexador com 1 bit de seleção para 16 bits

```
entity Multiplexer_1x16 is
  port(
    selector : in  std_logic;
    input_A,
    input_B : in  std_logic_vector(15 downto 0);
    output   : out std_logic_vector(15 downto 0)
  );
end Multiplexer_1x16;
```

- **selector** - Seletor de saída que define qual das entradas será a saída.
- **input\_A** e **input\_B** - Entradas de dados de 16 bits.
- **output** - Sinal de saída contendo o valor de uma das entradas de acordo com o **selector**.

Figure 10: Declaração de um mutiplexador com 3 bits de seleção para 16 bits

```
entity Multiplexer_3x16 is
  port(
    selector : in std_logic_vector(2 downto 0);
    input_A, input_B, input_C, input_D,
    input_E, input_F, input_G, input_H : in std_logic_vector(15 downto 0);
    output : out std_logic_vector(15 downto 0)
  );
end Multiplexer_3x16;
```

- **selector** - Seletor de saída, com valores de 000 a 111, que define qual das entradas será a saída.
- **input\_A** a **input\_H** - Entradas de dados de 16 bits.
- **output** - Sinal de saída contendo o valor de uma das entradas de acordo com o **selector**.

Figure 11: Declaração de um mutiplexador com 4 bits de seleção para 16 bits

```
entity Multiplexer_4x16 is
  port(
    selector : in std_logic_vector(3 downto 0);
    input_A, input_B, input_C, input_D,
    input_E, input_F, input_G, input_H,
    input_I, input_J, input_K, input_L,
    input_M, input_N, input_O, input_P: in std_logic_vector(15 downto 0);
    output : out std_logic_vector(15 downto 0)
  );
end Multiplexer_4x16;
```

- **selector** - Seletor de saída, com valores de 0000 a 1111, que define qual das entradas será a saída.
- **input\_A** a **input\_P** - Entradas de dados de 16 bits.
- **output** - Sinal de saída contendo o valor de uma das entradas de acordo com o **selector**.

### 1.3.5 Demultiplexadores

Os demultiplexadores usados estão descritos nas figuras 12 e 13.

Figure 12: Declaração de um demultiplexador com 4 bits de seleção para 1 bit

```
entity Demultiplexer_4x1 is
  port(
    selector : in std_logic_vector(3 downto 0);
    input : in std_logic;
    output_A, output_B, output_C, output_D,
    output_E, output_F, output_G, output_H,
    output_I, output_J, output_K, output_L,
    output_M, output_N, output_O, output_P : out std_logic
  );
end Demultiplexer_4x1;
```

- **selector** - Seletor de saída, com valores de 0000 a 1111, que define para qual saída irá o valor de entrada.
- **input** - Sinal de entrada em 1 bit.

- **output\_A** a **output\_P** - Saidas de 1 bit. A saida que corresponde ao codigo recebido em **selector** recebe o valor contido em **input**, enquanto as outras recebem 0 por padrão.

Figure 13: Declaração de um demultiplexador com 4 bits de seleção para 16 bits

```
entity Demultiplexer_4x16 is
port(
    selector : in std_logic_vector( 3 downto 0);
    input : in std_logic_vector(15 downto 0);
    output_A, output_B, output_C, output_D,
    output_E, output_F, output_G, output_H,
    output_I, output_J, output_K, output_L,
    output_M, output_N, output_O, output_P : out std_logic_vector(15 downto 0)
);
end Demultiplexer_4x16;
```

- **selector** - Seletor de saída, com valores de 0000 a 1111, que define para qual saída irá o valor de entrada.
- **input** - Sinal de entrada em 16 bits.
- **output\_A** a **output\_P** - Saidas de 16 bits. A saida que corresponde ao codigo recebido em **selector** recebe o valor contido em **input**, enquanto as outras recebem 0 por padrão.

### 1.3.6 Deslocador de bit para esquerda

Como o nome do componente entrega, o componente desloca 1 bit para a esquerda adicionando um 0. Ele está descrito na figura 14.

Figure 14: Declaração de um deslocador de bit para esquerda

```
entity ArithmeticalLogicalLeftShifter_x16 is
port(
    input : in std_logic_vector(15 downto 0);
    output : out std_logic_vector(15 downto 0)
);
end ArithmeticalLogicalLeftShifter_x16;
```

- **input** - Sinal de entrada em 16 bits.
- **output** - Saída deslocada 1 bit para a esquerda.

### 1.3.7 Deslocadores de bit para direita

O processador Holocron Battle Droid 16 bits usa 2 deslocadores de bit para direita: Um lógico e um Aritmético. A principal diferença entre eles é que o lógico, ao deslocar o bit para a direita, põe sempre um 0 no lugar, enquanto o Aritmético duplica o último bit. Ambos estão descritos na figuras 15 e 16 respectivamente.

Figure 15: Declaração de um deslocador lógico de bit para direita

```
entity LogicalRightShifter_x16 is
  port(
    input  : in  std_logic_vector(15 downto 0);
    output : out std_logic_vector(15 downto 0)
  );
end LogicalRightShifter_x16;
```

- **input** - Sinal de entrada em 16 bits.
- **output** - Saída deslocada 1 bit para a direita.

Figure 16: Declaração de um deslocador aritmético de bit para direita

```
entity ArithmeticalRightShifter_x16 is
  port(
    input  : in  std_logic_vector(15 downto 0);
    output : out std_logic_vector(15 downto 0)
  );
end ArithmeticalRightShifter_x16;
```

- **input** - Sinal de entrada em 16 bits.
- **output** - Saída deslocada 1 bit para a direita.

### 1.3.8 Comparador Aritmético

O comparador está descrito na figura 17. Nele é possível realizar 6 operações, são elas: Igual, diferente, menor que, menor que ou igual, maior que, maior que ou igual. O resultado das comparações é sempre uma sequência com 16 0's ou 1's.

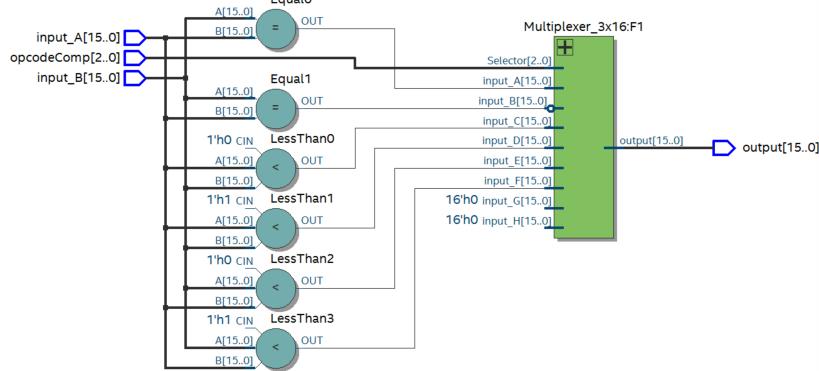
Figure 17: Declaração de um comparador Aritmético

```
entity ArithmeticalComparator_x16 is
  port(
    opcodeComp  : in  std_logic_vector(2 downto 0);
    input_A,
    input_B : in  std_logic_vector(15 downto 0);
    output      : out std_logic_vector(15 downto 0)
  );
end ArithmeticalComparator_x16;
```

- **opcodeComp** - Código que decide qual operação será realizada pelo comparador.
- **input\_A** e **input\_B** - Valores para serem comparados.
- **output** - Resultado da comparação.

A visualização simbólica, gerada pelo Quartus, é mostrada na figura 18. É possível ver um multiplexador com 3 bits de seleção para 16 bits, o mesmo foi previamente descrito na figura 10.

Figure 18: Vizualização do comparador gerada automaticamente pelo Quartus



### 1.3.9 Somador Aritmético

O somador está descrito na figura 19. Não possui nada de incomum, é apenas um somador simples.

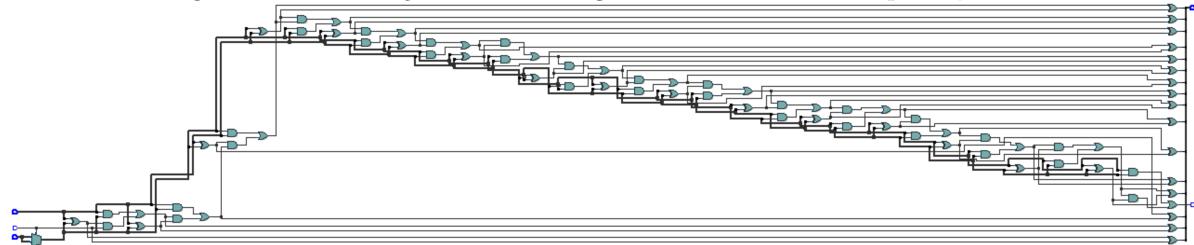
Figure 19: Declaração de um somador aritmético

```
entity Adder_2x16 is
  port (
    isSubtraction : in std_logic;
    input_A, input_B : in std_logic_vector(15 downto 0);
    carry_out      : out std_logic;
    output         : out std_logic_vector(15 downto 0)
  );
end Adder_2x16;
```

- **isSubtraction** - Modificador que indica se os valores serão somados ou subtraídos, no caso da subtração o valor de **input\_B** é negado e somado à **input\_A** com 1 (complemento de 2).
- **input\_A** e **input\_B** - Valores a serem somados.
- **carry\_out** - Indicador de *Overflow*.
- **output** - Resultado da soma.

