

Bruno Bernardo de Moura

# **Laboratório de Arquitetura e Organização de Computadores: Desenvolvimento de um Sistema Computacional**

São José dos Campos - Brasil

Julho de 2017

Bruno Bernardo de Moura

# **Laboratório de Arquitetura e Organização de Computadores: Desenvolvimento de um Sistema Computacional**

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

Julho de 2017

# Resumo

O relatório em questão apresenta detalhes a respeito da implementação e do desenvolvimento de um sistema computacional com arquitetura baseada na arquitetura *MIPS* utilizando da linguagem de descrição de *hardware* Verilog. Durante o decorrer deste relatório serão expostos detalhes à respeito do conjunto de instruções, caminho de dados e funcionamento em geral do sistema em questão, abordando todas as suas unidades e componentes.

**Palavras-chaves:** Unidade de Processamento, *MIPS*, Verilog.

# Listas de ilustrações

Figura 1 – Hierarquia de Memória . . . . .	8
Figura 2 – Caminho de Dados - MIPS . . . . .	12
Figura 3 – Caminho de Dados . . . . .	18
Figura 4 – <i>Waveform 1</i> : Contador de Programa . . . . .	31
Figura 5 – <i>Waveform 2</i> : Memória de Instruções . . . . .	31
Figura 6 – <i>Waveform 3</i> : Banco de Registradores e Memória de Dados . . . . .	32
Figura 7 – <i>Waveform 4</i> : <i>BigMux</i> . . . . .	32
Figura 8 – <i>Waveform 5</i> : Simulação Final . . . . .	33
Figura 9 – <i>Waveform 6</i> : add . . . . .	35
Figura 10 – <i>Waveform 7</i> : subi . . . . .	36
Figura 11 – <i>Waveform 8</i> : inc/dec . . . . .	36
Figura 12 – <i>Waveform 9</i> : slt . . . . .	37
Figura 13 – <i>Waveform 10</i> : shfr/shfl . . . . .	37
Figura 14 – <i>Waveform 11</i> : not, and, or e xor . . . . .	38
Figura 15 – <i>Waveform 12</i> : mult/div . . . . .	38
Figura 16 – <i>Waveform 13</i> : instruções remanescentes . . . . .	39
Figura 17 – FPGA - Identificação . . . . .	40
Figura 18 – Fibonacci na FPGA - 1 . . . . .	43
Figura 19 – Fibonacci na FPGA - 2 . . . . .	44
Figura 20 – Fibonacci na FPGA - 3 . . . . .	44
Figura 21 – FPGA - Acesso à memória . . . . .	46

# **Lista de tabelas**

Tabela 1 – Formato R de Instrução . . . . .	11
Tabela 2 – Formato I de Instrução . . . . .	11
Tabela 3 – Formato J de Instrução . . . . .	11
Tabela 4 – Conjunto de Instruções . . . . .	17
Tabela 5 – Conjunto de Operações da ALU . . . . .	23
Tabela 6 – Algoritmo Teste . . . . .	33
Tabela 7 – Sinais de Controle . . . . .	34
Tabela 8 – Pseudocódigo - algoritmo teste . . . . .	39
Tabela 9 – Fibonacci - Pseudocódigo . . . . .	42
Tabela 10 – Acesso à memória - Pseudocódigo . . . . .	45

# Sumário

<b>1</b>	<b>INTRODUÇÃO</b>	<b>7</b>
<b>1.1</b>	<b>Motivação</b>	<b>7</b>
<b>1.2</b>	<b>Objetivos</b>	<b>7</b>
<b>2</b>	<b>FUNDAMENTAÇÃO TEÓRICA</b>	<b>8</b>
<b>2.1</b>	<b>Hierarquia de Memória</b>	<b>8</b>
<b>2.2</b>	<b>RISC vs CISC</b>	<b>9</b>
<b>2.3</b>	<b>MIPS</b>	<b>10</b>
2.3.1	Tipos de Instruções	10
2.3.2	<i>Datapath</i>	12
2.3.3	Contador de Programa	12
2.3.4	Memória de Instruções	13
2.3.5	Banco de Registradores	13
2.3.6	Unidade Lógica e Aritmética	13
2.3.7	Memória de Dados	14
2.3.8	Unidade de Controle	14
2.3.9	Auxiliares	14
<b>2.4</b>	<b>Verilog</b>	<b>14</b>
<b>2.5</b>	<b>FPGA</b>	<b>15</b>
<b>3</b>	<b>DESENVOLVIMENTO</b>	<b>16</b>
<b>3.1</b>	<b>Conjunto de Instruções</b>	<b>16</b>
3.1.1	Formatos de Instrução	16
3.1.2	Modos de Endereçamento	17
<b>3.2</b>	<b>Arquitetura-base do Processador</b>	<b>18</b>
<b>3.3</b>	<b>Contador de Programa</b>	<b>18</b>
<b>3.4</b>	<b>Memória de Instruções</b>	<b>20</b>
<b>3.5</b>	<b>Banco de Registradores</b>	<b>21</b>
<b>3.6</b>	<b>Unidade Lógica e Aritmética</b>	<b>23</b>
<b>3.7</b>	<b>Memória de Dados</b>	<b>25</b>
<b>3.8</b>	<b><i>BigMux</i></b>	<b>27</b>
<b>3.9</b>	<b>Unidade de Controle</b>	<b>28</b>
<b>4</b>	<b>RESULTADOS OBTIDOS E DISCUSSÕES</b>	<b>30</b>
<b>4.1</b>	<b>Simulações Individuais - Parte 1</b>	<b>30</b>
4.1.1	Contador de Programa	30

4.1.2	Memória de Instruções . . . . .	31
4.1.3	Banco de Registradores e Memória de Dados . . . . .	31
4.1.4	<i>BigMux</i> . . . . .	32
<b>4.2</b>	<b>Simulação Total (Unidade de Processamento) - Parte 1</b> . . . . .	<b>32</b>
<b>4.3</b>	<b>Simulação Total (Outras instruções) - Parte 1</b> . . . . .	<b>35</b>
4.3.1	Adição - (add) . . . . .	35
4.3.2	Subtração com imediato - (subi) . . . . .	36
4.3.3	Incremento e decremento - (inc/dec) . . . . .	36
4.3.4	<i>Set on less than</i> - (slt) . . . . .	37
4.3.5	Deslocamentos - (shfr/shfr) . . . . .	37
4.3.6	Operações lógicas - (not, and, or e xor) . . . . .	38
4.3.7	Multiplicação e divisão - (mult/div) . . . . .	38
4.3.8	Instruções remanescentes - (ldi, beq, jmpr) . . . . .	39
<b>4.4</b>	<b>Simulação Total (UP, UC e FPGA) - Parte 2</b> . . . . .	<b>40</b>
4.4.1	Algoritmo 1 - Fibonacci . . . . .	41
4.4.2	Algoritmo 2 - Funções de acesso à memória . . . . .	45
<b>5</b>	<b>CONSIDERAÇÕES FINAIS</b> . . . . .	<b>47</b>
 <b>REFERÊNCIAS</b> . . . . .		<b>48</b>
 <b>APÊNDICES</b> . . . . .		<b>49</b>
 <b>APÊNDICE A – APÊNDICE 1</b> . . . . .		<b>50</b>

# 1 Introdução

## 1.1 Motivação

Cada vez mais complexos e aptos a realizarem mais tarefas, computadores são algo que vem sendo altamente importante para o desenvolvimento de novas tecnologias e para o avanço da sociedade em quase todos os seus aspectos.

Por mais que quase todas as pessoas que viveram dentro dos últimos 20 ou 30 anos saibam de pronto dizer o que é um computador, define-se um, segundo (1), como uma máquina composta de um número variável de unidades especializadas, comandadas por um mesmo programa gravado, que, sem interveção humana direta, permite efetuar complexas operações aritméticas e lógicas com fins estatísticos, administrativos, contabilísticos.

Tendo em mente a definição acima, é fácil de se imaginar que computadores de fato tenham um funcionamento tanto em alto quanto em baixo nível bastante complexo. Para que a execução de suas tarefas se dê de maneira adequada sem que falhas significativas venham a ocorrer, o computador necessita de um conjunto de instrução, também chamado de ISA (*Instruction Set Architecture*). Esse conjunto é tratado pela UP (Unidade de Processamento), que é responsável por sanar todas as possíveis necessidades de um programa a ser executado pelo computador, sejam elas acesso à memória, operações lógicas e aritméticas, operações de entrada e saída de dados, etc. Contudo, uma outra unidade é de suma importância para o funcionamento da máquina: a Unidade de Controle.

A presença de um componente no computador responsável por agir como um cérebro, coordenando e auxiliando os outros componentes é esperada, dado o grau de complexidade do seu funcionamento. Esse é o papel da UC (Unidade de Controle): controlar a execução de uma instrução durante todas as suas etapas de forma que todos os componentes presentes na UP operem corretamente.

Sendo assim, o projeto apresentado neste relatório será apresentado na ordem citada acima: Arquitetura do Conjunto de Instruções, Unidade de Processamento e Unidade de Controle (ou Caminho de Dados), e por fim a interação entre tais unidades.

## 1.2 Objetivos

O objetivo deste projeto é a criação de um sistema computacional capaz de realizar operações de maneira similar a um processador baseado na arquitetura MIPS, testando cada uma de suas possíveis instruções e garantindo o seu funcionamento para que ao seu término seja possível se realizar sua simulação com elementos de *hardware*.

## 2 Fundamentação Teórica

### 2.1 Hierarquia de Memória

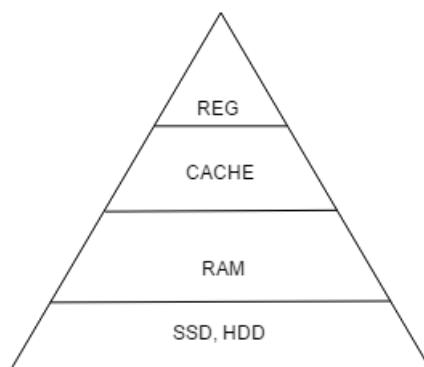
Dada a complexidade das operações realizadas por um sistema computacional, é de se esperar que dentro dele existam diferentes níveis de armazenamentos de dados. O conceito utilizado para a criação de tais níveis, de acordo com, (2) é o chamado de hierarquia de memória.

Esse conceito basicamente representa a divisão das memórias de um sistema computacional em quatro níveis: registradores, memória CACHE, memória RAM e HD.

De baixo para cima primeiramente se encontram dispositivos de armazenamento, tais como HD's ou DVD's, responsáveis por armazenar dados que necessitam ser buscados antes de utilizados, pois possuem baixa velocidade de acesso. Logo acima se encontra a memória RAM (*Random Access Memory*), ou memória de acesso aleatório, que, possuindo uma alta velocidade de acesso, mantém consigo dados que abrangem os que atualmente estão sendo executados pelo processador. Na penúltima posição, logo antes do topo, se encontra a memória CACHE, que, sendo subdividida em níveis e por possuir uma alta velocidade de acesso, intermedia a troca rápida de informações entre os registradores e a memória RAM, organizando e selecionando os blocos de memória que contêm os dados mais pertinentes ao atual processo sendo executado. Por fim, no topo da hierarquia se encontram os registradores, que são as unidades de memória diretamente processadas pelas instruções atualmente sendo executadas pelo sistema.

Visualmente, o conceito de hierarquia de memória pode ser representado como visto na Figura 1:

Figura 1 – Hierarquia de Memória



Fonte: Autor

Cada nível da hierarquia de memória citada possui diferentes características, tais como custo e velocidade de acesso de dados e capacidade de armazenamento, de forma que, conforme a velocidade cresce em direção ao topo, com ela também cresce o custo para o acesso e se diminui a capacidade de armazenamento de tal nível. Consequentemente, quanto mais baixo o nível da memória na hierarquia, maior sua capacidade e menores são seu custo e velocidade de acesso.

## 2.2 RISC vs CISC

Ao se estudar mais a baixo nível o funcionamento de um sistema computacional como o a ser desenvolvido neste trabalho existem teoricamente dois tipos de arquiteturas de conjunto de instruções que podem ser escolhidos; a arquitetura *RISC* e a *CISC*.

A arquitetura *CISC* (*Complex Instruction Set Computers*), fazendo jus ao nome, é uma arquitetura composta por centenas de instruções diferentes, sendo elas simples ou complexas.

Essa arquitetura tem como objetivo completar uma tarefa no menor número de linhas de codificação *assembly* possível. Para isso ela possui características únicas que a diferenciam das outras arquiteturas, tais como:

1. Formatos de instrução de tamanhos variados;
2. Grande número de modos de endereçamento;
3. Ênfase em *hardware*;
4. Possui em seu ISA instruções que levam mais de um ciclo de *clock* para serem realizadas;

Em oposição à arquitetura *CISC* citada, existe a arquitetura *RISC* (*Reduced Instruction Set Computers*), que utiliza um número menor de, e mais simples instruções em prol do desempenho e da rapidez na execução das mesmas.

Essa arquitetura requer um trabalho maior do programador, uma vez que ela exige mais linhas em codificação *assembly* para completar determinada tarefa, contudo, suas instruções reduzidas requerem menos espaço em *hardware*, o que possibilita mais espaços para a criação de registradores de propósito geral. Algumas das principais características da arquitetura *RISC* são:

1. Formato fixo de instruções;
2. Número reduzido de instruções;
3. Ênfase em *software*;
4. Número de ciclos de *clock* por instrução reduzido;

5. Pequeno número de modos de endereçamento;
6. Utilização de registradores de propósito geral;

Vale ressaltar que hoje em dia no mercado não existem arquiteturas totalmente *RISC* nem totalmente *CISC*. Todas as empresas do ramo utilizam uma combinação de características de ambas as arquiteturas para a implementação do conjunto de instruções de seu produto, visando um melhor desempenho durante a execução de suas instruções.

## 2.3 MIPS

Responsável pelo funcionamento geral e da distribuição e operação de dados dentro de um aparelho, o processador age como um cérebro do circuito como um todo, de forma que seu *design* e sua implementação possam ser muitos complexos. No quesito do desenvolvimento de um processador ou de um projeto com circuits digitais, é comum se falar a respeito de sua arquitetura, que segundo (3), nada mais é do que seu comportamento funcional, como a decisão do tamanho de seus dados em *bits* ou seu número total de instruções.

No decorrer da história do desenvolvimento de sistemas computacionais foram criados diversos tipos de arquiteturas, dentre elas a arquitetura *MIPS* (4). Essa arquitetura foi desenvolvida nos anos 80 a partir da pesquisa em organização de computadores da Universidade de *Stanford* e segue características do padrão de arquiteturas *RISC*, ou seja, possui um número reduzido de instruções e também registradores de propósito geral. Vale ressaltar algumas das principais características dessa arquitetura uma vez que ela foi a utilizada para embasamento no desenvolvimento do projeto em questão:

1. Tamanho fixo do conjunto de instruções;
2. Apenas instruções *load* e *store* acessam a memória;
3. Possui execução de cinco estágios: busca, decodificação, execução, acesso à memória e escrita de dados;

### 2.3.1 Tipos de Instruções

Devido ao fato de possuir diversos tipos de instruções que realizam diversas operações, a arquitetura *MIPS* as dividiu em três tipos: R, I e J.

O primeiro tipo, R, é o tipo no qual se encontram todas as instruções que realizam operações lógicas ou aritméticas com os dados dos registradores, tais como somas, subtrações, deslocamentos e afins (2). Devido a isso são divididas em *opcode*, RS, RT, RD, *shamt* e *funct*. O *opcode*, presente em todas as instruções da arquitetura *MIPS*, é o responsável por diferenciar cada uma das instruções das outras, de forma que cada uma possua o

seu *opcode* específico; RS e RT representam os endereços do banco de registradores dos quais serão retirados dados; RD possui o valor do endereço no qual serão escritos os dados resultantes da operação; *shamt* representa a quantidade de *bits* a serem deslocados caso seja necessário e por fim *funct* é utilizado para a diferenciação de instruções na ULA (Unidade Lógica Aritmética), outro componente do processador que será abordado mais à frente neste capítulo.

Tabela 1 – Formato R de Instrução

Tamanho (bits)	6	5	5	5	5	6
Campo	opcode	RS	RT	RD	shamt	funct
Bits	31 - 26	25 - 21	20 - 16	15 - 11	10 - 6	5 - 0

Fonte: Autor

O segundo tipo de instrução é o tipo I, no qual se encontram todas as instruções que realizam operações lógicas e aritméticas com dados de *offset*, vindos do campo imediato, executam saltos condicionais e realizam acesso à memória. Neste caso a instrução é dividida, além do *opcode*, nos campos RS, RT e *offset*. O campo RS é o responsável por mostrar o valor do endereço do banco de registradores do qual será lido o dado a ser utilizado; RT representa o endereço do mesmo banco no qual será escrito o possível resultado da instrução e por fim o campo *offset*, ou imediato, carrega consigo um valor codificado que será usado diretamente na instrução, seja para possíveis saltos condicionais ou para operações aritméticas.

Tabela 2 – Formato I de Instrução

Tamanho (bits)	6	5	5	11
Campo	opcode	RS	RT	offset
Bits	31 - 26	25 - 21	20 - 16	15 - 0

Fonte: Autor

O terceiro e último tipo de instrução presente na arquitetura *MIPS* é o tipo J, responsável pelas instruções de saltos. Devido a isso sua divisão é a mais simples das instruções: *opcode* representa a identificação da instrução em si e todos os outros 26 *bits* são destinados ao endereço para o qual se quer saltar, como pode ser visto abaixo.

