

Mateus Vespasiano de Castro
159505

Desenvolvimento de um Processador MIPS com Utilização do kit FPGA em Verilog

São José dos Campos - Brasil

Abril de 2023

Mateus Vespasiano de Castro
159505

Desenvolvimento de um Processador MIPS com Utilização do kit FPGA em Verilog

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

Docente: Prof. Dr. Sérgio Ronaldo Barros dos Santos
Universidade Federal de São Paulo - UNIFESP
Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil
Abril de 2023

Resumo

Ao longo da disciplina de Arquitetura e Organização de Computadores os alunos aprenderam diversos conceitos sobre o funcionamento de processadores, desde periféricos, até componentes internos, como memória e caminho de dados. Conceitos estes, que foram colocados em prática neste projeto do laboratório de Arquitetura e Organização de Computadores. Utilizando a linguagem de descrição de *Hardware Verilog*, em que será implementado um processador funcional com todos os seus módulos, ou seja, banco de registradores, unidade lógica e aritmética, memória principal e de instruções e suas interconexões. Ele deverá ser submetido e testado na plataforma FPGA Altera DE2-115.

Palavras-chaves: Arquitetura e Organização de Computadores. MIPS. Kit FPGA.

Listas de ilustrações

Figura 1 – Arquitetura de Harvard	14
Figura 2 – Arquitetura de Von Neumann	15
Figura 3 – Arquitetura e <i>Datapath</i> para o MIPS	17
Figura 4 – FPGA Imagem	21
Figura 5 – Caminho de dados para o processador a ser desenvolvido.	24
Figura 6 – Caminho de dados tipo I	25
Figura 7 – Caminho de dados tipo R	26
Figura 8 – Caminho de dados tipo LW	27
Figura 9 – Caminho de dados tipo J	28
Figura 10 – Caminho de dados tipo JR	29
Figura 11 – Caminho de dados tipo JAL	30
Figura 12 – Caminho de dados tipo Branch	31
Figura 13 – Unidade Lógica e Aritmética	32
Figura 14 – Código em verilog para a unidade lógica e artimética	33
Figura 15 – Código em verilog para a unidade lógica e artimética	34
Figura 16 – Banco de Registradores	34
Figura 17 – Código em verilog para o banco de registradores.	35
Figura 18 – Unidade de Processamento	36
Figura 19 – Unidade de Processamento Verilog	37
Figura 20 – Unidade de Processamento Verilog	38
Figura 21 – Memoria de Dados (RAM)	39
Figura 22 – Memoria de Instruções (ROM)	40
Figura 23 – Contador de Programa (PC)	41
Figura 24 – Unidade de Controle	42
Figura 25 – Unidade de Controle da ULA	43
Figura 26 – Módulo de Entrada	44
Figura 27 – Módulo de Saída	45
Figura 28 – BCD 7 Segmentos	45
Figura 29 – Mux ALUSrc	46
Figura 30 – Mux MemReg	46
Figura 31 – Mux RegDst	47
Figura 32 – Forma de onda AND - Controle (0000)	48
Figura 33 – Forma de onda OR - Controle (0001)	48
Figura 34 – Forma de onda ADD - Controle (0010)	49
Figura 35 – Forma de onda SUB - Controle (0011)	49
Figura 36 – Forma de onda SLT - Controle (0100)	49

Figura 37 – Forma de onda SGT - Controle (0101)	49
Figura 38 – Forma de onda SGET - Controle (0110)	50
Figura 39 – Forma de onda SLET - Controle (0111)	50
Figura 40 – Forma de onda MULT - Controle (1000)	50
Figura 41 – Forma de onda DIV - Controle (1001)	51
Figura 42 – Forma de onda NOR - Controle (1010)	51
Figura 43 – Forma de onda SLL - Controle (1011)	51
Figura 44 – Forma de onda SRL - Controle (1100)	51
Figura 45 – Forma de onda de leitura do Banco de Registradores	52
Figura 46 – Forma de onda de escrita no Banco de Registradores	52
Figura 47 – Arquivo txt com as instruções	53
Figura 48 – Forma de onda instruções	53
Figura 49 – Sequência de Fibonacci Registradores	54
Figura 50 – Sequência de Fibonacci forma de onda	54
Figura 51 – Sequência de Fibonacci FPGA	55

Lista de tabelas

Tabela 1 – Formato de Instruções do tipo R para o processador MIPS	18
Tabela 2 – Formato de Instruções do tipo I para o processador MIPS	18
Tabela 3 – Formato de Instruções do tipo J para o processador MIPS	19
Tabela 4 – Instruções do tipo R	23
Tabela 5 – Instruções do tipo I	23
Tabela 6 – Instruções do tipo J	23
Tabela 7 – Operações da ULA, que são ativadas conforme o valor de controle. . .	32

Sumário

1	INTRODUÇÃO	8
2	OBJETIVOS	10
2.1	Geral	10
2.2	Específico	10
3	FUNDAMENTAÇÃO TEÓRICA	11
3.1	Arquiteturas CISC e RISC	11
3.1.1	Arquitetura CISC	11
3.1.2	Arquitetura RISC	12
3.2	Arquitetura de Von Neumann e Harvard	13
3.3	Processador MIPS	15
3.3.1	Arquitetura Básica	16
3.3.2	Formato de Instruções	17
3.3.3	Modos de Endereçamento	19
3.3.3.1	A Registrador	19
3.3.3.2	Endereçamento Imediato	19
3.3.3.3	Base-Deslocamento	19
3.3.3.4	Relativo ao PC	20
3.3.3.5	Absoluto	20
3.4	Linguagem de Descrição de Hardware	20
3.5	<i>Field Programmable Gate Array (FPGA)</i>	21
4	DESENVOLVIMENTO	22
4.1	Características do Processador	22
4.2	Conjunto de Instruções	22
4.3	Modos de Endereçamento	24
4.4	Caminho de Dados	24
4.5	Implementação do Projeto	31
4.5.1	Unidade Lógica e Aritmética (ULA)	31
4.5.2	Banco de Registradores	34
4.5.3	Unidade de Processamento	35
4.5.4	Memória de Dados (RAM)	39
4.5.5	Memória de Instrução (ROM)	40
4.5.6	Registrador Contador de Programa (PC)	40
4.5.7	Unidade de Controle	41

4.5.8	Unidade de Controle da ULA	42
4.5.9	Módulos de Entrada e Saída	43
4.5.10	Multiplexadores	46
4.5.11	Integração Final	47
5	RESULTADOS OBTIDOS E DISCUSSÕES	48
5.1	Formas de Onda	48
5.1.1	Teste da Unidade Lógica e Aritmética	48
5.1.2	Teste do Banco de Registradores	52
5.1.3	Teste da Memória ROM	52
5.2	Teste no <i>kit</i> FPGA	53
6	CONSIDERAÇÕES FINAIS	56
	REFERÊNCIAS	57

1 Introdução

Um processador, também conhecido como a CPU (*Unidade Central de Processamento*), é um componente essencial em praticamente todos os dispositivos eletrônicos modernos, desde *smartphone* e laptops até sistemas de navegação em aeronaves. O processador é responsável por executar todas as operações necessárias para o funcionamento de um dispositivo, possuindo quatro funções básicas: processamento de dados, armazenamento de dados, movimentação de dados e controle (STALLINGS, 2017). Com o aumento da demanda por tecnologias mais avançadas e recursos mais sofisticados, a importância do processador tem aumentado significativamente nas últimas décadas. A busca por processadores mais rápidos, eficientes e poderosos tem impulsionado a evolução da tecnologia de processadores, levando a avanços notáveis na capacidade de processamento, eficiência energética e desempenho gráfico.

Vale citar a explosão no uso de microprocessadores descrita no livro (HENNESY; PATTERSON, 2019):

... a melhoria contínua da fabricação de semicondutores, como previsto pela Lei de Moore, levou à dominância de computadores baseados em microprocessadores por toda a gama de projetos de computador. Os minicomputadores, que tradicionalmente eram feitos a partir de lógica pronta ou de gate arrays, foram substituídos por servidores montados com microprocessadores. Os mainframes foram praticamente substituídos por um pequeno número de microprocessadores encapsulados. Até mesmo os mainframes e os supercomputadores de ponta estão sendo montados com grupos de microprocessadores. Essas inovações de hardware levaram ao renascimento do projeto de computadores, que enfatizou tanto a inovação arquitetônica quanto o uso eficiente das melhorias da tecnologia. Essa taxa de crescimento foi aumentada de modo que, em 2003, os microprocessadores de alto desempenho eram cerca de 7,5 vezes mais rápidos do que teriam alcançado contando-se apenas com a tecnologia, incluindo a melhoria do projeto do circuito. Ou seja, 52% ao ano versus 35% ao ano.

Sob essa ótica, é necessário que profissionais especializados saibam cada vez mais a respeito da arquitetura e organização de um processador, para que possam projetá-los ou prestarem serviços de manutenção. Assim, na disciplina de Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores, o principal objetivo será projetar, desde a arquitetura básica, até a implementação efetiva de um processador com o funcionamento de instruções essenciais da arquitetura Mips.

Em suma, neste relatório será demonstrado os passos realizados para a implementação deste processador, desde o processo de busca de referencial teórico para escolher e planejar a melhor arquitetura, até o desenvolvimento prático utilizando a linguagem de descrição de *Hardware Verilog*. Ao final, será mostrado o projeto final, com os resultados obtidos pelo funcionamento correto de algumas linhas de instruções a serem escolhidas pelo usuário.

2 Objetivos

2.1 Geral

Este projeto tem como objetivo principal aplicar os conceitos teóricos aprendidos na disciplina de Arquitetura e Organização de Computadores e no Laboratório de Circuitos Digitais para desenvolver um processador, escolhendo previamente uma arquitetura adequada. Será necessário definir o conjunto de instruções e o caminho de dados, e por fim, implementar o processador utilizando uma linguagem de descrição de *hardware*. Para testar o processador, será utilizada a plataforma FPGA Altera.

2.2 Específico

Nesta seção, serão numerados cada um dos objetivos propostos para esse projeto:

1. Analisar e comparar diferentes arquiteturas disponíveis, a fim de selecionar a opção mais adequada para servir como base na implementação do processador.
2. Definir as instruções que deverão ser implementadas e, a partir disso, definir como serão os seus formatos a serem decodificados pelo processador.
3. Definir quais serão os modos de endereçamento que serão utilizados pelas instruções durante o funcionamento do processador.
4. Construir um diagrama contendo um *Datapath* que mostre o caminho realizado pelas instruções. Ele deverá ser capaz de indicar isso para todas as instruções escolhidas para compor o projeto final.
5. Implementar, utilizando a linguagem de descrição de *hardware*, a Unidade Lógica e Aritmética e o Banco de Registradores que consigam efetuar as instruções escolhidas anteriormente. Esse processador deverá ser projetado para ser funcional na plataforma FPGA Altera DE2-115.

3 Fundamentação Teórica

3.1 Arquiteturas CISC e RISC

De maneira geral, um processador (CPU - Unidade Central de Processamento) é um pequeno circuito integrado que implementa funções fundamentais como aritmética, lógica, desvios e transferência de dados (HENNESSY; PATTERSON, 2019). Embora essas funções sejam comuns a todos os processadores, sua organização e arquitetura podem variar, resultando em diferentes desempenhos e eficiências energéticas.

Antes de iniciar a construção do processador na linguagem Verilog, é preciso discutir as duas arquiteturas de processadores mais importantes: RISC (*Reduced Instruction Set Computer*) e CISC (*Complex Instruction Set Computer*). Cada uma delas possui vantagens e desvantagens em cenários específicos, as quais devem ser cuidadosamente analisadas para uma escolha mais apropriada.

3.1.1 Arquitetura CISC

O objetivo da arquitetura CISC é trazer um conjunto de instruções complexas, em que uma única instrução é capaz de realizar diversas ações diferentes em um único ciclo de clock, como realizar operações de carregamento de memória, salvar em memória e realizar operações aritméticas e lógicas. Dessa forma, obtemos um conjunto grande de instruções capazes de realizar todas as operações necessárias. Essa arquitetura foi desenvolvida para facilitar a programação, permitindo que os programadores escrevessem códigos em linguagem de montagem mais simples e expressivos, sem a necessidade de realizar tantas operações de baixo nível manualmente. Diminuindo assim, o tamanho dos códigos e facilitando sua escrita em linguagem de máquina.

Os processadores CISC também possuem uma grande quantidade de modos de endereçamento, permitindo o acesso direto a uma ampla variedade de tipos de dados, o que é útil para trabalhar com instruções mais complexas. No entanto, devido à grande quantidade de instruções e modos de endereçamento, os processadores CISC podem ser mais lentos e menos eficientes em relação aos processadores com arquiteturas mais simples, como a RISC (*Reduced Instruction Set Computer*), que utilizam um conjunto de instruções reduzido e mais simples.

Por fim, é importante ressaltar que embora os processadores CISC tenham instruções mais específicas, seu desenvolvimento pode ser custoso e a implementação pode ser difícil e demorada. Portanto, é necessário avaliar cuidadosamente as necessidades do projeto e considerar outras arquiteturas de processadores antes de escolher a arquitetura CISC como

a melhor opção.

3.1.2 Arquitetura RISC

A arquitetura RISC (Reduced Instruction Set Computer) é um tipo de arquitetura de processador que tem como princípio a redução do número de instruções disponíveis para execução. Nessa arquitetura, cada instrução é projetada para executar apenas uma operação simples, de forma que o conjunto de instruções é menor e mais simples do que o conjunto de instruções de uma arquitetura CISC (Complex Instruction Set Computer). Por essa razão, elas são criadas para finalizar em um único ciclo. Essa simplificação tornou os processadores baseados nessa arquitetura simples de criar e mais baratos para produzir. Além de executar as tarefas com maior rapidez e eficiência.

Ademais, a simplificação das instruções também permite uma maior facilidade de implementação em hardware. Outra característica importante da arquitetura RISC é o uso de registradores como o principal meio de armazenamento de dados, em detrimento do acesso direto à memória. Isso aumenta a velocidade de acesso aos dados, já que os registradores estão localizados dentro do processador e podem ser acessados com maior rapidez do que a memória principal.

Em geral, a arquitetura RISC é mais adequada para aplicações que envolvem uma grande quantidade de instruções aritméticas e lógicas, além de operações com vetor e memória. Porém, pode não ser a melhor escolha para aplicações que demandam um grande número de instruções complexas, já que a redução do conjunto de instruções pode tornar a implementação dessas instruções mais difícil e demorada.

Por possuir menos instruções, alguns problemas acabam ocorrendo. Primeiramente, existe uma dificuldade maior de criação de códigos de máquina, já que será preciso utilizar mais instruções para realizar os mesmos procedimentos de uma única instrução em CISC. Ademais, o processador precisará realizar repetidamente as mesmas instruções quando precisar executar um programa que possua uma complexidade elevada. Em terceiro lugar, existe um consumo maior de memória RAM, já que será preciso armazenar as informações antes de utilizá-las pelas demais instruções.

Esses problemas podem ser contornados utilizando um bom compilador, que fará a maior parte do processo de escrita para o código de máquina, sem precisar de um grande esforço do programador. Além disso, um maior armazenamento de memória RAM pode ser o suficiente para evitar que falte espaço para as instruções.

