

ID...

**Desenvolvimento de um sistema computacional
em lógica programável baseado na arquitetura
MIPS monociclo utilizando o kit Altera
DE2-115 com FPGA**

São José dos Campos - Brasil

Maio de 2019

ID...

**Desenvolvimento de um sistema computacional em lógica
programável baseado na arquitetura MIPS monociclo
utilizando o kit Altera DE2-115 com FPGA**

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. Fábio Augusto Menocci Cappabianco

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Maio de 2019

Resumo

O objetivo deste projeto é o desenvolvimento de um sistema computacional em lógica programável composto por processador, memória e interface de comunicação utilizando o kit Altera DE2-115 com FPGA. Inicialmente foi definida uma arquitetura base para esse processador, a arquitetura MIPS-32 monociclo, e um conjunto de instruções a ser reconhecido pelo sistema computacional. Baseando-se nessas definições, o caminho de dados do processador é implementado utilizando uma ferramenta de síntese de sistemas digitais e nesse caminho de dados são simuladas as instruções anteriormente determinadas para verificar a funcionalidade do projeto. O desenvolvimento do sistema computacional consiste da implementação em lógica programável do processador utilizando a linguagem de descrição de hardware Verilog e o software Quartus Prime, realização de testes e comparação das funcionalidades do circuito implementado com o esperado, implementação de um sistema de memória e de um sistema de comunicação entre o processador, a memória e o ambiente externo. A validação do projeto ocorre através da execução de algoritmos de teste.

Palavras-chaves: processador. arquitetura base. conjunto de instruções.

Lista de ilustrações

Figura 1 – Relação entre lógica combinatória e elementos de estado	13
Figura 2 – Divisor de frequência	13
Figura 3 – Visão alto nível de um sistema computacional	15
Figura 4 – Visão geral da Unidade Central de Processamento com barramento de sistema	16
Figura 5 – Visão geral da Unidade de Controle	18
Figura 6 – Visão geral da Unidade Lógica Aritmética	19
Figura 7 – Formato de instrução simples	22
Figura 8 – Diagrama de estado do ciclo de instrução	23
Figura 9 – Convenções de registradores MIPS	26
Figura 10 – Formato da instrução tipo R	27
Figura 11 – Formato da instrução tipo I	27
Figura 12 – Formato da instrução tipo J	28
Figura 13 – Conjunto de instruções da arquitetura MIPS	29
Figura 14 – Modos de endereçamento utilizados na arquitetura MIPS.	30
Figura 15 – Caminho de dados simplificado da arquitetura MIPS, sem as linhas controle.	31
Figura 16 – Caminho de dados simplificado da arquitetura MIPS, com as linhas de controle.	31
Figura 17 – Categorias entre os dispositivos lógicos programáveis.	33
Figura 18 – Estrutura básica de um FPGA.	33
Figura 19 – Configuração básica de um sistema para programação de PLD ou FPGA. .	34
Figura 20 – Diagrama de blocos do kit FPGA.	35
Figura 21 – Caminho de dados do projeto com sinais de controle	40
Figura 22 – Caminho de dados do projeto com sinais de controle e as unidades funcionais utilizadas nas instruções do tipo R em vermelho	42
Figura 23 – Caminho de dados do projeto com sinais de controle e as unidades funcionais utilizadas nas instruções do tipo I em vermelho	43
Figura 24 – Caminho de dados do projeto com sinais de controle e as unidades funcionais utilizadas nas instruções do tipo J em vermelho	44
Figura 25 – Caminho de dados conforme a implementação em lógica programável .	63
Figura 26 – Simulação do funcionamento do banco de registradores	65
Figura 27 – Simulação do funcionamento do extensor de 16 bits	65
Figura 28 – Simulação do funcionamento do extensor de 26 bits	66
Figura 29 – Simulação do funcionamento da unidade que incrementa o PC	66

Figura 30 – Simulação do funcionamento do multiplexador do dado de escrita no banco de registradores	66
Figura 31 – Simulação do funcionamento do multiplexador do segundo operando fornecido à ULA	67
Figura 32 – Simulação do funcionamento do multiplexador da próxima instrução fornecida ao PC	67
Figura 33 – Simulação do funcionamento do multiplexador do endereço do registrador de escrita	68
Figura 34 – Simulação do funcionamento <i>dprogram counter - PC</i>	68
Figura 35 – Simulação do funcionamento da unidade que soma o imediato ao PC .	69
Figura 36 – Simulação do funcionamento da unidade de entrada de dados	69
Figura 37 – Simulação do funcionamento da unidade de saída de dados	70
Figura 38 – Simulação do funcionamento da ULA	71
Figura 39 – Simulação do funcionamento da ULA	71
Figura 40 – Simulação do funcionamento da ULA	72
Figura 41 – Simulação do funcionamento da ULA	72
Figura 42 – Simulação do funcionamento da ULA	73
Figura 43 – Simulação do funcionamento da ULA	73
Figura 44 – Simulação do funcionamento da ULA	74
Figura 45 – Simulação do funcionamento da ULA	74
Figura 46 – Simulação do funcionamento da ULA	75
Figura 47 – Simulação do funcionamento da ULA	75
Figura 48 – Simulação do funcionamento da ULA	76
Figura 49 – Simulação do funcionamento da ULA	76
Figura 50 – Simulação do funcionamento da ULA	77
Figura 51 – Simulação do funcionamento da ULA	77
Figura 52 – Simulação do funcionamento das unidades interconectadas	78
Figura 53 – Registro do kit FPGA executando um conjunto de instruções qualquer.	79

Lista de tabelas

Tabela 1 – Conjunto de instruções elaborado para o projeto 38

Sumário

1	INTRODUÇÃO	9
2	OBJETIVOS	10
2.1	Geral	10
2.2	Específico	10
3	FUNDAMENTAÇÃO TEÓRICA	12
3.1	Circuitos lógicos e combinacionais	12
3.2	Círculo divisor de frequência	13
3.3	Sistema computacional	14
3.3.1	Unidade Central de Processamento	15
3.3.1.1	Unidade de Controle	16
3.3.1.2	Registradores	18
3.3.1.3	Unidade Lógica Aritmética (ULA)	19
3.3.2	Memória	19
3.3.3	Módulo de Entrada e Saída (E/S)	19
3.4	CISC x RISC	20
3.5	Conjunto de instruções	20
3.6	Arquitetura MIPS	24
3.6.1	PC	24
3.6.2	ULA	25
3.6.3	Memória	25
3.6.4	Extensor de bits	25
3.6.5	Banco de registradores	26
3.6.6	Unidade de Controle	26
3.6.7	Conjunto de instruções	26
3.6.7.1	Tipo R de registrador	27
3.6.7.2	Tipo I de imediato	27
3.6.7.3	Tipo J de <i>jump</i>	28
3.6.8	Modos de endereçamento	29
3.6.9	Caminho de dados	30
3.7	Quartus Prime e Linguagem de Descrição de Hardware	32
3.8	Arranjo de portas programáveis por campo - FPGA	32
4	DESENVOLVIMENTO	36
4.1	Arquitetura base	36

4.2	Conjunto de instruções	36
4.2.1	Formato de instrução	36
4.2.2	Modos de endereçamento	37
4.3	Caminho de dados	39
4.4	Verificação das instruções	41
4.5	Implementação em lógica programável	45
4.5.1	PC	45
4.5.2	Banco de registradores	45
4.5.3	Extensor de 16 bits	47
4.5.4	Extensor de 26 bits	48
4.5.5	Somador que incrementa o PC	48
4.5.6	Multiplexador para o registrador de escrita	49
4.5.7	Multiplexador para o endereço da próxima instrução	49
4.5.8	Multiplexador para o dado fornecido à ULA	50
4.5.9	Multiplexador para o dado que será escrito	51
4.5.10	Somador para o imediato	52
4.5.11	Unidade Lógica e Aritmética	52
4.6	Unidades de memória em lógica programável	54
4.6.1	Memória de instruções	54
4.6.2	Memória de dados	55
4.7	Unidades de entrada e saída em lógica programável	56
4.7.1	Unidade de entrada	56
4.7.2	Unidade de saída	58
4.8	Unidades funcionais interconectadas sem a Unidade de Controle	60
4.8.1	Novo caminho de dados	62
4.8.2	Unidade de controle	64
5	RESULTADOS OBTIDOS E DISCUSSÃO	65
5.0.1	Banco de registradores	65
5.0.2	Extensor de 16 bits	65
5.0.3	Extensor de 26 bits	66
5.0.4	Incremento ao PC	66
5.0.5	Multiplexador do dado de escrita no banco de registradores	66
5.0.6	Multiplexador do segundo operando fornecido à ULA	67
5.0.7	Multiplexador da próxima instrução fornecida ao PC	67
5.0.8	Multiplexador do endereço do registrador de escrita	68
5.0.9	<i>Program Counter - PC</i>	68
5.0.10	Somador do imediato ao PC	69
5.0.11	Unidade de entrada de dados	69
5.0.12	Unidade de saída de dados	70

5.0.13	Unidade Lógica e Aritmética	70
5.0.13.1	Operações entrada (0), saída (1), adição (2) e subtração (3)	71
5.0.13.2	Operações multiplicação (4), divisão (5) e menor que (6)	71
5.0.13.3	Operações maior que (7), igual a (8) e <i>jump register</i> (9)	72
5.0.13.4	Operações and (10), or (11) e resto (12)	73
5.0.13.5	Operações xor (13), not (14) e mover (15)	74
5.0.13.6	Operações desloca esquerda (16), desloca direita (17) e ajuste de clock (18) . .	75
5.0.13.7	Operações <i>jump</i> (19), <i>jump and link</i> (20) e <i>load</i> (21)	75
5.0.13.8	Operações <i>store</i> (22), adição com imediato (23) e subtração com imediato (24) .	76
5.0.13.9	Operações desvie se igual (25), desvie se diferente (26) e carregar imediato (27)	76
5.0.13.10	Operação carrega superior imediato (28)	77
5.0.14	Unidades funcionais interconectadas	77
6	CONSIDERAÇÕES FINAIS	80
 REFERÊNCIAS		82
 APÊNDICES		83
APÊNDICE A – APÊNDICE A		84
APÊNDICE B – APÊNDICE B		88
APÊNDICE C – APÊNDICE C		93

1 Introdução

Se descrevermos um sistema computacional como aquele que processa informações que auxiliam na realização das mais diversas atividades humanas, pode-se dizer que os primeiros sistemas computacionais surgiram na antiguidade, como resultado da necessidade de se executar cálculos matemáticos, e eram essencialmente sistemas mecânicos de calcular, como, por exemplo, o ábaco, uma calculadora que se apresentou de diversas formas, em uma delas feita com fios paralelos e arruelas deslizantes que somavam e subtraiam.

Contudo, segundo (1), a primeira máquina de calcular foi desenvolvida em 1623 pelo alemão Wilhelm Schickard, então professor de hebraico e astronomia da *Universität Tübingen*, utilizando engrenagens de relógios, e, por essa razão, a máquina ficou conhecida como *relógio calculador*. Já a segunda calculadora, mais conhecida que a primeira, foi desenvolvida pelo francês Blaise Pascal aos 19 anos de idade, em 1642, e ficou conhecida como *Pascaline* ou *As rodas dentadas de Pascal*.

O fator comum entre o ábaco, o relógio calculador e a Pascaline é a finalidade, pois os três sistemas computacionais foram desenvolvidos, a princípio, para calcular, mas olhando além disso, foram desenvolvidos para facilitar atividades humanas. Atualmente os sistemas computacionais atendem a esse mesmo propósito e estão presentes nos dispositivos eletrônicos que fazem parte do nosso cotidiano e nos auxiliam na resolução das mais variadas tarefas, desde as mais básicas, como aquecer um prato de comida no forno micro-ondas, até as mais avançadas, como o lançamento de foguetes.

Diante desse contexto, pode-se observar que os sistemas computacionais surgiram da necessidade de desenvolvimento de ferramentas que facilitassem a execução de tarefas, como a criação do ábaco diante da necessidade de calcular, mas hoje são as próprias ferramentas para o desenvolvimento, como a utilização de sistemas computacionais para desenvolver e simular os mais diversos projetos, visto que todo o avanço atual, seja ele científico, social ou econômico, faz uso dos sistemas computacionais como ferramenta de pesquisa e desenvolvimento. Portanto, a importância do estudo da arquitetura e organização de computadores está no fato de que o avanço nesta área propicia o desenvolvimento dos sistemas computacionais que, consequentemente, propiciam o desenvolvimento de todas as demais áreas.

2 Objetivos

2.1 Geral

Projeto e implementação de um sistema computacional em lógica programável baseado na arquitetura MIPS monociclo utilizando o kit FPGA Altera DE2-115.

O sistema computacional deve ser composto por processador, memória e interface de comunicação entre o processador, a memória e o ambiente externo.

2.2 Específico

O projeto e a implementação do sistema computacional foi dividido em 7 etapas, listadas abaixo:

1. Descrição da arquitetura do projeto:

- Determinação da arquitetura base utilizada no projeto;
- Elaboração do conjunto de instruções a ser compreendido pelo sistema computacional;
- Elaboração do formato das instruções a serem utilizados no projeto;
- Elaboração dos modos de endereçamento a serem utilizados no projeto;
- Elaboração do esquemático da arquitetura que representa caminho de dados compatível com os elementos anteriormente estabelecidos utilizando ferramentas de síntese de sistemas digitais;
- Elaboração da unidade de controle;
- Verificação da funcionalidade do conjunto de instruções escolhido sobre o caminho de dados elaborado.

2. Implementação do processador utilizando lógica programável

Implementar os seguintes itens em linguagem de descrição de hardware:

- PC;
- Banco de registradores;
- Extensor de 16 bits;
- Extensor de 26 bits;
- Somador que incrementa o PC;

- Multiplexador para o registrador de escrita;
- Multiplexador para o endereço da próxima instrução;
- Multiplexador para o dado fornecido à ULA;
- Multiplexador para o dado que será escrito;
- Somador para o imediato;
- Unidade Lógica e Aritmética.

3. Desenvolvimento do sistema de memória em lógica programável

Utilizar os *templates* disponíveis no Quartus para configurar a memória de instruções e a memória de dados do processador.

4. Desenvolvimento do sistema de comunicação em lógica programável

Implementar as unidades de entrada e saída de informações do processador conforme o *datapath*.

5. Simulação e testes para verificação do funcionamento das unidades funcionais implementadas

Consiste na união dos elementos desenvolvidos e na simulação das unidades funcionais implementadas até o momento para verificar se funciona conforme previsto.

6. Implementação da Unidade de Controle do processador

7. Simulação e testes para verificação do sistema computacional

3 Fundamentação Teórica

A seção de fundamentação teórica aborda alguns dos conceitos necessários para a compreensão do projeto em função da arquitetura base e conjunto de instruções escolhidos e baseia-se essencialmente nas obras (2) e (3).

3.1 Circuitos lógicos e combinacionais

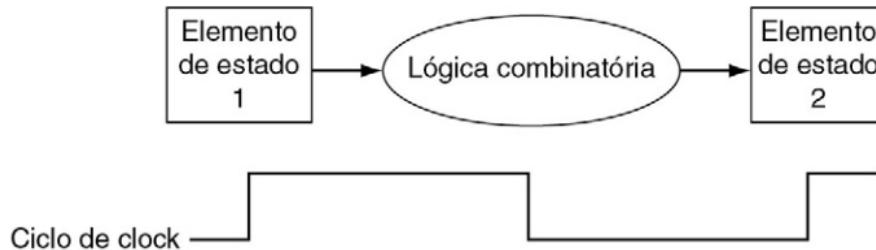
O desenvolvimento de um sistema computacional requer uma implementação lógica de cada elemento projetado e uma forma de sincronização desses elementos. Segundo (2), os elementos do caminho de dados em uma implementação consistem em dois tipos diferentes de elementos lógicos, sendo aqueles que operam nos valores dos dados e os que contêm estado.

Em um elemento combinacional, suas saídas dependem somente dos valores atuais da entrada, ou seja, dada uma mesma entrada esse elemento produz a mesma saída pois não possui armazenamento interno, um exemplo desse tipo de elemento é a ULA, discutida a seguir.

Os demais elementos do projeto são chamados elementos de estado, ou elementos sequenciais (um exemplo simples é o (flip-flop) tipo D), pois contêm estados e suas saídas dependem da entrada e do estado interno, ou seja, possuem armazenamento interno e atuam como uma memória. Alguns exemplos desse tipo de elemento são a memória de instruções e os registradores. Um elemento de estado possui ao menos duas entradas e uma saída, uma das entradas é o valor a ser escrito no elemento e a outra é o *clock* (que, através de sua transição, determina quando esse valor deve ser escrito), a saída fornece o valor escrito no elemento no ciclo de *clock* anterior.

A lógica combinatória e os elementos de estados estão intimamente relacionados conforme a figura 1. Pode-se observar que, na transição do nível baixo para o alto do *clock*, o elemento de estado 1 é atualizado e na sequência esse valor é processado para, então, novamente na transição do *clock* do nível baixo para o nível alto, o elemento de estado 2 ser atualizado.

Figura 1 – Relação entre lógica combinatória e elementos de estado



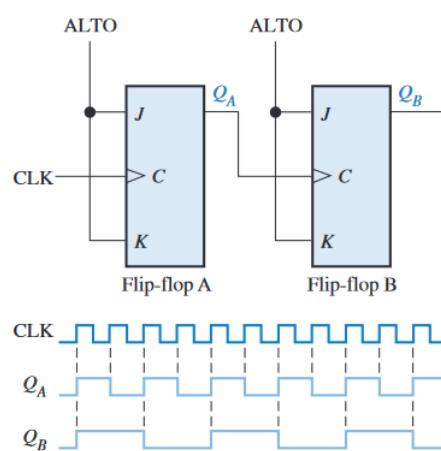
Fonte: Organização e Projeto de Computadores ([2](#))

Portanto, as unidades que compõem um sistema computacional são formadas da combinação de vários desses elementos combinacionais e de estado.

3.2 Circuito divisor de frequência

Segundo Floyd, um circuito divisor de frequência divide a frequência de uma onda periódica e pode ser implementado com o uso de *flip-flops*. A implementação requer que uma forma de onda retangular seja aplicada na entrada de *clock* de um *flip-flop* que esteja conectado no modo *toggle* (modo de comutação) e a saída do *flip-flop* é uma onda quadrada com metade da frequência do sinal de entrada do *clock*. Divisões posteriores podem ser obtidas ao conectar-se as saídas dos *flip-flops* anteriores às entradas de *clock* dos *flip-flops* posteriores e dessa forma pode-se calcular a frequência final que é a potência obtida ao se elevar a base dois ao número de *flip-flops* utilizados no circuito. Na figura [2](#) o sinal de *clock* é dividido pelo *Flip-flop A* e dividido novamente pelo *Flip-flop B*.

Figura 2 – Divisor de frequência



Fonte: Sistemas Digitais ([4](#))

3.3 Sistema computacional

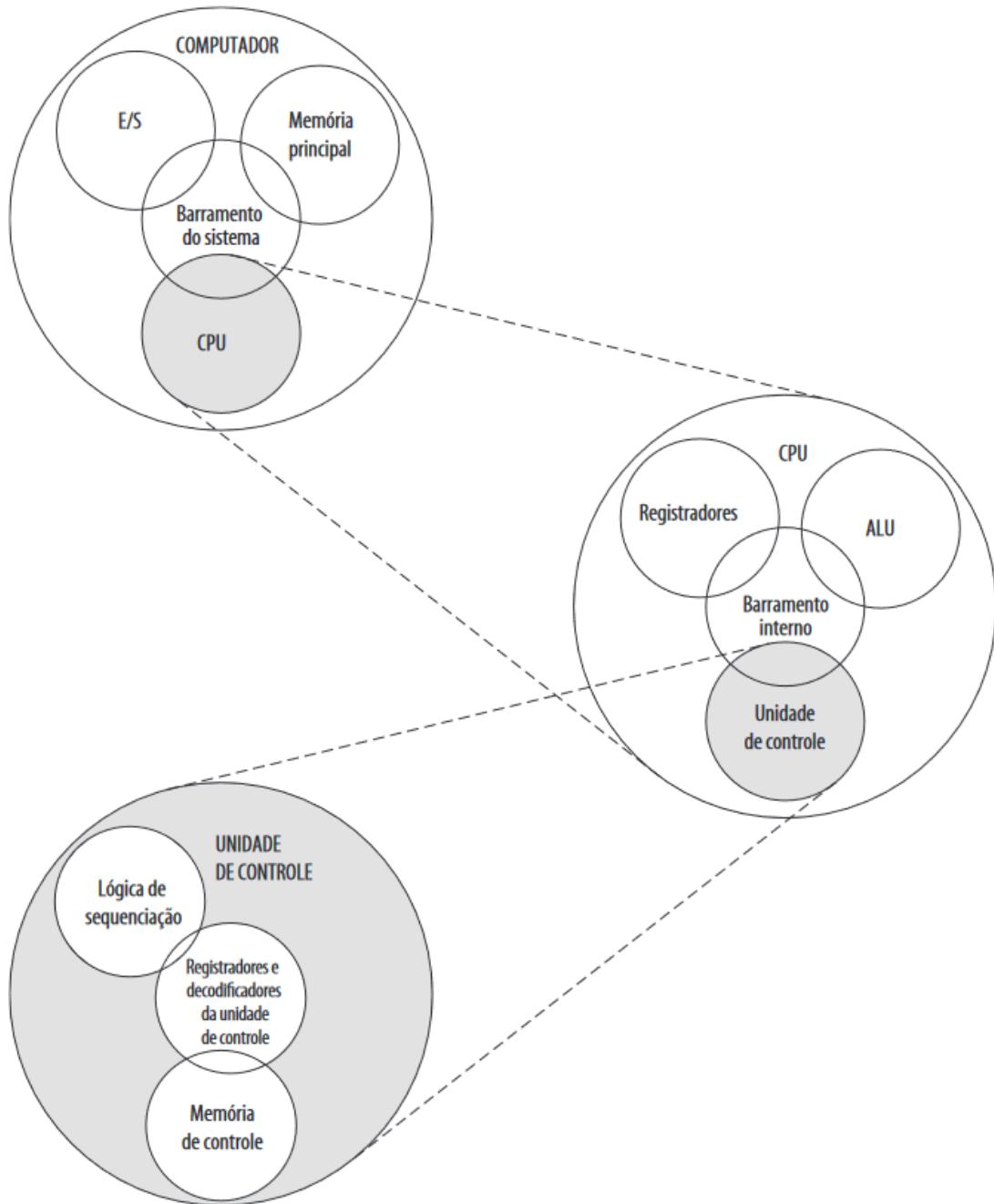
Segundo (3), definir um computador é uma tarefa desafiadora devido à dois fatores: o primeiro fator é o rápido ritmo de mudança que caracteriza essa tecnologia e o segundo é a grande diversidade de produtos que podem ostentar o nome "computador", que abrange desde microcomputadores até supercomputadores. Uma alternativa para a definição é através das funções básicas que um computador deve realizar, são quatro: processamento de dados, armazenamento de dados, movimentação de dados e controle.

Contudo, ainda segundo (3), um sistema computacional consiste da inter-relação de um conjunto de componentes e pode ser melhor definido em função de sua estrutura (modo como os componentes são inter-relacionados) e sua função (operação individual de cada componente na estrutura). Em uma análise superficial, pode-se descrever um sistema computacional hierarquicamente de cima para baixo, da seguinte forma:

- Sistema computacional: composto pela unidade central de processamento (também conhecida como processador ou CPU, do inglês *Central Processing Unit*), memória e a unidade de entrada e saída (E/S) que são interconectadas via barramentos.
- Unidade Central de Processamento: os principais componentes são a unidade de controle (UC), os registradores e a unidade lógica e aritmética (ULA) e o barramento interno, também conhecido como caminho de dados.
- Unidade de controle: oferece os sinais de controle da operação e coordena os demais componentes do processador.

A Figura 3 apresenta essa descrição hierárquica de um sistema computacional, o sistema computacional é expandido e possui a CPU que é também expandida e possui a Unidade de Controle que é também expandida.

Figura 3 – Visão alto nível de um sistema computacional



Fonte: Arquitetura e Organização de Computadores (3)

3.3.1 Unidade Central de Processamento

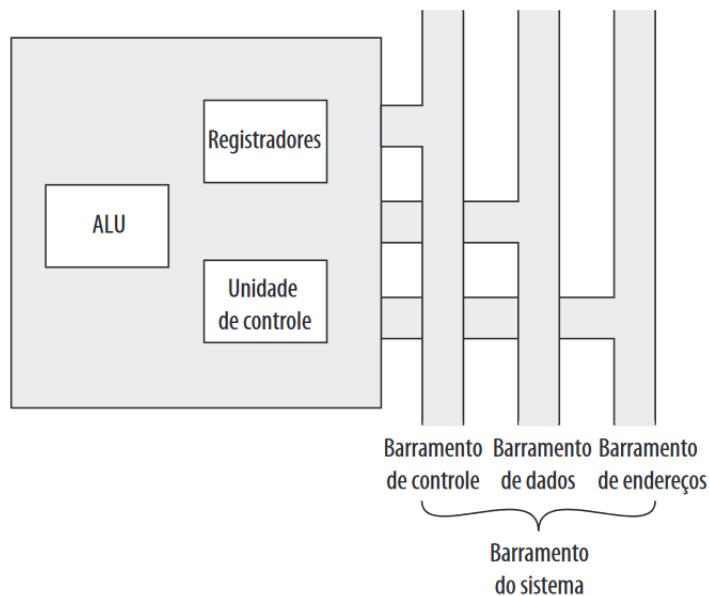
Conforme mencionado anteriormente, o processador possui como unidades principais a unidade de Controle (UC), a Unidade Lógica e Aritmética (ULA) e registradores. Superficialmente, segundo (3), pode-se descrever as funções do processador da seguinte maneira:

- Buscar instrução: lê uma instrução da memória;

- Interpretar a instrução: decodificar a instrução para identificar qual a ação requerida;
- Obter os dados: caso a instrução requeira leitura de dados da memória ou do módulo de E/S;
- Processar os dados: caso a instrução requeira uma operação aritmética ou lógica com os dados;
- Gravar os dados: caso a instrução requeira que dados sejam gravados na memória.

Além dos itens anteriormente mencionados, o processador também recebe sinais de interrupção. A figura 4 apresenta uma visão superficial do processador e seus elementos, pode-se observar os barramentos do processador interconectando as unidades funcionais.

Figura 4 – Visão geral da Unidade Central de Processamento com barramento de sistema



Fonte: Arquitetura e Organização de Computadores (3)

3.3.1.1 Unidade de Controle

A unidade de controle é a unidade do processador que coordena o funcionamento do processador através da emissão sinais de controle externos ao processador, que possibilitam a troca de dados com memória e módulos de E/S, e sinais de controle internos ao processador, que movem dados entre os registradores, fazem com que a ULA efetue uma determinada função e regulam outras operações internas.

A entrada para unidade de controle tem origem no registrador de instrução, em *flags* e sinais de controle de fontes externas (por exemplo, sinais de interrupção).

Como cada ciclo de instrução é composto por um conjunto de micro-operações que geram sinais de controle, a execução das instruções ocorre com o efeito desses sinais de

controle que se originam na unidade de controle e vão para ULA, registradores e estrutura de interconexão do sistema.