Tabela 3 – Formato J de Instrução

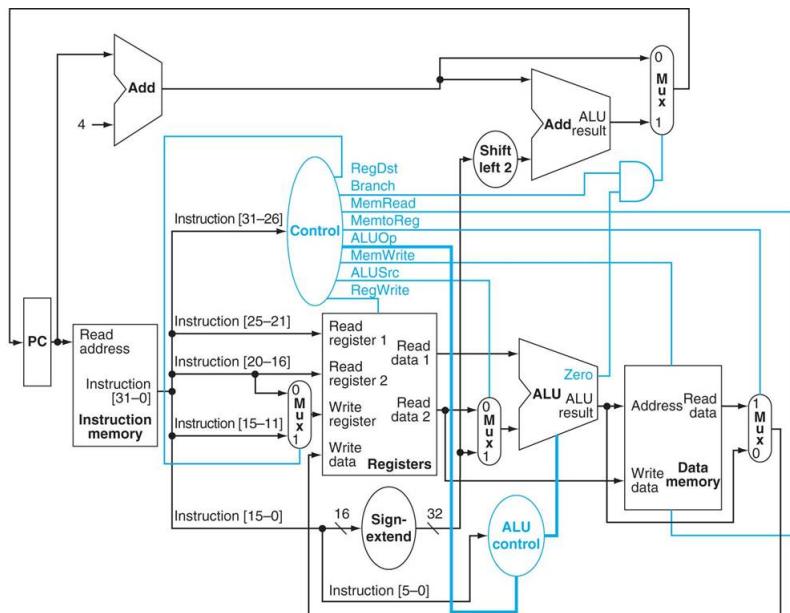
Tamanho (bits)	6	26
Campo	opcode	address
Bits	31 - 26	25 - 0

Fonte: Autor

### 2.3.2 Datapath

O *Datapath*, ou caminho de dados, consiste literalmente no caminho tomado pelos *bits* processados pelo processador em determinada instrução. Em outras palavras, o *Datapath* representa o total de componentes presentes em determinada arquitetura. Na arquitetura *MIPS*, para a realização de suas já citadas cinco etapas de execução de uma instrução, o *Datapath* consiste de uma série de componentes interligados que, uma vez sobre a ação de sinais de controle (*flags*), são capazes de direcionar diferentes operações para diferentes partes do circuito, gerando assim todos os possíveis resultados. Tais componentes são: Contador de Programa, Memória de Instruções, Banco de Registradores, Unidade Lógica Aritmética e Memória de Dados. Segue abaixo uma imagem do caminho de dados da arquitetura *MIPS* utilizada como exemplo neste relatório.

Figura 2 – Caminho de Dados - MIPS



Fonte: (1)

### 2.3.3 Contador de Programa

O Contador de Programa, ou PC (*Program Counter*), é o primeiro componente do caminho de dados. Nele é sempre armazenado em um registrador o endereço da atual instrução em execução no circuito como um todo, de forma que, com base em determinadas *flags*, tal endereço seja apenas incrementado, buscando instruções em posições contíguas da memória ou seja diretamente substituído por ou acrescido de certos valores, situação essa que ocorre em instruções de saltos, condicionais ou não.

### 2.3.4 Memória de Instruções

O componente responsável por armazenar todas as possíveis instruções a serem realizadas pelo processador é chamado de Memória de Instruções (*Instruction Memory*). Neste componente todas as instruções são armazenadas em posições contíguas da memória, de forma que dentre elas é selecionada para ser enviada à outras partes do circuito a respectiva ao endereço de entrada saído do PC, o que explica o fato dele ou substituir seu valor atual por um de entrada auxiliar ou simplesmente incrementá-lo em um, para que assim seja buscada a instrução armazenada no endereço seguinte sem que sejam realizados saltos não intencionais.

### 2.3.5 Banco de Registradores

Situado no topo da hierarquia de memória, o próximo componente na ordem de execução das instruções na arquitetura *MIPS* é o Banco de Registradores (*Register Bank*), componente no qual são armazenados dados temporários pertinentes à execução das atuais instruções.

O armazenamento de seus dados é realizado com base em uma *flag*, que indica quando dados vindos do final do circuito devem ou não ser escritos em um de seus endereços. Dentro deste componente existe um total de 32 registradores de propósito geral que são constantemente escritos e reescritos com base nos endereços de entrada das instruções. Vale ressaltar que nem todas as instruções realizam a escrita de dados no Banco de Registradores, mesmo algumas delas sendo do tipo R.

### 2.3.6 Unidade Lógica e Aritmética

Logo após o Banco de Registradores encontra-se a Unidade Lógica e Aritmética, ou ALU (*Arithmetic Logical Unity*). Responsável por realizar cálculos e deslocamentos com seus dados de entrada este componente é de suma importância para o funcionamento da maior parte das instruções. Isso porque quase todas as instruções realizadas em um caminho de dados *MIPS* o utiliza em algum momento, seja para calcular um endereço de escrita, para verificar igualdade entre dados para determinados saltos condicionais ou simplesmente parar realizar uma instrução que envolva operações lógicas ou aritméticas entre dados.

Uma vez que este componente pode conter quantas possíveis operações forem necessárias, ele possui um sinal único de controle, gerado com base na leitura do *opcode* da instrução, que simplesmente diferencia cada uma de suas possíveis execuções com os dados de entrada.

### 2.3.7 Memória de Dados

Situada na penúltima posição dos principais componentes do caminho de dados e em algum lugar do segundo e terceiro níveis da hierarquia de memória, encontra-se a Memória de Dados. Esse componente é o responsável por armazenar eventuais dados durante a execução de uma instrução.

Sua escrita ocorre de forma semelhante à do Banco de Registradores, armazenando determinado valor em um de seus endereços caso uma *flag* aponte para que o faça, contudo, apenas as instruções *load* e *store* realizam acesso direto aos seus endereços e aos dados neles contidos. Suas entradas são dadas por um dos dados saídos do banco de registradores e um endereço, no qual tal dado será escrito caso a instrução indique, e sua única saída é utilizada como entrada do dado de escrita do Banco de Registradores citado anteriormente.

### 2.3.8 Unidade de Controle

Por fim, o último componente do caminho de dados *MIPS* é a Unidade de Controle, que é o módulo responsável por realizar a troca de todas as *flags* com base na atual instrução em execução. Seu papel é de extrema importância para o funcionamento de todas as instruções contidas no conjunto de instruções do processador uma vez que muitas vezes o que diferencia o resultado de uma instrução de outra são os valores de seus sinais de controle.

Sendo assim, da mesma maneira que um processador em si funciona como um cérebro de um sistema computacional como um todo, pode se dizer que a Unidade de Controle funciona como um cérebro do processador, *setando* corretamente as *flags* de cada instrução e garantindo assim o seu funcionamento de acordo com o esperado e sem a aparição de eventuais *hazards*(1).

### 2.3.9 Auxiliares

Vale ressaltar que em um caminho de dados, além de todos os outros componentes já citados, existem multiplexadores, responsáveis por escolher como saída uma dentre duas ou mais entradas, e extensores, responsáveis por simplesmente receber uma entrada de um determinado tamanho em *bits* e fazer com que a saída seja essa mesma entrada, contudo, concatenada com 0's ou 1's até se atingir um tamanho de palavra específico.

## 2.4 Verilog

Define-se por Verilog (5), uma linguagem de descrição de *hardware* utilizada para o desenvolvimento e documentação de projetos de forma a permitir aos seus usuários a criação de vários níveis de abstração.

Ao ser utilizada, a linguagem se assimila muito à outras linguagens de programação como C/C++ e afins. Dessa forma, da mesma maneira que tais linguagens possuem variáveis tipadas como para valores inteiros, valores decimais, caractéres e variações, Verilog apresenta tipos relacionados à partes de circuitos digitais, tais como *wire*, *reg*, *input* e *output*.

A utilização desses tipos de variáveis permite a instanciação de *modules*, que representam literalmente as caixas de um projeto comum de circuitos digitais, e que quando ligados entre si, podem representar fielmente a simulação do funcionamento de sistemas computacionais.

A escolha dessa linguagem para esse projeto ao em vez de outras como HDL, *SpectreHDL* e afins foi o fato dela ser suportada pelo *software Quartus* (6), que é um programa no qual é possível se realizar o desenvolvimento de projetos em circuitos digitais, tanto da maneira estrutural, utilizando diagramas, blocos e ligando-os à mão como também em linguagens de descrição de *hardware*, de forma a poder simular o funcionamento de circuitos e de partes deles por meio de formas de onda (*waveforms*).

## 2.5 FPGA

Na parte final deste projeto, foi utilizado um elemento de *hardware* para que se tornasse possível a entrada e saída de dados do usuário no sistema desenvolvido: uma FPGA.

Como abreviação de *Full Programmable Gate Array*, ou Conjunto de Portas Programáveis em Campo(7), uma FPGA consiste em um circuito integrado que possui elementos programáveis, tais como sensores e memórias, o que lhe permite ser usada para realizar simulações a respeito de projetos como o em questão neste relatório ou para servir de interface entre dispositivos eletrônicos e/ou digitais. Neste projeto ela foi utilizada como interface de visualização/entrada de dados, de forma que, com a utilização do *software Quartus*, os códigos em Verilog puderam ser compilados para binário e ter seu funcionamento simulado diretamente na placa.

# 3 Desenvolvimento

Neste capítulo do relatório serão explicados detalhes à respeito da criação do conjunto de instruções do processador e em seguida detalhes da implementação de cada componente do caminho de dados, tanto da Unidade de Controle (UC) quanto da de Processamento (UP), de forma que fique claro o porque de cada um desses funcionar da maneira como o fez.

Assim, a ordem que se seguirá para os temas abordados neste capítulo será a ordem dos componentes presentes no caminho de dados desenvolvido: Contador de Programa, Memória de Instruções, Banco de Registradores, Unidade Lógica e Aritmética, Memória de Dados, *BigMux* e por fim a Unidade de Controle, contudo, antes explicando qual o conjunto de instruções desenvolvido.

## 3.1 Conjunto de Instruções

Para este projeto, foi criado um conjunto de instruções semelhante porém não tão extenso quanto o já citado MIPS. Tal conjunto consiste de 26 instruções, as quais visaram cobrir todas as possíveis operações básicas do processador, tais como operações lógicas, aritméticas, de acesso à memória e saltos, como pode ser visto na Tabela 4.

### 3.1.1 Formatos de Instrução

Foram projetados três formatos de instruções, semelhantes aos formatos R, I e J da arquitetura MIPS, todos compostos por um total de 32 bits.

- Formato R, de três registradores;
- Formato I, de dois registradores;
- Formato J, que contém apenas o endereço do salto;

A única real diferença entre os formatos de instrução da arquitetura MIPS e os utilizados neste projeto é o fato de que nestes o campo *funct* da instrução do tipo R não é utilizado, uma vez que no formato R original, devido ao seu grande número de instruções, tal campo é utilizado por alguns componentes do caminho de dados para diferenciar instruções que possuem *opcode* semelhante, e como o conjunto de instruções deste projeto em questão utilizou apenas 26 instruções, ele não se mostrou necessário.

Tabela 4 – Conjunto de Instruções

Instrução	abreviação	opcode	operação
adição	add	000000	$R[RD] \leftarrow R[RS] + R[RT]$
subtração	sub	000001	$R[RD] \leftarrow R[RS] - R[RT]$
adição com imediato	addi	000010	$R[RT] \leftarrow R[RS] + IM$
subtração com imediato	subi	000011	$R[RT] \leftarrow R[RS] - IM$
incremento	inc	000100	$R[RT] \leftarrow R[RS] + 1$
decremento	dec	000101	$R[RT] \leftarrow R[RS] - 1$
<i>set on less than</i>	slt	000110	$\text{if}(R[RS] < R[RT]) R[RD] = 1$
deslocamento à esquerda	shfl	000111	$R[RD] \leftarrow sl(shamt)R[RS]$
deslocamento à direita	shfr	001000	$R[RD] \leftarrow sr(shamt)R[RS]$
NOT	not	001001	$R[RT] \leftarrow !R[RS]$
AND	and	001010	$R[RD] \leftarrow R[RS] \& R[RT]$
OR	or	001011	$R[RD] \leftarrow R[RS]   R[RT]$
XOR	or	001100	$R[RD] \leftarrow R[RS] \wedge R[RT]$
multiplicação	mult	001101	$R[RD] \leftarrow R[RS] \times R[RT]$
divisão	div	001110	$R[RD] \leftarrow R[RS] / R[RT]$
<i>load</i>	ld	001111	$R[RT] \leftarrow M[IM + R[RS]]$
<i>load</i> imediato	ldi	010000	$R[RD] \leftarrow IM$
<i>store</i>	str	010001	$M[R[RS] + IM] \leftarrow R[RT]$
<i>branch on equal</i>	beq	010010	$\text{if}(R[RS] == R[RT]) PC \leftarrow IM$
<i>branch on not equal</i>	bneq	010011	$\text{if}(R[RS] != R[RT]) PC \leftarrow IM$
<i>jump</i>	jmp	010100	$PC \leftarrow IM$
<i>jump to register</i>	jmpr	010101	$PC \leftarrow R[RS]$
NOP	nop	010110	não realiza operação
HLT	hlt	010111	para a contagem do PC
<i>input</i>	in	011000	$R[RS] \leftarrow IN$
<i>output</i>	out	011001	$OUT \leftarrow R[RS]$

Fonte: Autor

### 3.1.2 Modos de Endereçamento

Com relação aos modos de endereçamento, neste projeto foram utilizados três dos existentes na arquitetura MIPS; endereçamento direto, endereçamento por registrador e por registrador indireto:

- Endereçamento Direto: os campos de endereço da instrução já contém como imediato o valor a ser utilizado como operando; utilizado, por exemplo, pelas instruções addi e subi;

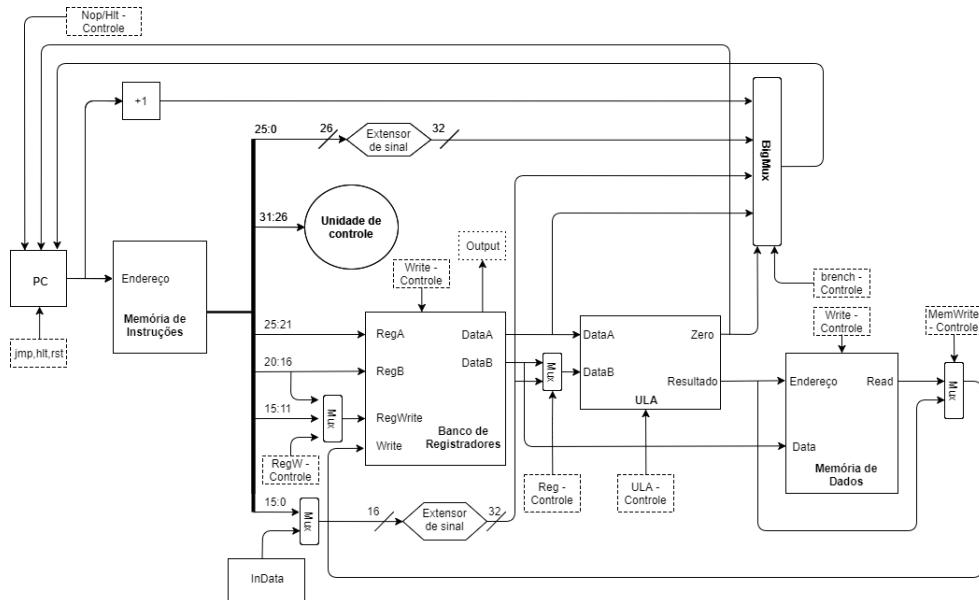
- Endereçamento por Registrador: modo utilizado para se ter acesso a operandos armazenados em registradores; utilizado pelas instruções que utilizam o formato de três registradores, tais como add, sub, and, or, etc;

Endereçamento por Registrador Indireto: utilizado apenas pela instrução Jmpr, na qual o endereço para o qual se deve ir se encontra inicialmente armazenado em um registrador do banco de registradores.

### 3.2 Arquitetura-base do Processador

O processador em questão possui um caminho de dados semelhante, contudo não tão complexo, ao da arquitetura MIPS. Nele estão presentes os já citados Contador de Programa, Memória de Instruções, Banco de Registradores, Unidade Lógica e Aritmética, Memória de Dados e o *BigMux*. Tal arquitetura pode ser vista na Figura 3.

Figura 3 – Caminho de Dados



Fonte: Autor

### 3.3 Contador de Programa

Neste projeto o contador de programa (PC) foi o primeiro elemento do caminho de dados a ser implementado. Neste módulo (Algoritmo 3.1), existe um registrador responsável por armazenar sempre o endereço da atual instrução em execução no circuito, de modo que, sempre na subida do *clock*, o seu valor é atualizado.

Essa atualização de valor pode ser feita de três maneiras; na primeira, o valor atual do registrador é simplesmente acrescido em uma unidade, para que assim a próxima instrução a ser realizada seja a do próximo endereço da memória de instruções; já na segunda, caso ocorra algum tipo de desvio (*branch* ou *jump*), o valor atual do PC é substituído por um valor respectivo ao novo endereço a ser acessado na memória de instruções, e no terceiro e último caso, caso o sinal de *reset* seja igual a 1, o valor do PC é zerado, ( $PC \leftarrow 0$ ). Como entrada do Contador de Programa existe também um sinal de *halt* (hlt), utilizado em algumas instruções para impedir a contagem até que o usuário a retome e também um sinal saído da Unidade Lógica e Aritmética (zero), utilizado para tratamentos de condições em alguns casos de saltos.

