

103962

**PC2:**  
**Implementação em Verilog da Unidade de  
Processamento**

**(Laboratório de sistemas computacionais: Arquitetura e organização  
de computadores)**

São José dos Campos - Brasil

Abril de 2018



103962

## **PC2:**

### **Implementação em Verilog da Unidade de Processamento (Laboratório de sistemas computacionais: Arquitetura e organização de computadores)**

Relatório apresentado à Universidade Federal de São Paulo como parte dos requisitos para aprovação na disciplina de Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores.

Docente: Prof. Dr. Tiago de Oliveira

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Abril de 2018

# Resumo

O relatório em questão apresenta detalhes a respeito da implementação e desenvolvimento de um sistema computacional com arquitetura baseada na MIPS monociclo. Durante este relatório será apresentado detalhadamente informações à respeito do conjunto e formato de instruções, modos de endereçamento e arquitetura-base, abordando os principais conceitos e componentes.

**Palavras-chaves:** MIPS, unidade de controle, arquitetura.

# Listas de ilustrações

|   |    |
|---|----|
| Figura 1 – Hierarquia de Memória . . . . .  | 11 |
| Figura 2 – Arquitetura Harvard . . . . .  | 13 |
| Figura 3 – MIPS Monociclo . . . . .   | 16 |
| Figura 4 – Arquitetura Base . . . . .   | 19 |
| Figura 5 – Caminho de dados para instruções do tipo R . . . . .                                   | 29 |
| Figura 6 – Caminho de dados para instrução lw . . . . .   | 30 |
| Figura 7 – <i>Waveform 1:</i> Contador de Programas . . . . .                                     | 31 |
| Figura 8 – <i>Waveform 2:</i> Contador de Programas (reset ativado) . . . . .                     | 31 |
| Figura 9 – <i>Waveform 1:</i> Memória de Instruções . . . . .                                     | 32 |
| Figura 10 – <i>Waveform 2:</i> Memória de Instruções . . . . .                                    | 32 |
| Figura 11 – <i>Waveform 1:</i> Banco de Registradores . . . . .                                   | 33 |
| Figura 12 – <i>Waveform 1:</i> Unidade Lógica e Aritmética . . . . .                              | 33 |
| Figura 13 – <i>Waveform 2:</i> Unidade Lógica e Aritmética . . . . .                              | 34 |
| Figura 14 – <i>Waveform 3:</i> Unidade Lógica e Aritmética . . . . .                              | 34 |
| Figura 15 – <i>Waveform 1:</i> Memória de Dados . . . . .   | 34 |
| Figura 16 – <i>Waveform 1:</i> Mux de Entrada da ULA . . . . .                                    | 35 |
| Figura 17 – <i>Waveform 2:</i> Mux de Saída da Memória de Instruções . . . . .                    | 35 |
| Figura 18 – <i>Waveform 3:</i> Mux de Entrada do Extensor de Bits . . . . .                       | 35 |
| Figura 19 – <i>Waveform 4:</i> Mux de Entrada do Banco de Registradores . . . . .                 | 35 |
| Figura 20 – <i>Waveform 5:</i> <i>BigMux (bneq=1 e zero=0)</i> . . . . .                          | 36 |
| Figura 21 – <i>Waveform 6:</i> <i>BigMux (bneq=1 e zero=1)</i> . . . . .                          | 36 |
| Figura 22 – <i>Waveform 7:</i> <i>BigMux (jump com saído dada por signal)</i> . . . . .           | 36 |
| Figura 23 – <i>Waveform 8:</i> <i>BigMux (jump register com saído dada por regdata)</i> . . . . . | 37 |
| Figura 24 – <i>Waveform 1:</i> <i>BigMux (jump register com saído dada por regdata)</i> . . . . . | 37 |
| Figura 25 – <i>PC/Instruction Memory</i> . . . . .  | 40 |
| Figura 26 – <i>PC/Instruction Memory/Mux pré Banco de Registradores</i> . . . . .                 | 41 |
| Figura 27 – <i>Módulos anteriores/Mux pré Bit Extender</i> . . . . .                              | 41 |
| Figura 28 – <i>Módulos anteriores/Mux pré Bit Extender/Bit Extender</i> . . . . .                 | 41 |
| Figura 29 – <i>Módulos anteriores/Banco de Registradores</i> . . . . .                            | 41 |
| Figura 30 – <i>Módulos anteriores/Banco de Registradores/Mux pré ULA</i> . . . . .                | 42 |
| Figura 31 – <i>Módulos anteriores/ULA (instrução add)</i> . . . . .                               | 42 |
| Figura 32 – <i>Módulos anteriores/ULA/Bit Extender (26 para 32 bits)</i> . . . . .                | 42 |
| Figura 33 – <i>Módulos anteriores/Memória de Dados</i> . . . . .                                  | 43 |
| Figura 34 – <i>Módulos anteriores/Memória de Dados/Mux pós Memória de Dados</i> .                 | 43 |

# **Lista de tabelas**

|  |    |
|--|----|
| Tabela 1 – Instruções de formato R . . . . . | 14 |
| Tabela 2 – Instruções de formato I . . . . . | 15 |
| Tabela 3 – Instruções de formato J . . . . . | 15 |
| Tabela 4 – Conjunto de Instruções . . . . .  | 18 |

# Sumário

|            |  |           |
|------------|--|-----------|
| <b>1</b>   | <b>INTRODUÇÃO</b>                      | <b>7</b>  |
| <b>2</b>   | <b>OBJETIVOS</b>                       | <b>9</b>  |
| <b>2.1</b> | <b>Geral</b>                           | <b>9</b>  |
| <b>2.2</b> | <b>Específico</b>                      | <b>9</b>  |
| <b>3</b>   | <b>FUNDAMENTAÇÃO TEÓRICA</b>           | <b>11</b> |
| <b>3.1</b> | <b>Sistema Computacional</b>           | <b>11</b> |
| <b>3.2</b> | <b>Hierarquia de memória</b>           | <b>11</b> |
| <b>3.3</b> | <b>Arquitetura Computacional</b>       | <b>12</b> |
| 3.3.1      | Arquitetura Havard                     | 12        |
| 3.3.2      | Arquitetura RISC                       | 13        |
| <b>3.4</b> | <b>MIPS</b>                            | <b>14</b> |
| 3.4.1      | Tipos de Instruções                    | 14        |
| 3.4.2      | Datapath                               | 15        |
| <b>4</b>   | <b>DESENVOLVIMENTO</b>                 | <b>17</b> |
| <b>4.1</b> | <b>Conjunto de instruções</b>          | <b>17</b> |
| 4.1.1      | Conjunto de Instruções                 | 17        |
| <b>4.2</b> | <b>Arquitetura-base do Processador</b> | <b>19</b> |
| 4.2.1      | Contador de Programa                   | 19        |
| 4.2.2      | Memória de Instruções                  | 20        |
| 4.2.3      | Banco de Registradores                 | 21        |
| 4.2.4      | Unidade Lógica e Aritmética            | 22        |
| 4.2.5      | Memória de Dados                       | 23        |
| 4.2.6      | Unidade de Controle                    | 24        |
| 4.2.7      | MUX                                    | 24        |
| 4.2.7.1    | Big MUX                                | 25        |
| 4.2.8      | Extensor                               | 26        |
| 4.2.9      | Entrada e Saída                        | 27        |
| 4.2.9.1    | I/O por programação                    | 27        |
| 4.2.9.2    | I/O por interrupção                    | 27        |
| 4.2.9.3    | I/O por DMA                            | 27        |
| <b>4.3</b> | <b>Exemplos</b>                        | <b>28</b> |
| 4.3.1      | Instruções do tipo R                   | 29        |
| 4.3.2      | Instrução <i>Load Word</i>             | 30        |

|                              |  |           |
|------------------------------|--|-----------|
| <b>5</b>                     | <b>RESULTADOS E DISCUSSÕES . . . . .</b> | <b>31</b> |
| <b>5.1</b>                   | <b>Módulos separados . . . . .</b>       | <b>31</b> |
| 5.1.1                        | Contador de Programa . . . . .           | 31        |
| 5.1.2                        | Memória de Instruções . . . . .          | 32        |
| 5.1.3                        | Banco de Registradores . . . . .         | 32        |
| 5.1.4                        | Unidade Lógica e Aritmética . . . . .    | 33        |
| 5.1.5                        | Memória de Dados . . . . .               | 34        |
| 5.1.6                        | Unidade de Controle . . . . .            | 34        |
| 5.1.7                        | MUX e BigMux . . . . .                   | 35        |
| 5.1.8                        | Extensor . . . . .                       | 37        |
| <b>5.2</b>                   | <b>Módulos unificados . . . . .</b>      | <b>37</b> |
| <b>6</b>                     | <b>CONSIDERAÇÕES FINAIS . . . . .</b>    | <b>45</b> |
| <b>REFERÊNCIAS . . . . .</b> |  | <b>47</b> |

# 1 Introdução

Com o avanço tecnológico, a utilização de sistemas computacionais em nosso dia-a-dia se tornou cada vez mais presente e relevante. E entender o funcionamento desta tecnologia é fundamental. Um computador é uma máquina capaz de realizar variados tipos de tratamentos de informações ou processamento de dados, possuindo inúmeros atributos como armazenamento e processamento de dados, operações lógicas e aritméticas, tratamento de imagens gráficas, etc...

Além disso, é composto por diversos componentes que possuem funções distintas e complexas, que quando são combinadas funcionam em perfeita harmonia. Um exemplo disso é a CPU (unidade central de processamento) que em termos didáticos poderia ser considerada o cérebro deste sistema.

Para garantir a realização correta de todas as suas funções, um computador necessita de um conjunto de instruções (código de máquina compreendido pela CPU, que atua como interface entre hardware e software), também conhecido como ISA (*Instruction Set Architecture*). Este conjunto é tratado pela Unidade de Processamento que garante todos os requisitos necessários para a execução de um programa, sejam elas operações lógicas e aritméticas, acesso à memória, entrada e saída de dados, etc. Outra unidade fundamental para o funcionamento do computador é a Unidade de Controle, que será mostrada mais adiante.

Este é apenas um resumo que mostra quão importante e complexo pode ser um computador, e nas próximas páginas iremos conhecer em detalhes o funcionamento e desenvolvimento de alguns destes componentes.



## 2 Objetivos

### 2.1 Geral

O objetivo geral deste projeto é o desenvolvimento de um sistema computacional que opere de forma similar a um processador baseado na arquitetura MIPS, de forma que suas instruções sejam testadas e seu funcionamento seja comprovado através de simulação em elementos de *hardware*.

### 2.2 Específico

O objetivo específico é o desenvolvimento e implementação em lógica programável da Unidade de Processamento, incluindo todos os componentes que a compõem. Como por exemplo, o Banco de Registradores, Unidade Lógica e Aritmética, Multiplexadores e todos os módulos apresentados na arquitetura base com exceção da Unidade de Controle.



# 3 Fundamentação Teórica

## 3.1 Sistema Computacional

Um sistema computacional consiste num conjunto de dispositivos eletrônicos (*hardware*) capazes de processar informações de acordo com um programa (*software*).

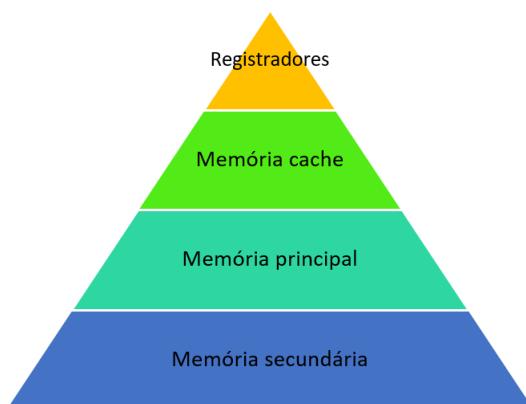
Um computador é composto de dispositivos de entrada e saída (como teclado, mouse, monitor, etc...), memória (que armazena todos os dados) e processador. Onde os dados que chegam através dos dispositivos de entrada são salvos na memória, para que após seu processamento sejam mostrados através do dispositivo de saída.

## 3.2 Hierarquia de memória

Dada a complexidade de um sistema computacional e suas operações realizadas é simples notar a necessidade da existência de diferentes níveis de armazenamento de dados, este conceito de criação é conhecido como hierarquia de memória.

Basicamente, a memória de um computador pode ser dividida em quatro níveis principais, são eles: registradores, memória CACHE, memória principal e memória secundária. A figura abaixo representa a ordenação desses níveis hierárquicos, de forma que quanto mais no topo estiver, maior será o custo e velocidade de acesso e menor será sua capacidade de armazenamento .

Figura 1 – Hierarquia de Memória



Fonte: MaxiEduca ([1](#))

De baixo para cima, a **memória secundária** é composta por dispositivos não voláteis como memórias externas de armazenamento em massa - por exemplo, disco rígido,

DVD, disquete, fita magnética, pen drive, entre outros. Sua principal característica é o armazenamento de dados que precisam se buscados antes de acessados, pois possuem baixa velocidade de acesso.

Subindo um nível, a **memória principal** possui uma alta velocidade de acesso, através da utilização da memória RAM (*Random Access Memory*). Que além de permitir leitura e escrita, mantém armazenado os programas operacionais básicos. A **memória CACHE** serve como ponte entre a troca de dados dos registradores e memória RAM. É uma memória de acesso rápido e otimiza os blocos de memória que são pertinentes ao processo em execução.

