UFAM - Engenharia da Computação

DESENVOLVIMENTO DE UM MODELO EM VHDL PARA AVALIAÇÃO DE CORRETUDE E DESEMPENHO DO PROCESSADOR MIPS USANDO ARQUITETURAS BASEADAS EM MULTICICLO E *PIPELINE*

Phillipe Arantes Pereira

Monografia de Graduação apresentada à Coordenação de Engenharia da Computação, da Universidade Federal do Amazonas, como parte dos requisitos necessários à obtenção do título de Engenheiro da Computação.

Orientador: Lucas Carvalho Cordeiro

Manaus

Março de 2015

DESENVOLVIMENTO DE UM MODELO EM VHDL PARA AVALIAÇÃO DE CORRETUDE E DESEMPENHO DO PROCESSADOR MIPS USANDO ARQUITETURAS BASEADAS EM MULTICICLO E *PIPELINE*

Phillipe Arantes Pereira

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO CURSO DE ENGENHARIA DA COMPUTAÇÃO DA UNIVERSIDADE FEDERAL DO AMAZONAS COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO DA COMPUTAÇÃO.

Aprovada por:

Prof. Lucas Carvalho Cordeiro, Ph.D.

Prof. Carlos Augusto de Moraes Cruz, D.Sc.

Prof. Celso Barbosa Carvalho, D.Sc

MANAUS, AM – BRASIL

MARÇO DE 2015

Pereira, Phillipe Arantes

Desenvolvimento de um Modelo em VHDL para Avaliação de Corretude e Desempenho do Processador MIPS usando Arquiteturas Baseadas em Multiciclo e *Pipeline*/Phillipe Arantes Pereira. – Manaus: UFAM, 2015.

XV, 67 p.: il.; 29, 7cm.

Orientador: Lucas Carvalho Cordeiro

Monografia (graduação) – UFAM / Curso de Engenharia da Computação, 2015.

Referências Bibliográficas: p. 66 – 67.

1. MIPS. 2. VHDL. 3. FPGA. I. Cordeiro, Lucas Carvalho. II. Universidade Federal do Amazonas, UFAM, Curso de Engenharia da Computação. III. Título.

Dedico este trabalho aos meus avós, Maria de Lourdes e José, "In Memorian", e Sahara, e aos meus pais, por me ensinarem a escolher os caminhos na vida.

Agradecimentos

- Aos meus pais Wilton e Aline e a minha irmã Raphaella por tudo que me proporcionaram, e todo apoio e confiança depositados em mim para chegar até aqui.
- À minha namorada Aline, por estar ao meu lado e me apoiar durante os anos em que estamos juntos.
- Ao meu orientador Prof. Lucas Carvalho Cordeiro por toda paciência, apoio e oportunidades visando uma melhor formação profissional.
- Aos meus amigos da longa caminhada, João e Carlos, pelos momentos de descontração e estudo em que estiveram presentes.
- À Universidade Federal do Amazonas por proporcionar a estrutura e a oportunidade para realizar o curso de Engenharia da Computação.
- A todos que contribuíram de forma direta ou indireta para que este trabalho fosse realizado.

Resumo da Monografia apresentada à UFAM como parte dos requisitos necessários para a obtenção do grau de Engenheiro da Computação

DESENVOLVIMENTO DE UM MODELO EM VHDL PARA AVALIAÇÃO DE CORRETUDE E DESEMPENHO DO PROCESSADOR MIPS USANDO ARQUITETURAS BASEADAS EM MULTICICLO E *PIPELINE*

Phillipe Arantes Pereira

Março/2015

Orientador: Lucas Carvalho Cordeiro

Programa: Engenharia da Computação

Este trabalho apresenta a implementação de um processador MIPS utilizando a linguagem de descrição de hardware VHDL para avaliar corretude e desempenho sobre arquiteturas baseadas em multiciclo e pipeline. O objetivo é apresentar em detalhes as características dos processadores MIPS e avaliar o correto funcionamento das arquiteturas projetadas, usando testes sobre os componentes e comparando o desempenho do processador através do número de ciclos de relógio por instrução, utilizando a simulação de benchmarks no software Qsim e embarcando o processador em uma placa de FPGA para obtenção de resultados práticos. A simulação mostrou que a arquitetura pipeline apresenta significantes melhorias no desempenho em relação à multiciclo. Nas simulações no FPGA, com os parâmetros de entrada dos benchmarks maiores, pode ser observado um ligeiro aumento no desempenho.