A vizualização, gerada pelo Quartus, é mostrada na figura 20.

Figure 20: Vizualização do somador gerada automaticamente pelo Quartus



### 1.3.10 Multiplicador

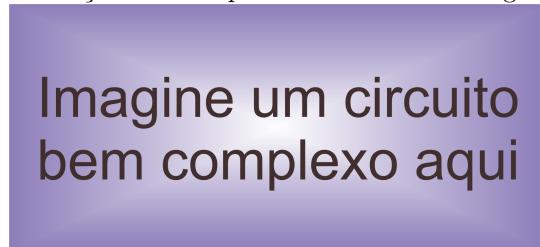
Descrito na imagem 21, o multiplicador foi implementado baseado no algoritmo de Booth<sup>4</sup>. A vizualização do componente é encontrada na figura 22.

Figure 21: Declaração de um multiplicador

```
entity Multiplier_2x16 is
  port(
    input_A, input_B      : in std_logic_vector(15 downto 0);
    outputLow, outputHigh : out std_logic_vector(15 downto 0);
    carryOut              : out std_logic
  );
end Multiplier_12x6;
```

- **input\_A** e **input\_B** - Valores que serão multiplicados.
- **outputLow** e **outputHigh** - Como os operandos possuem 16 bits - cada - o resultado será de 32 bits, dos quais 16 mais significativos ficam armazenados no **outputLow** e os 16 restantes no **outputHigh**.
- **carryOut** - Indicador de Overflow na multiplicação.

Figure 22: Vizualização do multiplicador utilizando o Algoritmo de Booth



### 1.3.11 Divisor

Descrito na figura 23, o divisor foi implementado baseado no algoritmo de divisão longa<sup>5</sup>. A vizualização do componente é encontrada na figura ??.

Figure 23: Declaração de um divisor

```
entity Divider_2x16 is
  port(
    input_A, input_B      : in std_logic_vector(15 downto 0);
    outputLow, outputHigh : out std_logic_vector(15 downto 0);
    carryOut              : out std_logic
  );
end Divider_2x16;
```

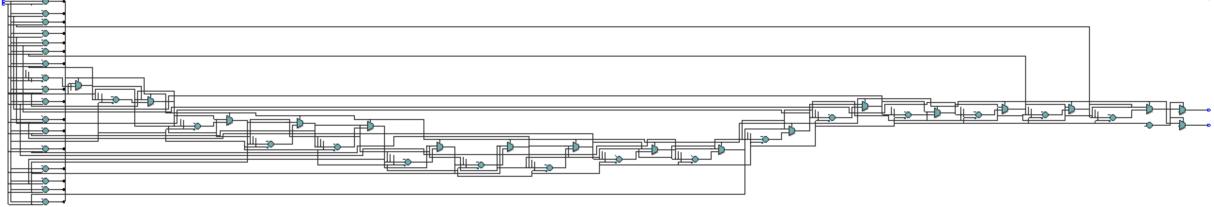
- **input\_A** - Valor do dividendo.

<sup>4</sup>O algoritmo de multiplicação de Booth é um algoritmo de multiplicação para números binários com sinal na notação complemento de dois. O algoritmo foi inventado por Andrew D. Booth em 1951 enquanto fazia pesquisas sobre Cristalografia no Colégio Birkbeck em Bloomsbury, Londres. Booth usava calculadoras que eram mais rápidas em deslocar do que em somar e criou o algoritmo para aumentar sua velocidade.

<sup>5</sup>Na aritmética, a divisão longa é um algoritmo de divisão padrão adequado para dividir números de vários dígitos que é simples o bastante para ser executado manualmente. Ele divide um problema de divisão em uma série de passos mais fáceis

- **input\_B** - Valor do divisor.
- **outputLow** - Resultado da divisão inteira.
- **outputHigh** - Resto da divisão.
- **carryOut** - Indicador de Overflow.

Figure 24: Vizualização do divisor utilizando o algoritmo de divisão longa



### 1.3.12 Unidade Lógica e Aritmética - ULA

A Unidade Lógica e Aritmética (ULA) desenvolvida está descrita na figura 25 e conta com 20 operações aritméticas, todas estão listadas na tabela 4.

Figure 25: Declaração da Unidade Lógica e Aritmética

```
entity ALU_x16 is
  port(
    opcode      : in  std_logic_vector(3 downto 0);
    negate      : in  std_logic;
    input_A, input_B : in  std_logic_vector(15 downto 0);
    output,
    overflowMultDiv  : out std_logic_vector(15 downto 0)
  );
end ALU_x16;
```

- **opcode** - seletor de operação.
- **negate** - Trabalha em conjunto com o **opcode**, indica se o resultado deverá ser negado: AND, OR, XOR ou NAND, NOR e XNOR, por exemplo.
- **input\_A** e **input\_B** - Operandos.
- **output** - Resultado da operação aritmética definida por **opcode**.
- **overflowMultDiv** - 16 bits extras quando a operação é, por exemplo, a multiplicação ou a divisão.

Table 4: Todas as operações aritméticas da ULA

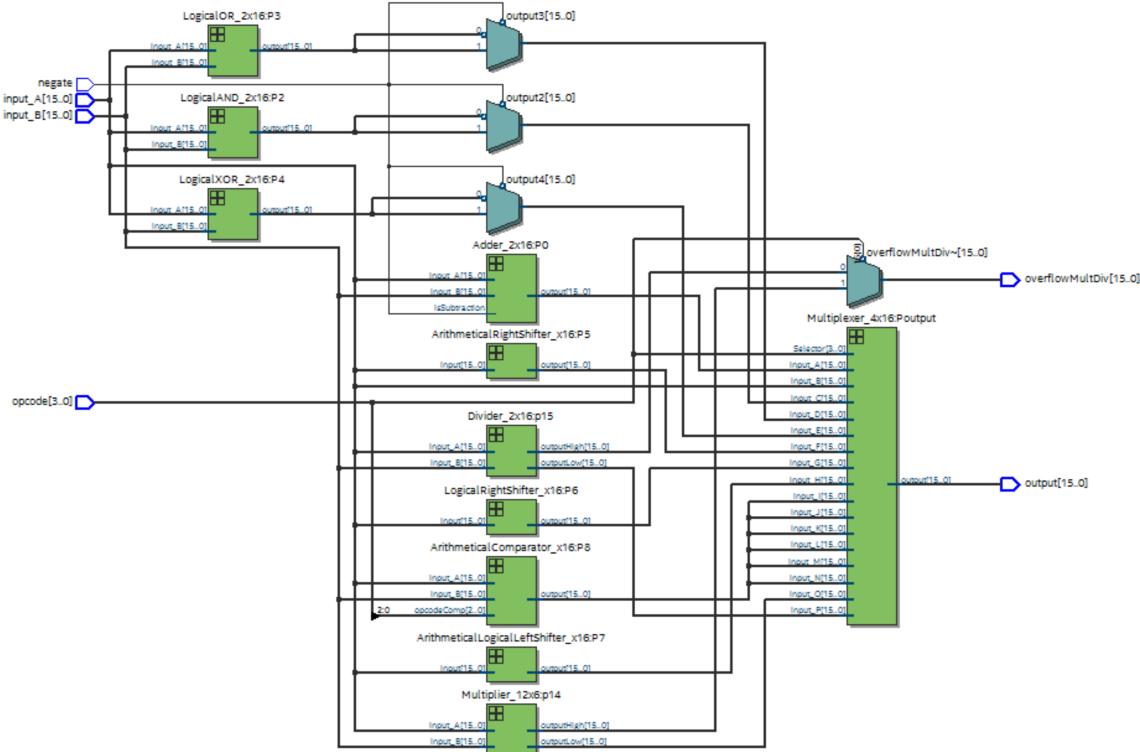
Opcode	Negate	Breve descrição
0000	0	Soma
0000	1	Subtração
0001	*	Retorna o Valor do segundo operando
0010	0	AND lógico

Continua na próxima página...