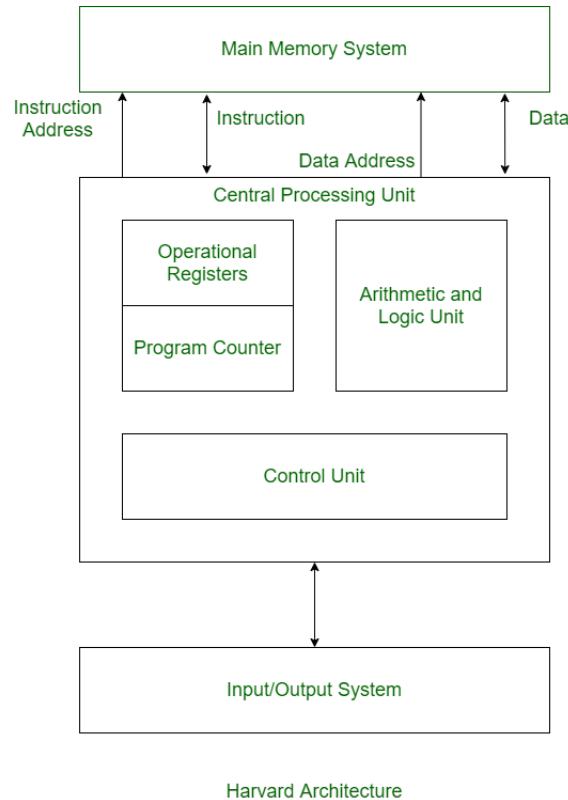
3.2 Arquitetura de Von Neumann e Harvard

As arquiteturas de Harvard e von Neumann são duas das principais arquiteturas de processadores utilizadas na história da computação. A arquitetura de Harvard é caracterizada por ter duas memórias separadas, uma para armazenar instruções (memória de programa) e outra para armazenar dados (memória de dados), enquanto a arquitetura von Neumann utiliza uma única memória para armazenar tanto as instruções quanto os dados. A arquitetura de Harvard se destaca pela velocidade e eficiência no acesso à memória, enquanto a arquitetura von Neumann é mais simples e mais fácil de implementar, sendo utilizada em computadores pessoais e servidores.

John Von Neumann foi o criador da arquitetura von Neumann, que possui apenas um único módulo de memória, em que são armazenados os dados e as instruções do programa. Isso significa que o processador precisa buscar alternadamente as instruções e os dados na mesma memória. Essa arquitetura é mais simples e mais fácil de implementar, porém apresenta o problema fundamental de interferência entre instruções e dados que utilizam o mesmo fio, exigindo pelo menos dois ciclos de clock para a transmissão.

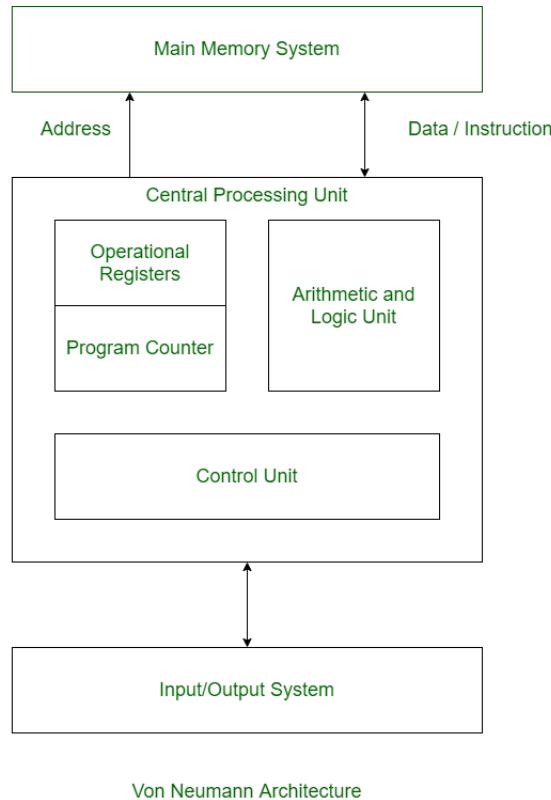
Para evitar esse problema e aumentar a eficiência, surgiu a arquitetura de Harvard, que implementa uma memória extra, utilizada apenas para armazenar as instruções do processador. Com essa separação, temos cabos diferentes para cada uma das memórias, permitindo a leitura das duas memórias em um mesmo ciclo. Consequentemente, as instruções de um processador com implementação de Harvard conseguem ser efetuadas em apenas um único ciclo de clock. Essa arquitetura é usada principalmente em microcontroladores, sistemas embarcados e processadores de sinais digitais. A [Figura 1](#) e a [Figura 2](#) mostram diagramas com as diferenças indicadas anteriormente.

Figura 1 – Arquitetura de Harvard



Fonte: ([ACERVO LIMA,](#))

Figura 2 – Arquitetura de Von Neumann



Fonte: ([ACERVO LIMA,](#))

3.3 Processador MIPS

A arquitetura MIPS (*Microprocessor without Interlocked Pipelined Stages*) é uma arquitetura RISC (Reduced Instruction Set Computer) desenvolvida pela MIPS Technologies em 1980. Ela é utilizada em uma ampla variedade de dispositivos, como roteadores, consoles de videogames, equipamentos médicos e dispositivos embarcados. Além de serem encontrados em produtos da ATI Technologies, Broadcom, Cisco, NEC, Nintendo, Silicon Graphics, Sony, Toshiba, entre outros ([PATTERSON; HENNESSY, 2005](#)).

A arquitetura MIPS é baseada em um conjunto de instruções reduzido, em que todas as instruções são executadas em um ciclo de clock. Isso significa que o processador pode executar uma instrução por ciclo, sem a necessidade de aguardar a conclusão de outras instruções. Isso torna a arquitetura MIPS muito rápida e eficiente.

Outra característica importante da arquitetura MIPS é o uso de pipeline, que é uma técnica de processamento em que as instruções são divididas em várias etapas, permitindo que várias instruções sejam executadas ao mesmo tempo. Cada etapa do pipeline é executada em um ciclo de clock, e as instruções são processadas em paralelo.

Isso aumenta ainda mais a velocidade do processamento.

A arquitetura MIPS também utiliza uma técnica chamada de atraso de slot, em que a instrução seguinte é carregada na fase de busca, mesmo que a instrução atual ainda não tenha sido executada completamente. Isso permite que o pipeline seja preenchido com instruções, aumentando a eficiência do processamento.

Além disso, a arquitetura MIPS possui um conjunto de registradores de propósito geral (32 no total), que são utilizados para armazenar dados temporários durante a execução das instruções. Os registradores são muito rápidos e são acessados diretamente pelo processador, sem a necessidade de acessar a memória principal.

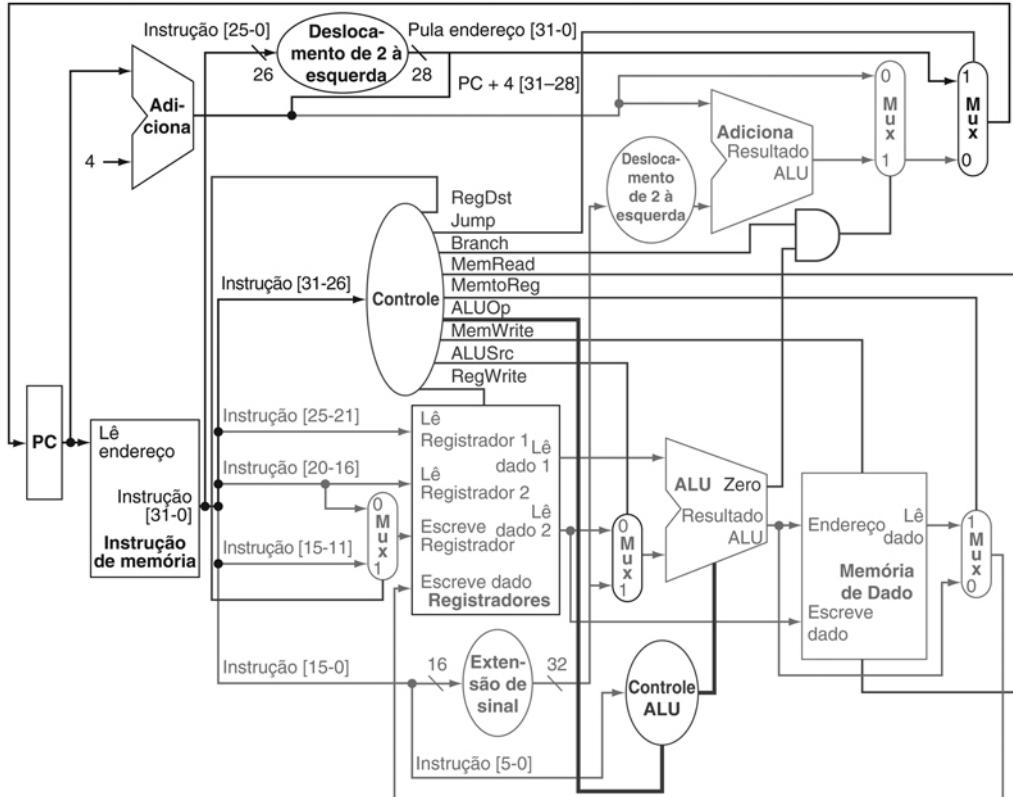
"Os registradores são como tijolos da construção do computador: os registradores são primitivas e usadas no projeto do hardware que também são visíveis ao programador quando o computador é completado. O tamanho de um registrador na arquitetura MIPS é de 32 Bits; os grupos de 32 bits ocorrem com tanta frequência que recebem o nome de *word* na arquitetura."(PATTERSON; HENNESY, 2005)

Em resumo, a arquitetura MIPS é uma arquitetura RISC que utiliza um conjunto de instruções reduzido e pipeline para aumentar a velocidade e eficiência do processamento. Ela também utiliza a técnica de atraso de slot e possui um conjunto de registradores de propósito geral para armazenar dados temporários. A arquitetura MIPS é amplamente utilizada em dispositivos embarcados e de rede devido à sua alta eficiência e velocidade.

3.3.1 Arquitetura Básica

O funcionamento adequado de um processador MIPS depende de diversas estruturas cruciais. Dentre elas, podemos destacar o registrador PC, o banco de registradores, a memória de instrução, a memória principal e a unidade lógica e aritmética (PATTERSON, 2017). Ademais, a arquitetura MIPS utiliza a abordagem de Harvard, a qual se caracteriza pela separação das memórias de dados e de instruções em duas unidades distintas. Um exemplo de arquitetura MIPS pode ser visualizado na figura [Figura 3](#), onde é possível observar um caminho de dados para instruções básicas.

Figura 3 – Arquitetura e Datapath para o MIPS



Fonte: ([PATTERSON, 2017](#))

3.3.2 Formato de Instruções

A fim de transmitir uma instrução para o processador, é imprescindível o uso de uma sequência de sinais eletrônicos que são codificados em binário por meio dos números 1 e 0. Cada algarismo binário é denominado de bit, os quais se juntam para formar as instruções que o processador irá executar. As instruções são segmentadas em campos distintos que possuem informações relevantes. Os conteúdos desses campos diferem conforme o tipo de instrução, o que acarreta em uma influência geral no funcionamento do que está sendo processado.

O MIPS é uma arquitetura RISC de 32 bits, portanto todas as instruções têm o mesmo tamanho em bits, como mencionado anteriormente. Existem três categorias de instruções no MIPS: registrador (R), imediato (I) e de desvio incondicional (J). Cada tipo de instrução é segmentado em campos que guardam informações distintas.

Na [Tabela 1](#) é possível verificar as segmentações para o formato para o tipo R.

Tabela 1 – Formato de Instruções do tipo R para o processador MIPS

Tamanho (bits)	6	5	5	5	5	6
Campo	OPcode	Reg1 (rs)	Reg2 (rt)	Reg3 (rd)	Shamt	Funct
Bits	31 - 26	25 - 21	20 - 16	15 - 11	10 - 6	5 - 0

Fonte: O Autor

O campo opcode indica ao processador a operação básica da instrução, segmento que será importante para os três tipos de instruções. Composto pelos primeiros 6 bits da instrução, ele assume um dos 64 possíveis valores, cada um correspondendo a uma operação específica. Essa informação irá para a unidade de controle, em que irá selecionar quais funcionalidades deverão ser ativadas ou desativadas para que a instrução seja efetuada corretamente.

Em conjunto com os bits restantes da instrução, o Opcode é utilizado para determinar os registradores, memórias e constantes que serão usados na operação, o que o torna um elemento fundamental na flexibilidade e versatilidade da arquitetura MIPS. Além disso, o campo “funct” também servirá para o propósito de controle, porém voltado apenas para a unidade lógica e aritmética, escolhendo qual operação deverá ser realizada.

Ademais, temos três campos para armazenar o endereço de um dos registradores, sendo eles o primeiro registrador de origem (rs), o segundo registrador de origem (rt) e o registrador de destino (rd). Então, ao realizar uma operação entre “rs” e “rt”, o valor final será armazenado em “rd”.

Por fim, temos o campo “shamt”, que possui o propósito de realizar instruções que envolvam deslocar valores de *bits*. Esse deslocamento pode ser realizado para a esquerda ou para a direita, dependendo de qual função será utilizada.

Tabela 2 – Formato de Instruções do tipo I para o processador MIPS

Tamanho (bits)	6	5	5	16
Campo	OPcode	Reg1 (rs)	Reg2 (rt)	Imediato
Bits	31 - 26	25 - 21	20 - 16	15 - 0

Fonte: O Autor

Na tabela [Tabela 2](#), é apresentado o formato de instrução para o tipo I. Observa-se que as principais diferenças em relação ao tipo R estão nos campos "rd", "shamt" e "funct", que foram modificados para acomodar um imediato de 16 bits. Esse imediato pode ser utilizado para diferentes finalidades, como fornecer um endereço de memória ou um valor para realizar uma operação aritmética.

Em algumas instruções do tipo I, é necessário acessar ou armazenar dados em registradores. Nesse caso, o registrador "rs" é utilizado para a leitura das informações e o registrador "rt" é o destino para o armazenamento dos dados.

Tabela 3 – Formato de Instruções do tipo J para o processador MIPS

Tamanho (bits)	6	26
Campo	OPcode	Imediato
Bits	31 - 26	25 - 0

Fonte: O Autor

Por fim, na [Tabela 3](#), temos o formato de instruções para o tipo J. Neste, os únicos campos existentes será o opcode e o imediato de 26bits. Esse imediato será importante para armazenar o endereço de uma instrução na memória de instruções, permitindo que seja possível realizar desvios incondicionais.

3.3.3 Modos de Endereçamento

Os modos de endereçamento é como são chamados os meios de acesso aos operandos das instruções que são salvos na memória do processador, seja na memória principal, de instrução ou o banco de registradores ([PATTERSON; HENNESSY, 2005](#)). Em uma arquitetura MIPS, existem apenas cinco modos de endereçamentos. A seguir, será explicado cada um desses modos.

3.3.3.1 A Registrador

Nesse modo, a operação é executada diretamente em um ou mais registradores, sem necessidade de acessar a memória. É possível observar que na instrução do tipo R ([Tabela 1](#)) será passado o endereço de três registradores (rs, rt e rd). Já nas instruções do tipo I ([Tabela 2](#)) temos apenas dois endereços (rs e rt).

3.3.3.2 Endereçamento Imediato

Nesse modo, o endereço da memória ou valor de registro é especificado diretamente na instrução por um valor imediato, com um valor máximo de 16bits. O único tipo de instrução a utilizar esse modo será a do tipo I ([Tabela 2](#)), que possuí um campo justamente para esse valor.

3.3.3.3 Base-Deslocamento

Nesse modo, o endereço da memória é calculado pela soma de um valor base (registrador) e um valor deslocamento (imediato), especificados na instrução. É um modo de endereçamento utilizado por instruções de *load* e *store*, que são do tipo I ([Tabela 2](#)), em que o valor armazenado em “rs” será o valor base e o imediato o valor do deslocamento.

3.3.3.4 Relativo ao PC

Diferentemente dos modos destacados acima, este próximo modo de endereçamento está relacionado a memória de instruções, ou seja, ele irá influenciar em qual instrução será a próxima a ser executada. Assim, o valor do registrador PC, que aponta para o endereço da instrução atual, será somado a um imediato de 16 *bits*, fornecendo um novo valor de endereço, utilizado para escolher a próxima instrução. Este tipo de endereçamento é utilizado pelo tipo I ([Tabela 2](#)).

3.3.3.5 Absoluto

Para finalizar, o último modo, da mesma forma que o relativo ao PC, também influencia apenas na memória de instrução. Neste caso, o valor do endereço é totalmente passado pelo imediato de 26 *bits*. Dizemos que ele é absoluto por ser o maior valor possível que podemos passar por meio de uma instrução, já que os demais 6 *bits* são destinados ao opcode. Dessa forma, para indicar isso, ele também pode ser chamado de pseudo-absoluto. O único tipo de instrução capaz de realizar esse modo de endereçamento é a do tipo J ([Tabela 3](#)).