## Algoritmo 3.1 – Contador de Programa

```

module PC( address ,
    clk ,
    reset ,
    jump ,
    beq ,
    bneq ,
    hlt ,
    progCount ,
    zero );

input clk , reset , jump , hlt , zero , bneq , beq ;
input [9:0] address ;
output reg [9:0] progCount ;

wire [9:0] newValue ;

assign newValue = address ;

always @ (posedge clk) begin

    if(bneq == 1 && zero == 1)
        progCount <= newValue + 1;

    else if(bneq == 1 && zero == 0)
        progCount <= newValue;

    else if(beq == 1 && zero == 1)
        progCount <= newValue;

    else if(beq == 1 && zero == 0 )
        progCount <= newValue + 1;

    else if(jump)
        progCount <= newValue;

    else if(reset) begin
        progCount = 0;
    end

```

```

    else if( hlt ) begin
        end

    else begin
        progCount <= newValue + 1;
        end

    end

endmodule

```

### 3.4 Memória de Instruções

A memória de instruções foi implementada de maneira simples, consistindo basicamente de um vetor de 32 posições (endereços), as quais cada uma é responsável pelo armazenamento de uma possível instrução de 32 *bits* (Algoritmo 3.2). Sua entrada é um número respectivo ao endereço da instrução a ser executada, que funciona diretamente como um índice do vetor que as armazena.

No exemplo mostrado abaixo não foi escrita nenhuma instrução nos endereços da memória apenas para a melhor visualização de sua implementação, assim, na seção final de simulações, quando for mostrado o funcionamento do processador para determinado algoritmo, elas terão seus códigos em binário mostrados.

Algoritmo 3.2 – Memória de Instruções

```

module InstrMem( address ,
                  clk ,
                  InstrMemOut );

input [9:0] address ;
input clk ;
output [31:0] InstrMemOut ;
integer first = 0;
reg [31:0] instrmem [31:0];

always @ (posedge clk) begin

    if (first == 0) begin
        //instr

```

```

    first <= 0;

  end

end

assign InstrMemOut = instrmem [ address ];

```

**endmodule**

### 3.5 Banco de Registradores

Os registradores temporários utilizados neste projeto se encontram todos no Banco de Registradores (Algoritmo 3.3). Nele existe um total de 32 registradores de propósito geral, que podem armazenar valores respectivos a dados do programa, dados de entrada ou até mesmo endereços.

Sua entrada consiste de três possíveis endereços, os quais são respectivos aos registradores dos quais serão carregados ou escritos dados, e ao do dado a ser armazenado. A escrita nestes registradores ocorre de acordo com uma *flag*, que sinaliza caso ela deva ser realizada ou não em um dos seus endereços de entrada. O único endereço no qual não é possível se realizar a escrita de dados é o respectivo ao primeiro endereço (posição 0), pois nele, devido à facilidade de tal atribuição para a realização de outras instruções, a cada subida de *clock* é atribuído o valor zero, de forma que qualquer dado escrito nessa posição seria substituído no ciclo seguinte.

Na arquitetura desenvolvida neste projeto, o Banco de Registradores não foi utilizado exclusivamente para o armazenamento de dados temporários. Nele foi realizado todo o tratamento de entrada e saída de dados, de forma que não existe fora dele nenhuma unidade de *input/output*. Assim, para a entrada de dados, o registrador em que deve ser escrito o dado é passado em um dos campos da instrução, o que ocorre de maneira similar à saída de dados, em que são enviados para os *displays* números respectivos ao endereço de que se está sendo realizado o *output* e o valor nele contido, sendo que este endereço também é passado nos campos da instrução.

O tratamento para os casos de entrada e saída de dados se deu de maneira muito simples. Como sempre que uma delas era executada pelo processador a *flag hlt* se encontrava alta, congelando assim seu funcionamento, bastou se realizar uma checagem de tal *flag*: caso estivesse alta, o valor do endereço atual e do dado contido nele eram enviados como saída; caso estivesse baixa, nada era feito a respeito

## Algoritmo 3.3 – Banco de Registradores

```

module RegBank( readAddress1 ,
    readAddress2 ,
    writeAddress ,
    dataWrite ,
    writeMark ,
    clk ,
    data1 ,
    data2 ,
    hlt ,
    displayAddress ,
    valueAddress );

input [31:0] dataWrite;
input [4:0] writeAddress , readAddress1 , readAddress2 ;
input writeMark , clk , hlt ;
output [31:0] data1 , data2 ;
output reg[31:0] displayAddress ;
output reg [7:0] valueAddress ;

reg [31:0] RB[31:0];

always @ (posedge clk) begin
    RB[0] = 32'b0;
    if (writeMark)
        RB[ writeAddress ] = dataWrite;

    if (hlt) begin
        valueAddress = RB[ readAddress1 ];
        displayAddress= readAddress1 ;
    end
    else begin
        valueAddress = 8'b0;
        displayAddress = 32'b0;
    end

end

assign data1 = RB[ readAddress1 ];

```

```

assign data2 = RB[ readAddress2 ] ;

endmodule

```

## 3.6 Unidade Lógica e Aritmética

A unidade lógica e aritmética, ou ALU (*Arithmetic Logic Unity*), é responsável pela realização de todas as operações lógicas e aritméticas com os dados do processador. Seu funcionamento não depende do *clock* como os outros componentes, mas sim de variações em seus sinais de entrada, que são quase cem por cento das vezes dados a serem manipulados.

Devido ao fato de ser capaz de realizar diversas operações diferentes, a ALU conta com um *opcode* próprio de 5 bits (Tabela 5), que quando com o circuito em total funcionamento, é a ela enviado pela Unidade de Controle. Vale ressaltar que a ALU deste projeto não faz uso da porção *funct* de suas instruções uma vez que apenas seu código identificador já é suficiente para o total diferenciamento de todas as suas possíveis operações, o que não acontece na arquitetura *MIPS* original.

Tabela 5 – Conjunto de Operações da ALU

<b>opcode</b>	<b>instrução</b>
0000	adição
0001	subtração
0010	incremento
0011	decremento
0100	<i>set on less than</i>
0101	multiplicação
0110	divisão
0111	<i>shift left</i>
1000	<i>shift right</i>
1001	negação
1010	AND bit a bit
1011	OR bit a bit
1100	XOR bit a bit

Fonte: Autor

Com relação a suas operações aritméticas, foi tratada a possibilidade de ocorrência de *overflow*, de maneira que, caso ocorresse, fosse comunicado ao resto do circuito e ao usuário por meio de um sinal (of). Tal medida foi tomada devido ao fato de que o *software* Quartus para representação de números binários utiliza o padrão de complemento de 2. Assim, caso uma soma entre dois binários positivos seja realizada e ocorra *overflow*, o resultado de tal operação apresentará sinal negativo, o que é a mesma coisa que ocorre com a soma de dois números negativos; seu resultado será positivo caso haja *overflow*. Na situação de uma subtração de números com sinais distintos têm-se dois casos: o primeiro

deles, caso o primeiro operando seja positivo e o segundo negativo, no qual o resultado da operação é negativo, e o segundo caso, no qual caso o primeiro operando seja negativo e o segundo possua sinal oposto, o resultado é positivo.

Uma vez conhecidos os casos acima, foi possível se criar uma maneira com a qual o tratamento do *overflow* fosse feito de forma efetiva. Para isso se utilizou variáveis auxiliares dentro da ALU que podem ser vistas no Algoritmo 4 para se ter conhecimento do sinal dos operandos. Contudo, o *overflow* ainda assim poderia ocorrer nos casos da multiplicação de valores muito altos ou na divisão por zero. No primeiro desses casos a solução foi simplesmente reduzir o tamanho dos operandos para um total de 16 bits; já no segundo foi utilizado um padrão no qual caso o divisor da operação fosse zero, o resultado simplesmente seria o número a ser dividido.

O uso dessa unidade não se deu apenas em instruções que explicitamente manipulam valores, como a de adição ou subtração, mas também em instruções como as de saltos condicionais, (beq, bneq), utilizando uma de suas *flags* (zero) para checar se ambos seus valores de entrada são ou não iguais, para que assim fosse decidido se de fato o salto ocorreria ou não.

Algoritmo 3.4 – Unidade Lógica e Aritmética

```
module ALU( op ,
            data1 ,
            data2 ,
            result ,
            zero ,
            shamt ,
            of );

input [31:0] data1 , data2 ;
input [4:0] op , shamt ;
output zero , of ;
output reg [31:0] result ;
wire [31:0] add_value , sub_value , n_zero ;
wire ofadd , ofsub ;

assign add_value = data1 + data2 ;
assign sub_value = data1 - data2 ;

assign ofadd = (data1[31] == data2[31]
                && data1[31] != add_value[31]) ? 1 : 0;
```

```

assign ofsub = (data1[31] == data2[31]
                && data1[31] != sub_value[31]) ? 1 : 0;
assign of = ofadd | ofsub;

assign n_zero = (data2 == 0) ? 1 : data2;

always @ (op or data1 or data2 or n_zero or shamt) begin

    case(op[4:0])
        5'b00000: result = data1 + data2;
        5'b00001: result = data1 - data2;
        5'b00010: result = data1 + 1;
        5'b00011: result = data1 - 1;
        5'b00100: result = data1 < data2 ? 1 : 0;
        5'b00101: result = data1[15:0] * data2[15:0];
        5'b00110: result = data1 / n_zero;
        5'b00111: result = data1 << shamt;
        5'b01000: result = data1 >> shamt;
        5'b01001: result = ~data1;
        5'b01010: result = data1 & data2;
        5'b01011: result = data1 | data2;
        5'b01100: result = data1 ^ data2;
        default: result = 0;
    endcase

end

assign zero = (result == 0);

endmodule

```

## 3.7 Memória de Dados

Acessada apenas por instruções de *load* e *store*, a Memória de Dados é o último dos componentes presentes no caminho de dados da arquitetura utilizada neste projeto. Ela foi o *module* responsável pelo real armazenamento de dados do sistema, sendo que a forma como operou é muito semelhante à forma com que se deu o funcionamento do Banco de Registradores: uma *flag* marcava quando era necessário de fato se realizar a escrita em um de seus endereços. Suas únicas duas entradas foram respectivamente o endereço de acesso e o dado a ser possivelmente escrito nesse mesmo endereço.

Mesmo não sendo todas as instruções que realizam acesso aos seus dados, a Memória de Dados, assim como a maioria dos outros componentes, manda sinais de *output* em todo ciclo de *clock*, sendo assim, em todo ciclo ela envia um dado lido de um de seus endereços, contudo isso não é problema uma vez que logo após a sua saída existe um multiplexador responsável por enviar como dado para ser escrito no Banco de Registradores tal sinal lido ou o resultado de uma operação da ALU. Referente ao sinal de *clock*, é importante ressaltar que a escrita realizada na memória ocorreu em sua descida (*negedge*). Isso foi feito para evitar que a memória tentasse ler um dado de um endereço ainda não calculado propriamente do Banco de Registradores.

### Algoritmo 3.5 – Memória de Dados

```
module DataMem( data ,
    address ,
    clk ,
    DataMemOut ,
    WriteFlag );

input [31:0] data ;
input [31:0] address ;
input clk , WriteFlag ;
output [31:0] DataMemOut ;

reg [31:0] DM[9:0];

always @ (negedge clk) begin

    if (WriteFlag)
        DM[ address ] = data ;

end

assign DataMemOut = DM[ address ] ;

endmodule
```

### 3.8 BigMux

Por mais que todos os principais componentes do processador já tenham tido seus projetos descritos, existe ainda um componente importante para o funcionamento de suas instruções: o neste projeto chamado de *BigMux*. Nele foram realizadas as decisões de qual o endereço correto de retorno ao Contador de Programa ao término de cada instrução.

Seu funcionamento se deu mais uma vez com base em *flags* e sinais seletores, os quais foram utilizados para se determinar qual de suas entradas seria usada como saída de volta ao PC. Tal tratamento foi necessário devido ao fato de que com apenas um sinal de controle (zero) não seria possível de se realizar as instruções de *branch on equal* e *branch on not equal* uma vez que uma delas é realizada quando seu valor é 1 e a outra é realizada quando seu valor é 0, gerando um total de quatro combinações possíveis de casos a serem tratados, como pode ser visto no Algoritmo 3.6.

Algoritmo 3.6 – *BigMux*

```
module BigMux( zero ,
    BMselect ,
    beq ,
    bneq ,
    extSum ,
    extSignal ,
    dataARegBank ,
    PcOut ,
    bmOut ) ;

input zero , beq , bneq ;
input [1:0] BMselect ;
input [31:0] PcOut , extSum , extSignal , dataARegBank ;
output reg [31:0] bmOut ;

always @ (PcOut or extSum or extSignal or dataARegBank or zero or
BMselect or beq or bneq) begin

    case( BMselect [1:0] )
        2'b01:
        begin
            if( bneq == 1 && zero == 1)
                bmOut = PcOut ;
        
```

```

        else if(bneq == 1 && zero == 0)
            bmOut = extSum;

        else if(beq == 1 && zero == 1)
            bmOut = extSum;

        else if(beq == 1 && zero == 0)
            bmOut = PcOut;
    end

    2'b10: bmOut = extSignal;
    2'b11 : bmOut = dataARegBank;
    default: bmOut = PcOut;

endcase
end

```

## 3.9 Unidade de Controle

A unidade de controle, como já citado anteriormente, é a unidade responsável por configurar o valor de todos os sinais de controle do processador em cada uma de suas instruções. Sendo assim, ao se utilizar a linguagem Verilog, sua implementação se torna apenas um grande "*switch case*" no *opcode* da instrução. Devido a isso e ao fato do número total de instruções ser relativamente elevado para ser mostrado por completo, no Algoritmo 3.7 constam apenas as primeiras linhas de sua implementação, onde são mostradas seus *wires* e *regs* e o primeiro caso de mudança de valor nos sinais de controle. Seu código completo, com todos os casos de todas as instruções, pode ser encontrado no apêndice do relatório.

O único caso da escolha dos sinais de controle da UC que vale a pena ser citado é o de seu funcionamento nas instruções de *input* e *output*, pois estas são as únicas em que o sinal de controle *hlt* (*halt*) é utilizado para se congelar o funcionamento do processador, fora a própria instrução de interrupção. Isso porque tal interrupção permite, no caso da instrução de entrada de dados, que o valor a ser inserido seja selecionado à mão pelo usuário, e no da instrução de saída, que o valor selecionado para *output* seja mostrado em um possível *display* e o continue sendo até que o usuário decida dar continuidade à execução das outras instruções do sistema acionando à mão algum sinal que o faça (*check*, que será explicado na seção de simulações).

## Algoritmo 3.7 – Unidade de Controle

```

module controlUnit (check , clock , opcode , PRBMs, PAMselect , bmSelect ,
hlt , jmp , beq , bneq , rst , pdmms , wf , wm, ALUcode , imControl);

    input check , clock ;
    input [5:0] opcode ;
    output reg PRBMs;
    output reg PAMselect ;
    output reg [1:0] bmSelect ;
    output reg hlt , jmp ,rst , beq , bneq ;
    output reg pdmms;
    output reg wf;
    output reg wm;
    output reg [4:0] ALUcode ;
    output reg imControl;

always @ (opcode or check) begin
    case(opcode [5:0])
        6'b000000: begin
            PRBMs = 1'b1 ;
            PAMselect = 1'b0 ;
            bmSelect = 2'b00 ;
            hlt = 1'b0 ;
            jmp = 1'b0 ;
            beq = 1'b0 ;
            bneq = 1'b0 ;
            rst = 1'b0 ;
            pdmms = 1'b1 ;
            wf = 1'b0 ;
            wm = 1'b1 ;
            ALUcode = 5'b00000 ;
            imControl = 1'b0 ;
        end
        .
        .
        .
    end
endmodule

```

# 4 Resultados Obtidos e Discussões

Uma vez implementados os módulos da maneira como descrito anteriormente, as simulações foram realizadas em duas etapas: uma sem e outra com a utilização da Unidade de Controle, respectivamente, partes 1 e 2.

A primeira destas também foi dividida em duas etapas, sendo a primeira delas a em que cada componente foi testado individualmente com *waveforms* próprios, e em seguida a etapa na qual foi elaborado um algoritmo que será detalhado mais à frente e que foi testado com uma instanciação final da Unidade de Processamento do processador no *software* Quartus, de forma a testar o funcionamento em conjunto dos componentes em diferentes tipos de instruções, sendo que também foram simuladas as outras instruções não contidas em tal algoritmo, contudo, avulsamente à ele.

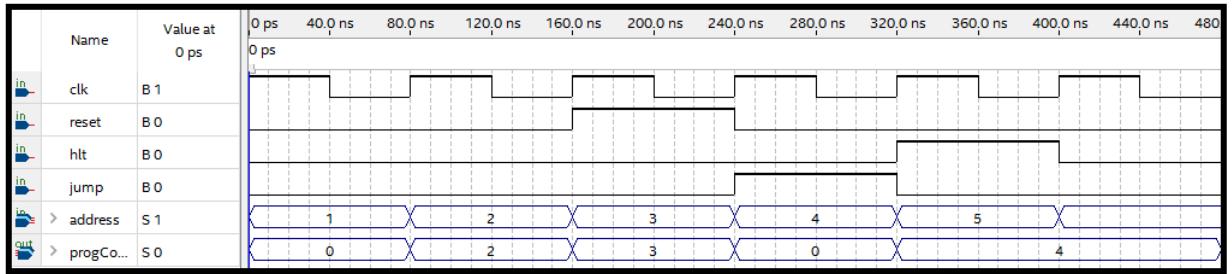
Assim, a segunda etapa de resultados e simulações foi realizada com a interação entre a Unidade de Processamento e a Unidade de Controle juntamente com a placa FPGA, tornando possível em seus testes a entrada de dados no processador para a execução de um algoritmo teste a fim de confirmar definitivamente seu funcionamento. O código em Verilog que descreve o módulo final a partir do qual foram realizadas as simulações com a FPGA podem ser encontrados no apêndice do relatório.

