

João Victor de Mesquita Cândido dos Santos

**Desenvolvimento de um Sistema
Computacional baseado na arquitetura MIPS
Monociclo**

São José dos Campos - Brasil

Julho de 2018

Resumo

O relatório em questão apresenta detalhes a respeito do desenvolvimento de um sistema computacional baseado na arquitetura *MIPS* monociclo utilizando a linguagem de descrição de *hardware Verilog* para sua implementação. Ao decorrer deste relatório serão expostos detalhes sobre a implementação em lógica programável da Unidade de Processamento definida a partir da arquitetura base que é composta de *Program Counter*, Memória de Instruções, Banco de Registradores, Unidade Lógica e Aritmética, Memória de Dados, Unidade de Controle, Módulo de Entrada e Módulo de Saída. Tal Unidade de Processamento pôde ter seu êxito comprovado através das simulações (*Waveforms*) abordando todos componentes no software *Quartus* e a validação do sistema como um todo através de testes no *hardware FPGA*.

Palavras-chaves: MIPS, arquitetura, verilog.

Listas de ilustrações

Figura 1 – Hierarquia de Memória	13
Figura 2 – Arquitetura Harvard	15
Figura 3 – Caminho de Dados	18
Figura 4 – Caminho de Dados	25
Figura 5 – <i>Waveform 1</i> : PC	47
Figura 6 – <i>Waveform 2</i> : Memória de Instruções	48
Figura 7 – ULA	48
Figura 8 – <i>Waveform 3</i> : ULA - Instruções de 1 a 6	49
Figura 9 – <i>Waveform 4</i> : ULA - Instruções de 7 a 9	49
Figura 10 – <i>Waveform 5</i> : ULA - Instruções de 10 a 12	49
Figura 11 – <i>Waveform 6</i> : Memória de Dados	50
Figura 12 – <i>Waveform 7</i> : Multiplexador	50
Figura 13 – <i>Waveform 6</i> : Extensor de Sinal	51
Figura 14 – <i>Waveform 7</i> : Simulação de Instruções no Processador - Parte 1	51
Figura 15 – <i>Waveform 8</i> : Simulação de Instruções no Processador com Módulo de Entrada e Unidade de Controle	55
Figura 16 – O FPGA	56
Figura 17 – Teste 1.1 - Entrada 1	59
Figura 18 – Teste 1.2- Entrada 1	60
Figura 19 – Teste 2- Entrada 4	61
Figura 20 – Teste 2.1- Entrada 4	62
Figura 21 – Teste 3- Entrada 15	63
Figura 22 – Teste 3.1- Entrada 15	64
Figura 23 – Teste - Entrada 1.3	67
Figura 24 – Teste - Entrada 2.3	68
Figura 25 – Teste - Entrada 3.3	69
Figura 26 – Teste - Entrada 4.3	70
Figura 27 – Teste - Saída com áreas iguais	71
Figura 28 – Teste - Entrada 3.2	72
Figura 29 – Teste - Entrada 4.2	73
Figura 30 – Teste - Saída com áreas diferentes	74
Figura 31 – Exemplo de uma instrução add	80
Figura 32 – Exemplo de uma instrução load word	81
Figura 33 – Exemplo de uma instrução beq	82

Lista de tabelas

Tabela 1 – Exemplo de um formato de Instruções do tipo R	16
Tabela 2 – Exemplo de um formato de Instruções do tipo I	17
Tabela 3 – Exemplo de um formato de Instruções do tipo J	17
Tabela 4 – Formato de Instruções do tipo R	22
Tabela 5 – Formato de Instruções do tipo I	22
Tabela 6 – Formato de Instruções do tipo J	22
Tabela 7 – Conjunto de Instruções	23
Tabela 8 – Conjunto de Operações Realizadas pela ULA	29
Tabela 9 – Sinais de Controle	36
Tabela 10 – Instruções de Teste	51
Tabela 11 – Instruções de Teste com Unidade de Controle	54
Tabela 12 – Instruções de Teste	57
Tabela 13 – Instruções do Algoritmo de Verificação de Igualdade da Área de Dois Triângulos	65

Sumário

1	INTRODUÇÃO	8
2	OBJETIVOS	10
2.1	Geral	10
2.2	Específico	10
3	FUNDAMENTAÇÃO TEÓRICA	12
3.1	Sistema Computacional	12
3.2	Conjunto de Instruções	12
3.3	Hierarquia de Memória	12
3.4	Conjunto de Instruções CISC VS RISC	14
3.5	Arquitetura	15
3.6	MIPS	15
3.6.1	Tipos de Instruções	16
3.6.1.1	Instruções do tipo R	16
3.6.1.2	Instruções do tipo I	16
3.6.1.3	Instruções do tipo J	17
3.6.2	O Caminho de dados MIPS	17
3.6.3	Contador de Programa	18
3.6.4	Memória de Instruções	18
3.6.5	Banco de Registradores	19
3.6.6	Unidade Lógica e Aritmética	19
3.6.7	Memória de dados	19
3.6.8	Unidade de Controle	19
3.6.9	Componentes Auxiliares	20
3.6.9.1	Extensor de Sinal	20
3.6.9.2	Multiplexadores	20
3.7	A Linguagem Verilog	20
3.8	Field Programmable Gate Array (FPGA)	21
4	DESENVOLVIMENTO	22
4.1	Formato de Instruções	22
4.2	Conjunto de Instruções	22
4.3	Modos de Endereçamento	23
4.4	Arquitetura do Processador	24
4.5	Implementação dos Componentes	25

4.5.1	Contador de Programa	25
4.5.2	Memória de Instruções	26
4.5.3	Banco de Registradores	27
4.5.4	Unidade Lógica e Aritmética	28
4.5.5	Memória de dados	30
4.5.6	Multiplexador	31
4.5.7	Extensor de Sinal	32
4.5.8	Entrada e Saída	32
4.5.8.1	Módulo de Entrada	32
4.5.8.2	Módulo de Saída	33
4.5.8.3	Temporizador	35
4.5.9	Unidade de Controle	36
4.5.10	Integração da Unidade de Processamento	44
5	RESULTADOS E DISCUSSÕES	47
5.1	Simulações Individuais	47
5.1.1	Contador de Programa	47
5.1.2	Memória de Instruções	48
5.1.3	Banco de Registradores	48
5.1.4	ULA	48
5.1.5	Memória de Dados	49
5.1.6	Multiplexador	50
5.1.7	Extensor de Sinal	50
5.1.8	Teste da Unidade de Processamento	51
5.2	Simulação Total	53
5.2.1	Simulação Total da Unidade de Processamento, Módulo de Entrada e Saída e Unidade de Controle	53
5.2.2	Simulação Total da Unidade de Processamento, Módulo de Entrada e Saída, Unidade de Controle e FPGA	56
5.2.3	Algoritmo de <i>Fibonacci</i>	57
5.2.4	Algoritmo de Verificação de Igualdade da Área de Dois Triângulos	64
6	CONSIDERAÇÕES FINAIS	75
	REFERÊNCIAS	77
	APÊNDICES	79
	APÊNDICE A – APÊNDICE 1	80

APÊNDICE B – APÊNDICE 2	83
B.1 Algoritmos de Componentes Auxiliares	83

1 Introdução

Segundo a definição do dicionário, o computador é um aparelho eletrônico usado para processar, guardar e tornar acessível uma informação de variados tipos (1). Os computadores vêm se tornando cada vez complexos, poderosos e desenvolvidos para desenvolver as mais diferenciadas tarefas. Com o avanço no desenvolvimento de tecnologias os computadores trazem um grande impacto e uma evolução na sociedade em diversas áreas. O uso de computadores na era das máquinas em que vivemos vem se tornando indispensável em várias atividades que são realizadas diariamente, fazendo com que a demanda por novas tecnologias ágeis, rápidas, com uma alta performance e baixo custo se torne cada vez mais crescente para agradar os usuários, entretanto para que isso ocorra é necessário entender qual é o funcionamento por trás dos sistemas computacionais, como são desenvolvidos e como cada componente é importante para que tudo flua da melhor maneira possível. Assim o desenvolvimento de sistemas computacionais traz consigo uma enorme importância para o mundo todo já que isso contribui para que a sociedade siga em constante evolução.

A especificação de um computador consiste na descrição do seu nível mais baixo, sua arquitetura, organização e conjunto de instruções. É importante ao descrever computadores diferenciar os conceitos entre arquitetura de computadores e organização de computadores. A arquitetura se refere aos atributos do sistema visíveis ao programador, ou seja, são as características que influenciam sobre a execução lógica de um programa, assim arquitetura se trata de atributos de *software*. Organização se refere às unidades operacionais e suas interconexões, ou seja, a organização é sobre detalhes de *hardware* transparentes ao programador (2).

O funcionamento de um computador tanto em alto nível quanto em baixo nível é bem complexo e para que este execute suas ações de maneira adequada e sem falhas é necessário um conjunto de instruções chamado de ISA(*Instruction Set Architecture*) que serve de fronteira entre o *hardware* e *software* de baixo nível(3). O conjunto de instruções precisa de um dispositivo chamado de Unidade de Processamento (UP), que faz o gerenciamento e tratamento das instruções. A Unidade de Processamento é capaz de atender a todas necessidades de um programa, é capaz de realizar operações lógicas e aritméticas, gerenciar entrada e saída de dados, fazer acessos à memória, etc.

Dentro da Unidade de Processamento se encontra outras subunidades como a ULA (Unidade Lógica e Aritmética), uma memória de dados, módulos de entrada e saída e a principal subunidade chamada de Unidade de Controle (UC). A Unidade de controle se comporta como o "cérebro" do computador, é responsável por fazer com que todos os

componentes da Unidade de Processamento funcionem corretamente, assim compete a UC fazer com que todos os componentes funcionem de forma sincronizada através de uma comunicação que ocorre por meio de sinais de controle.

Sendo assim, o projeto tem como foco o desenvolvimento e implementação de uma Unidade de Processamento utilizando a linguagem de descrição de *hardware Verilog* para que esta seja capaz de executar instruções definidas, validando o funcionamento do sistema computacional.

2 Objetivos

2.1 Geral

Desenvolvimento de um sistema computacional capaz de realizar operações de baseado em um processador na arquitetura *MIPS*. O projeto é desenvolvido através da construção da Unidade de Processamento em lógica programável utilizando a linguagem de descrição de *hardware Verilog* com auxílio do *software Quartus II* e por fim atingir o objetivo final através da integração com o *hardware FPGA* testando e simulando cada uma de suas definidas instruções e assim validar o seu funcionamento.

2.2 Específico

Para o desenvolvimento do trabalho foi necessário dividir em pequenas etapas:

- O tipo de arquitetura a ser utilizada quando se fala do conjunto de instruções e sua complexidade, sendo RISC ou CISC;
- O tipo de arquitetura quando se trata da comunicação da memória e o processador, sendo essa arquitetura seguindo o modelo *Harvard* ou o modelo de *Von Neumann*;
- Definição do conjunto de instruções e seu formato, bem como o tamanho das instruções;
- Definição dos modos de endereçamento;
- Esquemático do caminho de dados *Datapath*;
- Implementação do Banco de Registradores;
- Implementação da Unidade Lógica e Aritmética;
- Implementação da Memória de Dados;
- Implementação do Contador de Programa;
- Implementação da Memória de Instruções;
- Implementação dos multiplexadores e extensores de sinal;
- Integração de todos os componentes da Unidade de Processamento;
- Implementação do módulo de entrada e saída dos dados;

- Por fim, integração da Unidade de Processamento com o módulo de entrada e saída e os devidos testes utilizando o FPGA para validar o funcionamento do projeto.

3 Fundamentação Teórica

3.1 Sistema Computacional

É um conjunto de dispositivos eletrônicos capazes de processar informações a partir de um programa. Um sistema computacional é aquele que automatiza ou apoia a realização de atividades humanas através do processamento de informações (4).

3.2 Conjunto de Instruções

O conjunto de instruções é um dos pontos centrais na arquitetura de um processador pois representa as operações que o mesmo suporta, fornece ou disponibiliza ao programador, assim sendo, um conjunto de instruções é a representação em códigos simbólicos (mnemônicos) do código de máquina (5).

3.3 Hierarquia de Memória

Nos sistemas computacionais é de extrema importância a utilização de memórias, que são responsáveis pelo armazenamento dos dados que serão processados. Em um sistema ideal, as memórias deveriam ter uma ilimitada capacidade de armazenamento, acesso imediato de dados e baixo custo por bit, o que não acontece nos sistemas reais. Capacidade de armazenamento, velocidade e custo por bit são fatores que crescem proporcionalmente entre si (6).

As memórias são divididas em dois tipos: memórias voláteis e memórias não-voláteis. As memórias voláteis armazenam os dados gravados enquanto possui eletricidade, assim no momento em que não há mais eletricidade para alimentar todos os dados são perdidos. Já nas memórias não-voláteis os dados continuam sendo armazenados mesmo quando não há mais eletricidade alimentando.

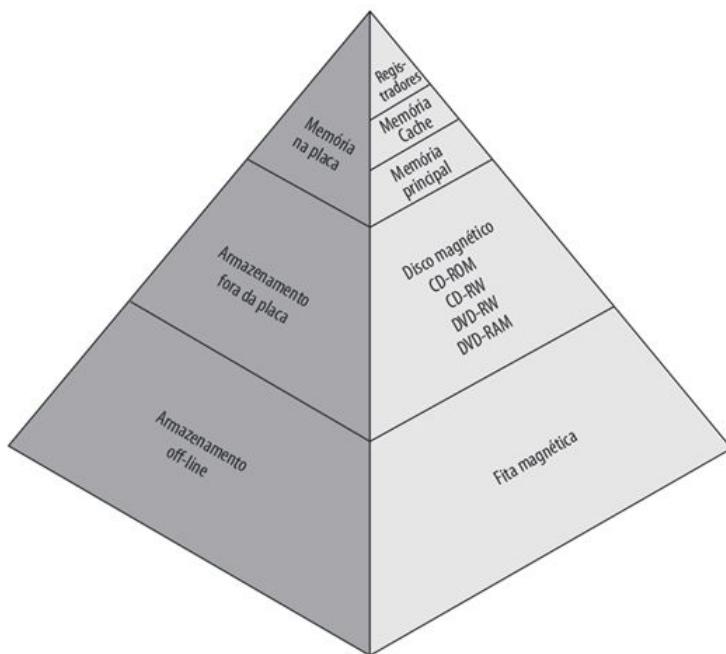
Assim, para se obter uma memória extremamente rápida e com grande capacidade de armazenamento demanda de um custo altíssimo. Visando esse problema de custo foi necessário balancear esses fatores desenvolvendo um sistema de organização de memórias. Para o armazenamento de grandes dados é necessário que haja uma Hierarquia de Memória (3), representada por uma pirâmide que relaciona e categoriza os vários tipos de memória de acordo com suas características como velocidade de acesso, capacidade de armazenamento e custo conforme mostra a Figura 1.

A hierarquia de memória tem como um dos principais objetivos aumentar a

velocidade de acesso aos dados no sistema, essa necessidade vem da diferença discrepante entre a velocidade dos processadores e dispositivos de memória(2).

De baixo para cima no primeiro nível da hierarquia se encontram as fitas magnéticas utilizadas para armazenamento *offline* e não-volátil com uma menor velocidade de acesso, subindo mais um nível se encontram as memórias de armazenamento externo, que armazenam dados que necessitam ser buscados antes de utilizados. Após se encontra a memória principal, também conhecida como memória RAM(*(Random Access Memory)*, memória volátil que apenas mantém os dados enquanto há eletricidade. Na penúltima posição se encontra a memória CACHE que é subdividida em 3 outros níveis de diferentes tamanhos e tem como objetivo de fazer a rápida troca de informações entre a memória principal e os registradores. No último nível se encontram os registradores que de maneira rápida armazenam dados processados pelas instruções que estão sendo executadas no sistema.

Figura 1 – Hierarquia de Memória



Fonte: Computer Organization and Architecture (5)

Como observado na Figura 1 conforme se sobe a pirâmide a velocidade de acesso de dados aumenta assim como o custo por *bit*, entretanto a capacidade de armazenamento diminui, assim quanto maior a capacidade de armazenamento, se tem um menor custo e uma menor velocidade de acesso

3.4 Conjunto de Instruções CISC VS RISC

Ao projetar um conjunto de instruções é importante iniciar com a escolha de uma entre as duas abordagens de arquitetura de instruções existentes, como a abordagem RISC e CISC. A arquitetura CISC (*Complex Instruction Set Computer*) é uma arquitetura complexa com muitas instruções diferentes. Os computadores CISC tem como objetivo a criação de arquiteturas complexas que facilitam na construção de compiladores(7), assim programas complexos são compilados em programas de linguagem de máquina(*assembly*) mais curtos. Assim com um menor número de linhas de código, os computadores CISC acessam menos a memória para buscar instruções(8) além de executar essas instruções em um número menor de ciclos de *clock* (9). A arquitetura CISC possui as seguintes características que a difere de outras arquiteturas:

1. Instruções complexas;
2. Conjunto de instruções grande;
3. Tamanho das instruções variado;
4. Muitos modos de endereçamento.
5. Instruções requerem múltiplos ciclos de *clock* para execução.

A arquitetura RISC (*Reduced Instruction Set Computer*) parte do pressuposto de um conjunto simples e pequeno de instruções em favor da rapidez de execução das mesmas (2). Em oposição aos computadores CISC a arquitetura RISC, as instruções mesmo sendo simples necessitam de programas maiores o que pode dificultar na construção de compiladores, entretanto instruções menores utilizam menos espaço de *hardware* e resulta numa Unidade de Controle simplificada, de baixo custo e rápida(7). A arquitetura RISC apresenta as seguintes características:

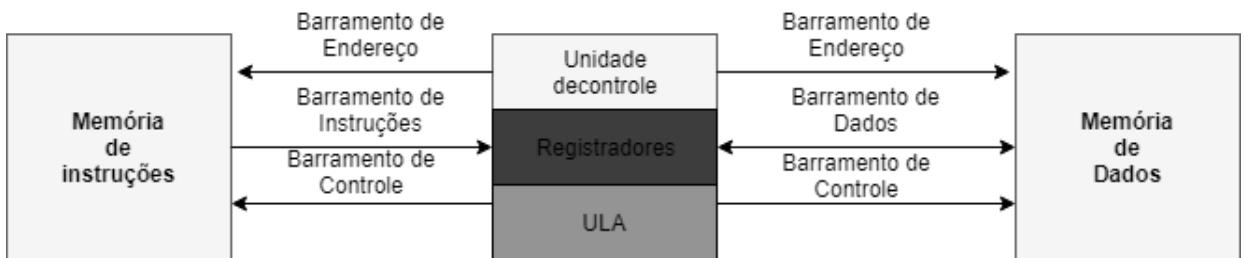
1. Instruções simples;
2. Conjunto de instruções reduzido;
3. Tamanho das instruções fixo;
4. Poucos modos de endereçamento;
5. Execução rápida de cada instrução (uma instrução por ciclo de *clock*).

Vale citar que nos últimos anos, as diferenças entre processadores CISC e RISC vem diminuindo pois processadores RISC têm incorporado funcionalidades complexas e processadores CISC têm adotado implementações baseadas em arquiteturas RISC.

3.5 Arquitetura

Quando se fala sobre arquitetura, se refere ao comportamento de um sistema computacional visível para o programador, ou seja, a execução lógica de uma dada instrução (5). Assim a arquitetura computacional se refere às unidades estruturais e seus relacionamentos lógicos e eletrônicos. Os computadores que se baseiam em um conjunto de instruções RISC utilizam a arquitetura *Harvard* que se difere de outras arquiteturas pois possui duas memórias diferentes e independentes, sendo elas a memória de dados e a memória de instruções permitindo que o computador execute uma instrução enquanto realiza a leitura de dados (10). Utilizando barramentos de dados das memórias, o processador é capaz de acessar as duas memórias simultaneamente afim de se obter um melhor desempenho. como é visto na Figura 2

Figura 2 – Arquitetura Harvard



Fonte: O Autor

3.6 MIPS

Responsável por ser o "cérebro" de um computador, o processador tem a função de supervisão e controle dos dados dentro de uma máquina bem como a realização das operações lógicas e aritméticas sobre os mesmos (11). Criada no início dos anos 80 na Universidade de Stanford, a arquitetura *MIPS* (*Microprocessor Without Interlocked Pipeline Stages*) segue características do padrão RISC simples, altamente escalável, possui número reduzido de instruções e registradores de propósito geral, fornecendo um alto nível de desempenho. Como o processador que será desenvolvido neste projeto, será baseado em uma arquitetura mais clássica, é conveniente mencionar os principais pontos característicos do *MIPS* :

1. Todas as operações da Unidade Lógica e Aritmética (ULA) são realizadas somente sobre registradores;
2. A memória é acessada somente através das instruções *Load* e *Store*;
3. Tamanho fixo do conjunto de instruções;

4. Apresenta cinco estágios de execução de uma instrução: busca, decodificação, execução, acesso à memória e escrita de dados.

3.6.1 Tipos de Instruções

As instruções na arquitetura *MIPS* foram divididas em três tipos: R,I e J. Cada tipo de instrução apresenta uma função diferente devido ao fato de ter diversos tipos de instruções e cada instrução têm um tamanho fixo de 32 bits.

3.6.1.1 Instruções do tipo R

O tipo de instrução R simboliza o uso de registradores e é nesse tipo que se encontram todas as instruções de operações lógicas e aritméticas tais como adição, subtração e multiplicação. A instrução é dividida em *Opcode*, RS, RT, RD, *shamt* e *funct* como pode ser visto na [Tabela 1](#). O *Opcode* é o identificador da instrução, assim ele é capaz de diferenciar uma instrução da outra pois cada uma tem seu próprio conjunto de bits para representar a operação a ser realizada; RS e RT representam os endereços de dois registradores fonte que contém os dados a serem utilizados; RD representa o endereço do registrador de destino que irá receber o resultado de uma operação; *shamt* (*shift amount*) é utilizado para instruções de deslocamento, assim tem a função de identificar a quantidade de bits a serem deslocados caso necessário; Por fim se tem o campo de *funct* que seleciona variações das operações especificadas pelo *Opcode*.

Tabela 1 – Exemplo de um formato de Instruções do tipo R

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

Fonte: O Autor

3.6.1.2 Instruções do tipo I

O tipo de instrução I como pode ser visto na [Tabela 2](#) simboliza instruções que executam operações lógicas e aritméticas com dados do *offset*, executam saltos condicionais e realizam acesso à memória. Como em todas instruções do *MIPS* se têm o campo de *Opcode*; RS representa o registrador fonte que contém o endereço de um registrador que contém o dado a ser utilizado; RT representa o endereço do registrador de destino no qual será armazenado o resultado de uma operação; Por fim o campo de *offset* contém o endereço de memória para possíveis saltos condicionais ou operações lógica e aritméticas.

Tabela 2 – Exemplo de um formato de Instruções do tipo I

Tamanho (bits)	6	5	5	16
Campo	<i>Opcode</i>	RS	RT	<i>offset</i>
Bits	31 - 26	25 - 21	20 - 16	15 - 0

Fonte: O Autor

3.6.1.3 Instruções do tipo J

O tipo de instruções J como pode ser visto na [Tabela 3](#) é o tipo mais simples e representa instruções de saltos. Contém apenas dois campos, o campo de *Opcode* para identificar a instrução e o campo *address* que representa o endereço de destino para onde irá saltar.

Tabela 3 – Exemplo de um formato de Instruções do tipo J

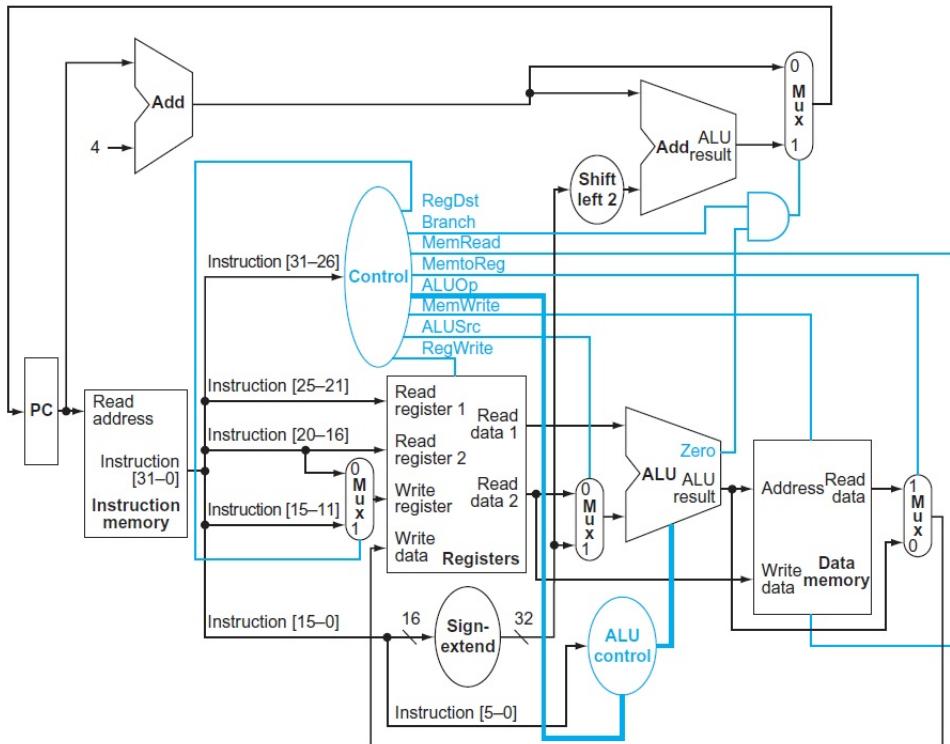
Tamanho (bits)	6	26
Campo	<i>Opcode</i>	<i>address</i>
Bits	31 - 26	25 - 0

Fonte: O Autor

3.6.2 O Caminho de dados MIPS

O caminho de dados, do inglês *datapath* consiste no caminho tomado pelos bits em uma determinada instrução executada pelo processador. Representa os componentes presentes no processador conectados entre si, formando uma estrutura capaz de executar as mais diversas instruções definidas pela arquitetura [\(2\)](#). Para a execução em cinco estágios da instrução na arquitetura *MIPS* os componentes presentes no caminho de dados são controlados através de sinais de controle, tais sinais têm como função direcionar os dados para o circuito de acordo com a instrução que está sendo executada. Os componentes principais que compõem uma implementação monociclo da arquitetura *MIPS* de acordo com a [Figura 3](#) são: *Program Counter* (Contador de Programa), Memória de Instruções, Banco de Registradores, Unidade Lógica e Aritmética, Memória de Dados e Unidade de Controle.