3.4 Linguagem de Descrição de Hardware

Uma Linguagem de Descrição de Hardware (HDL - Hardware Description Language) é uma linguagem de programação utilizada para modelar circuitos eletrônicos. Ela é usada para descrever o comportamento e a funcionalidade de sistemas digitais, incluindo processadores, circuitos integrados e outros componentes eletrônicos.

Ao contrário das linguagens de programação convencionais, que são utilizadas para programar software, as HDLs são usadas para descrever hardware e, portanto, são usadas para modelar componentes eletrônicos que são fisicamente construídos.

Existem várias HDLs disponíveis, sendo que as mais comuns são Verilog e VHDL. Essas linguagens permitem que o designer descreva o comportamento de um sistema digital em um nível alto de abstração, em termos de sinais elétricos, em vez de em termos de operações matemáticas ou lógicas. A linguagem Verilog mais especificamente, pode ser compilada em um programa como o Quartus Prime e o arquivo gerado pode ser utilizado em placas especializadas, como o kit FPGA. Dessa forma, ao invés de utilizar as próprias portas lógicas para desenvolver algum esboço do *hardware*, que pode tornar-se muito complexo, é utilizado essa linguagem para simplificar essa implementação.

As HDLs são importantes para o projeto de sistemas digitais complexos, pois permitem que o designer crie modelos abstratos de circuitos eletrônicos que podem ser simulados e testados antes de serem construídos fisicamente. Além disso, as HDLs são

usadas em conjunto com ferramentas de síntese de hardware para gerar automaticamente o layout físico do circuito eletrônico, o que ajuda a acelerar o processo de design e reduzir o tempo e custo de produção.

Em resumo, uma HDL é uma linguagem de programação usada para modelar circuitos eletrônicos, permitindo que o designer crie modelos abstratos que possam ser simulados e testados antes de serem fisicamente construídos.

3.5 Field Programmable Gate Array (FPGA)

FPGA é a sigla em inglês para *Field Programmable Gate Array*, que em português pode ser traduzido como "matriz de portas programável em campo". É um tipo de dispositivo semicondutor que pode ser programado e reprogramado depois de fabricado para realizar tarefas específicas em sistemas digitais.

Diferentemente de processadores convencionais, como os processadores de computadores pessoais, que são projetados para realizar uma ampla gama de tarefas, os FPGAs são projetados para serem altamente especializados em tarefas específicas. Isso significa que os FPGAs podem ser otimizados para realizar uma tarefa específica com muita eficiência.

A capacidade do FPGA de traduzir e implementar códigos escritos em linguagem de hardware é crucial para o desenvolvimento do projeto em pauta, que envolve a implementação de várias funcionalidades. Durante a disciplina de laboratório de arquitetura e organização de computadores, será utilizado um FPGA da família "Cyclone IV" modelo "EP4CE115F29C7".

Figura 4 – FPGA Imagem

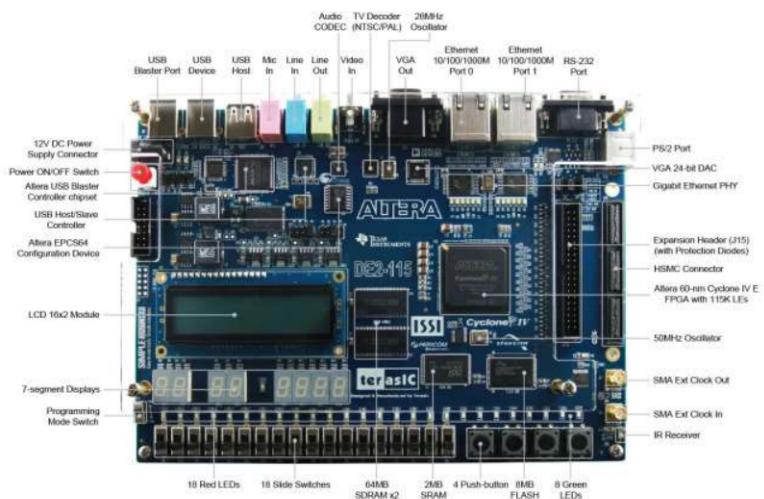


Figure 2-1 The DE2-115 board (top view)

(ALTERA, 2013)

4 Desenvolvimento

A partir do que foi estudado em capítulos anteriores, foi possível definir características básicas para o desenvolvimento do processador, além das arquiteturas que serão usadas para a sua construção utilizando a linguagem de descrição de *hardware*.

4.1 Características do Processador

O processador será descrito por essas seguintes características:

1. O processador irá operar com 32 *bits*. Assim, as instruções, banco de registradores e memória principal possuem essa mesma quantidade de *bits*.
2. Será utilizado um modelo de processador baseado em MIPS. Portanto, o conjunto de instruções serão parecidas com as já implementadas por este modelo, além de possuírem o mesmo formato de instruções.
3. A arquitetura de instruções escolhida será do tipo RISC.
4. O processador será baseado no tipo de arquitetura de memória de Harvard.

4.2 Conjunto de Instruções

Os formatos de instruções a serem utilizados serão os mesmos que compõe a arquitetura MIPS. Esses formatos estão descritos detalhadamente nas [Tabela 1](#), [Tabela 2](#) e [Tabela 3](#). Com os formatos determinados, podemos definir quais as instruções que serão efetuadas pelo processador. Assim, elas foram descritas nas tabelas [4](#), [5](#) e [6](#).

Tabela 4 – Instruções do tipo R

OPcode	Funct	Nome	Função
000000	100000	ADD	$rd \leftarrow rs + rt$
000000	010000	SUB	$rd \leftarrow rs - rt$
000000	100100	AND	$rd \leftarrow rs \& rt$
000000	100101	OR	$rd \leftarrow rs rt$
000000	110100	JR	$PC \leftarrow rs$
000000	001100	JALR	$rd \leftarrow \text{endereço}; PC \leftarrow rs$
000000	101100	SLT	$rd \leftarrow (rs < rt); 0 \text{ se falso; } 1 \text{ se verdadeiro}$
000000	011100	SLET	$rd \leftarrow (rs \leq rt); 0 \text{ se falso; } 1 \text{ se verdadeiro}$
000000	111100	SGT	$rd \leftarrow (rs > rt); 0 \text{ se falso; } 1 \text{ se verdadeiro}$
000000	000010	SGET	$rd \leftarrow (rs < rt); 0 \text{ se falso; } 1 \text{ se verdadeiro}$
000000	000111	NOT	$rd \leftarrow \text{not}(rs)$
000000	000100	NOR	$rd \leftarrow rs \sim rt$
000000	100100	SLL	$rd \leftarrow rt \ll \text{shamt}$
000000	010100	SRL	$rd \leftarrow rt \gg \text{shamt}$
000000	110000	DIV	$rd \leftarrow rs / rt$
000000	001000	MULT	$rd \leftarrow rs * rt$

Fonte: O Autor

Tabela 5 – Instruções do tipo I

OPcode	Nome	Função
100011	LW	$rt \leftarrow \text{Mem}[\text{base} + \text{offset}]$
100011	LWI	$rt \leftarrow \text{Imm}$
101011	SW	$\text{Mem}[\text{base} + \text{offset}] \leftarrow rt$
001000	ADDI	$rt \leftarrow rs + \text{Imm}$
001001	SUBI	$rt \leftarrow rs - \text{Imm}$
001100	ANDI	$rt \leftarrow rs \& \text{Imm}$
001101	ORI	$rt \leftarrow rs \text{Imm}$
000100	BEQ	Se $(rs == rt)$ então branch
000101	BNE	Se $(rs != rt)$ então branch
001010	SLTI	$rt \leftarrow (rs < \text{Imm}); 0 \text{ se falso; } 1 \text{ se verdadeiro}$
001010	JI	$PC \leftarrow \text{Imm}$
001010	Nop	Nenhuma operação
011111	IN	$rt \leftarrow \text{Imm}$
011110	OUT	saída $\leftarrow rs$

Fonte: O Autor

Tabela 6 – Instruções do tipo J

OPcode	Nome	Função
000010	J	Jump para o endereço
000011	JAL	$ra \leftarrow PC + 1; \text{Jump para o endereço}$
111111	HALT	Parar o processador

Fonte: O Autor

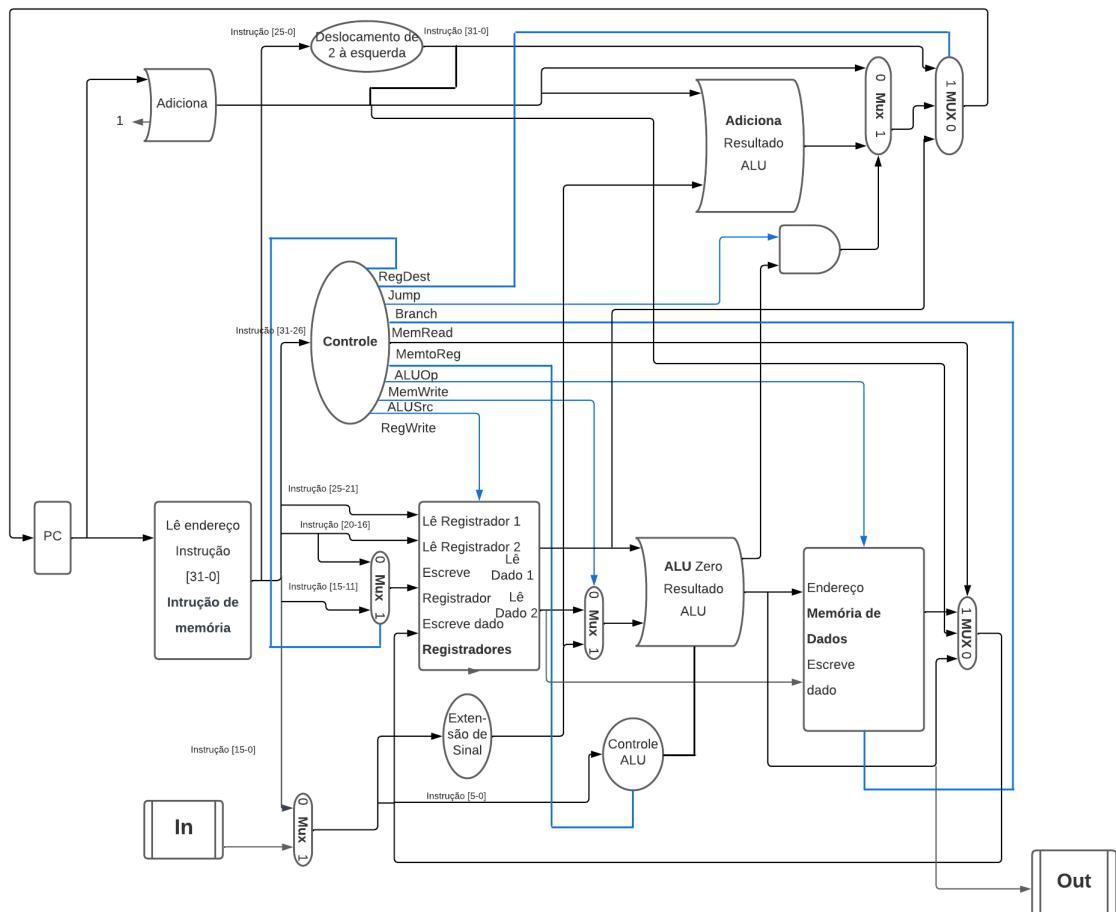
4.3 Modos de Endereçamento

Baseado nas instruções que serão implementadas na seção anterior, serão utilizados os mesmos cinco modos de endereçamento da própria arquitetura MIPS, que já foram discutidos na seção de referencial teórico.

4.4 Caminho de Dados

Para desenvolver as instruções indicadas anteriormente, será preciso verificar quais serão as etapas e os componentes utilizados do processador quando for preciso executá-las. A construção de um *Datapath* facilita a visualização do caminho efetuado por cada instrução, permitindo uma facilidade maior para construir o processador. Na Figura 5 está indicado o caminho de dados para todas as instruções que serão implementadas.

Figura 5 – Caminho de dados para o processador a ser desenvolvido.



Fonte: O Autor

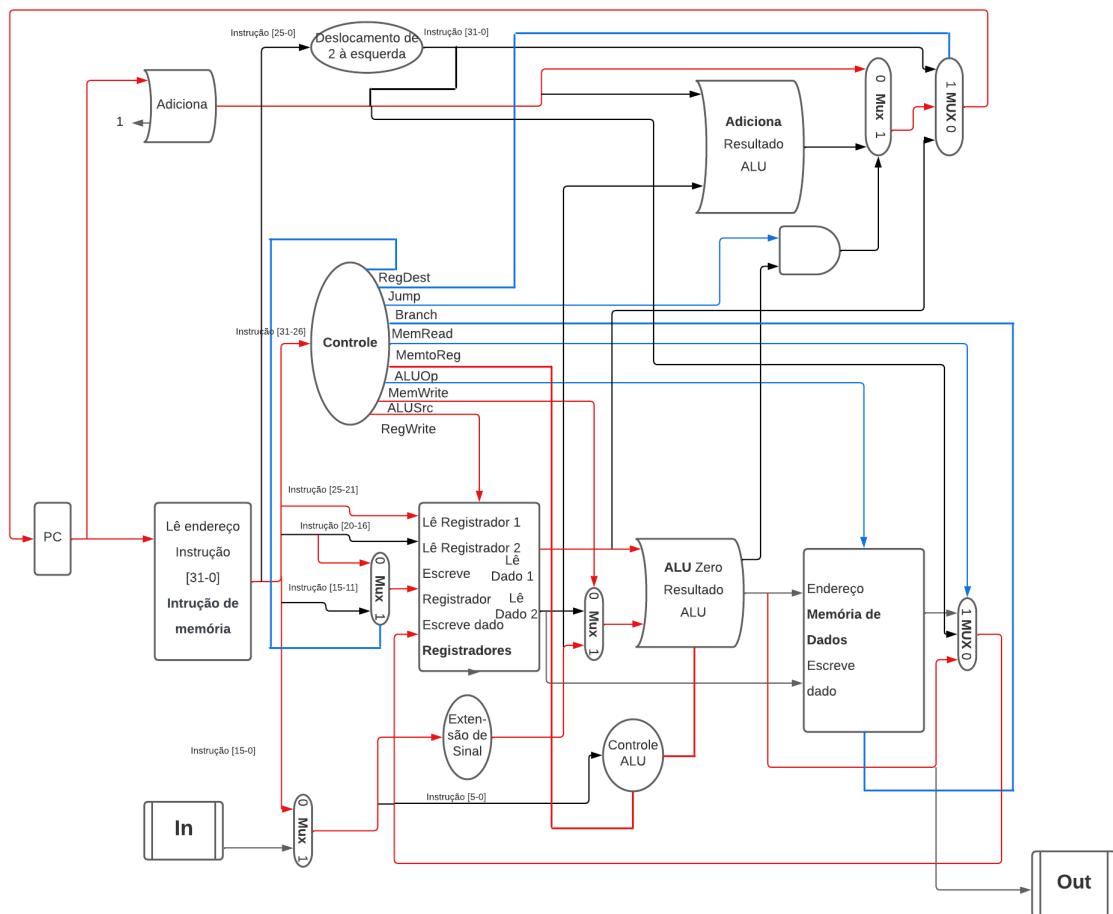
No diagrama do processador, alguns componentes têm maior relevância para o projeto. A memória de instrução, por exemplo, tem apenas uma entrada que recebe o

valor do registrador PC, indicando qual instrução será enviada aos demais segmentos. A saída dessa memória será dividida em vários segmentos, cada um correspondente a um dos campos dos tipos de instruções.

O banco de registradores é uma parte fundamental de um processador e possui cinco entradas. Essas entradas incluem: o endereço do primeiro registrador de entrada (rs), o endereço do segundo registrador de entrada (rt), o endereço do registrador de destino (rd ou rt), o dado a ser escrito em um dos registradores e um bit de controle para ativar ou desativar a escrita no banco. As saídas correspondem aos valores armazenados nos registradores indicados pelos endereços de entrada.

Os valores de saída do banco de registradores são enviados diretamente para a Unidade Lógica e Aritmética (ULA). Esses valores são utilizados como operandos para realizar operações lógicas e aritméticas. Além disso, o valor da segunda entrada também pode ser o valor de um imediato de 16 bits extraído de uma instrução do tipo I como pode ser visto em [Figura 6](#). Para compatibilidade com a ULA, esse valor deve ser estendido para 32 bits antes de ser utilizado nas operações.