## 4.1 Simulações Individuais - Parte 1

### 4.1.1 Contador de Programa

A primeira simulação realizada foi no Contador de Programa (Figura 4). Nela pode ser visto que seu funcionamento se deu como o esperado: caso a instrução não necessitasse de nenhum sinal auxiliar, o seu valor era apenas recebido, incrementado e em seguida enviado; caso houvesse um sinal de desvio (*jump*), o valor do contador seria então o endereço em um de seus *inputs*; caso o sinal de *reset* fosse alto, seu valor era reiniciado, e por fim, caso ocorresse o sinal de *halt*, a contagem cessaria.

Figura 4 – Waveform 1: Contador de Programa



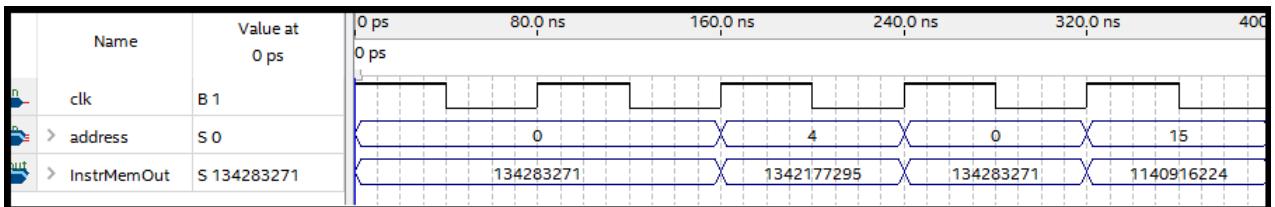
Fonte: Autor

#### 4.1.2 Memória de Instruções

Na simulação da memória de instruções também foi obtido um resultado esperado, de forma que nenhuma mudança em sua implementação precisou ser feita (Figura 5). Sua instrução de saída foi a armazenada no endereço de entrada que é saída do PC, sendo assim, para diferentes valores de entrada, foram obtidos diferentes valores de saída, respectivos às instruções armazenadas em tais endereços.

Vale ressaltar que na simulação presente no relatório, os valores de saída referentes à instrução se encontram sem nenhum padrão ou grau de continuidade entre si devido ao fato de que, apenas para uma melhor visualização de que são diferentes, foram mostrados como números inteiros, contudo são na realidade binários referentes à determinadas instruções.

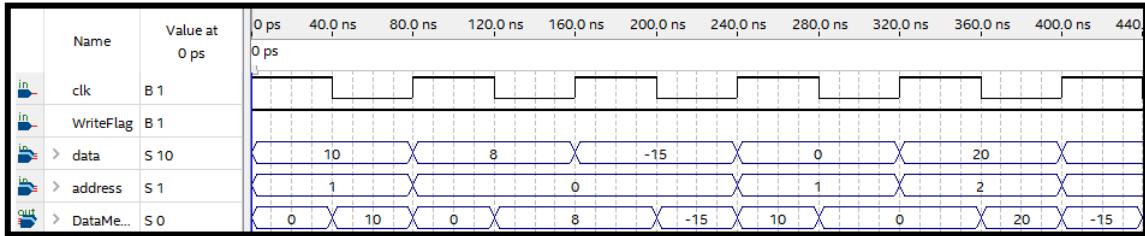
Figura 5 – Waveform 2: Memória de Instruções



Fonte: Autor

#### 4.1.3 Banco de Registradores e Memória de Dados

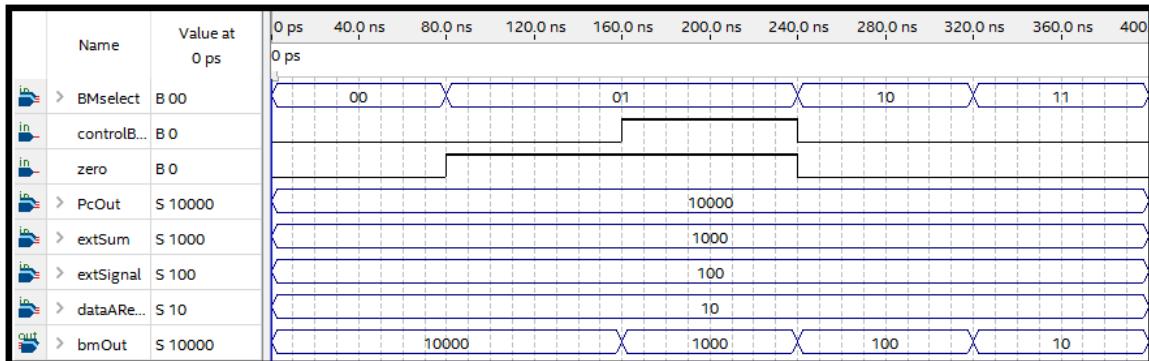
A Memória de Dados operou como esperado de acordo com o desenvolvimento prévio dos módulos e do conjunto de instruções do projeto, de forma a realizar a escrita em seu endereço de entrada caso uma *flag* apontasse para que o fizesse e caso isso ocorresse na descida do sinal de *clock* (*negedge*). Seu funcionamento ocorreu da mesma forma que o do Banco de Registradores, motivo pelo qual foi representado na (Figura 6), apenas um *waveform* para ambos os módulos, com exceção apenas do fato de que a escrita nele foi feita na subida do sinal de *clock* (*posedge*).

Figura 6 – *Waveform 3: Banco de Registradores e Memória de Dados*

Fonte: Autor

#### 4.1.4 *BigMux*

Por fim, o último módulo testado individualmente foi o *BigMux*. Mais uma vez seu funcionamento ocorreu como deveria de acordo com o planejamento do projeto. Seu sinal de saída foi o selecionado por suas *flags*, indicando assim se o sinal do PC seria apenas incrementado ou se haveria algum tipo de desvio, sendo ele condicional ou não. Cada um dos sinais do *waveform* (Figura 7), representa cada uma dessas *flags* utilizadas para determinar os possíveis sinais de saída do módulo.

Figura 7 – *Waveform 4: BigMux*

Fonte: Autor

## 4.2 Simulação Total (Unidade de Processamento) - Parte 1

Para o teste total do processador com seus módulos em conjunto (Anexo 1), foi criado um pequeno algoritmo composto de oito instruções, as quais envolvem algumas do tipo R, I e J, para assim comprovar o funcionamento dele em diferentes situações que envolvem diferentes sinais de controle. Para um melhor entendimento das etapas serão utilizados índices de vetores para representar os endereços de escrita da memória, do PC e o banco de registradores.

O algoritmo consiste como dito em oito etapas, que, em instruções, são: addi, addi, addi, bneq, *jump*, *store*, *load* e por fim um sub. Respectivamente à cada uma dessas instruções, os valores utilizados nelas podem ser vistos abaixo (Tabela 6).

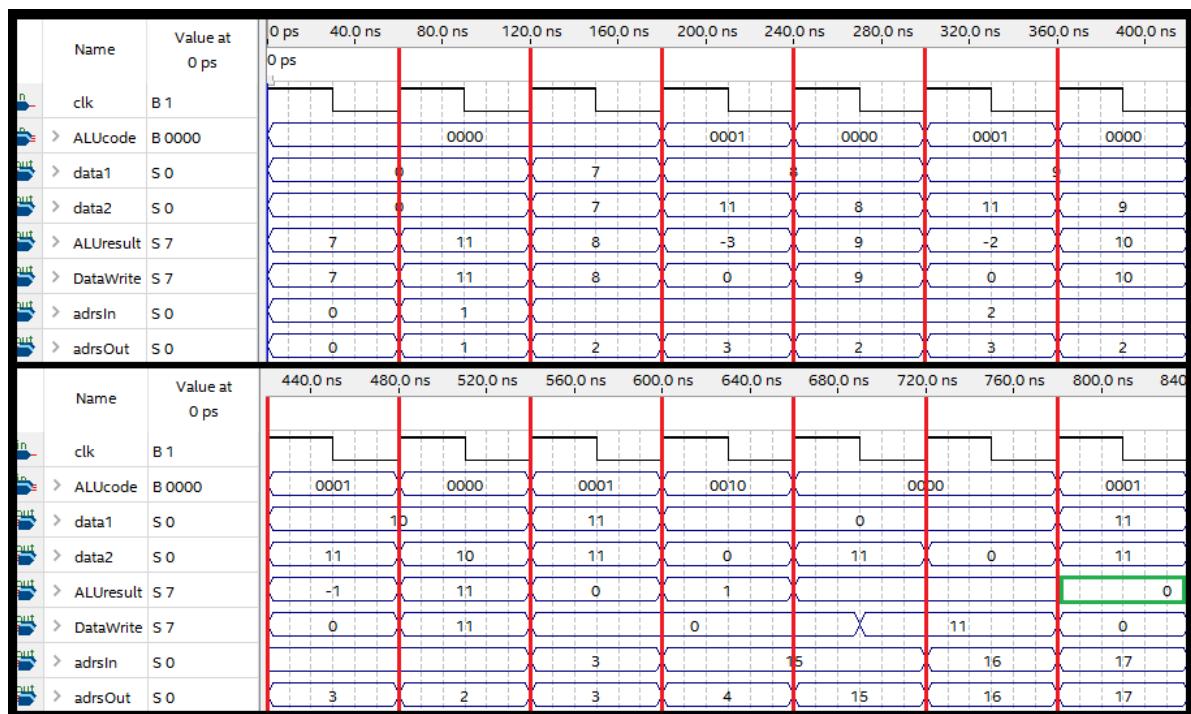
Tabela 6 – Algoritmo Teste

(0) - R[1] = R[0] + 7 - addi
(1) - R[5] = R[0] + 11 - addi
(2) - R[1] = R[1] + 1 - addi
(3) - if(R[1] != R[5]) <i>jump</i> to 2 - bneq
(4) - PC $\leftarrow$ 15 - jmp
(15) - M[0] $\leftarrow$ R[1] - <i>store</i>
(16) - R[6] $\leftarrow$ M[0] - <i>load</i>
(17) - R[4] = R[1] - R[6] - sub

Fonte: Autor

Segue sua explicação e a imagem da simulação (Figura 8):

Figura 8 – Waveform 5: Simulação Final



Fonte: Autor

Na simulação do algoritmo propriamente dita, foi omitida grande parte dos sinais de controle para uma melhor identificação dos resultados. Sendo assim, na simulação são visíveis os valores do sinal de clock (clk), opcode da ALU (ALUcode), as saídas do banco de registradores,(data1 e data2), o resultado da ALU (ALUresult), o dado a ser escrito no banco de registradores (DataWrite) e por fim os endereços da atual e da próxima instrução, respectivamente (adrsIn e adrsOut).

Em seu primeiro ciclo de clock foi realizada a primeira instrução de soma com imediato, de forma que, ao seu término, o valor contido em R[1] fosse igual a 7; no segundo e no terceiro ciclos o mesmo ocorre, apenas com registradores e valores diferentes, de forma que ao término do segundo o valor contido em R[5] fosse igual a 11 e o em R[1] igual a 8. Assim, no quarto ciclo de clock é dado início ao laço condicional criado no algoritmo, que agia de forma a enquanto o valor dos registradores R[1] e R[5] fossem diferentes, o valor de R[1] fosse incrementado. Mais afundo do que isso, caso os valores contidos em ambos os registradores fossem diferentes, seria realizado um pulo para o endereço 2, que é onde se encontrava a terceira instrução de soma com imediato, que apenas acrescia 1 ao valor em R[1]. Tais acréscimos podem ser vistos com o valor de *DataWrite* nos intervalos de 240ns-280ns, 360ns-400ns, e 480ns-520ns, os quais respectivamente mostram os valores desse sinal como 9, 10 e 11, mostrando também os endereços respectivos à essas instruções, que foram o 2 e o 3.

Terminado o laço de repetição criado com as primeiras instruções, no endereço 4 foi realizado um salto (*jump*) dele para o endereço 15, onde deu-se início às instruções de acesso à memória. Na primeira delas foi armazenada o dado contido em R[1] em M[0] (*load*), ou seja, atribuiu-se o valor 11 à primeira posição da memória, o que pode ser visto na descida do *clock* no valor de *DataWrite* no intervalo de 680ns-720ns; em seguida o mesmo valor armazenado na memória foi de lá usado e atribuído à R[6] (*store*), o que pode ser visto no intervalo seguinte.

Por fim, realizou-se a subtração do valor contido em R[6] do valor contido em R[1], de forma que tal valor foi atribuído a R[4]. Assim, como tais valores dos endereços 1 e 6 eram iguais, o valor de tal operação foi igual a zero, o que pode ser visto destacado no último ciclo de *clock* da simulação. Os valores de todas as *flags* e sinais de controle no decorrer da execução do algoritmo estão citados a seguir na Tabela 7 e suas definições em binários de 32 bits podem ser vistos logo em sequência.

Tabela 7 – Sinais de Controle

	<b>addi</b>	<b>sub</b>	<b>bneq</b>	<b>load</b>	<b>store</b>	<b>jump</b>
<b>Pamselect</b>	1	0	0	1	1	0
<b>PRBMs</b>	0	1	0	0	0	0
<b>beq</b>	0	0	0	0	0	1
<b>bneq</b>	0	0	1	0	0	0
<b>controlBranch</b>	0	0	0	0	0	0
<b>hlt</b>	0	0	0	0	0	0
<b>jmp</b>	0	0	1	0	0	1
<b>pddms</b>	1	1	0	0	0	0
<b>wf</b>	0	0	0	0	1	0
<b>wm</b>	1	1	0	1	0	0
<b>imControl</b>	0	0	0	0	0	0
<b>ALUop</b>	00001	00000	00010	xxxxx	xxxxx	xxxxx

Fonte: Autor

### Algoritmo 4.1 – Instruções do Algoritmo

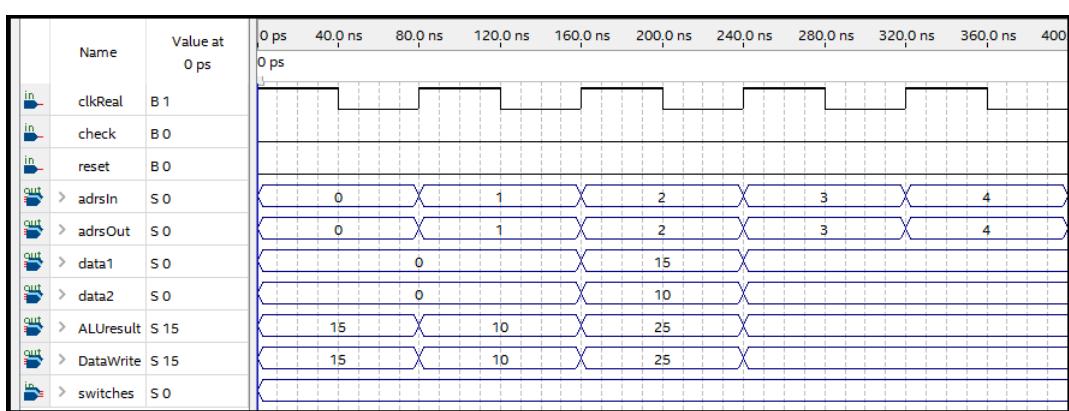
```
instrmem[0] = 32'b000010_0000_0001_00000000000010011;
instrmem[1] = 32'b000010_0000_00101_00000000000010111;
instrmem[2] = 32'b000010_00001_00001_00000000000000001;
instrmem[3] = 32'b010011_00001_00101_000000000000000010;
instrmem[4] = 32'b010100_00000000000000000000000000001111;
instrmem[15]= 32'b010001_00000_00001_00000000000000000000;
instrmem[16]= 32'b001111_00000_00110_00000000000000000000;
instrmem[17]= 32'b000001_00001_00110_00100_000000000000;
```

## 4.3 Simulação Total (Outras instruções) - Parte 1

Nessa seção, serão diretamente expostos e brevemente explicados alguns exemplos de *waveforms* a respeito das outras instruções do conjunto de instruções do processador, sendo estas aquelas não presentes no algoritmo descrito na seção anterior. Assim, tais instruções são: adição, subtração com imediato, incremento, decremento, *set on less than*, deslocamento à esquerda, deslocamento à direita, NOT, AND, OR, XOR, multiplicação, divisão, *load* imediato, *branch on equal* e *jump to register*.