Finalizando, os **registradores** são os meios mais rápidos e computacionalmente caros de armazenamento de dados. Usualmente utilizados de forma temporária, registradores são as unidades de memória diretamente utilizadas pelas instruções atualmente sendo executadas pelo sistema.

### 3.3 Arquitetura Computacional

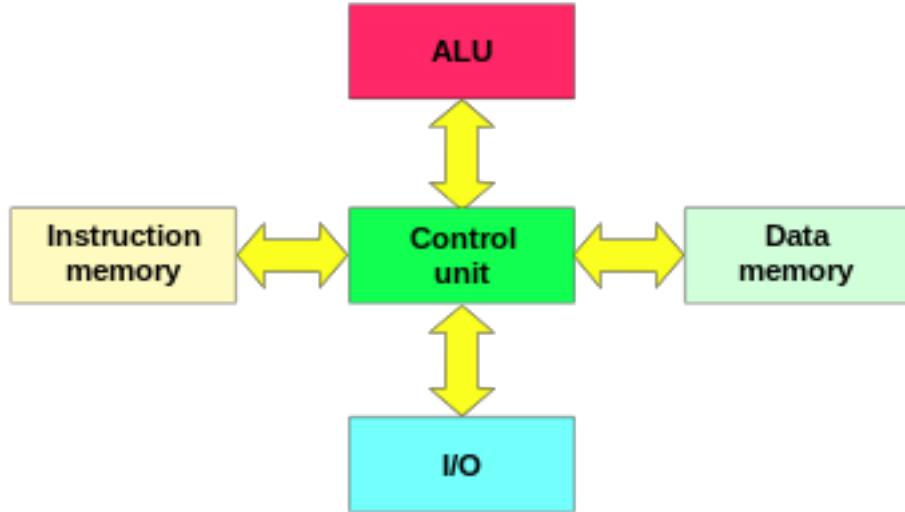
Para a construção de um processador é necessário definir quais instruções este deverá executar, definindo desta forma sua arquitetura. A arquitetura organiza a estrutura computacional de forma a otimizar seu funcionamento, a partir da combinação de diversas instruções e visando a execução de alguma função específica.

#### 3.3.1 Arquitetura Harvard

A Arquitetura de Harvard é uma arquitetura de computador que se distingue das outras por possuir separadamente circuito para sinais e armazenamento para, dados e instruções, que são independentes em termos de barramento e ligação ao processador (2).

Esta arquitetura se baseia no conceito da Arquitetura de Von Neumann e é composta por unidade de controle, memória de instrução, memória de dados, unidade lógica e aritmética e módulos de entrada e saída. A diferença entre a arquitetura de Von Neumann e a Harvard é que a última separa o armazenamento e o comportamento das instruções do CPU e os dados, enquanto a anterior utiliza o mesmo espaço de memória para ambos (2).

Figura 2 – Arquitetura Harvard



Fonte: Diego Macedo (2)

### 3.3.2 Arquitetura RISC

Ao contrário da CISC, a arquitetura RISC (*Reduced Instruction Set Computers*) possui um número reduzido e simplificado de instruções, podendo operar a velocidades maiores de clock.

Desta forma, este tipo de arquitetura possui uma ênfase maior em *software* e requer um trabalho maior do programador, uma vez que exige mais linhas de código. Suas principais características são:

1. Número reduzido de ciclos de clock por instrução;
2. Utiliza um formato fixo de instrução;
3. Conjunto reduzido de instruções.

Processadores baseados nesta arquitetura são mais simples e muito mais baratos, possuindo um menor número de circuitos internos, como por exemplo os processadores Alpha.

Apesar das diversas peculiaridades e diferenças destas arquiteturas, atualmente muitos modelos de processadores abrigam características de ambas. E as grandes fabricantes utilizam desta combinação visando melhorar o desempenho durante a execução das instruções.

## 3.4 MIPS

Vimos que a arquitetura de um processador é a forma como ele se comporta funcionalmente. Com o passar dos anos e do desenvolvimento de novas tecnologias, diversas arquiteturas foram criadas, dentre elas a arquitetura MIPS.

Desenvolvida nos anos 80 por pesquisadores da Universidade de Stanford, esta arquitetura segue o padrão de arquiteturas RISC. Utilizando de um número reduzido de registradores e um conjunto de instruções menor.

Diversos conjuntos de MIPS foram implementados utilizando diferentes números de registradores, contudo, os números principais são os de 32 bits (número que será utilizado neste projeto) e o de 64 bits.

Outra característica interessante é que a execução desta arquitetura é feita em cinco estágios, são estes: busca, decodificação, execução, acesso à memória e escrita de dados.

### 3.4.1 Tipos de Instruções

A arquitetura MIPS realiza diversos tipos de operações, de forma que seus tipos de instruções são divididos em três partes: R, I e J.

É no **tipo R** que todas as instruções de operação lógica e aritmética com dados dos registradores se encontram, como por exemplo soma e subtração. Ainda há subdivisões neste tipo de instrução chamados: opcode, RS, RT, RD, *shamt* e *funct*.

Tabela 1 – Instruções de formato R

| Campo                 | Opcode                                       | RS  | RT  | RD   | <i>shamt</i>  | <i>funct</i>   |
|-----------------------|--|---|---|--|---|--|
| <b>Tamanho (bits)</b> | 6  | 5   | 5   | 5  | 5   | 6  |
| <b>Bits</b>           | 31 - 26                                      | 25 - 21   | 20 - 16   | 15 - 11  | 10 - 6  | 5-0  |
| <b>Função</b>         | Funciona como um identificador de instruções | Representam os endereços dos bancos de registradores que os dados serão retirados | Representam os endereços dos bancos de registradores que os dados serão retirados | Endereço do registrador que receberá o resultado da operação | Quantidade de bits que serão deslocados (se for necessário) | Diferencia instruções na ULA, funciona como uma extensão do opcode |

Fonte: Autor

O **tipo I**, possui instruções que realizam operações lógicas e aritméticas com dados de *offset* a partir do campo imediato, ou seja, executam acesso à memória e executam saltos condicionais. A tabela abaixo ilustra quais são os campos deste tipo de instrução.

Tabela 2 – Instruções de formato I

| Campo                 | <i>Opcode</i>                                | RS  | RT   | <i>offset</i>   |
|-----------------------|--|---|--|---|
| <b>Tamanho (bits)</b> | 6  | 5   | 5  | 11  |
| <b>Bits</b>           | 31 - 26                                      | 25 - 21   | 20 - 16  | 15 - 0  |
| <b>Função</b>         | Funciona como um identificador de instruções | Representam os endereços dos bancos de registradores que os dados serão retirados | Endereço do registrador que receberá o resultado da operação | O campo offset ou imediato, possui um valor codificado que será usado para possíveis saltos condicionais ou operações aritméticas |

Fonte: Autor

Finalizando, o terceiro tipo de instrução é o **tipo J**, responsável pelas instruções de saltos. Por esta característica, possui uma divisão mais simples.

Tabela 3 – Instruções de formato J

| Campo                 | <i>Opcode</i>                                | <i>adress</i>                           |
|-----------------------|--|---|
| <b>Tamanho (bits)</b> | 6  | 26                                      |
| <b>Bits</b>           | 31 - 26                                      | 25 - 0                                  |
| <b>Função</b>         | Funciona como um identificador de instruções | Endereço no qual o salto será destinado |

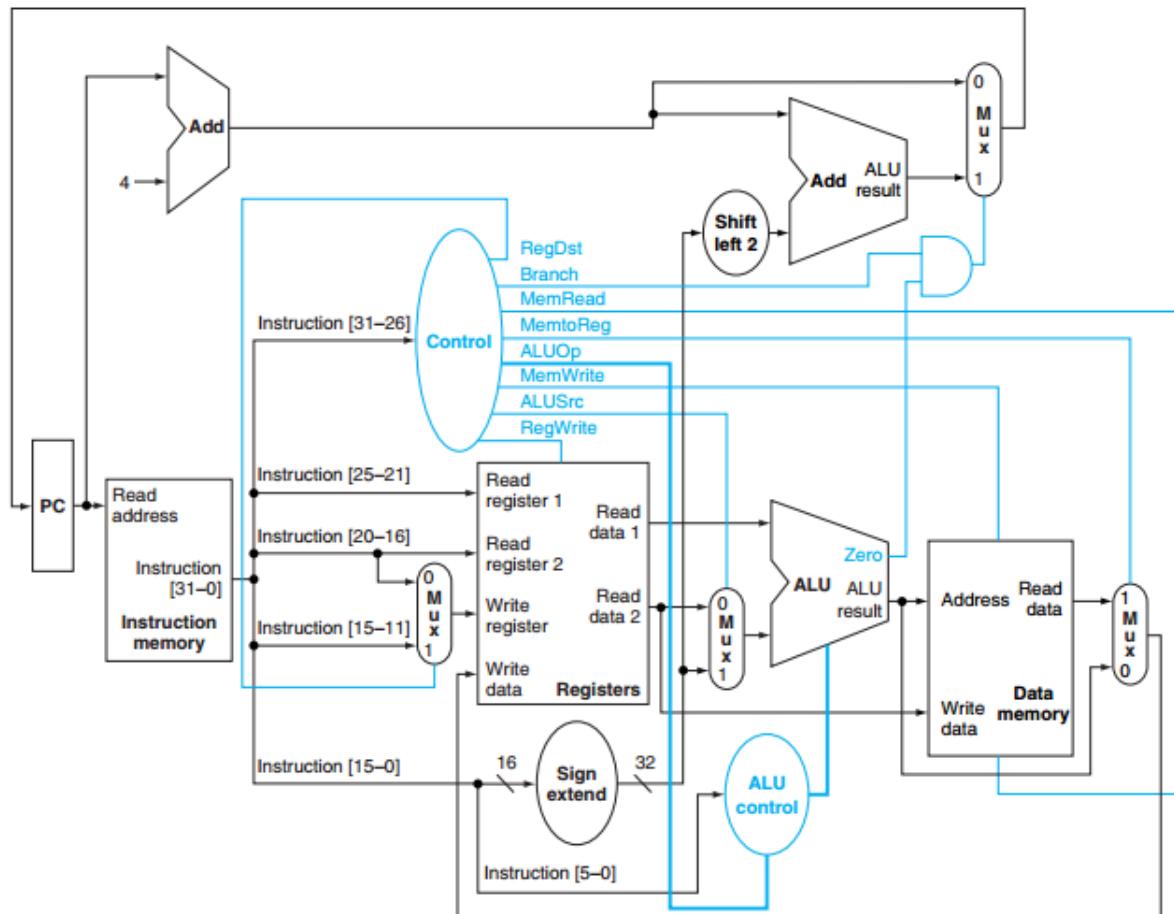
Fonte: Autor

### 3.4.2 Datapath

Como o nome sugere, o *Datapath* ou caminho de dados, é o caminho tomado pelos bits processados em uma instrução, representando o total de componentes presentes na arquitetura.

Quando estes componentes interligados recebem a ação de uma sinal de controle (*flag*), direcionam a operação através das diferentes partes do circuito com o intuito de obter um resultado específico. A figura 2, mostra o caminho de dados da arquitetura MIPS.

Figura 3 – MIPS Monociclo



Fonte: Acervo Livro (3)

# 4 Desenvolvimento

Para este projeto, o modelo MIPS foi escolhido pois é baseado na arquitetura RISC, portanto é mais simplificado e fácil de implementar. Além disso, é um modelo didaticamente viável e possui uma vasta literatura para fins de estudo.

## 4.1 Conjunto de instruções

A arquitetura MIPS diversos tipos de instruções com características que variam de acordo com o tipo de implementação. Em um MIPS de 32 bits, pode-se trabalhar com 32 registradores, acessados por um intervalo de 5 bits.

Para este projeto, utilizaremos um conjunto semelhante e reduzido ao MIPS. Consistindo por 26 instruções, que visam abranger todas as operações básicas do processador, como operações lógicas e aritméticas, saltos, acesso à memória, etc...

### 4.1.1 Conjunto de Instruções

Para este projeto, será utilizado quatro modos de endereçamento utilizados na arquitetura MIPS.

**Endereçamento imediato:** considerado o modo mais simples de endereçamento, o endereçamento imediato possui o operando contido no campo de endereço, portanto, não necessita acesso à memória para buscar o operando.

**Endereçamento por registrador:** este modo referencia o respectivo registrador no campo de endereço e nele já contém o operando, sem fazer acesso à memória. É utilizado por instruções como sub, add, or, etc.

**Endereçamento por deslocamento:** no endereçamento por deslocamento ou de base, o registrador contém um endereço base e um campo de endereço é um deslocamento.

**Endereçamento relativo ao PC:** neste modo, a próxima instrução a ser executada é relativa ao endereço da instrução atual. Desta forma, seu endereço é calculado através da soma entre uma constante da instrução e o PC. Geralmente é utilizada por instruções de desvio condicional, como a beq.

**Endereçamento pseudodireto:** utilizado por instruções de salto incondicional, no endereçamento pseudodireto, o endereço é formado pela concatenação dos 26 bits da instrução com os bits mais altos do PC.