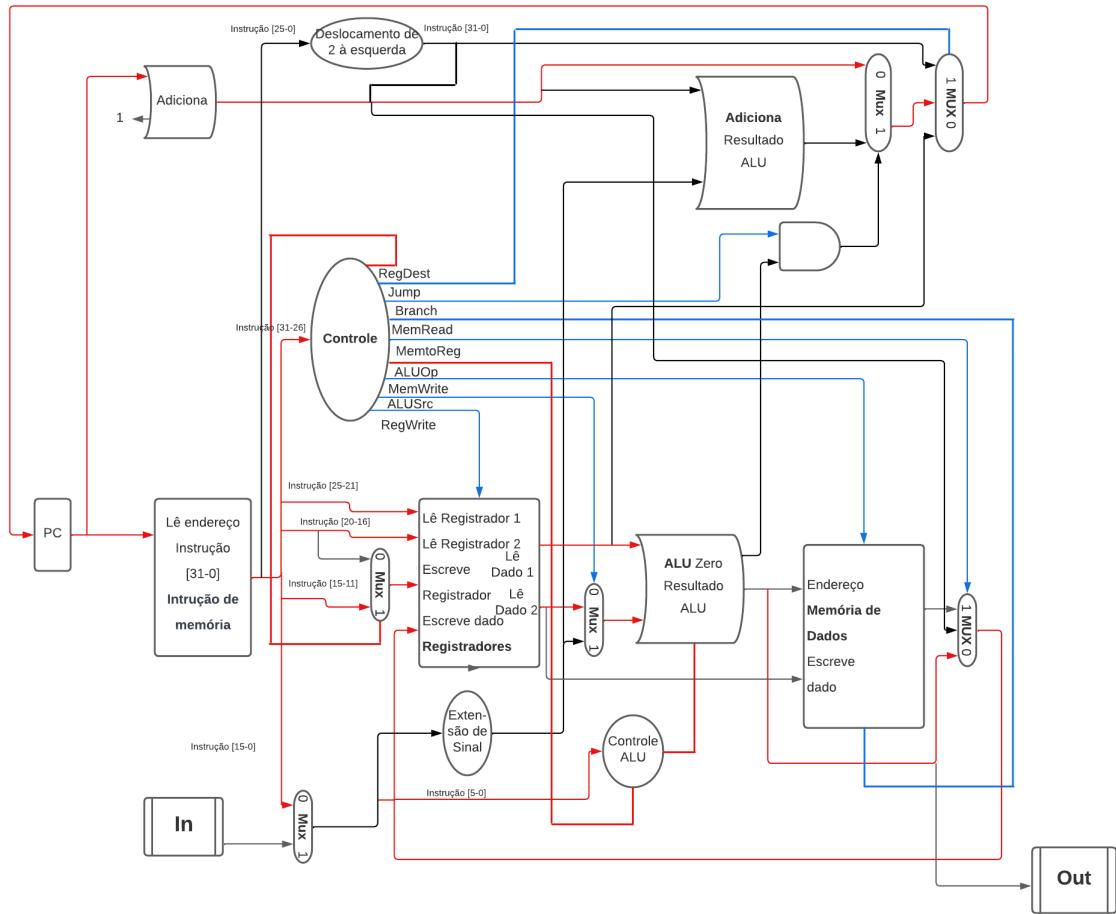
Figura 6 – Caminho de dados tipo I



Fonte: O Autor

Além das entradas mencionadas anteriormente, existem outras duas entradas importantes no processador. Uma delas é o valor de "shamt", que representa o deslocamento na instrução do tipo R como pode ser visto em [Figura 7](#). A outra entrada é constituída pelos bits de controle que indicam qual instrução será executada. Esses bits são essenciais para o correto funcionamento do processador e garantem que a operação adequada seja realizada.

Figura 7 – Caminho de dados tipo R

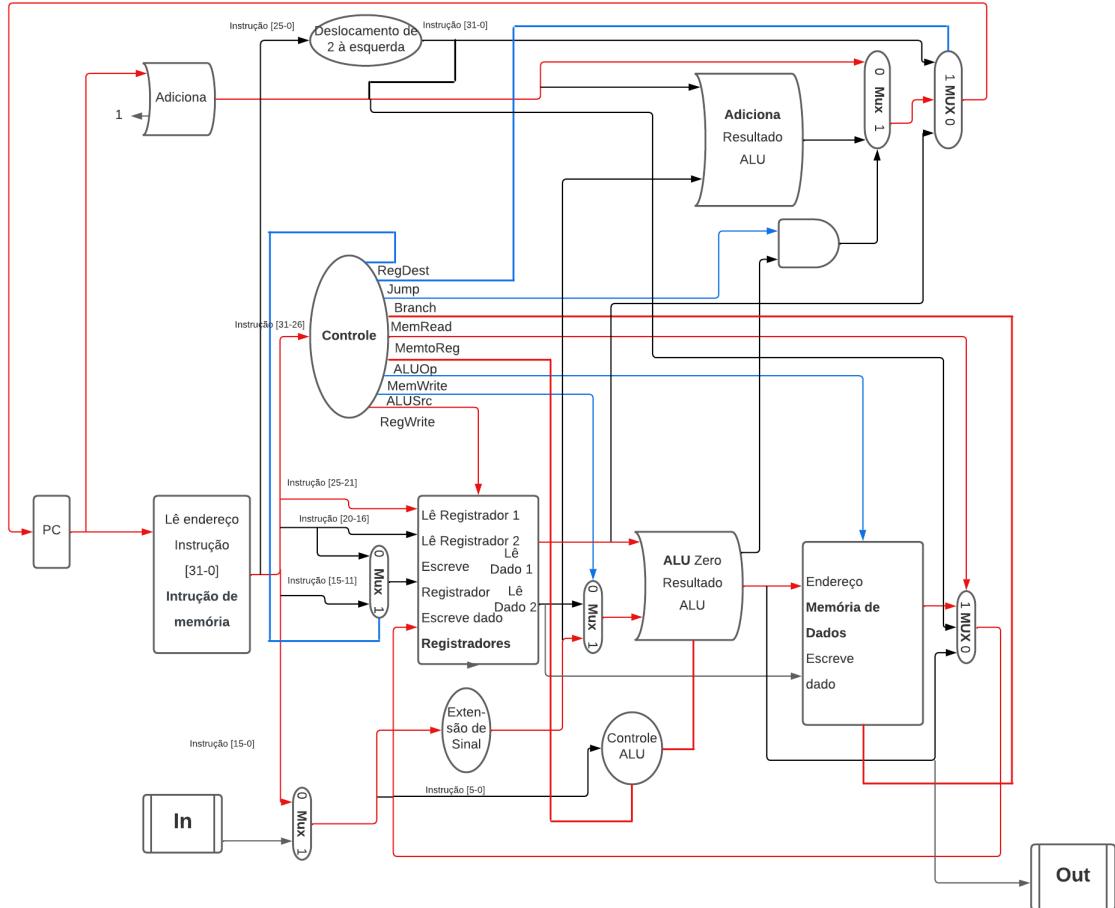


Fonte: O Autor

As saídas da ULA incluem o resultado da operação realizada e um bit que indica se o resultado é igual a zero. Essa informação é útil para verificar condições de igualdade ou tomar decisões com base no resultado obtido.

No contexto do processador, a memória principal também desempenha um papel importante. Ela recebe como entrada o resultado da ULA, que é utilizado como endereço para ler ou armazenar dados (como por exemplo a *Load Word* em [Figura 8](#)). Além disso, a memória recebe o valor do dado a ser armazenado e dois bits de controle, um para escrita e outro para leitura dos dados. Em resposta, a memória principal emite o valor do dado desejado, que será armazenado no banco de registradores para uso futuro.

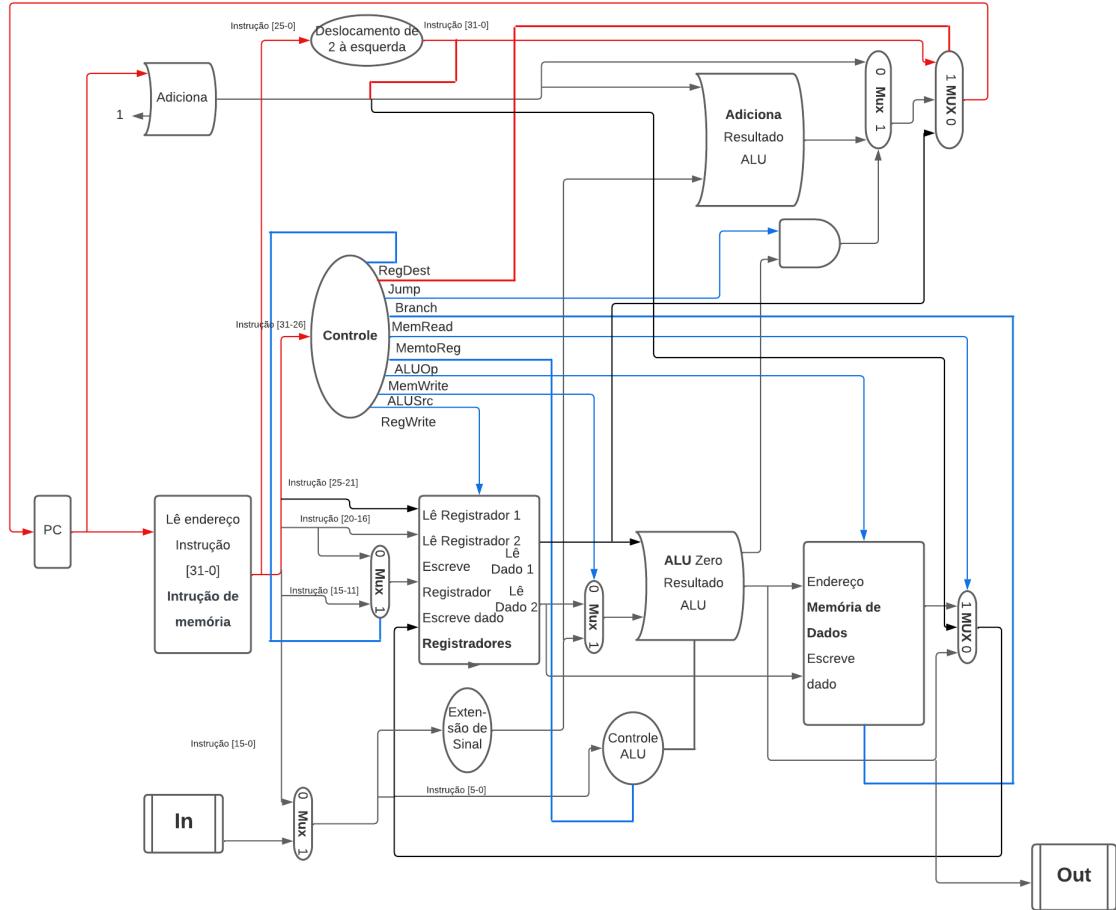
Figura 8 – Caminho de dados tipo LW



Fonte: O Autor

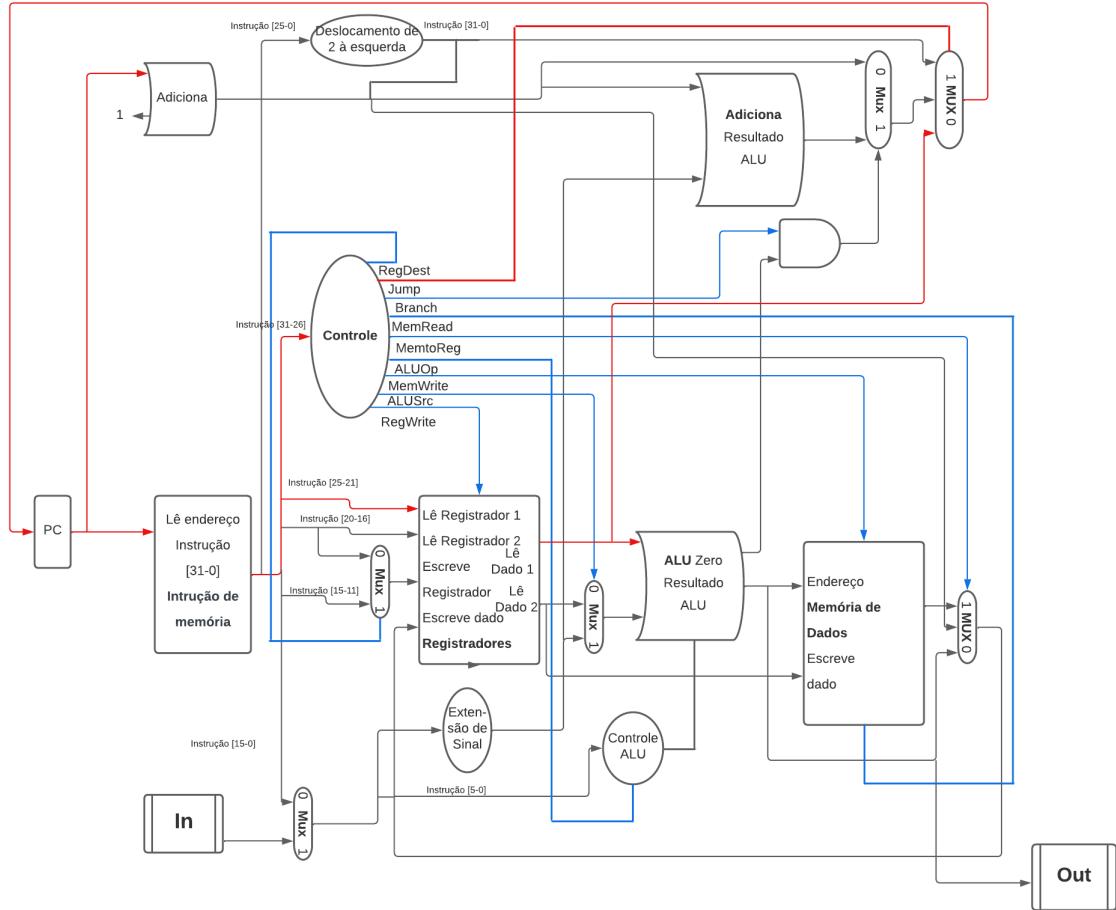
Ao executar as instruções, o fluxo de operações geralmente segue uma sequência, conforme as instruções estão dispostas na memória do programa. No entanto, em caso de instruções de salto condicional ou incondicional (estes podem ser vistos em), o valor do endereço obtido é direcionado para o registrador Program Counter (PC), que é atualizado para alterar a ordem de execução das instruções no processador projetado. Isso permite que desvios e saltos sejam realizados, permitindo maior flexibilidade na execução do programa.