### 4.3.1 Adição - (add)

Figura 9 – Waveform 6: add

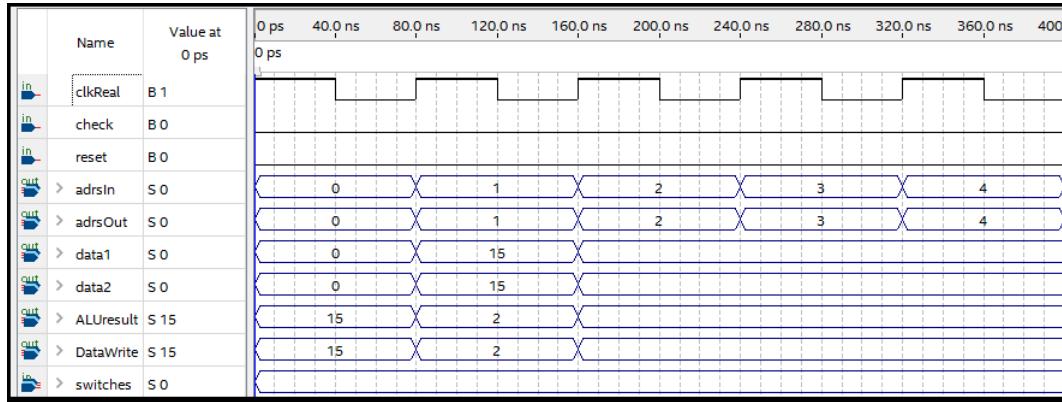


Fonte: Autor

Na figura acima, simplesmente é realizada adição entre dois valores carregados em dois registradores diferentes.

### 4.3.2 Subtração com imediato - (subi)

Figura 10 – Waveform 7: subi

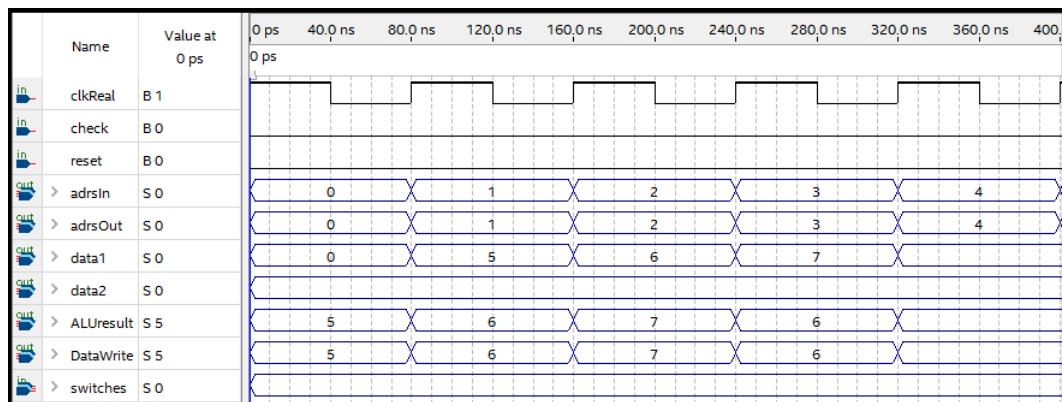


Fonte: Autor

Na instrução descrita pela imagem acima, um registrador foi carregado com o valor numérico 15 e então subtraído, pelo seu imediato, de 13, resultando assim no valor 2.

### 4.3.3 Incremento e decremento - (inc/dec)

Figura 11 – Waveform 8: inc/dec

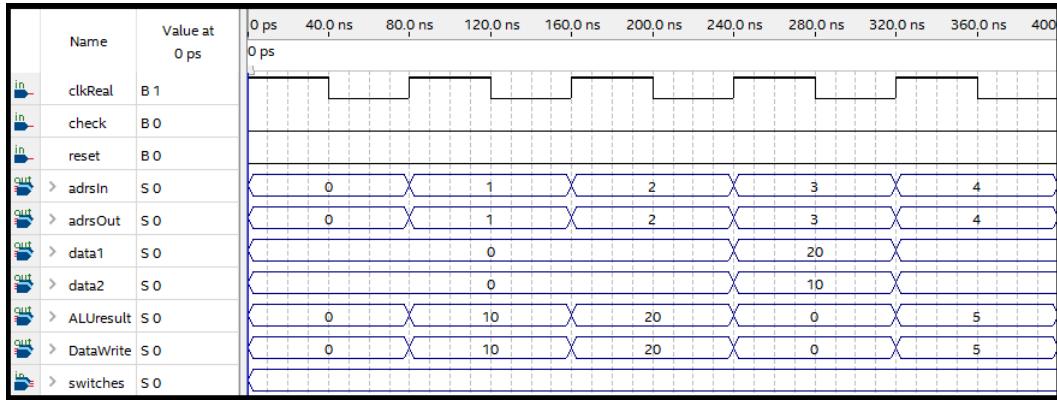


Fonte: Autor

Nesta simulação, primeiro foi carregado em um registrador o valor numérico 5, em seguida, tal valor simplesmente foi acrescido duas vezes e decrescido uma vez, o que gerou o resultado visto ( $5 + 1 + 1 - 1 = 6$ ).

#### 4.3.4 Set on less than - (slt)

Figura 12 – Waveform 9: slt

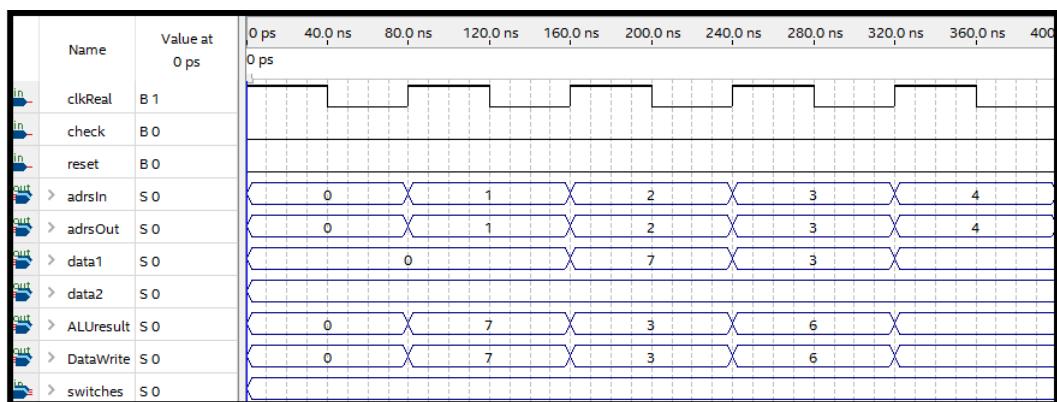


Fonte: Autor

Para o teste da instrução *set on less than* foi elaborado um pequeno algoritmo. Neste, como pode ser visto na imagem de sua simulação, foi dada uma instrução vazia (nop), carregados nos registradores R[2] e R[3] respectivamente os valores 10 e 20 e feita a comparação: caso o valor contido no registrador R[3] fosse menor do que o no R[2], o valor de R[4] seria posto em 1, e 0 caso contrário. Assim, como tal condição não foi verdadeira, o valor armazenado em R[4] foi 0, o que pode ser visto depois dele ser acrescido por imediato de 5 unidades, resultando no próprio valor acrescido ( $5 + 0 = 5$ ).

#### 4.3.5 Deslocamentos - (shfr/shfl)

Figura 13 – Waveform 10: shfr/shfl



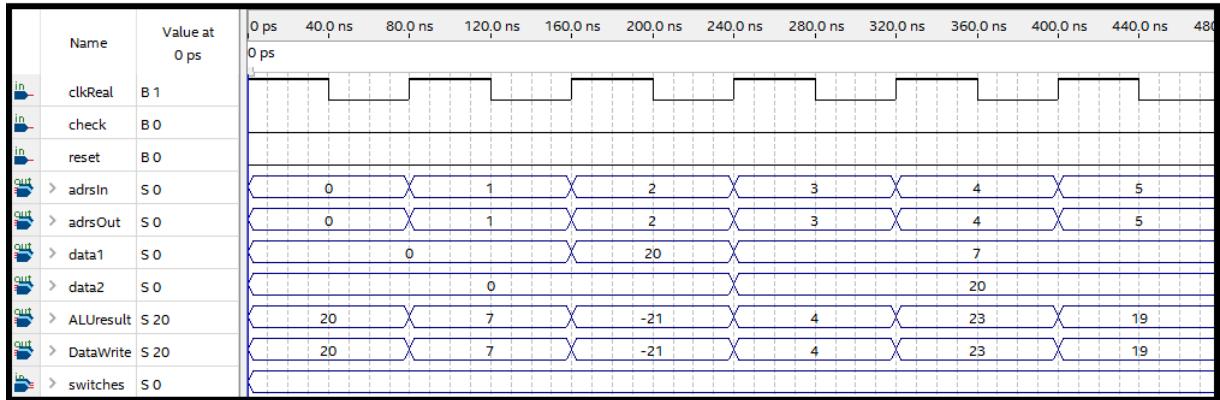
Fonte: Autor

Para a simulação das instruções de deslocamento, simplesmente foi carregado o valor 7 (0111) em um registrador e em seguida realizado seu deslocamento para ambos os

lados: primeiramente para a direita, resultando em 3 (0011), e em seguida para a esquerda, resultando em 6 (0110).

#### 4.3.6 Operações lógicas - (not, and, or e xor)

Figura 14 – *Waveform 11: not, and, or e xor*

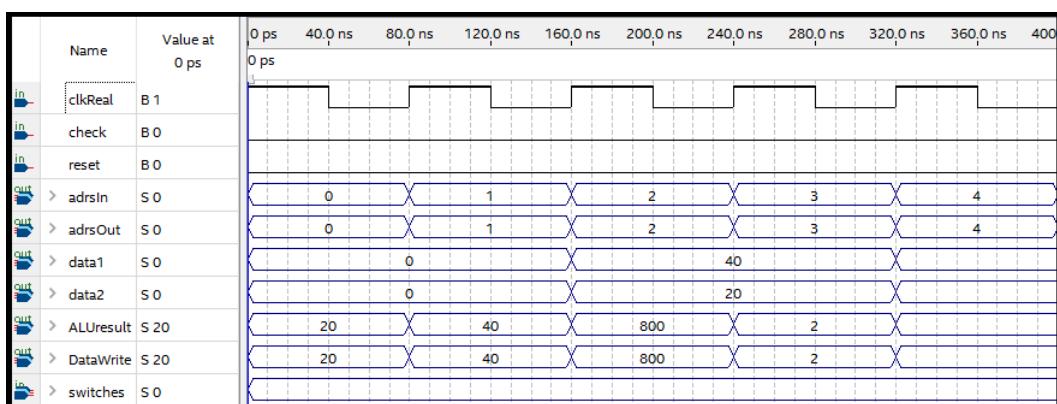


Fonte: Autor

As instruções lógicas, por serem do tipo R, mesmo tipo que as instruções aritméticas, tem seu funcionamento muito semelhante a elas, sendo apenas diferenciado pelo tratamento de seus dados na ALU. Sendo assim, para essa simulação, foram carregados nos registradores R[1] e R[2] respectivamente os valores 20 e 7 nos primeiros dois ciclos de *clock*, em seguida o registrador R[3] recebeu o valor do R[2] negado ( R[2]), e para finalizar, os registradores R[4], R[5] e R[6] receberam respectivamente os resultados das operações and, or e xor, todas essas utilizando os dados contidos em R[1] e R[2] carregados previamente.

#### 4.3.7 Multiplicação e divisão - (mult/div)

Figura 15 – *Waveform 12: mult/div*



Fonte: Autor

Similarmente às instruções de adição e subtração, as instruções de multiplicação e divisão são realizadas pela ALU, assim, para sua simulação foram carregados nos registradores R[1] e R[2] respectivamente os valores 20 e 40, em seguida, eles foram multiplicados e divididos entre si, sendo que na divisão, o menor valor foi utilizado como divisor ( $40*20 = 800$ ;  $40/20 = 2$ ).

#### 4.3.8 Instruções remanescentes - (ldi, beq, jmpr)

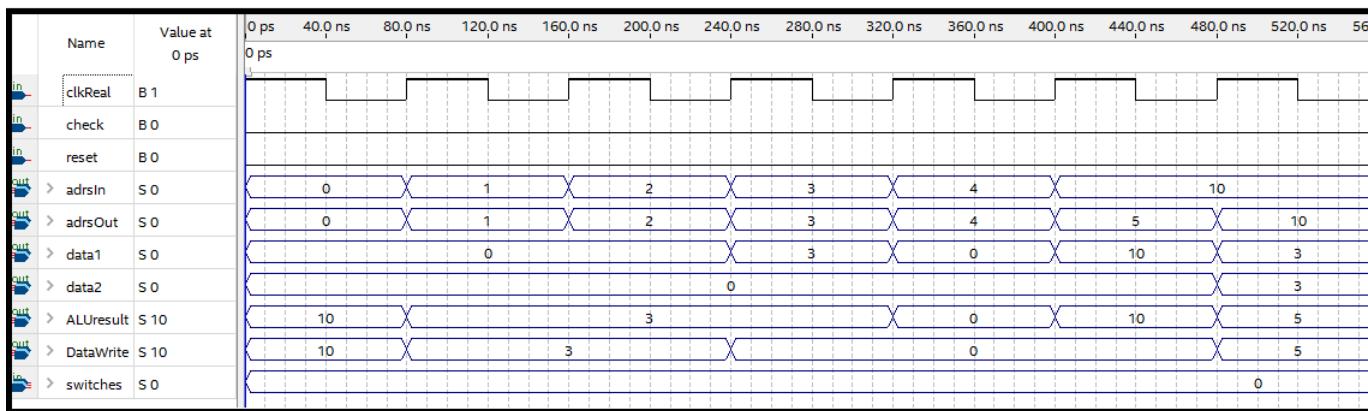
Dadas as simulações anteriores, as únicas instruções que não foram ainda simuladas foram as de *load* imediato, *branch on equal* e *jump to register*, sendo assim, elas o serão nesta seção de acordo com o algoritmo a seguir:

Tabela 8 – Pseudocódigo - algoritmo teste

(0) - R[5] ← 10 - ldi
(1) - R[3] ← 3 - ldi
(2) - R[2] ← 3 - ldi
(3) - if(R[2]==R[3]) jump to 5 - beq
(4) - NOP - nop
(5) - jump to R[5] - jmpr
(6) - NOP - nop
(7) - NOP - nop
(8) - NOP - nop
(9) - NOP - nop
(10) - R[2] += 2 - addi

Fonte: Autor

Figura 16 – Waveform 13: instruções remanescentes



Fonte: Autor

Para o teste dessas três últimas instruções, primeiramente foram carregados nos registradores R[5], R[3] e R[2] respectivamente os valores 10, 3 e 3, em seguida foi realizada a comparação dos valores contidos em R[2] e R[3]: caso fossem iguais, o endereço da próxima instrução seria 5, o que, no caso desta simulação, ocorreu. Assim, do endereço 5 foi

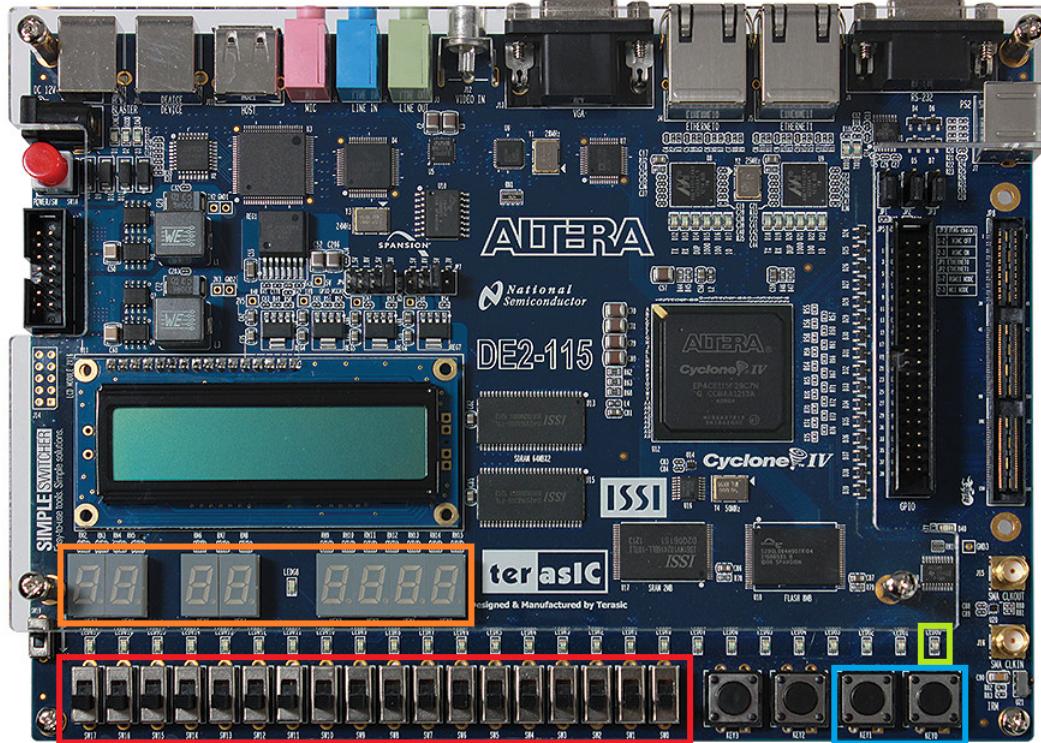
realizado um salto para o endereço de valor armazenado no registrador R[5], de forma que, nesse endereço (10), foi por fim realizada a adição por imediato do valor 2 ao registrador R[2], apenas para comprovar os saltos realizados, resultando assim em 5 ( $2 + 3 = 5$ ).

## 4.4 Simulação Total (UP, UC e FPGA) - Parte 2

Nesta seção serão mostrados os resultados finais do processador e sua interação com a FPGA. Para isso serão mostrados dois algoritmos, um que calcula o n-ésimo termo da sequência de Fibonacci, inserido como *input* pelo usuário, e um segundo, genérico e sem motivos práticos, apenas para demonstrar o funcionamento de algumas outras intruções não contidas no primeiro juntamente com as de acesso à memória.