Figura 3 – Caminho de Dados



Fonte: *Computer Organization and Design* (2)

3.6.3 Contador de Programa

Contador de Programa é o componente responsável pela sequência das operações. Neste componente existe um registrador que armazena o endereço da instrução atual que está sendo executada, assim sempre na subida de *clock* seu valor é atualizado. A atualização do valor do contador é feita através dos sinais de controle que realiza soma de uma unidade ao valor do registrador ou substitui por algum outro valor caso ocorra alguma instrução de salto, assim a próxima instrução pode ser localizada através de uma busca na memória de instruções.

3.6.4 Memória de Instruções

A memória de instruções têm como objetivo armazenar todas as instruções que serão realizadas pelo processador, recebe como parâmetro de entrada um endereço proveniente do PC, com esse endereço é possível localizar a instrução e a enviar para outras partes do processador.

3.6.5 Banco de Registradores

Registradores são componentes compostos de um conjunto de *flip flops* capazes de armazenar valores de dados do programa, dados de entrada e endereços. O Banco de Registradores é um componente responsável por armazenar dados temporários provenientes da execução das instruções da arquitetura. O componente contém um total de 32 de registradores de propósito geral e cada registrador têm um tamanho de 32 bits, sendo que esses registradores podendo ser constantemente escritos e reescritos sempre que necessário levando em conta o endereço de entrada de cada instrução .

3.6.6 Unidade Lógica e Aritmética

A Unidade Lógica e Aritmética (ULA) é um conjunto combinacional que tem o papel de realizar as operações lógicas e aritméticas sobre os dados que estão armazenados no registrador. A ULA possui um *opcode* para diferenciar e determinar qual operação será executada. Para que seja realizada uma operação aritmética, é implementado um circuito aritmético composto de somadores completos ligados em cascata com isso é possível efetuar operações de soma, subtração, multiplicação que é nada mais uma sequência de somas e divisão que é uma sequência de subtrações. Para a realização de uma operação lógica como AND, OR, XOR e NOT será necessário um circuito composto por portas lógicas produzindo um resultado de comparação bit a bit e armazenar o resultado em um registrador de destino no Banco de Registradores.

3.6.7 Memória de dados

Tendo um funcionamento semelhante ao Banco de Registradores, a Memória de Dados é responsável por armazenar dados das operações realizadas pelo sistema. Acessada apenas pelas instruções *load word* e *store word* a memória contém um vetor que armazena os dados, sendo cada posição do vetor um análogo ao endereço de onde o dado está armazenado, a diferença é que nesse componente há uma *flag* que simboliza a escrita de um dado.

3.6.8 Unidade de Controle

A Unidade de Controle é responsável por gerenciar os sinais de controle do processador em cada instrução direcionando todo o fluxo de dados.

O gerenciamento só é possível porque todas as instruções são enviadas para a Unidade de Controle fazendo com que ela possa identificar todos os sinais que serão necessários em cada componente do processador para que o fluxo dos dados ocorra de maneira correta.

3.6.9 Componentes Auxiliares

subsubsection Bloco de *Input/Output*

O Bloco de *Input/Output* representa as ações de entrada e saída de dados do processador. O bloco de *Input* representam os *switches* do *kit* FPGA aonde o dado que entra é armazenado em um registrador no Banco de Registradores. O bloco de *Output* representa o *display* de 7 segmentos do *kit* FPGA, aonde será mostrado o resultado de alguma operação onde o dado está salvo em uma posição da Memória de Dados.

3.6.9.1 Extensor de Sinal

O extensor de sinal é muito importante pois pode ocorrer a existência de palavras que não tenham um tamanho 32 *bits* fazendo com haja um problema com a comunicação de dados. Assim, o papel do extensor é fazer com que seja extraído o subconjunto de interesse afim de transformar o dado em um dado de 32 *bits*.

3.6.9.2 Multiplexadores

Os multiplexadores são controlados através dos sinais de controle e com isso é possível determinar qual é a informação de dados que um respectivo bloco irá receber, assim os multiplexadores ajudam no controle do fluxo dos bits no processador.

3.7 A Linguagem *Verilog*

Verilog HDL(Hardware Description Language) é uma linguagem de descrição de *hardware* usada para projetar e documentar sistemas eletrônicos. *Verilog HDL* permite que os usuários possam projetar, verificar e implementar projetos nos mais variados níveis de abstração. ([12](#))

Uma das principais vantagens da modelagem de circuitos por linguagem descriptiva frente à modelagem por criação de blocos, é que desta maneira o desenvolvimento do projeto se torna independente da plataforma de desenvolvimento (*IDE*) na qual se está trabalhando.

A escolha dessa linguagem foi levada em consideração devido ao fato de ser suportada pelo *software Intel Quartus Prime* que é capaz de simular o funcionamento do circuito através de formas de onda (*waveforms*). Além de que a similaridade da linguagem com outras linguagens de programação torna o desenvolvimento do código mais rápido e prático provovendo um ganho de desempenho.

3.8 *Field Programmable Gate Array (FPGA)*

Com placas de desenvolvimento baseadas nos Circuitos Integrados (*CI's*) específicos é possível descarregar o código gerado nessa linguagem para matrizes de portas lógicas combinacionais (*gates*) e sequenciais (*flip-flops*) e/ou seus híbridos, que constituem o que se chama de uma célula padrão (*standard cells*), como por exemplo em *FPGAs* (*field programmable gate array* - ou matriz de portas programáveis).(13)

Os CI's programáveis como o FPGA são constituídos por milhares de blocos lógicos e fazem basicamente o mesmo que vários circuitos integrados tais como sensores e memória possibilitando a realização de simulações e servir de interface entre dispositivos lógicos e digitais. Assim o compilador interpreta o código e a ferramenta de programação realiza a gravação da lógica que define as interconexões desses blocos de modo a atender à descrição textual modelada na linguagem.(12)

4 Desenvolvimento

4.1 Formato de Instruções

De acordo com o que foi dito anteriormente, a arquitetura que foi desenvolvida no projeto é semelhante a arquitetura *MIPS* monociclo, bem como seu formato de instruções. Entretanto a única diferença é que nas instruções do tipo R não foi utilizado o campo de *funct*. Tais formatos de instruções podem ser vistos a seguir conforme a [Tabela 4](#), [Tabela 5](#) e [Tabela 6](#), tais formatos não necessitam de explicação pois já foram abordados na seção de Fundamentação Teórica.

Tabela 4 – Formato de Instruções do tipo R

Tamanho (bits)	6	5	5	5	5	6
Campo	Opcode	RS	RT	RD	<i>shamt</i>	não utilizado
Bits	31 - 26	25 - 21	20 - 16	15 - 11	10 - 6	5 - 0

Fonte: O Autor

Tabela 5 – Formato de Instruções do tipo I

Tamanho (bits)	6	5	5	16
Campo	Opcode	RS	RT	Endereço de Desvio
Bits	31 - 26	25 - 21	20 - 16	15 - 0

Fonte: O Autor

Tabela 6 – Formato de Instruções do tipo J

Tamanho (bits)	6	26
Campo	Opcode	Endereço de Salto
Bits	31 - 26	25 - 0

Fonte: O Autor

4.2 Conjunto de Instruções

Como este projeto é baseado na arquitetura *MIPS*, foram escolhidas 25 instruções para o desenvolvimento do projeto, afim de suprir todas as necessidades básicas do processador. As instruções são similares aos três formatos da arquitetura *MIPS* (R, I e J) respeitando suas especificações definidas pela [Tabela 4](#), [Tabela 5](#) e [Tabela 6](#) já apresentadas anteriormente. Todas as instruções apresentam um tamanho fixo de 32 bits e podem realizar operações lógicas, aritméticas, de salto e acesso à memória, como pode

ser visto na [Tabela 7](#). Abaixo segue uma legenda para facilitar na compreensão dos dados apresentados na tabela:

1. RS e RT representam os registradores fonte;
2. RD representa o registrador de destino;
3. IM representado um valor de imediato;
4. PC representa o *Program Counter* ou Contador de Programa;
5. M representa um acesso a memória;
6. R representa o uso de Registradores;

Tabela 7 – 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	$R[RD] \leftarrow R[RS]+IM$
subtração com imediato	subi	000011	$R[RD] \leftarrow R[RS]-IM$
multiplicação	mult	000100	$R[RD] \leftarrow R[RS] \times R[RT]$
divisão	div	000101	$R[RD] \leftarrow R[RS] / R[RT]$
set on less than	slt	000110	$\text{if}(R[RS] < R[RT]) R[RD]=1$
not	NOT	000111	$R[RD] \leftarrow \neg R[RS]$
and	AND	001000	$R[RD] \leftarrow R[RS] \& R[RT]$
or	OR	001001	$R[RD] \leftarrow R[RS] R[RT]$
xor	XOR	001010	$R[RD] \leftarrow R[RS] \wedge R[RT]$
load word	lw	001011	$R[RT] \leftarrow M[IM+R[RS]]$
load imediato	ldi	001100	$R[RD] \leftarrow IM$
store word	sw	001101	$M[R[RS]+IM] \leftarrow R[RT]$
branch if equal	beq	001110	$\text{IF}(R[RS] == R[RT]) PC \leftarrow IM$
branch if not equal	bne	001111	$\text{IF}(R[RS] != R[RT]) PC \leftarrow IM$
jump	jmp	010000	$PC \leftarrow IM$
jump to register	jmpr	010001	$PC \leftarrow R[RS]$
not an operation	NOP	010010	não realiza operação
halt	HALT	010011	para o processador (contagem do pc)
input	IN	010100	$R[RS] \leftarrow IN$
output	OUT	010101	$OUT \leftarrow R[RS]$
move	MOVE	010110	$R[RD] \leftarrow R[RS]$
shift left	shl	010111	$R[RD] \leftarrow sl(shamt)R[RS]$
shift right	shr	011000	$R[RD] \leftarrow sr(shamt)R[RS]$

Fonte: O Autor

4.3 Modos de Endereçamento

Após definido um conjunto de instruções foi necessário definir quais seriam os tipos de endereçamento realizados pelo processador. Em relação a esses modos de endereçamento,

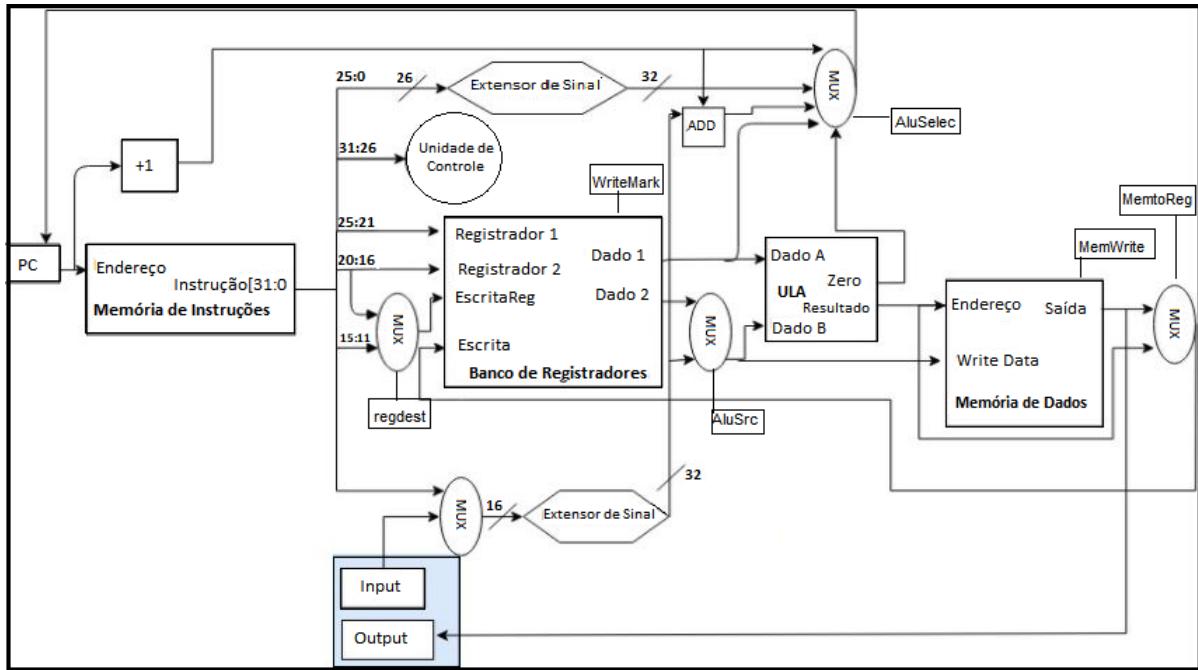
neste projeto foram utilizados três tipos existentes na arquitetura *MIPS*. Os modos de endereçamento escolhidos foram o de endereçamento imediato, endereçamento por registrador e endereçamento por registrador indireto que serão explicados a seguir.

- Endereçamento Imediato: É um tipo de endereçamento utilizado para definir valores para as variáveis. Neste tipo de endereçamento, o campo de endereço irá conter o próprio operando como endereço de memória, assim tem a vantagem de não fazer nenhuma referência de memória é necessária para se obter o operando. Entretanto apresenta a desvantagem de que o tamanho do valor é limitado ao tamanho do campo de endereço. (5).
- Endereçamento por Registrador: É um tipo de endereçamento utilizado quando se tem operandos armazenados em registradores, assim o campo de endereço do registrador se refere a um operando que está armazenado dentro desse registrador que está localizado no banco de registradores.
- Endereçamento por Registrador Indireto: É um tipo de endereçamento utilizado quando um registrador armazena um endereço que explicita uma posição da memória onde o operando está armazenado. É utilizado principalmente pela instrução *jump to register*.

4.4 Arquitetura do Processador

A arquitetura utilizada apresenta um *Datapath* semelhante ao da arquitetura *MIPS* como mostra a [Figura 4](#). No processador desenvolvido se encontram alguns componentes característicos já mencionados com o objetivo de se obter seu devido funcionamento de acordo com as instruções presentes no conjunto de instruções da [Tabela 7](#), o caminho que cada tipo de instrução e como ela se comporta no caminho de dados está presente no Apêndice 1. Os componentes principais do processador que já foram descritos anteriormente são o Contador de Programa (*Program Counter*), Memória de Instruções, Banco de Registradores, Unidade Lógica e Aritmética, Memória de Dados e foi adicionado um Bloco de *Input/Output* responsável pela entrada e saída de dados do processador. Além dos componentes principais o processador também apresenta seus componentes auxiliares tais como os multiplexadores e extensores de sinais, ambos os componentes podem ser visualizados na [Figura 4](#).

Figura 4 – Caminho de Dados



Fonte: O Autor

4.5 Implementação dos Componentes

A implementação de todos os componentes do caminho de dados como é visto na Figura 4 foi dada de maneira simples graças ao auxílio do software *Quartus Prime* com o qual o ambiente de desenvolvimento possibilitou a escrita dos códigos representando um circuito através da linguagem de descrição de hardware *Verilog* assim foi possível compilar os códigos e também realizar testes para comprovar a veracidade de cada um dos principais componentes, vale citar que a implementação dos componentes auxiliares se encontram no Apêndice B do presente relatório.

4.5.1 Contador de Programa

O contador de programa, como é visto no Algoritmo 1 abaixo, é responsável pelo fluxo das instruções no processador, nele há um registrador que tem a responsabilidade de armazenar o endereço da instrução atual que está em execução, assim sempre que há uma subida no *clock*, o valor é atualizado de acordo com alguns sinais de controle.

Há três possibilidades para que ocorra a atualização do valor do registrador. Na primeira situação o valor atual do *Pc* é incrementado com uma unidade para que haja a execução da próxima instrução, na segunda situação caso ocorra algum desvio seja ele de *branch* ou *jumo* é o valor do *Pc* é atualizado com o valor de endereço de desvio. Na terceira situação caso o sinal de *reset* seja igual a 1, o valor do registrador *Pc* é zerado

inciando a contagem novamente. Também há uma situação que utiliza um outro sinal de *halt*, caso esse sinal seja igual a 1, ocorre uma parada na contagem do *Pc* até que o usuário decida retomar a contagem do mesmo.

Algoritmo 1 - Contador de Programa

```

1
2 module PC(entrada,Pc,clock,reset,halt,zero,branchneq,branchneq,jump);
3
4 input clock, reset, halt, jump,branchneq,branchneq, zero;
5 input [31:0] entrada;
6 output reg[31:0] Pc;
7 wire [31:0] proximo,proximoinc;
8 assign proximo=entrada;
9 assign proximoinc= entrada+1;
10
11 always @ (posedge clock) begin
12
13     if (branchneq==1 && zero ==1)
14         Pc<=proximoinc;
15     else if (branchneq==1 && zero == 0)
16         Pc<=proximo;
17     else if (branchneq==1 && zero == 0)
18         Pc<=proximoinc;
19     else if (branchneq==1 && zero == 1)
20         Pc<=proximo;
21     else if (jump)
22         Pc<=proximo;
23     else if(reset)
24         begin
25             Pc = 0;
26         end
27     else if(halt)
28         begin
29             end
30     else
31         begin
32             Pc<=proximoinc;
33         end
34
35 end
36
37 endmodule

```

4.5.2 Memória de Instruções

A Memória de Instruções, como é vista no Algoritmo 2 abaixo, é um módulo bem simples, apresenta a capacidade de armazenar até 32 instruções e como segue o padrão *MIPS*, cada instrução tem um tamanho fixo de 32 *bits*. A entrada consiste em um valor representando um endereço referente a posição de memória aonde a instrução está, tal posição de memória é representado por um índice de um vetor. No algoritmo referente à memória de instruções há um exemplo de três instruções. Na primeira e segunda posição se encontram as representações em binário de uma instrução de *load immediate* e na terceira

posição está localizada a representação em binário de uma instrução de soma.

Algoritmo 2 - Contador de Programa

```

1  module MemoriadeInstrucoes(address,clock,instruction);
2
3  input [31:0] address;
4  input clock;
5  output [31:0] instruction;
6  integer flag=0;
7  reg [31:0] instrmem[31:0];
8
9  always @(posedge clock) begin
10
11      if(flag==0)
12          begin
13              instrmem[0] = 32'b010000_00000_00010_0000000000000000; // r[2] = 2
14              instrmem[1] = 32'b010000_00000_00001_0000000000000001; // r[1] = 2
15              instrmem[2] = 32'b000000_00011_00010_00100_000000000000; // r[3]= r[1]+r[2]
16              flag<=1;
17          end
18      end
19
20  assign instruction=instrmem[address];
21
22 endmodule

```

4.5.3 Banco de Registradores

O Banco de Registradores, como é visto no Algoritmo 3 abaixo, é responsável por armazenar os 32 registradores de propósito geral utilizados em todas as operações para o armazenamento de dados do programa, dados de entrada ou endereços. O banco foi implementado de tal forma que há um vetor de 32 posições com cada um a das posições desse vetor contendo um tamanho de 32 bits.

O banco apresenta quatro diferentes entradas e duas saídas, sendo três delas possíveis endereços de registradores de onde se deseja ler ou escrever dados. A primeira entrada é dada pelo endereço do primeiro registrador fonte, a segunda entrada se refere ao endereço do segundo registrador fonte, a terceira entrada de acordo com a Figura 4 pode receber o endereço do registrador de destino no caso de operações lógicas e aritméticas ou recebe um endereço para o registrador de onde se deseja inserir um valor de imediato. A quarta entrada se refere aos dados provenientes da memória de dados que serão armazenados no registrador de escrita. As duas saídas se referem aos dados que estão armazenados e serão lidos através do endereço dos dois primeiros registradores fonte.

A escrita em algum registrador é feita de acordo com um sinal de controle que caso tenha seu valor em nível alto (valor 1) sinaliza a escrita no registrados, caso contrário é feito a leitura.

Há apenas uma posição do banco no qual não há escrita de dados essa posição é a

primeira posição, de forma que qualquer dado que seja escrito nesse registrador o mesmo será perdido na próxima subida do *clock*.

Algoritmo 3 - Banco de Registradores

```

1  module bancoreg(Reg1,Reg2,writeRegister,regWrite,clock,data1,data2,writeMark);
2
3
4  input [31:0] regWrite;
5  input [4:0] writeRegister, Reg1,Reg2;
6  input clock,writeMark;
7  output [31:0] data1,data2;
8  reg [31:0] REG[31:0];
9
10 always @(posedge clock) begin
11     REG[0]=32'b0;
12     if(writeMark) // flag para escrita
13         REG[writeRegister] = regWrite;
14 end
15     assign data1=REG[Reg1];
16     assign data2=REG[Reg2];
17
18 endmodule

```

4.5.4 Unidade Lógica e Aritmética

A ULA (Unidade Lógica e Aritmética), como é vista no Algoritmo 4 abaixo, é responsável por todas as operações realizadas pelo processador, sejam elas operações lógicas ou operações aritméticas utilizando dos dados fornecidos pelo processador. A ULA apresenta um funcionamento independente do *clock* fazendo com que funcione apenas com os seus sinais de entrada.

A ULA pode realizar diferentes operações, assim para que possa haver uma diferenciação dessas operações o componente apresenta um sinal de controle de cinco *bits* chamado de *opcode* que irá selecionar a devida operação de acordo com o seu valor como é visto na Tabela 8.

Tabela 8 – Conjunto de Operações Realizadas pela ULA

<i>opcode</i>	instrução
00000	adição
00001	subtração
00010	<i>set on less than</i>
00011	multiplicação
00100	<i>branch if equal</i>
00101	divisão
00110	<i>shift left</i>
00111	<i>shift right</i>
01000	negação
01001	AND bit a bit
01010	OR bit a bit
01011	XOR bit a bit

Fonte: O Autor

Se tratando de operações aritméticas pode ocorrer uma situação de *overflow*, ou seja, um "estouro" de *bits*, isso foi tratado através de um sinal que comunica ao usuário que isso ocorreu. Essa medida teve de ser tomada pois o *software Quartus Prime* representa os valores de numeros binários em complemento de 2. Assim no caso da soma entre dois números números binários positivos e haja *overflow* o resultado da soma será negativo, o que acontece ao contrário na soma de dois números negativos em que o resultado será positivo na ocorrência de *overflow*. Caso a operação seja de subtração, há dois casos a serem analisados: o primeiro é se o primeiro número for positivo e o segundo negativo o resultado será positivo e o segundo caso é quando o primeiro número for negativo e o segundo positivo o sinal do resultado será positivo.

O algoritmo desenvolvido trata de forma efetiva a ocorrência de *overflow* para os casos de soma e subtração, entretanto esse problema ainda poderia ocorrer caso não houvesse um tratamento no caso de multiplicação de valores enormes e na divisão por zero, no primeiro caso a solução foi reduzir o tamanho dos operandos em 16*bits*, no segundo caso a solução foi transformar o divisor em um valor do qual a divisão é feita por 1.

No componente também há um sinal de controle chamado de *zero* utilizado nas instruções de saltos condicionais como *branch if equal* e *branch if not equal*, tal sinal de controle verifica se o valor das entradas são iguais ou diferentes para que haja um desvio ou não.

Algoritmo 4 - Unidade Lógica e Aritmética

```

1 module ULA(op,data1,data2,result,zero,shamt,of);
2
3 input [31:0] data1,data2;
4 input [4:0] op,shamt;
5 output reg [31:0] result;
6 output zero,of;
7 wire [31:0] not_zero,addd,subb;
8 wire resultadd,resultsub;
```

```

9 assign addd= data1+data2;
10 assign subb= data1-data2;
11 assign resultadd = (data1[31] == data2[31] && data1[31] != addd[31]) ? 1 : 0;
12 assign resultsub = (data1[31] == data2[31] && data1[31] != subb[31]) ? 1 : 0;
13 assign not_zero = (data2==0)? 1 : data2;
14 assign of = resultadd | resultsub;
15
16
17 always @ ( op or data1 or data2 or not_zero) begin
18     case (op[4:0])
19         5'b00000 : result = data1 + data2; //add
20         5'b00001 : result = data1 - data2; //sub
21         5'b00010 : result = data1 < data2 ? 1 : 0 ; //slt
22         5'b00011 : result= data1[15:0] * data2[15:0]; //mult
23         5'b00100 : result= data1 == data2 ? 0:1;//beq
24         5'b00101 : result= data1 / not_zero; //div
25         5'b00110 : result= data1 << shamt; //shl
26         5'b00111 : result= data1 >> shamt; //shr
27         5'b01000 : result= ~data1; //not
28         5'b01001 : result= data1 & data2; //and
29         5'b01010 : result= data1 | data2; //or
30         5'b01011 : result= data1 ^ data2; //xor
31         default : result= 0 ;
32     endcase
33 end
34 assign zero = (result==0);
35
36 endmodule

```

4.5.5 Memória de dados

A Memória de Dados, como é vista no Algoritmo 5 abaixo, é um dos últimos componentes que compõem o caminho de dados do processador em questão e é acessada apenas nas instruções de *load* e *store*. Tal componente é responsável pelo armazenamento de dados do sistema, funciona de maneira similar ao banco de registradores, apresentando um sinal de controle para controlar a escrita ou leitura da memória. Entretanto a diferença está nas entradas em que há apenas duas, a primeira é a do endereço que se deseja armazenar o dado e a segunda entrada é o dado que de fato será armazenado.