Opcode	Negate	Breve descrição
0010	1	NAND lógico
0011	0	OR lógico
0011	1	NOR lógico
0100	0	XOR lógico
0100	1	XNOR lógico
0101	*	deslocamento aritmético de bit pra direita
0110	*	deslocamento lógico de bit pra direita
0111	*	deslocamento de bit para esquerda
1000	*	Comparador igual
1001	*	Comparador diferente
1010	*	Comparador menor que
1011	*	Comparador menor que ou igual
1100	*	Comparador maior que
1101	*	Comparador maior que ou igual
1110	*	Multiplicação
1111	*	Divisão

A visualização simbólica da ULA está na figura 26, nela podemos ver que a mesma faz uso de alguns componentes descritos anteriormente, como um somador (figura 19), deslocador de bit para esquerda (figura 14), deslocador lógico de bit para direita (figura 15), deslocador aritmético de bit para direita (figura 16), comparador aritmético (figura 17), multiplicador (figura 21), divisor (figura 23), multiplexador de 4 bits de seleção para 16 bits (figura 11) e as portas lógicas AND, OR e XOR(figura 2).

Figure 26: Vizualização da Unidade Lógica e Aritmética



### 1.3.13 Registrador D Flip-Flop

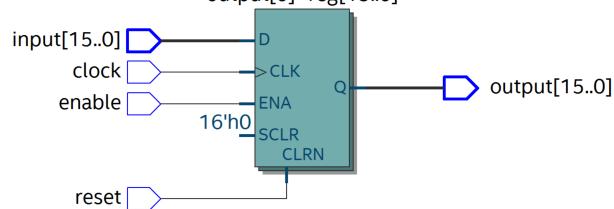
O Holocron Battle Droid 16 bits utiliza registradores flip-flop<sup>6</sup> do tipo D (*data* ou dado, pois armazena o bit de entrada) de 16 bits. O modelo de descrição é mostrado na figura 27 e a sua visualização na figura 28.

Figure 27: Declaração de um registrador D Flip-Flop

```
entity RegisterDFlipFlop_x16 is
  port (
    clock,
    reset,
    enable : in std_logic;
    input : in std_logic_vector(15 downto 0);
    output : out std_logic_vector(15 downto 0)
  );
end RegisterDFlipFlop_x16;
```

- **clock** - Sinal de clock do processador.
- **reset** - Flag de limpeza do registrador, quando 1 o valor armazenado se torna 0's.
- **enable** - Indicada se o valor contido em **input** deve ser armazenado.
- **input** - Valor de entrada.
- **output** - Valor armazenado no registrador.

Figure 28: Vizualização do registrador D Flip-Flop  
output[0]~reg[15..0]



### 1.3.14 Contador do Programa - PC

O Contador do Programa (PC) é apenas um registrador D Flip-Flop parecido com o da sessão que armazena o endereço da instrução atual, o que o difere é o fato do PC não conter **enable** e **reset**. Sua descrição e sua visualização seguem, respectivamente, nas figuras 29 e 30.

Figure 29: Declaração do contador do Programa

```
entity ProgramCounterRegister_x16 is
  port(
    clock : in std_logic;
    input : in std_logic_vector(15 downto 0);
    output: out std_logic_vector(15 downto 0)
  );
end ProgramCounterRegister_x16;
```

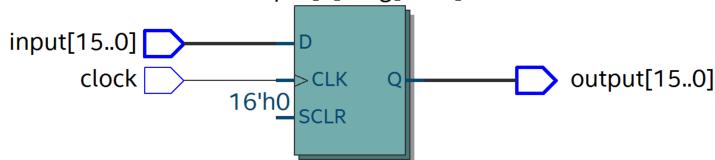
- **clock** - Sinal de clock do processador.

---

<sup>6</sup>O flip-flop ou multivibrador biestável é um circuito digital pulsado capaz de servir como uma memória de um bit.

- **input** - Valor de entrada.
- **output** - Valor armazenado no PC.

Figure 30: Vizualização do contador do Programa  
output[0]~reg[15..0]



### 1.3.15 Banco de registradores

O banco de registradores do Holocron Battle Droid 16 bits conta com 16 registradores, como citado na sessão **1.2 Conjunto de Instruções**. Ele foi declarado com 10 pinos de entrada e 2 de saída que podem ser vistos na figura 31 abaixo:

Figure 31: Declaração do banco de registradores

```

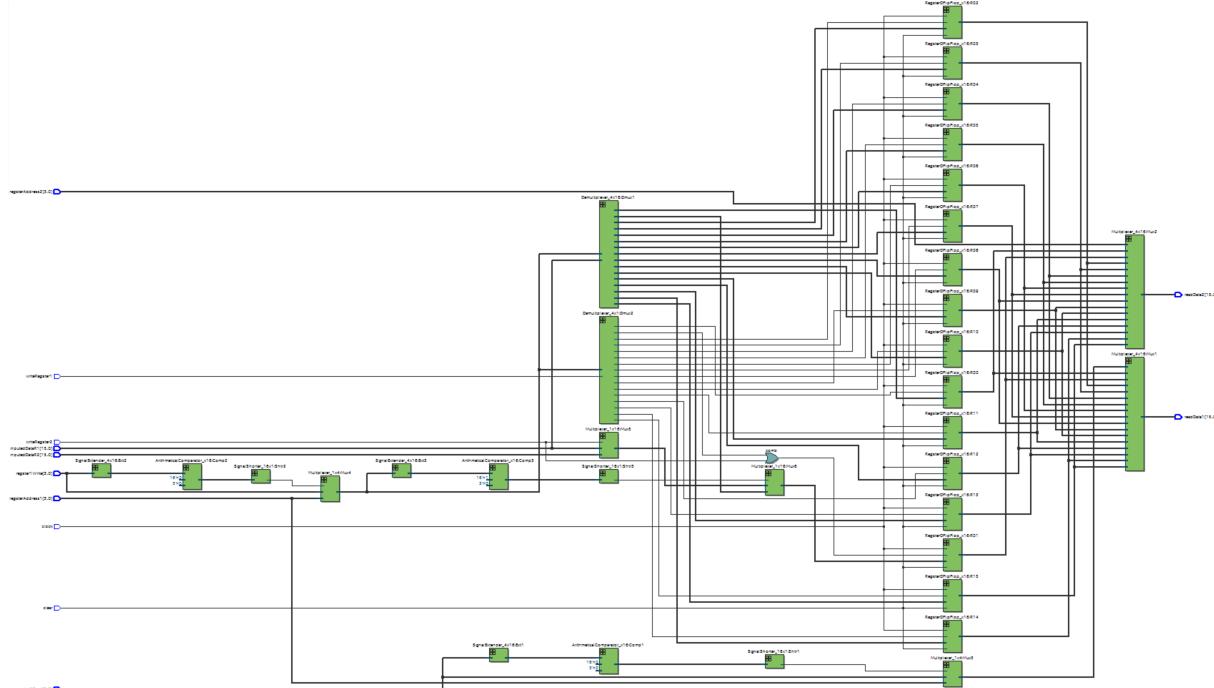
entity RegisterBank_4x16 is
  port(
    clock, clear, writeRegister1, writeRegister2: in std_logic;
    registerAddress1, registerAddress2,
    register1Read, register1Write : in std_logic_vector(3 downto 0);
    inputedDataR1, inputedDataR2 : in std_logic_vector(15 downto 0);
    readData1, readData2 : out std_logic_vector(15 downto 0)
  );
end RegisterBank_4x16;
  
```

- **clock** - Sinal de clock do processador.
- **clear** - Flag que indica que o conteúdo do registrador acessado será 0.
- **writeRegister1** - Flag para indicar que o valor em **inputedDataR1** será armazenado no registrador **register1Write**.
- **writeRegister2** - Flag para indicar que o valor em **inputedDataR2** será armazenado no registrador **\$high**.
- **registerAddress1** - Endereço do registrador 1.
- **registerAddress2** - Endereço do registrador 2.
- **register1Read** - Confirmação do endereço do registrador 1 para leitura. **readData1** assume o valor armazenado nesse endereço, quando esse valor é 0000, **readData1** assume o valor armazenado em **registerAddress1**.
- **register1Write** - Confirmação do endereço do registrador 1 para escrita. **inputedDataR1** é armazenado nesse endereço, quando esse valor é 0000, **inputedDataR1** é armazenado em **registerAddress1**.
- **inputedDataR1** - Dado de entrada para o registrador 1.
- **inputedDataR2** - Dado de entrada para o registrador 2.
- **readData1** - Valor lido do registrador 1.

