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

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

RELATÓRIO DO PROJETO: PROCESSADOR KRAKEN 2.0

ALUNAS:
Ewelly Fabiane Cunha de Sousa – 2016011439
Débora Bianca Taveira de Moura – 2016011555

DEZEMBRO - 2018
BOA VISTA - RORAIMA

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

ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

RELATÓRIO DO PROJETO: PROCESSADOR KRAKEN 2.0

DEZEMBRO - 2018
BOA VISTA - RORAIMA

RESUMO

Este relatório aborda o projeto, a implementação do processador de 16 bits **Kraken 2.0**, desenvolvido em forma de projeto final pelas alunas Débora Bianca e Ewelly Fabiane na disciplina de arquitetura e organização de computadores ofertada pela Universidade Federal de Roraima aos alunos do curso de Ciência da Computação.

Neste documento estará descrita as especificações do processador, suas instruções, componentes, datapath, simulações e testes. O nome do processador faz analogia ao Kraken da mitologia nórdica, sendo 8 de seus tentáculos referência aos 8 bits do processador em sua edição anterior. Os seus tentáculos têm a intenção de capturar alimento, assim como nosso processador captura suas oito instruções, mergulhado em um mar de binários. Agora, apresentado como Kraken 2.0, por ser sua segunda versão e agora possuindo 16 bits.

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

1 Especificações

O processador Kraken em sua versão 2.0 agora possui 16 bits, foi desenvolvido como projeto final para a disciplina de arquitetura e organização de computadores na Universidade Federal de Roraima (UFRR) no semestre 2018.2. O processador é uniciclo, executando uma única instrução por vez a cada ciclo de clock, baseia-se na arquitetura MIPS.

1.1 Plataforma de desenvolvimento

Para a implementação do processador Kraken 2.0 em VHDL (VHSIC Hardware Description Language) a IDE utilizada foi o Quartus Prime Lite Edition em sua versão 17.0.

Figura 1: Especificações do projeto gerado no Quartus

Flow Status	Successful - Tue Dec 04 23:25:09 2018
Quartus Prime Version	17.1.0 Build 590 10/25/2017 SJ Lite Edition
Revision Name	KrakenProcessor
Top-level Entity Name	ULA_16
Family	Cyclone V
Device	5CGXFC7C7F23C8
Timing Models	Final
Logic utilization (in ALMs)	N/A
Total registers	0
Total pins	52
Total virtual pins	0
Total block memory bits	0
Total DSP Blocks	0
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0

1.2 Conjunto de instruções

O processador Kraken 2.0 possui 16 registradores: $0, $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7, $s8, $s9, $s10, $s11, $s12, $s13, $s14. Assim como três formatos de instruções, do **tipo R, I** e **J,** de 16 bits cada.

1.2.1 Formato de instruções do tipo R

Este formato aborda instruções baseadas em lógica e aritmética.

Tabela 1: Formato das instruções tipo R

Opcode	rs	rd	Funct
4 bits	4 bits	4 bits	4 bits
0 - 3	4 - 7	8 - 11	12 - 15

1.2.2 Formato de instruções do tipo I

Este formato aborda instruções baseadas no carregamento, tranferência e gravação de dados na memória primária. Onde o campo **endereço** desloca o conteúdo de **rs** como base.

Tabela 2: Formato das instruções tipo I

Opcode	rs	rt	Endereço
4 bits	4 bits	4 bits	4 bits
0 - 3	4 - 7	8 - 11	12 - 15

1.2.3 Formato de instruções do tipo J

Este formato aborda instruções baseadas em desvios condicionais.

Tabela 3: Formato das instruções tipo J

Opcode	Endereço
4 bits	12 bits
0 - 3	4 - 15

1.2.4 Visão geral das instruções do Processador Kraken 2.0:

O número de operação de cada instrução é igual a 4 bits, sendo assim obtemos um total de 16 bits para o **Opcode** (2^4 = 16), além do campo **Funct** que é composto por 4 bits (2^4 = 16), no processador Kraken 2.0, dos 16 bits de opcode, 15 são distribuídos entre operações do tipo I e J, e 1 bit é voltado para operações do tipo R, que com auxílio do campo funct aumenta o número de operações do tipo R para 16.

Tabela 4 – Tabela da lista de Opcodes utilizadas pelo processador Kraken 2.0

Opcode	Funct	Nome	Formato	Breve descrição
0000	0000	**ADD**	R	Soma
0000	0001	**SUB**	R	Subtração
0000	0010	**MULT**	R	Multiplicação
0000	0011	**AND**	R	Porta lógica AND
0000	0100	**OR**	R	Porta lógica OR
0000	0101	**XOR**	R	Porta lógica XOR
0000	0110	**NOR**	R	Porta lógica NOR
0000	0111	**NAND**	R	Porta lógica NAND
0000	1000	**NOR**	R	Porta lógica XOR
0000	1001	**XNOR**	R	Porta lógica XNOR
0000	1010	**COMPARADOR**	R	Porta lógica que compara se o primeiro valor é menor que o segundo
0000	1011	**MOVE**	R	Move um dado para o registrador
0000	1100	**SLT**	R	Set if less than
0000	1101	**SRA**	R	Shift aritmético para direita
0000	1111	**BoothMultiplier**	R	Realiza a multiplicação utilizando o algoritmo de Booth
0001	x	**LW**	I	Lê uma palavra da memória
0010	x	**SW**	I	Guarda uma palavra na memória
0011	x	**BEQ**	I	branch on equal
0100	x	**BNE**	I	branch on not equal
0101	x	**MOVI**	I	Move um dado imediato para o registrador
0110	x	**SLTI**	I	Set if less than immediate
0111	x	**ADDI**	I	Soma imediata
1000	x	**ADDIU**	I	Soma imediata sem overflow
1001	x	**J**	J	Salto
1010	x	**JR**	J	Jump register
1011	x	**JAL**	J	Jump and link

1.3 Descrição do Hardware

O processador Kraken 2.0 de 16 bits possui 18 componentes descritos em hardware, os subtópicos a seguir irão descrevê-los detalhadamente (suas funcionalidades, valores de entrada e saída).