A escrita é feita na memória através da decisão de cada *clock* afim de evitar que haja a leitura de um dado que ainda não foi calculado. A saída da memória de dados é transmitida em todo o ciclo de *clock*, entretanto como na sua saída há um multiplexador que irá decidir se vai transmitir o dado lido da memória ou o resultado de uma operação realizada pela ULA, tal dado pode ser escrito ou não no banco de registradores.

Algoritmo 5 - Memória de Dados

```

1 module MemDados(Writedata,address,clock,dataOut,MemWrite);
2
3 input [31:0] Writedata,address;
4 input clock, MemWrite;
5 output [31:0] dataOut;

```

```

6
7 reg [31:0] Mem[9:0];
8
9 always @ (posedge clock) begin
10
11     if(MemWrite)
12         Mem[address] = Writedata;
13 end
14
15 assign dataOut = Mem[address];
16
17 endmodule

```

4.5.6 Multiplexador

O Multiplexador é um componente auxiliar porém muito importante para o fluxo dos dados dentro do processador, dentre todos os multiplexadores descritos na Figura 4 há um que é mais importante do que todos, que é o multiplexador que determina qual vai ser o endereço de retorno ao PC.

Neste componente há um sinal de controle que determina qual das suas entradas irá ser enviada o PC. Esse tratamento se fez necessário pois caso ocorra instruções de desvio como *branch if equal* e *branch if not equal* o sinal de controle *zero* por si só não seria suficiente para realizar o desvio pois uma delas é realizada caso o valor de *zero* seja 0 e a outra funciona quando o valor do sinal de controle for 1, assim há um total de quatro combinações possíveis para que possa ser feita a escolha certa da saída, vale citar que há outras duas entradas no multiplexador que é o sinal estendido de 32 bits e o próprio valor do PC conforme é mostrado no Algoritmo 6 abaixo. A implementação dos demais multiplexadores podem ser encontrados no Apêndice 2.

Algoritmo 6 - Multiplexador

```

1 module MUX4(zero,signalExtend,brancheq,branchneq,sumOut,regOut,PC,PCout,control);
2
3 input zero,brancheq,branchneq;
4 input [1:0] control;
5 input [31:0] PC, signalExtend, sumOut,regOut;
6 output reg[31:0] PCout;
7
8 always @ (zero or brancheq or branchneq or control or PC or signalExtend or sumOut or
9   regOut)
10
11     case(control[1:0])
12       2'b00:
13         begin
14           if(branchneq == 1 && zero == 1)
15               PCout = PC;
16           else if(branchneq == 1 && zero == 0)
17               PCout = sumOut;
18           else if(brancheq == 1 && zero == 1)
19               PCout= sumOut;
20           else if(brancheq == 1 && zero == 0)
21               PCout = signalExtend;
22         end
23     endcase
24 endmodule

```

```

20                                     PCout= PC;
21
22                     end
23             2'b01: PCout = signalExtend;
24             2'b10: PCout= PC;
25         endcase
26 endmodule

```

4.5.7 Extensor de Sinal

O Extensor de Sinal é um componente auxiliar utilizado para transformar um dado de 16 ou 16 $bits$ em um de 32 $bits$ pois todas as operações no processador só pode ser realizada com dados de 32 $bits$. No Algoritmo 7 abaixo está representado o extensor com a entrada de 16 $bits$, caso o *bit* mais significativo seja de valor lógico 1, isso significa que o dado tem valor negativo, sendo assim os demais *bits* devem ser completados com valor lógico alto, caso contrário o dado é preenchido com zeros formando assim uma palavra de 32 $bits$.

Algoritmo 7 - Extensor de Sinal

```

1 module Extensordebit(entrada,saida);
2
3     input [15:0] entrada;
4     output reg [31:0] saida;
5
6     always @(*) begin
7
8         if(entrada[15] == 1'b1) //caso for um numero negativo
9             saida= {{16{1'b1}}, entrada};
10        else
11            saida = {{16{1'b0}}, entrada};
12    end
13
14 endmodule

```

4.5.8 Entrada e Saída

4.5.8.1 Módulo de Entrada

O módulo de entrada, como o próprio nome já diz, é responsável pela entrada de algum dado no processador que é selecionado pelo usuário através a leitura das chaves(*switches*) no FPGA para que tal dado pode ser utilizado posteriormente em outras instruções.

Para que a leitura fosse feita de maneira correta foi preciso uma intervenção da Unidade de Controle para realizar uma interrupção no processador através do sinal de *halt* que para a contagem do *Program Counter* que deve permanecer assim até que o usuário intervenha, após selecionar todos os *switches* que representam o dado o usuário

deve selecionar outro switch significando que a escolha já foi feita e possa ocorrer a leitura dando prosseguimento para o restante das instruções.

No Algoritmo 8 abaixo é possível visualizar que o dado de *4 bits* é transformado em um de *16 bits*, isso ocorre porque como é visto na [Figura 4](#), tal dado entra em um multiplexador e posteriormente é transformado em um outro dado de *32 bits* que é o tamanho padrão para que seja possível realizar operações na arquitetura de maneira correta.

Algoritmo 8 - Módulo de Entrada

```

1 module IN(entrada,saida);
2
3 input [15:0] entrada;
4 output reg [15:0] saida;
5
6 always@(*) begin
7
8     saida = {entrada};
9
10 end
11 endmodule

```

4.5.8.2 Módulo de Saída

O módulo de saída é responsável por mostrar algum valor de saída para o usuário através dos *displays* do FPGA. No caso o módulo é implementado apenas como um conjunto de *displays* responsáveis para que possa ser feita a visualização de um dado de saída de *32 bits*, tal conjunto é composto por 4 *displays*, sendo cada um deles representando a unidade, dezena, centena e milhar do número. Para que isso ocorra de maneira correta a uma intervenção da Unidade de Controle que ao identificar a instrução de saída (instrução *OUT*) seleciona um dado que está armazenado na Memória de Dados e envia um sinal para o Módulo de Saída indicando que o dado está pronto para ser visto pelo usuário. Os *displays* são verificados a cada pulso de *clock*, enquanto o sinal de saída não mudar os mesmos ficaram ativos. Para que os resultados pudessem ser mostrados nos *displays*, foi necessário dividir o módulo em 3 algoritmos responsáveis por transformar o dado de *32bits* em suas devidas unidades já ditas anteriormente (milhar, centena, dezena e unidade), para isso cada unidade deve corresponder a *4bits* do dado representando um número decimal de 0 até 9, para que isso pudesse ser feito o dado foi deslocado de 4 bits para cada uma das unidades. Tendo feito isso o dado vai para um outro módulo que converte esse valor binário em um número BCD de 7 bits que será interpretado pelo *display de 7 segmentos*. Tais módulos podem ser vistos nos Algoritmos 9, 10 e por fim o Algoritmo 11 representa a redistribuição de cada valor para o seu respectivo *display*.

Algoritmo 9 - Módulo de Divisão do Dado

```

1 module binario_BCD (D_Binary, DATA_BCD);
2
3     input [15:0] D_Binary; //dado escolhido do registrador
4     output reg [15:0] DATA_BCD;// saida para os displays (unidade, dezena, centena e
5                                milhar)
6
7     integer cont;
8     reg [31:0] aux;
9
10    always @ (D_Binary) begin
11        aux={16'h0000, D_Binary};
12
13        for(cont=0; cont < 15; cont=cont + 1) begin
14            aux = aux << 1;
15            if(aux[19:16] > 4) aux[19:16] = aux[19:16] + 4'b0011;
16            if(aux[23:20] > 4) aux[23:20] = aux[23:20] + 4'b0011;
17            if(aux[27:24] > 4) aux[27:24] = aux[27:24] + 4'b0011;
18            if(aux[31:28] > 4) aux[31:28] = aux[31:28] + 4'b0011;
19
20        end
21        aux = aux << 1;
22        DATA_BCD = aux[31:16];
23    end
24
25 endmodule

```

Algoritmo 10 - Conversão de um número binário para BCD

```

1 module DISPLAYBCD(entrada, saidaBCD);
2     input wire [31:0] entrada;
3     output reg [6:0] saidaBCD;
4     always@(*)
5
6         begin
7             case (entrada)
8
9                 32'b0: saidaBCD = 7'b0000001;
10                32'b1: saidaBCD = 7'b1001111;
11                32'b10: saidaBCD = 7'b0010010;
12                32'b11: saidaBCD = 7'b00000110;
13                32'b100: saidaBCD = 7'b1001100;
14                32'b101: saidaBCD = 7'b0100100;
15                32'b110: saidaBCD = 7'b0100000;
16                32'b111: saidaBCD = 7'b0001111;
17                32'b1000: saidaBCD = 7'b0000000;
18                32'b1001: saidaBCD = 7'b0000100;
19
20            default: saidaBCD = 7'b1111111;
21        endcase
22    end
23 endmodule

```

Algoritmo 11 - Módulo de distribuição dos dados para os *displays*

```

1 module Display_BIG (binary, out, R1, R2, R3, R4, clock);
2
3     input [31:0] binary;

```

```

4      input  out ,clock;
5      output reg [6:0] R1 , R2 , R3 , R4;
6      wire  [15:0] aux;
7      wire  [6:0] A1 , A2 , A3 , A4;
8
9      binario_BCD binario_BCD (binary[15:0] , aux);
10
11
12      DISPLAYBCD U1 (aux[15:12] , A1); //unidade
13      DISPLAYBCD U2 (aux[11:8] , A2); //dezena
14      DISPLAYBCD U3 (aux[7:4] , A3); //centena
15      DISPLAYBCD U4 (aux[3:0] , A4); //mil
16
17      always@ (posedge clock)
18          begin
19              if(out==1)
20                  begin
21                      R1=A1;
22                      R2=A2;
23                      R3=A3;
24                      R4=A4;
25                  end
26          end
27
28 endmodule

```

4.5.8.3 Temporizador

Esse módulo foi criado para que o *clock* não tivesse uma frequência muito alta ao utilizar o *clock* automático do FPGA, sendo assim ele é responsável por diminuir a frequência do cristal de 50Mhz da placa em uma frequência observável, pois ao utilizar a frequência muito rápida há risco de pular alguma instrução e impede de ver com melhor clareza os valores de saída nos *displays*. A implementação desse módulo pode ser visto no Algoritmo 12 abaixo

Algoritmo 12 - Temporizador

```

1 module temporizador(clockin , clockout);
2
3     input  clockin;
4         output wire clockout;
5
6
7     reg [30:0] count;
8
9     always@(posedge clockin)
10    begin
11        count <= count + 1;
12    end
13
14    assign clockout = count[22];
15 endmodule

```

4.5.9 Unidade de Controle

A Unidade de Controle é responsável pela configuração de todos os sinais de controle no processador. Existem dois tipos de Unidade de Controle, a *Hardwired* e a Micropogramada, no caso do projeto em questão é utilizada a *Hardwired* em que os sinais de controle são gerados a partir da identificação de qual instrução está sendo executada, sendo assim, através da análise do *opcode* é capaz de configurar as especificações necessárias para que ocorra o fluxo de dados de maneira correta na Unidade de Processamento. Tais sinais podem ser vistos na [Tabela 9](#) abaixo.

Tabela 9 – Sinais de Controle

Sinais de Controle	Módulo	Função
WriteMark	Banco de Registradores	Escrita no Banco
AluSrc	MUX ULA	Escolha entre imediato ou dado do banco
AluOp	Operação da ULA	ULA
AluSelec	MUX PC	Escolha do próximo endereço de PC
MemReg	MUX pós Memória de Dados	Escolha entre dado da memória ou resultado da ULA
RegDest	MUX pré Banco de Registradores	Escolhe registrador de destino
MemWrite	Memória de Dados	Escrita na Memória
beq	ULA,MUXPC e PC	verifica <i>branch</i>
bne	ULA,MUXPC e PC	verifica <i>branch</i>
hlt	PC	Para a contagem do PC
rst	PC	Reseta a contagem do PC
flag	Módulo de Saída	Mostra algum valor no <i>display</i>
flagIN	Módulo de Entrada	Lê algum dado para o Processador
checkin	Módulo de Entrada	Para o processador para fazer a leitura do dado

Fonte: O Autor

O Algoritmo 13 recebe todos os sinais de controle que são utilizados no processador que apresentam valores 0 significando que o sinal está desativado para aquela instrução, e sinal com valor 1 que ativa o sinal para a instrução. Os únicos dois sinais que funcionam de maneira diferente é a Unidade Lógica e Aritmética recebe um *opcode* determinando a operação a ser realizada e o multiplexador que controla o próximo endereço de PC pois este apresenta quatro possibilidades diferentes de seleção. Além disso há um sinal de entrada chamado *checkin* tal sinal é selecionado pelo usuário através de um *switch* do FPGA indicando o fim da seleção de um dado que será lido pelo registrador, enquanto esse *switch* não for selecionado o sinal de controle *hlt* permanece ligado impedindo que o processador vá para a próxima instrução.

Dentre todos os sinais de controle existentes nesse módulo, há dois deles que valem ser citados pois estes monitoram as instruções de *Input* e *Output*.

O primeiro sinal de controle é a *flagIN* que é utilizado apenas na instrução de *Input*, tal sinal de controle é responsável pela seleção de um multiplexador que irá optar por fazer a leitura das chaves(*switches*) realizando assim a entrada de algum valor escolhido pelo usuário no processador ou então selecionará um valor que virá do Banco de Registradores representando um endereço que é utilizado pela instrução de *jump*.

O segundo sinal é o *flag* tal sinal é utilizado apenas na instrução de *Output* para que quando seja ativado possa mostrar o valor de saída desejado em um *display*.

Algoritmo 13 - Unidade de Controle

```

1  module UnidadedeControle(WriteMark ,RegDest ,AluSrce ,AluOp ,MemWrite ,MemReg ,beq ,bne ,AluSelec
2    ,jmp ,hlt ,rst ,opcode ,flag ,flagIN ,checkin);
3
4  input [5:0] opcode;
5  input checkin;
6  output reg hlt,jmp,rst,beq,bne,flag,flagIN;
7  output reg RegDest;
8  output reg WriteMark;
9  output reg AluSrce;
10 output reg[4:0] AluOp;
11 output reg[1:0] AluSelec;
12 output reg MemReg;
13 output reg MemWrite;
14 always @(opcode) begin
15
16   case(opcode[5:0])
17     6'b000000: begin //add
18       WriteMark = 1'b1;
19       AluSrce = 1'b0;
20       AluOp = 5'b00000;
21       AluSelec= 2'b10;
22       MemReg= 1'b0;
23       RegDest= 1'b1;
24       MemWrite= 1'b0;
25       beq= 1'b0;
26       bne= 1'b0;
27       jmp= 1'b0;
28       hlt= 1'b0;
29       rst= 1'b0;
30       flag=1'b0;
31       flagIN=1'b0;
32       end
33     6'b000001: begin //sub
34       WriteMark = 1'b1;
35       AluSrce = 1'b0;
36       AluOp = 5'b000001;
37       AluSelec= 2'b10;
38       MemReg= 1'b0;
39       RegDest= 1'b1;
40       MemWrite= 1'b0;
41       beq= 1'b0;
42       bne= 1'b0;
43       jmp= 1'b0;
44       hlt= 1'b0;
45       rst= 1'b0;
46       flag=1'b0;

```

```
46          flagIN=1'b0;
47      end
48 6'b000010: begin //addi
49      WriteMark = 1'b1;
50      AluSrce = 1'b1;
51      AluOp = 5'b00000;
52      AluSelect= 2'b10;
53      MemReg= 1'b0;
54      RegDest= 1'b0;
55      MemWrite= 1'b0;
56      beq= 1'b0;
57      bne= 1'b0;
58      jmp= 1'b0;
59      hlt= 1'b0;
60      rst= 1'b0;
61      flag=1'b0;
62      flagIN=1'b0;
63  end
64 6'b000011: begin //subi
65      WriteMark = 1'b1;
66      AluSrce = 1'b1;
67      AluOp = 5'b00001;
68      AluSelect= 2'b10;
69      MemReg= 1'b0;
70      RegDest= 1'b0;
71      MemWrite= 1'b0;
72      beq= 1'b0;
73      bne= 1'b0;
74      jmp= 1'b0;
75      hlt= 1'b0;
76      rst= 1'b0;
77      flag=1'b0;
78      flagIN=1'b0;
79  end
80 6'b000100: begin //mult
81      WriteMark = 1'b1;
82      AluSrce = 1'b0;
83      AluOp = 5'b00011;
84      AluSelect= 2'b10;
85      MemReg= 1'b0;
86      RegDest= 1'b1;
87      MemWrite= 1'b0;
88      beq= 1'b0;
89      bne= 1'b0;
90      jmp= 1'b0;
91      hlt= 1'b0;
92      rst= 1'b0;
93      flag=1'b0;
94      flagIN=1'b0;
95  end
96 6'b000101: begin //div
97      WriteMark = 1'b1;
98      AluSrce = 1'b0;
99      AluOp = 5'b00101;
100     AluSelect= 2'b10;
101     MemReg= 1'b0;
102     RegDest= 1'b1;
103     MemWrite= 1'b0;
104     beq= 1'b0;
105     bne= 1'b0;
```

```
106                      jmp= 1'b0;
107                      hlt= 1'b0;
108                      rst= 1'b0;
109                      flag=1'b0;
110                      flagIN=1'b0;
111                      end
112 6'b000110: begin //slt
113          WriteMark = 1'b1;
114          AluSrce = 1'b0;
115          AluOp = 5'b000010;
116          AluSelec= 2'b10;
117          MemReg= 1'b0;
118          RegDest= 1'b1;
119          MemWrite= 1'b0;
120          beq= 1'b0;
121          bne= 1'b0;
122          jmp= 1'b0;
123          hlt= 1'b0;
124          rst= 1'b0;
125          flag=1'b0;
126          flagIN=1'b0;
127          end
128 6'b000111: begin //NOT
129          WriteMark = 1'b1;
130          AluSrce = 1'b0;
131          AluOp = 5'b01000;
132          AluSelec= 2'b10;
133          MemReg= 1'b0;
134          RegDest= 1'b1;
135          MemWrite= 1'b0;
136          beq= 1'b0;
137          bne= 1'b0;
138          jmp= 1'b0;
139          hlt= 1'b0;
140          rst= 1'b0;
141          flagIN=1'b0;
142          end
143 6'b001000: begin //AND
144          WriteMark = 1'b1;
145          AluSrce = 1'b0;
146          AluOp = 5'b01001;
147          AluSelec= 2'b10;
148          MemReg= 1'b0;
149          RegDest= 1'b1;
150          MemWrite= 1'b0;
151          beq= 1'b0;
152          bne= 1'b0;
153          jmp= 1'b0;
154          hlt= 1'b0;
155          rst= 1'b0;
156          flag= 1'b0;
157          flagIN=1'b0;
158          end
159 6'b001001: begin //OR
160          WriteMark = 1'b1;
161          AluSrce = 1'b0;
162          AluOp = 5'b01010;
163          AluSelec= 2'b10;
164          MemReg= 1'b0;
165          RegDest= 1'b1;
```

```
166                         MemWrite= 1'b0;
167                         beq= 1'b0;
168                         bne= 1'b0;
169                         jmp= 1'b0;
170                         hlt= 1'b0;
171                         rst= 1'b0;
172                         flag= 1'b0;
173                         flagIN=1'b0;
174                         end
175 6'b0001010: begin //xor
176                         RegDest = 1'b1;
177                         WriteMark = 1'b1;
178                         AluSrce = 1'b0;
179                         AluOp = 5'b01011;
180                         AluSelect= 2'b10;
181                         MemReg= 1'b0;
182                         RegDest= 1'b1;
183                         MemWrite= 1'b0;
184                         beq= 1'b0;
185                         bne= 1'b0;
186                         jmp= 1'b0;
187                         hlt= 1'b0;
188                         rst= 1'b0;
189                         flag= 1'b0;
190                         flagIN=1'b0;
191                         end
192 6'b0001011: begin //lw
193                         WriteMark = 1'b1;
194                         AluSrce = 1'b1;
195                         AluOp = 5'b00000;
196                         AluSelect= 2'b10;
197                         MemReg= 1'b1;
198                         RegDest= 1'b0;
199                         MemWrite= 1'b0;
200                         beq= 1'b0;
201                         bne= 1'b0;
202                         jmp= 1'b0;
203                         hlt= 1'b0;
204                         rst= 1'b0;
205                         flag=1'b0;
206                         flagIN=1'b0;
207                         end
208 6'b0001100: begin//ldi
209                         WriteMark = 1'b1;
210                         AluSrce = 1'b1;
211                         AluOp = 5'b00000;
212                         AluSelect= 2'b10;
213                         MemReg= 1'b0;
214                         RegDest= 1'b0;
215                         MemWrite= 1'b0;
216                         beq= 1'b0;
217                         bne= 1'b0;
218                         jmp= 1'b0;
219                         hlt= 1'b0;
220                         rst= 1'b0;
221                         flag=1'b0;
222                         flagIN=1'b0;
223                         end
224 6'b0001101: begin //sw
225                         WriteMark = 1'b0;
```

```
226                     AluSrce = 1'b1;
227                     AluOp = 5'b00000;
228                     AluSelec= 2'b10;
229                     MemReg= 1'b0;
230                     RegDest= 1'b0;
231                     MemWrite= 1'b1;
232                     beq= 1'b0;
233                     bne= 1'b0;
234                     jmp= 1'b0;
235                     hlt= 1'b0;
236                     rst= 1'b0;
237                     flag=0;
238                     flagIN=1'b0;
239                     end
240 6'b001110: begin //beq (parei aqui)
241                     WriteMark = 1'b0;
242                     AluSrce = 1'b0;
243                     AluOp = 5'b00100;
244                     AluSelec= 2'b00;
245                     MemReg= 1'b0;
246                     RegDest= 1'b0;
247                     MemWrite= 1'b0;
248                     beq= 1'b1;
249                     bne= 1'b0;
250                     jmp= 1'b0;
251                     hlt= 1'b0;
252                     rst= 1'b0;
253                     flag=1'b0;
254                     flagIN=1'b0;
255                     end
256 6'b001111: begin //bne
257                     WriteMark = 1'b0;
258                     AluSrce = 1'b0;
259                     AluOp = 5'b00100;
260                     AluSelec= 2'b00;
261                     MemReg= 1'b0;
262                     RegDest= 1'b0;
263                     MemWrite= 1'b0;
264                     beq= 1'b0;
265                     bne= 1'b1;
266                     jmp= 1'b0;
267                     hlt= 1'b0;
268                     rst= 1'b0;
269                     flag=1'b0;
270                     flagIN=1'b0;
271                     end
272 6'b010000: begin //jmp
273                     WriteMark = 1'b0;
274                     AluSrce = 1'b0;
275                     AluOp = 5'b00000;
276                     AluSelec= 2'b01;
277                     MemReg= 1'b0;
278                     RegDest= 1'b0;
279                     MemWrite= 1'b0;
280                     beq= 1'b0;
281                     bne= 1'b0;
282                     jmp= 1'b1;
283                     hlt= 1'b0;
284                     rst= 1'b0;
285                     flag=1'b0;
```

```
286          flagIN=1'b0;
287      end
288 6'b010001: begin //jmp
289      WriteMark = 1'b0;
290      AluSrce = 1'b0;
291      AluOp = 5'b00000;
292      AluSelect= 2'b11;
293      MemReg= 1'b0;
294      RegDest= 1'b0;
295      MemWrite= 1'b0;
296      beq= 1'b0;
297      bne= 1'b0;
298      jmp= 1'b1;
299      hlt= 1'b0;
300      rst= 1'b0;
301      flag=1'b0;
302      flagIN=1'b0;
303  end
304 6'b010010: begin //NOP
305      WriteMark = 1'b0;
306      AluSrce = 1'b0;
307      AluOp = 5'b00000;
308      AluSelect= 2'b10;
309      MemReg= 1'b0;
310      RegDest= 1'b0;
311      MemWrite= 1'b0;
312      beq= 1'b0;
313      bne= 1'b0;
314      jmp= 1'b0;
315      hlt= 1'b0;
316      rst= 1'b0;
317      flag= 1'b1;
318      flagIN=1'b0;
319  end
320 6'b010011: begin //HALT
321      WriteMark = 1'b0;
322      AluSrce = 1'b0;
323      AluOp = 5'b00000;
324      AluSelect= 2'b10;
325      MemReg= 1'b0;
326      RegDest= 1'b0;
327      MemWrite= 1'b0;
328      beq= 1'b0;
329      bne= 1'b0;
330      jmp= 1'b0;
331      hlt= 1'b1;
332      rst= 1'b0;
333      flag= 1'b0;
334      flagIN=1'b0;
335  end
336 6'b010100: begin//IN
337  if(checkin==1)
338  begin
339      WriteMark = 1'b1;
340      AluSrce = 1'b1;
341      AluOp = 5'b00000;
342      AluSelect= 2'b10;
343      MemReg= 1'b0;
344      RegDest= 1'b1;
345      MemWrite= 1'b0;
```

```

346                      beq= 1'b0;
347                      bne= 1'b0;
348                      jmp= 1'b0;
349                      hlt= 1'b0;
350                      rst= 1'b0;
351                      flag= 1'b0;
352                      flagIN=1'b1;
353      end
354  else if(checkin==0)
355 begin
356      WriteMark = 1'b1;
357      AluSrce = 1'b1;
358      AluOp = 5'b00000;
359      AluSelec= 2'b10;
360      MemReg= 1'b0;
361      RegDest= 1'b1;
362      MemWrite= 1'b0;
363      beq= 1'b0;
364      bne= 1'b0;
365      jmp= 1'b0;
366      hlt= 1'b1;
367      rst= 1'b0;
368      flag= 1'b0;
369      flagIN=1'b1;
370  end
371      end
372 6'b010101: begin //OUT
373      WriteMark = 1'b0;
374      AluSrce = 1'b0;
375      AluOp = 5'b00000;
376      AluSelec= 2'b10;
377      MemReg= 1'b0;
378      RegDest= 1'b1;
379      MemWrite= 1'b0;
380      beq= 1'b0;
381      bne= 1'b0;
382      jmp= 1'b0;
383      hlt= 1'b0;
384      rst= 1'b0;
385      flag= 1'b1;
386      flagIN=1'b0;
387  end
388 6'b010110: begin //MOVE
389      WriteMark = 1'b1;
390      AluSrce = 1'b0;
391      AluOp = 5'b00000;
392      AluSelec= 2'b10;
393      MemReg= 1'b0;
394      RegDest= 1'b1;
395      MemWrite= 1'b0;
396      beq= 1'b0;
397      bne= 1'b0;
398      jmp= 1'b0;
399      hlt= 1'b0;
400      rst= 1'b0;
401      flag= 1'b0;
402      flagIN=1'b0;
403  end
404 6'b010111: begin //SHL
405      WriteMark = 1'b1;

```

```

406                     AluSrcE = 1'b0;
407                     AluOp = 5'b000110;
408                     AluSelect= 2'b10;
409                     MemReg= 1'b0;
410                     RegDest= 1'b1;
411                     MemWrite= 1'b0;
412                     beq= 1'b0;
413                     bne= 1'b0;
414                     jmp= 1'b0;
415                     hlt= 1'b0;
416                     rst= 1'b0;
417                     flag= 1'b0;
418                     flagIN=1'b0;
419                     end
420             6'b011000: begin //SHR
421                 WriteMark = 1'b1;
422                 AluSrcE = 1'b0;
423                 AluOp = 5'b000111;
424                 AluSelect= 2'b10;
425                 MemReg= 1'b0;
426                 RegDest= 1'b1;
427                 MemWrite= 1'b0;
428                 beq= 1'b0;
429                 bne= 1'b0;
430                 jmp= 1'b0;
431                 hlt= 1'b0;
432                 rst= 1'b0;
433                 flag= 1'b0;
434                 flagIN=1'b0;
435                 end
436
437 endcase
438 end
439 endmodule

```

4.5.10 Integração da Unidade de Processamento

Como o módulo da Unidade de Processamento ainda está em desenvolvimento, houve a interligação de todos os módulos exceto a Unidade de Controle e o módulo de Entrada e Saída que serão desenvolvidos e integrados ao processador. A integração dos módulos da unidade de processamento podem ser vistos conforme o Algoritmo 14 abaixo. Vale citar que o módulo de integração de todos os componentes ainda não é o final e que grande parte dos sinais de entrada e saída são utilizados por questões de teste já que a Unidade de Controle ainda será implementada.