- **readData2** - Valor lido do registrador 2.

É, provavelmente, o componente mais complexo do processador Holocron Battle Droid 16 bits. Ele é composto por 9 componentes diferentes, são eles: 3 extensores de sinal de 4 para 16 bits (figura 5), 3 comparadores aritméticos (figura 17), 3 encurtadores de sinal de 16 para 1 bit (figura 4), 2 multiplexadores com 4 bits de seleção para 16 bits (figura 11), 2 multiplexadores com 1 bit de seleção para 4 bits (figura 8), 2 multiplexadores com 1 bit de seleção para 16 bits (figura 9), 1 demultiplexador com 4 bits de seleção para 16 bits (figura 13), 1 demultiplexador com 4 bits de seleção para 1 bit (figura 12), 16 registradores do tipo D Flip-Flop (figura 27) e uma porta OR simples de 1 bit. Totalizando 34 componentes que podem ser vistos na figura 32 abaixo.

Figure 32: Vizualização do banco de registradores



### 1.3.16 Memória de instruções

A memória de instruções foi implementada como uma memória somente de leitura<sup>7</sup>. Sua descrição é bem simples e pode ser vista na imagem 33 logo abaixo:

Figure 33: Declaração da memória de instruções

```
entity ROMMemory_x16 is
  port(
    address : in std_logic_vector(15 downto 0);
    data : out std_logic_vector(15 downto 0)
  );
end ROMMemory_x16;
```

---

<sup>7</sup>A memória somente de leitura ou ROM (acrônimo em inglês de *Read-Only Memory*) é um tipo de memória que permite apenas a leitura, ou seja, as suas informações são gravadas pelo fabricante uma única vez e após isso não podem ser alteradas ou apagadas, somente acessadas.

- **address** - Endereço do qual se deseja ler.
- **data** - valor armazenado na memória no endereço **address**.

### 1.3.17 Memória de dados

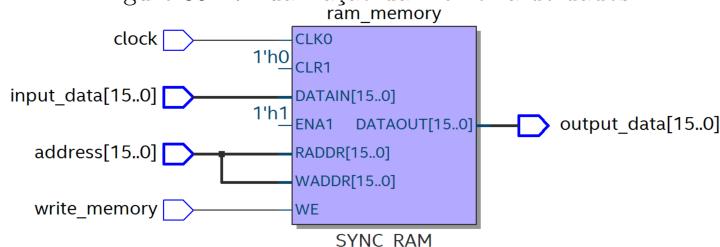
A memória de dados foi implementada como uma memória síncrona de acesso dinâmico aleatório<sup>8</sup>. Sua descrição é como na figura 34 e sua visualização está na figura 35.

Figure 34: Declaração da memória de dados

```
entity RAMMemory_x16 is
  port(
    clock      : in  std_logic;
    write_memory: in  std_logic;
    address    : in  std_logic_vector(15 downto 0);
    input_data  : in  std_logic_vector(15 downto 0);
    output_data : out std_logic_vector(15 downto 0)
  );
end entity RAMMemory_x16;
```

- **clock** - Sinal de clock do processador.
- **write\_memory** - Flag para definir se o valor **input\_data** deve ser escrito na memória no endereço **address**.
- **address** - Endereço da memória que está sendo acessado.
- **input\_data** - Dado a ser potencialmente escrito.
- **output\_data** - Dado escrito na memória no endereço **address**.

Figure 35: Vizualização da memória de dados



### 1.3.18 Unidade de controle

A unidade de controle é um componente fundamental para “orquestrar” o processador Holocron Battle Droid 16 bits. Ela recebe o **Opcode** e o campo **Funct** e gera flags para todos os outros componentes (especificamente 11 flags). A sua declaração é vista na figura 36.

---

<sup>8</sup> *Synchronous dynamic random access memory* (SDRAM) é uma memória de acesso dinâmico randômico (DRAM) que é sincronizada com o barramento do sistema, ou mais precisamente, com a transição de subida do clock. Permite uma operação mais justa pois o CPU saberá exatamente quando os dados estarão disponíveis.

Figure 36: Declaração da unidade de controle

```
entity ControlUnity_x16 is
  port(
    opcode          : in STD_LOGIC_VECTOR(4 downto 0);
    funct           : in STD_LOGIC_VECTOR(2 downto 0);
    ula, r1w, r1r  : out std_logic_vector(3 downto 0);
    wr1, wr2, ron,
    row, dvc, sri, mou : out std_logic;
    bool            : out STD_LOGIC_VECTOR(1 downto 0)
  );
end ControlUnity_x16;
```

- **opcode** - Código da operação desejada.
- **funct** - Modificador da função do **opcode**.
- **ula** - Código da operação que a ULA fará.
- **r1w** - Confirmação de escrita do dado no registrador 1 do banco de registradores.
- **r1r** - Confirmação de leitura do dado no registrador 1 do banco de registradores.
- **wr1** - Indica ao banco de registradores se o dado no registrador 1 deve ser gravado.
- **wr2** - Indica ao banco de registradores se o dado no registrador 2 deve ser gravado.
- **ron** - Indica para a ULA se o segundo operando é o valor contido no registrador ou o valor imediato como constante.
- **row** - Indica pra memória de dados se é para gravar ou ler do endereço que ela receber.
- **dvc** - Indica a intenção de um desvio no fluxo do programa.
- **sri** - Indica se, caso haja um salto condicional, o novo endereço deverá ser calculado a partir do valor contido em um registrador ou deve-se usar o valor imediato recebido.
- **mou** - indica se o valor que será escrito no registrador 1 é proveniente da memória de dados ou de uma operação da ULA.
- **bool** - Indica se deve ser checado o valor do registrador \$bool para efetuar o salto, se sim, indica também o tipo de checagem (*True* ou *False*).

Para cada Instrução que o Holocron Battle Droid 16 bits realiza, existem valores únicos para cada flag. A tabela 5, logo abaixo, revela o comportamento de cada uma das flags para cada instrução, porém, mostra apenas o código de cada instrução, a descrição do que ela faz foi mostrada na tabela 3. Os campos com \* indicam que o valor que determinada flag assume não tem relevância naquele contexto, podendo assim ser 0 ou 1 sem que isso mude algo (comumente chamado de textitdon't care).

Table 5: Flags da unidade de controle

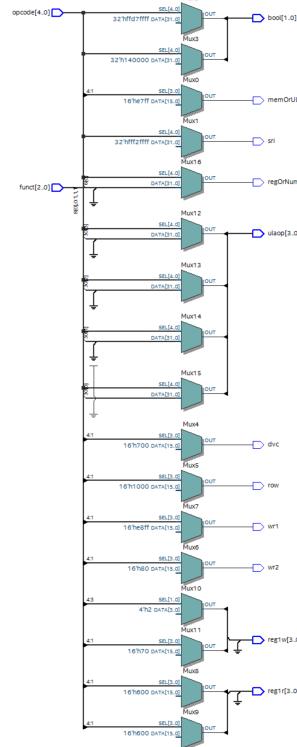
Nome	opcode	funct	ula	r1w	r1r	wr1	wr2	ron	row	dvc	sri	mou	bool
ADD	00000	**0	0000	0000	0000	1	0	0	0	0	*	1	**
SUB	00000	**1	0000	0000	0000	1	0	0	0	0	*	1	**
ADDI	00001	**0	0000	0000	0000	1	0	1	0	0	*	1	**
SUBI	00001	**1	0000	0000	0000	1	0	1	0	0	*	1	**
AND	00010	*00	0010	0000	0000	1	0	0	0	0	*	1	**

Continua na próxima página...