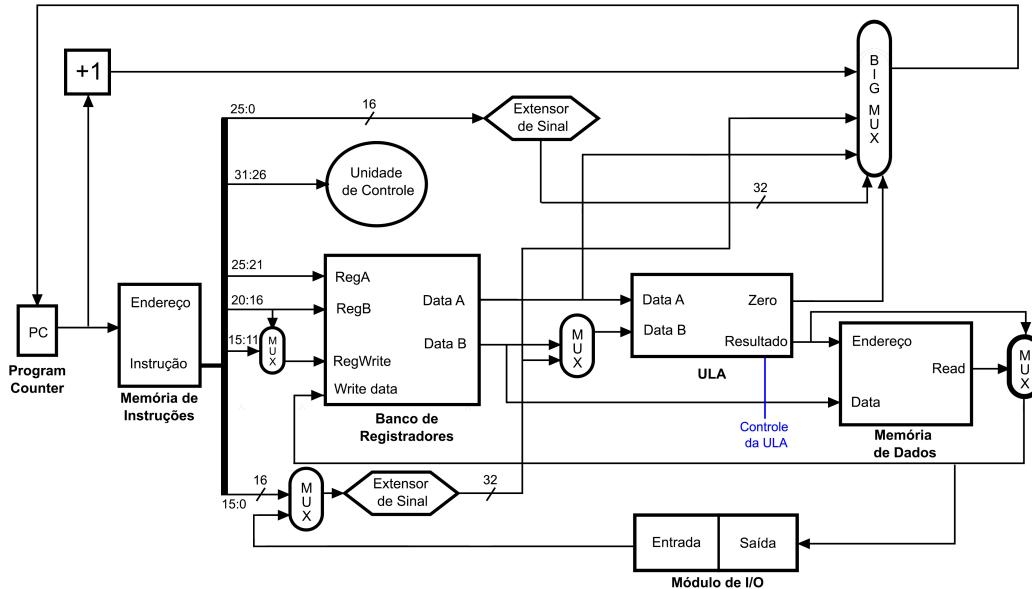
Tabela 4 – Conjunto de Instruções

| Categoría  | Instrucción             | Tipo  | tipo de instrucción |
|------------|-------------------------|-------|---------------------|
| Aritmética | adição                  | add   | R                   |
| Aritmética | subtração               | sub   | R                   |
| Aritmética | adição com imediato     | addi  | R                   |
| Aritmética | subtração com imediato  | subi  | R                   |
| Aritmética | multiplicação           | mult  | R                   |
| Lógicas    | NOT                     | not   | R                   |
| Lógicas    | AND                     | and   | R                   |
| Lógicas    | OR                      | or    | R                   |
| Lógicas    | XOR                     | xor   | R                   |
| Lógicas    | set less than           | stl   | R                   |
| Lógicas    | deslocamento à esquerda | shfl  | R                   |
| Lógicas    | deslocamento à direita  | shfr  | R                   |
| Memória    | load word               | lw    | I                   |
| Memória    | load imediato           | li    | I                   |
| Memória    | store worf              | sw    | I                   |
| Saltos     | branch and equal        | beq   | I                   |
| Saltos     | branch and not equal    | bne   | I                   |
| Saltos     | branch and equal zero   | beqz  | I                   |
| Saltos     | jump                    | jump  | J                   |
| Saltos     | jump to register        | jumpr | J                   |
| Outros     | NOP                     | nop   | R                   |
| Outros     | Input                   | in    | Outros              |
| Outros     | Output                  | out   | Outros              |
| Outros     | HLT                     | hlt   | R                   |

Fonte: Autor

## 4.2 Arquitetura-base do Processador

Figura 4 – Arquitetura Base



Fonte: Autor

O caminho de dados deste projeto é semelhante ao da arquitetura MIPS monociclo, porém com um grau de complexidade menor. Nele, estão presentes alguns componentes importantes para o funcionamento do processador, que serão explicados detalhadamente.

### 4.2.1 Contador de Programa

O **Contador de Programa**, ou *Program Counter* (PC), é o primeiro elemento do caminho de dados. Nele é armazenado o endereço atual da instrução em execução através de um registrador, que sempre é atualizado a cada subida do *clock*.

Essa atualização de valor pode ser efetuada de três formas principais. Na primeira, o valor do registrador simplesmente é acrescido em uma unidade, indicando qual será a próxima instrução a ser executada.

A segunda maneira é voltada para quando ocorre algum tipo de desvio, como nas instruções do tipo *branch* ou *jump*. Neste caso, o valor atual do PC não é acrescido de uma unidade, mantendo-se o mesmo. Para estas verificações utilizou-se também a variável chamada zero, que é um sinal provindo da Unidade Lógica e Aritmética, utilizado no tratamento de algumas condições de salto.

No terceiro e último caso, quando o sinal de *reset* é igual a 1, o valor do PC é zerado. Outro aspecto, é que existe um sinal de halt (*hlt*) como entrada no módulo, que é

utilizado em algumas instruções com o objetivo de impedir a mudança no PC até que o usuário indique

```

1 module Program_Counter (clock, adress, reset, hlt, pgcount, zero, jump, jr, PCBranch);
2
3     input clock, reset, hlt, zero, jump, jr, PCBranch;
4     input [9:0] adress;
5     output reg [9:0] pgcount;
6
7     wire [9:0] newValue;
8     assign newValue = adress;
9
10
11    always @(posedge clock) begin
12
13        if((PCBranch && zero ) || jump || jr)
14            pgcount <= newValue;
15
16        else if(hlt) begin
17
18            end
19
20        else if(reset) begin
21            pgcount = 0;
22            end
23
24        else begin
25            pgcount <= adress + 1;
26            end
27
28    end
29
30
31 endmodule

```

#### 4.2.2 Memória de Instruções

A **Memória de Instruções** (*Instruction Memory*) é responsável pelo armazenamento de todas as instruções realizadas pelo processador, em posições contíguas de memória.

Estas instruções são enviadas para outros componentes referenciados pelo endereço contido no PC, que após isso é atualizado na próxima subida de *clock* e busca a instrução armazenada no endereço seguinte.

A implementação deste módulo, consiste basicamente em registradores que guardam cada instrução do processador ou possivelmente algum programa específico, como o de Fibonacci que será utilizado futuramente.

Dado um determinado endereço, a saída do programa é a instrução equivalente a posição de memória deste vetor de registradores. E a execução deste módulo ocorre na descida do *clock* pois **PORQUE OCORRE NA DESCIDA??**

O código exemplificado abaixo, contém apenas 3 instruções no endereço da memória, para melhor visualização de sua implementação. Porém, na parte final deste relatório, será possível visualizar a implementação completa.

```

1  module MemoryInstruction (address, InstructionOut, clock);
2
3      input [9:0] address;
4      input clock;
5      output [31:0] InstructionOut;
6      reg [31:0] mem [31:0];
7      integer flag = 1;
8
9      always @ (posedge clock) begin
10
11          if(flag == 1) begin
12
13              mem[0] = {32'd0}; //nop
14              mem[1] = {32'd20}; //add
15              mem[2] = {32'd50}; //sub
16              flag <= 1;
17
18      end
19
20  end
21  assign InstructionOut = mem[address];
22
23 endmodule

```

### 4.2.3 Banco de Registradores

O Banco de Registradores é o componente em que são armazenados dados temporários pertinentes à execução das instruções, sendo o tipo de memória situada no topo da hierarquia (4).

Neste componente, existem 32 registradores de propósito geral que são constantemente alterados com base nos endereços de entrada das instruções. E seu armazenamento é realizado com base em um sinal, que indica quando os dados provenientes do final do circuito devem ou não ser escritos nestes endereços. Este módulo possui 4 entradas e 2 saídas. As variáveis **Reg1** e **Reg2** representam os dois endereços dos registradores de entrada, já o **WriteRegister** é o endereço do registrador de escrita, que recebe a saída do multiplexador anterior ao Banco de Registradores, visto na Arquitetura-Base da Figura 4. Já o **WriteData** designou-se para receber os dados escritos no registrador de escrita, provenientes do multiplexador que seleciona a saída da ULA ou da Memória de Instruções. As duas saídas **Data1** e **Data2**, representam os dados de leitura de registradores de entrada.

O Banco de Registradores foi implementado de forma síncrona ao *clock* em borda de subida a cada ciclo, para que o mesmo funcione de acordo com um sinal de seleção determinado **Reg Write** no algoritmo. Quando este sinal é positivo, significa que o dado

será escrito no registrador, caso contrário, nada seria escrito nos registradores.

```

1 module RegBank (clock, Reg1, Reg2, RegWrite, WriteData, WriteRegister, Data1, Data2);
2
3     input [4:0] Reg1, Reg2, WriteRegister;
4     input [31:0] WriteData;
5     input RegWrite, clock;
6     output [31:0] Data1, Data2;
7     reg [31:0] Register [31:0];
8
9     always @(posedge clock)
10    begin
11        Register[0] = 32'b0;
12        if(RegWrite)
13            Register [WriteRegister] = WriteData;
14        end
15
16        assign Data1 = Register[Reg1];
17        assign Data2 = Register[Reg2];
18
19
20 endmodule

```

#### 4.2.4 Unidade Lógica e Aritmética

O componente responsável por realizar cálculos e deslocamentos dos dados de entrada é a **Unidade Lógica e Aritmética** (ULA) ou ALU (*Arithmetic Logical Unity*). Pode ser utilizada para diversos propósitos, verificar igualdade entre dados para determinando saltos condicionais, calcular um endereço de escrita ou ainda realizar operações lógicas e aritméticas.

Seu funcionamento não depende do *clock*, como outros componentes, mas de variações de seus sinais de entrada. Devido ao fato de realizar diversas operações, a ULA conta com um único sinal de controle, gerado com base na leitura do opcode da instrução, diferenciando suas possíveis formas de execução.

Este componente conta com um *opcode* que identifica qual é a instrução a ser executada. Além disso, como este código já é suficiente para todas as possíveis operações, não foi necessário utilizar o *funct* de determinados tipos de instruções. Utilizou-se também uma variável para determinar o sinal das operações, sendo possível identificar se o resultado é positivo ou negativo.

```

1 module ULA (op, data1, data2, result, signal_zero, signal_neg, shamt);
2
3
4     input [31:0] data1;
5     input [31:0] data2;
6     input [4:0] shamt;
7     input [4:0] op;
8
9     output signal_zero;
10    output signal_neg;
11    output reg [31:0] result;

```

```

12
13
14     always @(data1 or data2 or op or shamt) begin
15         case(op[4:0])
16             5'b00000: result = data1 + data2; //add
17             5'b00001: result = data1 - data2; //sub
18             5'b00010: result = data1 + 1;           //addi
19             5'b00011: result = data1 - 1;           //subi
20             5'b00100: result = data1 < data2 ? 1 : 0; //slt
21             5'b00101: result = data1[15:0] * data2[15:0]; //mult
22             5'b00110: result = data1 > data2 ? 1 : 0; // Set greater than (Branch)
23             5'b00111: result = data1 << shamt; //shfl
24             5'b01000: result = data1 >> shamt; //shfr
25             5'b01001: result = ~data1; //not
26             5'b01010: result = data1 & data2; //and
27             5'b01011: result = data1 | data2; //or
28             5'b01100: result = data1 ^ data2; //xor
29                 5'b01101 : result = data1 == data2 ? 1 : 0; // Set equal than (
30                             Branch)
31                 5'b01110 : result = data1 <= data2 ? 1 : 0; // Set less or equal
32                             than (Branch)
33                 5'b01111 : result = data1 >= data2 ? 1 : 0; // Set greater or
34                             equal than (Branch)
35             5'b10000 : result = data1 != data2 ? 1 : 0; // Set different
36             5'b10001 : result = data1 == 0 ? 1 : 0; //Branch and equal zero -
37                             beqz
38             default : result = 0;
39         endcase
40     end
41
42     assign signal_zero = (result == 0);
43     assign signal_neg = (($signed(result) < 0));
44
45 endmodule

```

#### 4.2.5 Memória de Dados

A **Memória de Dados** tem o objetivo de armazenar eventuais informações durante a execução de uma instrução.

Sua escrita ocorre a partir do direcionamento de uma *flag*, semelhante ao Banco de Registradores e suas únicas entradas são o endereço de acesso e o dado que será salvo neste respectivo endereço. Contudo, apenas as instruções load e store realizam acesso direto aos seus endereços e os dados contidos nele.

Portanto, sua implementação pode ser considerada mais simplificada em relação a outros módulos da arquitetura. Sua entrada principal consiste em um dado e seu respectivo endereço. Um sinal de controle determina se este dado será gravado na memória (no caso da instrução *store word*) ou se irá retornar o valor referente ao endereço de entrada (caso a instrução seja *load word*).

Este módulo atua na borda de descida do *clock*, para evitar que a memória tente

ler um dado de um endereço ainda não calculado do Banco de Registradores. Além disso, a Memória de Dados envia sinais na saída em todo ciclo de *clock*, porém estes valores são utilizados apenas quando selecionados pelo seu respectivo multiplexador.

```

1 module DataMemory(clock, address, flag, DataOut, data);
2
3   input [31:0] data;
4   input [31:0] address;
5   input clock , flag;
6   output [31:0] DataOut;
7   reg [31:0] Out[9:0];
8
9   always @ (negedge clock) begin
10
11     if (flag)
12       Out[address] = data;
13   end
14   assign DataOut = Out[address];
15
16 endmodule

```

#### 4.2.6 Unidade de Controle

Já a **Unidade de Controle** é o módulo responsável por realizar a troca de todas as *flags* com base na instrução em execução. Possui um papel de extrema importância no funcionamento do processador, uma vez que diferencia o resultado das instruções a partir dos seus sinais de controle.