A unidade de controle precisa ter entradas que lhe permitam determinar o estado do sistema e saídas que lhe permitam controlar o comportamento do sistema e, internamente, a unidade de controle precisa ter a lógica necessária para desempenhar as suas funções de sequenciamento e execução. A seguir estão listadas algumas das entradas necessárias na Unidade de Controle:

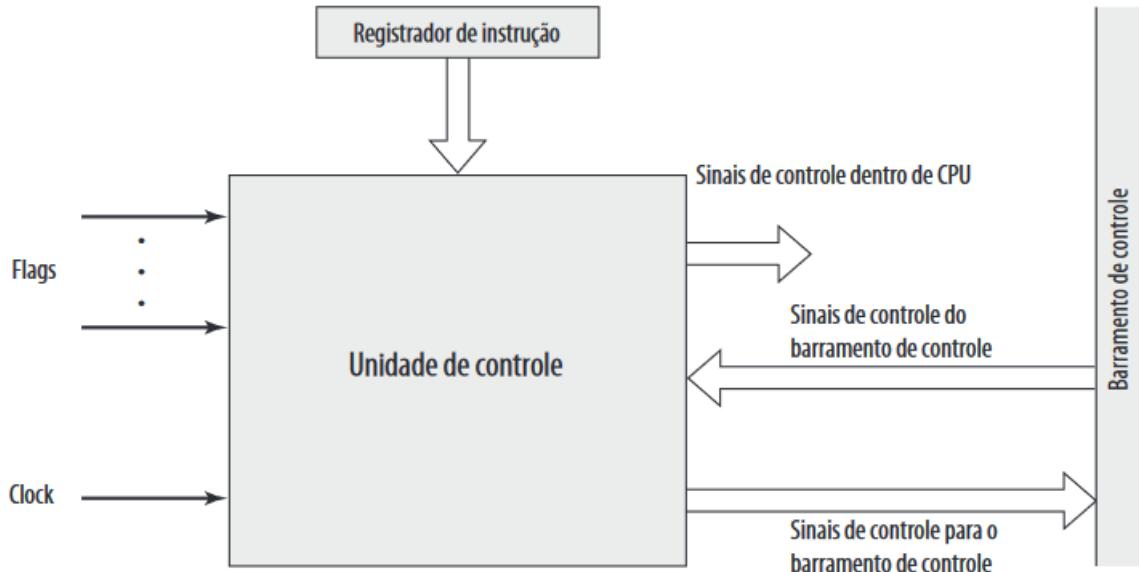
- *Clock*: é como a UC temporiza as instruções. No caso de processadores monociclo, a UC faz com que um conjunto de micro-operações sejam executadas dentro de um mesmo ciclo de *clock*;
- Registrador de instrução: o opcode e o modo de endereçamento da instrução atual são usados para determinar qual a micro-operação executar;
- *Flags*: são necessárias para a UC determinar o estado atual do processador;
- Sinais de controle do barramento de controle: o barramento de controle é uma parte do barramento de sistema que fornece sinais para a UC.

A seguir estão listadas as saídas da Unidade de Controle:

- Sinais de controle dentro do processador: são separados em dois tipos, aqueles que movem dados entre registradores e aqueles que ativam funções específicas da ULA;
- Sinais de controle para barramento de controle: separados também em dois tipos, os sinais de controle para a memória e os sinais de controle para módulos de E/S.

A figura 5 apresenta uma visão geral da UC com suas entradas e saídas, pode-se observar que ela recebe a instrução, algumas *flags* e o sinal de *clock* e gera sinais de controle.

Figura 5 – Visão geral da Unidade de Controle



Fonte: Arquitetura e Organização de Computadores (3)

A cada ciclo de *clock*, a unidade de controle lê todas as suas entradas e emite um conjunto de sinais de controle que vão para três destinos diferentes:

- Caminho de dados: a UC controla o fluxo de dados;
- Unidade Lógica Aritmética: a UC controla a operação da ULA através de um conjunto de sinais de controle, esses sinais ativam circuitos e portas lógicas dentro da ULA;
- Barramento do sistema: a UC envia sinais de controle para as linhas de controle de barramento do sistema.

3.3.1.2 Registradores

Para realizar as operações acima mencionadas o processador precisa armazenar instruções e dados temporariamente enquanto uma instrução está sendo executada, ou seja, precisa de uma pequena memória interna e essa é a função dos registradores, ou seja, oferecem armazenamento interno à CPU.

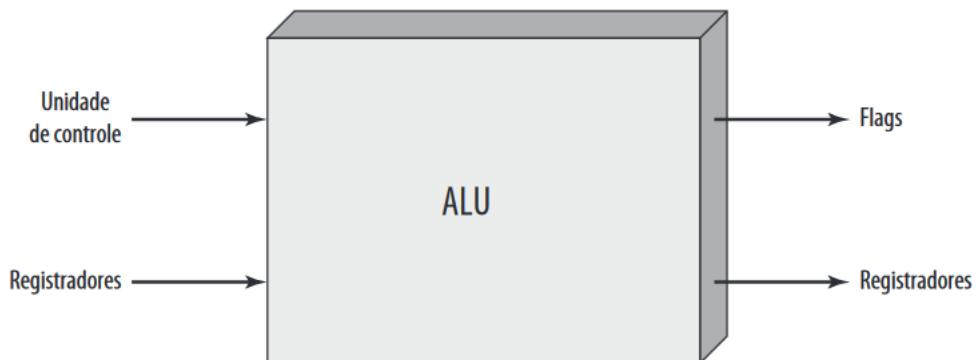
Os registradores da CPU podem ser visíveis ou não ao usuário. Quando são invisíveis são usados como registradores de controle de operação do processador, por exemplo o *Program Counter* (PC), quando são visíveis os registradores podem ser referenciados implícita ou explicitamente nas instruções de máquina e podem ter uso geral ou específico, usados para armazenar dados, endereços, etc.

3.3.1.3 Unidade Lógica Aritmética (ULA)

A ULA é o componente do computador que realiza operações lógicas e aritméticas sobre os dados. Todos os outros elementos do sistema de computação (unidade de controle, registradores, memória, E/S) existem principalmente para trazer dados para a ULA processar.

Os dados temporariamente armazenados nos registradores são fornecidos para a ULA, processados e armazenados novamente em registradores. A Unidade de Controle é responsável por emitir os sinais que coordenam essas tarefas. A figura 6 apresenta uma visão geral da ULA com os sinais da UC e os dados dos registradores como entradas e os registradores e *flags* como possíveis saídas.

Figura 6 – Visão geral da Unidade Lógica Aritmética



Fonte: Arquitetura e Organização de Computadores (3)

3.3.2 Memória

Dentre as formas de se implementar a memória nos sistemas computacionais, duas se destacam: a arquitetura Harvard e a Von Neumann. Na arquitetura Harvard há duas memórias separadas, uma para armazenar dados e outra para armazenar instruções. Na arquitetura de Von Neumann os dados são armazenados em uma única memória de leitura e escrita e esse conteúdo é endereçável por local, independente do tipo de dado, a execução ocorre na sequência de uma instrução para a seguinte, a não ser que seja modificado explicitamente.

Portanto, um módulo de memória consiste em um conjunto de locais, definidos por endereços numerados sequencialmente. Cada local contém um número binário que pode ser interpretado como uma instrução ou um dado.

3.3.3 Módulo de Entrada e Saída (E/S)

O módulo de E/S contém uma lógica para realizar uma função de comunicação entre o periférico e o barramento, a arquitetura do módulo de entrada e saída (E/S) em

um sistema de computação é a sua interface com o mundo exterior e oferece um meio estruturado de controlar a interação com o mundo exterior.

Existem três técnicas principais de E/S:

- E/S programada: a E/S ocorre sob o controle direto e contínuo do programa solicitando a operação de E/S;
- E/S controlada por interrupção: um programa emite um comando de E/S e depois continua a executar até que seja interrompido pelo hardware de E/S para sinalizar o final da operação de E/S;
- E/S por Acesso Direto à Memória (DMA, do inglês *Direct Memory Access*): um processador especializado de E/S assume o controle de uma operação de E/S para mover um grande bloco de dados.

3.4 CISC x RISC

Projetar um conjunto de instruções requer a escolha de uma entre duas abordagens, as abordagens RISC, do inglês *Reduced Instruction Set Computer* ou Computador de Conjunto de Instruções Reduzido, e CISC, do inglês *Complex Instruction Set Computer* ou Computador de Conjunto de Instruções Complexo.

Os computadores RISC visam resultar em uma Unidade de Controle simples, barata e rápida, já os computadores CISC visam desenvolver arquiteturas complexas mas que facilitam a construção de compiladores.

As arquiteturas RISC geralmente optam por instruções mais simples, com pouca variedade, poucos endereços e modos de endereçamento e tamanho fixo. As arquiteturas CISC possuem mais tipos de dados e instruções com muitos endereços, muitos modos de endereçamento, e tamanho de instrução variado.

3.5 Conjunto de instruções

Da mesma forma que um programa em alto nível possui instruções em linguagem C utilizadas para multiplicar, subtrair, somar, fazer desvios ou *loops*, o conjunto de instruções é composto instruções compreendidas pela CPU e executadas dentro do processador e é conhecido como código de máquina, que nada mais é do que o programa em binário.

O código de máquina comumente é representado por códigos em Assembly, que são representações das instruções em código de máquina porém não binária, são caracteres alfanuméricos que facilitam a leitura humana.

Esse conjunto de instruções, ou ISA, do inglês (*instruction set architecture*), é a porção da máquina visível ao programador em nível de montagem ou aos projetistas de compiladores, pode-se dizer que trata-se de uma interface entre o hardware e o software.

A construção de um conjunto de instruções adequado requer que o conjunto sobreviva a várias implementações, que seja generalista e versátil para que possa ser utilizado de diferentes formas, deve prover funcionalidades para os níveis superiores e garantir que os programadores saibam e consigam implementar aquilo que for necessário e permitir uma implementação eficiente.

Portanto, segundo (3), a operação do processador é determinada pelas instruções que ele executa, conhecidas como instruções de máquina ou instruções de computador. A coleção das diferentes instruções que o processador pode executar é conhecida como conjunto de instruções do processador.

Abaixo estão listados os elementos que compõem uma instrução de máquina e suas funções:

- Código de operação: especifica a operação a ser realizada através de um código binário conhecido como código de operação ou *opcode*, do inglês *operation code*;
- Referência à operando fonte: a operação pode envolver um ou mais operandos fontes que são utilizados como entradas para a operação;
- Referência à operando destino: utilizado para o resultado produzido pela operação;
- Referência à próxima instrução: indica ao processador onde buscar a próxima instrução após o término da instrução atual.

Os operandos fonte e destino podem estar em uma das seguintes áreas:

- Memória principal ou virtual;
- Registrador do processador;
- Imediato;
- Dispositivo de E/S.

A figura 7 apresenta um formato de instrução simples genérico.

Figura 7 – Formato de instrução simples



Fonte: Arquitetura e Organização de Computadores (3)

Qualquer programa escrito em linguagem de alto nível precisa ser traduzido para a linguagem de máquina para que possa ser executado e, dessa forma, o conjunto de instruções de máquina precisa ser suficiente para expressar qualquer instrução de um programa de alto nível. Portanto, as instruções podem ser categorizadas da seguinte forma:

- Processamento de dados: instruções lógicas e aritméticas;
- Armazenamento de dados: movimentação de dados para dentro e fora dos registradores ou memória;
- Movimentação de dados: instruções de E/S;
- Controle: instruções de teste e desvios.

O projeto de um conjunto de instruções possui algumas questões básicas, porém extremamente importantes, entre elas:

- Repertório de operações: quantas e quais serão oferecidas, qual a complexidade dessas instruções;
- Tipos de dados: quais os tipos de dados sobre os quais as operações são realizadas (endereços, números, caracteres e dados lógicos);
- Formato de instrução: tamanho da instrução (em bits), número de endereços, tamanho de cada campo.
- Registradores: quantos registradores poderão ser referenciados pelas instruções;
- Endereçamento: os modos de especificar o endereço de um operando.

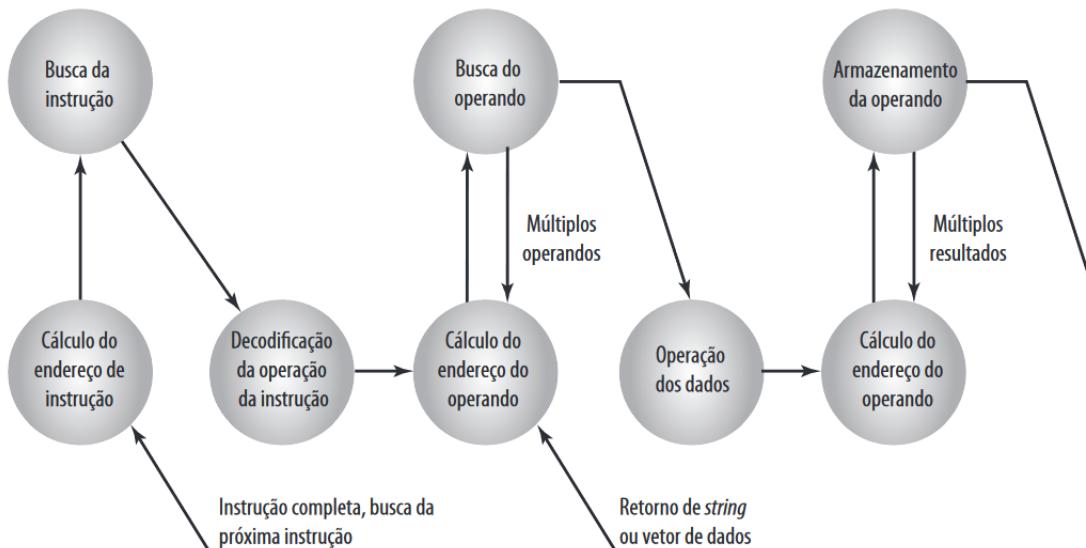
A forma como uma instrução se refere ao operando é denominada modo de endereçamento, as instruções podem conter o valor atual do operando ou uma referência para o endereço desse operando, há diversos modos de endereçamento com o intuito de aumentar a quantidade de locais que possam ser referenciados, conforme segue.

- Endereçamento Imediato: o valor do operando está contido na própria instrução.

- Endereçamento Direto: o campo de endereço possui o endereço efetivo do operando.
- Endereçamento Indireto: o campo de endereço possui um endereço de memória, esse endereço de memória contém o endereço efetivo do operando na memória.
- Endereçamento por Registrador: o campo de endereço possui o endereço de um registrador que contém o valor do operando.
- Endereçamento Indireto por registrador: o campo de endereço possui o endereço de um registrador, esse registrador contém o endereço efetivo do operando na memória.
- Endereçamento por Deslocamento: combina o endereçamento direto e o indireto por registradores, a instrução possui dois campos de endereço, sendo um deles explícito e o outro contém refere-se a um registrador, os conteúdos são adicionados e produzem o endereço efetivo. Esse tipo de endereçamento é subdividido em endereçamento relativo (o campo do registrador é o PC), endereçamento por registrador base (o registrador base aponta um endereço da memória e o outro contém um deslocamento), indexação(nesse caso o registrador referenciado contém o deslocamento e o outro campo um endereço da memória principal).
- Endereçamento Pilha: é uma forma de endereçamento implícito, as instruções da máquina não precisam incluir uma referência de memória e sim operam no topo da pilha.

A execução de uma instrução de máquina é dividida em pequenas micro-operações, conforme o diagrama de estado de ciclo de instrução da figura 8:

Figura 8 – Diagrama de estado do ciclo de instrução



Fonte: Arquitetura e Organização de Computadores (3)

As micro-operações que ocorrem na execução de uma instrução podem ser divididas da seguinte forma:

- Busca da instrução (if, do inglês *instruction fetch*): lê a instrução do seu local da memória para o processador.
- Decodificação da operação da instrução (iod, do inglês *instruction operation decoding*): determina a ação a ser realizada e os operandos envolvidos.
- Cálculo do endereço do operando (oac, do inglês *operation address calculation*): se necessário, calcula o endereço do operando.
- Busca do operando (of, do inglês *operation fetch*): busca o operando na memória ou E/S.
- Operação dos dados (do, do inglês *data operation*): realiza a ação indicada na instrução.
- Armazenamento do operando (os, do inglês *operand store*): escreve o resultado na memória ou envia para a E/S.

3.6 Arquitetura MIPS

Um MIPS, ou Microprocessador sem Estágios Intertravados de Pipeline, do inglês *Microprocessor without Interlocked Pipeline Stages*, é uma arquitetura desenvolvida pela *MIPS Technology Inc* que utiliza uma abordagem RISC.

Há diversas implementações diferentes para essa arquitetura, porém aquela que possui maior relevância e é utilizada como base para esse projeto é a arquitetura MIPS de 32 bits, monociclo, ou seja, cada instrução é executada em um único ciclo de *clock*, utilizando a arquitetura Harvard, ou seja, uma memória para armazenar dados e outra para armazenar instruções.

A implementação de um processador com arquitetura MIPS ocorre através da união das várias unidades funcionais listas a seguir, essa união se faz com base em um conjunto de instruções que se deseja atender.

3.6.1 PC

O contador de programa, do inglês *program counter*, é um registrador de uso específico cuja função é armazenar o endereço da instrução sendo executada. Trata-se de um registrador com 32 bits, que, teoricamente, seria capaz de referenciar 2^{32} posições da memória, entretanto, na arquitetura MIPS, a memória é dividida em bytes, ou seja, cada

palavra requer 4 posições da memória, dessa forma o PC é capaz de referenciar $2^{32}/4$ posições da memória, que equivale a 2^{30} posições da memória.

A implementação do PC consiste de um registrador com apenas uma entrada e uma saída.

3.6.2 ULA

A Unidade Lógica Aritmética, também conhecida como ULA ou ALU, do inglês *Arithmetic Logic Unit*, é o hardware que realiza operações matemáticas e lógicas no processador.

Com exceção da classe de instrução *jump*, todas as demais utilizam a ULA após a leitura dos registradores. No caso das instruções de referência à memória usa-se a ULA para o cálculo de endereço, no caso das operações lógicas ou aritméticas usa-se para a execução da operação ou comparação de desvios.

A ULA é implementada usando lógica combinacional e elementos de estado internos, possui como entradas os dois operandos e sinais de controle, possui como saídas o resultado da operação e uma *flag* para quando o resultado da operação for nulo.

3.6.3 Memória

A arquitetura MIPS implementa o conceito da arquitetura Harvard em sua memória, ou seja, há duas memórias separadas para armazenar dados e instruções, ambas as memórias são distribuídas em bytes e endereçadas a cada 4 bytes.

A implementação da memória de instrução consiste em uma matriz com 32 colunas e quantas linhas forem necessárias, com uma entrada e uma saída apenas. A implementação da memórias de dados consiste de uma matriz com 8 colunas e quantas linhas forem necessárias ao projeto, sendo que cada 4 linhas consiste em uma posição da memória, há duas entradas, uma de endereço e outra de dados, e sinais de controle para escrita e leitura que nunca são simultaneamente setados para evitar conflito na leitura/gravação de informações.

3.6.4 Extensor de bits

O campo imediato em algumas instruções contém um número de 16 bits que deve ser somado com um valor de 32 bits de um registrador, para que isso seja possível é necessário converter esse número de 16 bits em um número de 32 bits através de um extensor de bits.

A unidade de extensão de sinal possui uma entrada de 16 bits que tem o seu sinal estendido para que um resultado de 32 bits apareça na saída.

3.6.5 Banco de registradores

Um banco de registradores consiste em uma coleção de registradores que podem ser lidos ou escritos quando um número de registrador é fornecido.

O banco de registradores na arquitetura MIPS contém 32 registradores de propósito geral de 32 bits cada, os grupos de 32 bits ocorrem tão frequentemente na arquitetura MIPS que recebem o nome de palavra (*word*).

A figura 9 apresenta a distribuição geral dos registradores na arquitetura MIPS. Alguns registradores foram omitidos pois são reservados para uso específico, como o 1 (\$at), que é reservado para o montador, e o 26 e 27 (\$k0 e \$k1), que são reservados para o sistema operacional.

Figura 9 – Convenções de registradores MIPS

Nome	Número do registrador	Uso	Preservado na chamada?
\$zero	0	O valor constante 0	n.a.
\$v0-\$v1	2–3	Valores para resultados e avaliação de expressões	não
\$a0-\$a3	4–7	Argumentos	não
\$t0-\$t7	8–15	Temporários	não
\$s0-\$s7	16–23	Valores salvos	sim
\$t8-\$t9	24–25	Mais temporários	não
\$gp	28	Ponteiro global	sim
\$sp	29	Stack pointer	sim
\$fp	30	Frame pointer	sim
\$ra	31	Endereço de retorno	sim

Fonte: Organização e Projeto de Computadores (2)

3.6.6 Unidade de Controle

Uma unidade de controle possui a instrução como entrada e é usada para determinar os sinais das linhas de controle para as unidades funcionais e dois dos multiplexadores. A implementação da unidade de controle consiste de uma porção de lógica combinacional e elementos de estado internos, recebe a instrução como entrada e possui diversos sinais de controle na saída.

3.6.7 Conjunto de instruções

Na arquitetura MIPS, todas as instruções do processador são codificadas em um formato de uma única palavra de 32 bits e todas as operações de dados são de registrador para registrador. Os três formatos de instrução têm em comum os opcodes e referências aos registradores, isso simplifica a decodificação das instruções.

Devido ao compromisso da arquitetura MIPS em manter as instruções sempre com o mesmo tamanho, e à necessidade de campos maiores em algumas instruções e mais campos em outras, há três formatos diferentes para as instruções na arquitetura MIPS.

3.6.7.1 Tipo R de registrador

Figura 10 – Formato da instrução tipo R



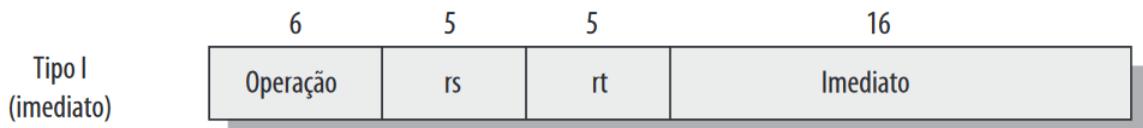
Fonte: Arquitetura e Organização de Computadores ([3](#))

As instruções do tipo R possuem 6 campos diferentes e são utilizadas para operações lógicas e aritméticas, cada campo possui um número de bits e uma função conforme descrito abaixo:

- operação: operação básica da instrução, conhecido também como *opcode*;
- rs: registrador do primeiro operando fonte;
- rt: registrador do segundo operando fonte;
- rd: registrador do operando destino, recebe o resultado da operação;
- deslocamento / shamt: do inglês *shift amount*, contém a quantidade de deslocamento para a instrução shift;
- função / funct: do inglês *function*, ou código de função, seleciona a variante específica da operação no campo *operação*.

3.6.7.2 Tipo I de imediato

Figura 11 – Formato da instrução tipo I



Fonte: Projeto e Organização de Computadores ([2](#))

As instruções do tipo I possuem 4 campos diferentes e são utilizadas para instruções imediatas ou de transferência de dados, cada campo possui um número de bits e uma função conforme descrito abaixo:

- operação: operação básica da instrução, conhecido também como *opcode*;
- rs: registrador do primeiro operando fonte;
- rt: registrador de destino;
- imediato (constante ou endereço): pode conter uma constante utilizada como imediato em uma operação aritmética ou ser somada ao endereço base no registrador rs para alcançar uma maior parcela da memória.

3.6.7.3 Tipo J de *jump*

Figura 12 – Formato da instrução tipo J



Fonte: Projeto e Organização de Computadores ([2](#))

As instruções do tipo J possuem 2 campos diferentes e são utilizadas para distinguir entre desvios condicionais e incondicionais, cada campo possui um número de bits e uma função conforme descrito abaixo:

- operação: operação básica da instrução, conhecido também como *opcode*;
- alvo: contém o endereço alvo do salto. A figura [13](#) apresenta o conjunto de instruções da arquitetura MIPS.

Figura 13 – Conjunto de instruções da arquitetura MIPS

Categoría	Instrução	Exemplo	Significado	Comentários
Aritmética	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Três operandos; dados nos registradores
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Três operandos; dados nos registradores
	add immediate	addi \$s1,\$s2, <u>20</u>	$\$s1 = \$s2 + 20$	Usada para somar constantes
Transferência de dados	load word	lw \$s1, <u>20(\$s2)</u>	$\$s1 = \text{Memória}[\$s2 + 20]$	Dados da memória para o registrador
	store word	sw \$s1, <u>20(\$s2)</u>	$\text{Memória}[\$s2 + 20] = \$s1$	Dados do registrador para a memória
	load half	lh \$s1, <u>20(\$s2)</u>	$\$s1 = \text{Memória}[\$s2 + 20]$	Halfword da memória para registrador
	load half unsigned	lhu \$s1, <u>20(\$s2)</u>	$\$s1 = \text{Memória}[\$s2 + 20]$	Halfword da memória para registrador
	store half	sh \$s1, <u>20(\$s2)</u>	$\text{Memória}[\$s2 + 20] = \$s1$	Halfword de um registrador para memória
	load byte	lb \$s1, <u>20(\$s2)</u>	$\$s1 = \text{Memória}[\$s2 + 20]$	Byte da memória para registrador
	load byte unsigned	lbu \$s1, <u>20(\$s2)</u>	$\$s1 = \text{Memória}[\$s2 + 20]$	Byte da memória para registrador
	store byte	sb \$s1, <u>20(\$s2)</u>	$\text{Memória}[\$s2 + 20] = \$s1$	Byte de um registrador para memória
	load linked word	ll \$s1, <u>20(\$s2)</u>	$\$s1 = \text{Memória}[\$s2 + 20]$	Carrega word como 1ª metade do swap atômico
	store condition, word	sc \$s1, <u>20(\$s2)</u>	$\text{Memória}[\$s2 + 20] = \$s1; \$s1 = 0 \text{ or } 1$	Armazena word como 2ª metade do swap atômico
Lógica	load upper immed.	lui \$s1, <u>20</u>	$\$s1 = 20 * 2^{16}$	Carrega constante nos 16 bits mais altos
	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Três operadores em registrador; AND bit a bit
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \$s3$	Três operadores em registrador; OR bit a bit
	nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim (\$s2 \$s3)$	Três operadores em registrador; NOR bit a bit
	and immediate	andi \$s1,\$s2, <u>20</u>	$\$s1 = \$s2 \& 20$	AND bit a bit registrador com constante
	or immediate	ori \$s1,\$s2, <u>20</u>	$\$s1 = \$s2 20$	OR bit a bit registrador com constante
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 << 10$	Deslocamento à esquerda por constante
Desvio condicional	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 >> 10$	Deslocamento à direita por constante
	branch on equal	beq \$s1,\$s2,25	if ($\$s1 == \$s2$) go to PC + 4 + 100	Testa igualdade; desvio relativo ao PC
	branch on not equal	bne \$s1,\$s2,25	if ($\$s1 != \$s2$) go to PC + 4 + 100	Testa desigualdade; relativo ao PC
	set on less than	slt \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compara menor que; usado com beq, bne
	set on less than unsigned	sltu \$s1,\$s2,\$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compara menor que sem sinal
	set less than immediate	slti \$s1,\$s2, <u>20</u>	if ($\$s2 < 20$) $\$s1 = 1$; else $\$s1 = 0$	Compara menor que constante
Desvio incondicional	set less than immediate unsigned	sltiu \$s1,\$s2, <u>20</u>	if ($\$s2 < 20$) $\$s1 = 1$; else $\$s1 = 0$	Compara menor que constante sem sinal
	jump	j 2500	go to 10000	Desvia para endereço de destino
	jump register	jr \$ra	go to \$ra	Para switch e retorno de procedimento
	jump and link	jal 2500	$\$ra = \text{PC} + 4$; go to 10000	Para chamada de procedimento

Fonte: Projeto e Organização de Computadores (2)

3.6.8 Modos de endereçamento

Há cinco modos de endereçamento diferentes utilizados na arquitetura MIPS, conforme descrição a seguir:

- Endereçamento imediato: o operando é uma constante dentro da própria instrução;
- Endereçamento por registrador: o operando está em um registrador;
- Endereçamento por deslocamento de base: o operando está em um local da memória cujo endereço é a soma de um registrador e uma constante que estão na instrução;
- Endereçamento pseudodireto: o endereço do salto são os 26 bits da instrução concatenados com os bits mais altos do PC.