Antes que sejam descritos os algoritmos e suas execuções, primeiramente segue uma imagem e a explicação a respeito dos pinos e saídas da FPGA utilizados nestas simulações, para assim se obter um maior entendimento do que representam cada um de seus sinais e saídas.

Figura 17 – FPGA - Identificação



Fonte: Autor

Primeiramente, na cor laranja, se encontram os *displays* da placa, os quais, da esquerda para a direita, representam; o endereço do PC nos dois primeiros, o endereço do

Banco de Registradores operado nos dois seguintes, e por fim, nos quatro últimos, o valor contido nesse mesmo endereço.

Representados em vermelho estão os *switches* da placa, os quais foram utilizados quase que em sua maioria para se determinar o valor binário do imediato a ser inserido na função de *input*, de forma que, quando para baixo, seu valor era 0, e 1 caso contrário, representando assim da esquerda para a direita os *bits* mais significativos para os menos significativos. A única exceção para esses *switches* foi o primeiro, que ao invés de fazer parte da composição do imediato foi utilizado como botão de *reset*, de forma que, caso acionado, faria com que o próximo endereço do PC fosse zero, reiniciando assim a execução das instruções.

Em azul, do lado direito da imagem acima, encontram-se dois botões, que, respectivamente, representam o botão *enter* (*check* da Unidade de Controle, já citado anteriormente e utilizado para o usuário determinar quando o processador deve voltar à contagem das instruções após a realização de uma que a parasse, no caso, *input* ou *output*) e o *clock* manual utilizado para testes e simulações. Por fim, em verde, encontra-se ressaltado o *led* utilizado para sinalizar a *flag hlt*, ou seja, utilizado para indicar ao usuário quando o sistema está parado e no aguardo de um sinal que o faça retomar o funcionamento.

#### 4.4.1 Algoritmo 1 - Fibonacci

Primeiramente, há de se definir formalmente a sequência de Fibonacci: dentro dela, seu  $n$ -ésimo termo é calculado pela soma de seus dois termos anteriores, sendo que seus primeiros dois termos são respectivamente 0 e 1. A sequência se extende indefinidamente de forma a gerar infinitos termos, sendo alguns deles, por exemplo, (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...) e assim por diante.

Tendo em vista essa definição, foi primeiramente criado um código em "*assembly*", que por meio de um código em C++, contido em anexo neste relatório, foi convertido para o binário colocado na Memória de Instruções e efetivamente utilizado para a realização das instruções do processador. O pseudocódigo referente à esse algoritmo pode ser visto na Tabela 9.

Tal código, quando convertido para o binário utilizado pelo processador, gerou as seguintes expressões, que foram postas dentro da Memória de Instruções e então executadas:

#### Algoritmo 4.2 – Fibonacci - Binário

```
instrmem [0] = 32'b010000_00000_00001_0000000000000001 ;
instrmem [1] = 32'b010000_00000_00010_0000000000000010 ;
instrmem [2] = 32'b010000_00000_00011_0000000000000011 ;
instrmem [3] = 32'b010000_00000_01101_0000000000000000 ;
```

Tabela 9 – Fibonacci - Pseudocódigo

(0) - R[1] ← 1 - ldi
(1) - R[2] ← 2 - ldi
(2) - R[3] ← 3 - ldi
(3) - R[13] ← 0 - ldi
(4) - R[31] ← <i>input</i> - in
(5) - if(R[31]==R[1]) <i>jump to 17</i> - beq
(6) - if(R[31]==R[2]) <i>jump to 18</i> - beq
(7) - if(R[31]==R[3]) <i>jump to 18</i> - beq
(8) - R[31] -= 3 - subi
(9) - R[4] = R[0] + 1 - addi
(10) - R[5] = R[0] + 1 - addi
(11) - if(R[31]==R[13]) <i>jump to 19</i> - beq
(12) - R[13]+=1 - addi
(13) - R[6] = R[5] + R[4] - add
(14) - R[4] = R[5] + 0 - addi
(15) - R[5] = R[6] + 0 - addi
(16) - <i>jump to 11</i> - jmp
(17) - LED = R[0] - out
(18) - LED = R[1] - out
(19) - LED = R[6] - out
(20) - HALT - hlt

Fonte: Autor

```

instrmem [4] = 32'b011000_00000_11111_0000000000000000 ;
instrmem [5] = 32'b010010_11111_00001_0000000000010001 ;
instrmem [6] = 32'b010010_11111_00010_0000000000010010 ;
instrmem [7] = 32'b010010_11111_00011_0000000000010010 ;
instrmem [8] = 32'b000011_11111_11111_0000000000000011 ;
instrmem [9] = 32'b000010_00000_00100_0000000000000001 ;
instrmem [10] = 32'b000010_00000_00101_0000000000000001 ;
instrmem [11] = 32'b010010_11111_01101_00000000000010011 ;
instrmem [12] = 32'b000010_01101_01101_0000000000000001 ;
instrmem [13] = 32'b000000_00101_00100_00110_00000_000000 ;
instrmem [14] = 32'b000010_00101_00100_0000000000000000 ;
instrmem [15] = 32'b000010_00110_00101_0000000000000000 ;
instrmem [16] = 32'b010100_000000000000000000000000000000001011 ;
instrmem [17] = 32'b011001_00000_00000000000000000000000000000000 ;
instrmem [18] = 32'b011001_00001_00000000000000000000000000000000 ;
instrmem [19] = 32'b011001_00110_00000000000000000000000000000000 ;
instrmem [20] = 32'b010111_00000000000000000000000000000000 ;

```

Esse algoritmo funciona da seguinte maneira: primeiramente são carregados nos registrador R[1] o valor numérico 1, no registrador R[2] o valor numérico 2 e no registrador R[3], o valor 3; em seguida, inicializa-se o contador do laço utilizado, localizado no

registrador R[13], com valor numérico 0. Em seguida é lido um *input* do usuário, em binário, referente ao n-ésimo termo da sequência de Fibonacci, que será por fim mostrado no *display* da FPGA. Após lido tal valor, são feitas algumas comparações; a primeira, caso o valor lido seja igual a 1, caso em que o valor a ser mostrado deve ser o primeiro da sequência (zero), é realizado um salto para o endereço 17, no qual é mostrado tal valor; a segunda e a terceira são realizadas de forma a se checar se os valores lidos são 2 ou 3, porque, caso o sejam, o valor numérico 1 deve ser mostrado nos *displays*, visto que a sequência se dá por (0, 1, 1, 2, 3, ...), o que ocorre no endereço 18.

Feitas tais comparações, caso nenhum dos saltos citados anteriormente tenham sido realizados, o valor do registrador que recebeu o *input* é subtraído de 3 uma vez que os três primeiros termos da sequência já foram calculados, em seguida, é utilizada a operação *addi* para se mover para ambos registradores R[4] e R[5] o valor numérico 1.

Assim, se da inicio ao laço principal do algoritmo: caso o valor do contador seja igual ao valor armazenado no registrador R[31], é realizado um salto para o endereço 19, no qual é mostrado o valor contido em R[6], que será o valor do n-ésimo termo da sequência de Fibonacci, isso porque, caso esse salto não seja realizado, primeiramente o valor contido em R[13] é incrementado, em seguida o registrador R[6] recebe o valor do registrador R[4] somado ao do registrador R[5], R[4] recebe o valor de R[5], e por fim R[5] recebe o valor de R[6], de forma que sempre o novo termo calculado na sequência se encontra no registrador R[6]. Assim esse laço é realizado n-3 vezes, sendo n o n-ésimo termo a ser mostrado, de forma que ao seu término, tal termo se encontrará no registrador R[6], que é por fim mostrado no *display* na instrução contida no endereço 19.

A efetividade da execução desse algoritmo pode ser provada com base nas imagens a seguir:

Figura 18 – Fibonacci na FPGA - 1

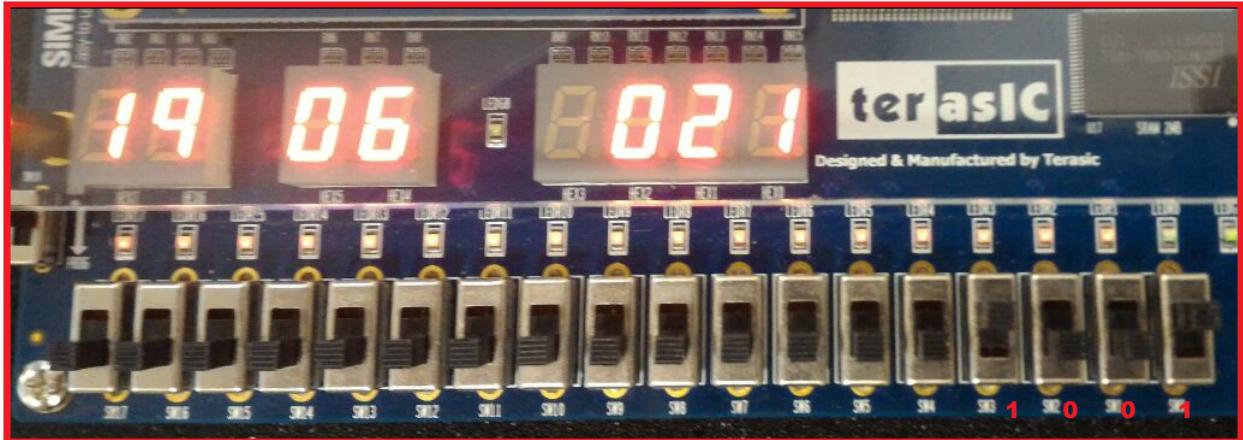


Fonte: Autor

Na Figura 18 pode ser visto no primeiro par de *displays* o número 18 sendo exibido,

que é o endereço da instrução armazenado no PC; nos *displays* seguintes pode ser visto o número 1, que é o número correspondente ao registrador do qual está sendo realizado *output*, e por fim, o número 1 nos *displays* mais à direita representam o valor armazenado no registrador R[1]. Para isso os *switches* foram setados como para representar o terceiro número da sequência (0011), como pode ser visto marcado abaixo dos primeiros à direita na figura.

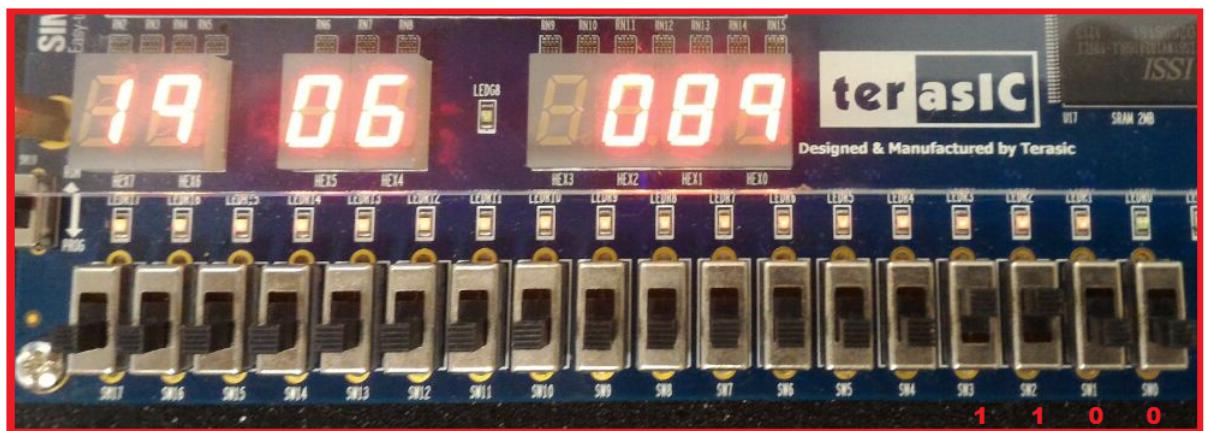
Figura 19 – Fibonacci na FPGA - 2



Fonte: Autor

De maneira similar à imagem anterior, na imagem Figura 19 são mostrados os números 19, 6 e 21, que representam, respectivamente, o endereço da instrução do PC, o registrador do qual está sendo realizado o *output* e por fim o valor contido em tal registrador. Nesse caso, o valor colocado como *input* pelos *switches* da FPGA representam o nono termo da sequência (1001), como pode ser visto na imagem.

Figura 20 – Fibonacci na FPGA - 3



Fonte: Autor

Por fim, semelhantemente às imagens anteriores, a última, Figura 20 mostra o cálculo do décimo-segundo termo da sequência (1100), como pode ser visto nos *switches*.

#### 4.4.2 Algoritmo 2 - Funções de acesso à memória

Esse segundo algoritmo, muito mais simples do que o descrito anteriormente, foi criado unica e exclusivamente para demonstrar a funcionalidade das instruções de acesso à memória (*load/store*), juntamente com algumas outras instruções básicas ainda não utilizadas. O pseudocódigo referente à esse algoritmo pode ser visto na Tabela 10.

Tabela 10 – Acesso à memória - Pseudocódigo

(0) - nop
(1) - nop
(2) - R[2] ← 2 - ldi
(3) - LED ← R[2] - out
(4) - R[1] ← 45 - ldi
(5) - LED ← R[1] - out
(6) - R[1] += 10 - addi
(7) - LED ← R[1] - out
(8) - R[1] -= 20 - subi
(9) - LED ← R[1] - out
(10) - R[1] = R[1] * R[2] - mult
(11) - LED ← R[1] - out
(12) - M[R[0] + 0] ← R[1] - str
(13) - R[6] ← M[R[0] + 0] - ld
(14) - LED ← R[6] - out

Fonte: Autor

Nesse algoritmo, primeiro inicializa-se o valor do registrador R[2] como 2 e o do registrador R[1] como 45, em seguida, adiciona-se 10 ao valor contido em R[1], e depois dele também é subtraído 20. Assim tal valor resultante então é multiplicado por 2 e armazenado no mesmo registrador R[1]; em sequida ocorrem as funções de acesso à memória: o valor contido R[1] é armazenado na posição 0 da memória e então retirado da mesma e colocado no registrador R[6], de onde é feito o *output* de seu valor para os *displays* da FPGA no endereço 14. Vale ressaltar que as operações de *output* contidas nos endereços 3, 5, 7, 9 e 11 tiveram suas explicações desconsideradas por motivos óbvios. E apenas como desencargo: ao final do algoritmo, o valor contido em ambos os registradores R[1] e R[6] é igual a 70 ( $(45 + 10 - 20) * 2 = 70$ ). A efetividade do funcionamento desse algoritmo pode ser visto na Figura 21.

Figura 21 – FPGA - Acesso à memória



Fonte: Autor

Na figura acima, da esquerda para a direita, o primeiro número mostrado representa o endereço da instrução no PC, o segundo indica o registrador do qual se está sendo realizado o *output* e por fim, no terceiro conjunto de *displays*, o valor contido em tal endereço, que como descritos anteriormente e mostrados no pseudocódigo da Tabela 10, são respectivamente iguais a 14, 6 e 70.

## 5 Considerações Finais

O projeto descrito por este relatório foi de extrema importância para o desenvolvimento do pensamento computacional dos alunos, uma vez que ele foi responsável por demonstrar o quão desafiador e literalmente difícil pode ser a implementação de um sistema computacional em mais baixo nível, explicitando o fato de que os mínimos detalhes em suas unidades, sejam elas quais forem, podem levar a erros e até mesmo possíveis mudanças na arquitetura.

As maiores dificuldades encontradas no desenvolvimento deste projeto foram o fato de exceções ocorrerem no caminho de dados, ou seja, o fato de instruções semelhantes não terem sempre as mesmas *flags* devido ao fato destas dependerem de sinais resultantes do próprio circuito, juntamente com o fato de que foi necessário por parte do aluno ter total conhecimento de sua arquitetura para garantir um bom funcionamento do circuito. Outro ponto desafiador que vale a pena ser ressaltado é o tratamento da entrada e saída de dados na etapa de simulações dos resultados na FPGA, pois mesmo garantido o funcionamento de todas as outras instruções do ISA, o funcionamento das instruções específicas a respeito de tal tratamento possuem funcionamento muito diferente das outras.

Contudo, por mais trabalhoso que possa ter sido a implementação do projeto no geral, nele foi desenvolvido tanto o corpo quanto o cérebro do processador, tanto a Unidade de Processamento quanto sua Unidade de Controle, e garantido o seu total funcionamento. Sendo assim ele foi de suma importância para estimular a abstração da operação lógica e sequêncial de circuitos digitais em um nível de complexidade superior ao normal.

O processador desenvolvido por este projeto conta com um conjunto de instruções altamente expansível, de maneira que se tornará fácil a implementação de possíveis novas instruções para usos futuros ou pequenas mudanças em sua arquitetura-base, para qualquer que seja o motivo.

## Referências

- 1 PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design*. 5th edition. ed. Waltham/MA, EUA: Morgan Kaufmann, 2007. Citado 3 vezes nas páginas 7, 12 e 14.
- 2 STALLINGS, W. *Computer Organization and Architecture*. 9th edition. ed. Upper Saddle River/New Jersey, EUA: Pearson, 2013. Citado 2 vezes nas páginas 8 e 10.
- 3 HEURING, M. M. e V. *Introdução à Arquitetura de Computadores*. 1999. <<https://www.gta.ufrj.br/ensino/EEL580/apresentacoes/Parte1.pdf>>. [Online, acessado 6/05/2017]. Citado na página 10.
- 4 BRITTON, R. *Introduction to the MIPS Architecture*. 2013. <<http://web.engr.oregonstate.edu/~walkiner/cs271-wi13/slides/02-MIPSArchitecture.pdf>>. [Online, acessado 5/05/2017]. Citado na página 10.
- 5 VERILOG.COM. *Verilog*. 2012. <<http://www.verilog.com/>>. [Online, acessado 5/05/2017]. Citado na página 14.
- 6 ALTERA. *Quartus*. 2011. <[https://www.altera.com/content/dam/altera-www/global/en\\_US/pdfs/literature/manual/quartus2\\_introduction.pdf](https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/quartus2_introduction.pdf)>. [Online, acessado 6/05/2017]. Citado na página 15.
- 7 WIKIPEDIA.COM. *FPGA, Field Programmable Gate Array*. 2012. <[https://pt.wikipedia.org/wiki/Field-programmable\\_gate\\_array](https://pt.wikipedia.org/wiki/Field-programmable_gate_array)>. [Online, acessado 5/05/2017]. Citado na página 15.