Podemos comparar este componente ao cérebro do processador, orquestrando seu funcionamento e ativando cada flag de cada instrução. Ou seja, a UC que é um circuito combinacional - é dito combinacional pois a saída depende única e exclusivamente das combinações das variáveis de entrada recebidas em um dado momento (5) - recebe e interpreta o opcode de cada instrução e a partir desta interpretação define a disposição adequada dos passos de execução do caminho de dados.

#### 4.2.7 MUX

Os **multiplexadores** (MUX), funcionam como um seletor do que qual informação enviará para a saída. Visto que, o mesmo pode possuir dois ou mais dados de entradas. Como a implementação é igual para todos os MUX's utilizados no processador (com exceção do Big Mux), abaixo tem apresentação de um destes componentes.

```

1 module MuxBank1(input1, input2, out, select); //Mux de entrada do Banco de Registradores
2
3   input select;
4   input [4:0] input1, input2;
5   output reg [4:0] out;
6
7   always @ (*) begin
8

```

```

9      case(select)
10         1'b0 : out = input1;
11         1'b1 : out = input2;
12     endcase
13
14   end
15
16 endmodule

```

#### 4.2.7.1 Big MUX

Um elemento importante no desenvolvimento da arquitetura é o chamado **Big MUX**, que possui o objetivo de determinar qual será o endereço correto do Contador de Programa ao término de cada instrução. Seu funcionamento é baseado em sinais seletores que selecionam qual entrada será a saída de retorno do PC.

Este componente foi pensado para contemplar corretamente o funcionamento das instruções *branch on equal* e *branch on not equal*. Que são executadas a partir de um sinal 0 ou 1.

Para tratar intruções de *branch on equal*, *branch on not equal* e *branch on equal zero*, além do sinal de controle (zero) utilizou-se *flags* para determinar qual das entradas fornecidas seriam usadas como saída de volta ao PC, isso porque existe uma diferenciação do valor deste sinal em sua comparação.

Neste componente, também efetuou-se um tratamento para as intruções de *jump* e *jump to register* e por padrão, o valor de saída era o endereço provindo do PC.

```

1 module BigMux(zero, beq, bneq, beqz, selectbm, outputbm, outputpc, sum, signal, regdata);
2
3   input zero, beq, bneq, beqz;
4   input [1:0] selectbm;
5   input [31:0] outputpc, sum, signal, regdata;
6   output reg [31:0] outputbm;
7
8   always @ (outputpc or sum or signal or regdata or zero or selectbm or beq or bneq) begin
9
10     case(selectbm[1:0])
11       2'b01: //branch
12         begin
13           if(bneq == 1 && zero == 1)
14             outputbm = outputpc;
15
16           else if(bneq == 1 && zero == 0)
17             outputbm = sum;
18
19           else if(beq == 1 && zero == 1)
20             outputbm = sum;
21
22           else if(beq == 1 && zero == 0)
23             outputbm = outputpc;
24
25           else if(beqz == 1 && zero == 1)
26             outputbm = sum;

```

```

27
28             else if(beqz == 1 && zero == 0)
29                 outputbm = outputpc;
30             end
31
32         2'b10: outputbm = signal;//jump
33
34         2'b11 : outputbm = regdata;//jump register
35
36     default: outputbm = outputpc; //2'b00
37
38     endcase
39 end
40
41 endmodule

```

#### 4.2.8 Extensor

Outro componente auxiliar do processador, é o extensor de *bit* (*Sign Extend*). Que possui a função de receber uma entrada de um determinado tamanho em bits, concatena-la com 0's ou 1's até atingir um tamanho específico e mandar esta nova informação para outro componente.

Por exemplo, um extensor pode transformar um código de 16 *bits* em um de 32 *bits*, sem alterar a informação original.

Para que esta operação fosse efetuada sem erros, levou-se em consideração a que os números binários são tratados em complemento de 2 na linguagem *Verilog*. Ou seja, caso um número apresentasse 0 como o *bit* mais significativo, bastou-se adicionar a quantidade restante de *bits* para completar 32 com mais 0's. Já se o bit mais significativo fosse 1, o número binário apresentava sinal negativo, e desta forma, seus *bits* restantes eram completados com 1's.

```

1 module BitExtender (DataIn, DataOut); //Extensor de bits de 16 para 32
2
3     input [15:0] DataIn;
4     output reg [31:0] DataOut;
5
6
7     always @ (*)
8         begin
9             if(DataIn[15])
10                 DataOut = {16'b1111111111111111, DataIn};
11             else
12                 DataOut = {16'b0, DataIn};
13         end
14
15
16 endmodule

```

### 4.2.9 Entrada e Saída

Em relação a entrada e saída de dados do processador, será utilizado um módulo de entrada e saída com o objetivo de efetuar transferências de dados entre o processador e os periféricos de I/O. Existem três possíveis formas de realizar esta ação, por interrupção, por programação e por DMA.

Tomando conhecimento das principais formas de implementação, para este projeto o I/O por programação será utilizado. Visto que ele é o que melhor se adequa às necessidades do processador e ao grau de complexidade.

#### 4.2.9.1 I/O por programação

Ocorre a partir do resultado das instruções de I/O que estão presentes no programa de um computador. Cada transferência de dados é iniciado por uma instrução no programa. Este tipo pode admitir dois tipos de transferência:

**Transferência incondicional:** a transferência é realizada independentemente do estado da interface ou periférico.

**Transferência condicional:** a operação de E/S só é realizada se o dispositivo estiver pronto para tal (6). Em geral, o programa deste tipo de transferência possui um laço de espera que efetua constantes testes do estado do dispositivo até que o mesmo indique que a operação pode ser realizada.

#### 4.2.9.2 I/O por interrupção

Neste modo, a transferência é acionada através de uma interrupção, ou seja, a operação De E/S é requerido pelo dispositivo externo através de um pedido de interrupção. Normalmente, este pedido é solicitado quando o dispositivo ou interface está pronto para realizar a transferência.

#### 4.2.9.3 I/O por DMA

O último tipo é o por DMA (*Direct Memory Access*), que pode ser definido como uma técnica de transferência de dados onde os periféricos se comunicam diretamente entre si, utilizando os barramentos de memória e removendo a intervenção da CPU (7). Basicamente, o controlador DMA assume os barramentos para gerenciar diretamente a transferência entre os dispositivos de E/S e a unidade de memória.

Apesar deste módulo fazer parte da Unidade de Processamento, sua implementação será finalização no último ponto de checagem do projeto. Isso porque o seu real funcionamento acontecerá através da utilização da placa FPGA. Contudo, o código abaixo já contempla sua primeira parte, que será finalizada adiante.

1 //Conversor de Binario para BCD (Display de 7 segmentos)

```

2 module BinToBCD2(binario, dezena, unidade);
3     input [6:0] binario;
4     output reg [3:0] dezena, unidade;
5     reg [3:0] centena;
6     integer i;
7
8     always@(binario) begin
9         centena = 4'D0;
10        dezena = 4'D0;
11        unidade = 4'D0;
12
13        for(i = 6; i>=0; i=i-1) begin
14            if(centena >= 5)
15                centena = centena + 3;
16            if (dezena >= 5)
17                dezena = dezena + 3;
18            if (unidade >= 5)
19                unidade = unidade + 3;
20
21            centena = centena << 1;
22            centena[0] = dezena[3];
23
24            dezena = dezena << 1;
25            dezena[0] = unidade[3];
26
27            unidade = unidade << 1;
28            unidade[0] = binario[i];
29        end
30    end
31 endmodule

```

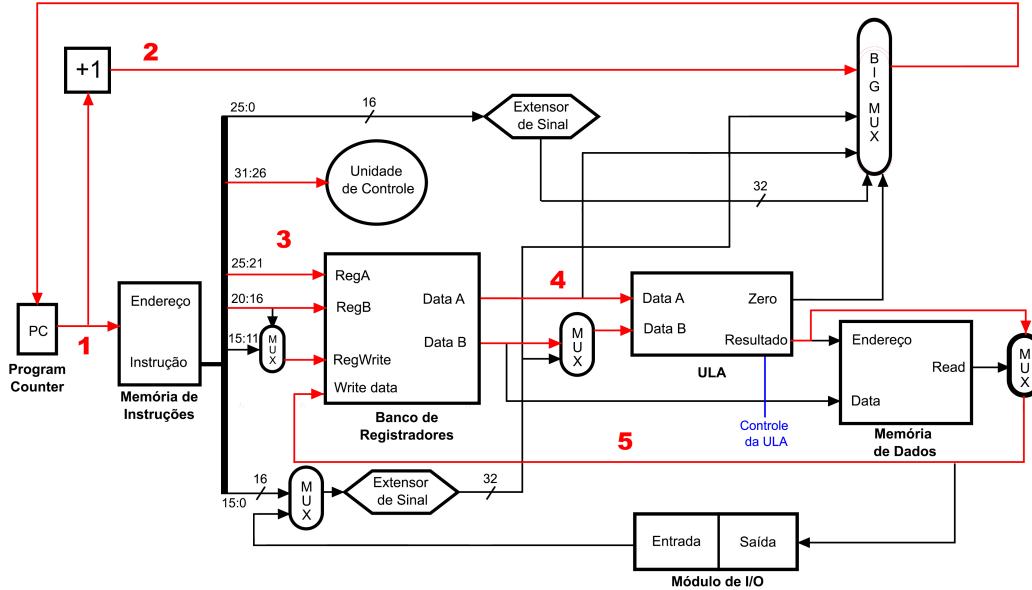
### 4.3 Exemplos

Para validar o desenvolvimento e funcionamento da arquitetura base proposta, foram efetuados testes no caminho de dados a partir da simulação de algumas instruções e seus diferentes tipos.

Esta etapa é importante pois servirá de guia para o funcionamento do processador ao longo de sua implementação.

### 4.3.1 Instruções do tipo R

Figura 5 – Caminho de dados para instruções do tipo R



Fonte: Autor

Inicialmente o primeiro teste é para instruções do tipo R, como soma e subtração. Vale ressaltar que todo o funcionamento dos passos citados são orquestrados a partir da unidade de controle que recebe o opcode e determina quais serão os passos adotados no *datapath*.

**Passo 1:** Inicialmente o endereço da instrução que está disponível no PC é encaminhado para a Memória de Instruções.

**Passo 2:** Em seguida, o endereço atual do PC é incrementado e ao ser selecionado como saída pelo Big MUX, este novo endereço atualizado é retornado ao módulo do *Program Counter*. Vale ressaltar que este novo código referencia a próxima instrução que será executada.

**Passo 3:** Ao mesmo tempo, o endereço da primeira etapa é interpretado pela Memória de Instruções que seleciona qual é a instrução referente a este código e envia estas informações ao Banco de Registradores e a Unidade de Controle.

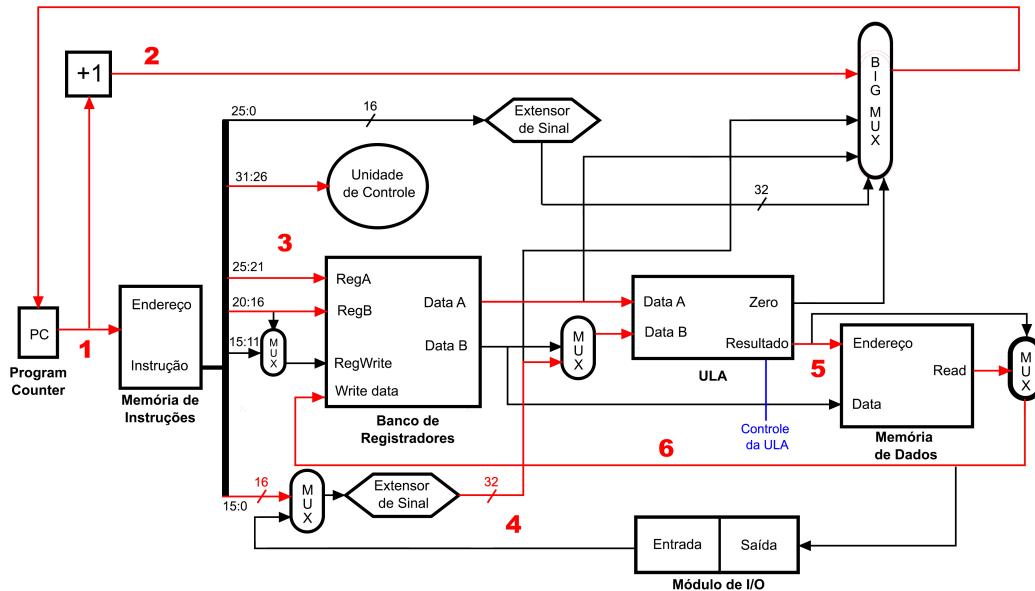
**Passo 4:** Instruções do tipo R, possuem três registradores. Dois registradores de origem (RegA e RegB), que recebem os dados que servirão como argumento para a ULA efetuar determinada operação. E um registrador (RegWrite) de destino, que recebe o resultado da operação realizada.