Algoritmo 14 - Integração da Unidade Processamento

```

1 module Processador(clkk,reset,D1,D2,D3,D4,D11,D22,D33,D44,D333,D444,entradaIN,checkin);
2
3 input wire reset;
4 input wire clkk,checkin;
5 wire clk;
6 wire Regdst,WM,AluSrcE,MemReg,beq,jmp,rst,zero,hlt,bne,MW;
7 wire[31:0] Instruction;

```

```

8  wire [1:0] AluSelect;
9  wire [4:0] AluOp;
10 wire [31:0] addressOut;
11 wire [31:0] Extend26_32;
12 wire [31:0] Extend16_32;
13 wire [31:0] AluOut;
14 wire [31:0] data1;
15 wire [31:0] data2;
16 wire [31:0] OutMux;
17 wire [31:0] DataWrite;
18 wire [4:0] RegBankInput;
19 wire [31:0] AluInput ,DataOut ,address ;
20 wire [1:0] AluSelect;
21 output wire[6:0] D1 ,D2 ,D3 ,D4 ,D11 ,D22 ,D33 ,D44 ,D333 ,D444 ;
22 wire flg,ff;
23 input wire [3:0] entradaIN;
24 wire [15:0] saidaMuxIn ,saidaIN ;
25 wire [16:0] entradaIn;
26 wire [31:0] saidadisplay;
27
28 temporizador temp(.clockin(clkk) , .clockout(clk));
29
30 UnidadedeControle UC(.opcode(Instruction [31:26]) ,
31 .WriteMark(WM) ,
32 .AluSrce(AluSrce) ,
33 .AluOp(AluOp) ,
34 .MemWrite(MW) ,
35 .RegDest(Regdst) ,
36 .MemReg(MemReg) ,
37 .beq(beq) ,
38 .bne(bne) ,
39 .AluSelect(AluSelect) ,
40 .jmp(jmp) ,
41 .hlt(hlt) ,
42 .rst(rst)
43 ,.flag(flg)
44 ,.flagIN(ff)
45 ,.checkin(checkin));
46
47 PC ProgramCounter(.entrada(address) ,
48
49
50
51
52
53
54
55
56
57 MemoriadeInstrucoes InstructionMemory(.address(addressOut) ,.clock(clk) ,
58 .instruction(Instruction));
59
60 MUX1 MuxBankReg(.inputReg1(Instruction [20:16]) ,.inputReg2(Instruction [15:11]) ,
61 .saidaMUX1(RegBankInput) ,.Regdest(Regdst));
62
63 bancoreg RegisterBank(.Reg1(Instruction [25:21]) ,.Reg2(Instruction [20:16]) ,
64
65
66
67 .writeRegister(RegBankInput) ,
       .regWrite(OutMux) ,
       .clock(clk) ,
       .data1(data1) ,

```

```
68          .data2(data2),
69          .writeMark(WM));
70
71 Extensordebit26 Extender_26_32(.entrada(Instruction[25:0]),.saida26(Extend26_32));
72
73 MUX5 MuxInput(.inputIN(saidaIN), .inputbanco(Instruction[15:0]),.saidaMUX5(saidaMuxIn),.
    flagIN(ff));
74
75 Extensordebit Extender_16_32(.entrada(saidaMuxIn),.saida16(Extend16_32));
76
77 MUX2 MuxRegAlu(.inputRT(data2),.inputextensor(Extend16_32),.saidaMUX2(AluInput),.AluSrc(
    AluSrce));
78
79 ULA ArithmeticLogicUnity(.op(AluOp),.data1(data1),.data2(AluInput),.result(AluOut),.zero(
    zero),.shamt(Instruction[10:6]));
80
81 MemDados DataMemory(.Writedata(data2),.address(AluOut),.clock(clk),.dataOut(DataOut),.
    MemWrite(MW));
82
83 MUX3 DataMemOut(.aluResult(AluOut),.readMemoria(DataOut),.saidaMUX3(OutMux),.memToReg(
    MemReg));
84
85 MUX4 MasterMux(.zero(zero),.signalExtend(Extend26_32),.brancheq(beq),.branchneq(bne),.
    sumOut(Extend16_32),.regOut(data1),.PC(addressOut),.PCout(address),.control(AluSelec)
);
86
87 Display_BIG SeteSegmentos(.binary(OutMux),.out(flg),.R1(D1),.R2(D2),.R3(D3),.R4(D4),.
    clock(clk));
88
89 IN moduloentrada(.entrada(entradaIN),.saida(saidaIN));
90
91 ExtensordebitIN ExtensorIN(.entrada(saidaIN),.saida16(saidadisplay),.clock(clk),.reset(
    reset));
92
93 Display_IN seteSeg(.binary(saidadisplay),.R1(D11),.R2(D22), .R3(D33), .R4(D44), .clock(
    clk));
94
95 Display_PC seteSeg2(.binary(addressOut),.R1(D111),.R2(D222), .R3(D333), .R4(D444), .clock
    (clk));
96
97 endmodule
```

5 Resultados e Discussões

Uma vez implementado todos os componentes que compõem a Unidade de Processamento foram realizadas simulações para comprovar seu funcionamento, tais simulações foram divididas em duas partes. A primeira parte também foi dividida em outras duas partes, a primeira tem como foco testar cada componente que compõem a Unidade de Processamento de maneira individual através de *waveforms*. A segunda etapa será simulada de maneira similar do que a primeira entretanto tem como objetivo simular a Unidade de Processamento integrada com todos os componentes que a compõe através de diferentes tipos instruções.

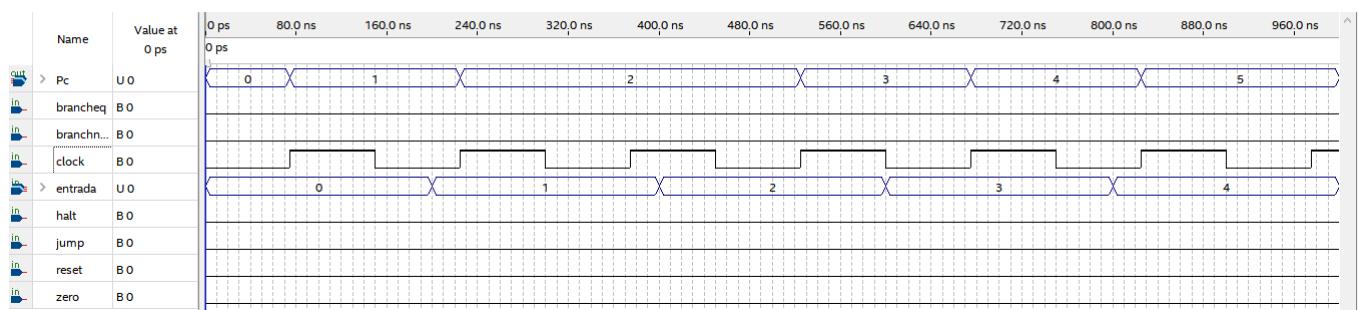
A última etapa também foi dividida em outras duas partes, a primeira visa comprovar o funcionamento da Unidade de Processamento integrada com o Módulo de Entrada, Módulo de Saída e a Unidade de Controle através de *waveforms*. A segunda etapa tem como foco realizar testes na arquitetura juntamente com o FPGA para comprovar o funcionamento do processador através de dois algoritmo testes que levarão em conta as entradas escolhidas pelo usuário e posteriormente a visualização dos resultados através dos *displays* de 7 Segmentos.

5.1 Simulações Individuais

5.1.1 Contador de Programa

A primeira simulação individual feita foi a do contador de programa conforme a [Figura 5](#). Na simulação foi verificado o correto funcionamento do bloco, assim caso a instrução não necessitasse de algum sinal auxiliar,

Figura 5 – *Waveform 1 : PC*

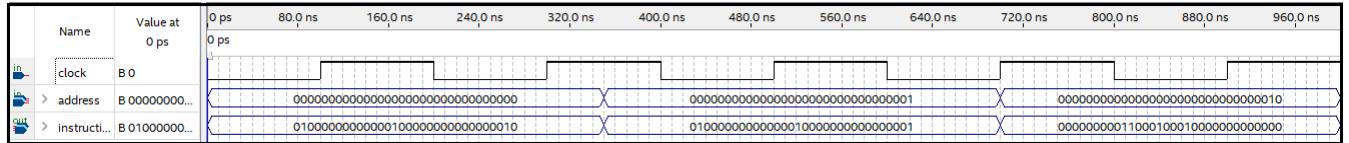


Fonte: O Autor

5.1.2 Memória de Instruções

A simulação da memória de instruções também obteve seu resultado esperado conforme a [Figura 6](#). Através de três instruções já pré definidas conforme mostra Algoritmo 2, cada vez que houvesse uma subida de borda no *clock* a saída era modificada de acordo com uma simulação de entrada do endereço obtido pelo PC.

Figura 6 – *Waveform 2* : Memória de Instruções

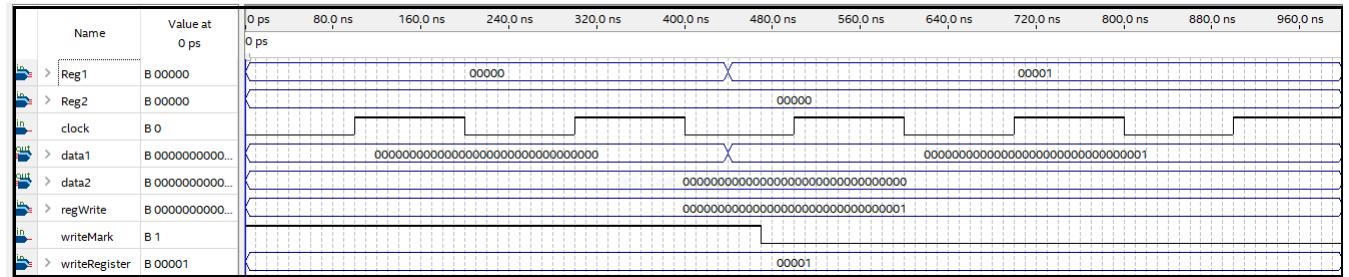


Fonte: O Autor

5.1.3 Banco de Registradores

O Banco de Registradores operou de maneira esperada conforme foi desenvolvido, na simulação como mostra [Figura 7](#) foi feita escrita e leitura de um valor em um registrador de acordo com o sinal de controle que caso esteja com um nível lógico alto realizará a escrita na subida do *clock*, caso o sinal esteja em nível lógico baixo irá realizar a leitura do dado armazenado no registrador.

Figura 7 – ULA

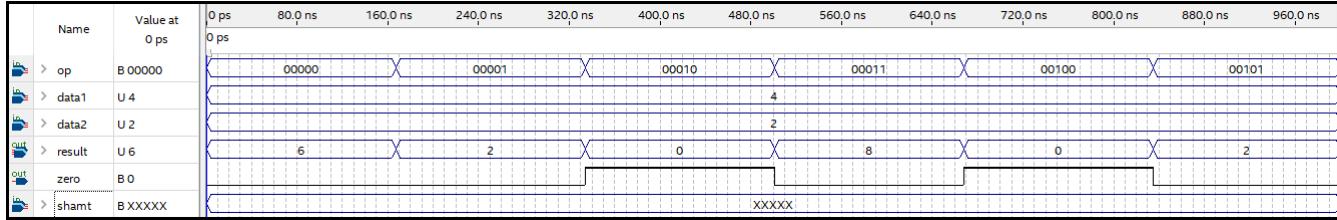


Fonte: O Autor

5.1.4 ULA

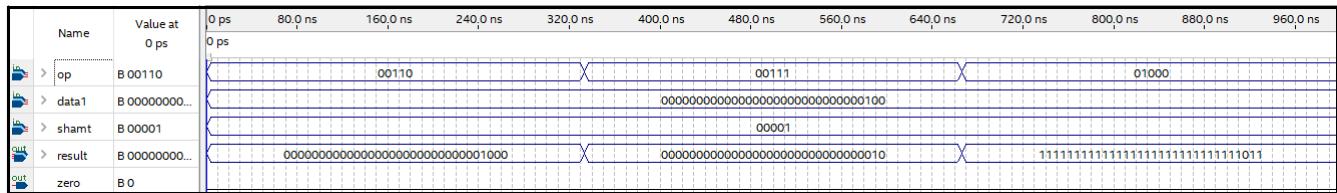
A simulação da ULA foi dividida em 3 partes para uma melhor visualização dos resultados simulando as instruções de acordo com a sequência do *opcode* definido pela [Tabela 8](#), tais resultados foram obtidos com êxito conforme mostra a [Figura 8](#) que representa as instruções adição, subtração, *set on less than*, multiplicação, *branch if equal* e divisão; A [Figura 9](#) representa a representação das instruções de *shift left* que deslocou o dado em dois *bits* para esquerda, *shift right* que deslocou o dado em dois *bits* para a direita, e NOT; Por último a [Figura 10](#) que representa as instruções de AND *bit a bit*, OR *bit a bit* e XOR *bit a bit*.

Figura 8 – *Waveform 3*: ULA - Instruções de 1 a 6



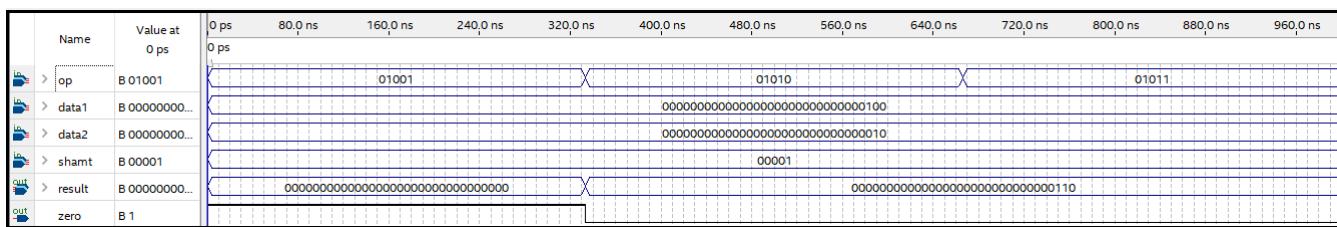
Fonte: O Autor

Figura 9 – *Waveform 4*: ULA - Instruções de 7 a 9



Fonte: O Autor

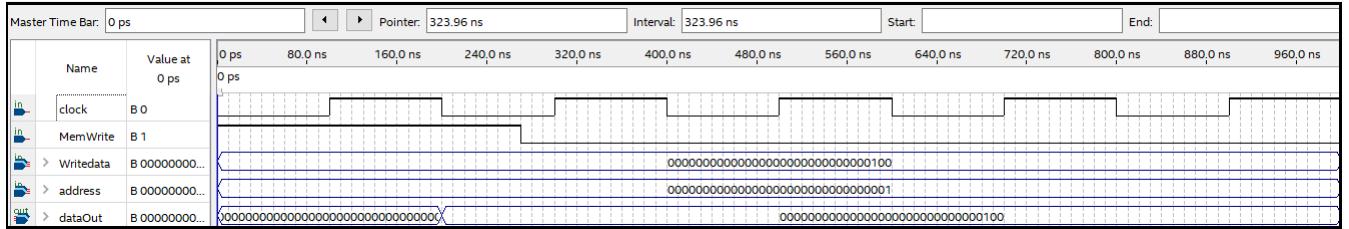
Figura 10 – *Waveform 5*: ULA - Instruções de 10 a 12



Fonte: O Autor

5.1.5 Memória de Dados

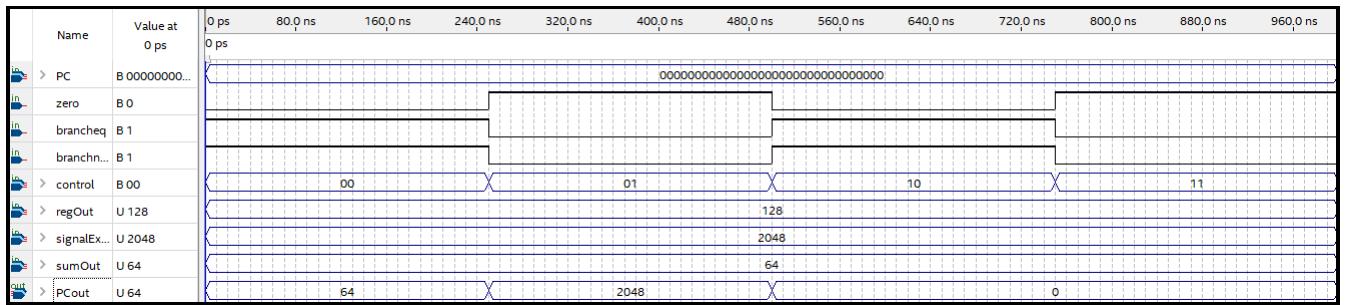
A Memória de Dados assim como os outros componentes apresentados também apresentou um resultado conforme era esperado, assim realiza a escrita de um dado em um endereço de entrada caso o sinal de controle para escrita esteja em nível lógico alto. Caso contrário é feito a leitura do dado armazenado, o funcionamento é parecido com o do Banco de Registradores e pode ser visto na [Figura 11](#).

Figura 11 – *Waveform 6: Memória de Dados*

Fonte: O Autor

5.1.6 Multiplexador

O multiplexador que controla o endereço de PC apresentou um resultado conforme era esperado através do desenvolvimento do componente. O sinal de saída foi selecionado através das mudanças nos seus sinais de controles, indicando qual será o próximo endereço de PC sendo ele apenas incrementado para a próxima posição ou se haverá algum desvio condicional ou não. A representação dos resultados obtidos pode ser visto na [Figura 12](#).

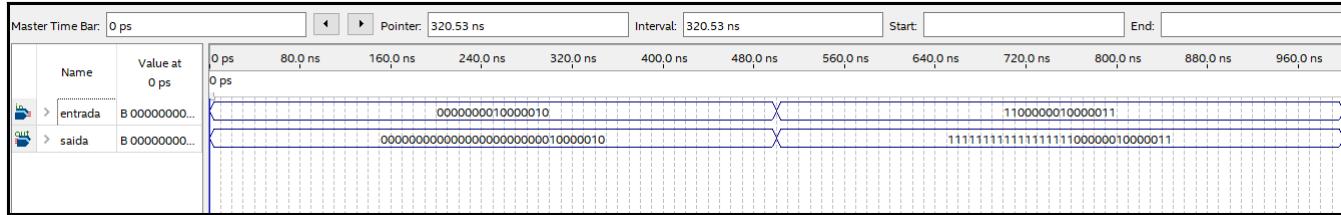
Figura 12 – *Waveform 7: Multiplexador*

Fonte: O Autor

5.1.7 Extensor de Sinal

O extensor de sinal por ser um componente bem simples apresentou o resultado conforme foi projetado. De acordo com a [Figura 13](#), a partir do momento que o dado entra com seus 16bits ele é estendido para 32bits, também foi simulado caso o dado tenha um valor negativo em que o dado é preenchido com mais 16 valores de nível lógico alto ao invés de nível lógico baixo.

Figura 13 – Waveform 6: Extensor de Sinal



Fonte: O Autor

5.1.8 Teste da Unidade de Processamento

Para o teste preliminar do processador com seus módulos conforme o Algoritmo 13, foram selecionadas algumas instruções tanto do tipo R, I e J para comprovar o funcionamento do mesmo em diferentes situações as instruções escolhidas podem ser vistas na Tabela 10 abaixo.

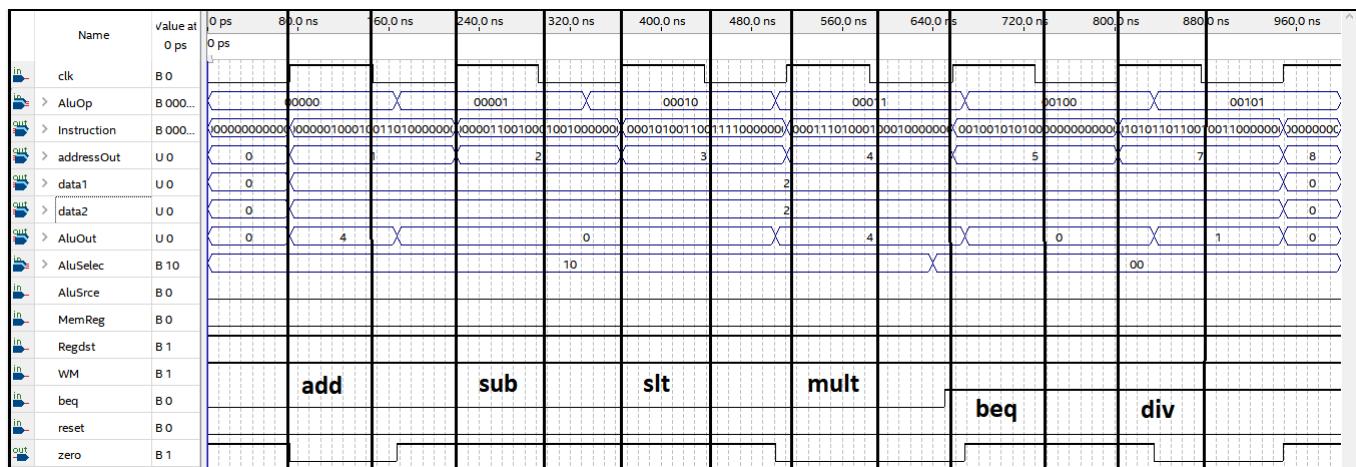
Tabela 10 – Instruções de Teste

- | |
|----------------------|
| (1) - add R1,R2,R3; |
| (2) - sub R1,R2,R3; |
| (3) - slt R1,R2,R3; |
| (4) - mult R1,R2,R3; |
| (5) - beq R1,R2,7; |
| (7) - div R1,R2,R3; |

Fonte: O Autor

As instruções ocuparam as posições de 1 a 7, na memória de instrução sendo a instrução 6 não utilizada. A devida simulação de cada uma das instruções citadas acima com os seus sinais de controle podem ser vistas na Figura 14 abaixo.

Figura 14 – Waveform 7: Simulação de Instruções no Processador - Parte 1



Fonte: O Autor

Conforme simulado na figura, a partir do primeiro pulso de *clock* se dá início a instrução de adição, para que essa instrução tenha seu funcionamento correto o *opcode* da ULA precisou ter um valor de 00000 representando a instrução de soma, o sinal de Regdst proveniente do multiplexador que está na entrada do banco de registradores está em 1 significando que irá salvar o resultado em algum registrador, tal registrador será salvo através setando o sinal de controle WriteMark do banco, e selecionando 0 no último multiplexador do caminho de dados que seleciona o resultado da operação, tendo feito isso ainda faltava mais dois sinais de controle aos multiplexadores que saem do banco de registradores e o que irá selecionar o próximo valor de PC. O sinal de controle AluSrc teve seu valor em 0 se referindo ao Dado 2 que sai do banco de registradores e o sinal de controle AluSel teve seu valor alterado em 10 que significa que não haverá desvio algum.