Figura 9 – Caminho de dados tipo J



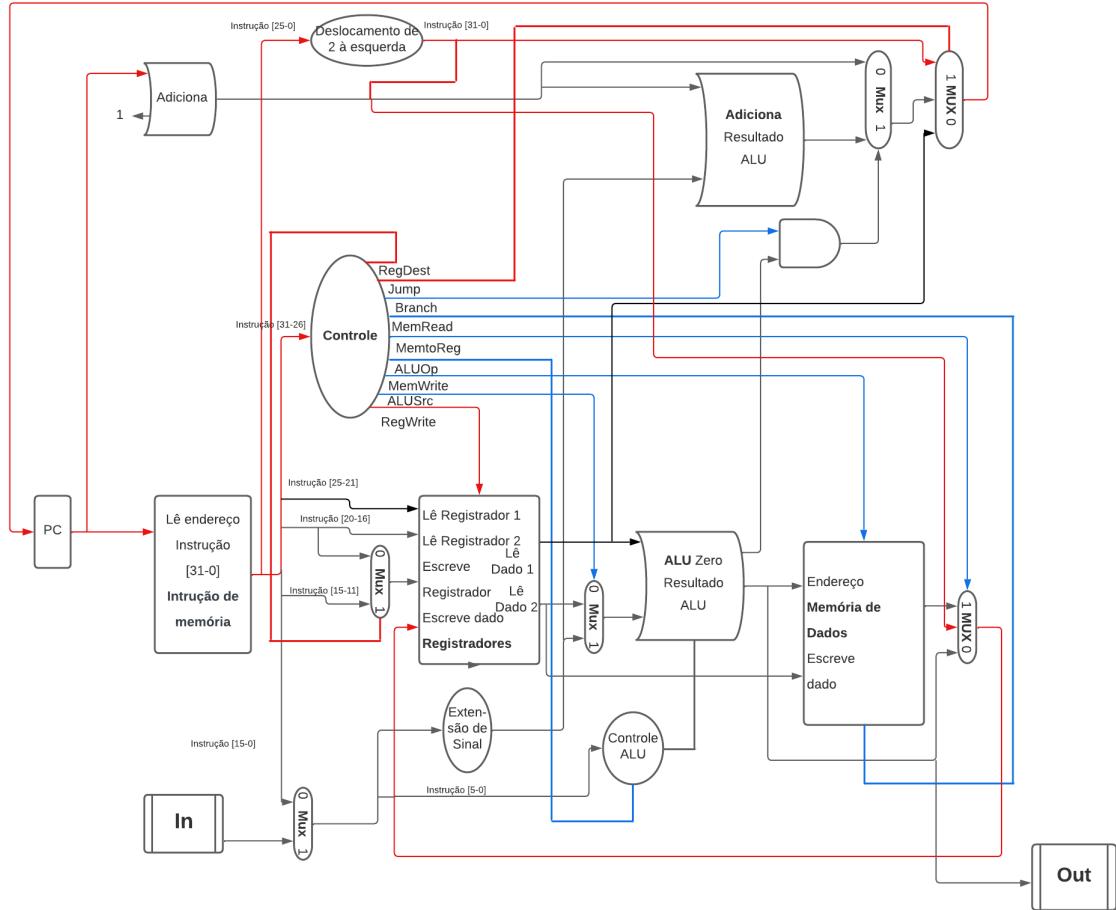
Fonte: O Autor

Figura 10 – Caminho de dados tipo JR



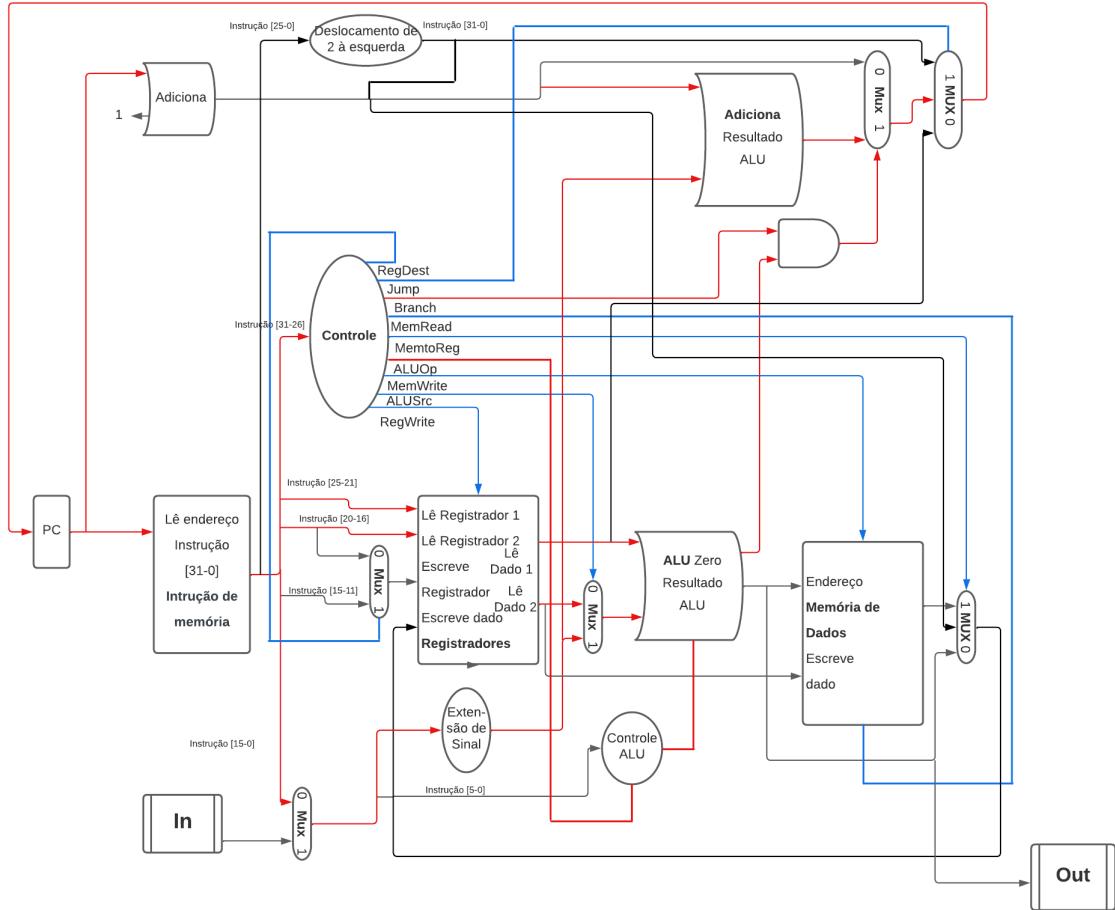
Fonte: O Autor

Figura 11 – Caminho de dados tipo JAL



Fonte: O Autor

Figura 12 – Caminho de dados tipo Branch



Fonte: O Autor

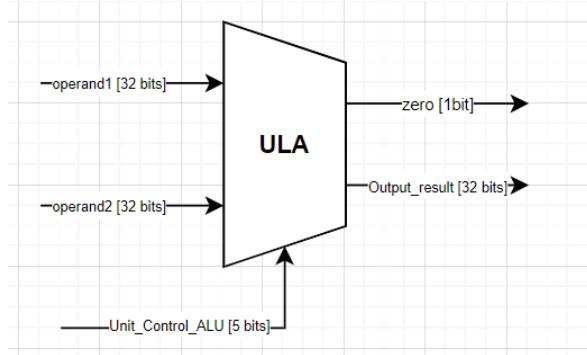
4.5 Implementação do Projeto

Utilizando da linguagem de descrição de hardware Verilog foi implementado os módulos funcionais segundo o caminho de dados da Figura 5 e especificações da arquitetura escolhida.

4.5.1 Unidade Lógica e Aritmética (ULA)

A ULA (Unidade Lógica e Aritmética) é a unidade funcional que irá realizar a maioria das operações do processador, sendo operações de como adição, subtração, multiplicação, divisão, deslocamento de bits, operações lógicas e comparação. Ela opera com palavras de tamanho fixo, no caso da arquitetura Mips em 32 bits; a ULA recebe dados de registradores, realiza as operações solicitadas e, em seguida, armazena o resultado em um registrador de destino ou na memória de dados. As operações que serão executadas pela ULA estão descritas na Tabela 7.

Figura 13 – Unidade Lógica e Aritmética



Fonte: O Autor

Tabela 7 – Operações da ULA, que são ativadas conforme o valor de controle.

Controle	Funcionalidade
0000	AND
0001	OR
0010	ADD
0011	SUB
0100	SLT
0101	SGT
0110	SGET
0110	SLET
1000	MULT
1001	DIV
1010	NOR
1011	Deslocar bits para a Esquerda (SLL)
1100	Deslocar bits para a Direita (SRL)

Fonte: O Autor

Com isso a implementação da ULA em Verilog pode ser verificada na [Figura 14](#).

Figura 14 – Código em verilog para a unidade lógica e artimética

```

Date: June 09, 2023           ULA.v          Project: mips
1  module ULA (
2
3    input[31:0] operand1, operand2,
4    input wire [3:0] Unit_Control_ALU,
5
6    output wire [31:0] Output_Result,
7    output wire zero,
8    output wire Output_1bit
9  );
10
11  reg result_1bit;
12  reg[31:0] result;
13  reg Reg_Zero;
14
15
16  initial begin
17    result_1bit <= 1'd0;
18    result <= 32'd0;
19  end
20
21
22  always @ (operand1 or operand2 or Unit_Control_ALU) begin
23
24    case (Unit_Control_ALU)
25      4'b0000: result = operand1 & operand2; // AND
26      4'b0001: result = operand1 | operand2; // OR
27      4'b0010: result = operand1 + operand2; // ADD
28      4'b0011: result = operand1 - operand2; // SUB
29      4'b0100: begin
30        if (operand1 < operand2) begin // SLT
31          result_1bit = 1'd1;
32        end
33        else begin
34          result_1bit = 1'd0;
35        end
36      end
37      4'b0101: begin
38        if (operand1 > operand2) begin // SGT
39          result_1bit = 1'd1;
40        end
41        else begin
42          result_1bit = 1'd0;
43        end
44      end
45      4'b0110: begin
46        if (operand1 >= operand2) begin // SGET
47          result_1bit = 1'd1;
48        end
49        else begin
50          result_1bit = 1'd0;
51        end
52      end
53      4'b0111: begin
54        if (operand1 <= operand2) begin // SLET
55          result_1bit = 1'd1;
56        end
57        else begin
58          result_1bit = 1'd0;
59        end
60      end
61      4'b1000: result = operand1 * operand2; // MULT
62      4'b1001: result = operand1 / operand2; // DIV
63      4'b1010: result = ~ (operand1 | operand2); // NOR
64      4'b1011: result = operand1 >> operand2; // SLL
65      4'b1100: result = operand1 << operand2; // SRL
66
67    endcase
68
69    if (Output_Result == 32'd0) begin
70      Reg_Zero = 1'b1;
71    end
72    else begin
73      Reg_Zero = 1'b0;
74    end
75
76  end
77

```

Fonte: O Autor

A ULA recebe dois operandos “operand1” e “operand2” cada um sendo um fio contendo 32 Bits de tamanho, além da entrada da Unidade de Controle “Unit_Control_ALU” que irá selecionar a operação a ser executada. O resultado das operações que retornam

valores numéricos serão guardados na variável “result” e as que retornam valores lógicos na “result1_bit”.

Os *Outputs* serão atribuídos de “result” para “Output-Result” contendo 32 *Bits*, e “result1_bit” para “Output_1bit” e por fim “zero” recebe “Reg_Zero” que será usado futuramente para avaliar desvios e fazer comparações.

Figura 15 – Código em verilog para a unidade lógica e artimética

```

end
assign Output_Result = result;
assign Output_1bit = result_1bit;
assign zero = Reg_Zero;
endmodule

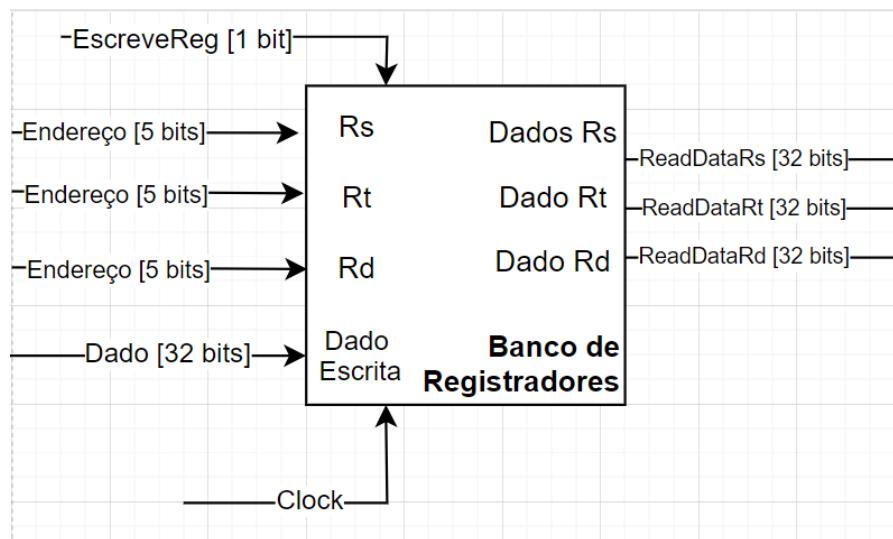
```

Fonte: O Autor

4.5.2 Banco de Registradores

Na arquitetura MIPS, o Banco de Registradores consiste em um conjunto de registradores de propósito geral de 32 bits (no Verilog isso foi implementado por meio de um vetor), numerados de 0 a 31 (índices do vetor). Cada registrador pode armazenar um valor de 32 bits, como números inteiros, endereços de memória ou outros dados. Isso se deve ao fato de que nesta arquitetura não podemos trabalhar com dados diretamente da memória de dados, sendo necessário esse banco para armazenar temporariamente o resultado/endereço das operações.

Figura 16 – Banco de Registradores



Fonte: O Autor

Figura 17 – Código em verilog para o banco de registradores.

```

Date: June 09, 2023           BancodeRegistradores.v          Project: mips
 1  module BancodeRegistradores (
 2
 3    input wire [4:0] ReadRegister1, ReadRegister2, WriteReg,
 4    input wire [31:0] WriteData,
 5    input wire [3:0] Unit_Control_RegWrite,
 6    input clock, writeEnable,
 7    output wire [31:0] ReadDataRD, ReadDataRS, ReadDataRT
 8  );
 9
10  integer First_clock=1;
11
12  reg [31:0] registers [31:0];
13
14
15  always @(posedge clock)
16  begin
17    if (First_clock == 1)
18      begin
19        // Separo os 2 últimos registradores do banco para iniciá-los com valores padrão
20        registers[31] = 32'b00000000000000000000000000000000111111; // Último registrador do
banco com valor 127 para traço (-) no display
21        registers[30] = 32'b00000000000000000000000000000000111111; // valor 126 para display
apagado
22        First_clock <= 2;
23      end
24
25    if (writeEnable)
26      begin
27        registers[WriteReg] = WriteData; // Se a escrita no registrador estiver permitida
pela UC, o dado será escrito no registrador
28                                // que de endereço WriteRegRT
29      end
30    end
31
32  assign ReadDataRS = registers[ReadRegister1];
33  assign ReadDataRD = registers[WriteReg]; // O dado que escrevi no registrador do banco
agora deve ser passado para RD
34  assign ReadDataRT = registers[ReadRegister2];
35
36
37  endmodule
38
39

```

Fonte: O Autor

Com isso temos o “ReadRegister 1” e “ReadRegister2” contendo 5 *Bits* cada, uma vez que 5 *Bits* é suficiente para descrever qual registrador queremos acessar dos 32.

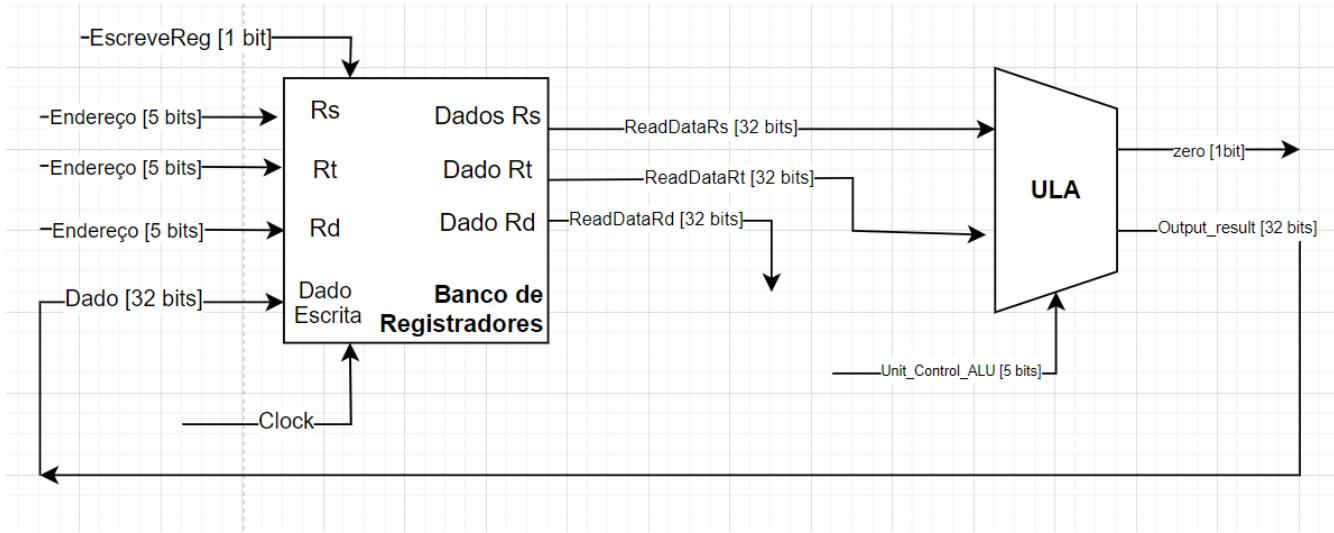
$$2^5 = 32 \quad (4.1)$$

Depois disso, temos o “ReadDataRS” e “ReadDataRT”, o primeiro irá ler o conteúdo do “ReadRegister 1” e o segundo o do “ReadRegister2”. Ademais, temos o “ReadDataRS” que é o registrador em que será escrito um dado de 32 Bits vindo do “WriteData” de endereço “WriteReg” caso a Unidade de Controle permita a escrita no banco de registradores pela variável “WriteEnable”.