Nome	opcode	funct	ula	r1w	r1r	wr1	wr2	ron	row	dvc	sri	mou	bool
ANDI	00010	*10	0010	0000	0000	1	0	1	0	0	*	1	**
NAND	00010	*01	0010	0000	0000	1	0	0	0	0	*	1	**
NANDI	00010	*11	0010	0000	0000	1	0	1	0	0	*	1	**
OR	00011	*00	0011	0000	0000	1	0	0	0	0	*	1	**
ORI	00011	*10	0011	0000	0000	1	0	1	0	0	*	1	**
NOR	00011	*01	0011	0000	0000	1	0	0	0	0	*	1	**
NORI	00011	*11	0011	0000	0000	1	0	1	0	0	*	1	**
XOR	00100	*00	0100	0000	0000	1	0	0	0	0	*	1	**
XORI	00100	*10	0100	0000	0000	1	0	1	0	0	*	1	**
XNOR	00100	*01	0100	0000	0000	1	0	0	0	0	*	1	**
NORI	00100	*11	0100	0000	0000	1	0	1	0	0	*	1	**
SRA	00101	***	0101	0000	0000	1	0	0	0	0	*	1	**
SRL	00110	***	0110	0000	0000	1	0	0	0	0	*	1	**
SLLA	00111	***	0111	0000	0000	1	0	0	0	0	*	1	**
EQU	01000	**0	1000	0011	0000	1	0	0	0	0	*	1	**
EQUI	01000	**1	1000	0011	0000	1	0	1	0	0	*	1	**
DIF	01001	**0	1001	0011	0000	1	0	0	0	0	*	1	**
DIFI	01001	**1	1001	0011	0000	1	0	1	0	0	*	1	**
SMA	01010	**0	1010	0011	0000	1	0	0	0	0	*	1	**
SMAI	01010	**1	1010	0011	0000	1	0	1	0	0	*	1	**
SMEQ	01011	**0	1011	0011	0000	1	0	0	0	0	*	1	**
SMEQI	01011	**1	1011	0011	0000	1	0	1	0	0	*	1	**
GRT	01100	**0	1100	0011	0000	1	0	0	0	0	*	1	**
GRTI	01100	**1	1100	0011	0000	1	0	1	0	0	*	1	**
GREQ	01101	**0	1101	0011	0000	1	0	0	0	0	*	1	**
GREQI	01101	**1	1101	0011	0000	1	0	1	0	0	*	1	**
MULT	01110	**0	1110	0010	0000	1	1	0	0	0	*	1	**
MULTI	01110	**1	1110	0010	0000	1	1	1	0	0	*	1	**
DIV	01111	**0	1111	0010	0000	1	1	0	0	0	*	1	**
DIVI	01111	**1	1111	0010	0000	1	1	1	0	0	*	1	**
JR	10000	***	****	****	0000	0	0	*	0	1	0	*	01
JIM	10001	***	****	****	0000	0	0	*	0	1	1	*	01
JRBT	10010	***	****	****	0011	0	0	*	0	1	0	*	11
JRBF	10011	***	****	****	0011	0	0	*	0	1	0	*	00
JIMBT	10100	***	****	****	0011	0	0	*	0	1	1	*	11
JIMBF	10101	***	****	****	0011	0	0	*	0	1	1	*	00
LDR	10110	***	****	0000	0000	1	0	0	0	0	*	0	**
LDI	10111	***	****	0000	0000	1	0	1	0	0	*	0	**
STR	11000	***	****	****	0000	0	0	0	1	0	*	0	**
STI	11001	***	****	****	0000	0	0	1	1	0	*	0	**
MOVE	11010	**0	0001	0000	0000	1	0	1	0	0	*	1	**
MOVI	11010	**1	0001	0000	0000	1	0	1	1	0	*	1	**

Uma visualização simbólica do circuito resultante, gerada pelo Quartus, é mostrada na figura 37, nela podemos ver todos os pinos descrito na figura 36.

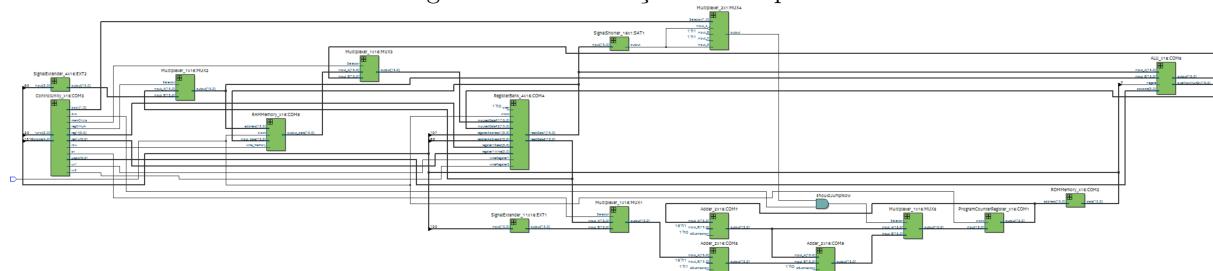
Figure 37: Vizualização da unidade de controle



## 1.4 Datapath

A coleção de unidades funcionais que executam operações de processamento de dados do Holocron Battle Droid 16 bits (*datapath*) é mostrada na figura 38 abaixo.

Figure 38: Vizualização do datapath



## 2 Simulações e Testes

Para testar o Holocron Battle Droid 16 bits foram escritos algoritmos simples, todos com o intuito de exemplificar algumas funções específicas do processador. Durante as simulações a frequência do clock foi definida como 1 GHz<sup>9</sup> (1 ciclo completo a cada nanosegundo) por questões de didática, para facilitar a visualização na waveform. A seguir veremos alguns desses algoritmos.

### 2.1 Sequência de Fibonacci

Na matemática, a sucessão de Fibonacci (também chamada de sequência de Fibonacci) é uma sequência de números inteiros, começando normalmente por 0 e 1, na qual, cada termo subsequente corresponde à soma dos dois anteriores<sup>10</sup>. A sequência recebeu o nome do matemático italiano Leonardo de Pisa, mais conhecido por Fibonacci, que descreveu, no ano de 1202, o crescimento de uma população de coelhos, a partir desta. Tal sequência já era no entanto, conhecida na antiguidade.

#### 2.1.1 Código

O código gerado calcula os 20 primeiros números dessa sequência. A tabela 6 mostra todo o código, bem como a construção de cada instrução.

Table 6: Algoritmo que calcula a sequência de fibonacci

Código	Opcode	R1	R2	Funct	Instrução
01 main: movi \$s0, 0;	11010	0100	0000	001	11010010000000001
02 movi \$s1, 1;	11010	0101	0001	001	1101001010001001
03 movi \$s3, 15;	11010	0111	1111	001	1101001111111001
04 addi \$s3, 5;	00001	0111	0101	000	0000101110101000
05 movi \$s4, 2;	11010	1000	0010	001	1101010000010001
06 smeq \$s3, \$zero;	01011	0111	0000	000	0101101110000000
07 jimb Endg;	10100		00000001111		10100000000001111
08 move \$s2, \$s0;	11010	0110	0100	001	1101001100100000
09 equ \$s3, \$s1;	01000	0111	0101	000	0100001110101000
10 jimb Endg;	10100		00000001100		10100000000001100
11 move \$s2, \$s1;	11010	0110	0101	001	1101001100101000
12 equi \$s3, 1;	01000	0111	0001	001	0100001110001001
13 jimb Endg;	10100		00000001001		10100000000001001
14 lpng: move \$t0, \$s0;	11010	1100	0100	000	1101011000100000
15 add \$t0, \$s1;	00000	1100	0101	000	0000011000101000
16 move \$s2, \$t0;	11010	0110	1100	000	1101001101100000
17 move \$s0, \$s1;	11010	0100	0101	000	1101001000101000
18 move \$s1, \$s2;	11010	0101	0110	000	1101001010110000
19 addi \$s4, 1;	00001	1000	0001	000	0000110000001000
20 equ \$s3, \$s4;	01000	0111	1000	000	0100001111000000
21 jimb lpng;	10100		11111111001		1010011111111001
22 Endg:					

Observe nas linhas 03 e 04, onde o registrador \$s3 recebe 15 e em seguida, é somado 5 a ele. Isso acontece por que o tamanho do endereço de cada valor (R1 e R2) é apenas 4 bits, dessa forma, o maior valor que

<sup>9</sup> 1 GHz =  $1 * 10^9 = 1.000.000.000$  ciclos por segundo

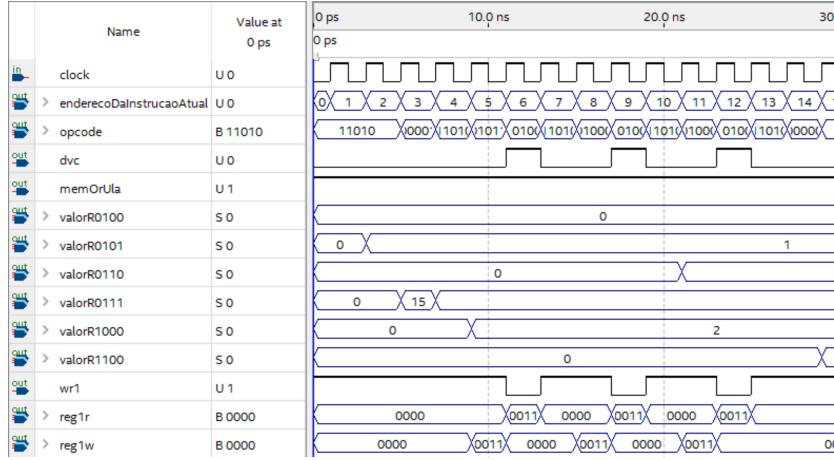
<sup>10</sup> Sequência de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, ...

poderá ser passado imediatamente é 1111, 15 em decimal. Um artifício que deve ser utilizado para sobrepor esse problema é somar os valores até que a quantidade desejada seja atingida.