A figura 14 ilustra esses cinco modos de endereçamento utilizados na arquitetura MIPS para facilitar a visualização.

Figura 14 – Modos de endereçamento utilizados na arquitetura MIPS.

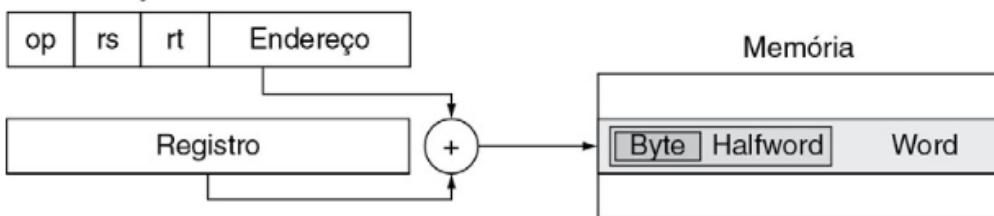
1. Endereçamento imediato



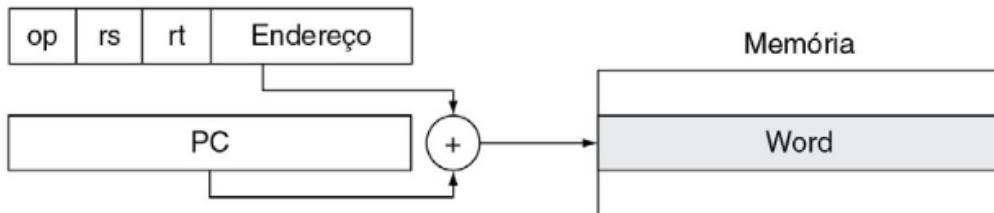
2. Endereçamento em registrador



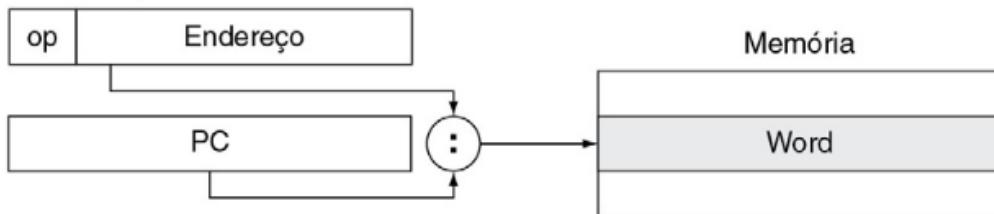
3. Endereçamento de base



4. Endereçamento relativo ao PC



5. Endereçamento pseudodireto

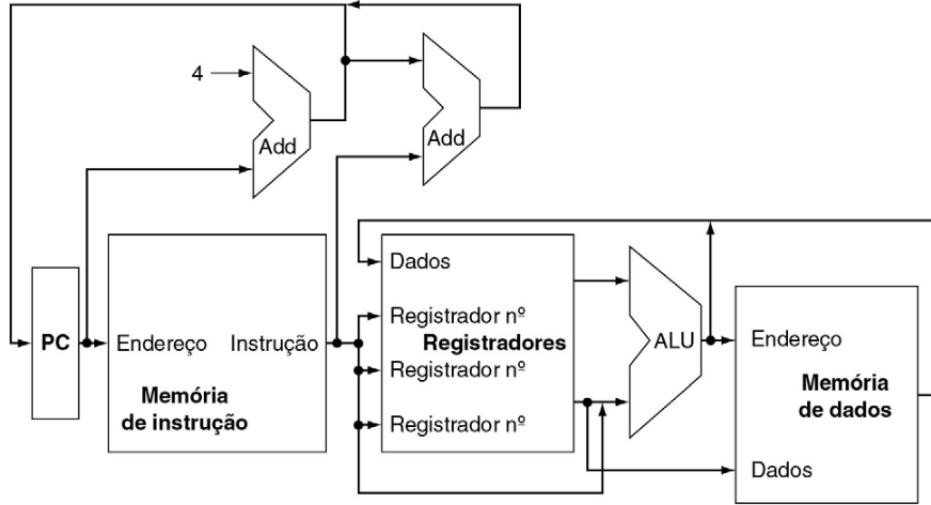


Projeto e Organização de Computadores (2)

3.6.9 Caminho de dados

O caminho de dados do processador é caracterizado pela união das unidades funcionais em função das instruções que devem ser executadas. A figura 15 apresenta cinco unidades funcionais elementares do MIPS, o PC, a memória de instrução, o banco de registradores, a ULA e a memória de dados, além desses há dois somadores menores responsáveis por somar 4 ao PC e somar um imediato de desvio ao endereço da próxima instrução quando necessário. Todas as unidades funcionais são interconectadas em função de suas necessidades de entradas e saídas.

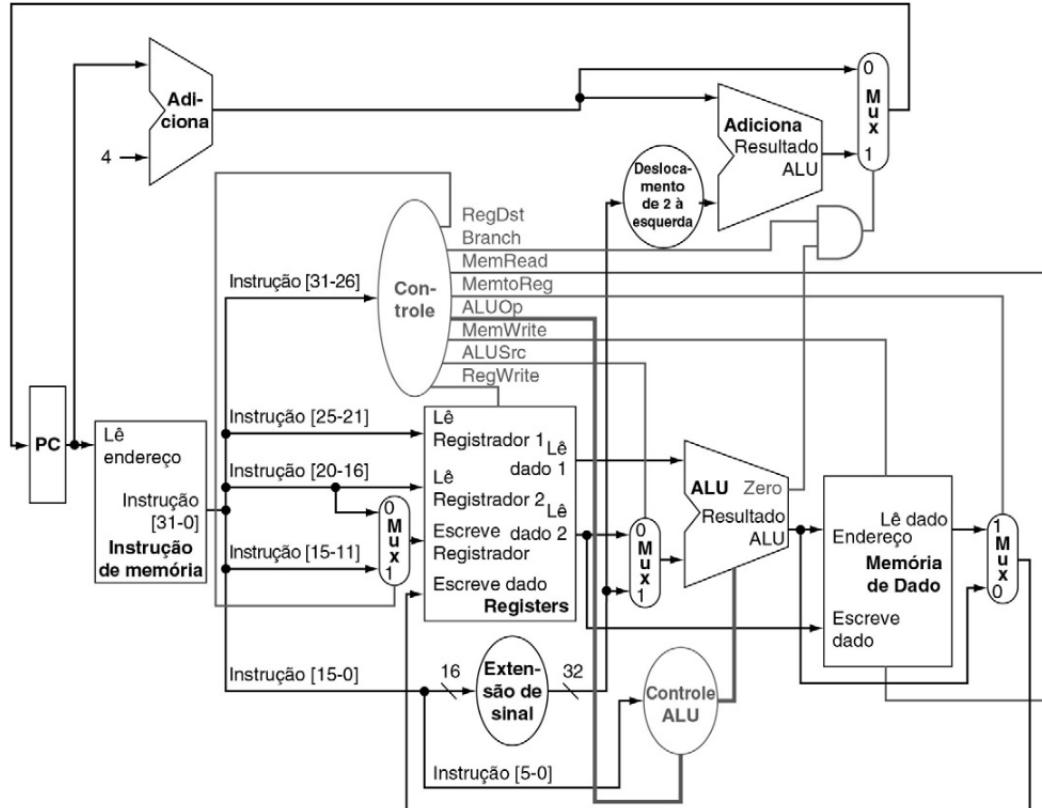
Figura 15 – Caminho de dados simplificado da arquitetura MIPS, sem as linhas controle.



Fonte: Projeto e Organização de Computadores (2)

A figura 16 apresenta um caminho de dados simples com a unidade de controle e os sinais de controle na arquitetura MIPS.

Figura 16 – Caminho de dados simplificado da arquitetura MIPS, com as linhas de controle.



Fonte: Projeto e Organização de Computadores (2)

3.7 Quartus Prime e Linguagem de Descrição de Hardware

O software de desenvolvimento Quartus Prime foi desenvolvido pela Intel e permite a análise e confecção de projetos em linguagem de descrição de hardware (HDL, do inglês *Hardware Description Language*).

Uma linguagem de descrição de hardware é uma ferramenta utilizada para descrição formal e projeto de circuitos eletrônicos. São linguagens que descrevem o funcionamento de um circuito e permitem sua simulação, sendo muito úteis em projetos de sistemas computacionais. Há inúmeras linguagens de descrição de hardware disponíveis, dentre elas está a linguagem Verilog que é muito utilizada para modelar sistemas eletrônicos e é utilizada neste projeto.

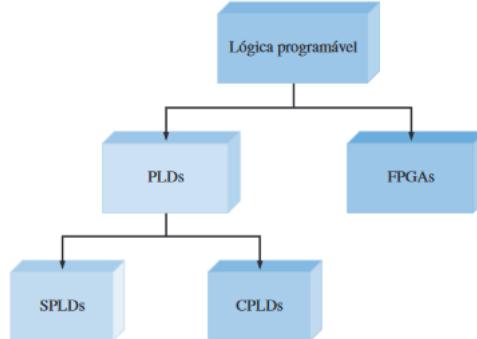
3.8 Arranjo de portas programáveis por campo - FPGA

Um dispositivo lógico programável é um componente eletrônico utilizado para construir circuitos digitais. Diferente das portas lógicas, que possuem funções fixas, esses dispositivos possuem função indefinida na fabricação, ou seja, antes que sejam utilizados esses dispositivos devem ser programados.

Segundo Floyd (4), dispositivos de lógica programável podem ser usados para realizar funções lógicas especificadas pelo fabricante ou pelo usuário. Duas vantagens dos dispositivos de lógica programável são que os dispositivos programáveis ocupam menos espaço na placa para uma mesma quantidade de lógica se comparados aos circuitos integrados (CIs) de função fixa e que projetos podem ser alterados com facilidade, sem alteração no hardware ou substituição de componentes.

Os dispositivos lógicos programáveis podem ser divididos em duas categorias principais de lógica programável, o dispositivo de lógica programável (*programmable logic device* - PLD) e o arranjo de portas programáveis por campo (*field programmable gate array* - FPGA), conforme a figura 17. Somente os FPGAs são abordados neste documento pois o projeto utiliza este dispositivo na lógica programável.

Figura 17 – Categorias entre os dispositivos lógicos programáveis.

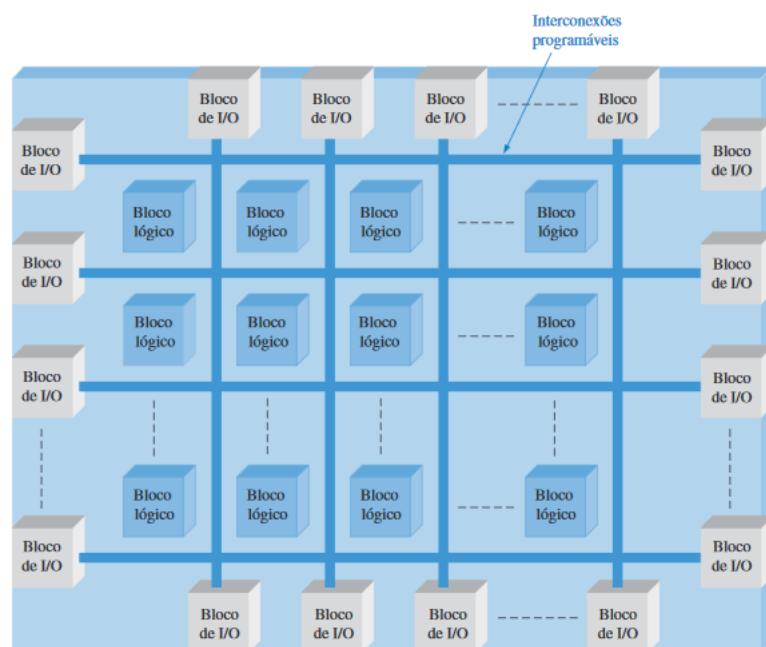


Fonte: Sistemas Digitais (4)

Um FPGA é composto por três elementos básicos, os blocos lógicos, as interconexões programáveis e os blocos de entrada/saída (I/O - *in/out*), conforme a figura 18.

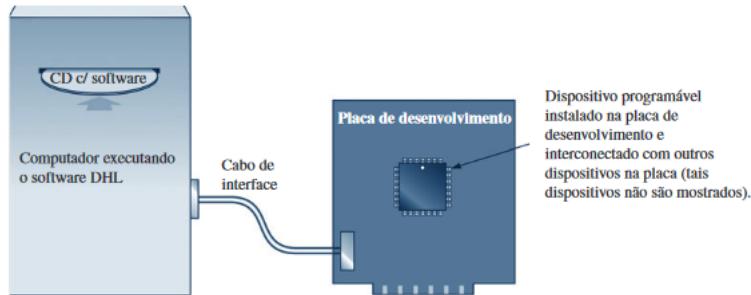
Tanto os PLD's quanto os FPGAs podem ser interpretados como "quadros brancos" nos quais são implementados circuitos fazendo uso de determinado processo. Esse processo requer um software de desenvolvimento instalado em um computador para a implementação de um projeto em um chip programável e o computador deve ter interface com uma placa de desenvolvimento contendo o dispositivo programável, conforme a figura 19.

Figura 18 – Estrutura básica de um FPGA.



Fonte: Sistemas Digitais (4)

Figura 19 – Configuração básica de um sistema para programação de PLD ou FPGA.

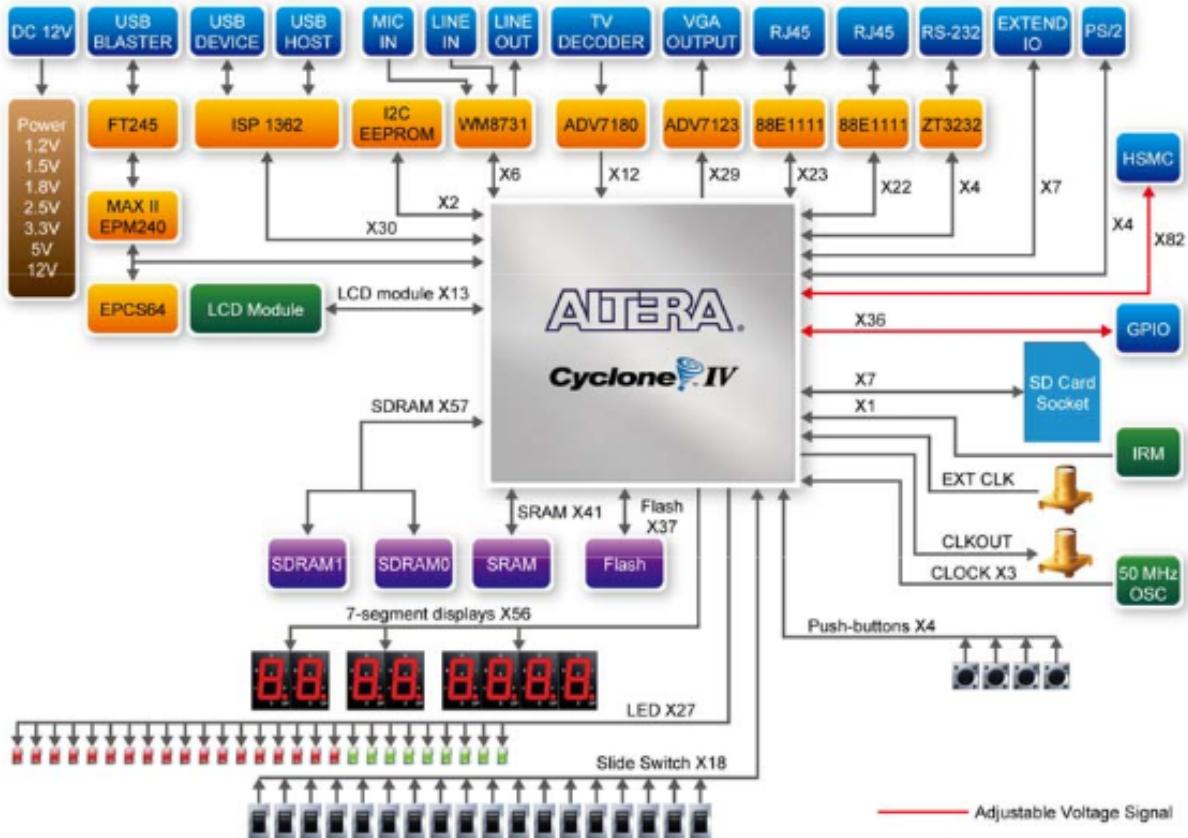


Fonte: Sistemas Digitais (4)

O kit FPGA utilizado neste projeto é o DE2-115 produzido pela Altera que utiliza o dispositivo FPGA Cyclone IV EP4CE115F29 e conta com a seguinte configuração:

- Dispositivo Altera Cyclone® IV 4CE115 FPGA;
- Dispositivo serial de configuração Altera EPICS64;
- Conexão USB para programação;
- Memória SRAM de 2MB;
- Duas memórias SRAM de 64MB;
- Memória Flash de 8MB;
- Soquete para cartão SD;
- 4 botões de pressão;
- 18 chaves do tipo *slide*;
- 18 LEDs vermelhos;
- 9 LEDs verdes;
- Fonte com sinal oscilatório de 50MHz;
- Conectores de diversos tipos;
- Módulo display LCD 16x2.

Figura 20 – Diagrama de blocos do kit FPGA.



Fonte: Sistemas Digitais (5)

4 Desenvolvimento

4.1 Arquitetura base

A arquitetura base deste projeto é a arquitetura MIPS-32 monociclo, que trata dos processadores MIPS de 32 bits. Trata-se de uma arquitetura com conjunto de instruções reduzido, ou seja, RISC. Optou-se pela implementação de um processador monociclo, ou seja, a execução de cada instrução utilizará um ciclo de *clock*.

Conforme apresentado na fundamentação teórica, a arquitetura MIPS-32 utiliza arquitetura Harvard, com memória de dados e instruções separadas e contém três formatos de instruções (R, S e J).

4.2 Conjunto de instruções

A tabela 1 apresenta as instruções escolhidas para compor o conjunto de instruções do projeto, juntamente com a descrição de cada uma delas. A proposta é que essa coleção de instruções seja capaz de solucionar problemas de lógica de computação, de acordo com as limitações do projeto no que diz respeito à memória, tipo de dados e capacidade de processamento.

De acordo com a tabela 1 pode-se observar que o conjunto de instruções contém 29 instruções, a coluna Mnem. contém o mnemônico de cada instrução, ou seja, a representação dessa instrução em linguagem de representação Assembly, a coluna Operação apresenta uma descrição simbólica da instrução, a coluna Form. indica qual o formato de instrução utilizado, a coluna Tipo apresenta as classes das instruções e a coluna Função descreve a função de cada instrução.

Foram utilizados três formatos de instrução, assim como na arquitetura base, dos tipos R, I e J. Os campos das instruções atuam exatamente como nas instruções da arquitetura MIPS apresentadas na seção de fundamentação teórica.

4.2.1 Formato de instrução

As instruções possuem o tamanho fixo de 32 bits e são divididas em três tipos, conforme a arquitetura MIPS-32. O número de endereços e tamanho de cada campo são iguais aos da arquitetura MIPS-32 apresentados na fundamentação teórica, conforme as figuras 10, 11 e 12.

Portanto, seguem as características do conjunto de instruções do projeto:

- Repertório de operação: contém 29 instruções, vide tabela 1;
- Tipos de dados: opera com valores inteiros com até 31 bits, pois um dos bits armazena o sinal;
- Registradores: os campos de referência aos registradores possuem 5 bits cada, isso significa que o projeto possui 2^5 registradores, ou seja, são 32 registradores de propósito geral;

4.2.2 Modos de endereçamento

O projeto comporta quatro modos de endereçamento diferentes, apenas o endereçamento relativo ao PC, que é utilizado na arquitetura base MIPS monociclo, não será utilizado neste projeto, sendo eles:

- Imediato;
- Registrador;
- Deslocamento por registrador base;
- Direto.

A figura 14 descreve os modos de endereçamento.

Tabela 1 – Conjunto de instruções elaborado para o projeto

Instrução	Mnem.	Operação	Form.	Tipo	Função
entrada	in	R[rt] <= input	R	(E/S)	recebe dados de uma entrada externa
saída	out	output <= R[rs]	R	(E/S)	emite dados para uma saída externa
adição	add	R[rd] <= R[rs] + R[rt]	R	aritmético	operação de soma
subtração	sub	R[rd] <= R[rs] - R[rt]	R	aritmético	operação de subtração
multiplicação	mul	R[rd] <= R[rs] * R[rt]	R	aritmético	operação de multiplicação
divisão	div	R[rd] <= R[rs] / R[rt]	R	aritmético	operação de divisão
menor que	slt	R[rd] <= (R[rs] < R[rt])	R	desvio condicional	configura nível alto se menor que
maior que	sgt	R[rd] <= (R[rs] > R[rt])	R	desvio condicional	configura nível alto se maior que
ignal a	set	R[rd] <= (R[rs] == R[rt])	R	desvio condicional	configura nível alto se igual
jump register	jr	PC <= R[rs]	R	desvio incondicional	salta para o endereço em rs
and	and	R[rd] <= R[rs] & R[rt]	R	lógico	operação lógica and
or	or	R[rd] <= R[rs] R[rt]	R	lógico	operação lógica or
resto	mod	R[rd] <= R[rs] % R[rt]	R	lógico	retorna o resto da divisão
xor	xor	R[rd] <= R[rs] ^ R[rt]	R	lógico	operação lógica xor
not	not	R[rd] <= ~R[rs]	R	lógico	operação lógica de negação
move	move	R[rd] <= R[rs]	R	transferência	move o dado de rs para rd
desloca esquerda	sll	R[rd] <= R[rs] << shamt	R	deslocamento	ajusta a frequência de clock para o valor do imediato
desloca direita	srl	R[rd] <= R[rs] >> shamt	R	deslocamento	salta para o endereço em IM
ajuste de clock	clkadj	ajusta o clock conforme imediato	J	controle	salta para a frequência de clock para o valor do imediato
jump	j	PC <= IM	J	desvio incondicional	carrega dado da memória
jump and link	jal	R[ra]<= PC+1 e PC <= IM	J	desvio incondicional	armazena dado na memória
carregar	load	R[rd] <= M[R[rs]+IM]	I	acesso à memória	operação de soma com operando imediato
armazenar	store	M[R[rs]+IM] <= R[rs]	I	acesso à memória	operação de subtração com operando imediato
adição imediata	addi	R[rd] <= R[rs] + IM	I	aritmético	desvia se igual
subtração imediata	subi	R[rd] <= R[rs] - IM	I	aritmético	desvia se diferente
desvie se igual	beq	if(R[rs] == R[rt]) PC=PC+1+IM	I	desvio condicional	carrega o imediato em rd
desvie se diferente	bne	if(R[rs] > R[rt]) PC=PC+1+IM	I	desvio condicional	carrega o imediato nos bits mais significativos de rs
carregar imediato	loadi	R[rd] <= IM	I	transferência	desloca para a esquerda
carregar superior imediato	lui	R[rd] <= IM << 16	I	transferência	desloca para a direita

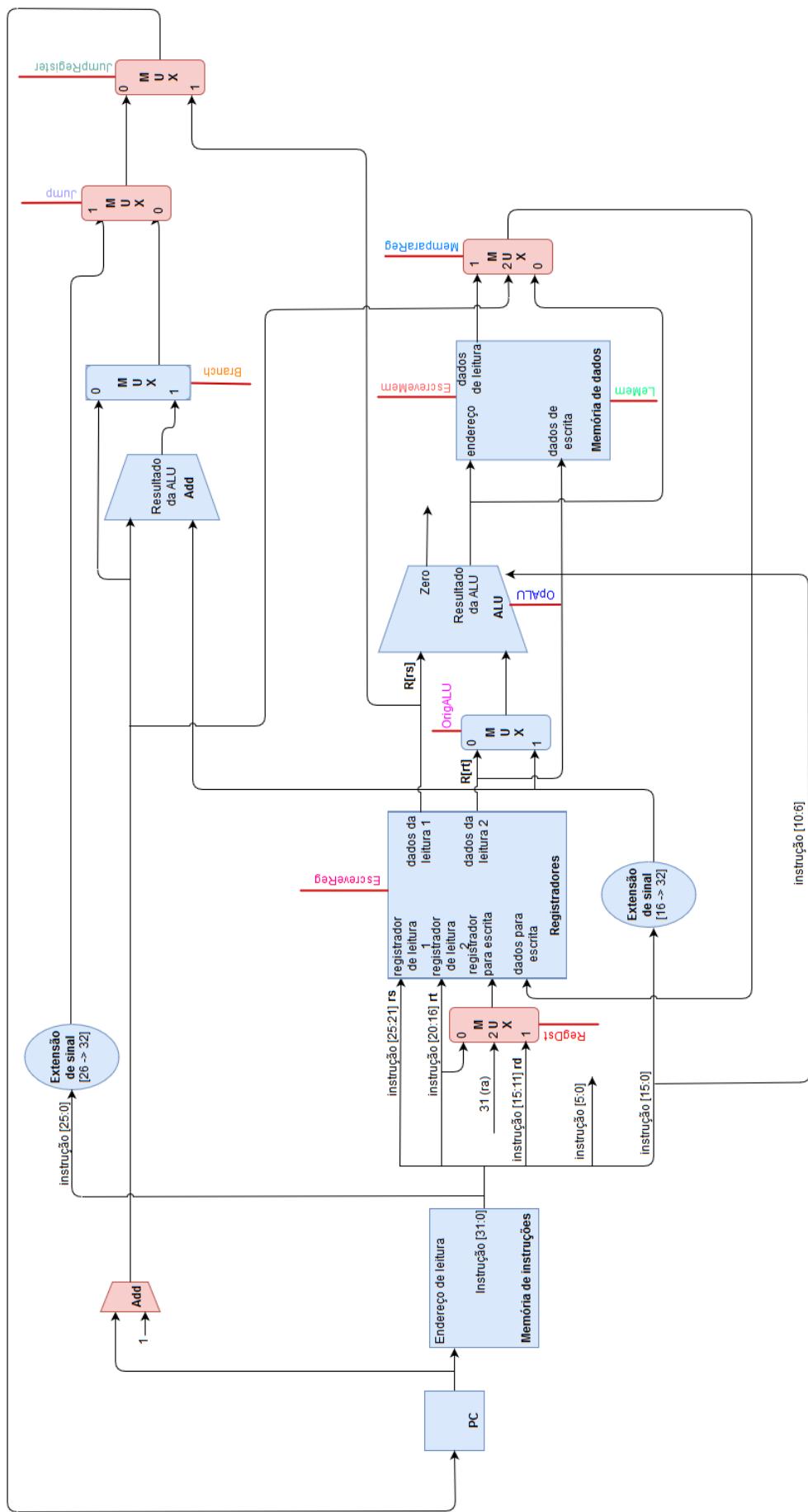
Fonte: elaborado pela autora.