Abstract of Monograph presented to UFAM as a partial fulfillment of the requirements for the degree of Engineer

DEVELOPMENT OF A MODEL IN VHDL FOR CORRECTNESS AND PERFORMANCE EVALUATION USING MIPS PROCESSOR ARCHITECTURES-BASED ON MULTICYCLE AND PIPELINE

Phillipe Arantes Pereira

March/2015

Advisor: Lucas Carvalho Cordeiro

Department: Computer Engineering

This work presents a MIPS processor's implementation using the VHDL hardware description language for correctness and evaluation performance on architectures based on multi-cycle and pipeline. The goal is to present details of the MIPS processor and to evaluate the correct behavior of the designed architectures, testing the components as well as comparing the processor's performance via instructions per clock-cycles number. The simulation tool Qsim and the FPGA platform are used to validate the implementation. The experimental results show that the pipeline architecture has significant performance improvements over multi-cycle architecture. During the simulations on FPGA, with higher input parameters benchmarks, a slight performance improvement can be observed.

Sumário

Lista de Figuras

Lista de Tabelas

Abreviações

ASIC - ***A****pplication* ***S****pecific* ***I****ntegrated* ***C****ircuits*

CP - **C**ontador de **P**rogramas

CPI - **C**iclos **p**or **I**nstrução

FPGA - ***F****ield* ***P****rogrammable* ***G****ate* ***A****rray*

IEEE - ***I****nstitute of* ***E****letrical and* ***E****lectronics* ***E****ngineers*

LCD - ***L****iquid* ***C****rystal* ***D****isplay*

MARS - ***M****IPS* ***A****ssembler and* ***R****untime* ***S****imulator*

MDR - ***M****emory* ***D****ata* ***R****egisters*

MIPS - ***M****icroprocessor without* ***I****nterlocked*

P*ipeline* ***S****tages*

MUX - **M**ultiplexador

RAM - ***R****andom* ***A****ccess* ***M****emory*

RISC - ***R****educed* ***I****nstruction* ***S****et* ***C****omputing*

ULA - **U**nidade **L**ógica e **A**ritmética

VHDL - ***VHSIC*** ***H****ardware* ***D****escription* ***L****anguage*

VHSIC - ***V****ery* ***H****igh* ***S****peed* ***I****ntegrated* ***C****ircuit*

Capítulo 1

Introdução

O MIPS foi um dos primeiros processadores do modelo RISC(*Reduced Instrcutions Set Computer*) caracterizado por apresentar um conjunto simples e reduzido de instruções. Criado por John Hennessy em 1981 com intuito de aumentar o desempenho dos processadores da época, tinha como princípio utilizar *pipelines* profundos para instruções serem executadas em diferentes estágios permitindo que mais de uma instrução fosse processada ao mesmo tempo. Atualmente os processadores MIPS são usados na indústria automotiva, de eletrônicos e na informática possuindo uma variedade de placas de desenvolvimento atendendo entusiastas, estudantes e desenvolvedores de produtos.

Além de sua importância comercial o processador MIPS traz sua característica de simplicidade propiciando sua utilização para fins didáticos no aprendizado de arquitetura de computadores. Isto motivou a elaboração deste trabalho que utiliza os conceitos básicos do processador MIPS para elaborar um modelo utilizando a linguagem de descrição de *hardware* VHDL(VHSIC[1] *Hardware Description Language*) com avaliação de corretude utilizando FPGA(*Field Programmable Gate Array*).

A linguagem VHDL foi inicialmente desenvolvida pelo Departamento de Defesa dos Estados Unidos para fins de documentação de Circuitos Integrados de Aplicação Específica(ASIC) que faziam parte de equipamentos utilizados por suas forças armadas. Em 1987 foi padronizada pelo IEEE(*Institute of Eletrical and Eletronic*

[1]VHSIC - *Very High Speed Integrated Circuits* em português Circuito Integrado de Velocidade Muito Alta.

Engineers) e atualmente é utilizada em descrição, documentação, síntese, simulação, teste, verificação formal e compilação de software.