**Passo 5:** Os dois valores dos registradores de origem são transferidos para a ULA através dos campos Data A e Data B.

A partir do opcode da instrução, a ULA irá determinar qual operação realizar com os dados e o respectivo resultado é encaminhado para Memória de Instruções (mais especificamente no RegWrite), após ser selecionado como saída do MUX.

#### 4.3.2 Instrução Load Word

Figura 6 – Caminho de dados para instrução lw



Fonte: Autor

Outra instrução pertinente é a *lw*, que possui a função de movimentação de dados da memória para registrador(8).

Os passos 1 e 2 são os mesmos citados na sessão 4.1

**Passo 3:** A instrução *lw* utiliza apenas dois registradores (RegA e RegB), o registrador de endereço base é transferido como dado de saída do Banco de Registradores (Data A).

**Passo 4:** Paralelamente a isso, um imediato de 16bits é estendido pelo módulo de Extensor de Sinal e através do MUX, este sinal estendido serve como entrada para ULA (mais especificamente na entrada Data B).

**Passo 5:** A partir do opcode da instrução, a ULA irá determinar somar as informações provindas das entradas, ou seja, do sinal estendido e do registrador de endereço base.

**Passo 6:** O resultado desta operação é transferido para a Memória de Dados que identifica qual é o dado referente ao endereço de entrada. A partir deste endereço, o dado referente volta para o banco de registradores e é salvo.

# 5 Resultados e Discussões

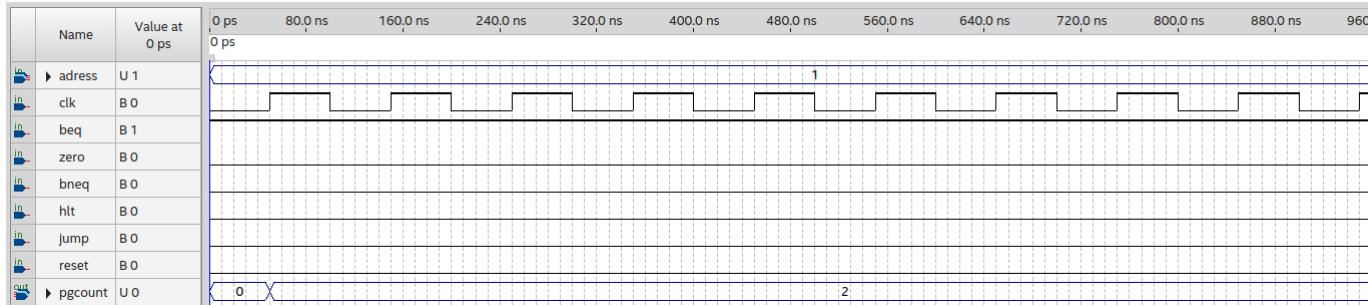
Para verificar o funcionamento e implementação nos modos apresentados anteriormente, efetuou-se simulações em duas etapas: primeiramente de cada módulo separadamente e em seguida com suas partes integradas. Em ambas as opções, seus resultados foram observados a partir de *waveforms* gerados pelo *Quartus*

## 5.1 Módulos separados

### 5.1.1 Contador de Programa

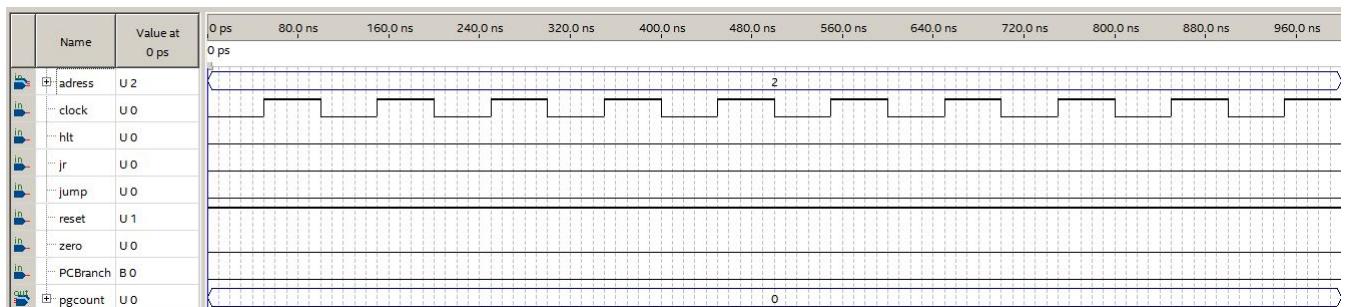
O primeiro módulo implementado foi o PC, seu funcionamento padrão é receber um endereço e somar uma unidade. As figuras abaixo, apresentam para o caso padrão e para o *reset* ativado.

Figura 7 – *Waveform 1: Contador de Programas*



Fonte: Autor

Figura 8 – *Waveform 2: Contador de Programas (reset ativado)*

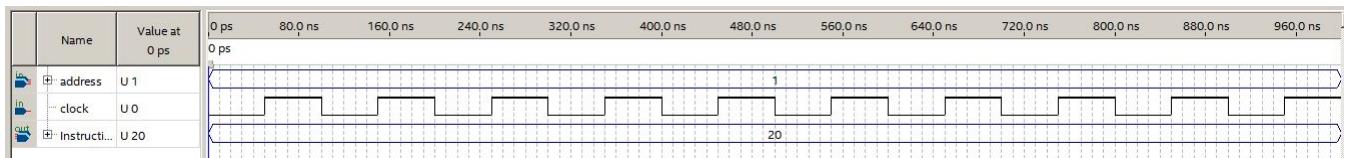


Fonte: Autor

### 5.1.2 Memória de Instruções

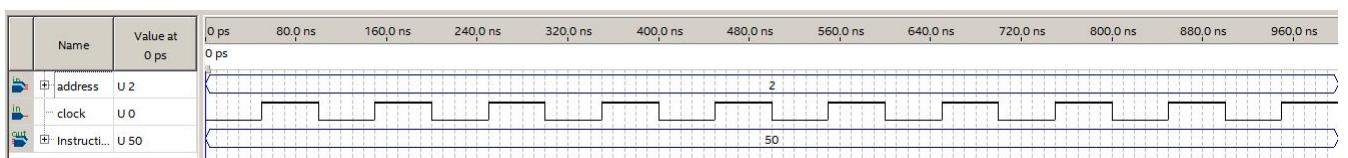
Para testar a Memória de Instruções, foram escolhidos dois endereços, o primeiro equivalente ao valor decimal de retorno 20 (referente a instrução *add*) e o segundo, ao valor 50 (referente a instrução *sub*). Ao selecionar qual seria o endereço, a cada subida de *clock* a saída referenciava o dado específico.

Figura 9 – *Waveform 1: Memória de Instruções*



Fonte: Autor

Figura 10 – *Waveform 2: Memória de Instruções*

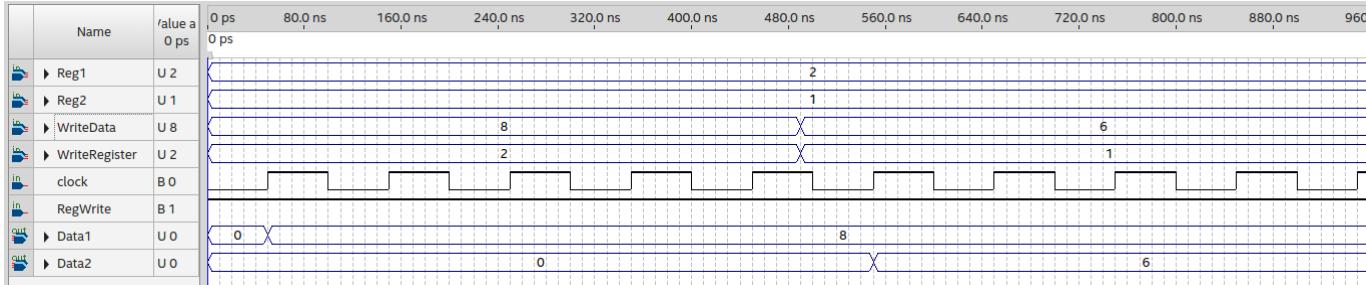


Fonte: Autor

### 5.1.3 Banco de Registradores

Para testar o banco de registradores, considerando que o sinal de controle **RegWrite** está ativo, significa que o dado de entrada da variável **WriteData** será salvo no vetor **Register**, na posição de memória dada pelo **WriteRegister**. No exemplo abaixo, foi salvo o valor 8 no vetor Register[2] e o valor 6 na posição Register[1].

Para o caso da leitura dos dados, as variáveis **Reg1** e **Reg2** indicavam qual endereço do dado do vetor Register, **Data1** e **Data2** iriam receber respectivamente. Para quesitos de simplificação, Reg1 e Reg2 receberam o mesmo valor do endereço dado pelo WriteRegister na escrita dos dados, para que dessa forma fosse possível conferir se os dados escritos, também poderiam ser lidos corretamente.

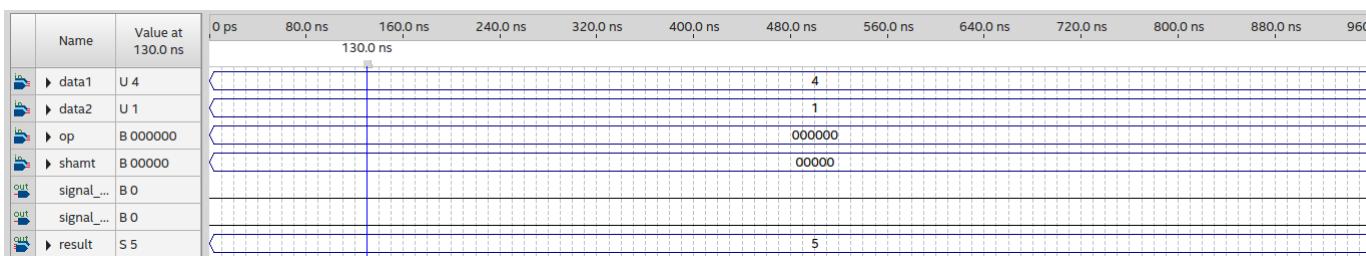
Figura 11 – *Waveform 1: Banco de Registradores*

Fonte: Autor

#### 5.1.4 Unidade Lógica e Aritmética

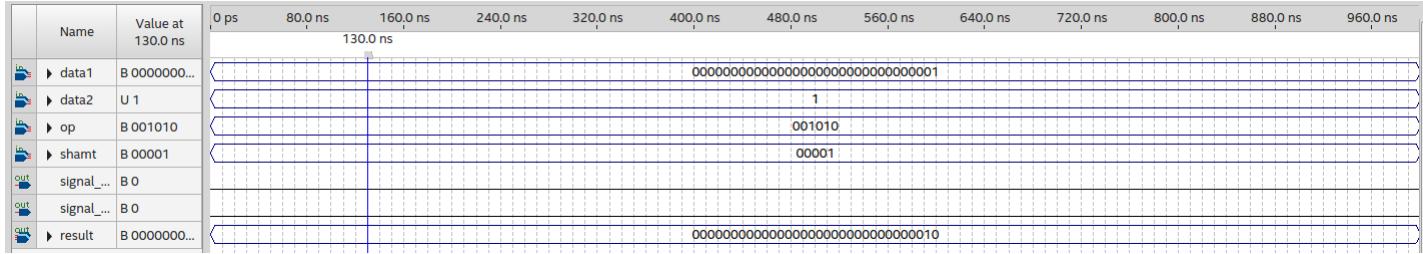
Para testar a Unidade Lógica e Aritmética, considerou-se três possíveis tipos de operações. Onde cada instrução era selecionada a partir de seu respectivo *opcode*.

A primeira, é referente a operação de soma. Ao selecionar esta operação a partir do *opcode* 00000, os valores das variáveis **Data1** e **Data2** eram somados e o resultado apresentou-se correto. Além disso, como o resultado da operação não é nulo e nem possui sinal negativo, suas respectivas *flags* não foram setadas.

Figura 12 – *Waveform 1: Unidade Lógica e Aritmética*

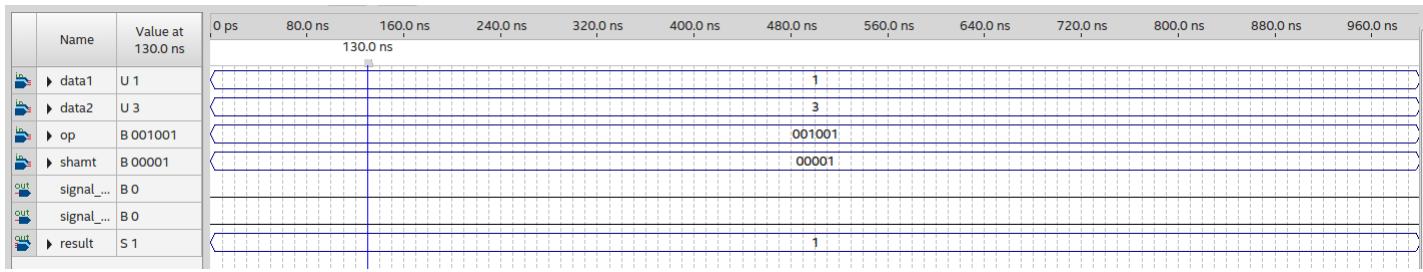
Fonte: Autor

A segunda operação refere-se a instrução *shift left* (shfl) que efetua um deslocamento de bits para a esquerda. A quantidade de bits a ser deslocada é dada pelo campo **shamt**. Portanto, o resultado apresentado foi do valor binário de **Data1** com um bit deslocado para a esquerda.