4.3 Caminho de dados

O caminho de dados é bastante similar ao da arquitetura base MIPS-32, porém algumas alterações foram realizadas. A figura 21 apresenta o caminho de dados desenvolvido para o projeto. Os elementos em azul estão conforme as unidades funcionais da arquitetura MIPS-32, os elementos em vermelho sofreram alterações. Foram executadas as seguintes alterações:

- Incremento do PC: na arquitetura base soma 4, no projeto soma 1. Essa alteração justifica-se pelo fato de que a memória será distribuída em 32 bits por linha, não em 1 byte por linha conforme a arquitetura base.
- Multiplexador que fornece o registrador para escrita no banco de registradores: uma terceira entrada foi inserida e contém o número 31 que corresponde ao registrador conhecido como ra, essa alteração permite que o registrador ra armazene o endereço da próxima instrução no caso da execução de uma instrução *jump and link*, que salta para uma nova instrução porém retorna para a instrução anterior. Esse seletor de dados é controlado via sinal de controle *RegDst*.
- Inserção de multiplexador (1): esse novo multiplexador permite selecionar qual a próxima instrução a ser executada e é utilizado para permitir a instrução *jump* ao fornecer o endereço da nova instrução que está armazenado nos 26 bits menos significativos da instrução. Esse seletor de dados é controlado via sinal de controle *Jump*.
- Inserção de multiplexador (2): esse novo multiplexador permite selecionar qual a próxima instrução a ser executada e é utilizado para permitir a instrução *jump register* ao fornecer o endereço da nova instrução que está armazenada em rs. Esse seletor de dados é controlado via sinal de controle *JumpRegister*.
- Multiplexador que recebe a saída da memória de dados: inserção de uma terceira entrada que contém o endereço da próxima instrução do PC e é utilizada para permitir a execução da instrução *jump and link*. Esse seletor de dados é controlado pelo sinal de controle *MemparaReg*.

Figura 21 – Caminho de dados do projeto com sinal de controle



Fonte: elaborado pela autora.

4.4 Verificação das instruções

A figura 22 apresenta o caminho de dados com as unidades funcionais necessárias às instruções do tipo R em vermelho, a figura 23 apresenta as unidades funcionais necessárias às instruções do tipo I em vermelho e a figura 24 apresenta as unidades funcionais necessárias às instruções do tipo J em vermelho.

Basicamente, todas as instruções utilizam o PC, o somador de 1 ao endereço do PC e a memória de instruções para ter acesso às instruções a serem executadas.

A função do multiplexador controlado pelo sinal *RegDst* é determinar o registrador que receberá os dados para escrita. Nas instruções do tipo R ele determina que o registrador rd receberá os dados, nas instruções tipo I ele seleciona entre rd e rt em função da instrução e nas instruções do tipo J ele determina que seja o ra caso a instrução seja *jump and link*.

O banco de registradores é usado por quase todas as instruções, apenas a instrução jump não o utiliza.

O multiplexador controlado pelo sinal *OrigALU* é responsável por determinar o segundo dado fornecido para a ULA, que pode ter origem no registrador rt nas instruções do tipo R ou pode ser o valor imediato das instruções do tipo I. A ULA é utilizada nas instruções do tipo R para operações lógicas e aritméticas e é utilizada para cálculo de endereços ou soma com imediatos nas instruções do tipo I.

A memória de dados é utilizada nas instruções do tipo I quando é necessário coletar ou armazenar dados na memória.

O extensor de sinal de 26 para 32 bits é utilizado nas instruções do tipo J para estender o tamanho do endereço.

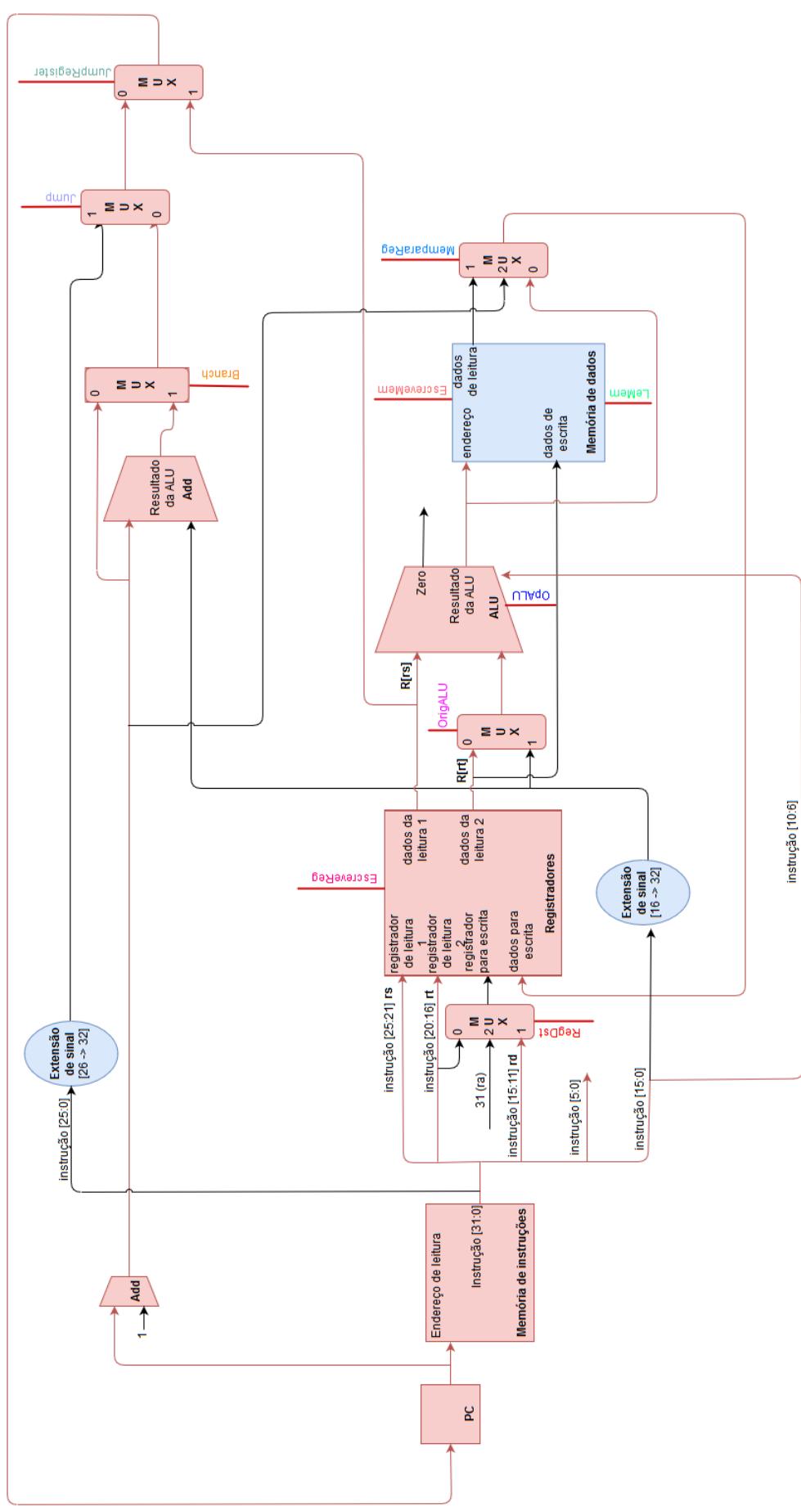
O extensor de sinal de 16 para 32 bits é utilizado nas instruções do tipo I para estender o tamanho dos imediatos para que possam ser operados aritmeticamente com outros dados de 32 bits.

O multiplexador controlado pelo sinal *MemparaReg* determina qual dado será enviado ao banco de registradores e é utilizado nos três tipos para selecionar entre a saída da ULA, a saída da memória de dados ou o valor de PC+1.

Os multiplexadores controlados pelos sinais *Branch*, *Jump* e *JumpRegister* selecionam entre os sinais que serão utilizados como próxima instrução.

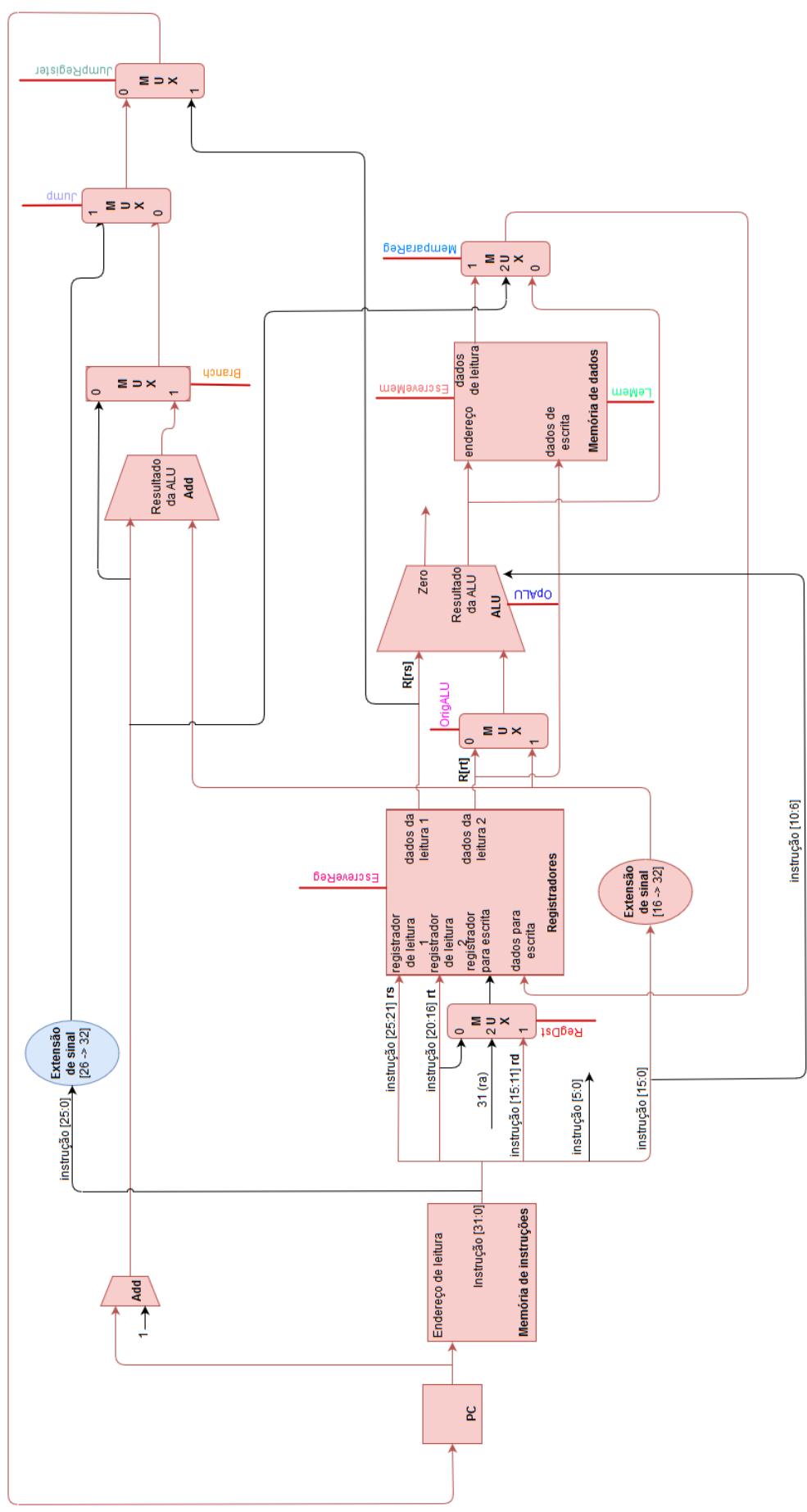
Há um pequeno somador responsável por incrementar o PC de acordo com a necessidade nas instruções R e I.

Figura 22 – Caminho de dados do projeto com sinais de controle e as unidades funcionais utilizadas nas instruções do tipo R em vermelho



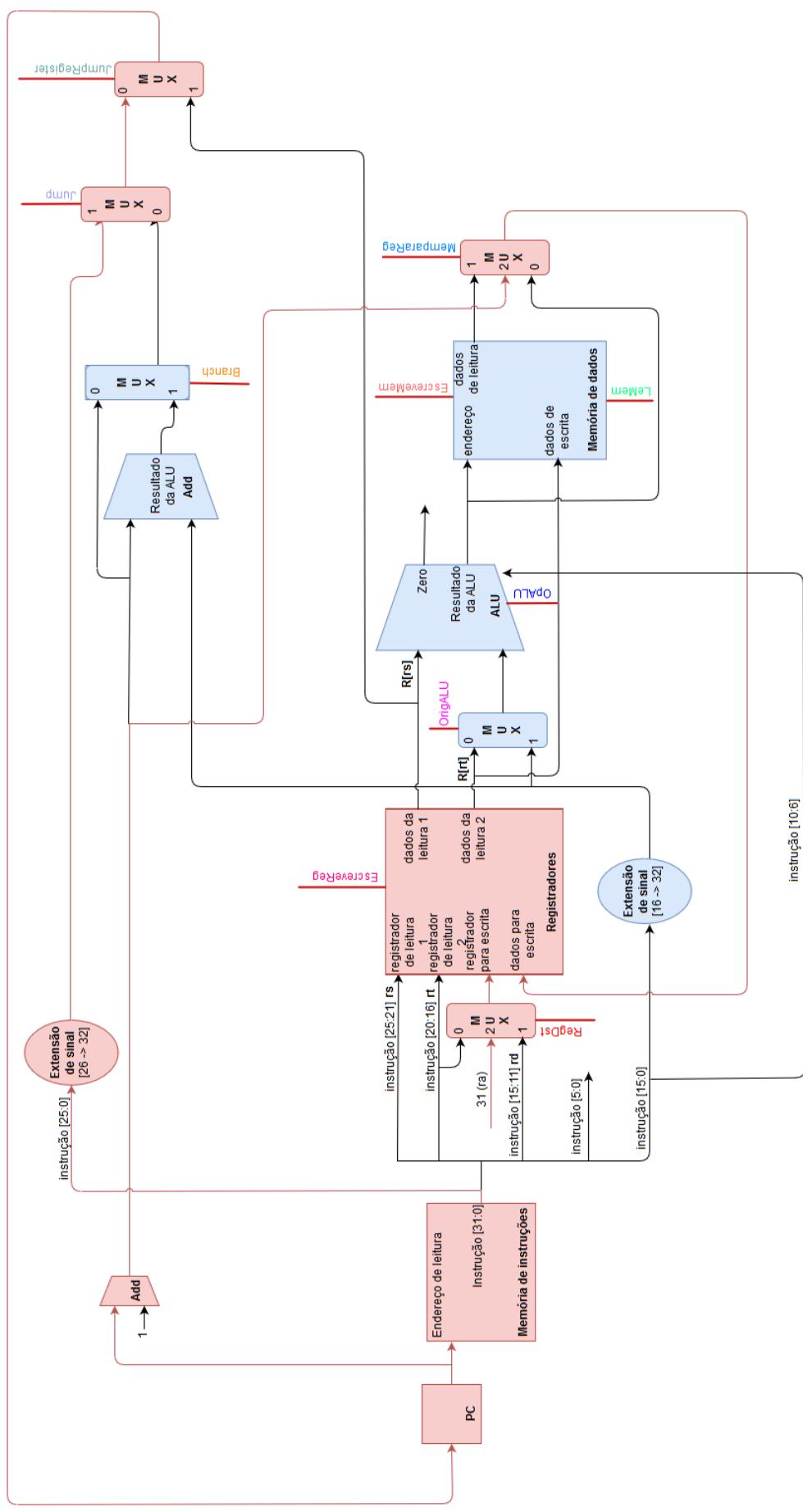
Fonte: elaborado pela autora.

Figura 23 – Caminho de dados do projeto com sinais de controle e as unidades funcionais utilizadas nas instruções do tipo I em vermelho



Fonte: elaborado pela autora.

Figura 24 – Caminho de dados do projeto com sinais de controle e as unidades funcionais utilizadas nas instruções do tipo J em vermelho



Fonte: elaborado pela autora.

4.5 Implementação em lógica programável

A implementação de cada unidade funcional em lógica programável ocorre utilizando a linguagem de descrição de hardware Verilog. A seguir estão listados os códigos e sua descrição de funcionamento.

4.5.1 PC

O PC é um registrador de 32 bits que armazena o endereço das instruções executadas, sua atualização ocorre na borda de descida do *clock*.

```

1 module PC (entrada, saida, clk, reset);
2   input [31:0] entrada; // endereco da instrucao
3   input clk; // clock do kit FPGA
4   input reset; // sinal que reinicia a execucao do codigo
5   output reg [31:0] saida; // endereco da instrucao
6   always @(negedge clk)
7   begin
8     if (reset) // o sinal de reset esta setado
9     begin
10       saida = 0; // a saida do PC eh a primeira instrucao do codigo
11     end
12     else
13     begin
14       saida = entrada; // a saida do PC corresponde a entrada do PC
15     end
16   end
17 endmodule

```

Esse módulo possui três entradas, sendo:

- *entrada* é um vetor de 32 bits que contém um endereço de instrução;
- *clk* é uma variável que contém o sinal de *clock* do processador;
- *reset* é uma variável que contém o sinal de *reset* do processador.

A *saida* é um vetor de 32 bits contendo o endereço da próxima instrução.

Na linha 9 um bloco *always* monitora a borda de descida da variável *clk*.

Quando detectada a descida, na linha 11, verifica-se se o sinal de *reset* encontra-se em nível alto e, caso esteja, a variável *saída* recebe o endereço 0 e o programa é reiniciado, caso esteja em nível baixo, a variável *saída* recebe a instrução da variável *entrada*.

4.5.2 Banco de registradores

O banco de registradores armazena dados utilizados nas instruções do processador. Neste projeto, foi construído de forma que na borda de subida do *clock* o sinal de escrita e o endereço de destino sejam guardados para que possam ser utilizados na borda de descida

do *clock*, pois o PC, a memória de instruções e a própria instrução fornecida ao banco de registradores são atualizados na borda de descida, quando o dado é gravado no registrador destino, se necessário.

```

1 module banco_reg (EscreveReg, clk, reg_leit1, reg_leit2, reg_esc, dado_esc, dado_leit1,
2   dado_leit2);
3
4   input EscreveReg; // contém o sinal que determina se haverá gravação no registrador
5   destino.
6   input clk; // contém o clock do processador
7   input [4:0] reg_leit1; // contém o endereço do registrador fonte rs
8   input [4:0] reg_leit2; // contém o endereço do registrador fonte rt
9   input [4:0] reg_esc; // contém o endereço do registrador destino rd
10  input [31:0] dado_esc; // contém o dado a ser gravado no registrador destino
11  output [31:0] dado_leit1; // contém o valor armazenado no registrador rs
12  output [31:0] dado_leit2; // contém o valor armazenado no registrador rt
13  reg [31:0] banco [31:0]; // cria uma matriz 32x32 que forma o banco de registradores
14  reg [4:0] reg_esc_anterior; // armazena o endereço do registrador destino na instrução
15  anterior
16  integer EscreveReg_anterior; // armazena o estado do sinal de gravação na instrução
17  anterior
18  initial EscreveReg_anterior = 0; // inicializa o sinal de escrita como 0;
19  always @ (negedge clk)
20  begin
21    if (EscreveReg_anterior) // verifica se a instrução anterior inclui uma gravação
22      no registrador destino
23    begin
24      banco [reg_esc_anterior] = dado_esc; // grava dados no registrador destino
25    end
26  end
27  always @ (posedge clk)
28  begin
29    reg_esc_anterior = reg_esc; // armazena o endereço de escrita da instrução
30    anterior
31    EscreveReg_anterior = EscreveReg; // armazena o sinal de escrita da instrução
32    anterior
33  end
34  assign dado_leit1 = banco [reg_leit1]; // atualiza o dado de leitura 1 do banco
35  assign dado_leit2 = banco [reg_leit2]; // atualiza o dado de leitura 2 do banco
36 endmodule

```

As 14 primeiras linhas do código contém as declarações de entradas e saídas, sendo:

- entrada *EscreveReg* contém o sinal de controle proveniente da Unidade de Controle. Esse sinal determina se o registrador destino receberá algum dado;
- entrada *clk* contém o sinal de *clock* do processador;
- entrada *reg_leit1* é um vetor de 5 bits que armazena a posição do primeiro registrador fonte;
- entrada *reg_leit2* é um vetor de 5 bits que armazena a posição do segundo registrador fonte;

- entrada *reg_esc* é um vetor de 5 bits que armazena a posição do registrador de destino;
- entrada *dado_esc* é um vetor de 32 bits que armazena o dado a ser gravado no registrador destino;
- saída *dado_leit1* contém o valor lido no primeiro registrador fonte;
- saída *dado_leit2* contém o valor lido no segundo registrador fonte;
- matriz *banco* contém uma matriz com 32 linhas de 32 bits cada que atua como banco de registradores do processador;
- vetor *reg_esc_anterior* de 32 bits armazena o endereço do registrador destino da instrução anterior;
- inteiro *EscreveReg_anterior* é usado para armazenar o sinal *EscreveReg* da instrução anterior;

Na linha 15 um bloco *always* monitora a borda de descida do *clock* do processador e, caso o sinal *EscreveReg* da instrução anterior seja nível alto, o registrador da posição *reg_esc_anterior* do banco de registradores recebe o valor contigo no vetor *dado_esc*.

Na linha 22 um bloco *always* monitora a borda de subida do *clock* e sempre atualiza a posição do registrador de escrita e do sinal *EscreveReg* da instrução atual para que sejam utilizados na borda de descida, caso necessário.

4.5.3 Extensor de 16 bits

O extensor recebe um valor imediato com 16 bits e estende para um vetor de 32 bits.

```

1 module extensor_16 (entrada, saida);
2 input [15:0] entrada; // contem o valor do imediato
3 output reg [31:0] saida; // contem o imediato extendido
4 always @(*)
5 begin
6     saida = {16'b0000000000000000,entrada}; // concatena 16 bits contendo 0 a
                                                 esquerda da entrada
7 end
8 endmodule

```

O início do código declara o vetor *entrada* de 16 bits e o vetor *saida* de 32 bits. Na linha 4 um bloco *always* monitora qualquer variação na *entrada*, caso uma variação ocorra a *saida* é imediatamente estendida.

4.5.4 Extensor de 26 bits

O extensor recebe um valor imediato com 26 bits e estende para um vetor de 32 bits.

```

1 module extensor_26 (entrada, saida);
2 input [25:0] entrada; // contem o valor que representa a posicao na memoria da proxima
   instrucao a ser executada com 26 bits
3 output reg [31:0] saida; // contem o valor de entrada extendido para 32 bits
4 always @(*)
5 begin
6     saida = {6'b000000,entrada}; // concatena 6 bits com 0s a esquerda da entrada
7 end
8 endmodule

```

O início do código declara o vetor *entrada* de 26 bits e o vetor *saida* de 32 bits. Na linha 4 um bloco *always* monitora qualquer variação na *entrada*, caso uma variação ocorra a *saida* é imediatamente estendida.

4.5.5 Somador que incrementa o PC

Esse somador incrementa 1 ao endereço do PC para que a próxima instrução da memória de instruções seja executada. Caso o sinal de *pausa* esteja em nível alto, isso significa que o processador deve aguardar a inserção de uma entrada no sistema antes de executar a próxima instrução.

```

1 module incrementa_PC (entrada, saida, pausa);
2 input [31:0] entrada; // contem o endereco atual do PC
3 input pausa; // determina se o processador deve aguardar o recebimento da entrada externa
4 output reg [31:0] saida; // contem o endereco de PC + 1
5 always @(*)
6 begin
7     if(pausa==0) // nao esta recebendo entrada
8         begin
9             saida = entrada + 1; // acrescenta 1 em PC
10        end
11    else
12        begin
13            saida = entrada; // mantem a saida atual
14        end
15 end
16 endmodule

```

As 4 linhas iniciais do código declaram duas entradas e uma saída. O vetor de 32 bits *entrada* contém o endereço da instrução atual do PC, a entrada *pausa* contém o sinal de controle que determina se é necessário aguardar a inserção de uma entrada no sistema ou não, enquanto o vetor de 32 bits *saida* contém o endereço da próxima instrução.

Na linha 5 um bloco *always* monitora qualquer variação das entradas e atualiza a saída de acordo com uma condicional na linha 7, caso o sinal *pausa* esteja em nível alto a saída recebe a entrada antiga, caso esteja em nível baixo a saída é incrementada em 1 unidade.

4.5.6 Multiplexador para o registrador de escrita

A função desse multiplexador é determinar em qual posição do banco de registradores será gravado um dado de acordo com um sinal de controle enviado pela Unidade de Controle.

```

1 module mux_reg_escr (instrucao, reg_destino, RegDst);
2 input [31:0] instrucao; // contém a instrucao sendo executada
3 input [1:0] RegDst; // sinal que determina qual registrador receberá dados
4 output reg [4:0] reg_destino; // contém a posição do registrador de destino
5 always @(*)
6 begin
7     case (RegDst) // sinal de controle do multiplexador
8         0:
9             reg_destino = instrucao[20:16]; // determina rt como registrador
10            destino
11        1:
12            reg_destino = instrucao[15:11]; // determina rd como registrador
13            destino
14        2:
15            reg_destino = 31; // determina ra como registrador destino
16        endcase
17    end
18 endmodule

```

As 4 linhas iniciais declaram como entradas o vetor de 32 bits *instrucao*, que contém a instrução sendo executada, e o vetor de 2 bits *RegDst*, que contém o sinal de controle que seleciona uma das entradas, e como saída o vetor de 5 bits *reg_destino*, que entrega ao banco de registradores a posição do registrador que receberá um dado.

Na linha 5 um bloco *always* monitora qualquer alteração nos sinais de entrada utilizando da função *case* e atualiza a saída em função do valor do sinal de controle da seguinte forma:

- caso 0: seleciona a posição contida no intervalo [20:16] da instrução;
- caso 1: seleciona a posição contida no intervalo [15:11] da instrução;
- caso 2: seleciona a posição 31.

O caso 3 é um *don't care*, ou seja, não há alteração do *reg_destino*.

4.5.7 Multiplexador para o endereço da próxima instrução

Esse multiplexador seleciona entre quatro possíveis endereços de instrução fornecidos ao PC, podendo ser a própria instrução acrescida de uma posição, um salto da instrução *Jump*, um salto da instrução *JumpAndLink* ou um desvio somado ao PC.

```

1 module mux_prox_instr (entrada0, entrada1, entrada2, entrada3, saida, controle);
2 input [31:0] entrada0; // contém o endereço de PC + 1
3 input [31:0] entrada1; // contém o endereço da instrução PC + desvio imediato
4 input [31:0] entrada2; // contém o endereço da instrução como imediato

```

```

5  input [31:0] entrada3; // contém o endereço da instrução do registrador
6  output reg [31:0] saída; // endereço da próxima instrução entregue ao PC
7  input [1:0] controle; // armazena os sinais de controle
8  always @(*)
9  begin
10    case(controle)
11      0: saída = entrada0; // PC + 1
12      1: saída = entrada1; // Salto com endereço em registrador
13      2: saída = entrada2; // Salto com endereço no valor imediato
14      3: saída = entrada3; // Desvio com posição calculada
15    endcase
16  end
17 endmodule

```

As sete primeiras linhas declaram as entradas e saídas, conforme a seguir:

- entrada *entrada0* é um vetor de 32 bits que contém o endereço da instrução de PC + 1;
- entrada *entrada1* é um vetor de 32 bits que contém o endereço da instrução de PC + desvio do imediato;
- entrada *entrada2* é um vetor de 32 bits que contém o endereço da instrução do imediato;
- entrada *entrada3* é um vetor de 32 bits que contém o endereço da instrução do registrador;
- entrada *controle* contém o sinal da Unidade de Controle que determina qual entrada deve ser atribuída à saída;
- saída *saída* é um vetor de 32 bits que contém o endereço da próxima instrução a ser atribuído ao PC.

Na linha 8 um bloco *always* monitora qualquer alteração nos elementos da entrada e atualiza a *saída* de acordo com o sinal *controle* através de um bloco *case* na linha 10.