1.3.1 And, Or, Not

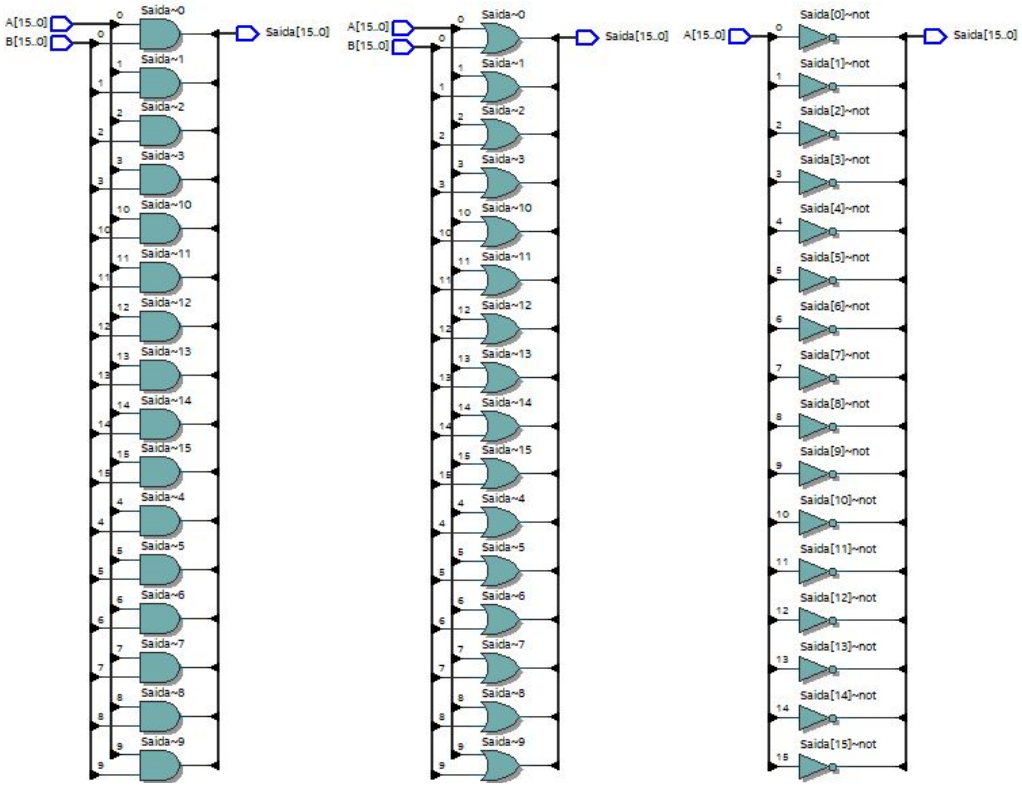
Os componentes And, Or e Not apesar de serem diferentes em relação à sua funcionalidade, possuem grandes semelhanças em seus valores de entrada e saída, por isso foram agrupados neste sub-tópico.

Figura 2: Declaração do AND

```
Entity And_16 is
        Port(
                A, B : IN STD_LOGIC_VECTOR(15 downto 0);
                Saida : OUT STD_LOGIC_VECTOR(15 downto 0)
                );
        End And_16;
```

- IN: A, B; tipo in STD_LOGIC_VECTOR; recebem valores em binário.
- OUT: Saida; tipo out STD_LOGIC_VECTOR.

Figura 3: RTL Viewer do AND, Or e Not (respectivamente)



1.3.2 Xor

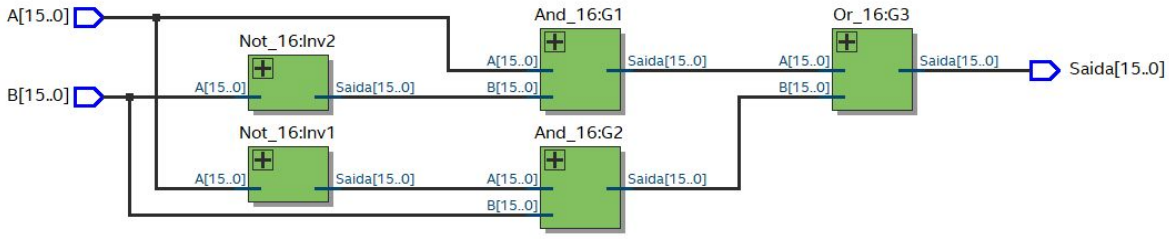
O XOR é um componente que verifica se apenas uma das alternativas dadas é verdadeira, apesar de sua entrada e saída ser semelhante aos componentes anteriores, a apresentação de sua RTL Viewer é diferente.

Figura 4: Declaração do XOR

```
Entity Xor_16 is
        Port(
                A, B : IN STD_LOGIC_VECTOR(15 downto 0);
                Saida : OUT STD_LOGIC_VECTOR(15 downto 0)
                );
        End Xor_16;
```

- IN: A, B; tipo in STD_LOGIC_VECTOR; recebem valores em binário.
- OUT: Saida; tipo out STD_LOGIC.

Figura 5: RTL Viewer do XOR



1.3.3 Right Shift e Left Shift

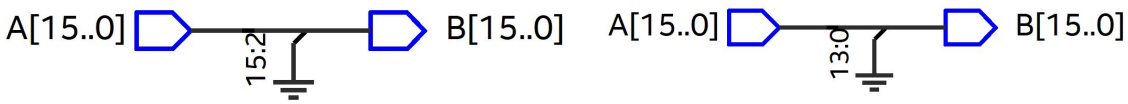
O Right Shift e Left Shift são componentes que fazem deslocamento de bits. O Right Shift realiza deslocamento para direita e o Left Shift para a esquerda.

Figura 6: Declaração do Right Shift

```
entity RigthShift_2_16 is
    Port (
                    A : IN STD_LOGIC_VECTOR (15 downto 0);
                    B : OUT STD_LOGIC_VECTOR (15 downto 0));
end RigthShift_2_16;
```

- IN: A; tipo in STD_LOGIC_VECTOR (15 downto 0) recebem valores em binário.
- OUT: B; tipo out STD_LOGIC_VECTOR (15 downto 0).