### 2.1.2 Resultados na Waveform

A execução do algoritmo mostrado na tabela 6 é vista continuamente nas figuras 39, 40, 41, 42 e 43.

Figure 39: Waveform executando o algoritmo de Fibonacci nos primeiros 30ns



A figura 39 acima mostra a execução do algoritmo durante os 30 primeiros nanosegundos. O Processador Holocron Battle Droid 16 bits leva em consideração a borda de subida de 0 para 1 do clock para realizar a operação.

Figure 40: Waveform executando o algoritmo de Fibonacci entre 30 e 60ns

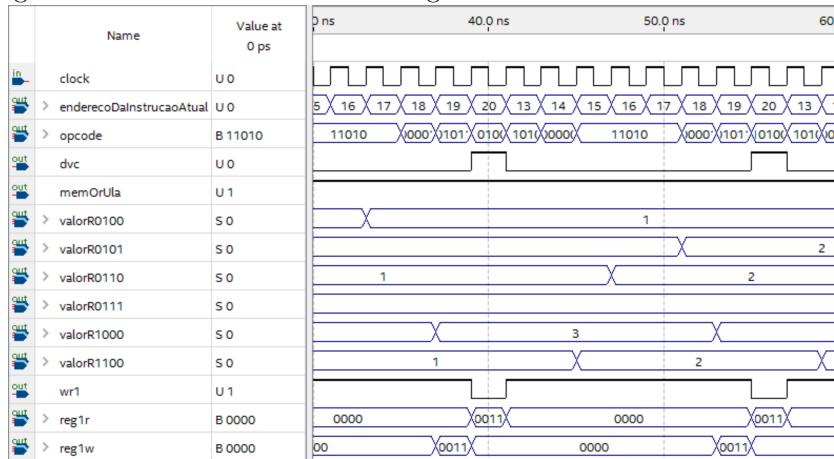
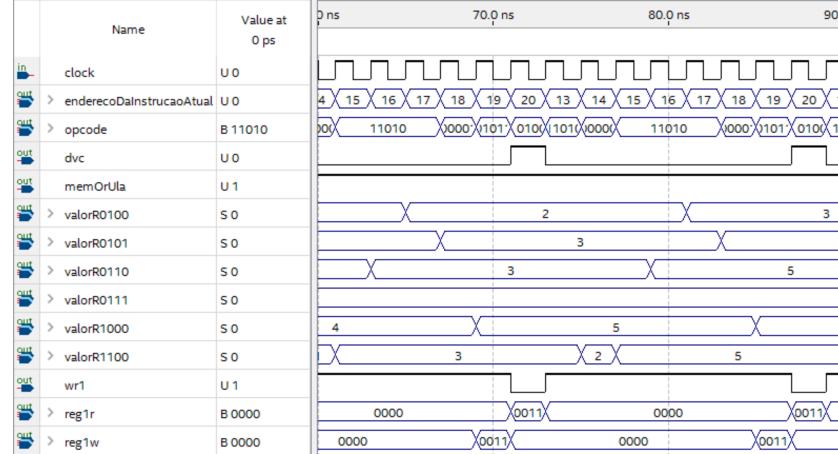


Figure 41: Waveform executando o algoritmo de Fibonacci entre 60 e 90ns



## 2.2 Números Primos

Números primos são os números naturais que têm apenas dois divisores diferentes: o 1 e ele mesmo<sup>11</sup>. Os números que têm mais de dois divisores são chamados números compostos. Exemplos:

- 2 tem apenas os divisores 1 e 2, portanto 2 é um número primo<sup>12</sup>.
- 17 tem apenas os divisores 1 e 17, portanto 17 é um número primo.
- 10 tem os divisores 1, 2, 5 e 10, portanto 10 não é um número primo.

### 2.2.1 Código

O código abaixo, mostrado na tabela 7, calcula todos os números primos entre 10 e 100, especificamente.

Table 7: Algoritmo que calcula números primos

Código	Opcode	R1	R2	Funct	Instrução
01 main: movi \$s0, 10;	11010	0100	1010	001	1101001001010001
02        movi \$t0, 12;	11010	1100	1100	001	1101011001100001
03        movi \$s5, 13;	11010	1001	1101	001	1101010011101001
04        add \$s5, \$t0;	00000	1001	1100	000	0000010011100000
05        add \$s5, \$s5;	00000	1001	1001	000	0000010011001000
06        add \$s5, \$s5;	00000	1001	1001	000	0000010011001000
07        move \$s1, \$s0;	11010	0101	0100	000	1101001010100000
08 lpn1: move \$s3, \$s1;	11010	0111	0101	000	1101001110101000
09        movi \$s4, 2;	11010	1000	0010	001	1101010000010001
10        div \$s3, \$s4;	01111	0111	1000	000	0111101111000000
11        move \$s2, \$low;	11010	0110	0010	000	1101001100010000
12 lpn2: equ \$s4, \$s2;	01000	1000	0110	000	0100010000110000
13        jimb <sub>t</sub> out1;	10100		000000000110		1010000000000110
14        div \$s3, \$s4;	01111	0111	1000	000	0111101111000000
15        equ \$high, \$zero;	01000	0001	0000	000	0100000010000000
16        jimb <sub>t</sub> out2;	10100		000000000100		1010000000000100
17        addi \$s4, 1;	00001	1000	0001	000	0000110000001000
18        jim lpn2;	10001		11111111010		1000111111111010
19 out1: move \$s7, \$s3;	11010	1011	0111	000	1101010110111000
20 out2: addi \$s1, 1;	00001	0101	0001	000	0000101010001000
21        equ \$s1, \$s5;	01000	0101	1001	000	0100001011001000
22        jimb <sub>f</sub> lpn1;	10101		11111110010		1010111111110010

Observe que da linha 02 a linha 06 faz-se, mais uma vez, uso do artifício citado no código que calcula a sequência de Fibonacci (tabela 6) para que o valor 100 possa ser passado para o registrador \$s5. Neste caso soma-se 12 à 13, em seguida duplica-se o resultado duas vezes ( $12 + 13 = 25$ ,  $25 + 25 = 50$ ,  $50 + 50 = 100$ ).

O algoritmo é bem simples: o primeiro laço de repetição percorre os valores de 10 a 100 enquanto o segundo laço tenta dividir cada um desses valores de 2 a  $n/2$ , onde  $n$  é o valor atual do iterador do laço 1. Caso o resto da divisão seja 0, quebra-se o laço e conclui-se que o número em questão não é primo (linhas 15 e 16). Quando um número é considerado primo, seu valor é armazenado no registrador \$s7 (linha 19) para facilitar nossa visualização na Waveform.

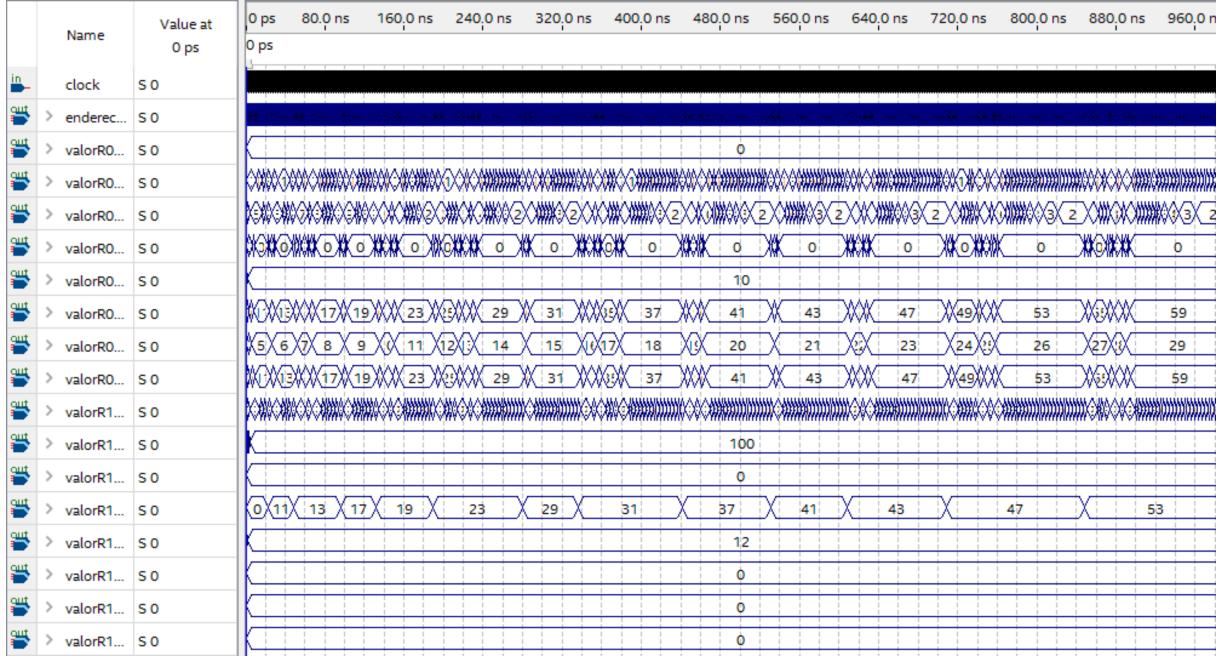
<sup>11</sup>1 não é um número primo, porque ele tem apenas um divisor que é ele mesmo.