FPGA são chips reprogramáveis que utilizam blocos lógicos que podem comunicar entre si. É utilizado para implementação de funcionalidades personalizadas de *hardware* sem ser necessário construir um protótipo com circuitos integrados. A funcionalidade dos componentes são implementadas em *software* e a conexão entre eles podem ser modificas e reprogramadas sendo necessário apenas recompilar as modificações. Seus benefícios vão de alto desempenho ao baixo custo e sua natureza paralela permite executar diversas tarefas em um ciclo de relógio além de permitir representar diversas implementações de circuitos sem ser necessário adquirir um novo *hardware*.

1.1 Descrição do problema

A implementação de circuitos utilizando uma linguagem de descrição de *hardware* exige validações para que seja certificado o correto funcionamento da arquitetura proposta.

O processador MIPS que será implementado possui uma alta complexidade, embora seja bastante utilizado para fins didáticos, por ser também um processador comercial. Além disso, por ser tratado didaticamente faz-se interessante detalhar a diferença arquitetural de suas abordagens multiciclo e *pipeline*. Para isso é preciso implementar os componentes que fazem parte das duas arquiteturas e os que as diferenciam, além de verificar o comportamento correto de cada um. Também é necessário interligar estes componentes e sincronizá-los para que seja possível analisar o funcionamento do processador. Para certificar todos estes passos é necessário por em prática a execução de algoritmos e avaliar se o funcionamento da implementação tem o desempenho esperado pelas características do processador. Por fim uma análise precisa ser realizada para justificar as diferenças em desempenho das duas abordagens.

1.2 Objetivos

O objetivo deste trabalho é fornecer a implementação do processador MIPS em suas abordagens multiciclo e *pipeline* descrevendo os componentes na linguagem VHDL e validar seu funcionamento através do processamento de *benchmarks* analisando seu desempenho.

Os objetivos específicos estão listados a seguir:

- Detalhar a arquitetura do processador MIPS
- Descrever os componentes que compõem o processador MIPS multiciclo e *pipeline* utilizando a linguagem VHDL
- Realizar testes unitários sobre os componentes para vertificar seu funcionamento
- Simular os *benchmarks* com auxílio do *software* Qsim[2]
- Embarcar o código descritivo na FPGA e executar os *benchmarks*
- Relatar e analisar os resultados obtidos

1.3 Metodologia

Será realizado um estudo para definir as características do processador MIPS, dos componentes que serão implementados, das abordagens multiciclo e *pipeline* e da utilização de FPGA's para execução de circuitos descritos em VHDL.

Os componentes serão descritos utilizando o *software* Quartus II[3], desenvolvido pela empresa Altera, em linguagem VHDL. Em seguida serão realizados testes para validar o correto funcionamento dos componentes utilizando o *software* Qsim[2] também desenvolvido pela Altera. Ainda no Quartus II será construído o controlador principal das abordagens multiciclo e *pipeline* conectando os componentes de acordo com a arquitetura de cada abordagem. O passo seguinte é a execução do funcionamento do processador, convertendo os *benchmarks* da linguagem C para a

linguagem *assembly* da arquitetura MIPS. Com os algoritmos serão feitas simulações de execução utilizando o software Qsim e em seguida os algoritmos serão embarcados com o Quartus II na placa de FPGA do modelo Stratix EP1S10F780C6[4] desenvolvida pela Altera e será realizado o processamento dos algoritmos.

1.4 Organização da Monografia

A monografia está organizada da seguinte maneira:

- O Capítulo 2 tem por objetivo expor uma breve explicação da teoria envolvida na implementação do processador. Serão mencionados os conceitos e a estrutura da linguagem VHDL utilizados, o comportamento dos circuitos combinacionais que fazem parte da arquitetura do processador e conceitos sobre testes que serão utilizados para validar o funcionamento do processador.

- O Capítulo 3 detalha os registradores da arquitetura MIPS, as instruções que serão implementadas e a construção dos componentes. Além disso, é detalhado o funcionamento da abordagem multiciclo mostrando o controlador principal do processador, as etapas de execução das instruções e a arquitetura construída. A abordagem *pipeline* é explicada em modelos abordando o suporte a instruções simples até o tratamento de perigos, e com detalhes do funcionamento do controlador principal do *pipeline*.

- O Capítulo 4 valida a implementação de cada componente definindo casos de teste e simulando sua execução. É também apresentado os resultados referente a simulação dos *benchmarks* no processador com o *software* Qsim e na placa de FPGA. Por fim é realizada uma análise quantitativa em cima do desempenho apresentado pelas abordagens do processador MIPS.