Figura 7: RTL Viewer do RightShift e LeftShift (respectivamente)



1.3.4 Comparador

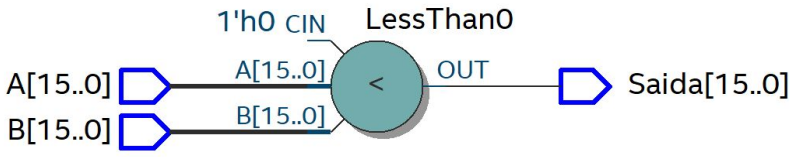
O Comparador recebe dois valores e caso o primeiro seja menor que o segundo retorna “1111111111111111” caso não seja retorna “0000000000000000”.

Figura 8: Declaração do Comparador

```
entity Comparador is
    Port (
                    A, B : STD_LOGIC_VECTOR (15 downto 0);
                    Saida : out STD_LOGIC_VECTOR (15 downto 0)
        );
end Comparador;
```

- IN: A, B; tipo STD_LOGIC_VECTOR (15 downto 0)
- OUT: Saida; tipo out STD_LOGIC_VECTOR (15 downto 0).

Figura 9: RTL View do Comparador



1.3.5 Multiplexador 3_8

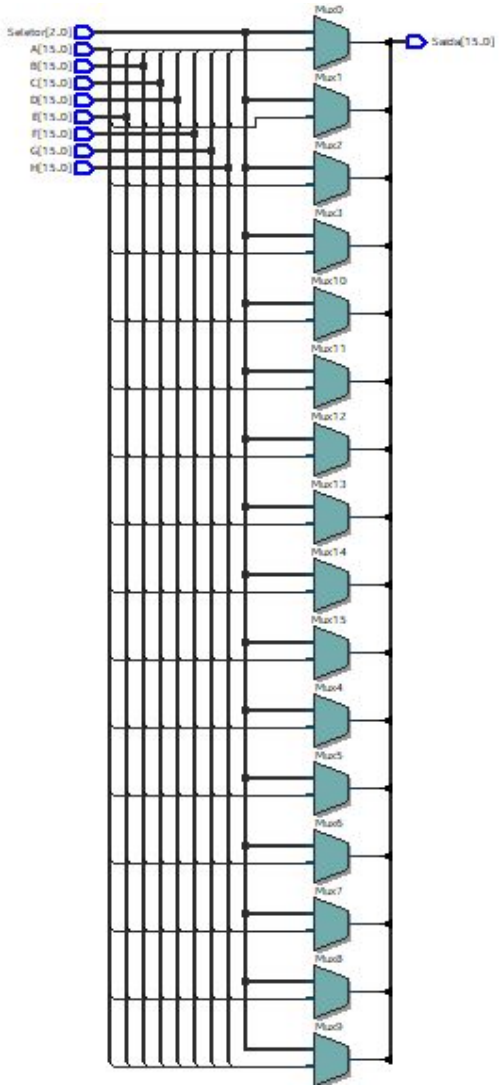
O multiplexador é um componente que recebe várias entradas e retorna uma única saída.

Figura 10: Declaração do Multiplexador

```
entity Multiplexer3_8 is
    Port(
                        Seletor : in  STD_LOGIC_VECTOR (2 downto 0);
          A, B, C, D, E, F, G, H: in STD_LOGIC_VECTOR (15 downto 0);
          Saida  : out STD_LOGIC_VECTOR (15 downto 0));
end Multiplexer3_8;
```

- IN:
    - Seletor; tipo in STD_LOGIC_VECTOR (2 downto 0)
    - A, B, C, D, E, F, G, H; tipo in STD_LOGIC_VECTOR (15 downto 0)
- OUT:
    - Saida; tipo out STD_LOGIC_VECTOR (15 downto 0)

Figura 11: RTL Viewer do Multiplexador



1.3.6 Unidade Lógica Aritmética (ULA)

A Unidade Lógica Aritmética (ULA) desenvolvida está descrita na figura 12 e conta com 20 operações aritméticas, todas estão listadas na tabela 4.

Figura 12: Declaração da ULA

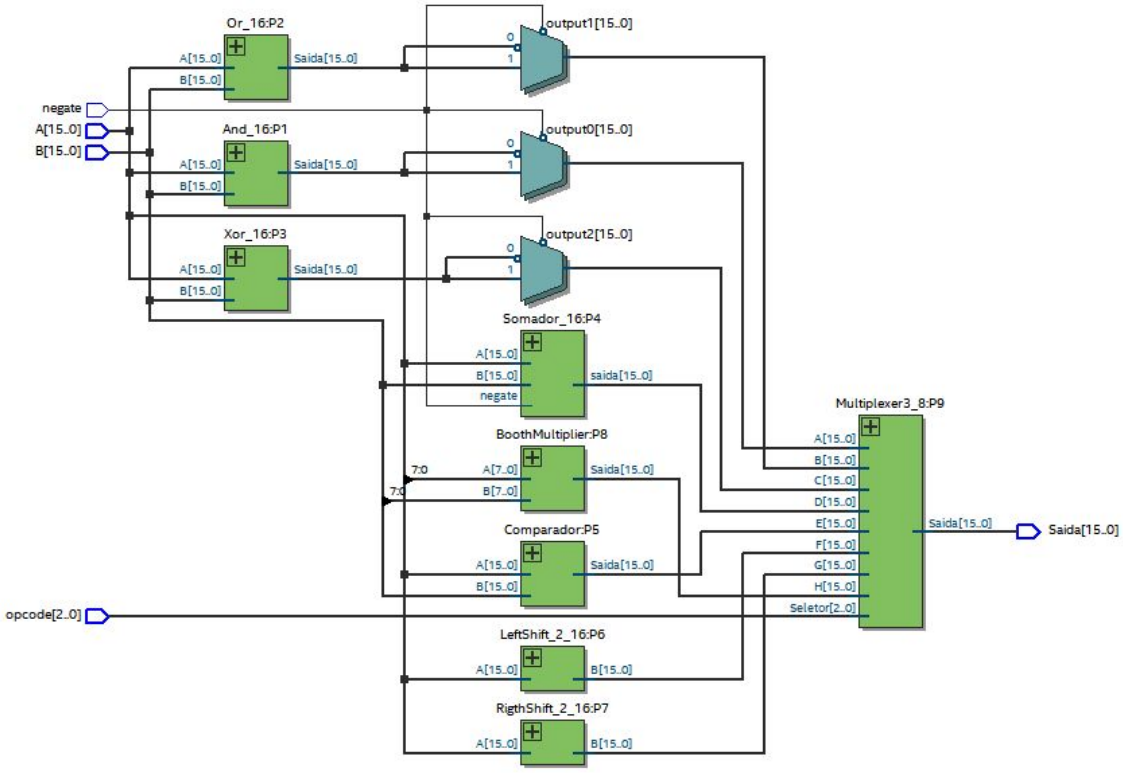
```
entity ULA_16 is
        port (
                opcode : IN STD_LOGIC_VECTOR(2 downto 0);
                negate : IN STD_LOGIC;
                A, B : IN STD_LOGIC_VECTOR(15 downto 0);
                Saida: OUT STD_LOGIC_VECTOR(15 downto 0)
        );
end ULA_16;
```