4.5.3 Unidade de Processamento

A unidade de processamento [Figura 18](#) consiste na união do Banco de Registradores e da Unidade Lógica e Aritmética em um só módulo. Para tal, foram utilizadas as importações de módulos no verilog e os caminhos de dados intermediários entre as entradas e saídas foram ligados pela função “wire”.

Figura 18 – Unidade de Processamento



Fonte: O Autor

No entanto, no decorrer do projeto este módulo da unidade de processamento foi usado para fazer a integração de todos os módulos funcionais do processador. Uma vez que ele é a base de tudo, foi necessário integrar cada uma das partes com base nele. O resultado final pode ser visto no código abaixo na [Figura 19](#).

Figura 19 – Unidade de Processamento Verilog

```

Date: July 14, 2023           UnidadedeProcessamento.v          Project: mips

1  module UnidadedeProcessamento (
2
3      //input [4:0] RegisterRS, RegisterRT, RegisterRD,
4
5      input clock,
6      input [15:0] interruptores,
7      output [6:0] Display1, Display2, Display3, Display4,
8      output led,
9      output zero,
10     //output [6:0] display,
11     output [31:0] Out_ULA,
12     output [31:0] Inst,
13     output [4:0] auxRS,
14     output [4:0] auxRT,
15     output [31:0] Wave_Saida_RegDST, Wave_Saida_Alusrc, Wave_Saida_MemToReg, Wave_Dado1,
16     output [3:0] Wave_Unit_Control_ALU
17 );
18
19     // Instrucao em binario saindo da Memoria ROM
20     wire [31:0] InstMem;
21
22     // Saída da ULA
23     wire [31:0] auxALUOut;
24     wire auxZero;
25
26     // Saída do dado 1 e dado 2 do banco de registradores
27     wire [31:0] ReadDataRS;
28     wire [31:0] ReadDataRT;
29
30     // Fios dos bits de saída da unidade de controle
31     wire RegDst, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite, PCFunct, BEQ, BNE,
32     ControlJump, Halt, EnableClock, JAL, In, Out;
33     wire [2:0] AluOp;
34
35     wire [31:0] Saída_MemDados; // Saída da memoria RAM
36
37     // Unidade de controle da ULA
38     wire [3:0]Unit_Control_ALU; // Bits para controle da ULA
39     wire JALR, JR; // Bits de controle para os jumps específicos do tipo R
40
41     // MUX
42     wire [4:0] Saída_RegDST;
43     wire [31:0] Saída_Alusrc;
44     wire [31:0] Saída_MemToReg;
45
46     // Modulo de Entrada
47     wire novo_clock;
48     wire [15:0] resultado_entrada;
49
50     integer auxSelec = 0; // Temporário, teste MEM e PC
51     wire [9:0] valorPC; // Índice de saída do PC
52
53     Entrada Chamada11(.clock(clock), .novo_clock(novo_clock), .in(In), .interruptores(
54     interruptores), .resultado_entrada(resultado_entrada));
55
56     UnidadedeControle Chamada6(.Opcode(InstMem[31:26]), .Aluop(AluOp), .RegDst(RegDst), .
57     MemRead(MemRead), .MemtoReg(MemtoReg), .Memwrite(MemWrite),
58     .ALUSrc(ALUSrc), .Regwrite(Regwrite), .PCFunct(PCFunct), .BEQ(BEQ), .BNE(BNE), .
59     ControlJump(ControlJump), .Halt(Halt), .EnableClock(EnableClock),
60     .JAL(JAL), .Out(Out), .In(In));
61
62     UnidadedeControleULA Chamada8(.Funct(InstMem[5:0]), .Aluop(AluOp), .ControleALU(
63     Unit_Control_ALU), .JALR(JALR), .JR(JR));
64
65     PC Chamada5(.clock(novo_clock), .Indice(valorPC), .IndiceAux(InstMem[25:0]), .Selecao(
66     ControlJump));
67
68     MemoriaInstrucoesROM Chamada3(.addr(valorPC), .q(InstMem));
69
70     MuxRegDst Chamada9(.reg1(InstMem[15:11]), .reg2(InstMem[20:16]), .reg_saida(
71     Saída_RegDST), .controle(RegDst));
72
73     BancoDeRegistradores Chamada2(.clock(novo_clock), .ReadRegister1(InstMem[25:21]), .
74     WriteEnable(Regwrite),

```

Figura 20 – Unidade de Processamento Verilog

```

Date: July 14, 2023          UnidadedeProcessamento.v          Project: mips
69      .ReadRegister2(InstMem[20:16]), .WriteReg(Saida_RegDST), .WriteData(Saida_MemToReg), .
70      ReadDataRS(ReadDataRS), .ReadDataRT(ReadDataRT);
71      MuxALUSRC Chamada13 (.Dado2(ReadDataRT), .Imediato(InstMem[15:0]), .controlle(ALUSrc),
72      .Saida_Alusrc(Saida_Alusrc), .interruptores(resultado_entrada), .in(In));
73      ULA Chamada1(.Output_Result(auxALUOut), .operand1(ReadDataRS), .operand2(Saida_Alusrc),
74      .zero(zero), .Unit_Control_ALU(Unit_Control_ALU));
75      MemoriadeDadosRAM Chamda7(.data(ReadDataRT), .addr(auxALUOut), .we(MemWrite), .re(
76      MemRead), .clk(novo_clock), .q(Saida_MemDados));
77      MuxMemReg Chamada10(.Dado_ULA(auxALUOut), .Dado_Mem(Saida_MemDados), .Saida_MemReg(
78      Saida_MemToReg), .controlle(MemtoReg));
79      assign led = novo_clock;
80      assign Out_ULA = auxALUOut;
81      assign Inst = InstMem;
82      assign auxRS = InstMem[25:21];
83      assign auxRT = InstMem[20:16];
84      assign Wave_Saida_RegDST = Saida_RegDST;
85      assign Wave_Saida_Alusrc = Saida_Alusrc;
86      assign Wave_Saida_MemToReg = Saida_MemToReg;
87      assign Wave_Dado1 = ReadDataRS;
88      assign Wave_Unit_Control_ALU = Unit_Control_ALU;
89
90 endmodule
91

```

4.5.4 Memória de Dados (RAM)

A memória principal do computador é chamada de memória RAM (*Random Access Memory*) ou no caso do processador MIPs a Memória de Dados, uma vez que é nela que os dados gerais serão armazenados. Ela possui um tamanho bem maior que a memória de instruções, pois precisa lidar com maiores quantidades de dados sendo armazenados.

Para a criação da memória, foi utilizado um modelo já disponibilizado pela Quartus Prime. Foi preciso apenas modificar os valores "DATA_WIDTH" e "ADDR_WIDTH", que indicam o tamanho do dado em *bits* do dado armazenado e a quantidade de registradores na memória, o modelo pode ser visto na [Figura 21](#).

Figura 21 – Memoria de Dados (RAM)

```
// Quartus Prime Verilog Template
// Single port RAM with single read/write address

module MemoriadeDadosRAM
#(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=10)
(
    input [(DATA_WIDTH-1):0] data,
    input [(ADDR_WIDTH-1):0] addr,
    input we, re, clk,
    output reg [(DATA_WIDTH-1):0] q
);
    // Declare the RAM variable
    reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

    always @ (posedge clk)
    begin
        // Write
        if (we)
            ram[addr] <= data;
    end

    // Continuous assignment implies read returns NEW data.
    // This is the natural behavior of the TriMatrix memory
    // blocks in Single Port mode.
    always @ (negedge clk)
    begin
        if (re)
            q <= ram[addr];
    end
endmodule
```

Fonte: O Autor

O processador projetado terá 32 *bits* de dados para operar, portanto "ADDR_WIDTH" precisa receber 32 como argumento. Já o "ADDR_WIDTH" irá receber 10 como parâmetro, ou seja, temos uma memória RAM de 1024 espaços para informações de 32 *bits* cada.

A escrita na memória só poderá ser efetuada a cada subida de borda do *clock* ("clk"). Da mesma maneira, a leitura será apenas realizada em descidas de *clock*, sendo necessário que o *bit* de controle "we" esteja ativo para que a escrita seja realmente efetivada naquele ciclo e *bit* "re" precisa estar ativo para permitir a leitura.

4.5.5 Memória de Instrução (ROM)

A memória de Instruções (ROM) também será retirada de um modelo do Quartus Prime. Este modelo permite que as instruções a serem executadas pelo processador sejam salvas em um arquivo txt, que irá assumir esse papel de armazenar as informações. Da mesma forma que a memoria RAM, é preciso modificar os valores dos parâmetros: em “DATA WIDTH” o valor será 32, que indica o tamanho da informação a ser salva na memória; e em “ADDR WIDTH” será a quantidade de espaços na memória, que inicialmente irá possuir o valor 5, ou seja, 32 endereços.

Figura 22 – Memoria de Instruções (ROM)

```
// Quartus Prime Verilog Template
// Single Port ROM

module MemoriaInstrucoesROM
#(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=5)
(
    input [(ADDR_WIDTH-1):0] addr,
    output reg [(DATA_WIDTH-1):0] q
);
    // Declare the ROM variable
    reg [DATA_WIDTH-1:0] rom[2**ADDR_WIDTH-1:0];
    // Initialize the ROM with $readmemb. Put the memory contents
    // in the file single_port_rom_init.txt. Without this file,
    // this design will not compile.
    // See Verilog LRM 1364-2001 Section 17.2.8 for details on the
    // format of this file, or see the "Using $readmemb and $readmemh"
    // template later in this section.
    initial
    begin
        $readmemb("single_port_rom_init.txt", rom);
    end
    always @ (*)
    begin
        q <= rom[addr];
    end
endmodule
```

Fonte: O Autor

4.5.6 Registrador Contador de Programa (PC)

O Contador de Programa tem por objetivo guardar o endereço da instruções que está atualmente sendo executada. Sua implementação pode ser vista na [Figura 23](#). Este módulo funciona aumentando em um o valor da “InstrucaoAtual” a cada ciclo de *clock*, fazendo com que o programa avance para a próxima instrução informada no arquivo txt da Memória de Instruções. Porém, existem casos em que desvios ou *branches* podem ocorrer. Para tal, a variável “InstrucaoModificada” foi criada para receber o endereço desses desvios caso o *bit* de “Selecao” esteja ativo.

Figura 23 – Contador de Programa (PC)

```

module PC (clock, Indice, IndiceAux, Selecao);
    input wire clock, Selecao;
    input wire [9:0] IndiceAux; // Depende do tamanho do ADDR_WIDTH
    output wire [9:0] Indice;
    reg [9:0] IndiceAtual;
    integer contador = 1;

    always @(posedge clock) begin
        if (contador == 1) begin
            IndiceAtual = 10'b0;
            contador = 0;
        end
        else if (Selecao == 1)
        begin
            IndiceAtual = IndiceAux;
        end
        else
        begin
            IndiceAtual = IndiceAtual + 1;
        end
    end
    assign Indice = IndiceAtual;
endmodule

```

Fonte: O Autor

4.5.7 Unidade de Controle

A Unidade de Controle é um módulo funcional combinacional, que servirá como um módulo gerador de *flags*. Estas flags servem para dizer quais multiplexados devem ser usados e qual dos terminais devem ser liberados, bem como permitir a leitura ou escrita na memória RAM e Banco de Registradores, dita as intruções do tipo I e J que serão executadas pela ULA, delega as de tipo R para a Unidade de Controle da ULA e portanto, também controla os desvios e saltos. Em suma, ela é fundamental para o processador como um todo, pois ela controla quase todas os módulos.

Figura 24 – Unidade de Controle

```

module UnidadedeControle (
    input [5:0] Opcode,
    output [2:0] AluOp,
    output RegDst, MemRead, MemToReg, MemWrite, ALUSrc,
    RegWrite, PCFunct, BEQ, BNE, ControlJump, Halt,
    EnableClock, JAL, Out, In
);

reg auxRegDst, auxMemRead, auxMemToReg, auxMemWrite,
auxALUSrc, auxRegWrite, auxPCFunct, auxBEQ, auxBNE,
auxControlJump, auxHalt, auxJAL, auxOut, auxIn;

reg [2:0] auxAluOp;
reg auxEnable;
always @(*)
begin
    case(Opcode)
        6'b000000: begin // Tipo R
            auxRegWrite <= 1;
            auxPCFunct <= 1;
            auxAluOp <= 3'b000;
            auxMemRead <= 0;
            auxMemWrite <= 0;
            auxMemToReg <= 0;
            auxALUSrc <= 0;
            auxRegDst <= 0;
            auxBEQ <= 0;
            auxBNE <= 0;
            auxControlJump <= 0;
            auxHalt <= 0;
            auxJAL <= 0;
            auxOut <= 0;
            auxIn <= 0;
        end
        6'b000001: begin //ADDI
            auxRegWrite <= 1;
            auxPCFunct <= 1;
            auxAluOp <= 3'b001;
            auxMemRead <= 0;
            auxMemWrite <= 0;
            auxMemToReg <= 0;
            auxALUSrc <= 1;
            auxRegDst <= 0;
            auxBEQ <= 0;
            auxBNE <= 0;
            auxControlJump <= 0;
            auxHalt <= 0;
            auxJAL <= 0;
            auxOut <= 0;
        end
    endcase
end

```

Fonte: O Autor

4.5.8 Unidade de Controle da ULA

A Unidade de Controle da ULA ficará responsável por todas as intruções do tipo R. Assim como o módulo anterior, ela é combinacional e irá selecionar certos bits de controle com base na entrada. Portanto, essa unidade terá duas entradas, uma será o campo “Funct”, que será um sinalizador de qual instrução será realizada para os tipos R, e a entrada “ALUOP”, que será advinda da outra unidade de controle.

Figura 25 – Unidade de Controle da ULA

```

module UnidadedeControleULA (Funct, AluOp, ControleALU, JALR, JR);

    input [5:0] Funct;
    input [2:0] AluOp;
    output [3:0] ControleALU;
    output JALR, JR;

    reg[3:0] RegControle;
    reg RegJALR, RegJR;

    always @(*)
    begin
        case (AluOp)
            3'b000: begin
                case (Funct)
                    6'b000001: begin // ADD
                        RegControle <= 4'b0010;
                        RegJALR <= 0;
                        RegJR <= 0;
                    end
                    6'b000010: begin // SUB
                    6'b000011: begin // DIV
                    6'b000100: begin // MULT
                    6'b000101: begin // AND
                    6'b000110: begin // OR
                    // 6'b000111: begin // NOT
                        RegControle = 4'b0000;
                    end
                    6'b000111: begin // NOR
                    6'b001000: begin // SLL
                    6'b001001: begin // SRL
                    6'b001010: begin // JR
                    6'b001011: begin // JALR
                    6'b001100: begin // SLT
                    6'b001101: begin // SLET
                    6'b001110: begin // SGT
                    6'b001111: begin // SGET
                endcase
            end
            3'b001: begin // SOMA
            3'b010: begin // SUB
            3'b011: begin // AND
            3'b100: begin // OR
            3'b101: begin // SLT
        endcase
    end

    assign ControleALU = RegControle;
    assign JALR = RegJALR;
    assign JR = RegJR;

endmodule

```

Fonte: O Autor

4.5.9 Módulos de Entrada e Saída

A comunicação entre o processador e os módulos externos será realizada pelos módulos de Entrada e Saída, ou seja, o usuário informa uma entrada e recebe uma resposta do sistema.