Figura 13 – *Waveform 2: Unidade Lógica e Aritmética*

Fonte: Autor

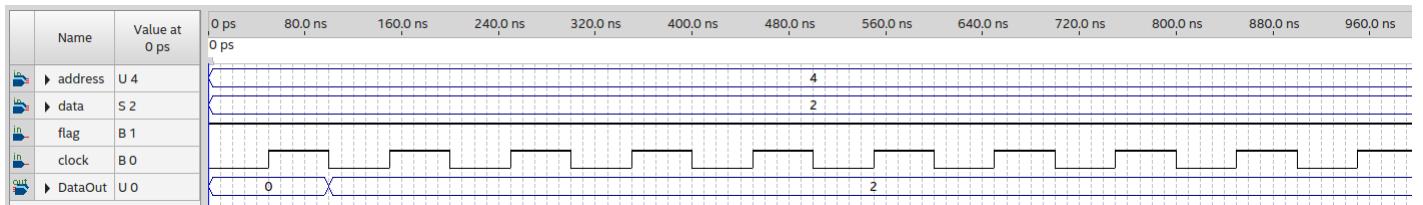
Finalizando, a terceira operação é a comparação *set on less than* (stl), que verifica se o valor **Data1** é menor que o de **Data2**. Como no exemplo da Figura 14, esta afirmação é verdadeira, o valor retornado é igual a 1.

Figura 14 – *Waveform 3: Unidade Lógica e Aritmética*

Fonte: Autor

### 5.1.5 Memória de Dados

Para testar a Memória de Dados, após a descida do *clock* o valor presente em **data** foi salvo na posição dada por **adress**. Além disso, **DataOut** indica qual dado foi salvo na memória.

Figura 15 – *Waveform 1: Memória de Dados*

Fonte: Autor

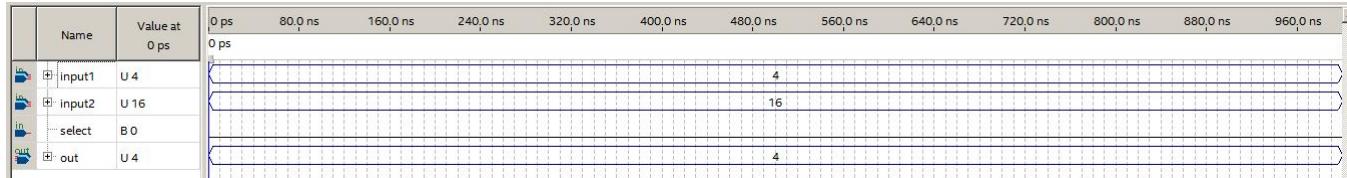
### 5.1.6 Unidade de Controle

Os testes do módulo referente a Unidade de Controle serão apresentados no Ponto de Checagem 3.

### 5.1.7 MUX e BigMux

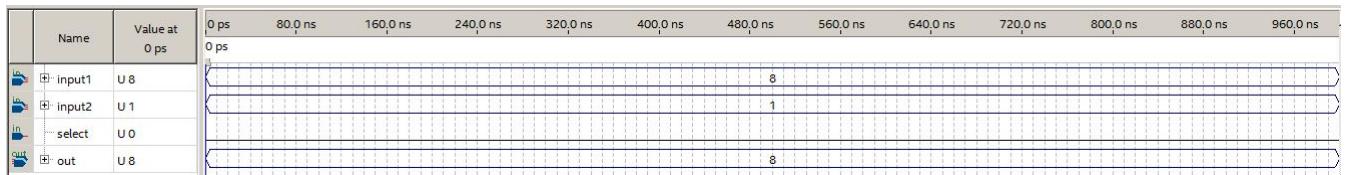
Inicialmente, o funcionamento de todos os multiplexadores do processador é o mesmo. Consistindo em um sinal seletor que determina qual será sua saída dada duas determinadas entradas.

Figura 16 – *Waveform 1: Mux de Entrada da ULA*



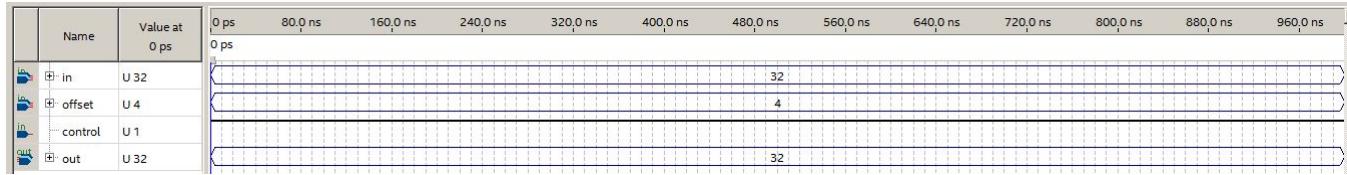
Fonte: Autor

Figura 17 – *Waveform 2: Mux de Saída da Memória de Instruções*



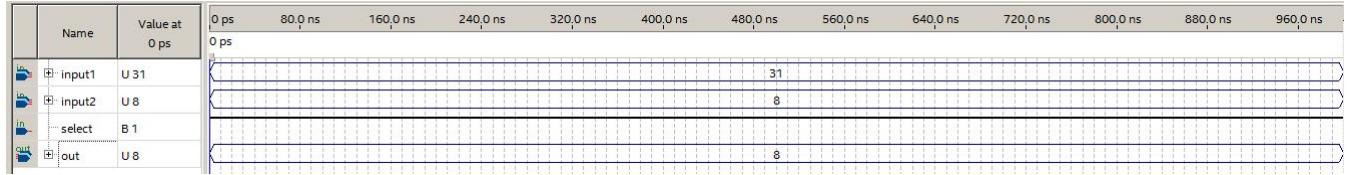
Fonte: Autor

Figura 18 – *Waveform 3: Mux de Entrada do Extensor de Bits*



Fonte: Autor

Figura 19 – *Waveform 4: Mux de Entrada do Banco de Registradores*

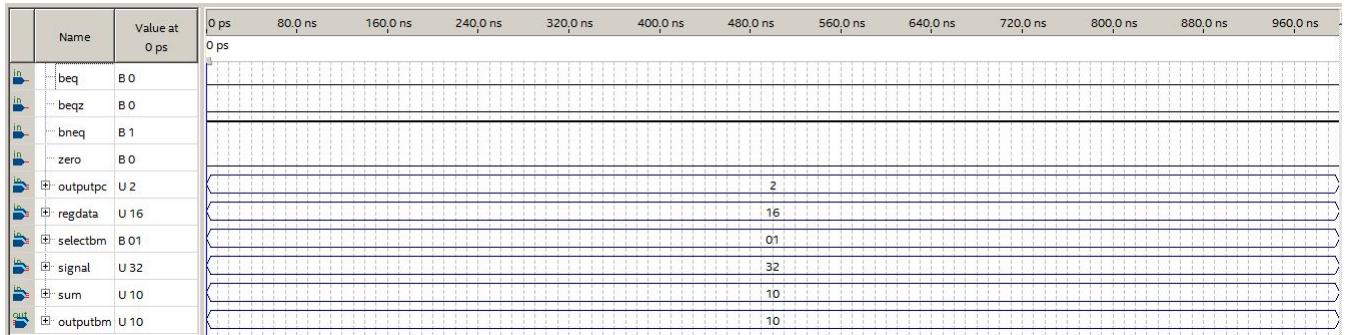


Fonte: Autor

O funcionamento do módulo BigMux é semelhante a todos os multiplexadores citados, sua única diferença é o tratamento das instruções do tipo *branch* e *jump*. Vale lembrar que este módulo é responsável por indicar qual será a entrada do PC.

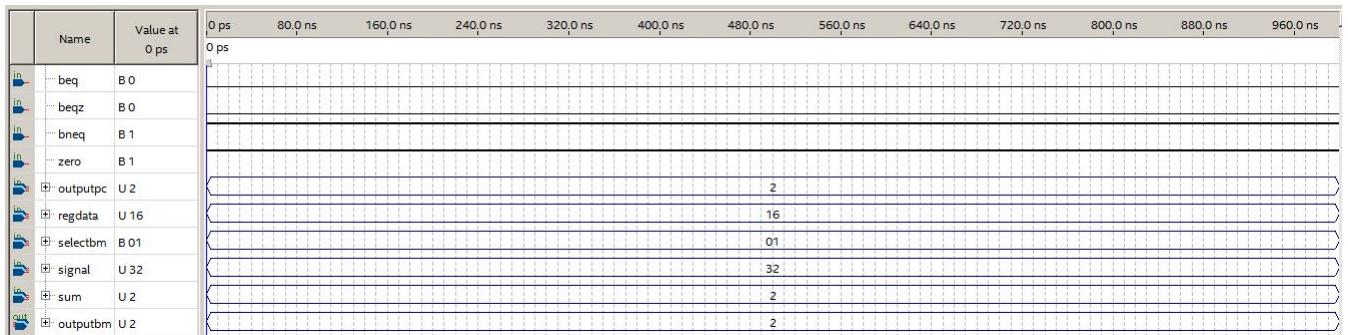
As Figuras 20 e 21 apresentam respectivamente os casos da instrução *banch on not equal*. Um sinal pronvindo da **selectbm** indica quando os casos de *branch* deverão ser considerados. Nos exemplos abaixo, quando **bneq** e **zero** são igual a 1, o sinal é o endereço de saída provindo do próprio PC ou se será o valor de saída da ULA.

Figura 20 – Waveform 5: BigMux (*bneq=1 e zero=0*)



Fonte: Autor

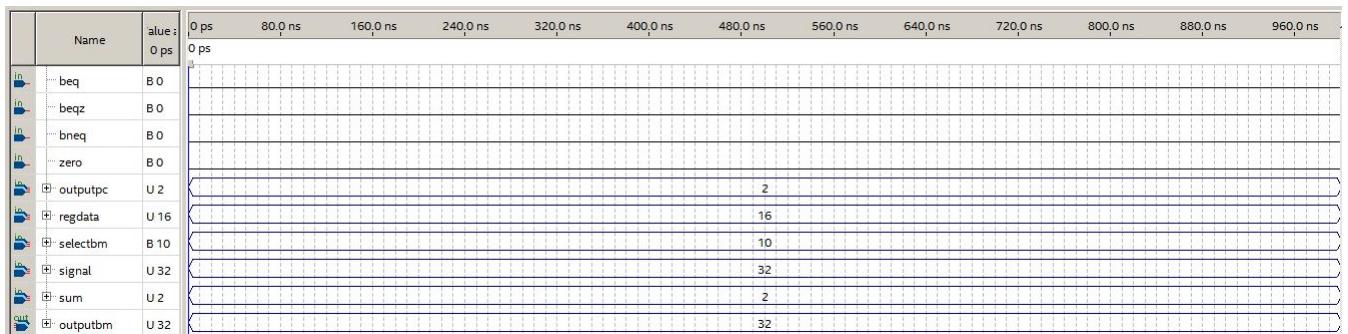
Figura 21 – Waveform 6: BigMux (*bneq=1 e zero=1*)



Fonte: Autor

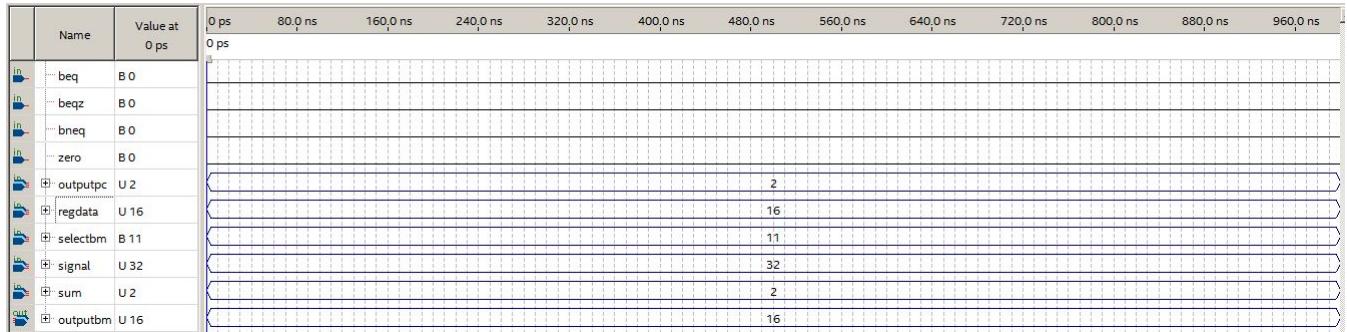
No caso das instruções de *jump* e *jump register*, a diferenciação está no dado de saída, sendo representado por **signal** e **regdata** respectivamente.