- IN:
  - opcode; tipo in STD_LOGIC_VECTOR (2 downto 0)
  - negate; tipo in STD_LOGIC
  - A, B; tipo in STD_LOGIC_VECTOR (15 downto 0)
- OUT:
  - Saida; tipo out STD_LOGIC_VECTOR (15 downto 0)

Tabela 5 – Operações da ULA

INSTRUÇÃO	SELETOR 1	SELETOR 2	SELETOR 3	Negate
AND	0	0	0	0
NAND	0	0	0	1
OR	0	0	1	0
NOR	0	0	1	1
XOR	0	1	0	0
XNOR	0	1	0	1
ADD	0	1	1	0
SUB	0	1	1	1
Compara	1	0	0	x
ShiftLeft2	1	0	1	x
ShiftRigth2	1	1	0	x
Mul	1	1	1	x

Figura 13: RTL Viewer da ULA



1.3.7 Somador Aritmético

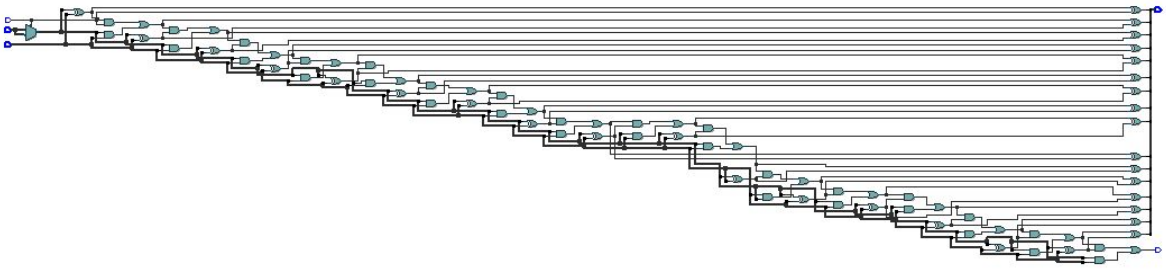
O somador simples que soma dois valores de até 16 bits.

Figura 14: Declaração do somador aritmético

```
Entity Somador_16 is
	Port(
		A, B : IN STD_LOGIC_VECTOR(15 downto 0);
		negate : IN STD_LOGIC;
		cout : OUT STD_LOGIC;
		saida : OUT STD_LOGIC_VECTOR(15 downto 0)
		);
End Somador_16;
```

- IN:
  - A, B; tipo in STD_LOGIC_VECTOR; recebem valores em binário.
  - negate; tipo in STD_LOGIC; recebe valor 0 para realizar uma soma e valor 1 para realizar uma subtração.
- OUT:
  - saida; tipo out STD_LOGIC_VECTOR; retorna o valor das duas entrada adicionados.
  - cout; tipo out STD_LOGIC; retorna um buffer caso a soma ultrapasse o limite de bits.

Figura 15: RTL View do Somador Aritmético



1.3.8 Contador do 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.

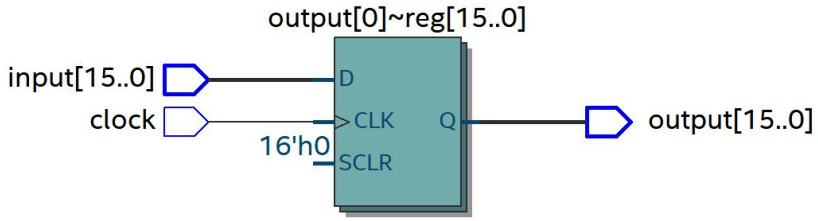
Figura 16: Declaração do Contador do PC

```
ENTITY Count_PC IS
        PORT (

        clock: IN STD_LOGIC ;
        input : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
        output : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
        );
END Count_PC ;
```

- IN:
    - clock; tipo in STD_LOGIC (15 downto 0)
    - input; tipo in STD_LOGIC_VECTOR (15 downto 0)
- OUT:
    - output; tipo in STD_LOGIC_VECTOR (15 downto 0)

Figura 17: RTL Viewer do Contador do PC



1.3.9 Memória de Instruções

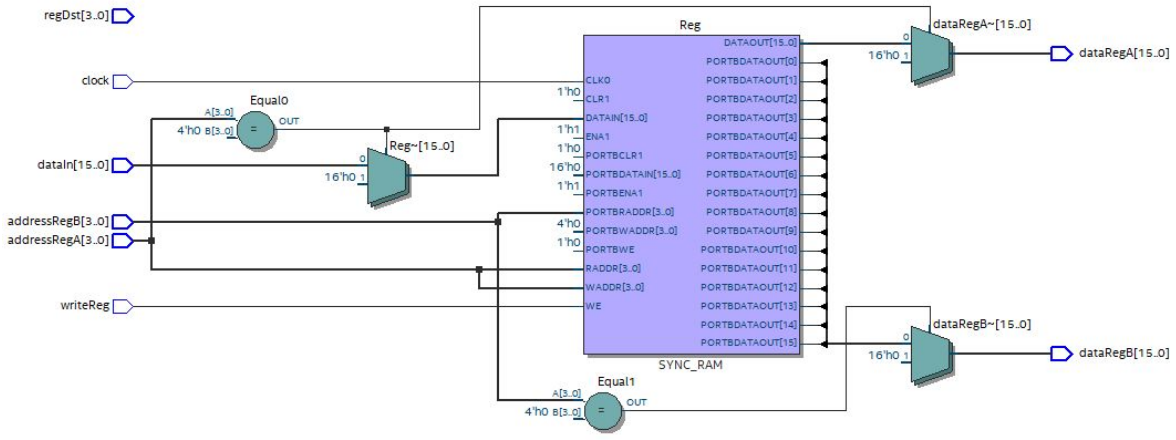
A memória de instruções, também conhecida como memória ROM (*Read-Only-Memory*), é onde as instruções são escritas e podem ser lidas.