Com todos os sinais de controle prontos, a primeira instrução conforme é visto no campo addressOut (saída do endereço calculado pelo PC) é realizada de maneira correta de modo que o resultado pode ser observado através do campo AluOut.

No segundo ciclo de *clock* é realizada a operação de subtração, conforme mostra no campo addressOut será realizada a intrução 2, ou seja, a subtração propriamente dita que por ser uma instrução do tipo R os sinais de controle já mencionados permanecem os mesmos, exceto o *opcode* da ULA que seleciona a operação de subtração representada pelo valor 00001, assim como é visto no campo de AluOut.

No terceiro ciclo é realiza a operação de *set on less than*, conforme pode ser visto no campo addressOut será realiza a instrução 3. No caso a instrução slt também é uma instrução do tipo R, então a única modificação feita foi no *opcode* da ULA representando o valor 00010, como é visto no resultado de AluOut como os dois valores não são iguais a saída foi igual a 0.

No quarto ciclo temos uma instrução de *branch if equal*, como ainda não há uma unidade controle para controlar os sinais, o sinal beq foi setado porque o usuário já sabia que aconteceria um branch além de disso o sinal do multiplexador que ajuda no calculo do novo endereço de PC foi mudado para 00 tal condição verifica o *branch*. A operação é feita então como o resultado da operação visto em AluOut foi 0, ou seja, houve *branch*, o sinal zero é trocado para 1, ao chegar no multiplexador ele verifica que zero e beq estão setadas então a saída que entrará no PC é o endereço de *branch* que foi estendido em 32 bits. A mesma verificação do multiplexador é feita no PC, assim o próximo endereço será o endereço 7 aonde está a instrução de divisão.

No último ciclo de *clock* está mais uma instrução do tipo R, no caso a instrução de divisão, que segue o mesmo padrão das outras instruções já mencionadas, aonde o resultado da operação pode ser visto no campo AluOut.

Assim com algumas instruções de testes de funcionalidade processador sem Unidade

de Controle e módulo de *Input* e *Output*, foi possível validar através da manipulação dos sinais de controle feitos pelo usuário que a Unidade de Processamento funciona como esperado.

5.2 Simulação Total

Nesta última seção serão mostrados os resultados finais com os testes feitos no processador primeiramente através de uma *waveform* que visa mostrar o funcionamento de todas as instruções presentes no Conjunto de Instruções conforme a [Tabela 7](#) juntamente com o Módulo de Entrada e a Unidade Controle.

Logo em seguida será apresentada a simulação do processador juntamente com o Módulo de Saída (*displays*) e sua integração com o FPGA. Para isso serão mostrados dois algoritmos. O primeiro algoritmo a ser apresentado têm como objetivo calcular o n-ésimo termo da sequência de *Fibonacci*, sendo que tal termo é inserido pelo usuário; Já o segundo algoritmo verifica se dois triângulos apresentam áreas de valores de iguais, sendo que os valores de base e altura de cada um deles é inserido pelo usuário através dos *switches* do FPGA.

5.2.1 Simulação Total da Unidade de Processamento, Módulo de Entrada e Saída e Unidade de Controle

Para o teste final do processador com seus módulos e a integração com Unidade de Controle e Módulo de Entrada e Saída, foi criada uma sequência de instruções na Memória de Instruções com objetivo de comprovar o funcionamento de todas as instruções sejam elas do tipo R, I ou J com o intuito de comprovar o funcionamento da arquitetura em diferentes situações. Tal sequência pode ser vista na [Tabela 11](#) abaixo.

Tabela 11 – Instruções de Teste com Unidade de Controle

(0) - input r1
(1) - input r2
(2) - ldi 15, r4
(3) - add r3, r1, r2
(4) - sub r3, r1, r2
(5) - mult r3, r1, r2
(6) - div r3, r1, r2
(7) - r1, shr[r1]
(8) - r2, shl[r2]
(9) - move r3, r1
(10) - move r1, r2
(11) - subi r3, r3, 1
(12) - addi r2, r2, 1
(13) - jmp r4
(15) - nop
(16) - move r3, r4
(17) - not r3
(18) - not r3
(19) - and r5, r3, r4
(20) - or r6, r3, r4
(20) - or r6, r3, r4
(21) - xor r7, r3, r2
(22) - beq r5, r3, 28
(23) - jmp , 28
(25) - slt r10, r5, r3
(26) - bne r5, r10, 28
(28) - SW r5, 2
(29) - LW 2, r8
(30) - out r10
(31) - halt

Fonte: O Autor

As instruções ocuparam as posições de 0 a 31, na Memória de Instruções. Abaixo no Algoritmo 15 pode ser visto como as instruções estão dispostas na memória de instruções levando em conta o tipo e formato de cada uma delas.

Algoritmo 15 - Instruções de Teste do Processador

```

1 instrmem[0]= 32'b010100_00000_00001_000000000000; // input r[1]
2 instrmem[1]= 32'b010100_00000_00000_00010_000000000000; // input r[2]
3 instrmem[2]= 32'b001100_00000_00100_00000_00000001111; // r[4]=15 //ldi
4 instrmem[3]= 32'b000000_00001_00010_00011_000000000000; // r[3]= r[1]+r[2] //add
5 instrmem[4]= 32'b000001_00001_00010_00011_000000000000; // r[3]= r[1]-r[2] //sub
6 instrmem[5]= 32'b000100_00001_00010_00011_000000000000; // r[3]= r[1]*r[2] //mult
7 instrmem[6]= 32'b000101_00001_00010_00011_000000000000; // r[3]= r[1]/r[2] //div
8 instrmem[7]= 32'b011000_00001_00000_00001_00000; // r[1]=SHFR[r1]
9 instrmem[8]= 32'b010111_00010_00000_00001_00000; //r[2]=SHFL[r2]
10 instrmem[9]= 32'b010110_00001_00000_00011_000000000000; //r3=r1 //move
11 instrmem[10]= 32'b010110_00010_00000_00001_000000000000; //r1=r2 //move
12 instrmem[11]= 32'b000011_00011_00000_00000_000000000001; // r[3]=r[3]-1 //subi
13 instrmem[12]= 32'b000010_00010_00000_00000_000000000001; // r[2]=r[2]+1 //addi
14 instrmem[13]= 32'b010001_00100_00000_00000_000000000000; // jmp [15]
15 instrmem[15]= 32'b010010_00000_00000_00000_000000000000; //nop

```

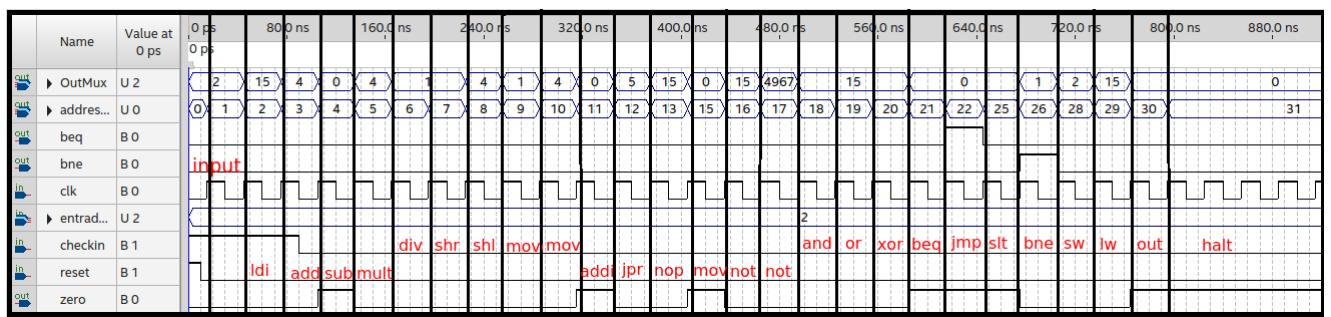
```

16 instrmem[16]= 32'b010110_00100_00000_00011_000000000000; //r3=r4 //move
17 instrmem[17]= 32'b000111_00011_00000_00011_000000000000; //NOT r3
18 instrmem[18]= 32'b000111_00011_00000_00011_000000000000; //NOT r3
19 instrmem[19]= 32'b001000_00011_00100_00101_000000000000; // r[5]= r3 AND r4
20 instrmem[20]= 32'b001001_00011_00100_00110_000000000000; // r[6]= r3 OR r4
21 instrmem[21]= 32'b001010_00011_00100_00111_000000000000; // r[7]= r3 XOR r2
22 instrmem[22]= 32'b001110_00101_00011_00000_00000011001; // beq 25,[r5],r[3] //beq
23 instrmem[23]= 32'b010000_00000_00000_00000_00000011100; //jump to 28 //jmp
24 instrmem[25]= 32'b000110_00101_00011_01010_000000000000; // slt r[10],r[5],r[3] //slt
25 instrmem[26]= 32'b001111_00101_01010_00000_00000011100; // bne 28,[r5],r[10] //beq
26 instrmem[28]= 32'b001101_00000_00101_00000_00000000010; // SW r[5]
27 instrmem[29]= 32'b001011_00000_01000_00000_00000000010; // LW r[8]
28 instrmem[30]= 32'b010101_01010_00000_00000_000000000000; // out r[10] //output
29 instrmem[31]= 32'b010011_00000_00000_00000_000000000000; //halt

```

A devida simulação de cada uma das instruções citadas acima com os seus sinais de controle podem ser vistas na [Figura 15](#) abaixo.

Figura 15 – *Waveform 8*: Simulação de Instruções no Processador com Módulo de Entrada e Unidade de Controle



Fonte: O Autor

Conforme a figura, o valor de entrada e *checkin* indicam que está sendo feita uma escrita de imediato através do Módulo de Entrada. Os sinais de *branch* (beq e bne) serão setados automaticamente através da Unidade de Controle que analisará o *opcode* da instrução, esses sinais irão funcionar junto com o sinal de *zero* que verifica se irá ou não realizar o desvio, caso beq ou bne tiver seu valor lógico alterado para 1 isto significa que o *Program Counter* irá receber o endereço da instrução de desvio, tal desvio pode ser observado através do sinal de *addressOut* que indica qual instrução está sendo realizada no momento.

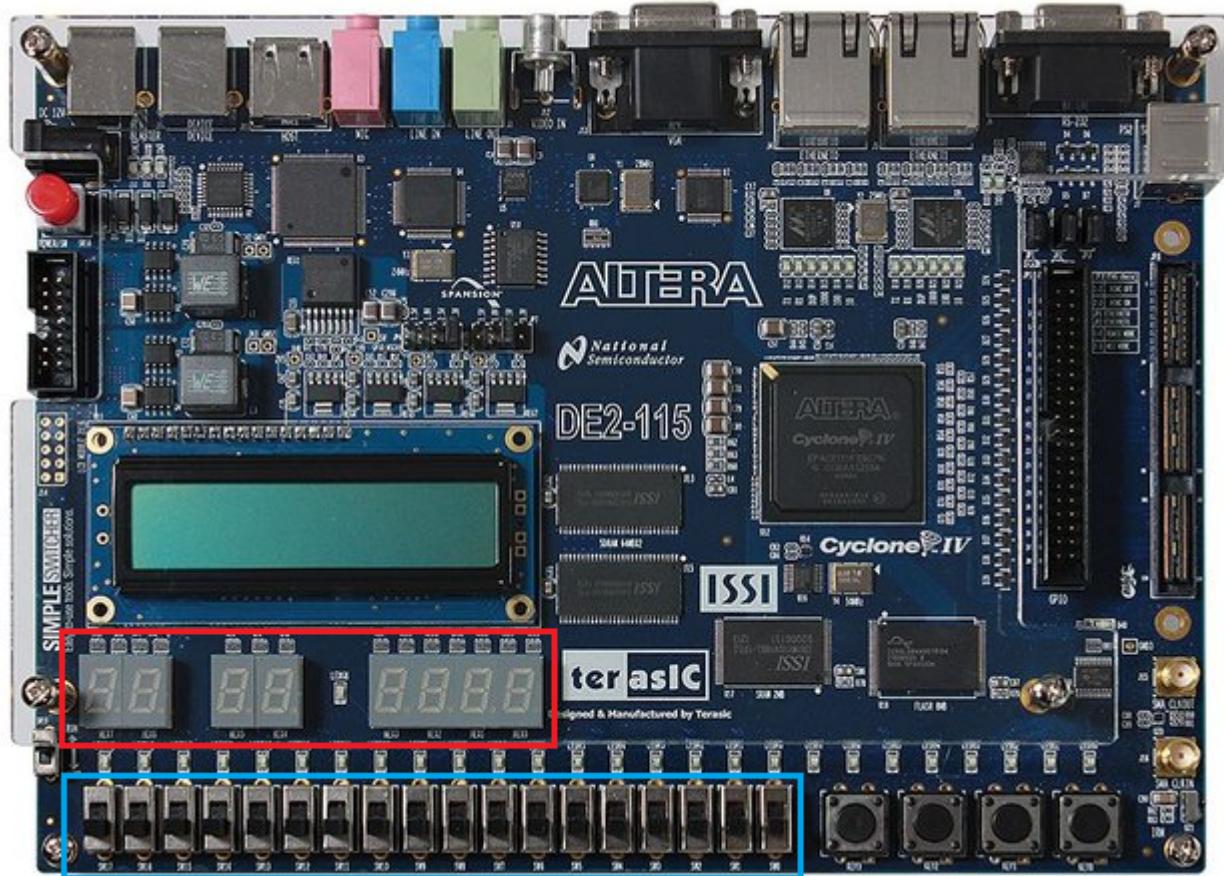
Por fim o sinal *OutMux* representa o valor de saída em decimal do resultado da operação, posteriormente esse valor será convertido em um código BCD e o Módulo de Saída ficará responsável por mostrar esse sinal no *hardware FPGA* para o usuário. Vale citar que os demais sinais de controle não fizeram parte desse teste pois o intuito era testar que a Unidade de Controle fosse capaz de controlar o fluxo de cada uma das instruções no processador respeitando a [Tabela 9](#) e o Algoritmo 13.

Concluído esse teste assim foi possível analisar que a Unidade de Controle exerce um papel muito importante no projeto pois ela é capaz de identificar e setar todos os sinais necessário para o fluxo das instruções, e com isso o teste de todas as instruções foi otimizado e simulado sem muitos problemas.

5.2.2 Simulação Total da Unidade de Processamento, Módulo de Entrada e Saída, Unidade de Controle e FPGA

Antes de dar início a descrição do algoritmo utilizado e posteriormente sua execução e resultados com as entradas de teste, é importante mostrar como é o FPGA e explicar quais componentes desse *hardware* serão utilizados. O FPGA pode ser visto na Figura 16 abaixo.

Figura 16 – O FPGA



Fonte: O Autor

Na cor vermelha se encontram os *displays* dos quais serão utilizados da esquerda para a direita nos dois primeiros *displays* HEX6 e HEX7 está representado o endereço da instrução oriundo do *Program Counter* que está sendo executada pela Memória de Instruções; Em seguida nos *displays* HEX5 e HEX4 está representado o dado em decimal

do valor de entrada escolhido pelo usuário que está sendo lido pelas chaves; E por fim nos quatro últimos *displays* HEX3, HEX2, HEX1 e HEX0 será mostrado algum resultado, no caso do primeiro algoritmo estará representado em decimal o valor do n-ésimo termo da sequência de *Fibonacci*.

Na cor azul estão as chaves da placa (*switches*) que foram utilizados para determinar o valor de entrada em binário do imediato que será inserido pela função *INPUT*, sendo que quando a chave estiver para cima significa que o valor inserido é o valor 1 em binário e quando estiver para baixo o valor é 0 em binário. No projeto em questão para compor o imediato foram utilizadas as chaves de 0 a 15 sendo que do maior número para o menor número estão representados os *bits* do mais significativo para o menos significativo. A chave 16 foi utilizada como *reset* significando que a contagem do *Program Counter* deve ser reiniciado. Por fim a chave 17 indica o sinal de *checkin* que o usuário seleciona indicando ao processador que terminou de inserir o valor de entrada do imediato.

5.2.3 Algoritmo de *Fibonacci*

A conhecida sequência de *Fibonacci* é muito conhecida e se estende de maneira infinita sendo que dentro dela o n-ésimo termo da sequência é determinado através da soma dos seus dois termos anteriores, sendo que os primeiros termos são respectivamente 0 e 1, assim a sequência gera os valor (0, 1, 1, 2, 3, 5, 8, 13, 21,...) e assim sucessivamente.

Partindo do pressuposto da sua definição foi criada uma sequência utilizando grande parte das instruções presentes no processador, a sequência das instruções pode ser vista na [Tabela 12](#) abaixo.

Tabela 12 – Instruções de Teste

instrmem[0] - nop
instrmem[1] - ldi 1, r[1]
instrmem[2] - ldi 1, r[2]
instrmem[3] - ldi 0, r[4]
instrmem[4] - input , r[3]
instrmem[5] - subi r[3],r[3],1
instrmem[6] - slt r[10],r[0],r[3]
instrmem[7] - beq 12,[r10],r[0]
instrmem[8] - add r[2],r[1],r[4]
instrmem[9] - move r[4],r[1]
instrmem[10] - move r[1],r[2]
instrmem[11] - jump, 5
instrmem[12] - ldi R5, 15
instrmem[13] - jumpr, 5
instrmem[15] - sw, 2
instrmem[16] - lw, 2
instrmem[17] - out, r[2]
instrmem[18] - halt

Fonte: O Autor

Abaixo no Algoritmo 16 pode ser visto como as instruções estão dispostas na memória de instruções levando em conta o seu tipo e formato.

Algoritmo 16 - Instruções de *Fibonacci*

```

1      instrmem[0]= 32'b010010_00000_00000_00000_000000000000; //nop
2      instrmem[1]= 32'b001100_00000_00001_00000_00000000001; // r[1]=1
3      instrmem[2]= 32'b001100_00000_00010_00000_00000000001; // r[2]=1
4      instrmem[3]= 32'b001100_00000_00100_00000_00000000000; // r[4]=0
5      instrmem[4]= 32'b010100_00000_00000_00011_00000000000; // input
6      instrmem[5]= 32'b000011_00011_00011_00000_00000000001; // r[3]=r[3]-1
7      instrmem[6]= 32'b000110_00000_00011_01010_00000000000; // slt r[10],r[0],
8          r[3]
9      instrmem[7]= 32'b001110_01010_00000_00000_00000001100; // beq 12,[r10],r
10         [0]
11     instrmem[8]= 32'b000000_00001_00100_00010_00000000000; // r[2]= r[1]+r[4]
12     instrmem[9]= 32'b010110_00001_00000_00100_00000000000; //r4=r1
13     instrmem[10]= 32'b010110_00010_00000_00001_00000000000; //r1=r2
14     instrmem[11]= 32'b010000_00000_00000_00000_00000000101; //jump to 5
15     instrmem[12]= 32'b001100_00000_00101_00000_00000001111; // r[5]=15 ldi
16     instrmem[13]= 32'b010001_00101_00000_00000_00000000000; // jmpr
17     instrmem[15]= 32'b001101_00000_00010_00000_00000000010; // SW
18     instrmem[16]= 32'b001011_00000_00110_00000_00000000010; //lw
19     instrmem[17]= 32'b010101_00110_00000_00000_00000000000; // out r[2]
20     instrmem[18]= 32'b010011_00000_00000_00000_00000000000; //halt

```

Esse algoritmo funciona de tal forma que primeiramente temos uma instrução *Not an Operation* que não faz nada para evitar que tenha algum lixo ou valor armazenado anteriormente influencie no processo das demais instruções, após isso é carregado um valor 1 nos registradores r[2] e r[3] e posteriormente no registrador r[4] é armazenado o valor 0. Em seguida é lido um valor imediato através da instrução de *Input* onde o usuário seleciona o valor em binário que deseja saber o termo de *Fibonacci* correspondente para ele e após o mesmo selecionar a chave de *checkin* significando que o processador pode fazer leitura do dado, o dado é armazenado no registrador r[3] após isso é feita a instrução de subtração com imediato em que é subtraído 1 do registrador r[3] significando que já há um valor da sequência lido que é o valor já armazenado do registrador r[2].

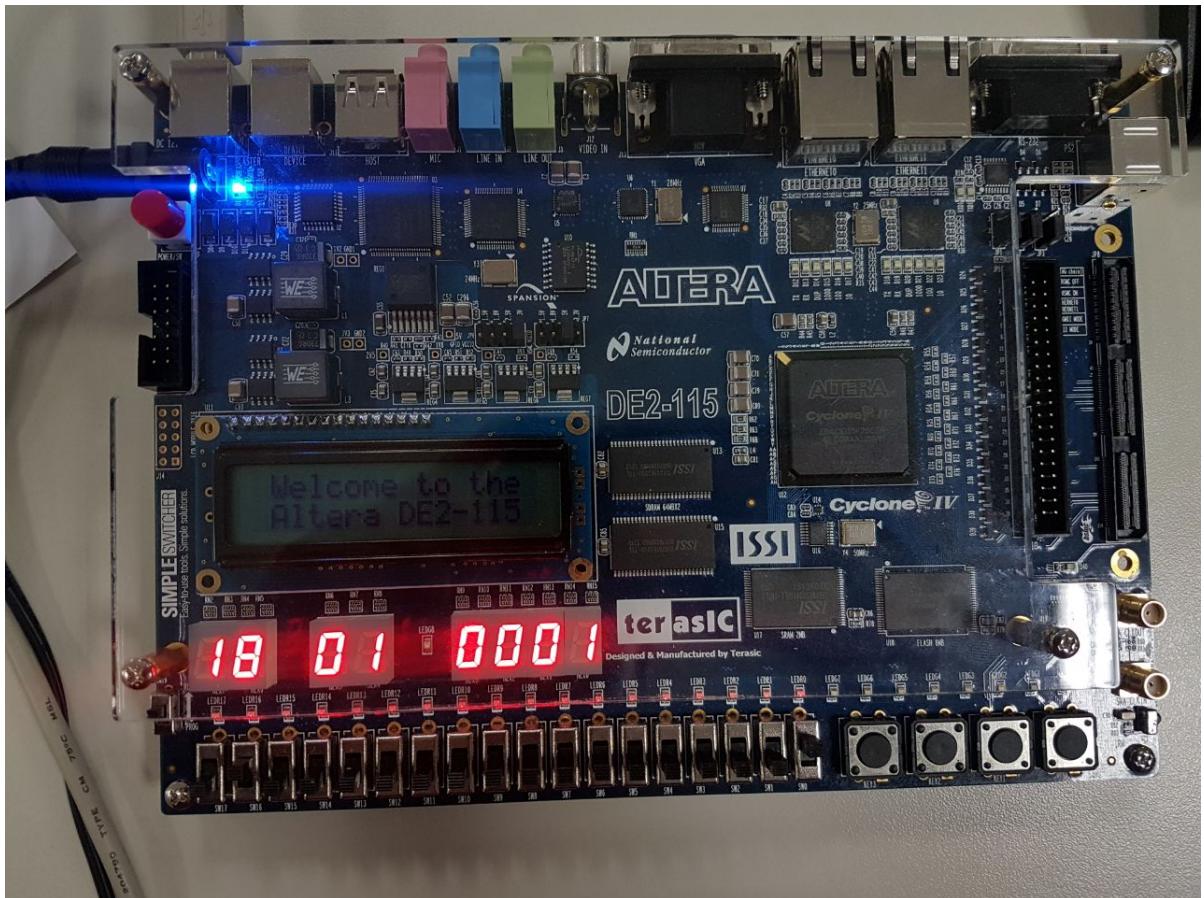
Após isso é feita uma comparação para verificar se o algoritmo já analisou todas as posições da sequência para aquele valor, assim sendo quando esse valor chegar em 0 significa que o dado está pronto para ser mostrado, tal comparação é feita pela instrução *set on less than* e salvo no r[10], caso o r[10] for igual a 0 o *branch* é tomado na próxima instrução a partir daí se tem duas possibilidades: a primeira é se não for tomar o branch é somado os valores armazenados nos registradores r[4] e r[1] esignificando os dois valores anteriores da sequência atual e atualizamos os valores dos mesmos para analisar o próximo valor, (basicamente o que estamos fazendo aqui é um contador que verifica qual é o valor da n-ésima posição de fibonacci desde 1 até o valor que se deseja saber o resultado) então volta para a verificação do *set on less than* através de uma instrução de *jump*; A segunda possibilidade é caso o *branch* for tomado, a partir daí há uma instrução *jump to register*

onde o campo de endereço faz a referência a um registrador no Banco de Registradores e nesse registrador contem o valor 15 que representa o endereço de *jump*, tal valor foi armazenado anteriormente no registrador r[5] através da instrução *load immediate*. Após isso o valor que está armazenado no registrador r[2] é escrito e logo em seguida lido da Memória de Dados através das instruções *Store Word* e *Load Word*, por fim a instrução *Out* permite que o dado seja mostrado nos *displays* e há também uma instrução de *Halt* que paralisa a sequência do *Program Counter* impedindo que o valor que está no *display* mude ou se perca.

Para verificar a funcionalidade do algoritmo de *Fibonacci* foram feitos alguns testes que serão mostrados e comentados a seguir.

O primeiro teste conforme a [Figura 17](#) abaixo, representa a simulação do valor escolhido pelo usuário selecionando as chaves do FPGA, no caso o valor escolhido foi o valor 1.