Figura 22 – Waveform 7: BigMux (*jump com saído dada por signal*)



Fonte: Autor

Figura 23 – Waveform 8: BigMux (jump register com saído dada por regdata)

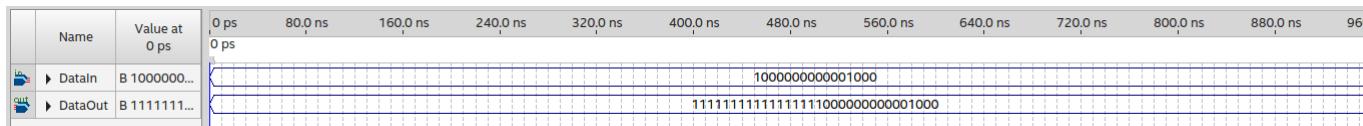


Fonte: Autor

### 5.1.8 Extensor

O ultimo módulo é o Extensor de *Bits*, que no exemplo abaixo representa uma entrada de 16 *bits* com *bit* mais significativo igual a 1.

Figura 24 – Waveform 1: BigMux (jump register com saído dada por regdata)



Fonte: Autor

## 5.2 Módulos unificados

Após a implementação e testes de cada módulo da arquitetura-base, juntou-se todas as partes, para que as mesmas funcionassem de forma unificada.

```

1 module Processador (clockReal, adressIn,
2                                     RegWrite, WriteData, WriteRegister,
3                                     Data1, Data2, RegOut1, RegOut2,
4                                     MUXULAOOutput, MUXULASelect,
5                                     ULAcode, ULAout, zero,
6                                     WriteFlag, DataMemOutput,
7                                     MUXDMSelect, MUXDMOutput
8 );
9
10 //PC
11     input clockReal;
12     wire reset, hlt, zero;
13     wire jump, jr, pcb;
14     input wire [9:0] adressIn;
15     wire [9:0] addressOut;
16
17     //Instruction Memory
18     wire [31:0] Instruction;
19
20     //MuxBank1

```

```
20      wire MuxBankSelect;
21      wire [4:0] MuxBankOut;
22
23      //MuxBitExtender
24
25      wire MUXBEControl;
26      wire [15:0] MUXBEOut;
27      wire [15:0] switches;
28
29      //BitExtender16
30      wire [31:0] BitExtender16_32;
31
32      //BitExtender26
33      wire [31:0] BitExtender26_32;
34
35      //Register Bank
36      input [4:0] WriteRegister;
37      input [31:0] WriteData;
38      input RegWrite;
39      output [31:0] Data1, Data2;
40      output [4:0] RegOut1, RegOut2;
41
42      //MUXULA
43      output [31:0] MUXULAOOutput;
44      input MUXULASelect;
45
46      //ULA
47      input [4:0] ULAcode;
48      output [31:0] ULAout;
49      output zero;
50
51      //Data Memory
52      input WriteFlag;
53      output [31:0] DataMemOutput;
54
55      //MUXMemoria de Dados
56
57      input MUXDMSelect;
58      output [31:0] MUXDMOutput;
59
60
61
62 Program_Counter Program_Counter (
63
64
65
66
67
68
69
70
71
72
73
74
75      MemoryInstruction InstrMem (
76
77
78
79      .clock(clockReal),
      .adress(adressIn),
      .reset(reset),
      .hlt(hlt),
      .pgcount(addressOut),
      .zero(out),
      .jump(jump),
      .jr(jr),
      .PCBranch(pcb)
    );
```

```
80  MuxBank1 MuxBR (
81          .input1(Instruction[20:16]),
82          .input2(Instruction[15:11]),
83          .out(MuxBankOut),
84          .select(MuxBankSelect));
85
86  MuxBitExtender MUXBE (.control(MUXBEControl),
87                      .offset(Instruction[15:0]),
88                      .out(MUXBEOut),
89                      .in(switches));
90
91  BitExtender BitExtender32(
92          .DataIn(MUXBEOut),
93          .DataOut(BitExtender16_32)
94      );
95
96  RegBank RegBank (.clock(clockReal),
97                  .Reg1(Instruction[25:21]),
98                  .Reg2(Instruction[20:16]),
99                  .RegWrite(RegWrite),
100                 .WriteData(WriteData),
101                 .WriteRegister(MuxBankOut),
102                 .Data1(Data1),
103                 .Data2(Data2),
104                 .RegOut1(RegOut1),
105                 .RegOut2(RegOut2)
106             );
107
108
109
110
111
112  BitExtender26 BitExtender26 (
113          .DataIn(Instruction[25:0]),
114          .DataOut(BitExtender26_32)
115      );
116
117
118
119  MuxULA MUXULA(.input1(Data2),
120                  .input2(BitExtender16_32),
121                  .out(MUXULAOOutput),
122                  .select(MUXULASelect));
123
124
125  ULA ULA (.op(ULACode),
126              .data1(Data1),
127              .data2(MUXULAOOutput),
128              .result(ULAout),
129              .signal_zero(zero),
130              .shamt(Instruction[10:6])
131      );
132
133
134
135  BigMux BIGMUX(.zero(zero),
136                  .beq(beq) ,
137                  .bneq(bneq),
138                  .beqz(beqz),
139                  .selectbm(BIGMUXSelect),
```

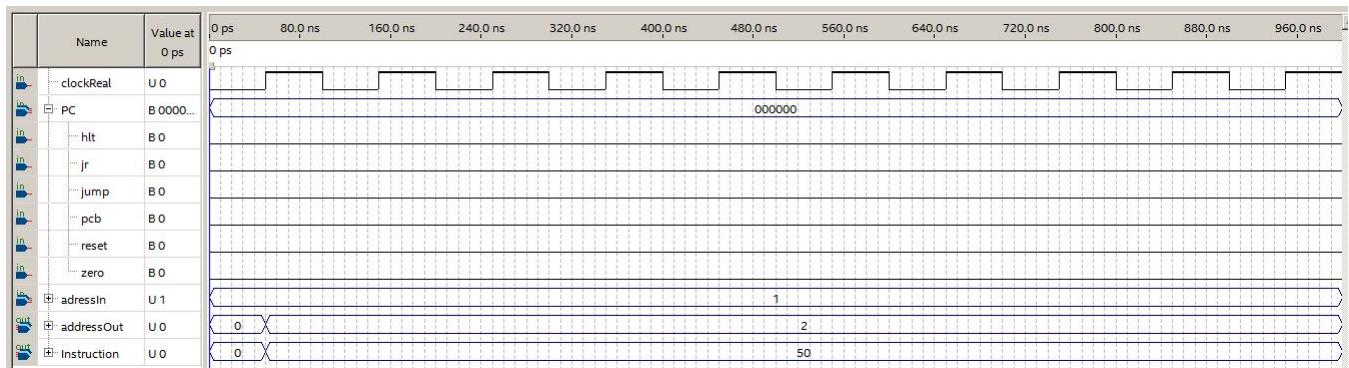
```

140                               .outputbm(addressIn),
141                               .outputpc(addressOut),
142                               .sum(BitExtender16_32),
143                               .signal(BitExtender26_32),
144                               .regdata(Data1) );
145
146 DataMemory DataMemory(
147                               .clock(clockReal),
148                               .address(ULAout),
149                               .flag(WriteFlag),
150                               .DataOut(DataMemOutput),
151                               .data(Data2)
152 );
153
154 MuxDataMem MUXDM(
155                               .input1(ULAout),
156                               .input2(DataMemOutput),
157                               .out(MUXDMOutput),
158                               .select(MUXDMSel)
159 );
160
161
162
163
164
165
166
167
168
169 endmodule

```

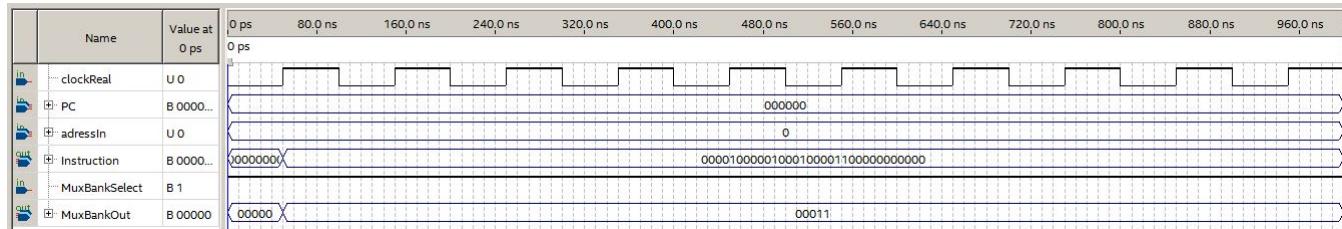
Apesar da implementação em *Verilog*, é possível comparar esta etapa em conectar fios em cada módulo, para que a entrada de um fosse a saída do outro e que tudo estivesse interconectado. Para facilitar o processo de implementação, cada módulo foi testado na ordem apresentada pela arquitetura base, começando pelo PC e indo até o *BigMux*. E o número de componentes conectados crescia módulo a módulo.