Figura 18: Declaração da memória de instruções

```
ENTITY ROM_16 IS
PORT(
        PC_address:     IN STD_LOGIC_VECTOR(15 DOWNTO 0);
        INSTRUCTION:    OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
);
END ROM_16;
```

- IN: PC_address; tipo STD_LOGIC_VECTOR (15 downto 0)
- OUT: INSTRUCTION; tipo STD_LOGIC_VECTOR (15 downto 0)

Figura 19: RTL Viewer da Memória ROM



1.3.10 Banco de Registradores

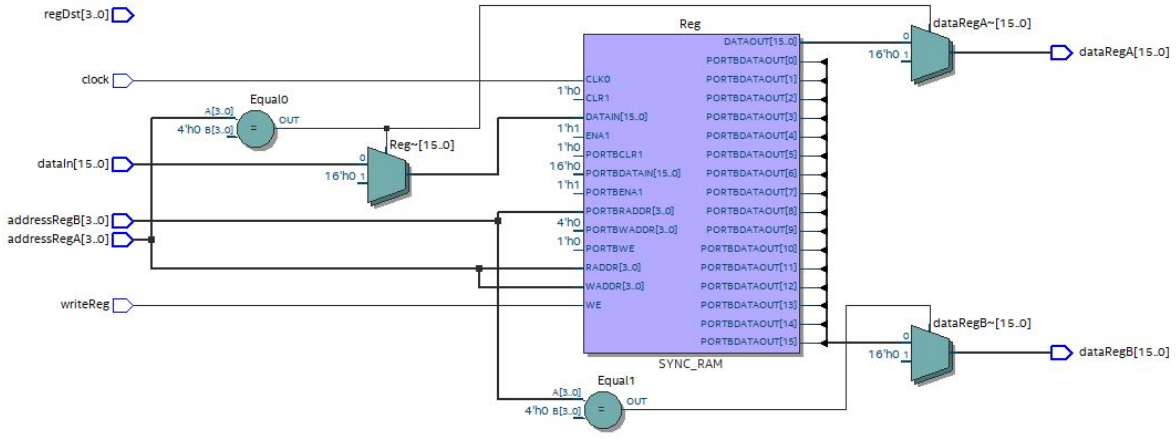
O banco de registradores é um componente composto por um conjunto de registradores que podem ser acessados de forma organizada. Nele podem ser executadas operações de leitura dos dados anteriormente gravados e de escrita de dados para modificar as informações internas.

Figura 20: Declaração do Banco de registradores

```
Entity REGBANK16 is port
	(
		clock : in std_logic;
		writeReg: in std_logic; -- Sinal da unidade de controle
		dataRegA: out std_logic_vector (15 downto 0);
		dataRegB: out std_logic_vector (15 downto 0);
		dataIn: in std_logic_vector (15 downto 0); -- Dado a ser escrito
		regDst: in std_logic_vector (3 downto 0);  -- Registrador de destino
		addressRegA: in std_logic_vector (3 downto 0);  -- Endereço do resgistrador 1
		addressRegB: in std_logic_vector (3 downto 0)   -- Endereço do resgistrador 2
	);
end REGBANK16;
```

- IN:
  - clock; tipo in STD_LOGIC;
  - whiteReg; tipo in STD_LOGIC; Sinal da unidade de controle.
  - dataIn; tipo in STD_LOGIC_VECTOR; Dado a ser escrito.
  - regDst; tipo in STD_LOGIC_VECTOR; Registrador de destino.
  - addressRegA; tipo STD_LOGIC_VECTOR; Endereço do registrador 1.
  - addressRegB; tipo STD_LOGIC_VECTOR; Endereço do registrador 2.
- OUT:
  - daraRegA; tipo out STD_LOGIC_VECTOR (15 downto 0)
  - daraRegB; tipo out STD_LOGIC_VECTOR (15 downto 0)

Figura 21: RTL Viewer do Banco de Registradores



1.3.11 Registrador Flip-Flop tipo D

O Kraken 2.0 register 16 bits utiliza registradores flip-flop do tipo D (data ou dado, pois armazena o bit de entrada) de 16 bits.

Figura 22: Declaração da memória de instruções

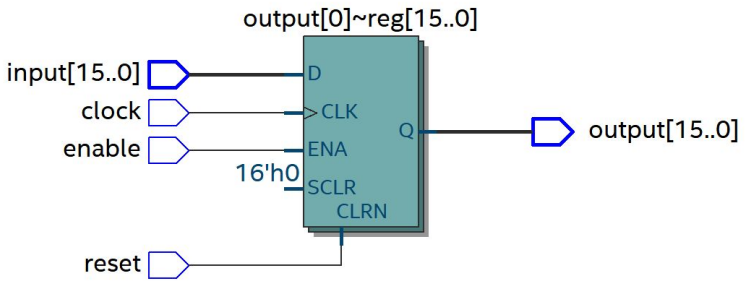
```
ENTITY D_FLIPFLOP IS
	PORT (

	clock, reset, enable: in STD_LOGIC;
	input: in STD_LOGIC_VECTOR(15 DOWNTO 0);
	output: buffer STD_LOGIC_VECTOR(15 DOWNTO 0)
	);

END D_FLIPFLOP;
```

- IN:
  - clock, reset, enable; tipo in STD_LOGIC;
  - input; tipo in STD_LOGIC_VECTOR;
- OUT:
  - output; tipo buffer STD_LOGIC_VECTOR;

Figura 23: RTL Viewer do Registrador FlipFlop tipo D



1.3.12 Memória RAM

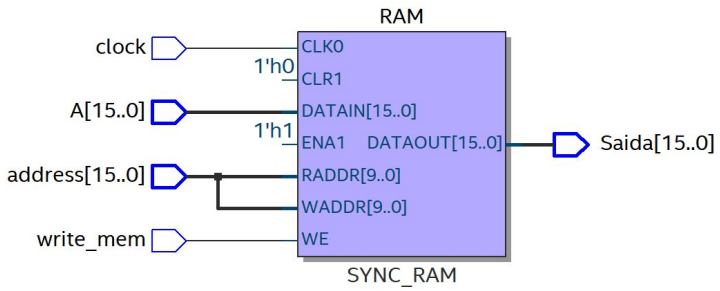
A memória RAM (***R**andom **A**ccess **M**emory*) é um tipo de memória que permite o acesso aos arquivos armazenados no computador. É uma memória de leitura e escrita.