Para o módulo de entrada, será preciso que ele faça a obtenção e armazenamento dos valores atuais dos interruptores e botões do kit; e realizar um divisor de frequência para o *clock* interno do kit, já que será preciso diminuir a frequência final do processador, essa divisão acarretará em um tempo de 10 segundos para o usuário digitar uma entrada (instrução do tipo "in") ou 1 segundo para cada instrução normal.

Figura 26 – Módulo de Entrada

```

moduTe Entrada(
    input clock,
    output novo_clock,
    input in,
    input [15:0] interruptores,
    output [15:0] resultado_entrada
);

reg[27:0] out;
reg [15:0] resultado;
reg RegClock;

integer contador = 0;

always@(posedge clock) begin
    if (contador == 0) begin
        RegClock = 0;
        contador = 1;
    end

    if (in == 1) begin
        if(out == 28'd250000000) begin
            out = 28'd0;
            RegClock = ~RegClock;
        end
        else
            out = out + 1;
    end
    else begin
        if(out == 28'd25000000) begin
            out = 28'd0;
            RegClock = ~RegClock;
        end
        else
            out = out + 1;
    end
end
always @(*) begin
    resultado = interruptores[15:0];
end

// assign novo_clock = RegClock;
// assign novo_clock = clock;
// assign resultado_entrada = resultado;
endmodule

```

Fonte: O Autor

Assim, o módulo terá como entrada os interruptores e o *clock* de 50 MHz do FPGA. Após passar pelo divisor de frequência, o valor do *clock* final foi de 1 Hz. O código em descrição de *hardware* está apresentado na [Figura 26](#).

Para o módulo de saída, será preciso enviar a informação pedida pela instrução para os leds e para os *displays* presentes no kit FPGA. Vale ressaltar, que o módulo de saída recebe o resultado das operações do processador e soma com zero, uma vez que algo somado a zero, mantém seu valor. Portanto, essa soma será realizada pela ULA e esse resultado será repassado para os displays.

Figura 27 – Módulo de Saída

```

module Out (
    input [31:0] result_ULA,
    input controle, clock,
    output [6:0] display1, display2, display3, display4
);

    reg [31:0] saidaDisplay1, saidaDisplay2, saidaDisplay3, saidaDisplay4;
    integer contador1 = 1;

    always @(negedge clock) begin
        if (contador1 == 1) begin
            saidaDisplay1 = 4'b1110;
            saidaDisplay2 = 4'b1110;
            saidaDisplay3 = 4'b1110;
            saidaDisplay4 = 4'b1110;
            contador1 = 0;
        end
        else begin
            if (controle) begin
                saidaDisplay1 = result_ULA%10;
                saidaDisplay2 = (result_ULA%100)/10;
                saidaDisplay3 = (result_ULA%1000)/100;
                saidaDisplay4 = (result_ULA%10000)/1000;
            end
        end
    end
    Display1 bc1 (saidaDisplay1, display1);
    Display2 bc2 (saidaDisplay2, display2);
    Display3 bc3 (saidaDisplay3, display3);
    Display4 bc4 (saidaDisplay4, display4);
endmodule

```

Fonte: O Autor

Mais detalhadamente, foi preciso criar um módulo conversor binário para decimal (BCD) e, assim, adicionar as pinagens corretas para o *display* de sete segmentos para que os valores sejam atribuídos corretamente a eles. O código do módulo de saída e do BCD estão ilustrados, respectivamente, em [Figura 27](#) e [Figura 28](#).

Figura 28 – BCD 7 Segmentos

```

module Display1 (
    input [3:0] bcd,
    output reg [6:0] saida_display
);

    always @(*) begin
        case (bcd)
            4'b0000 : saida_display = 7'b1000000;
            4'b0001 : saida_display = 7'b1111001;
            4'b0010 : saida_display = 7'b0100100;
            4'b0011 : saida_display = 7'b0110000;
            4'b0100 : saida_display = 7'b0011001;
            4'b0101 : saida_display = 7'b00010010;
            4'b0110 : saida_display = 7'b0000010;
            4'b0111 : saida_display = 7'b1111000;
            4'b1000 : saida_display = 7'b0000000;
            4'b1001 : saida_display = 7'b0011000;
            default : saida_display = 7'b1111111;
        endcase
    end
endmodule

```

Fonte: O Autor

4.5.10 Multiplexadores

Os multiplexadores são módulos combinacionais que direcionam o caminho de dados do processador, já que dependendo do bit de controle sinalizado pelas unidades de controle será escolhido um dado entre dois. No total foram realizados três dos principais multiplexadores do processador, como indicado pela [Figura 5](#) do caminho de dados. Assim, as [Figura 29](#), [Figura 30](#) e [Figura 31](#) mostram os códigos desses multiplexadores.

Figura 29 – Mux ALUSrc

```
module MuxALUSRC(
    input[31:0] Dado2,
    input [15:0] Imediato,
    input [15:0] interruptores,
    input controle, in,
    output[31:0] Saida_AluSrc
);
    reg [31:0] reg_auxiliar;
    always @(*) begin
        if(controle == 1 && in == 0) begin
            reg_auxiliar = {16'd0, Imediato};
        end
        else if(controle == 1 && in == 1) begin
            reg_auxiliar = {16'd0, interruptores};
        end
        else begin
            reg_auxiliar = Dado2;
        end
    end
    assign Saida_AluSrc = reg_auxiliar;
endmodule
```

Fonte: O Autor

Figura 30 – Mux MemReg

```
module MuxMemReg(
    input[31:0] Dado_ULA, Dado_Mem,
    input controle,
    output[31:0] Saida_MemReg
);
    reg [31:0] reg_auxiliar;
    always @(*) begin
        if(controle) begin
            reg_auxiliar = Dado_Mem;
        end
        else begin
            reg_auxiliar = Dado_ULA;
        end
    end
    assign Saida_MemReg = reg_auxiliar;
endmodule
```

Fonte: O Autor

Figura 31 – Mux RegDst

```
module MuxRegDst(
    input[4:0] reg1, reg2,
    input controle,
    output[4:0] reg_saida
);
    reg [4:0] reg_auxiliar;
    always @(*) begin
        if(controle) begin
            reg_auxiliar = reg2;
        end
        else begin
            reg_auxiliar = reg1;
        end
    end
    assign reg_saida = reg_auxiliar;
endmodule
```

Fonte: O Autor

4.5.11 Integração Final

A integração final foi realizada no módulo da Unidade de Processamento, que pode ser vista na [Figura 19](#). Ela consiste na união de todos os módulos funcionais, é a parte final para que o processador funcione, será aqui onde serão adicionadas as interconexões entre eles, fazendo com que cada um interaja com o outro. Para isso, é preciso criar todos os fios que serão utilizados para realizar essas conexões. Cada um desses fios estão ilustrados no datapath [Figura 5](#) e na [Figura 19](#) citada da Unidade de Processamento.

5 Resultados Obtidos e Discussões

5.1 Formas de Onda

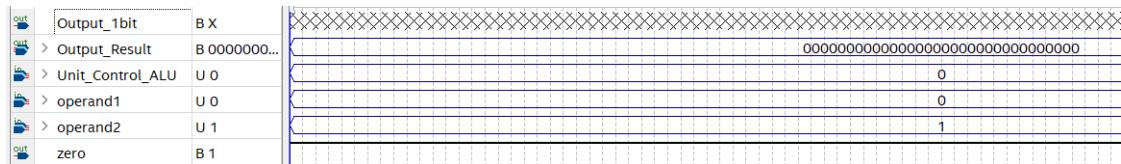
Como forma de testar se as unidades funcionais implementadas anteriormente estão operando adequadamente, utilizaremos do *software* Quartus para gerar formas de onda.

5.1.1 Teste da Unidade Lógica e Aritmética

A ULA deve ser testada levando em conta a [Tabela 7](#) que descreve suas funções. A Unidade de Controle irá definir qual operação será executada em cada momento.

Na [Figura 32](#) vemos que o “operand1” recebeu o valor 0 e o “operand2” o valor 1, pela lógica da porta AND teremos uma saída baixa, o que se confirma no binário retornado em “Output_Result”.

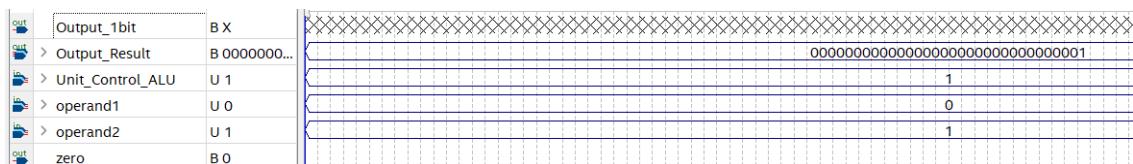
Figura 32 – Forma de onda AND - Controle (0000)



Fonte: O Autor

Na [Figura 33](#) vemos que o “operand1” recebeu o valor 0 e o “operand2” o valor 1, pela lógica da porta OR teremos uma saída alta, o que se confirma no binário retornado em “Output_Result”.

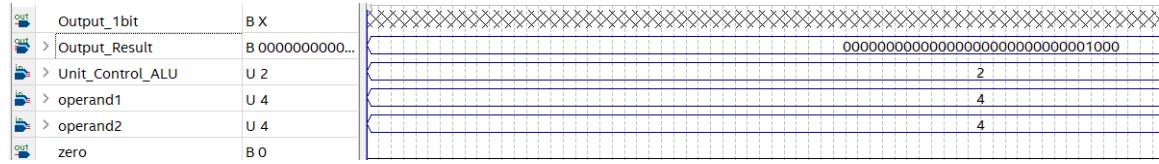
Figura 33 – Forma de onda OR - Controle (0001)



Fonte: O Autor

Na [Figura 34](#) vemos que o “operand1” recebeu o valor 4 e o “operand2” o valor 4, a soma deve retornar 8, o que se confirma no binário retornado em “Output_Result”.

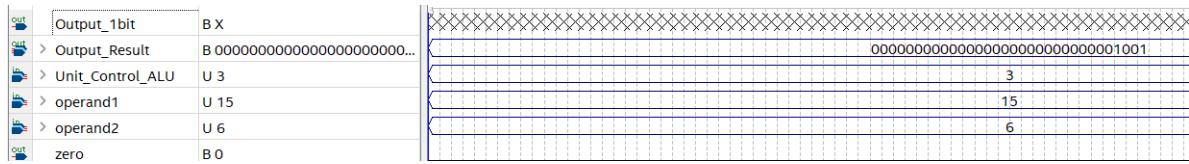
Figura 34 – Forma de onda ADD - Controle (0010)



Fonte: O Autor

Na Figura 35 vemos que o “operand1” recebeu o valor 15 e o “operand2” o valor 6, a subtração deve retornar 9, o que se confirma no binário retornado em “Output_Result”.

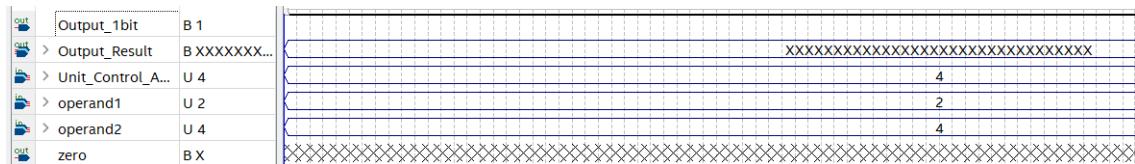
Figura 35 – Forma de onda SUB - Controle (0011)



Fonte: O Autor

Na Figura 36 vemos que o “operand1” recebeu o valor 4 e o “operand2” o valor 6, a comparação deve retornar saída alta uma vez que se trata de um *set less than*, o que se confirma no binário retornado em “Output_1bit”.

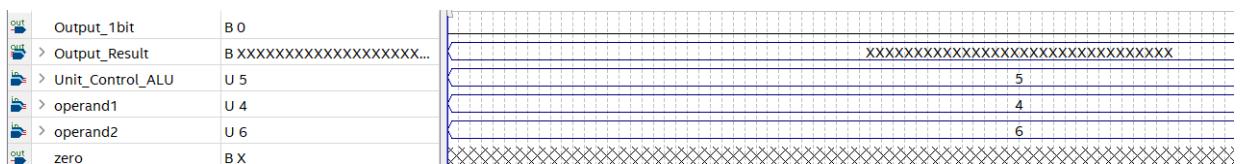
Figura 36 – Forma de onda SLT - Controle (0100)



Fonte: O Autor

Na Figura 37 vemos que o “operand1” recebeu o valor 4 e o “operand2” o valor 6, a comparação deve retornar saída baixa uma vez que se trata de um *set grater than*, o que se confirma no binário retornado em “Output_1bit”.

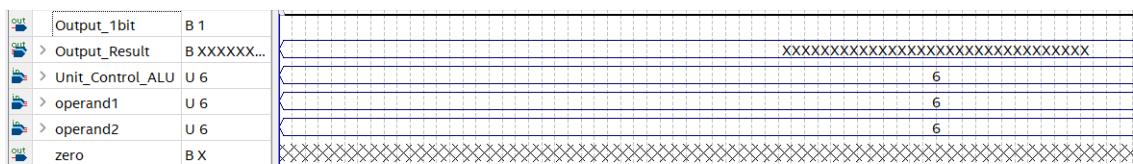
Figura 37 – Forma de onda SGT - Controle (0101)



Fonte: O Autor

Na [Figura 38](#) vemos que o “operand1” recebeu o valor 6 e o “operand2” o valor 6, a comparação deve retornar saída alta uma vez que se trata de um *set grater or equal than*, o que se confirma no binário retornado em “Output_1bit”.

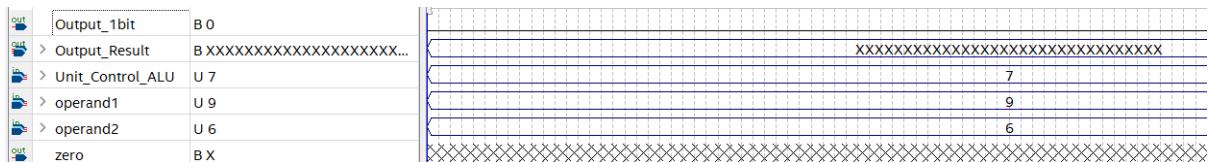
Figura 38 – Forma de onda SGET - Controle (0110)



Fonte: O Autor

Na [Figura 39](#) vemos que o “operand1” recebeu o valor 9 e o “operand2” o valor 6, a comparação deve retornar saída baixa uma vez que se trata de um *set less or equal than*, o que se confirma no binário retornado em “Output_1bit”.

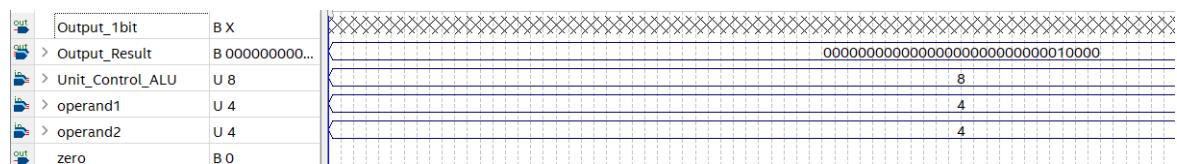
Figura 39 – Forma de onda SLET - Controle (0111)



Fonte: O Autor

Na [Figura 40](#) vemos que o “operand1” recebeu o valor 4 e o “operand2” o valor 4, a multiplicação deve retornar 16, o que se confirma no binário retornado em “Output_Result”.

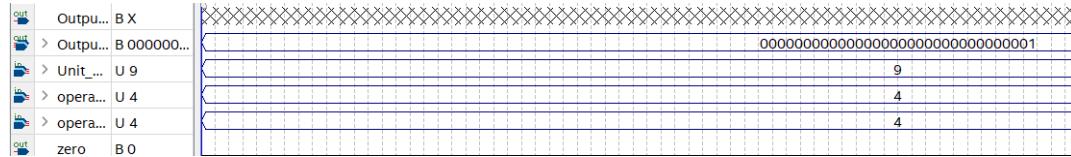
Figura 40 – Forma de onda MULT - Controle (1000)



Fonte: O Autor

Na [Figura 40](#) vemos que o “operand1” recebeu o valor 4 e o “operand2” o valor 4, a divisão deve retornar 1, o que se confirma no binário retornado em “Output_Result”.