4.5.8 Multiplexador para o dado fornecido à ULA

A função desse multiplexador é selecionar qual dado será entregue à ULA, o dado fornecido pelo banco de registradores ou o valor do imediato estendido.

```

1 module mux_dado_ULA (dado, imediato, origALU, saída);
2   input [31:0] dado; // contém o valor do registrador rt em instruções tipo R
3   input [31:0] imediato; // contém o valor do imediato em instruções tipo I
4   input origALU; // sinal que determina qual será o valor entregue a ULA
5   output reg [31:0] saída; // contém o valor a ser entregue a ULA
6   always @(*)
7   begin
8     if(origALU==0)
9       begin

```

```

10         saida = dado; // determina que a ULA receba o valor do registrador rt
11     end
12     if(origALU==1)
13     begin
14         saida = imediato; // determina que a ULA receba o valor do imediato
15     end
16 end
17 endmodule

```

As cinco primeiras linhas de códigos declaram entradas e saídas, sendo três entradas e uma saída. São declarados dois vetores de 32 bits cada, o *dado* que contém o valor fornecido pelo bando de registradores e o *imediato*, que contém o valor do imediato estendido, além da entrada *origALU* que contém o sinal de controle da Unidade de Controle e determina qual entrada deve ser selecionada. O vetor de 32 bits *saida* fornece à ULA o valor do segundo operando.

Na linha 6 um bloco *always* monitora qualquer variação nas entradas e atualiza o vetor *saida* através de duas condicionais nas linhas 8 e 12 que atribuem o valor armazenado em *dado* à *saida* caso *origALU* esteja em nível baixo e o valor de *imediato* à *saida* caso contrário.

4.5.9 Multiplexador para o dado que será escrito

A função desse multiplexador é selecionar a origem do dado escrito em um registrador do banco de registradores. Esse dado pode ter quatro origens, sendo elas:

- memória de dados: caso de uma instrução *Load*;
- unidade lógica e aritmética: caso de operações lógicas e aritméticas;
- *program counter* (PC): caso da instrução *JumpAndLink* que requer que o endereço da instrução atual seja armazenado na posição 31 do banco de registradores;
- entrada externa: permite a comunicação com o mundo externo através da instrução de entrada.

```

1 module mux_dado_breg (entrada0, entrada1, entrada2, entrada3, saida, controle);
2 input [31:0] entrada0; // contem a saida da ULA
3 input [31:0] entrada1; // contem a saida da memoria de dados
4 input [31:0] entrada2; // contem PC + 1
5 input [31:0] entrada3; // contem o valor da entrada externa
6 input [1:0] controle; // determina qual a entrada correta em funcao do opcode
7 output reg [31:0] saida; // contem o dado a ser armazenado no registrador destino
8 always @(*)
9 begin
10     case(controle)
11         0: saida = entrada0; // registrador de escrita recebe o resultado da ULA
12         1: saida = entrada1; // registrador de escrita recebe dado da memoria de
13             dados
14         2: saida = entrada2; // registrador de escrita recebe PC + 1

```

```

14      3: saida = entrada3; // registrador de escrita recebe entrada externa
15      endcase
16  end
17 endmodule

```

As sete primeiras linhas declaram as entradas e saídas do código, sendo cinco entradas e uma saída, conforme descrito a seguir:

- entrada *entrada0* contém a saída da ULA para operações lógicas e aritméticas;
- entrada *entrada1* contém o dado da memória de dados para a instrução *Load*;
- entrada *entrada2* contém o valor de PC + 1 para a instrução *JumpAndLink*;
- entrada *entrada3* contém o valor da entrada externa para a instrução de entrada;
- entrada *controle* contém o sinal da Unidade de Controle que determina qual entrada será fornecida ao banco de registradores;
- saída *saida* contém o dado fornecido ao banco de registradores.

O bloco *always* da linha 8 monitora qualquer variação das entradas e seleciona uma saída baseando no sinal *controle* através do bloco *case* da linha 10.

4.5.10 Somador para o imediato

A função desse somador é somar o valor do imediato já estendido ao endereço de instrução do PC.

```

1 module soma_immediato (entrada, imediato, saida);
2   input [31:0] entrada; // contem o valor de PC + 1
3   input [31:0] imediato; // contem o valor do imediato
4   output reg [31:0] saida; // contem a soma de PC + 1 + imediato
5   always @(*)
6   begin
7     saida = entrada + imediato; // soma PC + 1 + imediato e armazena na saida
8   end
9 endmodule

```

O módulo contém como entrada dois vetores de 32 bits cada, um contém a instrução do PC, chamado *entrada*, e o outro o valor do imediato estendido, chamado *imediato*. A saída é um vetor de 32 bits chamado *saida*.

Um bloco *always* na linha 5 monitora qualquer variação nas entradas e imediatamente atualiza o vetor *saida* com a soma das entradas.

4.5.11 Unidade Lógica e Aritmética

A Unidade Lógica e Aritmética realiza operações lógicas e aritméticas. As operações contempladas por esse projeto podem ser observadas na tabela 1.

O código que compõe a ULA é extenso e pode ser analisado na íntegra no Apêndice A.

```

1 module ULA (dado1, dado2, zero, saida, OpALU, shamt);
2
3 input [31:0] dado1; // contém o valor do primeiro operando
4 input [31:0] dado2; // contém o valor do segundo operando
5 input [4:0] shamt; // contém o numero de bits a serem deslocados em operacoes do tipo
6   shift
7 input [5:0] OpALU; // contém o sinal que determina a operacao a ser realizada com origem
8   na Unidade de Controle
9
10 output reg zero; // flag de controle para desvios e operacoes logicas
11 output reg [31:0] saida; // contém o valor resultante da operacao
12 initial zero = 0; //inicia a flag zero com nivel baixo
13 initial saida = 0; //inicia a saida com valor nulo
14
15 always @(*)
16 begin
17   case(OpALU) // monitora o sinal de controle da ULA
18     6'd2: // verifica se o sinal de controle vale 2
19       begin
20         saida = dado1+dado2; //soma atribui valor dado1+dado2 a saida
21         zero = 0; // atribui valor nulo a flag zero
22       end
23     6'd3:
24       begin
25         saida = dado1-dado2; //subtracao
26         zero = 0;
27       end

```

As 9 primeiras linhas do código da ULA declaram entradas e saídas da unidade funcional, conforme descrito a seguir:

- entrada *dado1*: é um vetor de 32 bits que contém o dado do primeiro operando;
- entrada *dado2*: é um vetor de 32 bits que contém o dado do segundo operando;
- entrada *shamt*: é um vetor de 5 bits contém a quantidade de bits a serem deslocadas em operações *shift*;
- entrada *OpALU*: é um vetor de 6 bits que contém o sinal de controle da Unidade de Controle;
- saída *saida*: é um vetor de 32 bits que contém o resultado da operação;
- saída *zero*: é uma *flag* utilizada para operações de comparação e resultados lógicos.

As 9 linhas iniciais do código declaram e inicializam variáveis e entradas. As linhas 8 e 9 iniciam o vetor *saida* e a *flag zero* com o valor 0 para evitar incoerências. Inicia-se na linha 10 um bloco *always* que estende-se até o final do código e monitora as variações nos elementos de entrada da ULA.

Internamente ao bloco *always*, um bloco *case* estende-se até o final do código e atribui ao vetor *saida* e à *flag zero* novos valores de acordo com o sinal de controle OpALU.

Pode-se observar que não há operações na ULA para alguns valores entre 0 e 28, isso ocorre porque nem todas as instruções utilizam a ULA. Para esses casos há um caso *default* que zera as saídas.

4.6 Unidades de memória em lógica programável

Essa etapa do projeto consiste no desenvolvimento das memórias de instruções e de dados. O software Quartus possui templates otimizados para criação de memórias utilizados no projeto.

4.6.1 Memória de instruções

A memória de instruções é responsável por armazenar o programa que está sendo executado.

```

1 // Quartus Prime Verilog Template
2 // Single Port ROM
3 module mem_instr
4 #(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=10)
5 (
6     input [(ADDR_WIDTH-1):0] addr,
7     input clk,
8     output reg [(DATA_WIDTH-1):0] q
9 );
10    // Declare the ROM variable
11    reg [DATA_WIDTH-1:0] rom[2**ADDR_WIDTH-1:0];
12
13    initial
14    begin
15        $readmemb("programa.txt", rom);
16        q = 32'd1;
17    end
18    always @ (negedge clk)
19    begin
20        q <= rom[addr];
21    end
22 endmodule

```

A memória de instruções consiste em uma memória de leitura somente (*Read Only Memory*), portanto o template *Single Port ROM*, que define uma memória somente leitura de porta única, é suficiente para criar essa memória.

Na linha 4 o parâmetro *DATA_WIDTH* determina o comprimento de cada palavra, nesse caso 32 bits. O parâmetro *ADDR_WIDTH* determina o tamanho do vetor de endereçamento como 10 bits, ou seja, estabelece quantas posições essa memória possui, nesse caso 2^{10} posições, ou seja, 1024 posições de 32 bits de comprimento cada.

Na linha 15 ocorre a leitura do arquivo que contém as instruções do programa em código de máquina, nesse projeto o arquivo deve estar na pasta do projeto e deve ser nomeado *programa.txt*.

Na linha 18 um bloco *always* monitora a borda de descida do sinal de *clock clk* do processador e atualiza o vetor de saída de 32 bits *q* com a instrução armazenada na posição *addr* da memória.

4.6.2 Memória de dados

A memória de dados é responsável por armazenar os valores utilizados ou gerados pelo programa.

```

1 // Quartus Prime Verilog Template
2 // Simple Dual Port RAM with separate read/write addresses and
3 // single read/write clock
4
5 module mem_dados
6 #(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=10)
7 (
8     input [(DATA_WIDTH-1):0] data,
9     input [31:0] read_addr, write_addr,
10    input we, clk,
11    output reg [(DATA_WIDTH-1):0] q
12 );
13     reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];
14
15     always @ (posedge clk)
16     begin
17         // Write
18         if (we)
19             ram[write_addr] <= data;
20
21         q = ram[read_addr];
22     end
23 endmodule

```

A memória de dados consiste em uma memória de leitura e escrita (*Random Access Memory*), portanto o template *Simple Dual Port RAM*, que define uma memória de leitura e escrita com endereços de escrita e leitura separados e *clock* único para leitura e escrita, é suficiente para criar essa memória.

Na linha 6 o parâmetro *DATA_WIDTH* determina o comprimento de cada palavra, nesse caso 32 bits. O parâmetro *ADDR_WIDTH* determina o tamanho do vetor de endereçamento como 10 bits, ou seja, assim como na memória de instruções, estabelece quantas posições essa memória possui, nesse caso 2^{10} posições, ou seja, 1024 posições de 32 bits de comprimento cada.

Essa memória possui cinco entradas e uma saída, um vetor de 32 bits contém o dado em caso de gravação, dois vetores de 32 bits cada, *read_addr* e *write_addr* recebem os endereços de leitura e gravação, um sinal de entrada *we* determina se há gravação e o

clk recebe o *clock* do processador. O vetor de 32 bits de saída *q* contém o dado de saída em caso de leitura.

Na linha 15 um bloco *always* monitora a borda de subida do sinal de *clock clk* do processador e atualiza o vetor de saída de 32 bits *q* com a instrução armazenada na posição *addr* da memória. Caso o sinal *we* esteja em nível alto, a posição *write_addr* da memória recebe o valor armazenado por *data*.

Este projeto se propõe ao desenvolvimento de um processador monociclo, dessa forma a gravação e leitura da memória de dados não ocorre simultaneamente, dispensando o uso de dois endereços diferentes para leitura e escrita. Da mesma forma, os endereços de leitura e escrita foram determinados para abranger um intervalo de 1024 posições, não necessitando de 32 bits para referência. Contudo, para manter o padrão do template e considerando que a saída da ULA, que calcula o endereço do dado a ser lido na memória, possui também 32 bits, optou-se por manter essas configurações recomendadas.

4.7 Unidades de entrada e saída em lógica programável

As unidades de entrada e saída exercem a comunicação do processador com o mundo exterior. O kit Altera DE2-115 com FPGA possui 18 chaves seletoras e 8 displays de 7 segmentos utilizados nessa comunicação, sendo as chaves as ferramentas de inserção de informações e os displays as ferramentas de saída de informações.

4.7.1 Unidade de entrada

A unidade de entrada é responsável por receber dados de até 32 bits e utiliza as 18 chaves seletoras do kit Altera DE2-115 para realizar essa tarefa.

```

1 module entrada (chaves, controle, dado, continua, sup, inf, ch0, clk);
2
3 input clk;
4 input ch0; // chave que determina a fase em que se encontra a entrada de dados
5 input [15:0] chaves; // chaves que contêm os 16 bits parciais de cada entrada
6 input controle; // sinal que determina se o processador está aguardando uma entrada de
    dados
7 output reg [31:0] dado; // vetor que armazena o dado recebido na entrada
8 output reg continua; // flag que sinaliza o término da leitura
9 output sup, inf;
10 reg [2:0] aux;
11 reg anterior, atual;
12 initial
13 begin
14     continua = 1'd1;
15     dado = 32'd0;
16     aux=0;
17     anterior=0;
18     atual=0;
19 end

```

```

21 assign sup = (controle && !ch0);
22 assign inf = (controle && ch0);
23
24 always@(posedge clk)
25 begin
26     if(controle)
27         begin
28             anterior=atual;
29             atual=ch0;
30             if(anterior==atual)
31                 begin
32                     continua=0;
33                     aux=2;
34                 end
35             else if(anterior!=atual)
36                 begin
37                     if(anterior<atual)
38                         begin
39                             dado[31:16]=chaves;
40                             continua=0;
41                             aux=3;
42                         end
43                     else if(anterior>atual)
44                         begin
45                             dado[15:0]=chaves;
46                             continua=1;
47                             aux=4;
48                         end
49                     end
50                 end
51             else
52                 begin
53                     continua=1;
54                     aux=1;
55                 end
56         end
57 endmodule

```

As oito linhas iniciais configuram as entradas e saídas do módulo, conforme descrito abaixo:

- entradas *ch0* e *ch1* correspondem às chaves 0 e 1 do kit com FPGA e atuam na identificação da fase em que se encontra o processo de recebimento de dados;
- entrada *controle* contém o sinal da Unidade de Controle que determina quando um recebimento de dados deve ocorrer;
- entrada *chaves* é um vetor de 16 bits que contém os valores das 16 chaves restantes do kit FPGA e são gravadas no bits mais ou menos significativo da entrada, dependendo da fase em que o recebimento se encontra;
- saída *dado* é um vetor de 32 bits que contém o valor da entrada;
- saída *continua* é uma flag que sinaliza à Unidade de Controle o término do recebimento de dados.

Na linha 9 um bloco *always* monitora variações nos sinais de entrada e atualiza as saídas de acordo com os sinais de controle.

4.7.2 Unidade de saída

A unidade de saída é responsável por apresentar dados ao mundo exterior através dos 8 displays de 7 segmentos do kit Altera DE2-115.

```

1 module saida (dado, controle,d0,d1,d2,d3,d4,d5,d6,d7,neg,continua,ch0,clk);
2
3 input clk;
4 input [31:0] dado;
5 input controle;
6 input ch0;
7 output reg [6:0] d0,d1,d2,d3,d4,d5,d6,d7;
8 output reg neg;
9 output reg continua;
10 reg [31:0] entrada;
11
12 reg ant,at;
13
14 reg sinal;
15 reg [2:0] aux;
16
17 initial
18 begin
19     continua = 1'd1;
20     sinal = 0;
21     aux=3'd0;
22     ant=0;
23     at=0;
24 end
25
26 always @ (negedge continua)
27 begin
28     if ($signed(dado)<$signed(32'd0))
29         begin
30             entrada = ~(dado-32'd1);
31             neg = 1;
32         end
33     else
34         begin
35             entrada = dado;
36             neg = 0;
37         end
38 end
39
40 always@ (controle or at)
41 begin
42     if(controle)
43         begin
44             if (!ant && !at)
45                 begin
46                     continua = 0;
47                 end
48             else if (!ant && at)
49                 begin
50                     continua =0;
51                 end
52         end
53 end
54
55 endmodule

```

```
51         end
52     else if(ant && at)
53     begin
54         continua = 0;
55     end
56     else if(ant && !at)
57     begin
58         continua = 1;
59     end
60 end
61 else
62 begin
63     continua=1;
64 end
65 end
66
67 always@(posedge clk)
68 begin
69     if(controle)
70     begin
71         if(ch0)
72         begin
73             if(!ant && !at)
74                 at=1;
75             else if(!ant && at)
76                 ant=1;
77         end
78     else
79     begin
80         if(!ant && !at)
81         begin
82             ant=0;
83             at=0;
84         end
85         else if(ant && at)
86             at=0;
87     end
88 end
89 else
90 begin
91     ant=0;
92     at=0;
93 end
94 end
95
96 always@(entrada)
97 begin
98     case (entrada[3:0])
99         4'b0000: d0=7'b1000000;
100        4'b0001: d0=7'b1111001;
101        4'b0010: d0=7'b0100100;
102        4'b0011: d0=7'b0110000;
103        4'b0100: d0=7'b0011001;
104        4'b0101: d0=7'b0010010;
105        4'b0110: d0=7'b0000010;
106        4'b0111: d0=7'b1111000;
107        4'b1000: d0=7'b0000000;
108        4'b1001: d0=7'b00010000;
109        4'b1010: d0=7'b00001000;
110        4'b1011: d0=7'b0000011;
```

```

111          4'b1100: d0=7'b1000110;
112          4'b1101: d0=7'b0100001;
113          4'b1110: d0=7'b0000110;
114          4'b1111: d0=7'b0001110;
115      endcase

```

O módulo de saída possui código extenso e o código completo pode ser analisado no apêndice B. Contudo, o código da unidade de saída é repetitivo, repetindo 8 vezes o bloco *case*, uma vez para cada *display* de 7 segmentos. A unidade diferença é o trecho do vetor de entrada que é monitorado pelo bloco *case* em cada repetição.

As quatro linhas iniciais do código declaram como entrada o vetor de 32 bits *entrada* que contém o dado a ser representado dos *displays* e o sinal de controle *controle* que determina quando os *displays* devem ser atualizados. São declarados também oito vetores de 7 bits cada, um para cada *display* do kit, os bits desses vetores sinalizam quais dos segmentos devem estar acesos ou apagados para formar o número ou letra desejado.

Optou-se pela representação dos valores de saída em base hexadecimal, pois há somente 8 *displays* no kit que não seriam suficientes para expressar esse dados em base decimal.

Cada *display* de 7 segmentos pode representar até 16 valores (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E e F), dessa forma a palavra de 32 bits foi quebrada em 8 partes iguais de 4 bits cada que variam de 0 até F e são convertidas e relacionadas ao seu *display* correspondente.

4.8 Unidades funcionais interconectadas sem a Unidade de Controle

As unidades funcionais desenvolvidas até o momento são interconectadas com o objetivo de simular o funcionamento das instruções do projeto.

```

1 module processador (RESET, CLOCK, CHAVES, D0, D1, D2, D3, D4, D5, D6, D7, RegDst,
                     EscreveReg, controle_pi, controle_in, controle_dbr, controle_out, origALU, OpALU,
                     EscreveMem);
2 // declaracao dos sinais de entrada fisicos (clock, reset e chaves) e sinais de controle
   // que serao atribuidos a Unidade de Controle
3 input CLOCK, RESET, EscreveReg, origALU, EscreveMem, controle_out, controle_in;
4 input [17:0] CHAVES;
5 input [1:0] controle_pi, controle_dbr, RegDst;
6 input [4:0] OpALU;
7 // declaracao das saidas que correspondem aos displays de 7 segmentos
8 output [6:0] D0, D1, D2, D3, D4, D5, D6, D7;
9
10 // chamada do modulo do PC
11 wire [31:0] pc_in, pc_out;
12 PC IM_0(.entrada(pc_in), .saida(pc_out), .clk(CLOCK), .reset(RESET));
13
14 // chamada do modulo de memoria de instrucoes
15 wire [31:0] instr;
16 mem_instr IM_1(.addr(pc_out), .clk(CLOCK), .q(instr));

```

```

17
18 // chamada do modulo do multiplexador do registrador que recebera dado
19 wire [4:0] rd;
20 mux_reg_escr IM_2(.instrucao(instr),.reg_destino(rd),.RegDst(RegDst));
21
22 // chamada do modulo que incrementa o PC e recebe o sinal de pausa para recebimento de
23 // dados
23 wire [31:0] pc_incr;
24 wire pausa;
25 incrementa_PC IM_3(.entrada(pc_out),.saida(pc_incr),.pausa(pausa));
26
27 // chamada do modulo que estende o dado de 26 para 32 bits
28 wire [31:0] ext_26;
29 extensor_26 IM_4(.entrada(instr[25:0]),.saida(ext_26));
30
31 // chamada do modulo que estende o dado de 16 para 32 bits
32 wire [31:0] ext_16;
33 extensor_16 IM_5(.entrada(instr),.saida(ext_16));
34
35 // chamada do modulo que faz a soma do PC+1 com o imediato
36 wire [31:0] desvio;
37 soma_immediato IM_6(.entrada(pc_incr),.immediato(ext_16),.saida(desvio));
38
39 // chamada do modulo multiplexador da proxima instrucao fornecida ao PC
40 mux_prox_instr IM_7(.entrada0(pc_incr),.entrada1(desvio),.entrada2(ext_26),.entrada3(
41     dado_rs),.saida(pc_in),.controle(controle_pi));
42
42 // chamada do modulo multiplexador do segundo operando da ULA
43 wire [31:0] dado2_ULA;
44 mux_dado_ULA IM_8(.dado(dado_rt),.immediato(ext_16),.origALU(origALU),.saida(dado2_ULA));
45
46 // chamada do modulo da Unidade Logica e Aritmetica
47 wire [31:0] result_ula;
48 wire zero;
49 ULA IM_9(.dado1(dado_rs),.dado2(dado2_ULA),.zero(zero),.saida(result_ula),.OpALU(OpALU),.
50     shamt(instr[10:6]));
51
51 // chamada do modulo da memoria de dados
52 wire [31:0] dado_mem;
53 mem_dados IM_10(.data(dado_rt),.read_addr(result_ula),.write_addr(result_ula),.we(
54     EscreveMem),.clk(CLOCK),.q(dado_mem));
55
55 // chamada do modulo de entrada de dados
56 wire [31:0] dado_in;
57 wire fim_receb;
58 entrada IM_11(.ch0(CHAVES[0]),.ch1(CHAVES[1]),.chaves(CHAVES[17:2]),.controle(controle_in
59     ),.dado(dado_in),.continua(fim_receb));
60
60 // chamada do modulo multiplexador do dado fornecido ao banco de registradores
61 wire [31:0] dado_breg;
62 mux_dado_breg IM_12(.entrada0(result_ula),.entrada1(dado_mem),.entrada2(pc_incr),
63     .entrada3(dado_in),.saida(dado_breg),.controle(controle_dbr));
64
64 // chamada do modulo do banco de registradores
65 wire [31:0] dado_rs, dado_rt;
66 banco_reg IM_13(.EscreveReg(EscreveReg),.clk(CLOCK),.reg_leit1(instr[25:21]),.reg_leit2(
67     instr[20:16]),.reg_esc(rd),.dado_esc(dado_breg),.dado_leit1(dado_rs),.dado_leit2(
68     dado_rt));
68 // chamada do modulo de saida de dados

```

```
69 saída IM_14(.entrada(dado_rs),.controle(controle_out),.d0(D0),.d1(D1),.d2(D2),.d3(D3),.d4  
    (D4),.d5(D5),.d6(D6),.d7(D7));  
70  
71 endmodule
```

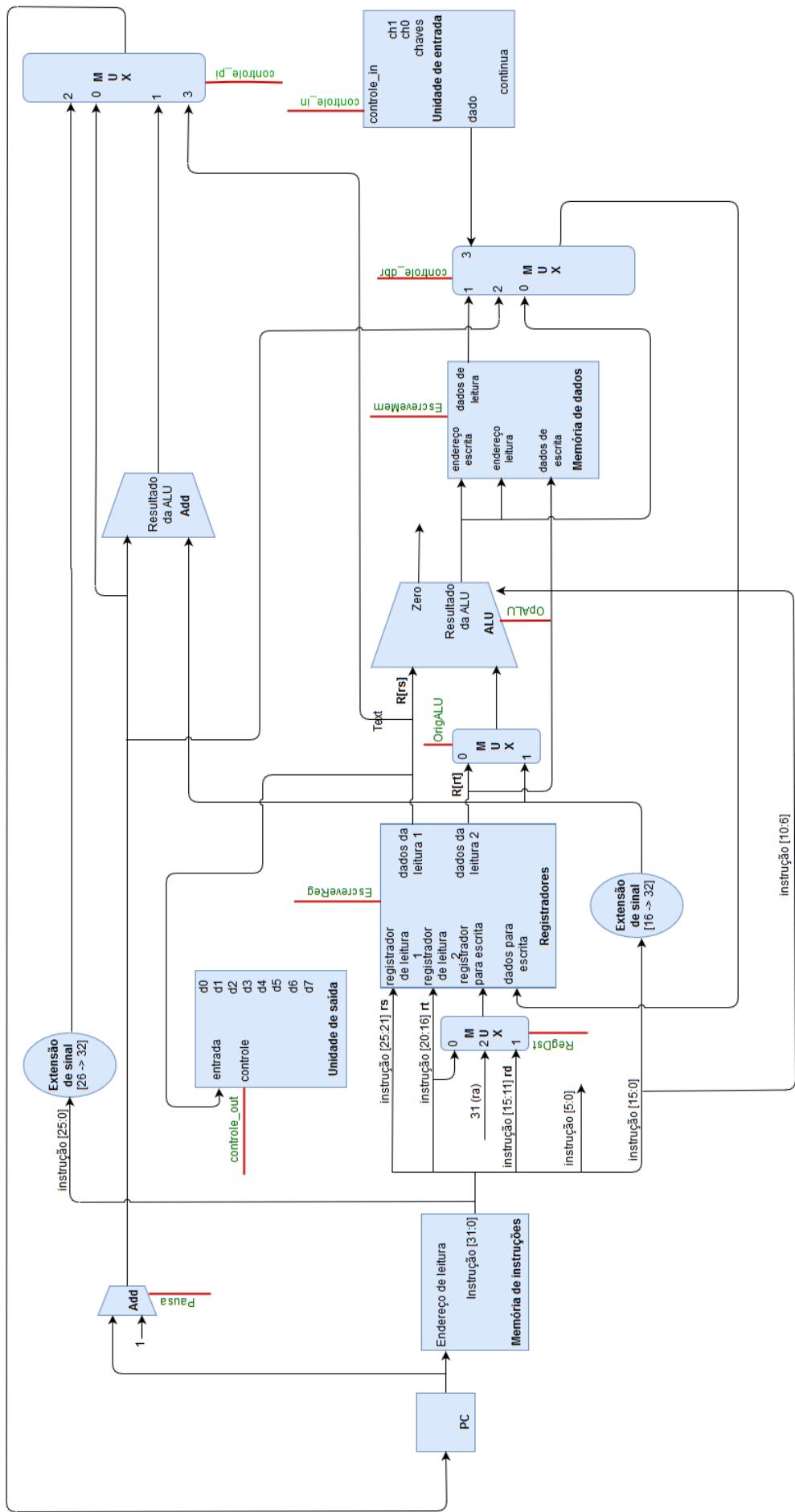
A interconexão das unidades funcionais na ausência da Unidade de Controle requereu que os sinais de controle fossem atribuídos como elementos de entrada para que os módulos pudessem ser simulados. Dessa forma, pode-se observar que os sinais de controle atribuídos à Unidade de Controle são elementos de entrada nesse módulo.

A descrição do código ocorre através de comentários para tornar o processo mais intuitivo em função da extensão do mesmo.