Figura 24: Declaração da memória RAM

```
entity RAM_16 is
  port (
    write_mem, clock : in  std_logic;
    address : in  std_logic_vector(15 downto 0);
    A : in  std_logic_vector(15 downto 0);
    Saida : out std_logic_vector(15 downto 0)
  );
end entity RAM_16;
```

- IN:
    - write_mem; tipo in STD_LOGIC; que quando ativado permite a escrita de um valor, e quando desativado permite a leitura de um valor.
    - clock; tipo in STD_LOGIC; quando ativo é possível realizar alguma ação entre as mostradas acima;
    - address; tipo STD_LOGIC_VECTOR; recebe um endereço. A do tipo STD_LOGIC_VECTOR, recebe um valor a ser escrito.
    - A; tipo STD_LOGIC_VECTOR; recebe um valor a ser escrito.
- OUT:
    - Saida; tipo STD_LOGIC_VECTOR; retorna o valor escrito ou lido.

Figura 25: RTL Viewer da memória RAM



1.3.13 Extensor de 4 para 16 bits

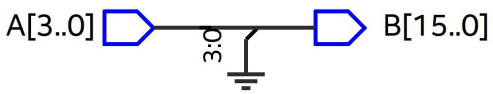
O extensor de 4 para 16 bits é um componente que aumenta o tamanho de uma variável em relação a quantidade de bits, para que esta seja alinhada caso necessário.

Figura 26: Declaração do extensor de 4 para 16 bits

```
entity Extender4to16 is
    Port (
                        A : IN STD_LOGIC_VECTOR (3 downto 0);
                        B : OUT STD_LOGIC_VECTOR (15 downto 0));
end Extender4to16;
```

- IN:
    - A; tipo in STD_LOGIC_VECTOR; recebe valor de 4 bits.
- OUT:
    - B; tipo STD_LOGIC_VECTOR; ao fazer a conversão do valor, retorna o mesmo com 16 bits.

Figura 27: RTL Viewer do Extensor de 4 para 16



1.3.14 Unidade de Controle

A Unidade de Controle opera as operações lógicas e aritméticas que passam por ela. Nela cada instrução aciona determinadas flags que serão usadas para controlar os outros componentes.

Tabela 6 – Flags da unidade de controle do Kraken 2.0

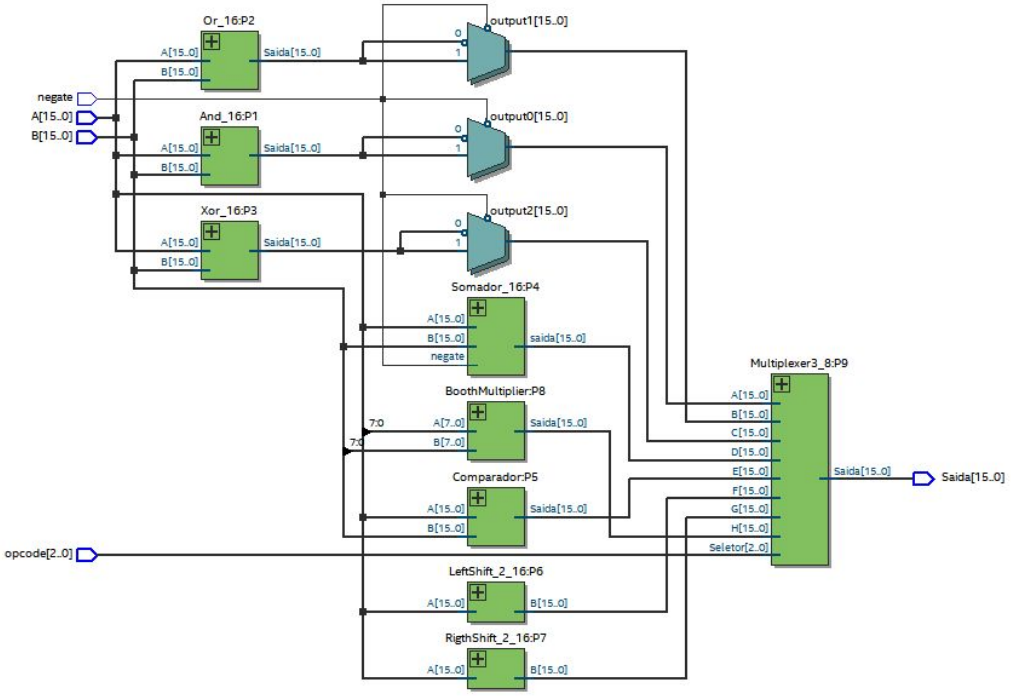
SINAIS DE CONTROLE UNICICLO										
Instrução	Opcode	RegWrite	ULAOp	ULANeg	ALUSrc	MemWrite	MemRead	MemParaReg	Branch	Jump
AND	0000	1	000	0	0	0	0	0	0	0
NAND	0000	1	000	1	0	0	0	0	0	0
OR	0000	1	001	0	0	0	0	0	0	0
NOR	0000	1	001	1	0	0	0	0	0	0
XOR	0000	1	010	0	0	0	0	0	0	0
XNOR	0000	1	010	1	0	0	0	0	0	0
ADD	0000	1	011	0	0	0	0	0	0	0
SUB	0000	1	011	1	0	0	0	0	0	0
COMPARADOR	0000	1	100	0	0	0	0	0	0	0
LeftShift2	0000	1	101	0	0	0	0	0	0	0
RigthShift2	0000	1	110	0	0	0	0	0	0	0
BoothMultiplier	0000	1	111	0	0	0	0	0	0	0
MOVE	0000	1	011	0	0	0	0	0	0	0
lw	0001	0	011	0	1	0	1	1	0	0
sw	0010	0	011	0	1	1	0	0	0	0
beq	0011	0	011	1	0	0	0	0	1	0
bne	0100	0	011	1	0	0	0	0	1	0
movi	0101	1	011	0	1	0	0	0	0	0
slti	0110	0	100	x	1	0	0	0	0	0
addi	0111	1	011	0	1	0	0	0	0	0
addiu	1000	1	011	0	1	0	0	0	0	0
j	1001	0	0	x	0	0	0	0	0	1
jr	1010	0	0	x	0	0	0	0	0	1
jal	1011	0	0	x	0	0	0	0	0	1