Figura 41 – Forma de onda DIV - Controle (1001)



Fonte: O Autor

Na Figura 42 vemos que o “operand1” recebeu o valor 0 e o “operand2” o valor 0, pela lógica da porta NOR teremos uma saída alta, o que se confirma no binário retornado em “Output_Result”.

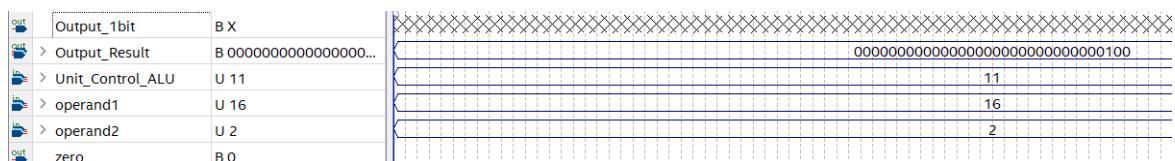
Figura 42 – Forma de onda NOR - Controle (1010)



Fonte: O Autor

Na Figura 43 vemos que o “operand1” recebeu o valor 16 e o “operand2” o valor 2, como estamos adicionando 2 zeros à esquerda teremos o resultado 4, o que se confirma no binário retornado em “Output_Result”.

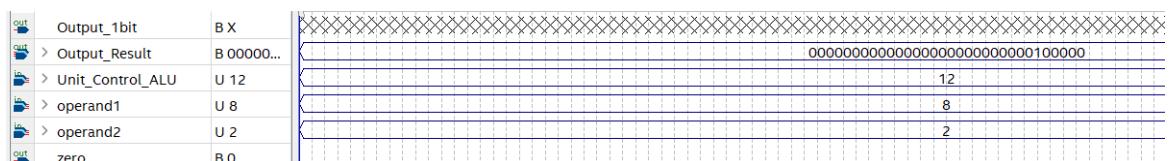
Figura 43 – Forma de onda SLL - Controle (1011)



Fonte: O Autor

Na Figura 44 vemos que o “operand1” recebeu o valor 8 e o “operand2” o valor 2, como estamos adicionando 2 zeros à direita teremos o resultado 32, o que se confirma no binário retornado em “Output_Result”.

Figura 44 – Forma de onda SRL - Controle (1100)

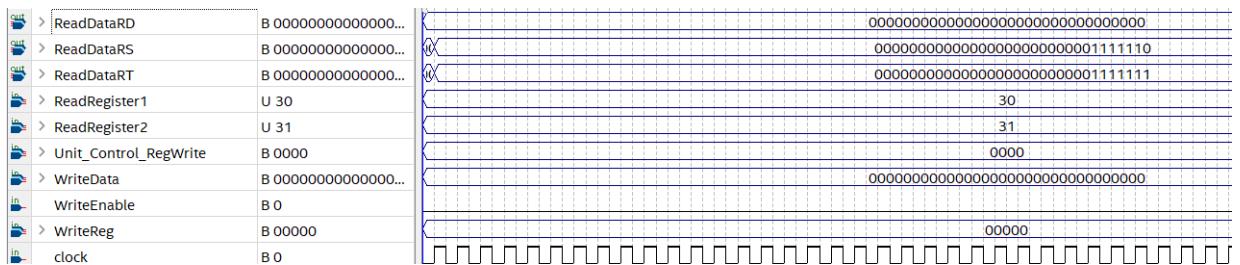


Fonte: O Autor

5.1.2 Teste do Banco de Registradores

Para começar a testar o banco de registradores foi feito o teste de leitura. Esse teste consistiu em atribuir os endereços dos registradores 30 e 31 do banco de registradores a “ReadRegister1” e a “ReadRegister2”. Estes que serão usados para ler os valores contidos e guardar nas variáveis “ReadDataRS” e “ReadDataRT”. Como podemos ver no código da Figura 46 colocamos nos registradores dois binários já inicialmente atribuídos, e podemos ver que estes binários foram lidos corretamente.

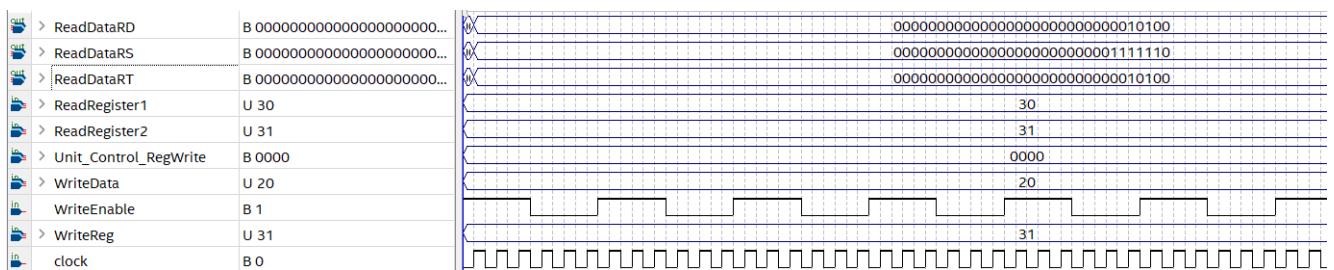
Figura 45 – Forma de onda de leitura do Banco de Registradores



Fonte: O Autor

Por fim, devemos testar a escrita no banco de registradores. Para tal, permitimos a escrita com o controle “WriteEnabled” em alta, passamos o valor 20 em decimal para o “WriteData” e o endereço 31 para “WriteReg”. Com isso sobrescrevemos a informação antes contida no registrador 31, e podemos ver que foi bem sucedido uma vez que o binário correspondente à 20 em decimal foi exibido em “ReadDataRT”.

Figura 46 – Forma de onda de escrita no Banco de Registradores



Fonte: O Autor

5.1.3 Teste da Memória ROM

A memória de Instruções (ROM) é a memória que irá conter os binários de todas as intruções que desejamos ser executadas pelo processador. Essa memória não permite que nada seja escrita nela, as informações somente são lidas (como a BIOS na placa-mãe dos computadores). Na Figura 47 podemos ver como podemos usar um arquivo txt para simular esse armazenamento.

Figura 47 – Arquivo txt com as instruções

```

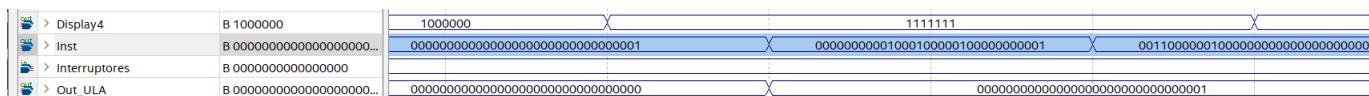
00000000000000000000000000000001
000000000100010000010000000001
00110000001000000000000000000000
00000000001000100001000000000001
00110000010000000000000000000000
00100000000000000000000000000000

```

Fonte: O Autor

Podemos ver se o funcionamento da mesma está correto, por meio da forma de onda. Na [Figura 48](#) a variável "inst" está recebendo os binários visto no arquivo de texto e avançando nas instruções. Portanto, a memória está sendo lida corretamente.

Figura 48 – Forma de onda instruções



Fonte: O Autor

5.2 Teste no *kit* FPGA

Para o teste final no *kit* FPGA, foi desenvolvido um pequeno código com instruções informadas no arquivo [Figura 47](#), que consiste na sequência de Fibonacci, código este que é capaz de mostrar que o processador está funcionando corretamente. A sequência pode ser conferida na [Equação 5.1](#).

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, \dots \quad (5.1)$$

A sequência deve começar com 0 e 1, para isso os registradores 1 e 2 do Banco de Registradores foram usados para guardar esses valores desde o começo. Essa atribuição pode ser vista na figura [Figura 49](#).

Figura 49 – Sequência de Fibonacci Registradores

```

input wire [31:0] WriteData,
input wire [3:0] Unit_Control_RegWrite,
input clock, WriteEnable,
output wire [31:0] ReadDataRD, ReadDataRS, ReadDataRT
);

integer First_clock=1;
reg [31:0] registers [31:0];

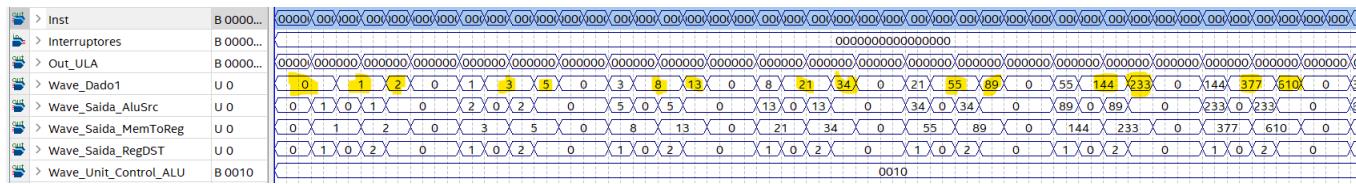
always @(posedge clock)
begin
  if (First_clock == 1)
    begin
      // Separo os 2 últimos registradores do banco para iniciá-los com valores padrão
      registers[31] = 32'b00000000000000000000000000111111; // Último registrador do banco com
      registers[30] = 32'b00000000000000000000000000111110; // Valor 126 para display apagado
      registers[1] = 32'd0;
      registers[2] = 32'd1;
    ..
  ..
end

```

Fonte: O Autor

O resultado final em forma de onda pode ser visto na imagem [Figura 50](#).

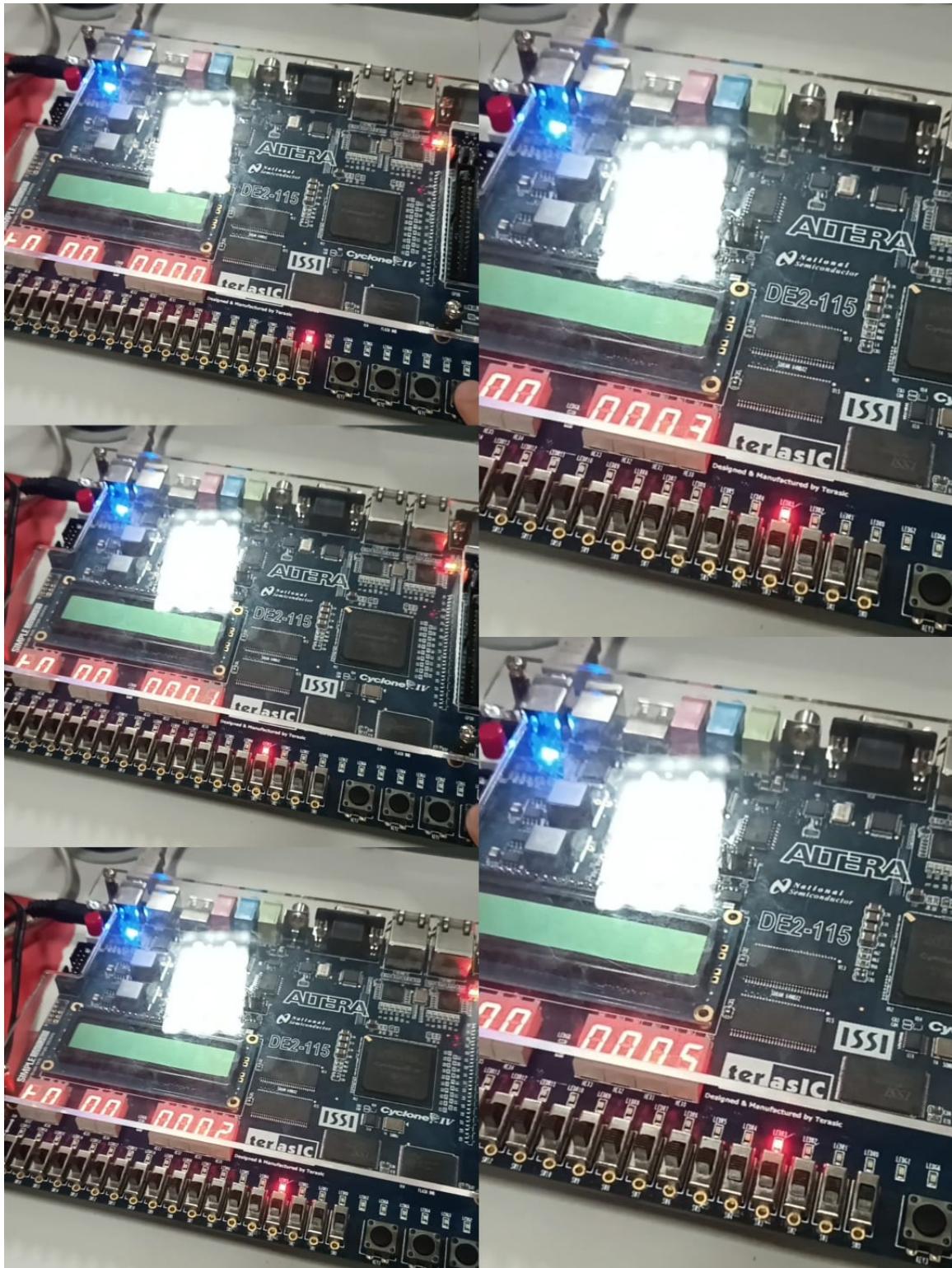
Figura 50 – Sequência de Fibonacci forma de onda



Fonte: O Autor

O resultado pode também ser visto nos displays na [Figura 51](#) que vão seguir a sequência até onde os 4 display conseguirem exibir.

Figura 51 – Sequência de Fibonacci FPGA



Fonte: O Autor

6 Considerações Finais

As principais dificuldades encontradas neste projeto foram a tradução dos conceitos e das unidades funcionais do datapath em código Verilog. Uma vez que sabemos do conceito por trás do Banco de Registradores e da Unidade Lógica e Aritmética, foi tanto necessário rever a sintaxe da linguagem bem como buscar referências que pudessem me tirar das etapas iniciais de desenvolvimento das unidades funcionais. Além disso, tive que repreender a gerar formas de ondas e lidar com os possíveis problemas na geração das mesmas.

Também vale ressaltar a demora para compilação, e de não conseguir exibir no relatório os testes dos módulos de entrada e saída. Realizei um algoritmo para calcular a área de um triângulo, porém como o laboratório remoto não estava funcionando não consegui capturar imagens que mostrassem o FPGA funcionando.

No mais, foi um processo de realmente colocar em prática e fixar os conteúdos vistos em Circuitos Digitais, Laboratório de Circuitos Digitais e Arquitetura e Organização de Computadores. Sem dúvida foi uma das matérias que mais me trouxeram empolgação ao ver o resultado final, e me impressionar com o tanto de integrações e conceitos diferentes sendo aplicados em conjunto.

Referências

ACERVO LIMA. *Diferença entre a arquitetura de von neumann e harvard*. Accesso em: 21 Abril 2023. Disponível em: <<https://acervolima.com/diferenca-entre-a-arquitetura-de-von-neumann-e-harvard/>>. Citado 2 vezes nas páginas 14 e 15.

ALTERA. *DE2-115 User Manual*: World leading fpga based products and design services. [s.n.], 2013. Disponível em: <https://grad.sead.unifesp.br/pluginfile.php/371633/mod_resource/content/1/%20DE2-115%20Manual.pdf/>. Citado na página 21.

HENNESSY, J. L.; PATTERSON, D. A. *Arquitetura de Computadores*: Uma abordagem quantitativa. 6. ed. [S.l.]: Elsevier, 2019. Citado 2 vezes nas páginas 8 e 11.

PATTERSON, D. *Organização e projeto de computadores*: a interface hardware/software. 5. ed. Rio de Janeiro: GEN LTC, 2017. Citado 2 vezes nas páginas 16 e 17.

PATTERSON, D. A.; HENNESSY, J. L. *Organização e Projeto de Computadores*: A interface hardware/software. 3. ed. [S.l.]: Elsevier Inc., 2005. Citado 3 vezes nas páginas 15, 16 e 19.

STALLINGS, W. *Arquitetura e Organização de Computadores*. 10. ed. São Paulo: Pearson, 2017. Citado na página 8.