# Apêndices

# APÊNDICE A – Apêndice 1

Código completo da Unidade de Controle:

Algoritmo A.1 – controlUnit

```

module controlUnit (check ,
                     clock ,
                     opcode ,
                     PRBMs,
                     PAMselect ,
                     bmSelect ,
                     hlt ,
                     jmp ,
                     beq ,
                     bneq ,
                     rst ,
                     pdmms,
                     wf ,
                     wm,
                     ALUcode ,
                     imControl);

input check , clock ;
input [5:0] opcode ;
output reg PRBMs;
output reg PAMselect ;
output reg [1:0] bmSelect ;
output reg hlt , jmp , rst , beq , bneq ;
output reg pdmms;
output reg wf;
output reg wm;
output reg[4:0] ALUcode ;
output reg imControl ;

always @ (opcode or check) begin
    case(opcode[5:0])
        6'b000000: begin //ADD - OK - 0
            PRBMs = 1'b1;
            PAMselect = 1'b0;
            bmSelect = 2'b00;
            hlt = 1'b0;
            jmp = 1'b0;
            beq = 1'b0;
            bneq = 1'b0;
            rst = 1'b0;
            pdmms = 1'b1;
            wf = 1'b0;
            wm = 1'b1;
            ALUcode = 5'b00000;
            imControl = 1'b0;
        end
        6'b000001: begin //SUB - OK - 1
            PRBMs = 1'b1;
            PAMselect = 1'b0 ;
        end
    endcase

```

```

bmSelect = 2'b00;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b1;
wf = 1'b0;
wm = 1'b1;
ALUcode = 5'b00001;
imControl = 1'b0;
end

6'b000010: begin //ADDI - OK - 2
PRBMs = 1'b0;
PAMselect = 1'b1;
bmSelect = 2'b00;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b1;
wf = 1'b0;
wm = 1'b1;
ALUcode = 5'b00000;
imControl = 1'b0;
end

6'b000011: begin //SUBI - OK - 3
PRBMs = 1'b0;
PAMselect = 1'b1;
bmSelect = 2'b00;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b1;
wf = 1'b0;
wm = 1'b1;
ALUcode = 5'b00001;
imControl = 1'b0;
end

6'b000100: begin //INC - OK - 4
PRBMs = 1'b1;
PAMselect = 1'b0;
bmSelect = 2'b00;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b1;
wf = 1'b0;
wm = 1'b1;
ALUcode = 5'b00010;
imControl = 1'b0;
end

6'b000101: begin //DEC - OK - 5
PRBMs = 1'b1;
PAMselect = 1'b0;

```

```

bmSelect = 2'b00;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b1;
wf = 1'b0;
wm = 1'b1;
ALUcode = 5'b00011;
imControl = 1'b0;
end

6'b000110: begin //SLT - OK - 6
PRBMs = 1'b1;
PAMselect = 1'b0;
bmSelect = 2'b00;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b1;
wf = 1'b0;
wm = 1'b1;
ALUcode = 5'b00100;
imControl = 1'b0;
end

6'b000111: begin //SHFL - OK - 7
PRBMs = 1'b1;
PAMselect = 1'b0;
bmSelect = 2'b00;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b1;
wf = 1'b0;
wm = 1'b1;
ALUcode = 5'b00111;
imControl = 1'b0;
end

6'b001000: begin //SHFR - OK - 8
PRBMs = 1'b1;
PAMselect = 1'b0;
bmSelect = 2'b00;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b1;
wf = 1'b0;
wm = 1'b1;
ALUcode = 5'b01000;
imControl = 1'b0;
end

6'b001001: begin //NOT - OK - 9
PRBMs = 1'b1;
PAMselect = 1'b0;

```

```

bmSelect = 2'b00;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b1;
wf = 1'b0;
wm = 1'b1;
ALUcode = 5'b01001;
imControl = 1'b0;
end

6'b001010: begin //AND - OK - 10
PRBMs = 1'b1;
PAMselect = 1'b0;
bmSelect = 2'b00;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b1;
wf = 1'b0;
wm = 1'b1;
ALUcode = 5'b01010;
imControl = 1'b0;
end

6'b001011: begin //OR - OK - 11
PRBMs = 1'b1;
PAMselect = 1'b0;
bmSelect = 2'b00;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b1;
wf = 1'b0;
wm = 1'b1;
ALUcode = 5'b01011;
imControl = 1'b0;
end

6'b001100: begin //XOR - OK - 12
PRBMs = 1'b1;
PAMselect = 1'b0;
bmSelect = 2'b00;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b1;
wf = 1'b0;
wm = 1'b1;
ALUcode = 5'b01100;
imControl = 1'b0;
end

6'b001101: begin //MULT- OK - 13
PRBMs = 1'b1;
PAMselect = 1'b0;

```

```

bmSelect = 2'b00;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b1;
wf = 1'b0;
wm = 1'b1;
ALUcode = 5'b00101;
imControl = 1'b0;
end

6'b001110: begin //DIV - OK - 14
PRBMs = 1'b1;
PAMselect = 1'b0;
bmSelect = 2'b00;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b1;
wf = 1'b0;
wm = 1'b0;
ALUcode = 5'b00110;
imControl = 1'b0;
end

6'b001111: begin //LD - OK - 15
PRBMs = 1'b0;
PAMselect = 1'b1;
bmSelect = 2'b00;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b0;
wf = 1'b0;
wm = 1'b1;
ALUcode = 5'b00000;
imControl = 1'b0;
end

6'b010000: begin //LDI - OK - 16
PRBMs = 1'b0;
PAMselect = 1'b1;
bmSelect = 2'b00;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b1;
wf = 1'b0;
wm = 1'b1;
ALUcode = 5'b00000;
imControl = 1'b0;
end

6'b010001: begin //STR - OK - 17 ;D
PRBMs = 1'b0;
PAMselect = 1'b1;

```

```

bmSelect = 2'b00;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b0;
wf = 1'b1;
wm = 1'b0;
ALUcode = 5'b00000;
imControl = 1'b0;
end

6'b010010: begin //BEQ - OK - 18
PRBMs = 1'b0;
PAMselect = 1'b0;
bmSelect = 2'b01;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b1;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b0;
wf = 1'b0;
wm = 1'b0;
ALUcode = 5'b00001;
imControl = 1'b0;
end

6'b010011: begin //BNEQ - OK - 19
PRBMs = 1'b0;
PAMselect = 1'b0;
bmSelect = 2'b01;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b1;
rst = 1'b0;
pdmms = 1'b0;
wf = 1'b0;
wm = 1'b0;
ALUcode = 5'b00001;
imControl = 1'b0;
end

6'b010100: begin //JMP - OK - 20
PRBMs = 1'b0;
PAMselect = 1'b0;
bmSelect = 2'b10;
hlt = 1'b0;
jmp = 1'b1;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b0;
wf = 1'b0;
wm = 1'b0;
ALUcode = 5'b00010;
imControl = 1'b0;
end

6'b010101: begin //JMPR - OK - 21
PRBMs = 1'b0;
PAMselect = 1'b0;

```

```

bmSelect = 2'b11;
hlt = 1'b0;
jmp = 1'b1;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b0;
wf = 1'b0;
wm = 1'b0;
ALUcode = 5'b00000;
imControl = 1'b0;
end

6'b010110: begin //NOP - OK - 22
PRBMs = 1'b1;
PAMselect = 1'b0;
bmSelect = 2'b00;
hlt = 1'b0;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b1;
wf = 1'b0;
wm = 1'b1;
ALUcode = 5'b00000;
imControl = 1'b0;
end

6'b010111: begin //HLT - OK - 23
PRBMs = 1'b0;
PAMselect = 1'b0;
bmSelect = 2'b00;
hlt = 1'b1;
jmp = 1'b0;
beq = 1'b0;
bneq = 1'b0;
rst = 1'b0;
pdmms = 1'b0;
wf = 1'b0;
wm = 1'b0;
ALUcode = 5'b00000;
imControl = 1'b0;
end

6'b011000: begin //IN - OK - 24

    if (check == 1) begin
        PRBMs = 1'b0;
        PAMselect = 1'b1;
        bmSelect = 2'b00;

        hlt = 1'b0;
        jmp = 1'b0;
        beq = 1'b0;
        bneq = 1'b0;
        rst = 1'b0;
        pdmms = 1'b1;
        wf = 1'b0;
        wm = 1'b1;
        ALUcode = 5'b00000;
        imControl = 1'b1;
    end

```

```

        else if (clock == 1) begin
            PRBMs = 1'b0;
            PAMselect = 1'b1;
            bmSelect = 2'b00;

            hlt = 1'b1;
            jmp = 1'b0;
            beq = 1'b0;
            bneq = 1'b0;
            rst = 1'b0;
            pdmms = 1'b1;
            wf = 1'b0;
            wm = 1'b1;
            ALUcode = 5'b00000;
            imControl = 1'b1;
        end

    end
6'b011001: begin //OUT - OK - 25

    if (check == 1) begin
        PRBMs = 1'b1;
        PAMselect = 1'b0;
        bmSelect = 2'b00;

        hlt = 1'b0;
        jmp = 1'b0;
        beq = 1'b0;
        bneq = 1'b0;
        rst = 1'b0;
        pdmms = 1'b1;
        wf = 1'b0;
        wm = 1'b0;
        ALUcode = 5'b00000;
        imControl = 1'b0;
    end
    else if (clock == 1) begin
        PRBMs = 1'b1;
        PAMselect = 1'b0;
        bmSelect = 2'b00;

        hlt = 1'b1;
        jmp = 1'b0;
        beq = 1'b0;
        bneq = 1'b0;
        rst = 1'b0;
        pdmms = 1'b1;
        wf = 1'b0;
        wm = 1'b0;
        ALUcode = 5'b00000;
        imControl = 1'b0;
    end
end
default: begin
    PRBMs = 1'b0;
    PAMselect = 1'b0;
    bmSelect = 2'b00;
    hlt = 1'b0;
    jmp = 1'b0;

```

```

        beq = 1'b0;
        bneq = 1'b0;
        rst = 1'b0;
        pdmms = 1'b0;
        wf = 1'b0;
        wm = 1'b0;
        ALUcode = 5'b00000;
        imControl = 1'b0;
    end
endcase
end

endmodule

```

Código do módulo utilizado para simulação final:

### Algoritmo A.2 – BMCOREv2

```

module BMCOREv2 (clkAuto ,
                  check ,
                  cent_out ,
                  dez_out ,
                  uni_out ,
                  switches ,
                  pin_button ,
                  hlt ,
                  negative ,
                  reset ,
                  dez_out_pc ,
                  uni_out_pc ,
                  dez_out_reg_address ,
                  uni_out_reg_address );

input wire check, pin_button, clkAuto, reset;
input wire [15:0] switches;
wire [31:0] vAddress;
wire [4:0] Address;
wire [9:0] adrsIn;
wire [9:0] adrsOut;
wire [31:0] DataWrite;
wire [31:0] data1, data2;
wire [31:0] ALUresult;
wire [31:0] Instruction;
wire [31:0] Extend26_32;
wire [31:0] Extend16_32;
wire [31:0] PAMoutput;
wire [31:0] DataMemOutput;
wire [15:0] InMuxOut;
wire [4:0] ALUcode, PRBMout;
wire [1:0] bmSelect;
wire zero, clkReal;
wire PRBMs, PAMselect, jmp, rst, pdmms, wf;
wire wm, beq, bneq, IMcontrol;
wire [3:0] cent, dez, uni, cent_pc, dez_pc;
wire [3:0] uni_pc, cent_reg_address;
wire [3:0] dez_reg_address, uni_reg_address;
wire [3:0] cent_a, dez_a, uni_a;
output wire hlt, negative;

```



```

RegBank RegisterBank (.readAddress1(Instruction[25:21]),
                      .readAddress2(Instruction[20:16]),
                      .writeAddress(PRBMout),
                      .dataWrite(DataWrite),
                      .writeMark(wm),
                      .clk(clkReal),
                      .data1(data1),
                      .data2(data2),
                      .hlt(hlt),
                      .displayAddress(Address),
                      .valueAddress(vAddress));

bin_BCD reg_address_value(.binario(Address),
                           .centena(cent_reg_address),
                           .dezena(dez_reg_address),
                           .unidade(uni_reg_address));

IN_OUT address_reg (.cent(cent_reg_address),
                     .dez(dez_reg_address),
                     .uni(uni_reg_address),
                     .hlt(hlt),
                     .dez_out(dez_out_reg_address),
                     .uni_out(uni_out_reg_address));

Extender_26_32 Extender26_32 (.extInput(Instruction[25:0]),
                                 .extOutput(Extend26_32));

Extender_16_32 Extender16_32 (.extInput(InMUXOut),
                               .extOutput(Extend16_32));

PreAluMux PreALUMultiplexer (.PAMinput1(data2),
                             .PAMinput2(Extend16_32),
                             .PAMoutput(PAMoutput),
                             .PAMselect(PAMselect));

ALU ArithmeticLogicUnity (.op(ALUcode),
                         .data1(data1),
                         .data2(PAMoutput),
                         .result(ALUresult),
                         .zero(zero),
                         .shamt(Instruction[10:6]));

DataMem DataMemory (.data(data2),
                    .address(ALUresult),
                    .clk(clkReal),
                    .DataMemOut(DataMemOutput),
                    .WriteFlag(wf));

PostDataMemMux PostDataMemoryMultiplexer (.PDMMinput1(DataMemOutput),
                                           .PDMMinput2(ALUresult),
                                           .PDMMoutput(DataWrite),
                                           .PDMMselect(pdmms));

BigMux BigMultiplexer (.zero(zero),
                       .BMselect(bmSelect),
                       .beq(beq),
                       .bneq(bneq));

```

```

        .extSum(Extend16_32),
        .extSignal(Extend26_32),
        .dataARegBank(data1),
        .PcOut(adrsOut),
        .bmOut(adrsIn));
    
```

```

bin_BCD bin_to_bcd_value(.binario(vAddress),
    .centena(cent),
    .dezena(dez),
    .unidade(uni),
    .neg(negative));
    
```

```

IN_OUT inptoutpt (.cent(cent),
    .dez(dez),
    .uni(uni),
    .hlt(hlt),
    .cent_out(cent_out),
    .dez_out(dez_out),
    .uni_out(uni_out));
    
```

**endmodule**

Exemplo de multiplexador usado no projeto, igual em todas as instanciações, apenas com nomes diferentes:

Algoritmo A.3 – *MUX*

```

module PostDataMemMux(PDMMinput1, PDMMinput2, PDMMoutput, PDMMselect);

input PDMMselect;
input [31:0] PDMMinput1, PDMMinput2;
output reg [31:0] PDMMoutput;

always @ (*) begin

    case(PDMMselect)
        1'b0 : PDMMoutput = PDMMinput1;
        1'b1 : PDMMoutput = PDMMinput2;
    endcase

end

endmodule
    
```

Exemplo de um dos extensores utilizados no projeto:

Algoritmo A.4 – *MUX*

```

module Extender_16_32(extInput, extOutput);

input [15:0] extInput;
output reg[31:0] extOutput;

always @ (*) begin

    if(extInput[15] == 1'b1)
        
```

```

        extOutput = {{16{1'b1}}}, extInput};
    else
        extOutput = {{16{1'b0}}}, extInput};
end

endmodule

```

Código em linguagem C++ utilizado para realizar a conversão do "assembly" para o binário do processador:

Algoritmo A.5 – InstructionConverter.cpp

```

#include<bits/stdc++.h>

std::string decimalToBinary(int n, std::string par)
{
    int x;
    int bin = 0, i = 1;
    std::string aux, ret;

    while(n != 0)
    {
        x = n%2;
        n = n/2;
        bin = bin + (x*i);
        i = i*10;
    }
    if(par == "r" || par == "R")//Rtype instructions
    {
        if(std::to_string(bin).size() != 5)
        {
            for(int i = std::to_string(bin).size(); i < 5 ; i++) {aux = aux + "0";}
            ret = aux + std::to_string(bin);
        }
        else ret = std::to_string(bin);
    }
    else if(par == "i" || par == "I")//Itype instructions
    {
        if(std::to_string(bin).size() != 16)
        {
            for(int i = std::to_string(bin).size(); i < 16 ; i++) {aux = aux + "0";}
            ret = aux + std::to_string(bin);
        }
        else ret = std::to_string(bin);
    }
    else if(par == "j" || par == "J")//Jtype instructions
    {
        if(std::to_string(bin).size() != 26)
        {
            for(int i = std::to_string(bin).size(); i < 26 ; i++) {aux = aux + "0";}
            ret = aux + std::to_string(bin);
        }
        else ret = std::to_string(bin);
    }

    return ret;
}