Figura 28: Declaração da Unidade de Controle

```
ENTITY Unidade_de_controle is
    Port (
        opcode : in STD_LOGIC_VECTOR(3 downto 0);
        ulaop : out std_logic_vector(2 downto 0);
        jump, aluSrc, memWrite, memRead, memToReg,  branch : out std_logic;
        regWrite: out std_logic
    );
END Unidade_de_controle;
```

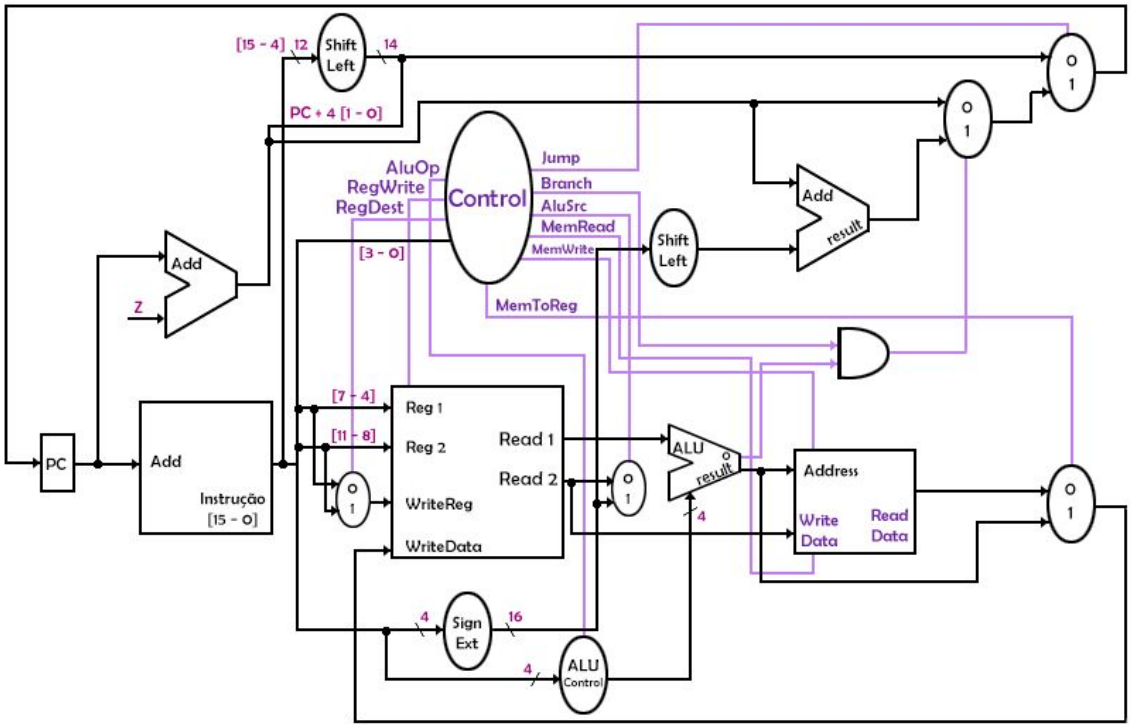
- IN:
    - **opcode**; tipo in STD_LOGIC; recebe o número das operações.
- OUT:
    - **ulaOp**; tipo out STD_LOGIC_VECTOR; código de operação que sairá para a ULA.
    - **jump**; tipo out STD_LOGIC_VECTOR; flag para verificar se será feito um salto incondicional - valores imediatos - ou para um endereço no registrador.
    - **aluSrc**; tipo out STD_LOGIC_VECTOR; flag para a ULA saber se o valor a ser lido será proveniente do banco de registradores ou dos 8 bits menos significativos.
    - **memWrite**; tipo out STD_LOGIC_VECTOR; flag para indicar para a memória RAM se é para escrever no endereço passado.
    - **memRead**; tipo out STD_LOGIC_VECTOR; flag para indicar para a memória RAM se é para ler do endereço passado.
    - **memToReg**; tipo out STD_LOGIC_VECTOR; flag para indicar de onde vem o valor a ser escrito no registrador destino (ULA OU Memória de dados).
    - **branch;** tipo out STD_LOGIC_VECTOR, flag para indicar que se será feito um salto condicional.
    - **regWrite;** tipo out STD_LOGIC, flag para indicar que se será escrito algum valor no banco de registradores.

Figura 29: RTL View da Unidade de Controle



1.4 Datapath

Figura 30: Visualização do Datapath Idealizado

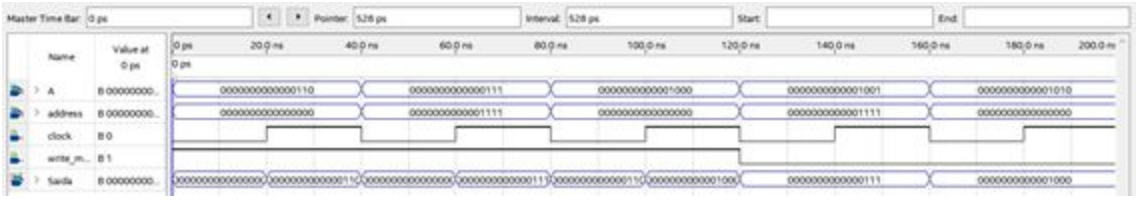


2 Simulações e Testes

O processador Kraken 2.0 possui todos os componentes funcionando, contudo sua implementação ainda não está completa, pois o mesmo ainda não está funcionando quando ligados todos os componente. Por conta disto será apresentado os testes apenas de alguns componentes que achamos necessários que fossem apresentados.