Figura 17 – Teste 1.1 - Entrada 1



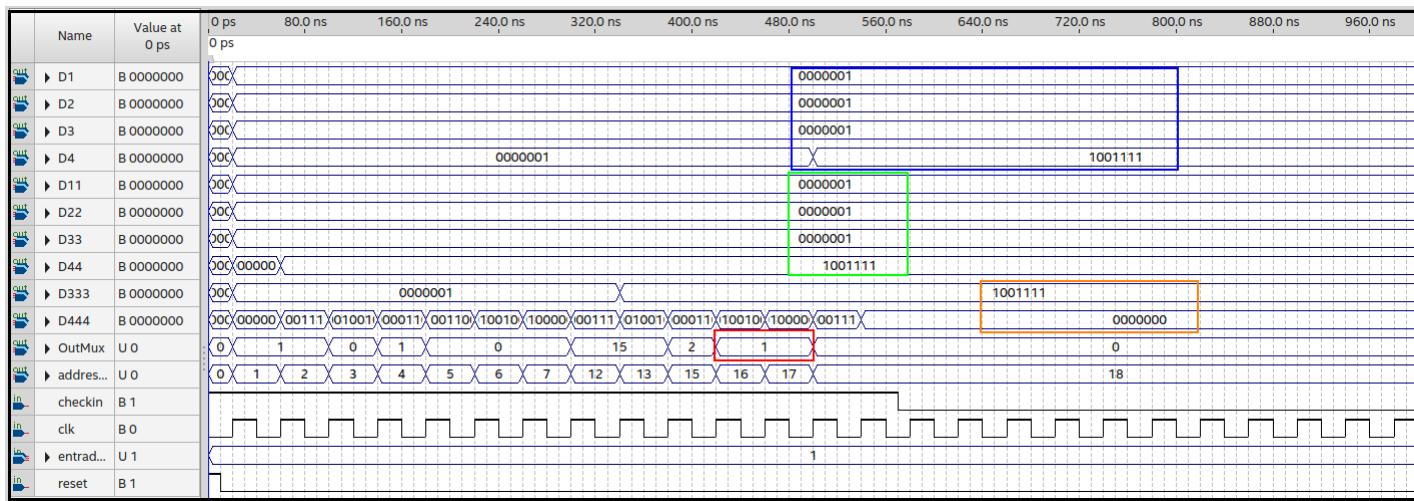
Fonte: O Autor

Da imagem acima se pode analisar que os dois primeiros *displays* representam a última instrução realizada pelo processador, no caso a instrução 18 que é a instrução de

halt usada para manter o dado no *display*. Os dois próximos *displays* mostram o valor lido pelas chaves, no caso o usuário selecionou apenas o *switch* 0 do FPGA representando o número '0000000000000001' em binário, ou melhor dizendo, o valor 1 em decimal. Os quatro últimos representam o valor de *fibonacci* para o número lido que foi convertido para o código BCD e foi utilizado na instrução *out* para visualização do resultado, no caso o resultado foi o valor decimal 1.

Além do teste no *hardware* FPGA foi feito um teste simulando a mesma entrada do *software Quartus* simulando o algoritmo do cálculo do n-ésimo termo da sequência de *Fibonacci*, sendo que o valor escolhido para o cálculo foi o valor 1, conforme mostra a Figura 18 a seguir.

Figura 18 – Teste 1.2- Entrada 1



Fonte: O Autor

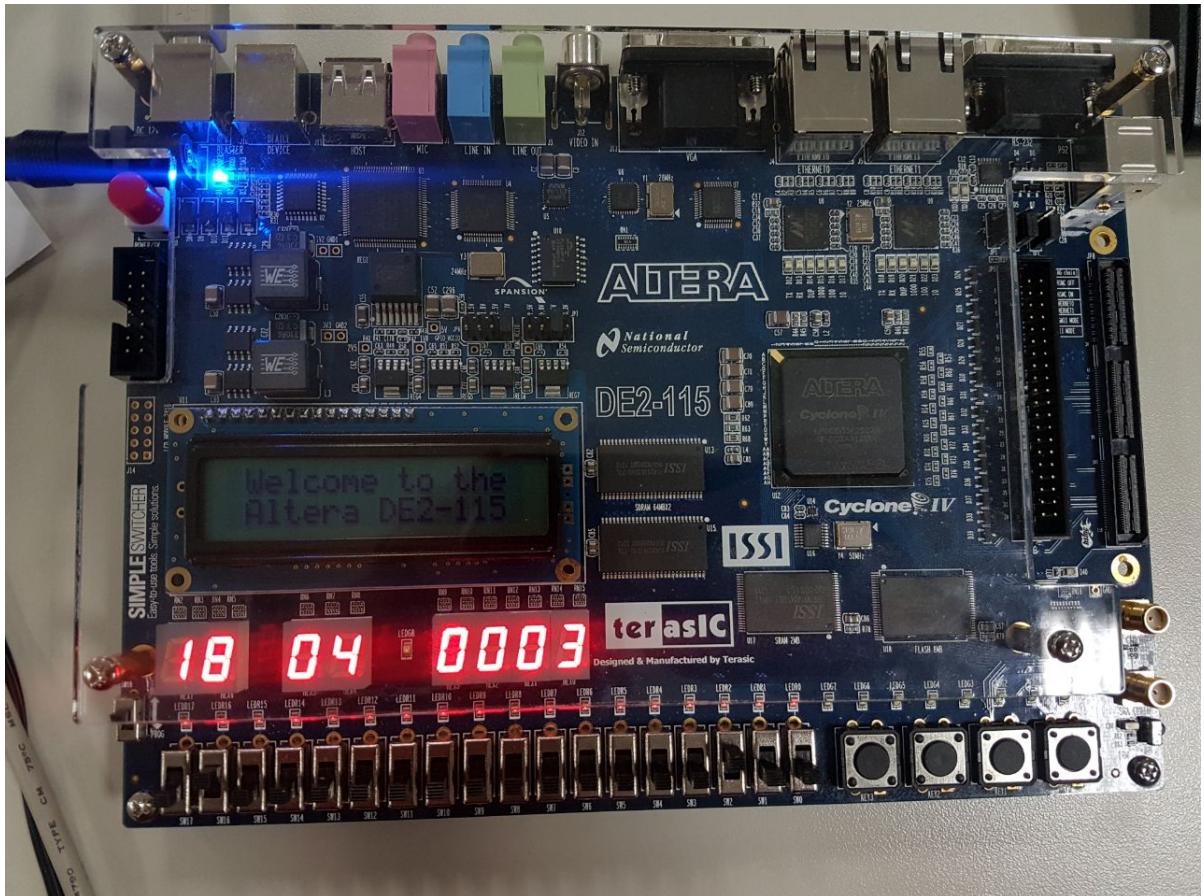
A imagem acima representa o processamento do termo de *Fibonacci*, sendo que as variáveis D1, D2, D3 e D4 indicam o valor do Módulo de Saída representando em código BCD o n-ésimo termo da sequência, da Figura 17 sabemos o que o resultado é 1, sendo assim nos quatro *displays* que representam esse valor se deve mostrar 0 nos *displays* D1, D2 e D3 e valor 1 no *display* D4. O quadrado em azul indicado na Figura 18 mostra que os três primeiros *displays* estão recebendo '0000001' representando o valor 0 código BCD, e o *display* D4 está mostrando '1001111' indicando o valor 1 em código BCD. Tal resultado que está entrando no Módulo de Saída está representado pelo quadrado vermelho.

Os *displays* D11, D22, D33 e D44 representam o valor lido pelo Módulo de Entrada através da escolha de entrada do usuário após o mesmo ter setado o sinal de *checkin* em 1. Dos quatro displays os únicos que são interessantes para serem citados são D33 e D44, onde no *display* D33 está o valor BCD '0000001' representando o valor 0 em decimal e no *display* D44 está o valor 1.

Por fim resta falar sobre os *displays* D333 e D444 circulados pelo quadrado laranja representam o valor do *Program Counter*, onde no *display* D33 está o valor BCD '1001111' representando o valor 1 em decimal e no *display* D44 está o valor 8, representado por '0000000' em código BCD, tal valor pode ser conferido através do campo address onde sua última posição se mantém no valor 18, ou seja, representa a última instrução que foi executada.

De maneira similar ao caso anterior, a [Figura 19](#) representa a simulação do valor escolhido pelo usuário selecionando as chaves do FPGA, no caso o valor escolhido foi o valor 4.

Figura 19 – Teste 2- Entrada 4



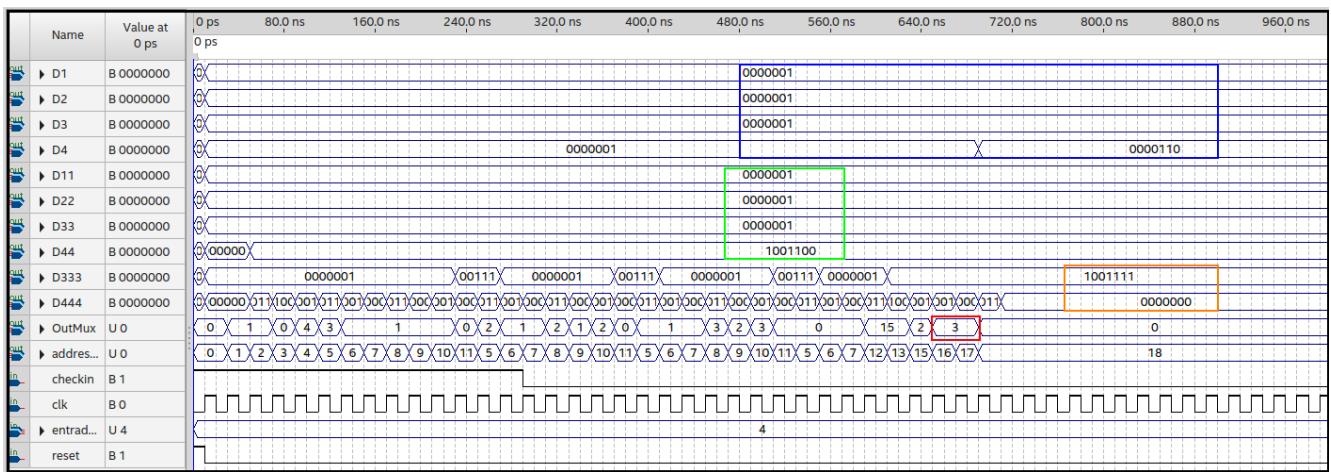
Fonte: O Autor

Da imagem acima se pode analisar que os dois primeiros *displays* representam a última instrução realizada pelo processador, no caso a instrução 18 que é a instrução de *halt* usada para manter o dado no *display*. Os dois próximos *displays* mostram o valor lido pelas chaves, no caso o usuário selecionou apenas *switch* 2 do FPGA representando o número '0000000000000100' em binário, ou melhor dizendo, o valor 4 em decimal. Os quatro últimos representam o valor do n-ésimo termo de *fibonacci* para o número lido, tal

valor foi convertido para o código BCD e foi utilizado na instrução *out* para visualização do resultado, no caso o resultado foi o valor decimal 3.

Novamente foi realizado o mesmo teste no *software Quartus* como mostra a Figura 20 abaixo.

Figura 20 – Teste 2.1- Entrada 4



Fonte: O Autor

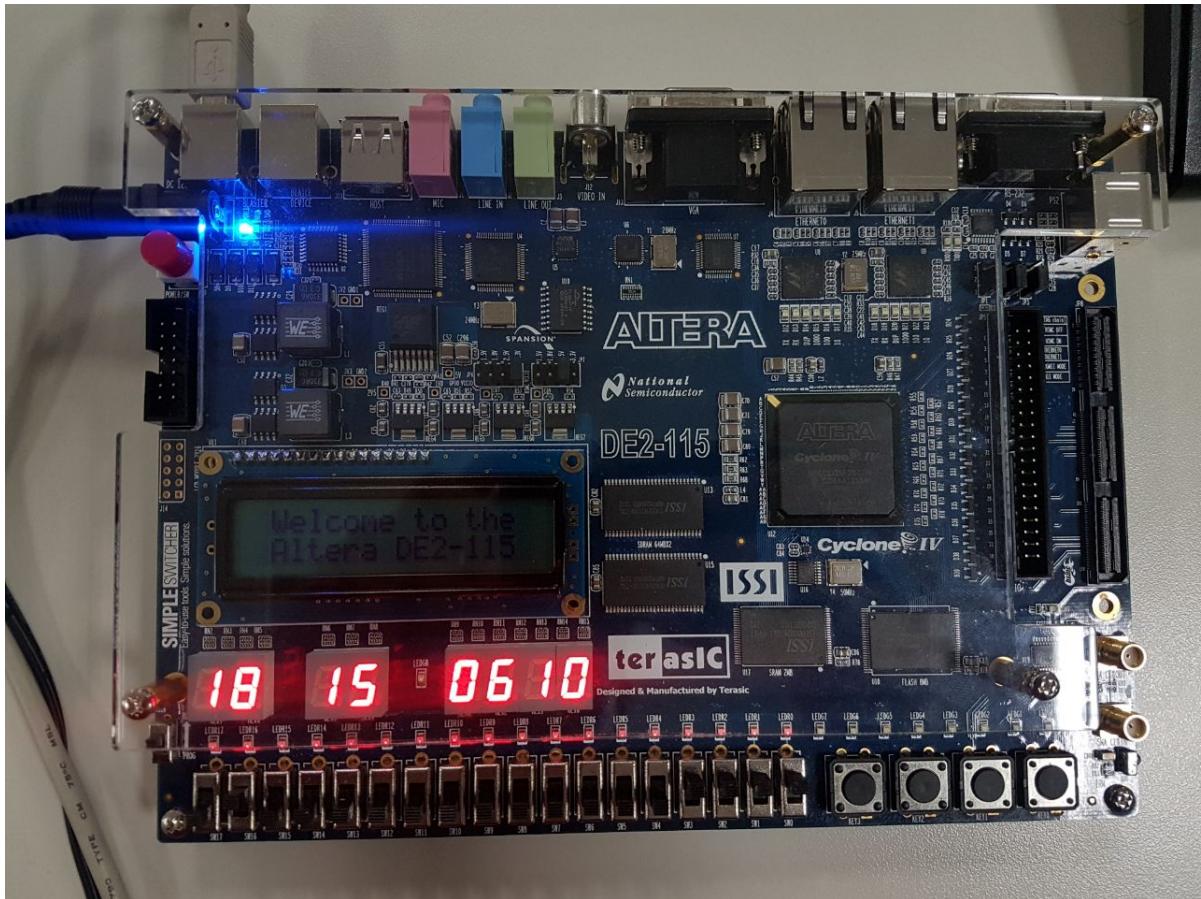
Assim como no teste anterior, o quadrado circulado em laranja representa o valor do último endereço do *Program Counter* realizado pelo processador, representando o valor 18 através do código BCD mostrado pelos *displays* D333 e D444.

O quadrado em vermelho representa o valor da quarta posição da sequência de *Fibonacci*, ou seja, o valor 3 que é representado pelos *displays* D1, D2, D3 e D4, onde os 3 primeiros *displays* estão recebendo '0000001' representando o valor 0 em código BCD e o último está recebendo o valor '0000110' representando o valor 3 em código BCD como mostra o quadrado azul.

Por fim o quadrado circulado em verde representa o valor lido pelo Módulo de Entrada através da escolha do usuário após ter setado o sinal de *checkin* representando o fim da escrita do imediato. No caso novamente os *displays* mais importantes a serem falados são os D33 e D44 que representam respectivamente 0 e 4.

Por último como mostra a Figura 21 representa a simulação do valor escolhido pelo usuário selecionando as chaves do FPGA, no caso o valor escolhido foi o valor 15. .

Figura 21 – Teste 3- Entrada 15

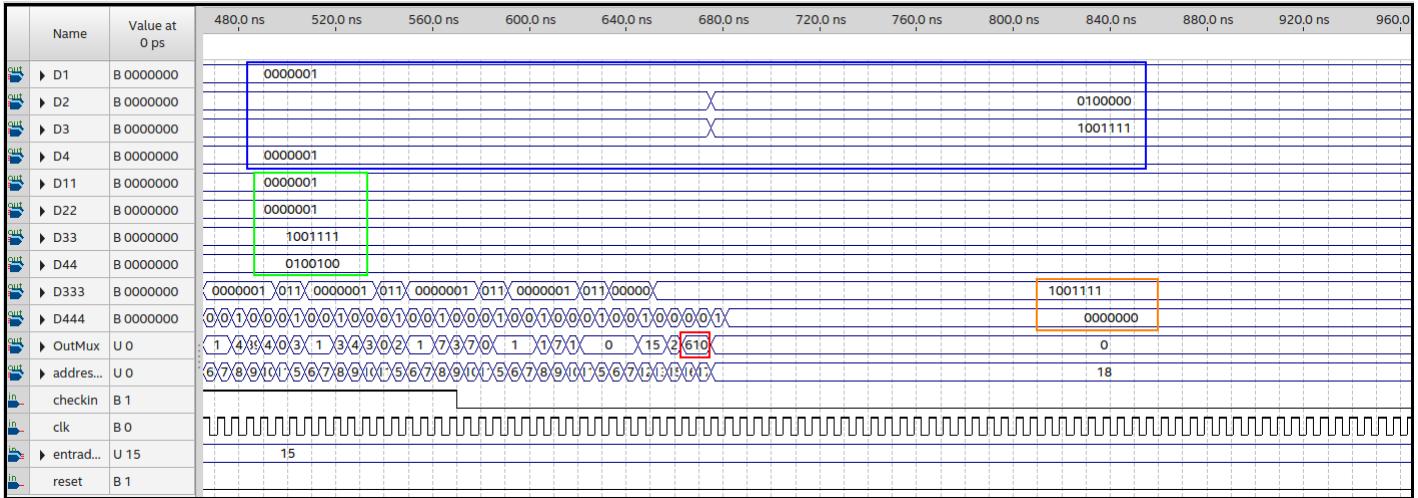


Fonte: O Autor

Da imagem acima se pode analisar que os dois primeiros *displays* representam a última instrução realizada pelo processador, no caso a instrução 18 que é a instrução de *halt* usada para manter o dado no *display*. Os dois próximos *displays* mostram o valor lido pelas chaves, no caso o usuário selecionou os *switches* 0, 1, 2 e 3 do FPGA representando o número '0000000000001111' em binário, ou melhor dizendo, o valor 15 em decimal. Os quatro últimos representam o valor do n-ésimo termo de *fibonacci* para o número lido, tal valor foi convertido para o código BCD e foi utilizado na instrução *out* para visualização do resultado, no caso o resultado foi o valor decimal 610.

Novamente foi realizado o mesmo teste no *software Quartus* como mostra a Figura 22 abaixo.

Figura 22 – Teste 3.1- Entrada 15



Fonte: O Autor

Como nos dois testes anteriores, o quadrado circulado em laranja representa o valor do último endereço do *Program Counter* realizado pelo processador, representando o valor 18 através do código BCD mostrado pelos *displays* D333 e D444.

O quadrado em vermelho representa o valor da décima quinta posição da sequência de *Fibonacci* , ou seja, o valor 610 que é representado pelos displays D1, D2, D3 e D4, onde o primeiros *display* está recebendo '0000001' representando o valor 0 em código BCD , o segundo '0100000' representando o valor 6, o terceiro '1001111' representando o valor 1 e o último está recebendo o valor '0000001' repesentando o valor 0 como mostra o quadrado azul.

Por fim o quadrado circulado em verde representa o valor lido pelo Módulo de Entrada através da escolha do usuário após ter setado o sinal de *checkin* representando o fim da escrita do imediato. No caso novamente os *displays* mais importantes a serem falados são os D33 e D44 que representam repectivamente a 1 e 5 como mostra o valor '0100100'.

5.2.4 Algoritmo de Verificação de Igualdade da Área de Dois Triângulos

O seguinte algoritmo tem como finalidade verificar se dois triângulos possuem área de valor igual, tal algoritmo foi feito para que se pudesse maximizar o uso de todas as instruções existentes no Conjunto de Instruções definido. A premissa do algoritmo é bem simples, o usuário entra com os valor de base e a altura de dois triângulos, o algoritmo calcula a área de ambos e verifica se são iguais ou não. Caso sejam iguais se deve ter um resultado de saída 2, caso sejam diferentes o valor de saída deve ser 1.

Tabela 13 – Instruções do Algoritmo de Verificação de Igualdade da Área de Dois Triângulos

(0) - nop
(1) - input r3
(2) - nop
(3) - nop
(4) - nop
(5) - input r2
(6) - mult r4, r3, r2
(7) - nop
(8) - nop
(9) - nop
(10) - shfr r4, r4
(11) - shfl r4, r4
(12) - ldi r5, 2
(13) - div r4, r4, r5
(14) - nop
(15) - input r1
(16) - nop
(17) - nop
(18) - input r6
(19) - mult r6, r6, r1
(20) - div r6, r6, r5
(21) - and r6, r6, r4
(22) - bne r6, r4, 26
(23) - subi r5, 1
(24) - out r5
(25) - halt
(26) - xor r4, r4, r6
(27) - bne r6, r4, 30
(28) - nop
(29) - nop
(30) - out r5
(31) - halt

Fonte: O Autor

Abaixo no Algoritmo 17 pode ser visto como as instruções estão dispostas na memória de instruções levando em conta o seu tipo e formato.

Algoritmo 17 - Verificação de Igualdade da Área de Dois Triângulos

```

1  instrmem[0]= 32'b010010_0000_0000_0000_000000000000; //nop
2  instrmem[1]= 32'b010100_0000_0000_00011_000000000000; // input base r3
3  instrmem[2]= 32'b010010_0000_0000_00000_000000000000; //nop
4  instrmem[3]= 32'b010010_0000_0000_00000_000000000000; //nop
5  instrmem[4]= 32'b010010_0000_0000_00000_000000000000; //nop
6  instrmem[5]= 32'b010100_0000_00000_00010_000000000000; // input altura r2
7  instrmem[6]= 32'b000100_00010_00011_00100_000000000000; // r[4]= base*altura //MULT r4
8  instrmem[7]= 32'b010010_0000_00000_00000_000000000000; //nop
9  instrmem[8]= 32'b010010_0000_00000_00000_000000000000; //nop
10 instrmem[9]= 32'b010010_0000_00000_00000_000000000000; //nop
11 instrmem[10]= 32'b011000_00100_00000_00100_00001_000000; // base*altura/2 - SHFR
12 instrmem[11]= 32'b010111_00100_00000_00100_00001_000000; // base*altura/2 - SHFL
13 instrmem[12]= 32'b001100_00000_00101_00000_00000000010; // r[5]=2 ldi r5

```

```

15 instrmem[13]= 32'b000101_00100_00101_00100_00000_00000; // r4 = base*altura/2 - div
16 instrmem[14]= 32'b010010_00000_00000_00000_000000000000; //nop
17 instrmem[15]= 32'b010100_00000_00000_00001_000000000000; // input base r1
18 instrmem[16]= 32'b010010_00000_00000_00000_000000000000; //nop
19 instrmem[17]= 32'b010010_00000_00000_00000_000000000000; //nop
20 instrmem[18]= 32'b010100_00000_00000_00110_000000000000; //input altura r6
21 instrmem[19]= 32'b000100_00001_00110_00110_000000000000; // r[6]= base*altura //MULT r4
22 instrmem[20]= 32'b000101_00110_00101_00110_000000000000; // base*altura/2 - SHFR - base*
   altura/2
23 instrmem[21]= 32'b001000_00100_00110_00110_00000_00000; // r4 = AND r4 r6
24 instrmem[22]= 32'b001111_00110_00100_00000_00000011010; // bne 26,[r4],r[6] //bne
25 instrmem[23]= 32'b000011_00101_00101_00000_00000000001; // r[5]-1 //subi
26 instrmem[24]= 32'b010101_00101_00000_00000_000000000000; // out r[5]
27 instrmem[25]= 32'b010011_00000_00000_00000_000000000000; //halt
28 instrmem[26]= 32'b001010_00100_00110_00100_000000000000; // r4 = xor r4 r6
29 instrmem[27]= 32'b001111_00100_00110_00000_00000011110; // bne 30,[r6],r[4] //bne
30 instrmem[28]= 32'b010010_00000_00000_00000_000000000000; //nop
31 instrmem[29]= 32'b010010_00000_00000_00000_000000000000; //nop
32 instrmem[30]= 32'b010101_00101_00000_00000_000000000000; // out r[5]
33 instrmem[31]= 32'b010011_00000_00000_00000_000000000000; //halt

```

O algoritmo acima funciona da seguinte maneira, primeiramente há uma instrução *Not An Operation* apenas para dar início ao programa, logo em seguida o usuário deve selecionar a base do primeiro triângulo e setar a entrada de *checkin*, tal valor é armazenado no registrador r3; As instruções 2, 3 e 4 não realizam nenhuma operação para que dê tempo do usuário entrar com outro valor que representa a altura do triângulo, tal valor o usuário seleciona na próxima instrução e esse valor é armazenado no registrador r2; Em seguida é multiplicado a base com a altura e o valor é salvo no registrador r4, logo após são utilizadas as instruções *shift right* e *shift left* com os *bits* de *offset* recebendo o valor '00001', isso divide o dado por 2 e multiplica o mesmo por 2 novamente, isso foi feito apenas para questões de teste; Logo após um registrador r5 recebe o valor 2, em seguida esse registrador é usado para o cálculo da área do primeiro triângulo, ou seja, a área será a base multiplicada pela altura e o resultado é dividido por 2 e salvo no registrador r4. Em seguida são feitas as leituras de base e altura do segundo triângulo e de maneira similar ao anterior é calculada a área e tal valor é salvo em um registrador r6.