4.8.1 Novo caminho de dados

A implementação em lógica programável do projeto proporcionou uma revisão do caminho de dados, de forma que o caminho foi adaptado às novas características do projeto. A figura 25 apresenta esse novo caminho adaptado às configurações da implementação em lógica programável.

Figura 25 – Caminho de dados conforme a implementação em lógica programável



Fonte: Elaborado pela autora utilizando o site Draw.io

4.8.2 Unidade de controle

A unidade de controle tem como função organizar o processador e fornecer os sinais adequados ao perfeito funcionamento deste.

O código em lógica programável é extenso e apresenta-se no apêndice C.

```

1 module UC (instr, zero, controle_out, RegDst, EscreveReg, OrigALU, OpALU, EscreveMem,
2   controle_dbr, controle_in, controle_pi);
3
4   input [31:0] instr;
5   input zero;
6   output reg controle_out;
7   output reg [1:0] RegDst;
8   output reg EscreveReg;
9   output reg OrigALU;
10  output reg [5:0] OpALU;
11  output reg EscreveMem;
12  output reg [1:0] controle_dbr;
13  output reg controle_in;
14  output reg [1:0] controle_pi;
15 ...
16
17 always @(instr or zero)
18 begin
19   case(instr[31:26])
20     6'd0:
21     begin
22       case(instr[5:0])
23         6'd0:
24           begin
25             RegDst = 0;
26             EscreveReg = 1;
27             OrigALU = 0;
28             OpALU = 0;
29             EscreveMem = 0;
30             controle_dbr = 3;
31             controle_pi = 0;
32           end
33         6'd1:
34           begin
35             ...
36           end
37         6'd2:
38           begin
39             ...
40           end
41         6'd3:
42           begin
43             ...
44           end
45         6'd4:
46           begin
47             ...
48           end
49         6'd5:
50           begin
51             ...
52           end
53         6'd6:
54           begin
55             ...
56           end
57         6'd7:
58           begin
59             ...
60           end
61       end
62     end
63   end
64 endmodule

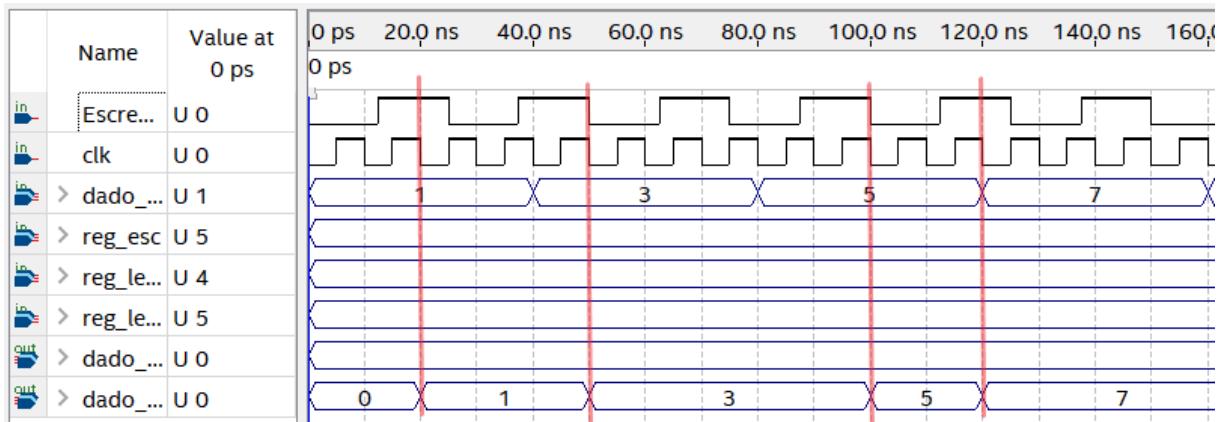
```

5 Resultados obtidos e discussão

Nessa etapa de desenvolvimento do projeto os resultados obtidos consistem de testes e simulação dos módulos individualmente e interconectados através de formas de onda geradas pelo software Quartus Prime através do módulo de simulação.

5.0.1 Banco de registradores

Figura 26 – Simulação do funcionamento do banco de registradores

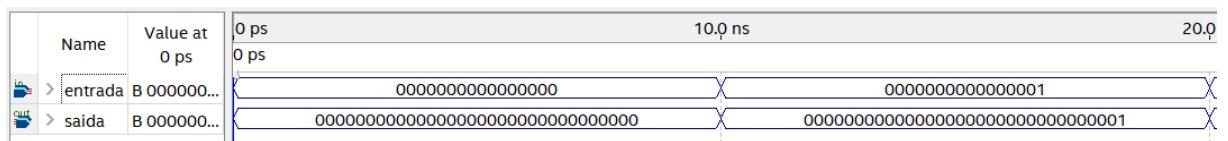


Fonte: Software Quartus Prime

Nesse teste o dado fornecido para gravação varia de 1 até 7 (1, 3, 5 e 7), o registrador de escrita é 5º e os registradores de leitura são o 4º e o 5º. Conforme previsto, as gravações ocorrem na borda de descida do *clock* quando o sinal de controle *EscreveReg* está em nível alto. A leitura é assíncrona.

5.0.2 Extensor de 16 bits

Figura 27 – Simulação do funcionamento do extensor de 16 bits

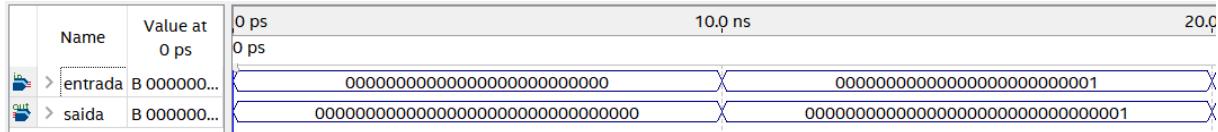


Fonte: Software Quartus Prime

Nesse teste a entrada consiste nos valores 0 e 1 binário com 16 bits. Conforme previsto, a saída corresponde à entrada estendida para 32 bits.

5.0.3 Extensor de 26 bits

Figura 28 – Simulação do funcionamento do extensor de 26 bits

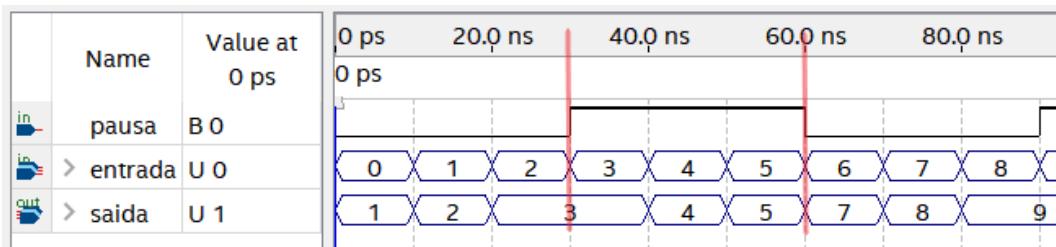


Fonte: Software Quartus Prime

Nesse teste a entrada consiste nos valores 0 e 1 binário com 26 bits. Conforme previsto, a saída corresponde à entrada estendida para 32 bits.

5.0.4 Incremento ao PC

Figura 29 – Simulação do funcionamento da unidade que incrementa o PC

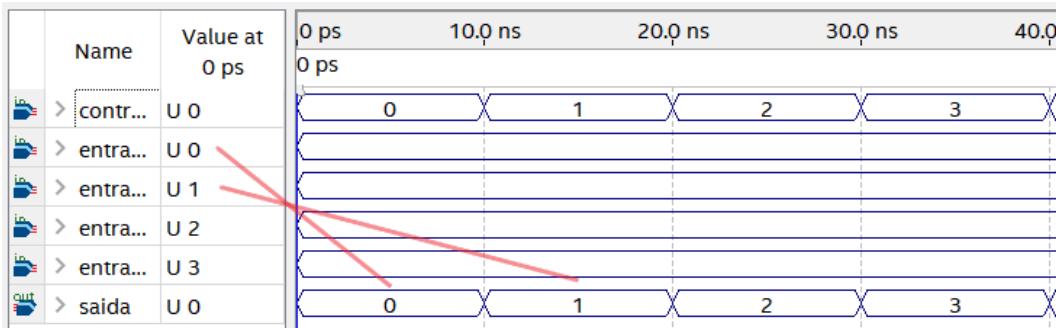


Fonte: Software Quartus Prime

Nesse teste a entrada varia de 0 até 8 e há um sinal de pausa. Conforme previsto, a saída corresponde à entrada acrescida de uma unidade caso o sinal de pausa esteja em nível baixo ou corresponde à própria entrada caso o sinal de pausa esteja em nível alto.

5.0.5 Multiplexador do dado de escrita no banco de registradores

Figura 30 – Simulação do funcionamento do multiplexador do dado de escrita no banco de registradores

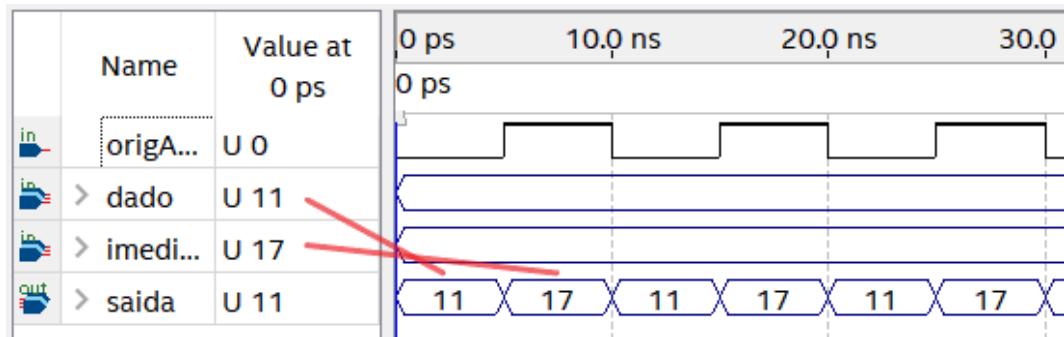


Fonte: Software Quartus Prime

Nesse teste são utilizadas 4 entradas que receberam, respectivamente, os valores 0, 1, 2 e 3. Um sinal de controle (controle_dbr) varia de 0 até 3 e as saídas variam de acordo com as entradas, conforme previsto.

5.0.6 Multiplexador do segundo operando fornecido à ULA

Figura 31 – Simulação do funcionamento do multiplexador do segundo operando fornecido à ULA

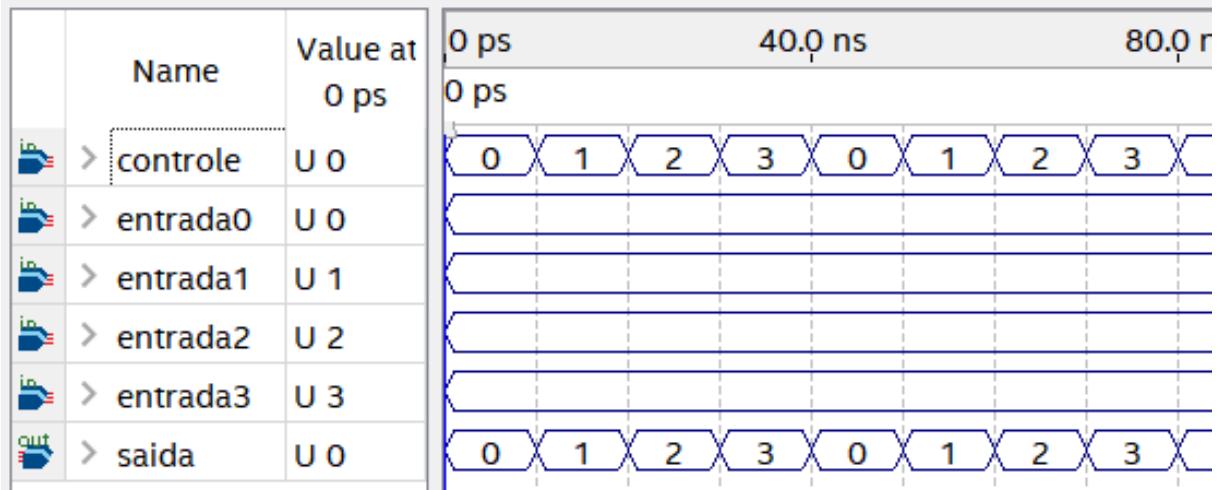


Fonte: Software Quartus Prime

Nesse teste uma entrada com valor 11 simula a saída do banco de registradores e uma entrada com valor 17 simula o valor do imediato. Conforme previsto, o sinal origALU quando em nível baixo seleciona o dado do banco de registradores e quando alto seleciona o dado do imediato.

5.0.7 Multiplexador da próxima instrução fornecida ao PC

Figura 32 – Simulação do funcionamento do multiplexador da próxima instrução fornecida ao PC

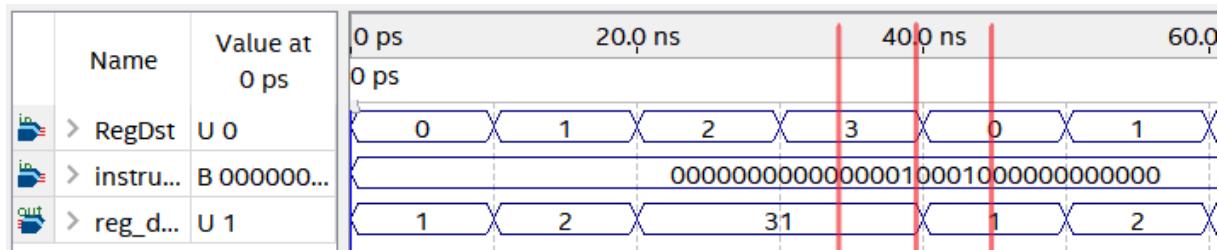


Fonte: Software Quartus Prime

Nesse teste são utilizadas 4 entradas constantes de valores 0, 1, 2 e 3. O sinal de controle varia de 0 até 3 e atribui à saída o sinal da entrada de acordo com esse sinal de seleção, conforme previsto.

5.0.8 Multiplexador do endereço do registrador de escrita

Figura 33 – Simulação do funcionamento do multiplexador do endereço do registrador de escrita



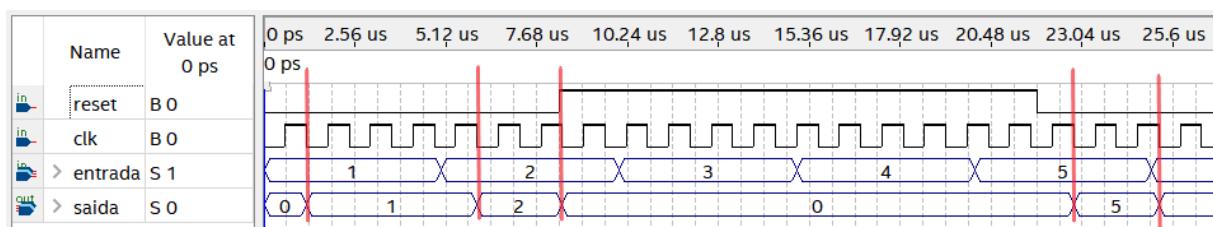
Fonte: Software Quartus Prime

Nesse teste o sinal de controle RegDst varia de 0 até 3 porém seleciona apenas entre 3 sinais de entrada, portanto, quando o sinal RegDst vale 3 não há variação na saída. O sinal de entrada consiste em uma instrução que contém a posição 1 como rs e a posição 2 como rt.

Conforme previsto, a saída equivale à posição 1 quando o sinal de controle é 0, posição 2 quando o sinal de controle é 1 e posição 31 quando o sinal de controle é 2. A posição 31 é fixa pois é utilizada na instrução *jump and link*.

5.0.9 Program Counter - PC

Figura 34 – Simulação do funcionamento dprogram counter - PC



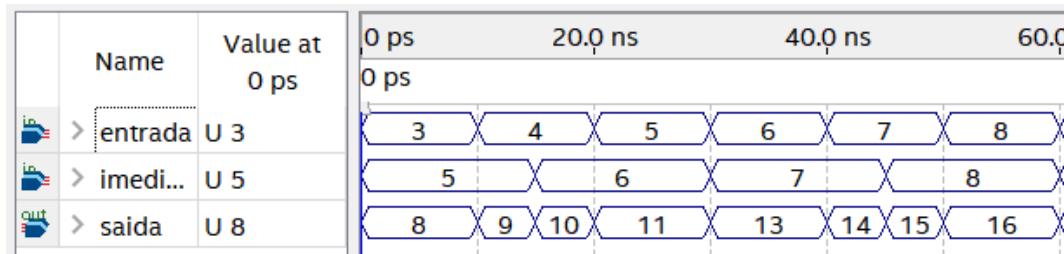
Fonte: Software Quartus Prime

Nesse teste uma entrada que simula instrução varia de 1 até 5, um sinal de *reset* também influencia na saída. Conforme previsto, o sinal de saída varia na borda de descida do *clock*. Se o sinal de *reset* estiver em nível alto o sinal de saída recebe o valor 0 e o programa é reiniciado.

Na prática a unidade PC é realimentada com a própria saída acrescida em uma unidade, o que implica no funcionamento do *reset*. Na simulação a entrada independe da saída, isso inviabiliza a verificação do funcionamento por completo.

5.0.10 Somador do imediato ao PC

Figura 35 – Simulação do funcionamento da unidade que soma o imediato ao PC

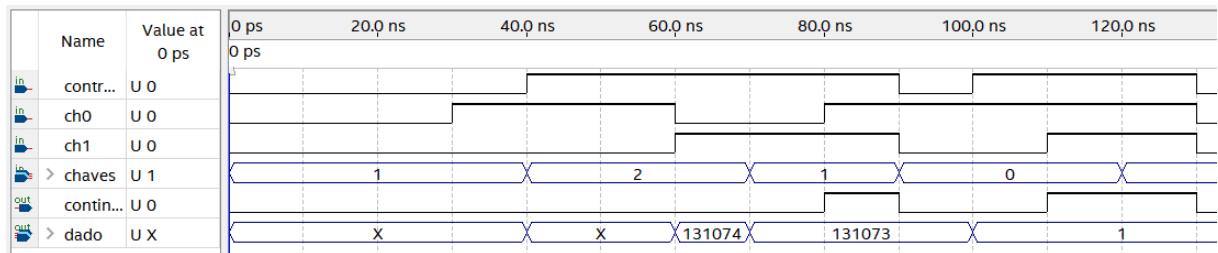


Fonte: Software Quartus Prime

Nesse teste uma entrada que simula o endereço da instrução varia de 3 até 8 e outra entrada que simula o valor do imediato varia de 5 até 8. Conforme previsto, a saída é a soma desse imediato com a entrada.

5.0.11 Unidade de entrada de dados

Figura 36 – Simulação do funcionamento da unidade de entrada de dados



Fonte: Software Quartus Prime

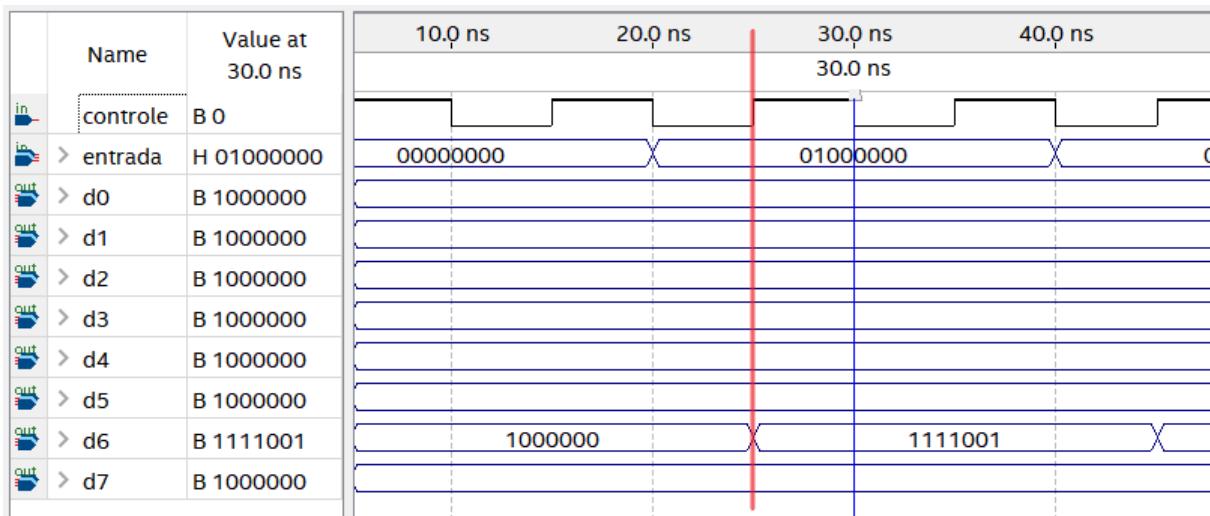
Nesse teste duas chaves (ch0 e ch1) determinam a fase do recebimento de dados, o sinal de controle determina se ocorre recebimento de dados ou não e a combinação das chaves resulta nos bits as serem recebidos. O sinal de saída continua sinalizando o término da gravação e a saída dado contém a entrada lida.

Conforme previsto, ch0 = 1 e ch1 = 0 atribuiu o valor 2 em binário aos bits mais significativos da entrada, ch0 = 0 e ch1 = 1 atribuiu o valor 2 aos bits menos significativos entre 60ns e 70ns, o que equivale à 131074 em decimal. Após 70ns o valor dos bits menos significativos passou a ser 1, o que implica no decimal 131073 na saída. Aos 80ns, ch0 e

ch1 vão para nível alto e o sinal de término do recebimento de dados é setado até a borda de descida do sinal de controle, que reinicia o sinal continua.

5.0.12 Unidade de saída de dados

Figura 37 – Simulação do funcionamento da unidade de saída de dados



Fonte: Software Quartus Prime

Nesse teste um sinal de controle determina quando a saída é atualizada, verifica-se na imagem que a saída é atualizada na borda de subida do sinal de controle. Uma entrada com os valores 0 e 1000000 em hexadecimal é aplicada e as saídas correspondem aos *displays* de 7 segmentos do kit.

Conforme previsto, quando a entrada vale 00000000 em hexadecimal os oito *displays* apresentam apenas o segmento g apagado, ou seja, valor 0. Quando a entrada vale 01000000 apenas o *display* 6 é diferente de zero, ele apresenta somente os segmentos a e b acesos, ou seja, valor 1.

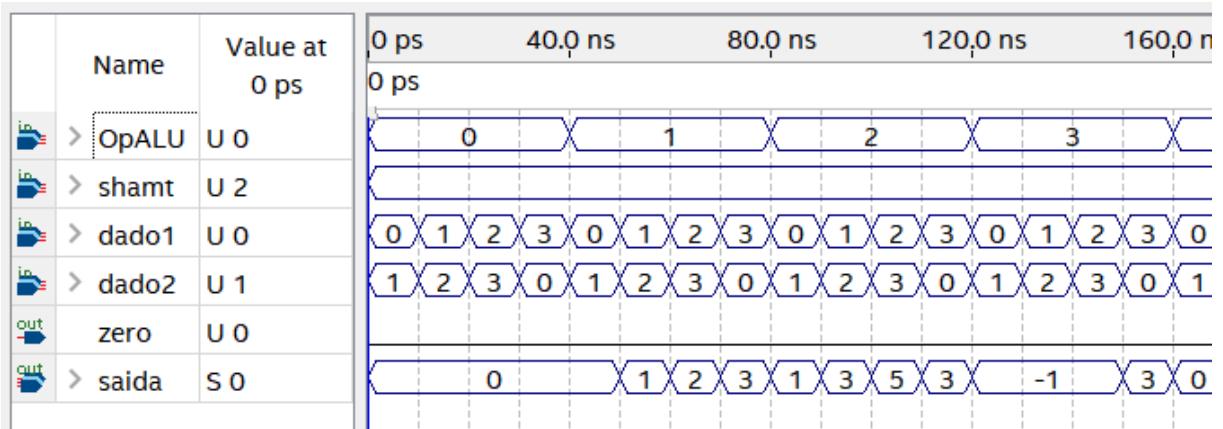
5.0.13 Unidade Lógica e Aritmética

A ULA executa sua operação com base no sinal de controle *OpALU*, um sinal que abrange todas as operações executadas pela ULA. Dessa forma, os resultados da ULA foram partitionados em várias imagens que permitem verificar seu funcionamento para cada operação.

O sinal de controle *OpALU* determina qual operação está sendo realizada, a entrada *shamt* contém a quantidade de bits que devem ser deslocadas em cado de instruções de deslocamentos. As entradas *dado1* e *dado2* simulam os dados entregues à ULA. A saída zero é útil em operação lógicas e a saída contém o resultado das operação.

5.0.13.1 Operações entrada (0), saída (1), adição (2) e subtração (3)

Figura 38 – Simulação do funcionamento da ULA

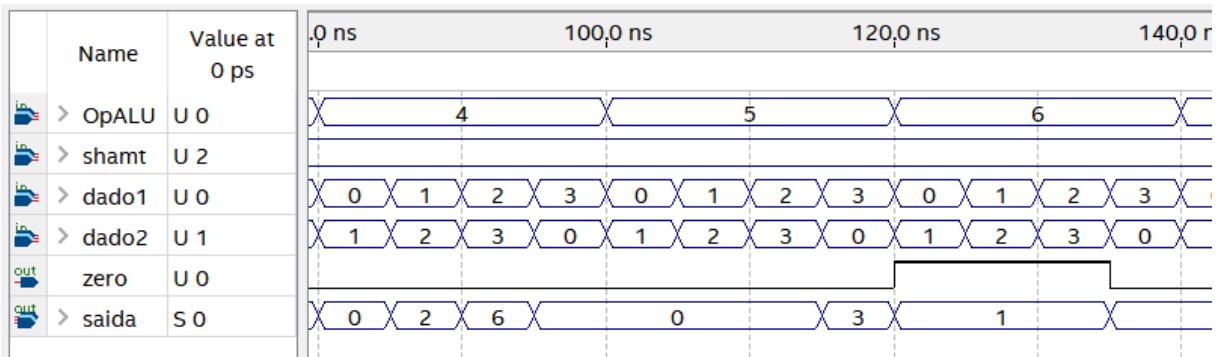


Fonte: Software Quartus Prime

- Quando $\text{OpALU} = 0$ ou 1 a operação é de entrada ou saída de dados, logo a saída da ULA é irrelevante.
- Quando $\text{OpALU} = 2$ a operação é de soma, logo as saídas estão conforme o previsto.
- Quando $\text{OpALU} = 3$ a operação é a subtração, logo as saídas estão conforme o previsto.

5.0.13.2 Operações multiplicação (4), divisão (5) e menor que (6)

Figura 39 – Simulação do funcionamento da ULA



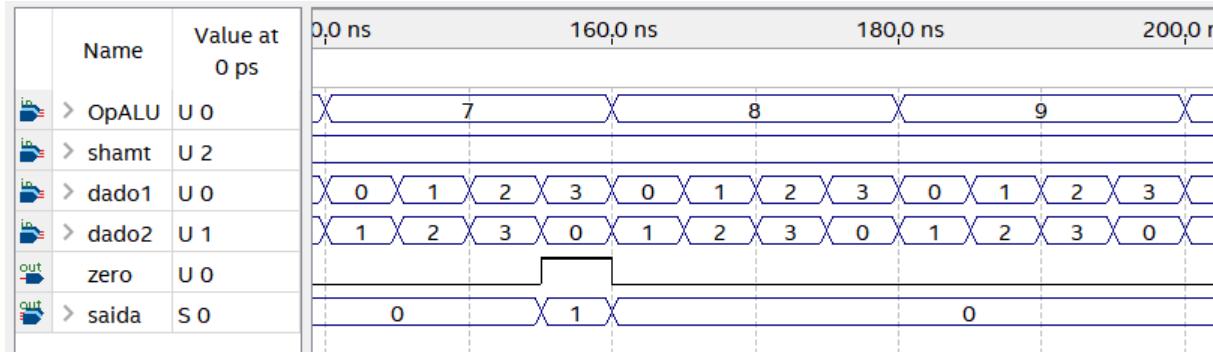
Fonte: Software Quartus Prime

- Quando $\text{OpALU} = 4$ a operação é a multiplicação, logo as saídas estão conforme o previsto.
- Quando $\text{OpALU} = 5$ a operação é a divisão, logo as saídas estão conforme previsto.