```

```

std::string find_type(std::string instr)
{
    std::string ret;

    if(instr == "add" || instr == "sub" || instr == "inc" || instr == "dec" ||
       instr == "slt" || instr == "shfl" || instr == "shfr" || instr == "not" ||
       instr == "and" || instr == "or" || instr == "xor" || instr == "mult" ||
       instr == "div")
        ret = "R";
    else if (instr == "addi" || instr == "subi" || instr == "ld" ||
              instr == "ldi" || instr == "str" || instr == "beq" ||
              instr == "bneq" || instr == "jmp" || instr == "nop" ||
              instr == "hlt" || instr == "in" || instr == "out")
        ret = "I";
    else if(instr == "jmp")
        ret = "J";
    else
        ret = "fim";

    return ret;
}

std::string RtypeInstruction(std::string instr)
{
    int aux;
    std::string opcode, RS, RT, RD, shamt, funct, out;
    std::string rs, rt, rd, shamt_c, comment, ini_instr = "32'b";

    if(instr == "add") // [add 4 1 6] => r[4] = r[1] + r[6]
    {
        opcode = "000000";
        funct = "000000";
        shamt = "00000";

        std::cin >> aux;
        rd = std::to_string(aux);
        RD = decimalToBinary(aux, "r");

        std::cin >> aux;
        rs = std::to_string(aux);
        RS = decimalToBinary(aux, "r");

        std::cin >> aux;
        rt = std::to_string(aux);
        RT = decimalToBinary(aux, "r");

        comment = "uu//ur[" + rd + "]=ur[" + rs + "]+ur[" + rt + "]";
        out = ini_instr + opcode + "_" + RS + "_" + RT + "_" + RD + "_" +
              shamt + "_" + funct + ";" + comment;
    }
    else if(instr == "sub") // [add 13 1 0] => r[13] = r[1] - r[0]
    {
        opcode = "000001";
        funct = "000000";
        shamt = "00000";

        std::cin >> aux;
        rd = std::to_string(aux);
        RD = decimalToBinary(aux, "r"); // register destiny
    }
}

```

```

    std::cin >> aux;
    rs = std::to_string(aux);
    RS = decimalToBinary(aux, "r"); // register source

    std::cin >> aux;
    rt = std::to_string(aux);
    RT = decimalToBinary(aux, "r");

    comment = "uu//ur[ " + rd + " ]=ur[ " + rs + " ]-ur[ " + rt + " ]";
    out = ini_instr + opcode + " " + RS + " " + RT + " " + RD + " " +
        shamt + " " + funct + ";" + comment;
}

else if(instr == "inc") // [inc 4 5] => r[4] = r[5] + 1
{
    opcode = "000100";
    funct = "000000";
    shamt = "00000";
    RT = "00000";

    std::cin >> aux;
    rd = std::to_string(aux);
    RD = decimalToBinary(aux, "r");

    std::cin >> aux;
    rs = std::to_string(aux);
    RS = decimalToBinary(aux, "r");

    comment = "uu//ur[ " + rd + " ]=ur[ " + rs + " ]+u" "1";
    out = ini_instr + opcode + " " + RS + " " + RT + " " + RD + " " +
        shamt + " " + funct + ";" + comment;
}

else if(instr == "dec") // [dec 4 5] => r[4] = r[5] - 1
{
    opcode = "000101";
    funct = "000000";
    shamt = "00000";
    RT = "00000";

    std::cin >> aux;
    rd = std::to_string(aux);
    RD = decimalToBinary(aux, "r");

    std::cin >> aux;
    rs = std::to_string(aux);
    RS = decimalToBinary(aux, "r");

    comment = "uu//ur[ " + rd + " ]=ur[ " + rs + " ]-u1";
    out = ini_instr + opcode + " " + RS + " " + RT + " " + RD + " " +
        shamt + " " + funct + ";" + comment;
}

else if(instr == "slt") // [slt 4 13 5] => if(r[13] < r[5]) ? r[4] = 1 : r[4] = 0
{
    opcode = "000110";
    funct = "000000";
    shamt = "00000";

    std::cin >> aux;
    rd = std::to_string(aux);
    RD = decimalToBinary(aux, "r");
}

```

```

    std::cin >> aux;
    rs = std::to_string(aux);
    RS = decimalToBinary(aux, "r");

    std::cin >> aux;
    rt = std::to_string(aux);
    RT = decimalToBinary(aux, "r");

    comment = "uu//uif(r[" + rs + "]<ur[" + rt + "])?ur[" + rd + "]=" + "u:ur[" + rd + "]=" +
    out = ini_instr + opcode + "_" + RS + "_" + RT + "_" + RD + "_" +
    shamt + "_" + funct + ";" + comment;
}

else if(instr == "shfl") //shfl 13 1 3 => r[13] = sl(3)r[1]
{
    opcode = "000111";
    funct = "000000";
    RT = "000000";

    std::cin >> aux;
    rd = std::to_string(aux);
    RD = decimalToBinary(aux, "r");

    std::cin >> aux;
    rs = std::to_string(aux);
    RS = decimalToBinary(aux, "r");

    std::cin >> aux;
    shamt_c = std::to_string(aux);
    shamt = decimalToBinary(aux, "r");

    comment = "uu//ur[" + rd + "]=sl(" + shamt_c + ")r[" + rs + "] ";
    out = ini_instr + opcode + "_" + RS + "_" + RT + "_" + RD + "_" +
    shamt + "_" + funct + ";" + comment;
}

else if(instr == "shfr") //shfr 13 1 3 => r[13] = sr(3)r[1]
{
    opcode = "001000";
    funct = "000000";
    RT = "000000";

    std::cin >> aux;
    rd = std::to_string(aux);
    RD = decimalToBinary(aux, "r");

    std::cin >> aux;
    rs = std::to_string(aux);
    RS = decimalToBinary(aux, "r");

    std::cin >> aux;
    shamt_c = std::to_string(aux);
    shamt = decimalToBinary(aux, "r");

    comment = "uu//ur[" + rd + "]=sr(" + shamt_c + ")r[" + rs + "] ";
    out = ini_instr + opcode + "_" + RS + "_" + RT + "_" + RD + "_" +
    shamt + "_" + funct + ";" + comment;
}

else if(instr == "not") //not 4 5 => r[4] = ~r[5]
{
    opcode = "001001";
    funct = "000000";
}

```

```

shamt = "00000";
RD = "00000";

std::cin >> aux;
rt = std::to_string(aux);
RT = decimalToBinary(aux, "r");

std::cin >> aux;
rs = std::to_string(aux);
RS = decimalToBinary(aux, "r");

comment = "uu//ur[ " + rt + " ]=ur[ " + rs + " ]";
out = ini_instr + opcode + "_" + RS + "_" + RT + "_" + RD + "_" +
shamt + "_" + funct + ";" + comment;
}

else if(instr == "and") // [and 1 2 3] => r[1] = r[2] & r[3]
{
    opcode = "001010";
    funct = "000000";
    shamt = "00000";

    std::cin >> aux;
    rd = std::to_string(aux);
    RD = decimalToBinary(aux, "r");

    std::cin >> aux;
    rs = std::to_string(aux);
    RS = decimalToBinary(aux, "r");

    std::cin >> aux;
    rt = std::to_string(aux);
    RT = decimalToBinary(aux, "r");

    comment = "uu//ur[ " + rd + " ]=ur[ " + rs + " ]&ur[ " + rt + " ]";
    out = ini_instr + opcode + "_" + RS + "_" + RT + "_" + RD + "_" +
shamt + "_" + funct + ";" + comment;
}

else if(instr == "or") // [or 1 2 3] => r[1] = r[2] / r[3]
{
    opcode = "001011";
    funct = "000000";
    shamt = "00000";

    std::cin >> aux;
    rd = std::to_string(aux);
    RD = decimalToBinary(aux, "r");

    std::cin >> aux;
    rs = std::to_string(aux);
    RS = decimalToBinary(aux, "r");

    std::cin >> aux;
    rt = std::to_string(aux);
    RT = decimalToBinary(aux, "r");

    comment = "uu//ur[ " + rd + " ]=ur[ " + rs + " ]|ur[ " + rt + " ]";
    out = ini_instr + opcode + "_" + RS + "_" + RT + "_" + RD + "_" +
shamt + "_" + funct + ";" + comment;
}

else if(instr == "xor") // [xor 1 2 3] => r[1] = r[2] ^ r[3]
{
}

```

```

{
    opcode = "001100";
    funct = "000000";
    shamt = "00000";

    std::cin >> aux;
    rd = std::to_string(aux);
    RD = decimalToBinary(aux, "r");

    std::cin >> aux;
    rs = std::to_string(aux);
    RS = decimalToBinary(aux, "r");

    std::cin >> aux;
    rt = std::to_string(aux);
    RT = decimalToBinary(aux, "r");

    comment = "/*r[" + rd + "]=r[" + rs + "]^r[" + rt + "]*/";
    out = ini_instr + opcode + "_" + RS + "_" + RT + "_" + RD + "_" +
        shamt + "_" + funct + ";" + comment;
}

else if(instr == "mult") // [mult 4 5 24] => r[4] = r[5] * r[24]
{
    opcode = "001101";
    funct = "000000";
    shamt = "00000";

    std::cin >> aux;
    rd = std::to_string(aux);
    RD = decimalToBinary(aux, "r");

    std::cin >> aux;
    rs = std::to_string(aux);
    RS = decimalToBinary(aux, "r");

    std::cin >> aux;
    rt = std::to_string(aux);
    RT = decimalToBinary(aux, "r");

    comment = "/*r[" + rd + "]*=r[" + rs + "]*r[" + rt + "]*/";
    out = ini_instr + opcode + "_" + RS + "_" + RT + "_" + RD + "_" +
        shamt + "_" + funct + ";" + comment;
}

else if(instr == "div") // [div 4 5 24] => r[4] = r[5] / r[24]
{
    opcode = "001110";
    funct = "000000";
    shamt = "00000";

    std::cin >> aux;
    rd = std::to_string(aux);
    RD = decimalToBinary(aux, "r");

    std::cin >> aux;
    rs = std::to_string(aux);
    RS = decimalToBinary(aux, "r");

    std::cin >> aux;
    rt = std::to_string(aux);
    RT = decimalToBinary(aux, "r");
}

```

```

comment = "uu//ur[ " + rd + " ]=ur[ " + rs + " ]u/ur[ " + rt + " ]";
out = ini_instr + opcode + " " + RS + " " + RT + " " + RD + " " +
shamt + " " + funct + ";" + comment;
}
return out;
}

std::string ItypeInstruction(std::string instr)
{
    int aux;
    std::string opcode, RS, RT, offset, out;
    std::string rs, rt, offs_c, comment, ini_instr = "32'b";

    if(instr == "addi") // [addi 4 5 76] => r[4] = r[5] + 76
    {
        opcode = "000010";

        std::cin >> aux;
        rt = std::to_string(aux);
        RT = decimalToBinary(aux, "r");

        std::cin >> aux;
        rs = std::to_string(aux);
        RS = decimalToBinary(aux, "r");

        std::cin >> aux;
        offs_c = std::to_string(aux);
        offset = decimalToBinary(aux, "i");

        comment = "uu//ur[ " + rt + " ]=ur[ " + rs + " ]+offs_c";
        out = ini_instr + opcode + " " + RS + " " + RT + " " + offset +
";" + comment;
    }
    if(instr == "subi") // [subi 4 5 76] => r[4] = r[5] - 76
    {
        opcode = "000011";

        std::cin >> aux;
        rt = std::to_string(aux);
        RT = decimalToBinary(aux, "r");

        std::cin >> aux;
        rs = std::to_string(aux);
        RS = decimalToBinary(aux, "r");

        std::cin >> aux;
        offs_c = std::to_string(aux);
        offset = decimalToBinary(aux, "i");

        comment = "uu//ur[ " + rt + " ]=ur[ " + rs + " ]-offs_c";
        out = ini_instr + opcode + " " + RS + " " + RT + " " + offset +
";" + comment;
    }
    if(instr == "ld") // [ld 6 1 13] => r[6] = m[r[1] + 13]
    {
        opcode = "001111";

        std::cin >> aux;
        rt = std::to_string(aux);
    }
}

```

```

RT = decimalToBinary(aux, "r");

std::cin >> aux;
rs = std::to_string(aux);
RS = decimalToBinary(aux, "r");

std::cin >> aux;
offs_c = std::to_string(aux);
offset = decimalToBinary(aux, "i");

comment = "uu//ur[ " + rt + " ]um[ r[ " + rs + " ]+" + offs_c + " ] ";
out = ini_instr + opcode + " " + RS + " " + RT + " " + offset +
";" + comment;
}

if(instr == "ldi") // [ldi 23 345] => r[23] = 345
{
    opcode = "010000";
    RS = "00000";

    std::cin >> aux;
    rt = std::to_string(aux);
    RT = decimalToBinary(aux, "r");

    std::cin >> aux;
    offs_c = std::to_string(aux);
    offset = decimalToBinary(aux, "i");

    comment = "uu//ur[ " + rt + " ]uu" + offs_c;
    out = ini_instr + opcode + " " + RS + " " + RT + " " + offset +
";" + comment;
}

if(instr == "str") // [str 0 24 6] => m[r[0] + 24] = r[6]
{
    opcode = "010001";

    std::cin >> aux;
    rs = std::to_string(aux);
    RS = decimalToBinary(aux, "r");

    std::cin >> aux;
    offs_c = std::to_string(aux);
    offset = decimalToBinary(aux, "i");

    std::cin >> aux;
    rt = std::to_string(aux);
    RT = decimalToBinary(aux, "r");

    comment = "uu//um[ r[ " + rs + " ]u+" + offs_c + " ]ur[ " + rt + " ] ";
    out = ini_instr + opcode + " " + RS + " " + RT + " " + offset +
";" + comment;
}

if(instr == "beq") // [beq 3 4 15] => if(r[3] == r[4]) jump to 15
{
    opcode = "010010";

    std::cin >> aux;
    rs = std::to_string(aux);
    RS = decimalToBinary(aux, "r");
}

```

```

std::cin >> aux;
rt = std::to_string(aux);
RT = decimalToBinary(aux, "r");

std::cin >> aux;
offs_c = std::to_string(aux);
offset = decimalToBinary(aux, "i");

comment = " if(r[" + rs + "] == r[" + rt + "]) jump to " + offs_c;
out = ini_instr + opcode + " " + RS + " " + RT + " " + offset +
"; " + comment;
}

if(instr == "bneq") // [bneq 4 2 9] => if(r[4] == r[2]) jump to 9
{
    opcode = "010011";

    std::cin >> aux;
    rs = std::to_string(aux);
    RS = decimalToBinary(aux, "r");

    std::cin >> aux;
    rt = std::to_string(aux);
    RT = decimalToBinary(aux, "r");

    std::cin >> aux;
    offs_c = std::to_string(aux);
    offset = decimalToBinary(aux, "i");

    comment = " if(r[" + rs + "] != r[" + rt + "]) jump to " + offs_c;
    out = ini_instr + opcode + " " + RS + " " + RT + " " + offset +
"; " + comment;
}

if(instr == "jmpr") // [jmpr 6] => jump to r[6]
{
    opcode = "010101";
    RT = "00000";
    offset = "00000000000000000000000000000000";

    std::cin >> aux;
    rs = std::to_string(aux);
    RS = decimalToBinary(aux, "r");

    comment = "jump to r[" + rs + "];";
    out = ini_instr + opcode + " " + RS + " " + RT + " " + offset +
"; " + comment;
}

if(instr == "nop")
{
    opcode = "010110";

    comment = " /nop";
    out = ini_instr + opcode + " " + "00000000000000000000000000000000" +
"; " + comment;
}

if(instr == "hlt")
{
    opcode = "010111";

    comment = " /hlt";
    out = ini_instr + opcode + " " + "00000000000000000000000000000000" +

```

```

    " ; " + comment;
}
if(instr == "in")
{
    opcode = "011000";

    std::cin >> aux;
    rt = std::to_string(aux);
    RT = decimalToBinary(aux, "r");

    comment = "uu//ur[ " + rt + " ]u=switches";
    out = ini_instr + opcode+ "_" + "00000" + "_" + RT + "_" +
    "0000000000000000" + ";" + comment;
}
if(instr == "out")
{
    opcode = "011001";

    std::cin >> aux;
    rs = std::to_string(aux);
    RS = decimalToBinary(aux, "r");

    comment = "uu//uLED=u[r[ " + rs + " ]";
    out = ini_instr + opcode + "_" + RS + "_" + "00000000000000000000000000" +
    ";" + comment;
}

return out;
}

std::string JtypeInstruction(std::string instr)
{
    int aux;
    std::string opcode, jumpAddress, out;
    std::string comment, address_c, ini_instr = "32'b";

    if(instr == "jmp") // [jmp 34] => jump to 34
    {
        opcode = "010100";

        std::cin >> aux;
        address_c = std::to_string(aux);
        jumpAddress = decimalToBinary(aux, "j");

        comment = "uu//ujumputo" + address_c;
        out = ini_instr + opcode + "_" + jumpAddress + ";" + comment;
    }

    return out;
}

int main()
{
    std::string type, instr, out, name;
    int cont = 0, flag = 0;

    while(std::cin >> instr)
    {
        // std::cin >> instr;
        type = find_type(instr);

```

```
if(type == "R" || type == "r")
    out = RtypeInstruction(instr);
else if (type == "I" || type == "i")
    out = ItypeInstruction(instr);
else if(type == "J" || type == "j")
    out = JtypeInstruction(instr);

std::cout << "instrmem[ " << cont << "] = " << out << std::endl;
cont++;
}

return 0;
}
```