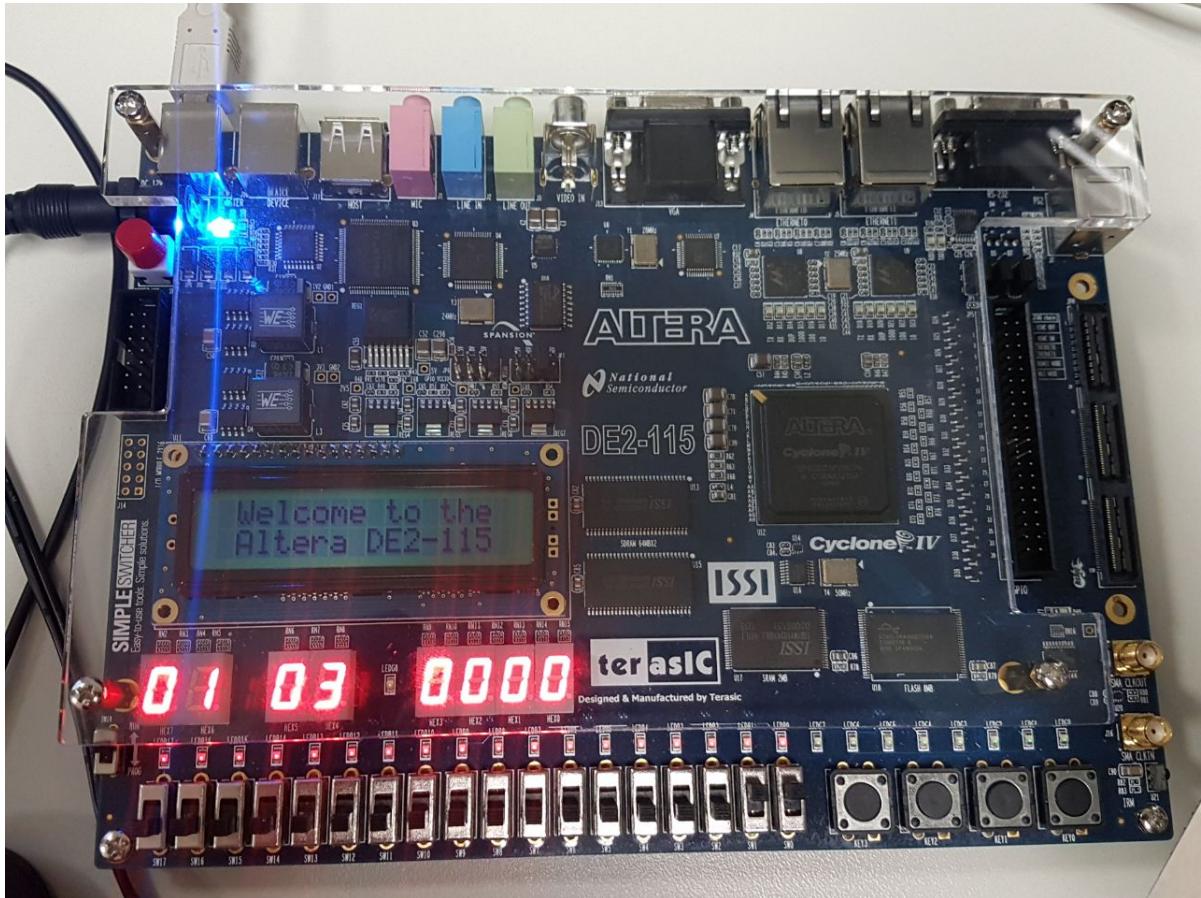
Tendo os valores das áreas há uma instrução de AND que tem como objetivo verificar se os triângulos são iguais, caso sejam iguais o valor que será armazenado no registrador 4 deve ser o mesmo que estava anteriormente, isso é verificado na próxima instrução que é a *branch if not equal*, caso as áreas dos triângulos sejam iguais, é subtraído o valor 1 do registrador r5, fazendo com que este fique armazenado com o valor 1, isso é feito através da instrução subi e o mesmo valor é mostrado através da instrução *out* seguida da instrução *halt* que irá manter o valor nos *displays* e para a contagem do *Program Counter*. Caso os triângulos sejam diferentes, o *branch* é tomado e vai para a instrução 26 onde é feita a instrução de XOR entre os registradores r4 e r6, tal valor é armazenado no registrador r4, caso o valor de r4 e r6 for diferente a próxima instrução de *branch if not equal* irá fazer um desvio para a instrução 30 que tem como objetivo mostrar o valor 2

nos *displays* indicando que as áreas do triângulos são diferentes, tal valor é mantido pela instrução de *halt* que paralisa o processador impedindo que o valor nos *displays* mude.

Para verificar a funcionalidade do algoritmo foram feitos dois testes no *hardware* FPGA, sendo o primeiro irá averiguar que as áreas são diferentes e o segundo verifica que as áreas são iguais. Nesse caso devido a complexidade ao receber os dados de entrada não foram feitos testes no *software Quartus*.

O primeiro teste recebe para ambos os triângulos o valor 3 tanto para base quanto para altura, assim sendo as áreas são iguais como mostram a Figura 23, Figura 24, Figura 25 e Figura 26 a seguir.

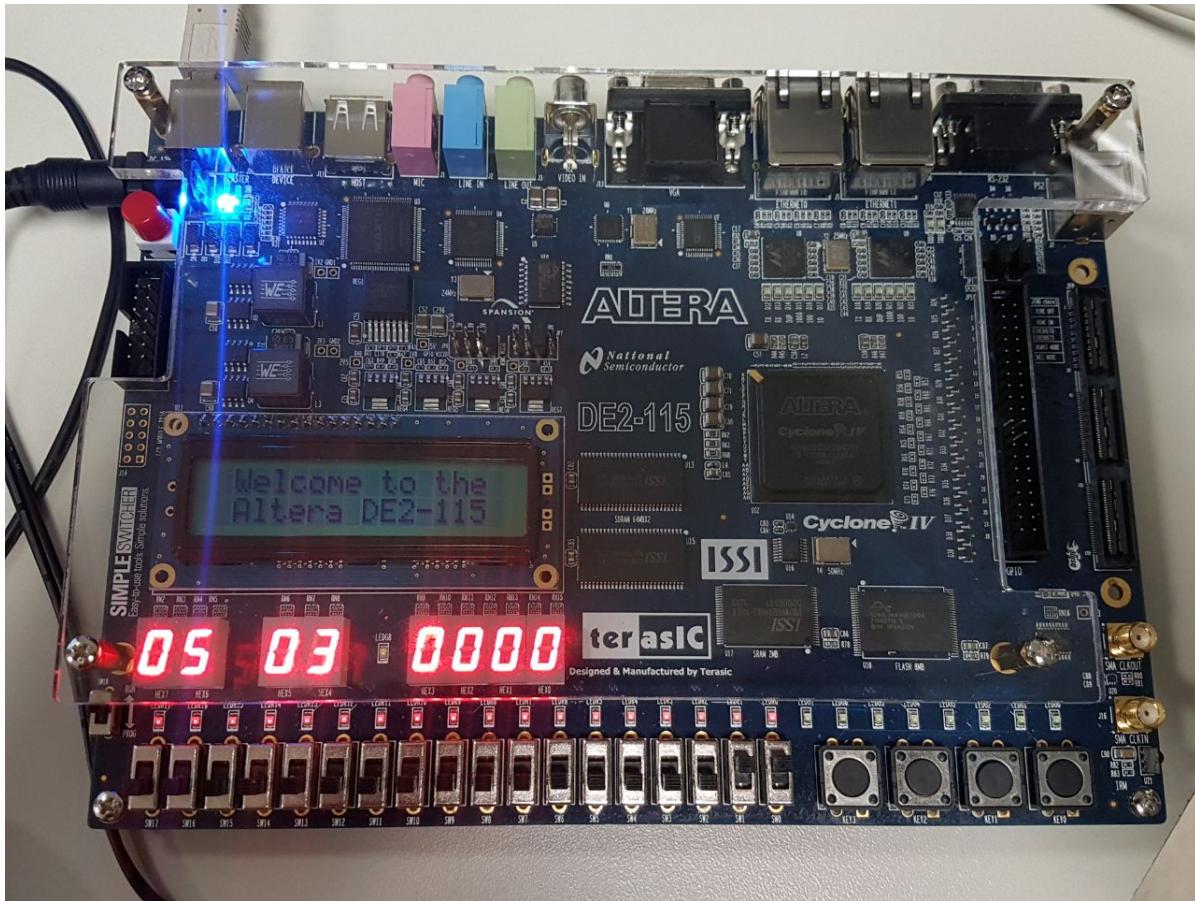
Figura 23 – Teste - Entrada 1.3



Fonte: O Autor

Na imagem acima, a instrução 1 de *Input* recebeu o valor 3 que foi selecionado pelo usuário ao setar as chaves 0 e 1 do FPGA representando o valor '11' em binário e que posteriormente lido após o sinal de *checkin* através da chave 17 e tal valor é mostrado nos displays HEX5 e HEX6.

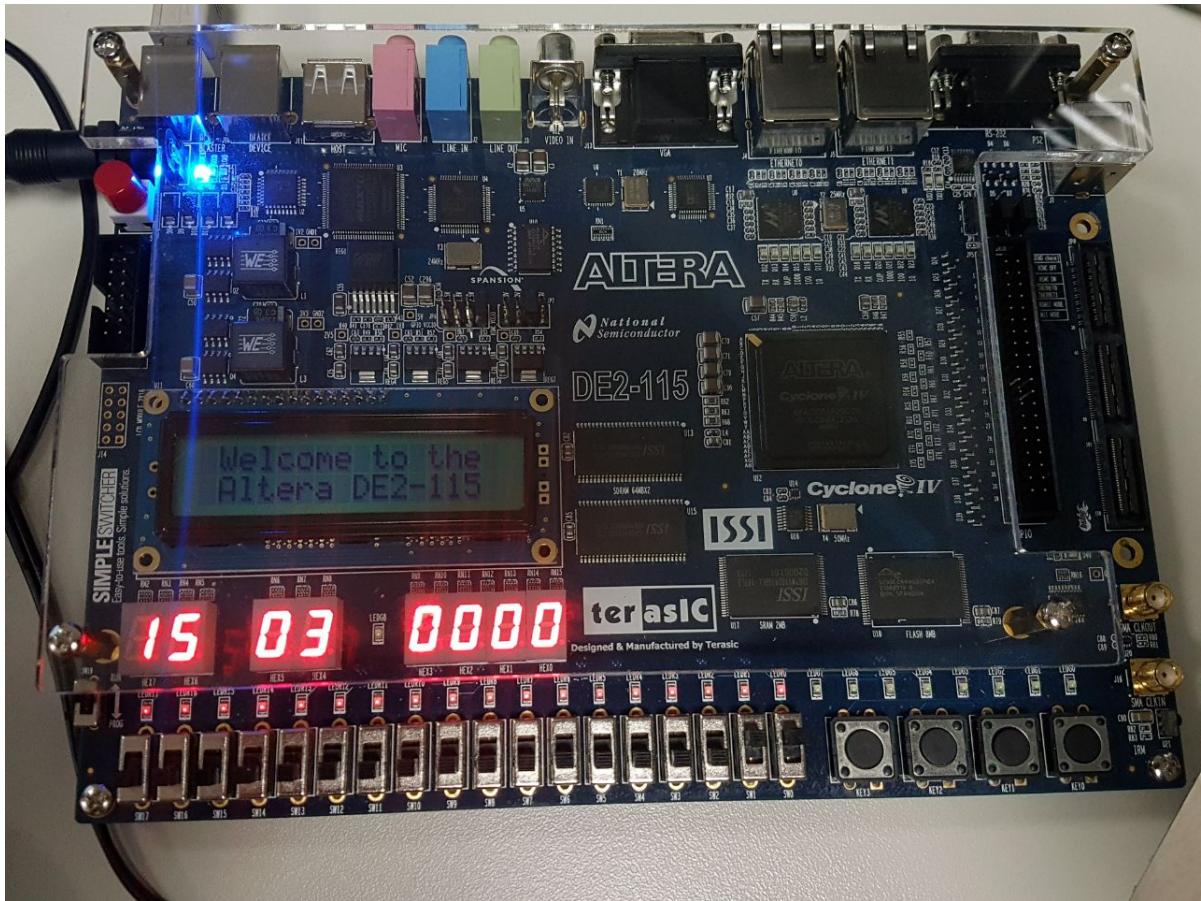
Figura 24 – Teste - Entrada 2.3



Fonte: O Autor

Na imagem acima, a instrução 5 de *Input* recebeu o valor 3 que foi selecionado pelo usuário ao setar as chaves 0 e 1 do FPGA representando o valor '11' em binário e que posteriormente foi lido após o sinal de *checkin* através da chave 17 e tal valor é mostrado nos displays HEX5 e HEX6..

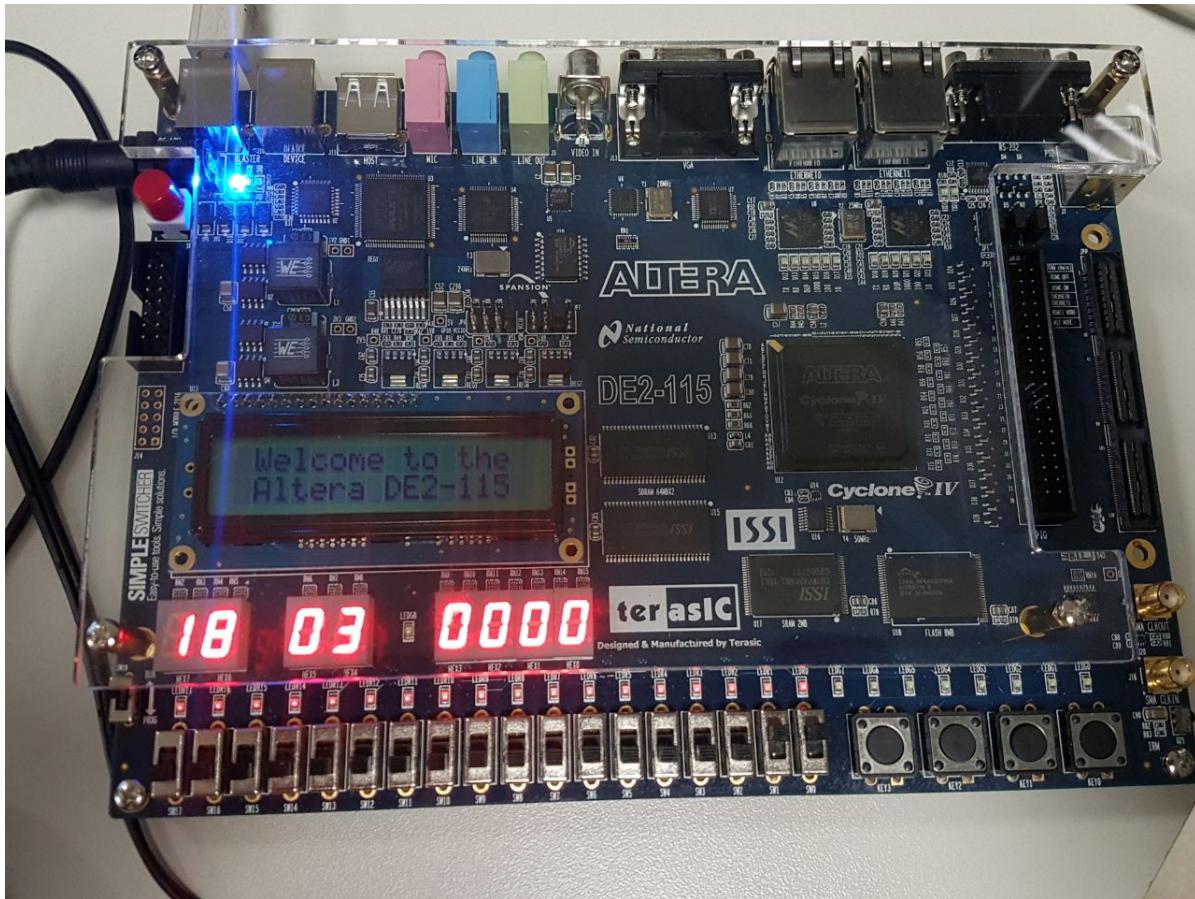
Figura 25 – Teste - Entrada 3.3



Fonte: O Autor

Na imagem acima, a instrução 15 de *Input* recebeu o valor 3 que foi selecionado pelo usuário ao setar as chaves 0 e 1 do FPGAr epresentando o valor '11' em binário e que posteriormente foi lido após o sinal de *checkin* através da chave 17 e tal valor é mostrado nos displays HEX5 e HEX6..

Figura 26 – Teste - Entrada 4.3

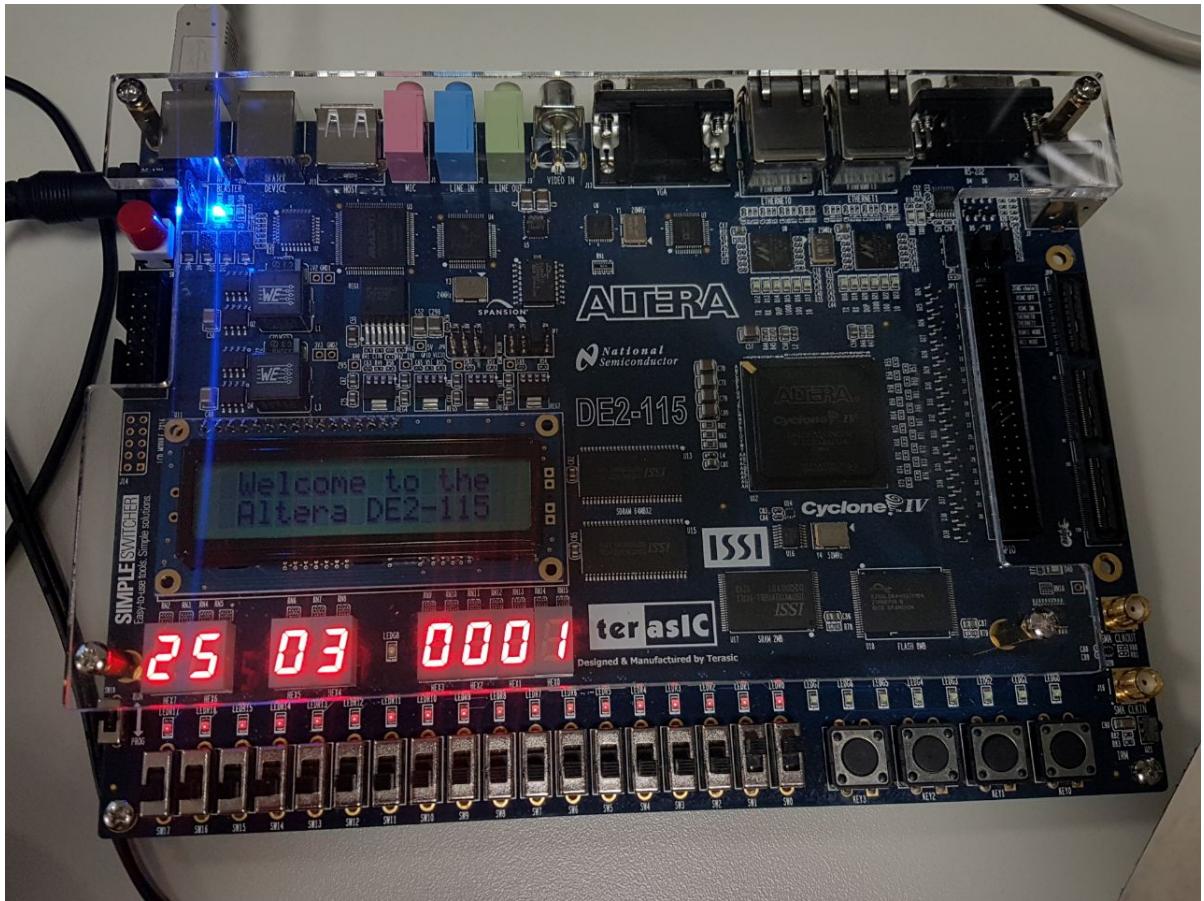


Fonte: O Autor

De acordo com a figura acima, a instrução 18 de *Input* recebeu o valor 3 que foi selecionado pelo usuário ao setar as chaves 0 e 1 do FPGA representando o valor '11' em binário e que posteriormente lido após o sinal de *checkin* através da chave 17 e tal valor é mostrado nos displays HEX5 e HEX6.

Tendo feito as leituras das bases e alturas o algoritmo foi capaz de calcular ambas as áreas dos triângulos e mostrar o resultado como mostra a [Figura 27](#) abaixo.

Figura 27 – Teste - Saída com áreas iguais

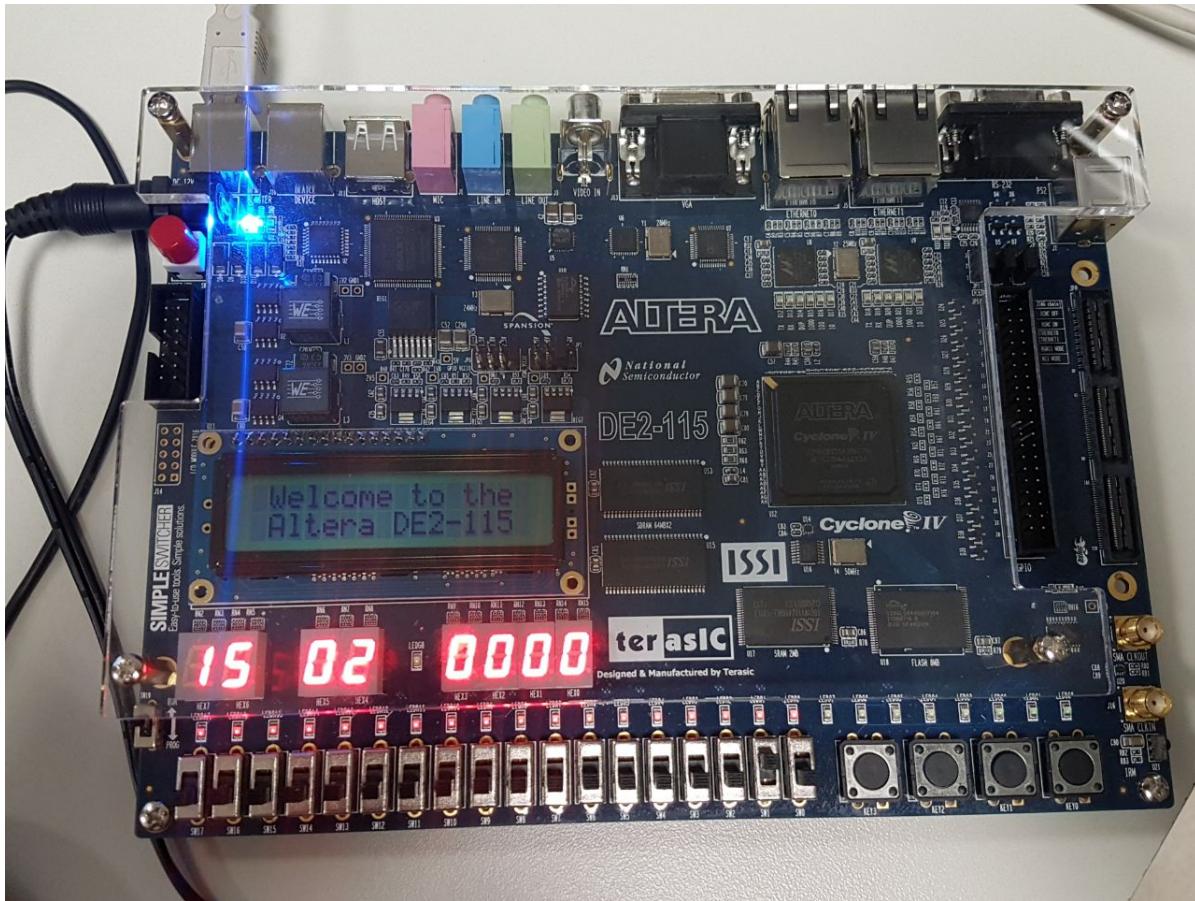


Fonte: O Autor

De acordo com a figura acima, o algoritmo parou na instrução 25 como mostram os dois primeiros *displays*, e o valor de saída que pode ser visto nos últimos 4 é igual a 1 indicando que de fato as áreas dos triângulos são iguais.

O segundo e último teste foi feito mudando apenas as entradas lidas pelas instruções 15 e 18 como mostram a [Figura 28](#) e [Figura 29](#) a seguir.