Memória RAM

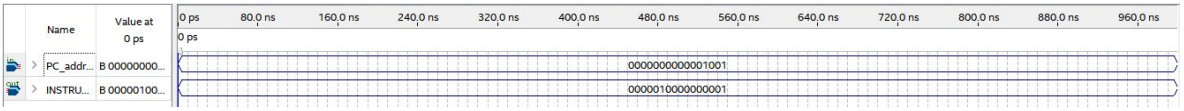
Figura 31: Waveform do teste da memória RAM



O teste na Waveform foi realizado usando 5 ciclos de clock, em que no primeiro é dado o valor 0000000000000110 para ser escrito no endereço 0000000000000000, no segundo clock o valor 0000000000000111 foi atribuído para ser escrito no endereço 0000000000001111, no terceiro clock o valor 0000000000001000, sobrescreve o valor 0000000000000110 no endereço 0000000000000000, no quarto e quinto clock são lidos os valores armazenados no endereço 0000000000000000 e 0000000000001111.

Memória ROM

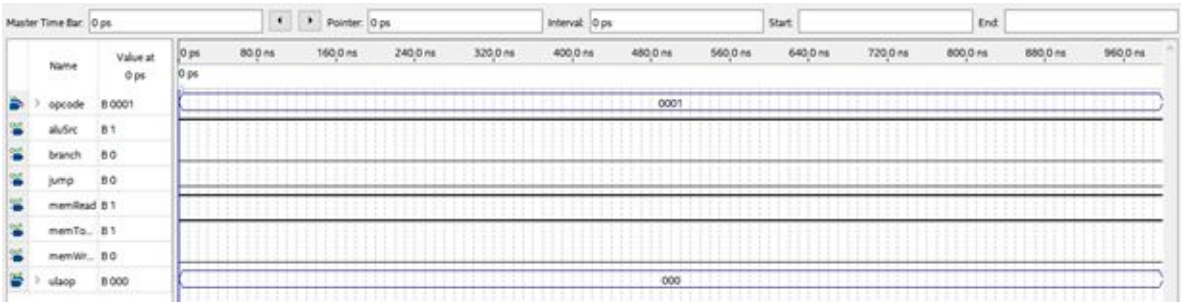
Figura 32: Waveform do teste da memória ROM



Para realizar o teste na Waveform, foi atribuído para o endereço o valor 0000000000001001, e o mesmo retornou a instrução 0000010000000001 que era correspondente ao endereço.

Unidade de Controle

Figura 33: Waveform do teste da unidade de controle

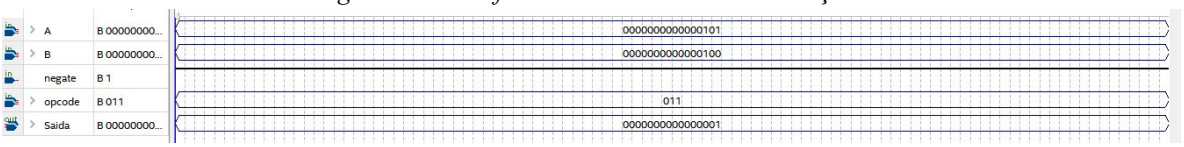


Para cada opcode de entrada na Waveform, há uma saída correspondente a um valor da tabela 6.

ULA

Subtração

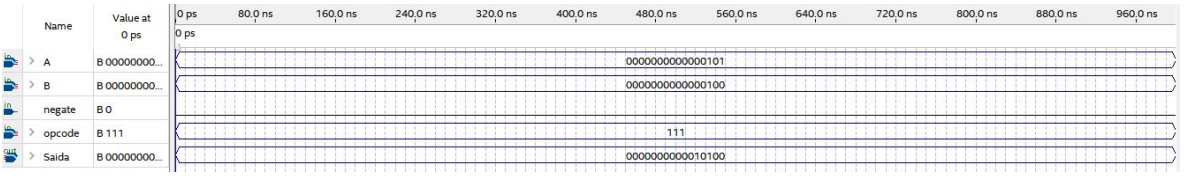
Figura 34: Waveform do teste da ULA - Subtração



Para realizar o teste na Waveform, foi atribuído dois valores para as entradas A e B, o opcode recebeu 011, que é referente a soma e subtração, e o negate recebeu 1, pois este determina se será realizado uma soma, quando recebe valor 0 (zero), ou uma subtração quando recebe valor 1 (um). A saída recebe o resultado da soma.

Multiplicação

Figura 35: Waveform do teste da ULA - Multiplicação



A multiplicação é efetuada utilizando técnicas do algoritmo de booth, como demonstrado no teste da Waveform foi atribuído dois valores para as entradas A e B, o opcode recebeu 111, que é referente multiplicação, não importando o valor passado para a flag negate, no processo da operação há dois sinalizadores de 8 bits, que recebem os 8 primeiros bits mais significativos das entradas A e B. A multiplicação pode ser realizada para valores menores ou iguais a 127. A saída recebe o resultado da operação.

3 Considerações finais

O objetivo deste trabalho foi relatar todo o processo de criação do processador Kraken 2.0 de 16 bits, seu conjunto de instruções, a descrição do hardware, o datapath, simulações e testes da forma mais detalhada possível.

O desenvolvimento do projeto ao todo foi um grande aprendizado, tanto pessoal quanto intelectual. Apesar do Kraken 1.0, sua versão inicial, apresentado em 2017.2 ter sido um fracasso, as suas falhas e os erros nos proporcionaram aprendizado. Aprendemos que não é na primeira e nem na segunda queda que devemos desistir, e que de fato sempre podemos melhorar.

Todo o conteúdo apresentado ao longo da disciplina foi visto com outros olhos, e essa nova perspectiva contribuiu para o desenvolvimento do Kraken 2.0. Não foi apenas uma nova versão do processador, foi uma nova versão da equipe. Em um ano é possível crescer muito. Analisamos as falhas do nosso antigo processador, e nesta segunda versão conseguimos corrigir muitas delas. Claro que nada é perfeito, mas nós fizemos o possível para apresentar o nosso melhor.