- Quando $\text{OpALU} = 6$ a operação é a de comparação menor que, ou seja, verifica se o dado1 é menor que o dado2. Nas três situações iniciais a hipótese é verdadeira e a *flag zero* vale 1, quando dado1 = 3 e dado2 = 0 a hipótese é falsa e *zero* vale 0.

5.0.13.3 Operações maior que (7), igual a (8) e *jump register* (9)

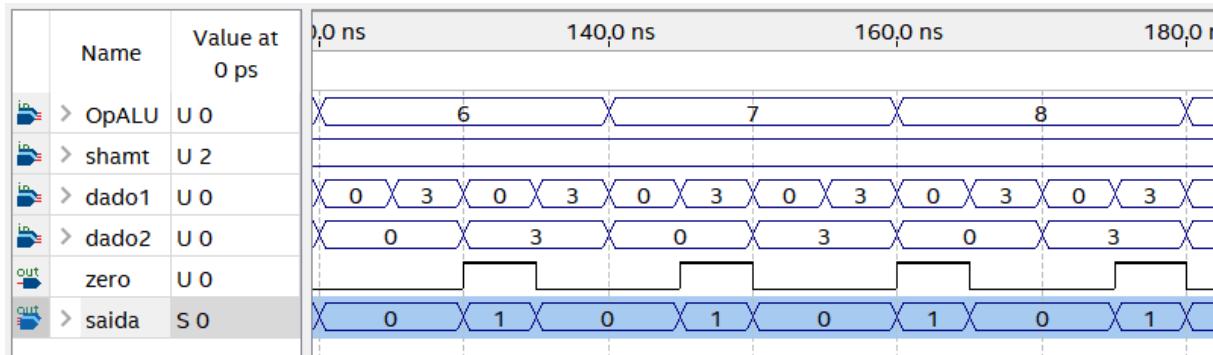
Figura 40 – Simulação do funcionamento da ULA



Fonte: Software Quartus Prime

- Quando $\text{OpALU} = 7$ a operação é a de comparação maior que, ou seja, verifica se o dado1 é maior que o dado2. Nas três situações iniciais a hipótese é falsa e a *flag zero* vale 0, quando dado1 = 3 e dado2 = 0 a hipótese é verdadeira e *zero* vale 1.
- Quando $\text{OpALU} = 8$ a operação é a de comparação igual a, ou seja, verifica se o dado1 e o dado2 são iguais. Nas quatro situações a hipótese é falsa e a *flag zero* vale 0.
- Quando $\text{OpALU} = 9$ a operação é a de salto *jump register* e os valores das saídas são irrelevantes, apenas o endereço da próxima instrução contido no registrador rs, correspondente ao dado1, é importante nessa instrução.

Figura 41 – Simulação do funcionamento da ULA

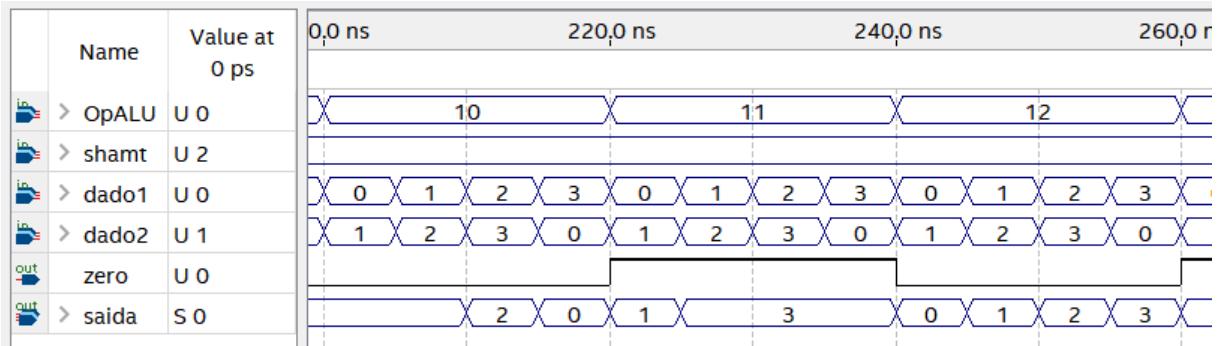


Fonte: Software Quartus Prime

A figura 41 atribui valores iguais às entradas para testar a operação de comparação igual a e as saídas obtidas estão de acordo com o esperado.

5.0.13.4 Operações and (10), or (11) e resto (12)

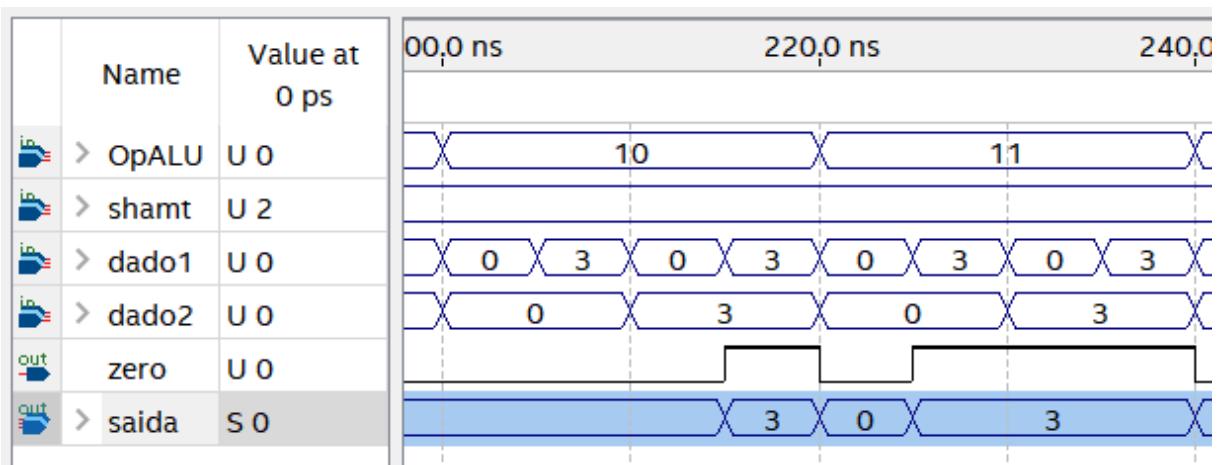
Figura 42 – Simulação do funcionamento da ULA



Fonte: Software Quartus Prime

- Quando OpALU = 10 a operação é a *and*, os quatro sinais de entrada são diferentes, dessa forma a saída *zero* é sempre nível baixo.
- Quando OpALU = 11 a operação é a *or*, conforme previsto a saída está sempre em nível alto pois não há entradas com dado1=0 e dado2=0.
- Quando OpALU = 12 a operação é de resto, as saídas estão de acordo com o previsto.

Figura 43 – Simulação do funcionamento da ULA

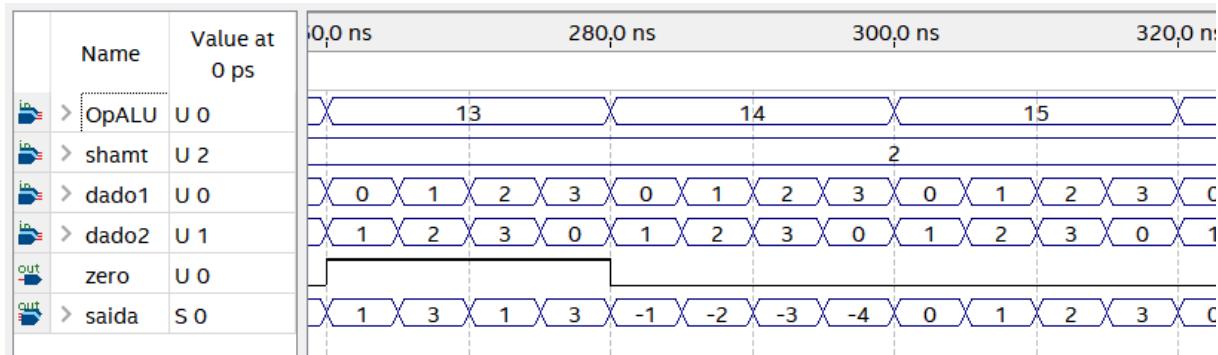


Fonte: Software Quartus Prime

A figura 43 apresenta as instruções *and* e *or* com entradas iguais e a saída é coerente com o resultado esperado.

5.0.13.5 Operações xor (13), not (14) e mover (15)

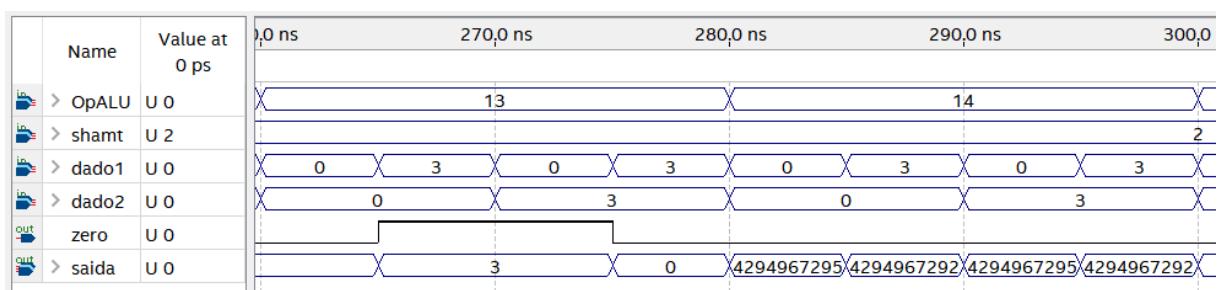
Figura 44 – Simulação do funcionamento da ULA



Fonte: Software Quartus Prime

Quando OpALU=13 a operação é *xor*, OpALU=14 a operação é *not* e OpALU=15 a operação é *move*. Todas as saídas ocorrem conforme o esperado.

Figura 45 – Simulação do funcionamento da ULA

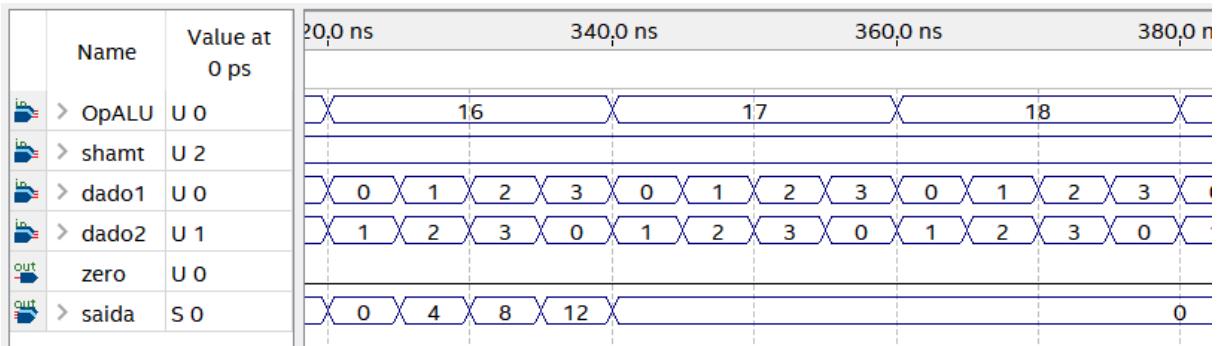


Fonte: Software Quartus Prime

A figura 45 apresenta as operações *and* e *or* com entradas iguais e as saídas são coerentes com o esperado.

5.0.13.6 Operações desloca esquerda (16), desloca direita (17) e ajuste de clock (18)

Figura 46 – Simulação do funcionamento da ULA

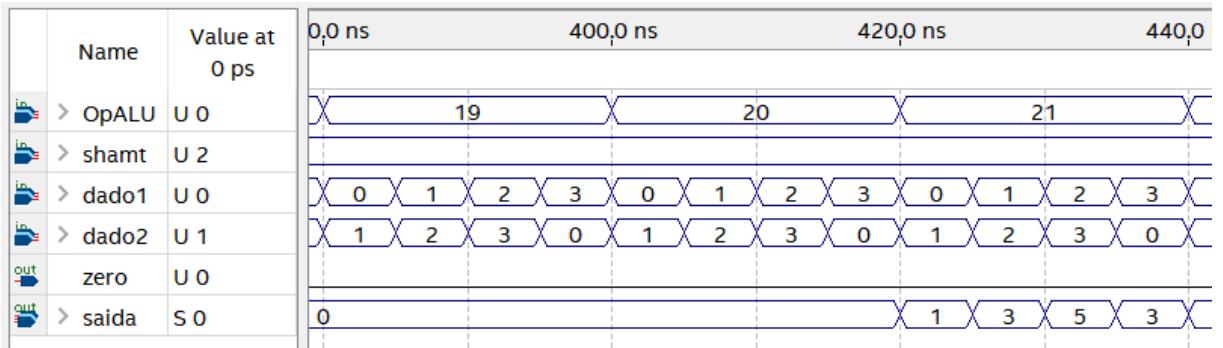


Fonte: Software Quartus Prime

Quando $\text{OpALU}=16$ a operação é de deslocamento do dado1 à esquerda, $\text{OpALU}=17$ a operação é de deslocamento do dado1 à direita e $\text{OpALU}=18$ a operação é de ajuste de *clock*. Todas as saídas ocorrem conforme o esperado, a saída é indiferente na operação de ajuste de *clock*.

5.0.13.7 Operações *jump* (19), *jump and link* (20) e *load* (21)

Figura 47 – Simulação do funcionamento da ULA

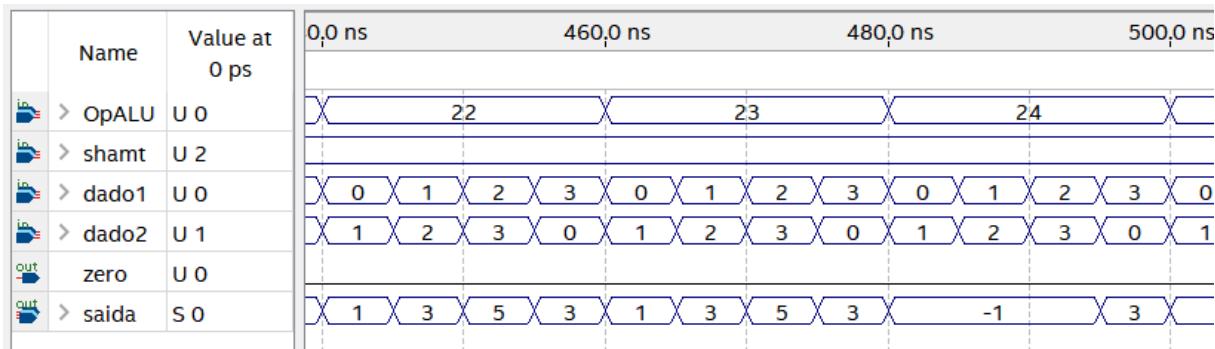


Fonte: Software Quartus Prime

Quando $\text{OpALU}=19$ a operação é de *jump*, $\text{OpALU}=20$ a operação é *jumo and link* e $\text{OpALU}=21$ a operação é *load*. As saídas estão de acordo com o previsto.

5.0.13.8 Operações *store* (22), adição com imediato (23) e subtração com imediato (24)

Figura 48 – Simulação do funcionamento da ULA

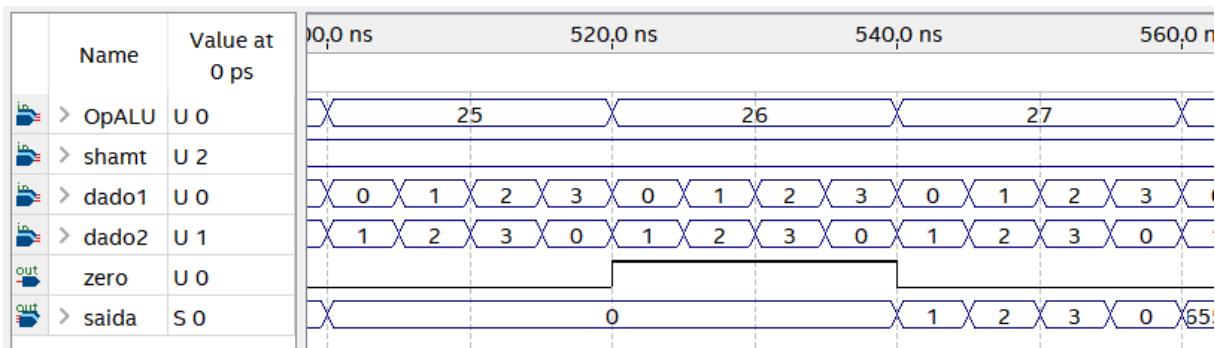


Fonte: Software Quartus Prime

Quando $\text{OpALU}=22$ a operação é *store*, $\text{OpALU}=23$ a operação é de adição com imediato e $\text{OpALU}=24$ a operação é de subtração com imediato. As saídas estão de acordo com o previsto.

5.0.13.9 Operações desvie se igual (25), desvie se diferente (26) e carregar imediato (27)

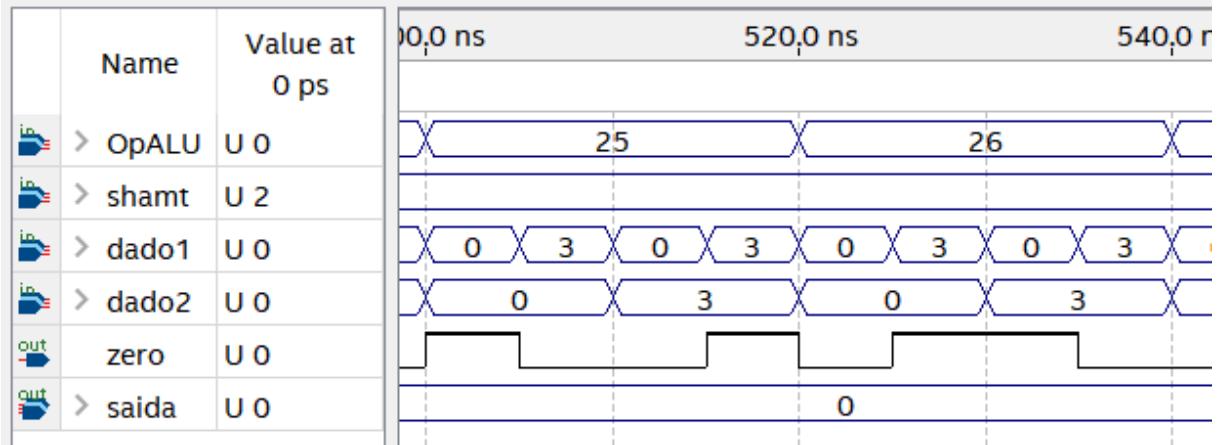
Figura 49 – Simulação do funcionamento da ULA



Fonte: Software Quartus Prime

Quando $\text{OpALU}=25$ a operação é desvie se igual, $\text{OpALU}=26$ a operação é desvie se diferente e $\text{OpALU}=27$ a operação é carregar imediato. As saídas estão de acordo com o previsto.

Figura 50 – Simulação do funcionamento da ULA

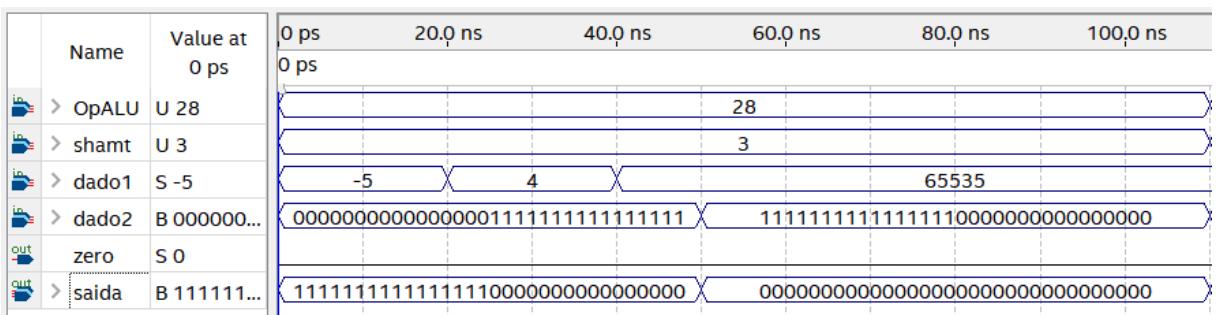


Fonte: Software Quartus Prime

A figura 50 apresenta as operações desvie se igual e desvie se diferente com entradas iguais. As saídas estão conforme o previsto.

5.0.13.10 Operação carrega superior imediato (28)

Figura 51 – Simulação do funcionamento da ULA



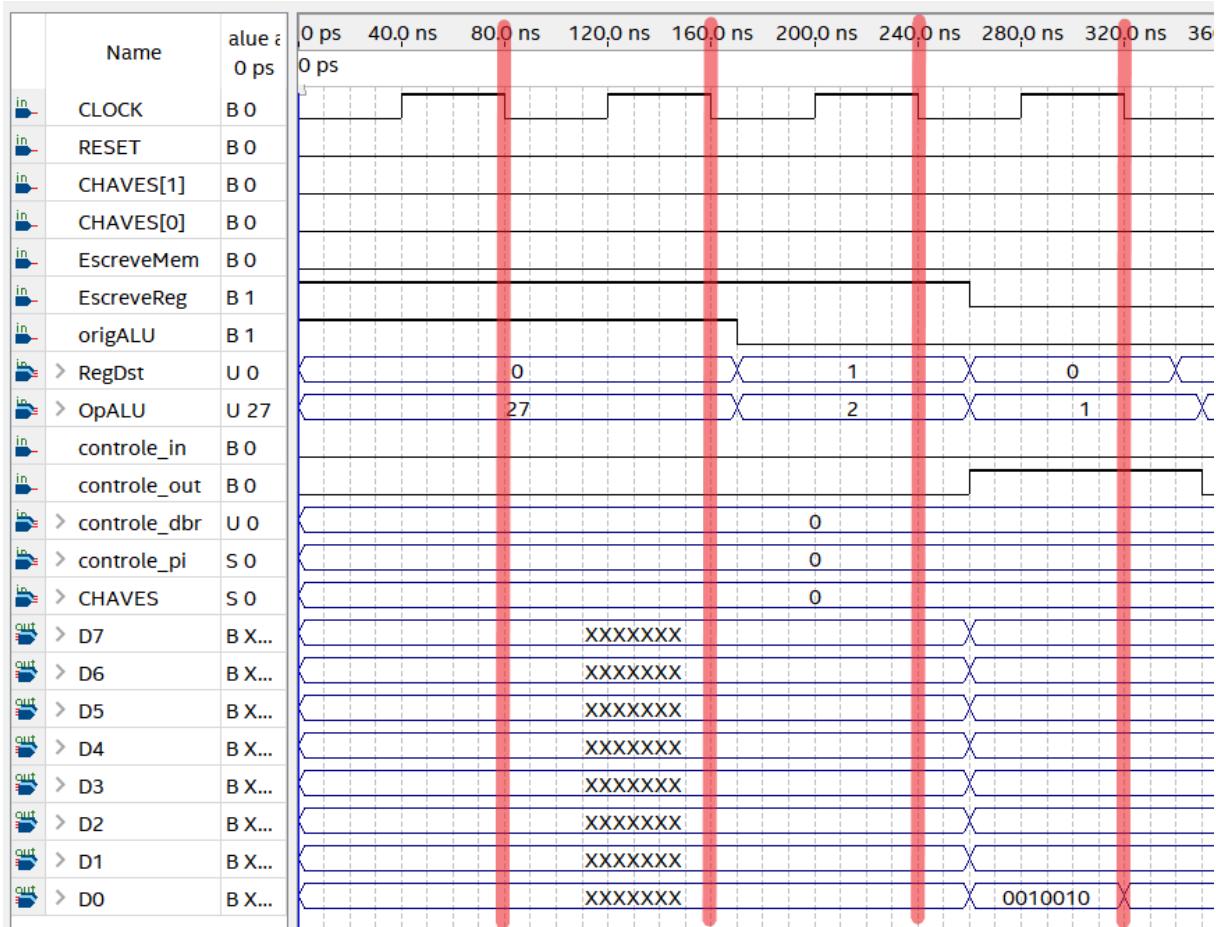
Fonte: Software Quartus Prime

Quando $\text{OpALU}=28$ a operação é de carregar superior imediato. As saídas estão conforme o previsto.

5.0.14 Unidades funcionais interconectadas

As unidades funcionais foram interconectadas entre si para simular a funcionalidade de algumas instruções.

Figura 52 – Simulação do funcionamento das unidades interconectadas



Fonte: Software Quartus Prime

O teste das unidades interconectadas utiliza quatro instruções, as duas primeiras são para carregar imediatos e atribuem os valores 2 e 3 aos registradores 1 e 2, respectivamente. A terceira instrução realiza a soma entre os registradores 1 e 2 e armazena o resultado no registrador 3. A quarta instrução utiliza o módulo de saída para imprimir o valor do registrador 3 nos *displays* de 7 segmentos.

As instruções utilizadas foram:

0000000000000000100000000000000011

0000000000000000100000000000000010

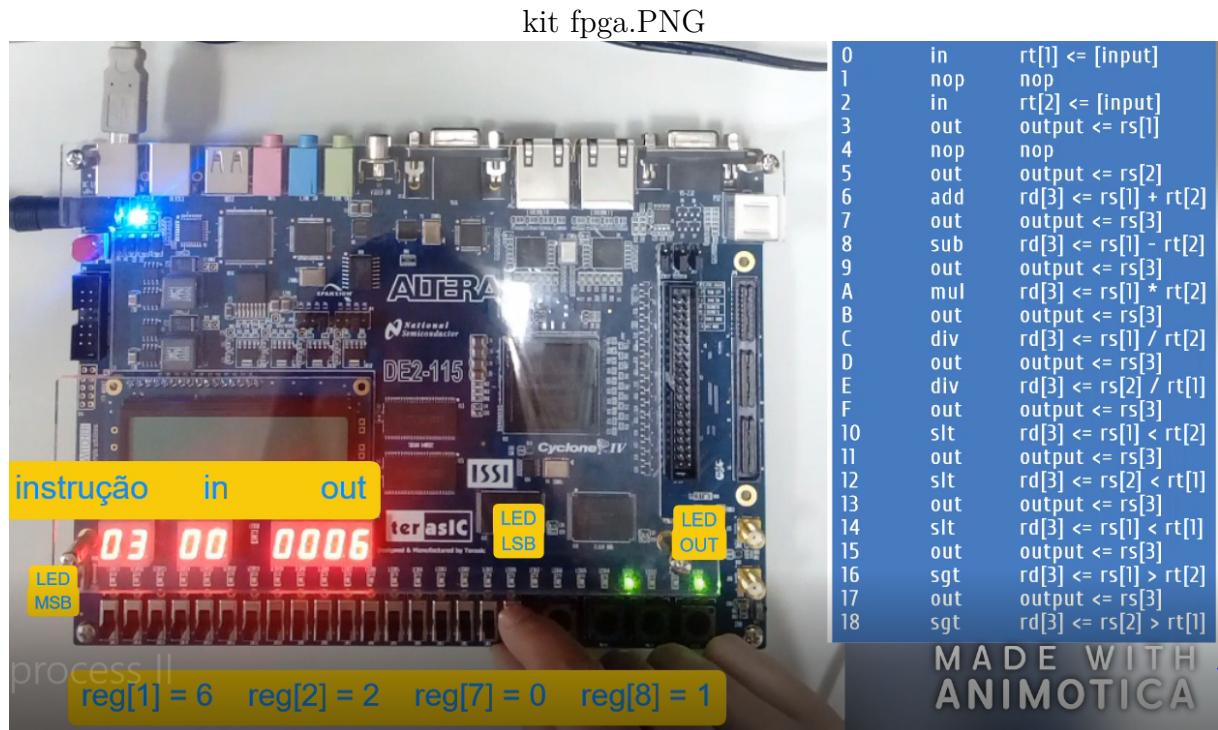
000000000010001000011000000000000

0000000000110000000000000000000000

Os resultados saíram conforme o previsto, a soma resultou 5 e os *displays* exibiriam o valor 00000005.