Figura 25 – *PC/Instruction Memory*



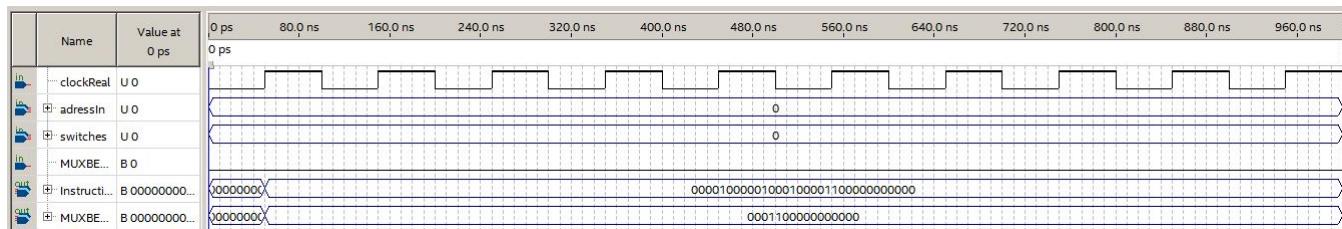
Fonte: Autor

Figura 26 – PC/Instruction Memory/Mux pré Banco de Registradores



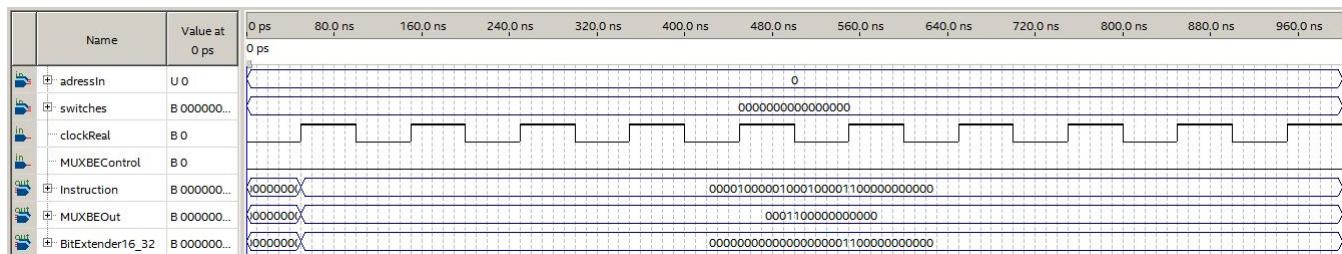
Fonte: Autor

Figura 27 – Módulos anteriores/Mux pré Bit Extender



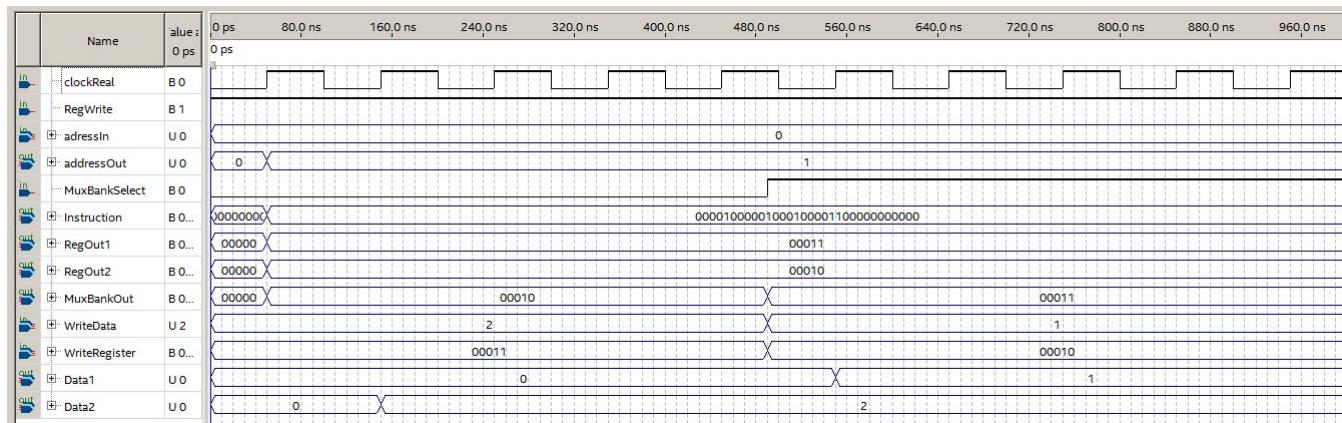
Fonte: Autor

Figura 28 – Módulos anteriores/Mux pré Bit Extender/Bit Extender



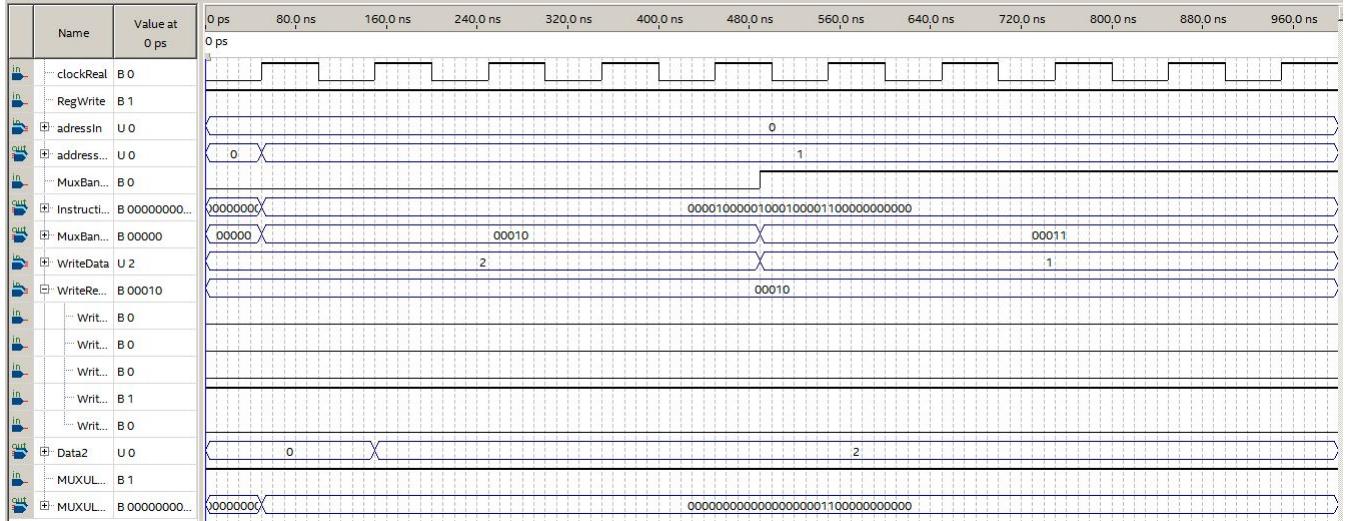
Fonte: Autor

Figura 29 – Módulos anteriores/Banco de Registradores



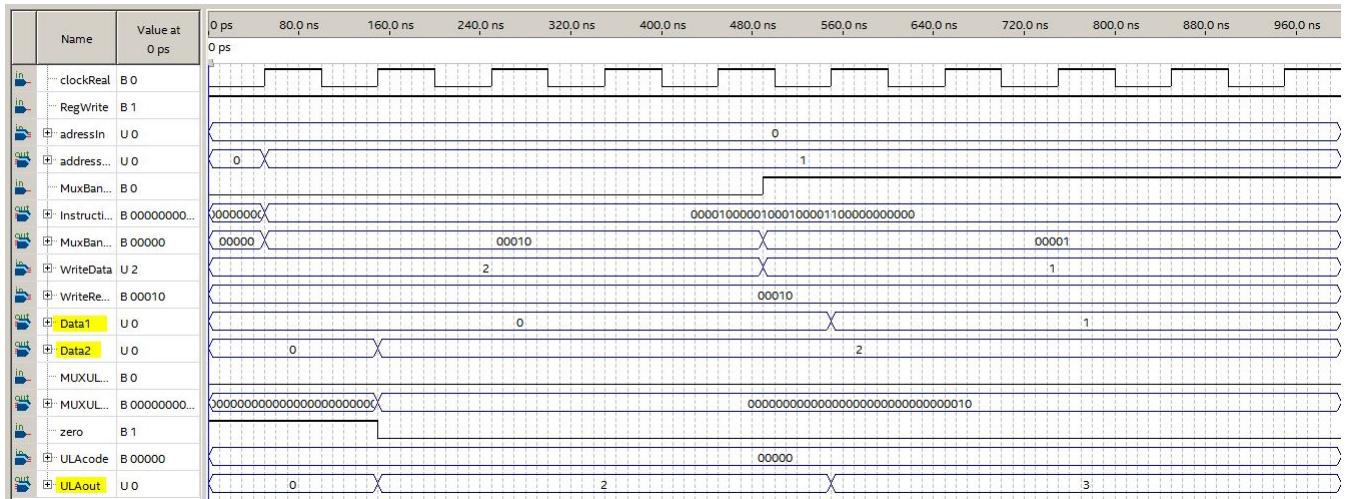
Fonte: Autor

Figura 30 – Módulos anteriores/Banco de Registradores/Mux pré ULA



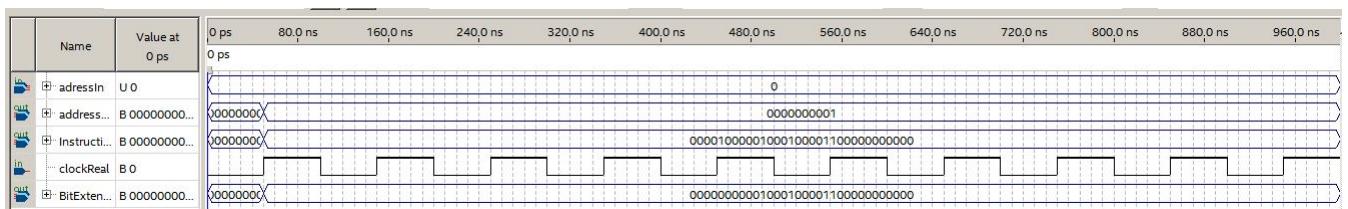
Fonte: Autor

Figura 31 – Módulos anteriores/ULA (instrução add)



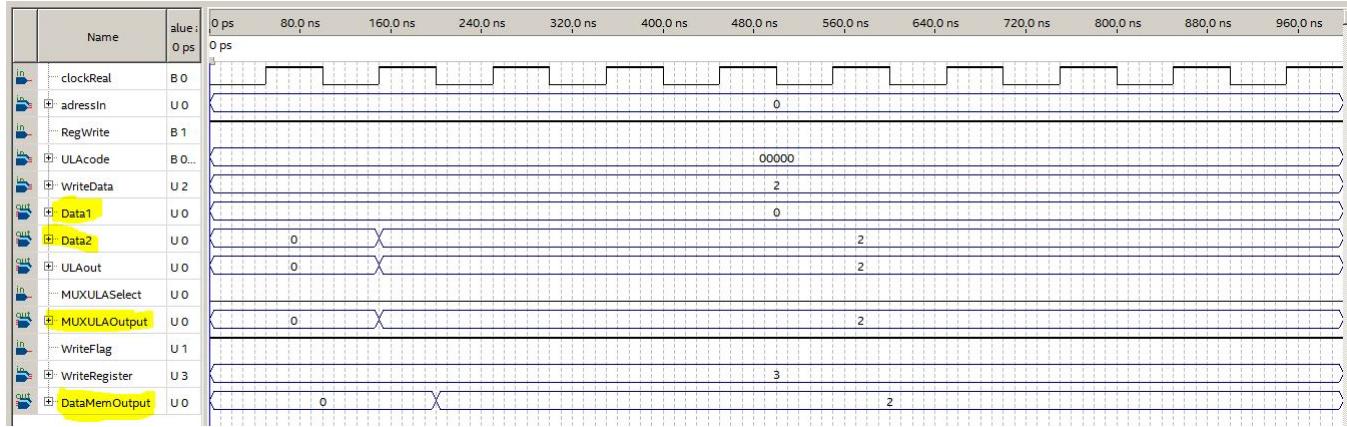
Fonte: Autor

Figura 32 – Módulos anteriores/ULA/Bit Extender (26 para 32 bits)



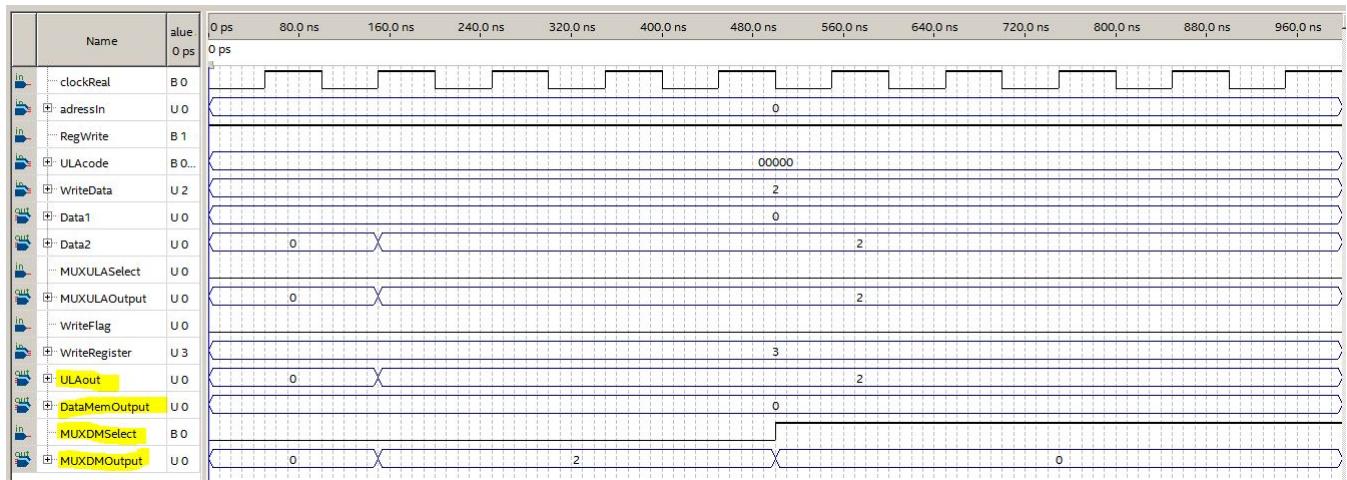
Fonte: Autor

Figura 33 – Módulos anteriores/Memória de Dados



Fonte: Autor

Figura 34 – Módulos anteriores/Memória de Dados/Mux pós Memória de Dados



Fonte: Autor



## 6 Considerações Finais

Analizando as etapas anteriores deste relatório, é possível perceber que desenvolver um sistema computacional pode ser algo complexo. E para esta arquitetura não será diferente.

Durante o desenvolvimento deste relatório, a principal dificuldade encontrada foi interligar todos os módulos, principalmente e relação aos testes, visto que conforme o número de módulos já conectados aumentavam, maior era o número de variáveis necessárias para verificar. Requisitando extrema atenção em cada ponto das formas de onde geradas

Os próximos passos para o desenvolvimento deste projeto será iniciar o processo de implementação da Unidade de Controle (considerada o cérebro do processador) e também a finalização do módulo de entrada e saída.

Após o desenvolvimento e testes de cada componente, será feito a junção final de todos os módulos, interligando cada bloco já desenvolvido e garantindo seu total funcionamento em conjunto. Com isso, o programa será compilado em uma placa FPGA (circuito integrado que possui componentes programados) e a funcionalidade será comprovada através da execução de alguns algoritmos.



# Referências

- 1 MAXIEDUCA. *Descubra a importância da memória cache para o computador*. 2017. Disponível em: <<http://blog.maxieduca.com.br/memoria-cache-computador/>>. Acesso em: 13/04/2018. Citado na página 11.
- 2 MACEDO, D. *Arquitetura: Von Neumann Vs Harvard*. 2012. Disponível em: <<http://www.diegomacedo.com.br/arquitetura-von-neumann-vs-harvard/>>. Acesso em: 13/04/2018. Citado 2 vezes nas páginas 12 e 13.
- 3 PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design*. 5th edition. ed. Waltham/MA, EUA: Morgan Kaufmann, 2007. Citado na página 16.
- 4 COMPUTER Hardware/Software Architecture. [S.l.]: Morgan Kaufmann, 1998. Citado na página 21.
- 5 UFMT. *Circuito Combinacional*. 2016. Disponível em: <[http://araguaia2.ufmt.br/professor/disciplina\\_arquivo/90/201608301202.pdf](http://araguaia2.ufmt.br/professor/disciplina_arquivo/90/201608301202.pdf)>. Acesso em: 13/04/2018. Citado na página 24.
- 6 MARTINO, P. J. M. D. *Interface de Entrada e Saída*. 2004. Disponível em: <[http://www.dca.fee.unicamp.br/courses/EA078/1s2004/arquivos/turma\\_ab/cap7.pdf](http://www.dca.fee.unicamp.br/courses/EA078/1s2004/arquivos/turma_ab/cap7.pdf)>. Acesso em: 10/04/2018. Citado na página 27.
- 7 GEEKS, G. for. *I/O Interface (Interrupt and DMA Mode)*. 2016. Disponível em: <<https://www.geeksforgeeks.org/io-interface-interrupt-dma-mode/>>. Acesso em: 11/04/2018. Citado na página 27.
- 8 CENTODUCATTE, P. C. *Conjunto de Instruções MIPS*. 2002. Disponível em: <[http://www.ic.unicamp.br/~pannain/mc542/aulas/ch3\\_arq.pdf](http://www.ic.unicamp.br/~pannain/mc542/aulas/ch3_arq.pdf)>. Acesso em: 08/04/2018. Citado na página 30.