<sup>12</sup>2 é o único número primo que é par

### 2.2.2 Resultados na Waveform

O algoritmo da tabela 7 sendo executado é mostrado na imagem 44.

Figure 44: Waveform executando o algoritmo que calcula números primos



Código	Opcode	R1	R2	Funct	Instrução
10 prt2: <b>movi</b> \$t1, 0;	11010	1101	0000	001	1101011010000001
11 lpn2: <b>equ</b> \$t1, \$s1;	01000	1101	0101	000	0100011010101000
12 <b>jimbt</b> bck2;	10100		00000101110		1010000000101110
13 <b>move</b> \$t3, \$s0;	11010	1111	0100	000	1101011110100000
14 <b>add</b> \$t3, \$t1;	00000	1111	1101	000	0000011111101000
15 <b>ldr</b> \$s7, \$t3;	10110	1011	1111	000	1011010111111000
16 <b>addi</b> \$t1, 1;	00001	1101	0001	000	0000111010001000
17 <b>jim</b> lpn2;	10001		11111111010		1000111111111010
18 main: <b>movi</b> \$s0, 0;	11010	0100	0000	001	1101001000000001
19 <b>movi</b> \$s1, 5;	11010	0101	0101	001	1101001010101001
20 <b>move</b> \$t0, \$s0;	11010	1100	0100	000	1101011000100000
21 <b>movi</b> \$t1, 5;	11010	1101	0101	001	1101011010101001
22 <b>str</b> \$t1, \$t0;	11000	1101	1100	000	1100011011100000
23 <b>addi</b> \$t0, 1;	00001	1100	0001	000	0000111000001000
24 <b>movi</b> \$t1, 3;	11010	1101	0011	001	1101011010011001
25 <b>str</b> \$t1, \$t0;	11000	1101	1100	000	1100011011100000
26 <b>addi</b> \$t0, 1;	00001	1100	0001	000	0000111000001000
27 <b>movi</b> \$t1, 9;	11010	1101	1001	001	1101011011001001
28 <b>str</b> \$t1, \$t0;	11000	1101	1100	000	1100011011100000
29 <b>addi</b> \$t0, 1;	00001	1100	0001	000	0000111000001000
30 <b>movi</b> \$t1, 7;	11010	1101	0111	001	1101011010111001
31 <b>str</b> \$t1, \$t0;	11000	1101	1100	000	1100011011100000
32 <b>addi</b> \$t0, 1;	00001	1100	0001	000	0000111000001000
33 <b>movi</b> \$t1, 1;	11010	1101	0001	001	1101011010001001
34 <b>str</b> \$t1, \$t0;	11000	1101	1100	000	1100011011100000
35 <b>jim</b> prt1;	10001		11111011111		100011111011111
36 bck1: <b>movi</b> \$t0, 0;	11010	1100	0000	001	1101011000000001
37 <b>move</b> \$t1, \$s1;	11010	1101	0101	000	1101011010101000
38 <b>subi</b> \$t1, 1;	00001	1101	0001	001	0000111010001001
39 lpn3: <b>movi</b> \$t2, 0;	11010	1110	0000	001	1101011100000001
40 <b>move</b> \$t3, \$t1;	11010	1111	1101	000	1101011111101000
41 <b>sub</b> \$t3, \$t0;	00000	1111	1100	001	0000011111100001
42 lpn4: <b>move</b> \$s2, \$s0;	11010	0110	0100	000	1101001100100000
43 <b>add</b> \$s2, \$t2;	00000	0110	1110	000	0000001101110000
44 <b>ldr</b> \$s7, \$s2;	10110	1011	0110	000	1011010110110000
45 <b>addi</b> \$s2, 1;	00001	0110	0001	000	0000101100001000
46 <b>ldr</b> \$s6, \$s2;	10110	1010	0110	000	1011010100110000
47 <b>sma</b> \$s6, \$s7;	01010	1010	1011	000	0101010101011000
48 <b>jimbf</b> nswp;	10101		000000000100		1010100000000100
49 <b>str</b> \$s7, \$s2;	11000	1011	0110	000	1100010110110000
50 <b>subi</b> \$s2, 1;	00001	0110	0001	001	0000101100001001
51 <b>str</b> \$s6, \$s2;	11000	1010	0110	000	1100010100110000
52 nswp: <b>addi</b> \$t2, 1;	00001	1110	0001	000	0000111100001000
53 <b>sma</b> \$t2, \$t3;	01010	1110	1111	000	0101011101111000
54 <b>jimbt</b> lpn4;	10100		111111110100		1010011111110100
55 <b>addi</b> \$t0, 1;	00001	1100	0001	000	0000111000001000
56 <b>sma</b> \$t0, \$t1;	01010	1100	1101	000	0101011001101000
57 <b>jimbt</b> lpn3;	10100		11111101110		1010011111101110

Continua na próxima página...

Código		Opcode	R1	R2	Funct	Instrução
58	jim      prt2;	10001		11111010000		100011111010000
59	bck2:					

O código acima pode ser dividido em 3 partes: A primeira parte, da linha 02 a 17, é composta pelas funções de impressão, servem para imprimir o vetor antes e depois da ordenação. A segunda parte, da linha 18 a 34, é onde o vetor é populado, isto é, preenchido. Nesse exemplo, especificamente, o vetor foi definido tendo tamanho 5 (linha 19) e foram inseridos os valores 5, 3, 9, 7 e 1 arbitrariamente. A terceira e última parte, da linha 36 a 57, é o algoritmo de ordenação propriamente dito.

### 2.3.2 Resultados na Waveform

Figure 45: Waveform executando o algoritmo de ordenação *bubble sort* no intervalo de 0 a 30ns

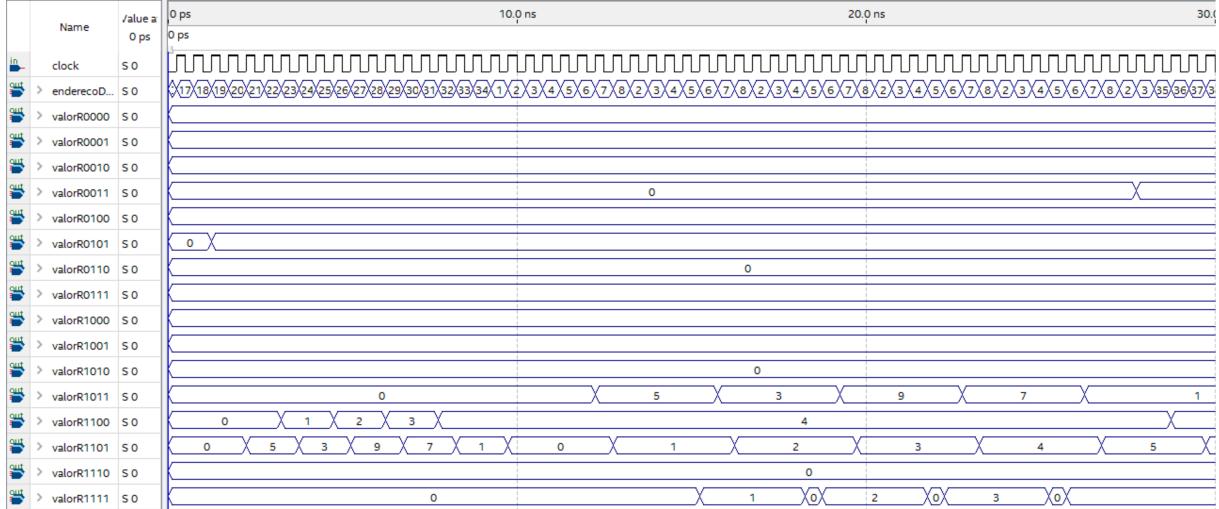


Figure 46: Waveform executando o algoritmo de ordenação *bubble sort* no intervalo de 30 a 60ns