Após da elaboração da unidade de controle, união dos elementos e testes individuais, a validação final ocorre com a execução de um conjunto de instruções no processador descrito em lógica programável, conforme a figura ??.

Figura 53 – Registro do kit FPGA executando um conjunto de instruções qualquer.



Fonte: Elaborado pela autora.

6 Considerações Finais

A primeira etapa do projeto consistiu, inicialmente, da escolha de uma arquitetura base e de um conjunto de instruções que fosse suficiente para atender aos requisitos do projeto. Foi necessário rever os conceitos de arquitetura e organização de computadores para que essas escolhas pudessem ser feitas.

Em um segundo momento, com ambas as informações foi possível desenvolver um caminho de dados coerente com o projeto para então verificar a funcionalidade do conjunto de instruções sobre esse caminho de dados.

Uma grande dificuldade nessa etapa foi a elaboração do mecanismo de entrada e saída de dados, ainda há a necessidade de modificar o caminho de dados para atender às instruções de entrada e saída de dados, essa parte do projeto ainda não foi implementada. Outra questão ainda pendente é a implementação da instrução de ajuste do *clock*.

A segunda etapa do projeto consistiu da descrição da maior parte das unidades funcionais do sistema computacional em lógica programável através da linguagem de descrição de hardware Verilog. Na etapa atual de desenvolvimento somente a Unidade de Controle não foi desenvolvida.

A disponibilidade dos templates para as memórias de instruções e dados facilitou essa parte do desenvolvimento. Uma preocupação dessa etapa do processo foi assegurar que um ciclo de *clock* fosse suficiente para a execução correta de todas as instruções, principalmente aquelas que requerem a gravação de dados no banco de registradores, pois a gravação ocorre na borda de descida do *clock*, juntamente com a atualização do PC e, consequentemente, da instrução sendo executada.

Outra dificuldade foi a interconexão dos módulos desenvolvidos até o momento e a execução do teste para verificação do funcionamento sem uma Unidade de Controle para gerar os sinais de controle necessários ao funcionamento correto. Para contornar essa situação os sinais de controle atribuídos à Unidade de Controle foram inseridos como entradas no módulo de interconexão dos elementos.

A última etapa consistiu no desenvolvimento da Unidade de Controle, interconexão dela com os demais elementos e simulação do funcionamento através de formas de onda. Após a finalização do desenvolvimento, verificou-se o funcionamento no kit Altera DE2-115 com FPGA e executar os algoritmos de teste conforme a figura ??.

Um dos grandes desafios foi a implementação da unidade de entrada e saída independentes do sinal do *clock*, que se mostrou complicada quanto às mudanças de estado da chave de controle. Para contornar esse problema o projeto utiliza o sinal de *clock* nos

elementos de entrada e saída.

Portanto, ainda há diversas melhorias a serem executadas, porém o processador está atuando conforme o desejado até o momento.

Referências

- 1 IFRAH, G. *The Universal History of Computing: From the Abacus to the Quantum Computer*. New York: John Wiley & Sons, 2001. Citado na página [9](#).
- 2 PATTERSON, D. A. *Organização e projeto de computadores*. 3^a edição. ed. Rio de Janeiro: Elsevier, 2005. Citado 8 vezes nas páginas [12](#), [13](#), [26](#), [27](#), [28](#), [29](#), [30](#) e [31](#).
- 3 STALLINGS, W. *Arquitetura e organização de computadores*. 8^a edição. ed. São Paulo: Pearson Pratice Hall, 2010. Citado 10 vezes nas páginas [12](#), [14](#), [15](#), [16](#), [18](#), [19](#), [21](#), [22](#), [23](#) e [27](#).
- 4 FLOYD, T. L. *Sistemas digitais: fundamentos e aplicações*. 9th edition. ed. Porto Alegre/RS, BR: Bookman, 2007. Citado 4 vezes nas páginas [13](#), [32](#), [33](#) e [34](#).
- 5 TERASIC. *DE2-115 User Manual*. [S.l.], 2010. Citado na página [35](#).

Apêndices

APÊNDICE A – Apêndice A

Código completo da Unidade Lógica e Aritmética

```
1 module ULA (dado1, dado2, zero, saida, OpALU, shamt);
2
3 input [31:0] dado1; // contem o valor do primeiro operando
4 input [31:0] dado2; // contem o valor do segundo operando
5 input [4:0] shamt; // contem o numero de bits a serem deslocados em operacoes do tipo
6   shift
7 input [5:0] OpALU; // contem o sinal que determina a operacao a ser realizada com origem
8   na Unidade de Controle
9
10
11 output reg zero; // flag de controle para desvios e operacoes logicas
12 output reg [31:0] saida; // contem o valor resultante da operacao
13
14
15 initial
16 begin
17   zero = 32'd0; //inicia a flag zero com nivel baixo
18   saida = 1'd0; //inicia a saida com valor nulo
19 end
20
21
22 always @(*)
23 begin
24   case(OpALU) // monitora o sinal de controle da ULA
25     6'd2: // verifica se o sinal de controle vale 2
26     begin
27       saida = $signed(dado1)+$signed(dado2); //soma // atribui valor
28         dado1+dado2 a saida
29       zero = 0; // atribui valor nulo a flag zero
30     end
31     6'd3:
32     begin
33       saida = $signed(dado1)-$signed(dado2); //subtra o
34       zero = 0;
35     end
36     6'd4:
37     begin
38       saida = $signed(dado1)*$signed(dado2); //multiplica com valores
39         signed
40       zero = 0;
41     end
42     6'd5:
43     begin
44       if(dado2==32'd0) //trata divisao por 0
45       begin
46         saida = 32'd0;
47       end
48       else
49       begin
50         saida = $signed(dado1)/$signed(dado2); //divide com
51           valores signed
52       end
53       zero = 0;
54     end
55   end
56 end
```

```

48      begin
49          if($signed(dado1)<$signed(dado2))
50              begin
51                  saida = 32'd1;
52                  zero = 1;
53              end
54          else if($signed(dado1)>=$signed(dado2))
55              begin
56                  saida=32'd0;
57                  zero=0;
58              end
59      end
60      6'd7: //maior que
61      begin
62          if($signed(dado1)>$signed(dado2))
63              begin
64                  saida = 32'd1;
65                  zero = 1;
66              end
67          else if($signed(dado1)<=$signed(dado2))
68              begin
69                  saida=32'd0;
70                  zero=0;
71              end
72      end
73      6'd8: //igual a
74      begin
75          if($signed(dado1)==$signed(dado2))
76              begin
77                  saida = 32'd1;
78                  zero = 1;
79              end
80          else
81              begin
82                  saida=32'd0;
83                  zero=0;
84              end
85      end
86      6'd10://and
87      begin
88          saida = (dado1 && dado2);
89          zero = 0;
90      end
91      6'd11://or
92      begin
93          saida = (dado1 || dado2);
94          zero = 0;
95      end
96      6'd12://resto
97      begin
98          saida = ($signed(dado1) % dado2); //resto
99          zero = 0;
100     end
101     6'd13://xor
102     begin
103         saida = (dado1 ^ dado2);
104         zero=0;
105     end
106     6'd14:
107     begin

```

```
108                     saida = ~dado1; // nega o
109                     zero = 0;
110                 end
111             6'd15:
112             begin
113                 saida = dado1; //move
114                 zero = 0;
115             end
116             6'd16:
117             begin
118                 saida = (dado1 <<< shamt); //desloca a esquerda
119                 zero = 0;
120             end
121             6'd17:
122             begin
123                 saida = (dado1 >>> shamt); //desloca a direita
124                 zero = 0;
125             end
126             6'd21:
127             begin
128                 saida = dado1+dado2; //load
129                 zero = 0;
130             end
131             6'd22:
132             begin
133                 saida = dado1+dado2; //store
134                 zero = 0;
135             end
136             6'd23:
137             begin
138                 saida = ($signed(dado1)+$signed(dado2)); //addi
139                 zero = 0;
140             end
141             6'd24:
142             begin
143                 saida = ($signed(dado1)-$signed(dado2)); //subi
144                 zero = 0;
145             end
146             6'd25: //desvia se igual
147             begin
148                 if($signed(dado1)==$signed(dado2))
149                     begin
150                         saida = 32'd1;
151                         zero = 1;
152                     end
153                 else
154                     begin
155                         saida = 32'd0;
156                         zero=0;
157                     end
158             end
159             6'd26: //desvia se diferente
160             begin
161                 if($signed(dado1)!=$signed(dado2))
162                     begin
163                         saida = 32'd1;
164                         zero = 1;
165                     end
166                 else
167                     begin
```

```
168                     saida = 32'd0;
169                     zero = 0;
170                 end
171             end
172         6'd27:
173         begin
174             saida = dado2; //carregar imediato
175             zero = 0;
176         end
177         6'd28:
178         begin
179             saida = (dado2 <<< 16); //carregar superior imediato
180             zero = 0;
181         end
182     default:
183     begin
184         saida = 32'd0;
185         zero = 0;
186     end
187     endcase
188 end
189
190 endmodule
```

APÊNDICE B – Apêndice B

Código completo da Unidade de Saída

```
1 module saida (dado, controle, d0, d1, d2, d3, d4, d5, d6, d7, neg, continua, ch0, clk);
2
3 input clk;
4 input [31:0] dado;
5 input controle;
6 input ch0;
7 output reg [6:0] d0, d1, d2, d3, d4, d5, d6, d7;
8 output reg neg;
9 output reg continua;
10 reg [31:0] entrada;
11
12 reg ant, at;
13
14 reg sinal;
15 reg [2:0] aux;
16
17 initial
18 begin
19     continua = 1'd1;
20     sinal = 0;
21     aux=3'd0;
22     ant=0;
23     at=0;
24 end
25
26 always @ (negedge continua)
27 begin
28     if ($signed(dado) < $signed(32'd0))
29         begin
30             entrada = ~(dado - 32'd1);
31             neg = 1;
32         end
33     else
34         begin
35             entrada = dado;
36             neg = 0;
37         end
38 end
39
40 always@ (controle or at)
41 begin
42     if (controle)
43         begin
44             if (!ant && !at)
45                 begin
46                     continua = 0;
47                 end
48             else if (!ant && at)
49                 begin
50                     continua = 0;
51                 end
52             else if (ant && at)
```

```

53         begin
54             continua = 0;
55         end
56         else if(ant && !at)
57             begin
58                 continua = 1;
59             end
60         end
61     else
62         begin
63             continua=1;
64         end
65     end
66
67 always@(posedge clk)
68 begin
69     if(controle)
70     begin
71         if(ch0)
72             begin
73                 if(! ant && !at)
74                     at=1;
75                 else if(! ant && at)
76                     ant=1;
77             end
78         else
79             begin
80                 if(! ant && !at)
81                 begin
82                     ant=0;
83                     at=0;
84                 end
85                 else if(ant && at)
86                     at=0;
87             end
88         end
89     else
90         begin
91             ant=0;
92             at=0;
93         end
94     end
95
96 always@(entrada)
97 begin
98     case (entrada[3:0])
99         4'b0000: d0=7'b1000000;
100        4'b0001: d0=7'b1111001;
101        4'b0010: d0=7'b0100100;
102        4'b0011: d0=7'b0110000;
103        4'b0100: d0=7'b0011001;
104        4'b0101: d0=7'b0010010;
105        4'b0110: d0=7'b0000010;
106        4'b0111: d0=7'b1111000;
107        4'b1000: d0=7'b0000000;
108        4'b1001: d0=7'b0010000;
109        4'b1010: d0=7'b0001000;
110        4'b1011: d0=7'b0000011;
111        4'b1100: d0=7'b1000110;
112        4'b1101: d0=7'b0100001;

```

```
113          4'b1110: d0=7'b0000110;
114          4'b1111: d0=7'b0001110;
115      endcase
116  case (entrada[7:4])
117      4'b0000: d1=7'b1000000;
118      4'b0001: d1=7'b1111001;
119      4'b0010: d1=7'b0100100;
120      4'b0011: d1=7'b0110000;
121      4'b0100: d1=7'b0011001;
122      4'b0101: d1=7'b0010010;
123      4'b0110: d1=7'b0000010;
124      4'b0111: d1=7'b1111000;
125      4'b1000: d1=7'b0000000;
126      4'b1001: d1=7'b0010000;
127      4'b1010: d1=7'b0001000;
128      4'b1011: d1=7'b0000011;
129      4'b1100: d1=7'b1000110;
130      4'b1101: d1=7'b0100001;
131      4'b1110: d1=7'b0000110;
132      4'b1111: d1=7'b0001110;
133  endcase
134  case (entrada[11:8])
135      4'b0000: d2=7'b1000000;
136      4'b0001: d2=7'b1111001;
137      4'b0010: d2=7'b0100100;
138      4'b0011: d2=7'b0110000;
139      4'b0100: d2=7'b0011001;
140      4'b0101: d2=7'b0010010;
141      4'b0110: d2=7'b0000010;
142      4'b0111: d2=7'b1111000;
143      4'b1000: d2=7'b0000000;
144      4'b1001: d2=7'b0010000;
145      4'b1010: d2=7'b0001000;
146      4'b1011: d2=7'b0000011;
147      4'b1100: d2=7'b1000110;
148      4'b1101: d2=7'b0100001;
149      4'b1110: d2=7'b0000110;
150      4'b1111: d2=7'b0001110;
151  endcase
152  case (entrada[15:12])
153      4'b0000: d3=7'b1000000;
154      4'b0001: d3=7'b1111001;
155      4'b0010: d3=7'b0100100;
156      4'b0011: d3=7'b0110000;
157      4'b0100: d3=7'b0011001;
158      4'b0101: d3=7'b0010010;
159      4'b0110: d3=7'b0000010;
160      4'b0111: d3=7'b1111000;
161      4'b1000: d3=7'b0000000;
162      4'b1001: d3=7'b0010000;
163      4'b1010: d3=7'b0001000;
164      4'b1011: d3=7'b0000011;
165      4'b1100: d3=7'b1000110;
166      4'b1101: d3=7'b0100001;
167      4'b1110: d3=7'b0000110;
168      4'b1111: d3=7'b0001110;
169  endcase
170  case (entrada[19:16])
171      4'b0000: d4=7'b1000000;
172      4'b0001: d4=7'b1111001;
```

```

173          4'b0010: d4=7'b0100100;
174          4'b0011: d4=7'b0110000;
175          4'b0100: d4=7'b0011001;
176          4'b0101: d4=7'b0010010;
177          4'b0110: d4=7'b0000010;
178          4'b0111: d4=7'b1111000;
179          4'b1000: d4=7'b0000000;
180          4'b1001: d4=7'b0010000;
181          4'b1010: d4=7'b0001000;
182          4'b1011: d4=7'b0000011;
183          4'b1100: d4=7'b1000110;
184          4'b1101: d4=7'b0100001;
185          4'b1110: d4=7'b0000110;
186          4'b1111: d4=7'b00001110;
187      endcase
188  case (entrada[23:20])
189          4'b0000: d5=7'b1000000;
190          4'b0001: d5=7'b1111001;
191          4'b0010: d5=7'b0100100;
192          4'b0011: d5=7'b0110000;
193          4'b0100: d5=7'b0011001;
194          4'b0101: d5=7'b0010010;
195          4'b0110: d5=7'b0000010;
196          4'b0111: d5=7'b1111000;
197          4'b1000: d5=7'b0000000;
198          4'b1001: d5=7'b0010000;
199          4'b1010: d5=7'b0001000;
200          4'b1011: d5=7'b0000011;
201          4'b1100: d5=7'b1000110;
202          4'b1101: d5=7'b0100001;
203          4'b1110: d5=7'b0000110;
204          4'b1111: d5=7'b00001110;
205      endcase
206  case (entrada[27:24])
207          4'b0000: d6=7'b1000000;
208          4'b0001: d6=7'b1111001;
209          4'b0010: d6=7'b0100100;
210          4'b0011: d6=7'b0110000;
211          4'b0100: d6=7'b0011001;
212          4'b0101: d6=7'b0010010;
213          4'b0110: d6=7'b0000010;
214          4'b0111: d6=7'b1111000;
215          4'b1000: d6=7'b0000000;
216          4'b1001: d6=7'b0010000;
217          4'b1010: d6=7'b0001000;
218          4'b1011: d6=7'b0000011;
219          4'b1100: d6=7'b1000110;
220          4'b1101: d6=7'b0100001;
221          4'b1110: d6=7'b0000110;
222          4'b1111: d6=7'b00001110;
223      endcase
224  case (entrada[31:28])
225          4'b0000: d7=7'b1000000;
226          4'b0001: d7=7'b1111001;
227          4'b0010: d7=7'b0100100;
228          4'b0011: d7=7'b0110000;
229          4'b0100: d7=7'b0011001;
230          4'b0101: d7=7'b0010010;
231          4'b0110: d7=7'b0000010;
232          4'b0111: d7=7'b1111000;

```

```
233          4'b1000: d7=7'b0000000;
234          4'b1001: d7=7'b0010000;
235          4'b1010: d7=7'b0001000;
236          4'b1011: d7=7'b0000011;
237          4'b1100: d7=7'b1000110;
238          4'b1101: d7=7'b0100001;
239          4'b1110: d7=7'b00000110;
240          4'b1111: d7=7'b00001110;
241      endcase
242  end
243 endmodule
```

APÊNDICE C – Apêndice C

Código completo da Unidade de Controle

```

1 module UC (instr, zero, controle_out, RegDst, EscreveReg, OrigALU, OpALU, EscreveMem,
2           controle_dbr, controle_in, controle_pi);
3
4 input [31:0] instr;
5 input zero;
6 output reg controle_out;
7 output reg [1:0] RegDst;
8 output reg EscreveReg;
9 output reg OrigALU;
10 output reg [5:0] OpALU;
11 output reg EscreveMem;
12 output reg [1:0] controle_dbr;
13 output reg controle_in;
14 output reg [1:0] controle_pi;
15
16 initial
17 begin
18     controle_out = 1'd0;
19     RegDst = 2'd0;
20     EscreveReg = 1'd0;
21     OrigALU = 1'd0;
22     OpALU = 6'd0;
23     EscreveMem = 1'd0;
24     controle_dbr = 2'd0;
25     controle_in = 1'd0;
26     controle_pi = 2'd0;
27 end
28
29 always@(instr)
30 begin
31     if(instr[31:26]==6'd0 && instr[5:0]==6'd0)
32         controle_in=1;
33     else
34         controle_in=0;
35 end
36
37 always@(instr)
38 begin
39     if(instr[31:26]==6'd0 && instr[5:0]==6'd1)
40         controle_out=1;
41     else
42         controle_out=0;
43 end
44
45 always @(*(instr or zero))
46 begin
47     case(instr[31:26])
48         6'd0:
49             begin
50                 case(instr[5:0])
51                     6'd0:

```

```
52      begin
53
54          RegDst = 0;
55          EscreveReg = 1;
56          OrigALU = 0;
57          OpALU = 0;
58          EscreveMem = 0;
59          controle_dbr = 3;
60          controle_pi = 0;
61      end
62      6'd1:
63      begin
64
65          RegDst = 0;
66          EscreveReg = 0;
67          OrigALU = 0;
68          OpALU = 1;
69          EscreveMem = 0;
70          controle_dbr = 0;
71          controle_pi = 0;
72      end
73      6'd2:
74      begin
75
76          RegDst = 1;
77          EscreveReg = 1;
78          OrigALU = 0;
79          OpALU = 2;
80          EscreveMem = 0;
81          controle_dbr = 0;
82          controle_pi = 0;
83      end
84      6'd3:
85      begin
86
87          RegDst = 1;
88          EscreveReg = 1;
89          OrigALU = 0;
90          OpALU = 3;
91          EscreveMem = 0;
92          controle_dbr = 0;
93          controle_pi = 0;
94      end
95      6'd4:
96      begin
97
98          RegDst = 1;
99          EscreveReg = 1;
100         OrigALU = 0;
101         OpALU = 4;
102         EscreveMem = 0;
103         controle_dbr = 0;
104         controle_pi = 0;
105     end
106     6'd5:
107     begin
108
109         RegDst = 1;
110         EscreveReg = 1;
111         OrigALU = 0;
```

```

112          OpALU = 5;
113          EscreveMem = 0;
114          controle_dbr = 0;
115          controle_pi = 0;
116      end
117 6'd6:
118 begin
119
120          RegDst = 1;
121          EscreveReg = 1;
122          OrigALU = 0;
123          OpALU = 6;
124          EscreveMem = 0;
125          controle_dbr = 0;
126          controle_pi = 0;
127      end
128 6'd7:
129 begin
130
131          RegDst = 1;
132          EscreveReg = 1;
133          OrigALU = 0;
134          OpALU = 7;
135          EscreveMem = 0;
136          controle_dbr = 0;
137          controle_pi = 0;
138      end
139 6'd8:
140 begin
141
142          RegDst = 1;
143          EscreveReg = 1;
144          OrigALU = 0;
145          OpALU = 8;
146          EscreveMem = 0;
147          controle_dbr = 0;
148          controle_pi = 0;
149      end
150 6'd9:
151 begin
152
153          RegDst = 1;
154          EscreveReg = 0;
155          OrigALU = 0;
156          OpALU = 9;
157          EscreveMem = 0;
158          controle_dbr = 0;
159          controle_pi = 3;
160      end
161 6'd10:
162 begin
163
164          RegDst = 1;
165          EscreveReg = 1;
166          OrigALU = 0;
167          OpALU = 10;
168          EscreveMem = 0;
169          controle_dbr = 0;
170          controle_pi = 0;
171      end

```

```
172          6'd11:
173          begin
174
175              RegDst = 1;
176              EscreveReg = 1;
177              OrigALU = 0;
178              OpALU = 11;
179              EscreveMem = 0;
180              controle_dbr = 0;
181              controle_pi = 0;
182          end
183          6'd12:
184          begin
185
186              RegDst = 1;
187              EscreveReg = 1;
188              OrigALU = 0;
189              OpALU = 12;
190              EscreveMem = 0;
191              controle_dbr = 0;
192              controle_pi = 0;
193          end
194          6'd13:
195          begin
196
197              RegDst = 1;
198              EscreveReg = 1;
199              OrigALU = 0;
200              OpALU = 13;
201              EscreveMem = 0;
202              controle_dbr = 0;
203              controle_pi = 0;
204          end
205          6'd14:
206          begin
207
208              RegDst = 1;
209              EscreveReg = 1;
210              OrigALU = 0;
211              OpALU = 14;
212              EscreveMem = 0;
213              controle_dbr = 0;
214              controle_pi = 0;
215          end
216          6'd15:
217          begin
218
219              RegDst = 1;
220              EscreveReg = 1;
221              OrigALU = 0;
222              OpALU = 15;
223              EscreveMem = 0;
224              controle_dbr = 0;
225              controle_pi = 0;
226          end
227          6'd16:
228          begin
229
230              RegDst = 1;
231              EscreveReg = 1;
```

```

232                         OrigALU = 0;
233                         OpALU = 16;
234                         EscreveMem = 0;
235                         controle_dbr = 0;
236                         controle_pi = 0;
237                     end
238                 6'd17:
239                 begin
240
241                         RegDst = 1;
242                         EscreveReg = 1;
243                         OrigALU = 0;
244                         OpALU = 17;
245                         EscreveMem = 0;
246                         controle_dbr = 0;
247                         controle_pi = 0;
248                     end
249                 default:
250                 begin
251
252                         RegDst = 0;
253                         EscreveReg = 0;
254                         OrigALU = 0;
255                         OpALU = 0;
256                         EscreveMem = 0;
257                         controle_dbr = 0;
258                         controle_pi = 0;
259                     end
260                 endcase
261             end
262         6'b000001:
263         begin
264             RegDst = 0;
265             EscreveReg = 0;
266             OrigALU = 0;
267             OpALU = 18;
268             EscreveMem = 0;
269             controle_dbr = 0;
270             controle_pi = 0;
271         end
272         6'b0000010:
273         begin
274
275             RegDst = 0;
276             EscreveReg = 0;
277             OrigALU = 0;
278             OpALU = 19;
279             EscreveMem = 0;
280             controle_dbr = 0;
281             controle_pi = 2;
282         end
283         6'b0000011:
284         begin
285
286             RegDst = 2;
287             EscreveReg = 1;
288             OrigALU = 0;
289             OpALU = 20;
290             EscreveMem = 0;
291             controle_dbr = 2;

```

```
292         controle_pi = 2;
293     end
294     6'b000100:
295     begin
296
297         RegDst = 0;
298         EscreveReg = 1;
299         OrigALU = 1;
300         OpALU = 21;
301         EscreveMem = 0;
302         controle_dbr = 1;
303         controle_pi = 0;
304     end
305     6'b000101:
306     begin
307
308         RegDst = 1;
309         EscreveReg = 0;
310         OrigALU = 1;
311         OpALU = 22;
312         EscreveMem = 1;
313         controle_dbr = 1;
314         controle_pi = 0;
315     end
316     6'b000110:
317     begin
318
319         RegDst = 0;
320         EscreveReg = 1;
321         OrigALU = 1;
322         OpALU = 23;
323         EscreveMem = 0;
324         controle_dbr = 0;
325         controle_pi = 0;
326     end
327     6'b000111:
328     begin
329
330         RegDst = 0;
331         EscreveReg = 1;
332         OrigALU = 1;
333         OpALU = 24;
334         EscreveMem = 0;
335         controle_dbr = 0;
336         controle_pi = 0;
337     end
338     6'b001000:
339     begin
340
341         RegDst = 0;
342         EscreveReg = 0;
343         OrigALU = 0;
344         OpALU = 25;
345         EscreveMem = 0;
346         controle_dbr = 0;
347         if(zero)
348             begin
349                 controle_pi = 2;
350             end
351         else if (!zero)
```

```

352          begin
353              controle_pi = 0;
354          end
355      end
356      6'b001001:
357      begin
358
359          RegDst = 0;
360          EscreveReg = 0;
361          OrigALU = 0;
362          OpALU = 26;
363          EscreveMem = 0;
364          controle_dbr = 0;
365          if(zero)
366              begin
367                  controle_pi = 2;
368              end
369          else if (!zero)
370              begin
371                  controle_pi = 0;
372              end
373          end
374      6'b001010:
375      begin
376
377          RegDst = 0;
378          EscreveReg = 1;
379          OrigALU = 1;
380          OpALU = 27;
381          EscreveMem = 0;
382          controle_dbr = 0;
383          controle_pi = 0;
384      end
385      6'b001011:
386      begin
387
388          RegDst = 0;
389          EscreveReg = 1;
390          OrigALU = 1;
391          OpALU = 28;
392          EscreveMem = 0;
393          controle_dbr = 0;
394          controle_pi = 0;
395      end
396      default:
397      begin
398
399          RegDst = 0;
400          EscreveReg = 0;
401          OrigALU = 0;
402          OpALU = 0;
403          EscreveMem = 0;
404          controle_dbr = 0;
405          controle_pi = 0;
406      end
407  endcase
408
409 end
410 endmodule

```