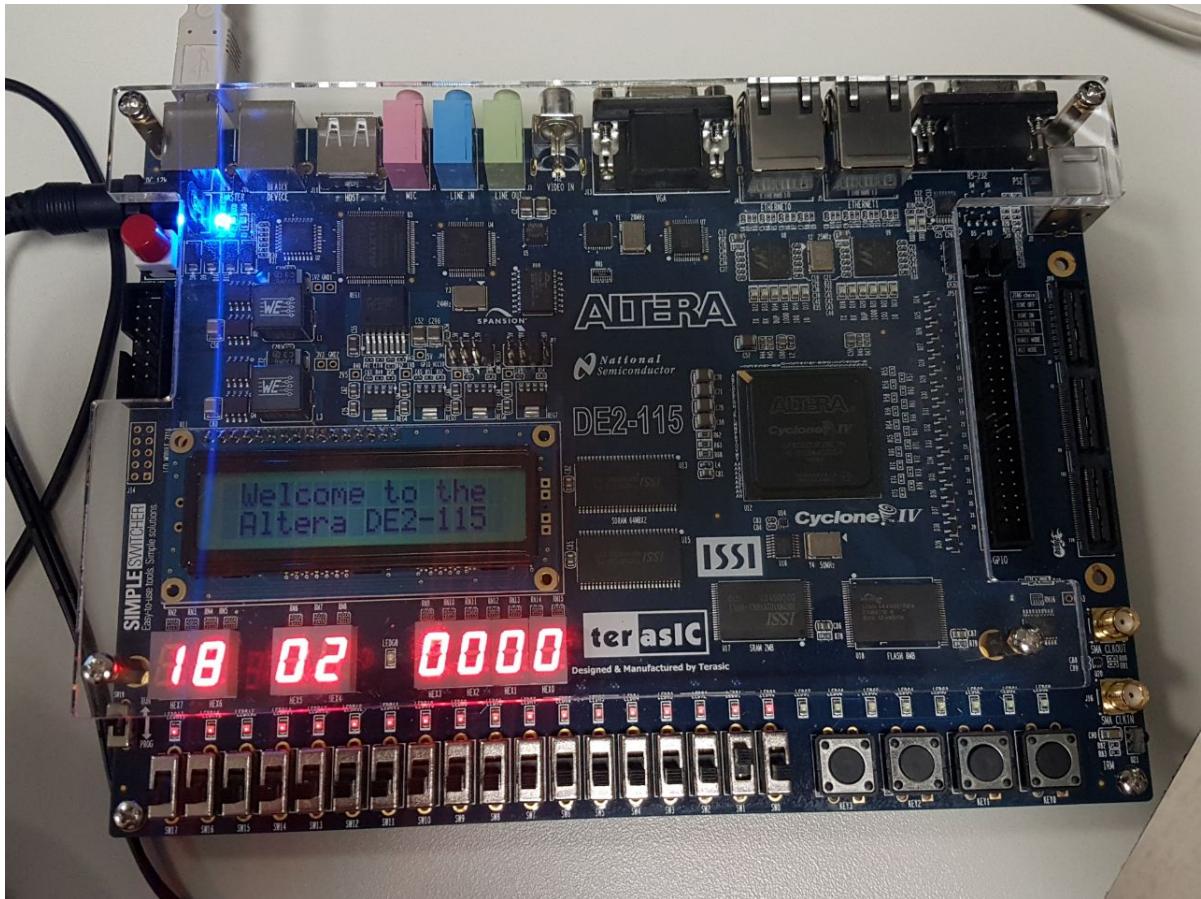
Figura 28 – Teste - Entrada 3.2



Fonte: O Autor

Na imagem acima, a instrução 15 de *Input* recebeu o valor 2 que foi selecionado pelo usuário ao setar as chaves 0 e 1 do FPGA representando o valor '10' em binário e que posteriormente foi lido após o sinal de *checkin* através da chave 17 e tal valor é mostrado nos displays HEX5 e HEX6..

Figura 29 – Teste - Entrada 4.2

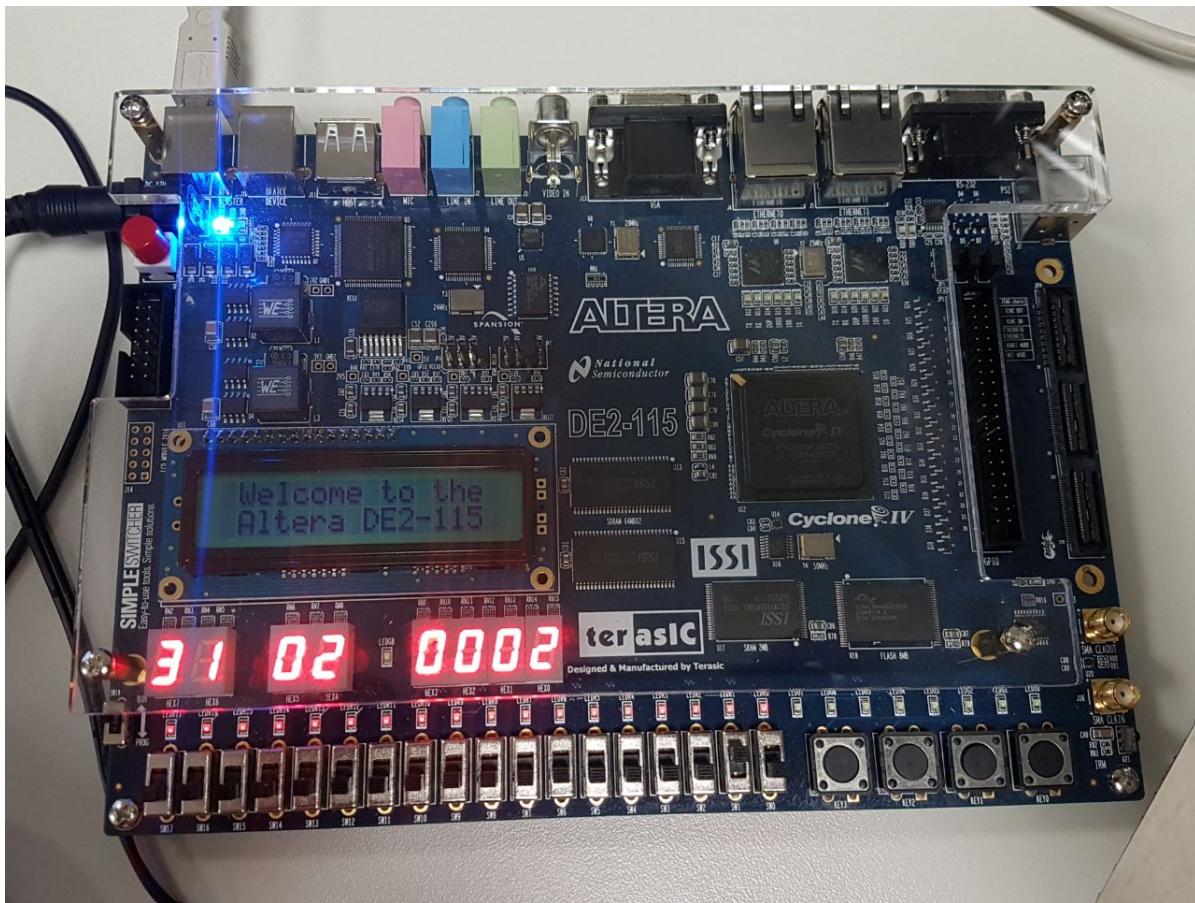


Fonte: O Autor

De acordo com a figura acima, a instrução 18 de *Input* recebeu o valor 2 que foi selecionado pelo usuário ao setar as chaves 0 e 1 do FPGA representando o valor '10' em binário e que posteriormente foi lido após o sinal de *checkin* através da chave 17 e tal valor é mostrado nos *displays* HEX5 e HEX6.

Por fim após o cálculo das áreas de ambos os triângulos o código verificou os valores e reproduziu um valor de saída para o FPGA que pode ser visto a seguir na [Figura 30](#)

Figura 30 – Teste - Saída com áreas diferentes



Fonte: O Autor

De acordo com a figura acima, os dois primeiros *displays* indicam que o algoritmo parou na instrução 31 indicando que realmente as áreas são diferentes e isso pode ser visto através dos últimos quatro *displays* que mostram um valor de resultado igual a 2.

Assim através da [Figura 27](#) e [Figura 30](#) foi possível averiguar a veracidade não só do algoritmo mas do processador como um todo além de que foi utilizado o maior número de instruções possíveis.

6 Considerações Finais

O planejamento de um sistema computacional eficaz é uma ideia um tanto quanto interessante pois permite que o aluno tenha um melhor contato prático com arquitetura e organização de computadores podendo colocar em prática todo o estudo já aprendido além de ter uma visualização em tempo real do que pode acontecer dentro do sistema.

O desenvolvimento da Unidade de Processamento com todos os componentes que a compõem bem como o desenvolvimento da Unidade de Controle e o Módulo de Entrada e Saída é um trabalho que exige paciência e dedicação mas que teve seu objetivo alcançado com êxito. Entretanto a implementação apresentou algumas dificuldades por conta da complexidade e a necessidade de se fazer alterações em uma arquitetura de processadores já existente. As maiores dificuldades encontradas no desenvolvimento do projeto foi em relação ao mapeamento de cada uma das instruções, pois cada uma funciona de maneira diferente, necessita de registradores diferentes, a maneira como um dado é salvo também se torna diferente, assim sendo a dificuldade já se inicia desde o desenvolvimento das instruções pois ao desenvolver a Unidade de Controle se fez necessário se atentar ao formato de cada instrução e como a mesma flui no *Datapath* para que os sinais de controle pudessem operar de tal forma a fazer com que o sistema funcione com exatidão. Além disso, outro problema relacionado as instruções vem também na parte de testes pois se torna trabalhoso o fato de ter que escrever todas as instruções.

Outra dificuldade encontrada foi na integração entre a Unidade de Processamento junto com os módulos de entrada e saída ao se comunicarem com a placa FPGA. Muitas das vezes as simulações no *software Quartus* ocorriam da maneira esperada, entretanto ao averiguar o funcionamento das mesmas simulações no *hardware* o resultado não era de fato que estava sendo esperado, isso ocorreu muitas vezes devido a problemas no *software* e até mesmo problemas de projeto na arquitetura que teve de ser modificada diversas vezes ao longo do desenvolvimento do projeto.

Entretanto, por mais trabalhoso que possa ter sido a implementação processador, foi interessante entender que o que foi desenvolvido é tanto o corpo quanto o cérebro do processador, que é análogo a Unidade de Processamento e sua Unidade de Controle, garantido assim um total funcionamento para o sistema. Assim foi possível ver a importância de um processador para estimular a abstração da operação lógica e sequêncial de circuitos digitais em um nível de complexidade superior. O fato de ter se desenvolvido um projeto deste tipo também serviu como uma ferramenta de aprendizagem aprofundada quando se fala de sistemas computacionais e como ele funciona, pois é de tremenda diferença analisar como um processador funciona na teoria em relação ao seu funcionamento na prática,

principalmente pois se deve levar em conta o passo a passo de construção de todos os componentes que o compõe.

Por fim é válido citar que processador baseado na arquitetura MIPS monociclo desenvolvido neste projeto conta com um conjunto de instruções que pode ser expansível, fazendo com que se torne fácil a implementação de possíveis novas instruções para usos futuros.

Referências

- 1 AURELIO, D. do. *Computador*. Disponível em: <<https://dicionariodoaurelio.com/computador>>. Citado na página 8.
- 2 PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design*. 5th edition. ed. Waltham/MA, EUA: Morgan Kaufmann, 2007. Citado 5 vezes nas páginas 8, 13, 14, 17 e 18.
- 3 TANENBAUM, A. S. *Organização Estruturada de Computadores*. 6th edition. ed. [S.l.]: Pearson, 2013. Citado 2 vezes nas páginas 8 e 12.
- 4 PEIXOTO, P. D. da S. *Introdução aos Sistemas Computacionais*. Disponível em: <<https://www1.univap.br/patriciadias/aula03SC.pdf>>. Citado na página 12.
- 5 STALLINGS, W. *Computer Organization and Architecture*. 9th edition. ed. [S.l.]: CL Engineering, 2013. Citado 4 vezes nas páginas 12, 13, 15 e 24.
- 6 DAMASCENO, A. L. *O Impacto da Hierarquia de Memória sobre a Arquitetura IPNOSYS*. Disponível em: <<https://ppgcc.ufersa.edu.br/wp-content/uploads/sites/42/2014/09/O-IMPACTO-DA-HIERARQUIA-DE-MEMÓRIA-SOBRE-A-ARQUITETURA-IPNOSYS-1.pdf>>. Citado na página 12.
- 7 BRITO, A. V. de. *RISC X CISC*. Disponível em: <<http://producao.virtual.ufpb.br/books/edusantana/introducao-a-arquitetura-de-computadores-livro/livro/chunked/ch04s04.html>>. Citado na página 14.
- 8 PASSOS, D. *Arquiteturas RISC vs. CISC*. Disponível em: <<https://www.kitronik.co.uk/blog/how-an-ldr-light-dependent-resistor-works/>>. Citado na página 14.
- 9 CHEN, C.; NOVICK, G.; SHIMANO, K. *RISC vs CISC*. Disponível em: <<https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/riscscisc/>>. Citado na página 14.
- 10 OLIVEIRA, N. M. de. Paralelo entre as arquiteturas von neumann e harvard. *Universidade Federal Rural de Pernambuco*, 2014. Disponível em: <<https://goo.gl/9DQbcG>>. Citado na página 15.
- 11 GONÇALVES, B. *Unidade Central de Processamento (CPU) Processador*. Citado na página 15.
- 12 COM, V. D. *Verilog Resources*. <<http://www.verilog.com/index.html>>. Citado 2 vezes nas páginas 20 e 21.
- 13 WIKIPEDIA. *Field programmable gate array*. 2018. <https://pt.wikipedia.org/wiki/Field-programmable_gate_array>. Citado na página 21.

Apêndices

APÊNDICE A – Apêndice 1

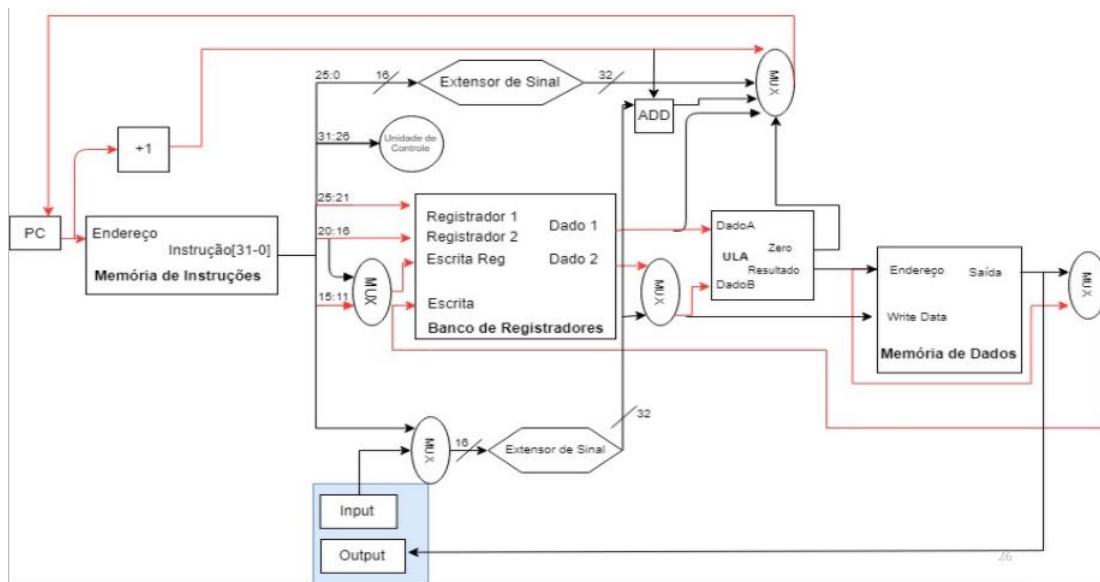
Nesta seção serão abordados exemplos de algumas instruções de exemplo no desenvolvimento do projeto.

A primeira instrução é uma adição. De início o endereço do *Program Counter* é somado com 1 para que seja feita a busca da próxima instrução.

O endereço da instrução atual chega na Memória de Instruções onde será feita a busca da instrução. Tendo feita a busca da instrução é buscado no Banco de Registradores os dois registradores de destino bem como o preparo do registrador de destino no Banco de Registradores.

Os dados dos registradores fonte são extraídos e entram diretamente na Unidade Lógica Aritmética. A ULA executa a operação e o resultado irá para um multiplexador que enviará o resultado da soma para o registrador de destino que está no banco.

Figura 31 – Exemplo de uma instrução add



Fonte: O Autor

A segunda instrução de exemplo é uma instrução de *load word*. De início o endereço do *Program Counter* é somado com 1 para que seja feita a busca da próxima instrução.

O endereço da instrução atual chega na Memória de Instruções onde será feita a busca da instrução. Tendo feita a busca da instrução é buscado no Banco de Registradores apenas dois registradores, um de fonte e outro de destino.

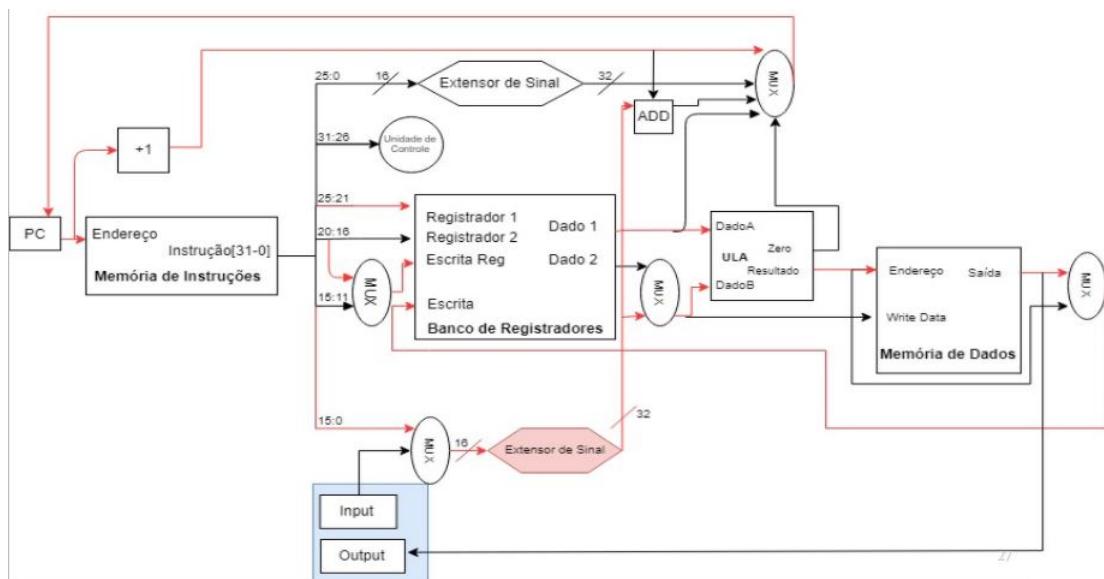
Simultaneamente um imediato de 16 bits é estendido em 32 bits pelo extensor de

sinal e esse valor entra na ULA.

A ULA irá calcular um endereço de uma posição na Memória de Dados onde está armazenado o que deseja ser armazenado através do imediato estendido e do endereço do registrador fonte.

Após feita a busca do dado, o mesmo é armazenado no registrador de destino no Banco de Registradores

Figura 32 – Exemplo de uma instrução load word



Fonte: O Autor

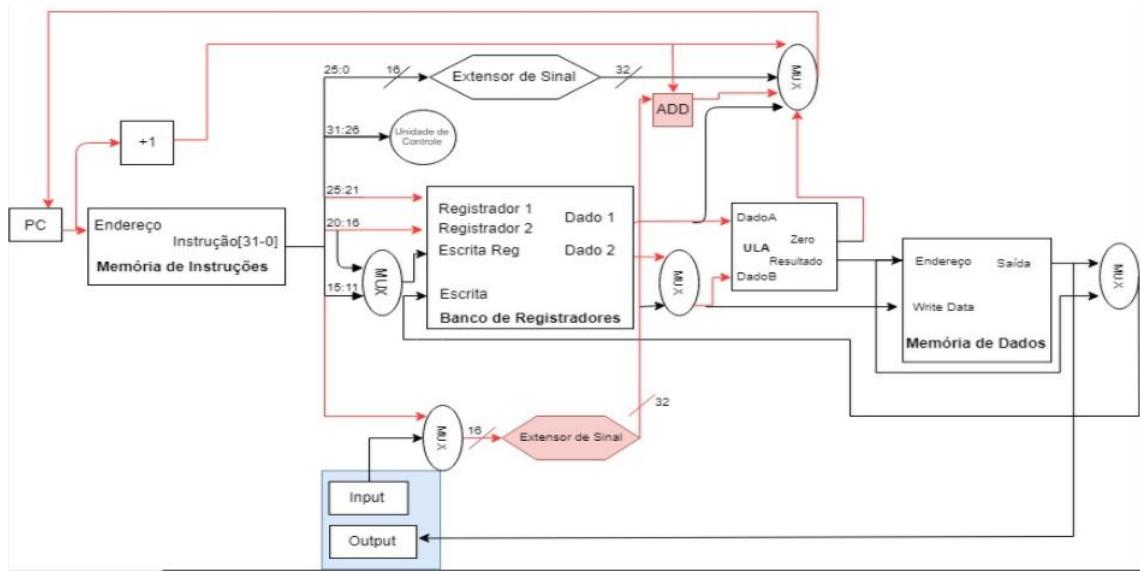
A terceira instrução de exemplo é a instrução de *branch if equal*. De início o endereço do *Program Counter* é somado com 1 para que seja feita a busca da próxima instrução.

O valor de PC entra na Memória de Instruções onde será feita a busca da instrução que será executada. Tendo feita a busca da instrução é buscado no Banco de Registradores dois registradores dos quais os valores armazenados serão usados para fazer a comparação.

Os dados são extraídos dos registradores e é feita a comparação na ULA, caso os valores sejam iguais, a saída *zero* da ULA emite um sinal em valor que vai para um multiplexador que emite uma saída de atualização do próximo valor de PC.

O PC será atualizado através de um imediato de 16 bits que é estendido em 32 bits pelo extensor de sinal. Esse valor estendido entra em um somador que fará a soma com o endereço de PC já somado com 1, resultando no endereço de desvio.

Figura 33 – Exemplo de uma instrução beq



Fonte: O Autor

APÊNDICE B – Apêndice 2

B.1 Algoritmos de Componentes Auxiliares

Algoritmo 18 - Extensor de 26 bits

```

1 module Extensordebit26(entrada,saida);
2
3 input [25:0] entrada;
4 output reg [31:0] saida;
5
6 always @(*) begin
7
8     saida = {{6{1'b0}}, entrada};
9 end
10
11 endmodule

```

Algoritmo 19 - MUX1

```

1 module MUX1(inputReg1,inputReg2,saida,Regdest);
2
3 input [31:0] inputReg1,inputReg2;
4 input Regdest;
5
6 output reg[31:0] saida;
7
8 always @ (*) begin
9
10    case(Regdest)
11        1'b0: saida = inputReg1;
12        1'b1: saida = inputReg2;
13    endcase
14 end
15 endmodule

```

Algoritmo 20 - MUX2

```

1 module MUX2(inputRT, inputextensor, saida, AluSrc);
2
3 input [31:0] inputRT,inputextensor;
4 input AluSrc;
5
6 output reg[31:0] saida;
7
8 always @(*) begin
9
10    case(AluSrc)
11        1'b0: saida = inputRT;
12        1'b1: saida = inputextensor;
13    endcase
14 end
15

```

16 **endmodule**

Algoritmo 21 - MUX3

```

1  module  MUX3(aluResult ,readMemoria ,saída ,memToReg );
2
3  input  [31:0] aluResult ,readMemoria;
4  input  memToReg ;
5
6  output reg [31:0] saída;
7
8  always @(*) begin
9
10         case(memToReg)
11             1'b0: saída = aluResult;
12             1'b1: saída = readMemoria;
13         endcase
14     end
15
16 endmodule

```

Algoritmo 22 - MUX5

```

1
2
3  module  MUX5(inputIN , inputbanco , saídaMUX5 , flagIN );
4
5  input  [15:0] inputIN ,inputbanco;
6  input  flagIN ;
7
8  output reg [15:0] saídaMUX5;
9
10 always @(*) begin
11
12         case(flagIN)
13             1'b0: saídaMUX5 = inputbanco;
14             1'b1: saídaMUX5 = inputIN;
15         endcase
16     end
17
18 endmodule

```

Algoritmo 23 - ExtensordebitIN

```

1  module  ExtensordebitIN(entrada ,saída16 ,reset ,clock );
2
3  input  [15:0] entrada;
4  input  reset ,clock;
5  output reg [31:0] saída16;
6
7  always @ (posedge clock)
8  begin
9         if(reset==1)
10             saída16= 32'b0;
11         else
12             saída16 = {{16{1'b0}} , entrada};
13     end
14

```

```
15  endmodule
```

Algoritmo 24 - DisplayPC

```

1 module Display_PC (binary , R1 , R2 , R3 , R4 , clock);
2
3     input [31:0] binary;
4     input clock;
5     output reg [6:0] R1 , R2 , R3 , R4;
6     wire [15:0] aux;
7     wire [6:0] A1 , A2 , A3 , A4;
8
9     binario_BCD binario_BCD (binary[15:0] , aux);
10
11
12     DISPLAYBCD U1 (aux[15:12] , A1); //unidade
13     DISPLAYBCD U2 (aux[11:8] , A2); //dezena
14     DISPLAYBCD U3 (aux[7:4] , A3); //centena
15     DISPLAYBCD U4 (aux[3:0] , A4); //mil
16
17     always@ (posedge clock)
18         begin
19             R1=A1;
20             R2=A2;
21             R3=A3;
22             R4=A4;
23         end
24 endmodule

```

Algoritmo 24 - DisplayIN

```

1 module Display_IN (binary , R1 , R2 , R3 , R4 , clock);
2
3     input [31:0] binary;
4     input clock;
5     output reg [6:0] R1 , R2 , R3 , R4;
6     wire [15:0] aux;
7     wire [6:0] A1 , A2 , A3 , A4;
8
9     binario_BCD binario_BCD (binary[15:0] , aux);
10
11
12     DISPLAYBCD U1 (aux[15:12] , A1); //unidade
13     DISPLAYBCD U2 (aux[11:8] , A2); //dezena
14     DISPLAYBCD U3 (aux[7:4] , A3); //centena
15     DISPLAYBCD U4 (aux[3:0] , A4); //mil
16
17     always@ (posedge clock)
18         begin
19             R1=A1;
20             R2=A2;
21             R3=A3;
22             R4=A4;
23         end
24 endmodule

```