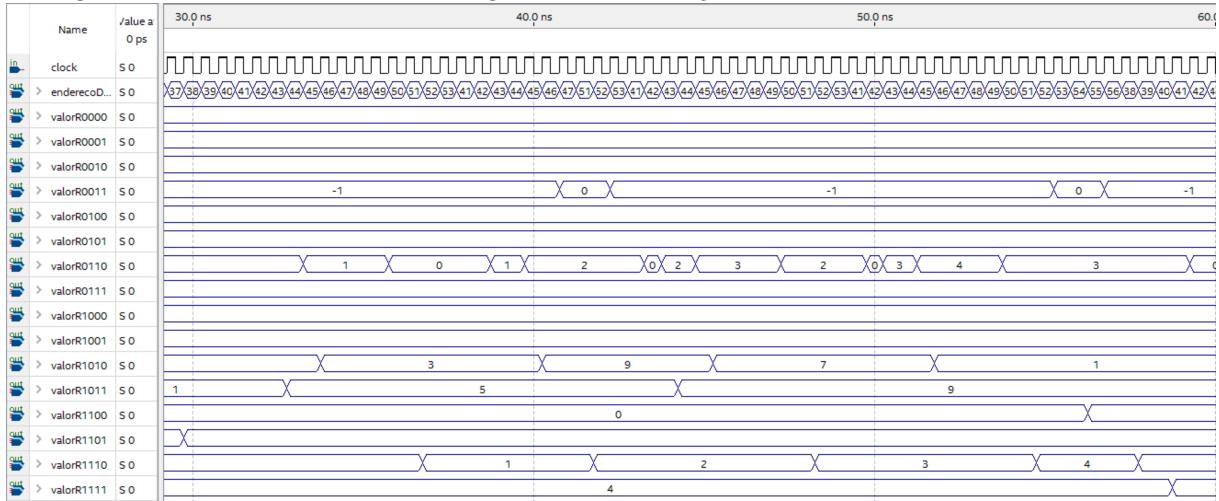


Figure 47: Waveform executando o algoritmo de ordenação *bubble sort* no intervalo de 60 a 90ns

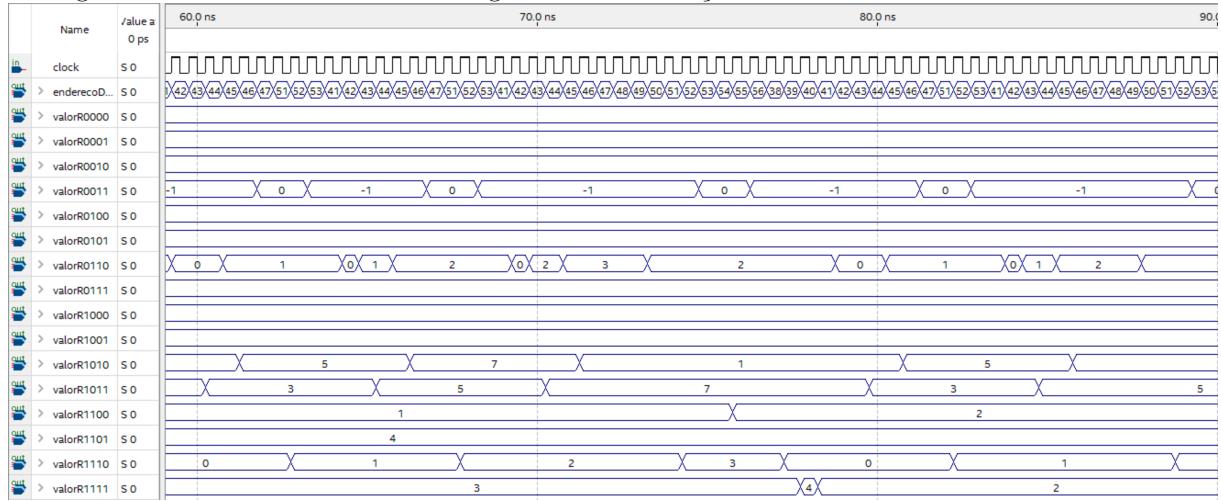
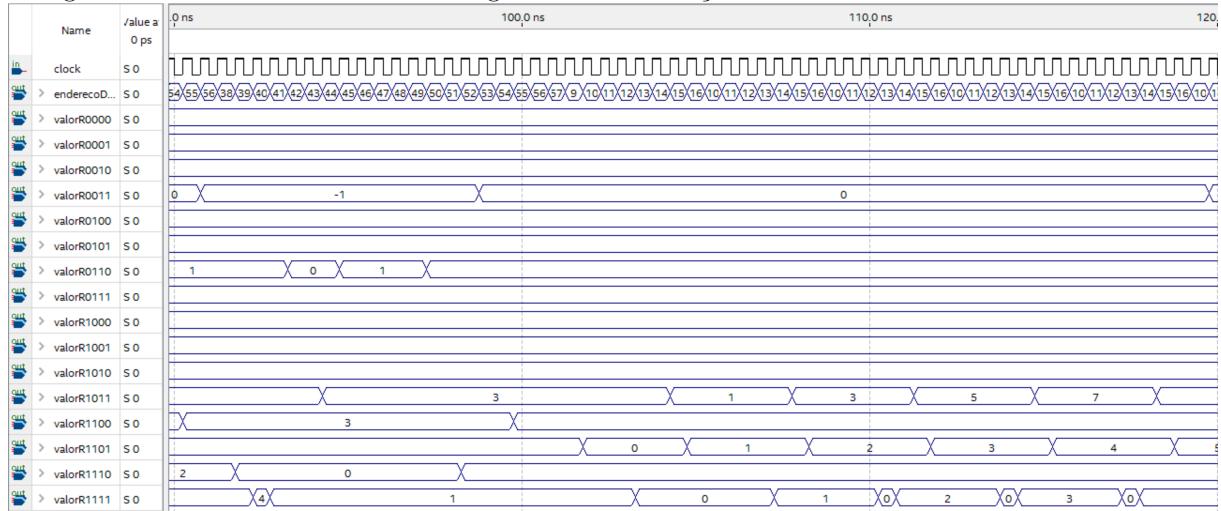


Figure 48: Waveform executando o algoritmo de ordenação *bubble sort* no intervalo de 90 a 120ns



### 3 Considerações Finais

O objetivo desse trabalho foi relatar todo o processo de criação do processador Holocron Battle Droid 16 bits.

No planejamento o relatório não cita, mas foi utilizada a ferramenta Logisim que é uma ferramenta educacional para a concepção e a simulação digital de circuitos lógicos, com uma interface simples e com ferramentas para simular circuitos a medida em que são construídos, é simples o bastante para facilitar a aprendizagem dos conceitos mais básicos relacionados aos circuitos lógicos. Esta foi muito útil para a concepção vizual e estrutural do processador. Os códigos escritos assim o foram no editor de texto notepad++, um editor de código fonte gratuito que suporta vários idiomas em primeira instância em MIPS32 e executados no QtSpim que é um simulador autônomo que executa programas MIPS32, ele lê e executa programas de linguagem de montagem escritos para este processador e também fornece um depurador simples e um conjunto mínimo de serviços do sistema operacional.

O nome Holocron Battle Droid 16 bits foi inspirado simbolicamente na franquia *Star Wars Expanded*: Holocron é uma abreviação de crônica holográfica (*Holographic Chronic*), ele foi um dispositivo cristalino lapidado orgânico, em que os Jedi antigos armazenavam quantidades fenomenais de dados, bem como o guardião do holocron. Os Sith também tinham sua própria forma de tecnologia de holocron, e parecem anteriores aos Jedi na utilização da tecnologia em pelo menos três mil anos. O termo *Droids* refere-se à um tipo de robô inteligente e *Battle Droid*, também chamados de droides de combate, foi um tipo de droid projetado para combate. 16 bits devido ao tamanho de cada instrução que o processador recebe.

Foi, sem dúvidas, um trabalho muito proveitoso. Não apenas no conhecimento do funcionamento de um processador uniciclo, mas em aperfeiçoamento moral. A própria franquia *Star Wars Expanded* nos ensina que “Grandes líderes inspiram a grandeza em outras pessoas” e, por sorte, estou cercado de grandes líderes aqui na Universidade Federal de Roraima que não se cansam de inspirar grandeza nas pessoas. Terminei citando um de meus professores que certa vez disse: “Não olha muito pro lado, faça o seu melhor e fique tranquilo em relação à isso.”.