- O Capítulo 5 expõe as conclusões sobre o alcance dos objetivos propostos e as considerações finais.

Capítulo 2

Fundamentação Teórica

Neste capítulo são apresentados os assuntos em que o desenvolvimento do trabalho foi baseado. Será apresentado uma introdução sobre a linguagem VHDL e algumas características que serão utilizadas para implementação dos componentes do processador, a teoria sobre circuitos combinacionais e algumas definições sobre teste de *software*.

2.1 Linguagem VHDL

A linguagem VHDL é destinada a um grande número de necessidades do processo de desenvolvimento de *hardware*. Primeiro, permite a descrição da estrutura de um sistema, isto é, sua decomposição em subsistemas e como são realizadas as interconexões entre eles. Segundo, permite a especificação da função do sistema utilizando construções familiares de linguagens de programação. Terceiro, permite que o sistema seja simulado antes de sua fabricação(antes que o dispositivo físico seja construído), possibilitando a identificação de problemas. Quarto, permite detalhar a estrutura de um projeto a ser sintetizado a partir de uma especificação, possibilitando ao projetista se concentrar mais nas estratégias de decisão de projeto e reduzindo o tempo de desenvolvimento[5].

O modelo geral no qual VHDL é baseado, é composto por três modelos interrelacionados: comportamento, tempo e estrutura. O modelo de comportamento

permite especificar a função de um objeto sem relacionar sua estrutura interna. O modelo estrutural permite descrever a função de um objeto de uma forma mais simples, interconectando objetos. O modelo de tempo, talvez o mais importante aspecto de VHDL, permite o desenvolvedor embutir informações de tempo no modelo[6].

Nas descrições estruturais são feitas instanciações e declarações de componentes, nesta é definida as interfaces dos subcomponentes utilizados na entidade. As instanciações dos componentes permitem o estabelecimento de interconexões com demais subcomponentes instanciados utilizando sinais globais[7].No quadro abaixo um exemplo de descrição estrutural de um *flip-flop* tipo D retirado de [8].

```
1  entity d_ff is
2    port(d,clk : in bit; q : out bit);
3  end d_ff;
4
5  architecture basic of d_ff is
6  begin
7    ff_behavior : process is
8    begin
9      wait until clk;
10     q <= d after 2 ns;
11   end process ff_behavior;
12 end architecture basic;
```

Em VHDL a descrição comportamental, ou seja, a implementação interna de uma entidade é chamada de *architecture body*. Há diferentes maneiras de implementar o corpo de uma entidade que realizam as mesmas funções. Nós podemos escrever um *architecture body* incluindo apenas instruções *process* que são uma coleção de ações a serem executadas em sequência. Estas ações são chamadas de instruções sequenciais e tem os mesmos tipos de instruções que encontramos em linguagens de programação convencionais. Os tipos de ações pode realizar expressões de avaliação, atribuir valores para variáveis, execuções condicionais, execuções repetidas e chamadas de funções. Além disso, há uma instrução sequencial que é única nas linguagens de descrição de *hardware*, a instrução *signal*. Ela é similar a uma variável de atribuição, exceto que armazena o valor de um sinal que será atualizado em algum momento no futuro[8]. O quadro abaixo representa uma implementação do

corpo de uma arquitetura comportamental adaptado de [8].

```
architecture behav of reg2 is
begin
  storage : process is
    variable stored_d0, stored_d1 : bit;
  begin
    wait until clk;
    if en then
      stored_d0 := d0;
      stored_d1 := d1;
    end if;
    q0 <= stored_d0 after 5 ns;
    q1 <= stored_d1 after 5 ns;
  end process storage;
end architecture behav;
```

Uma vez que o modelo estrutural e o comportamental tenham sido especificados, é possível simular o módulo por executar a descrição comportamental. Isto é feito simulando a passagem do tempo em passos discretos. Em algum tempo da simulação, um módulo de entrada pode ser estimulado para alterar o valor de uma porta de entrada. O módulo reage executando o código da descrição comportamental e seleciona novos valores para os sinais conectados à porta de saída. Isto é chamado de transição do sinal. Se o novo valor é diferente do antigo um evento ocorreu, e outros módulos com porta de entrada conectados ao sinal poderão ser ativados[5].

Modelos não precisam ser puramente estruturais ou comportamentais. Frequentemente é utilizado para especificar modelos com algumas partes compostas de instâncias de interconexões, e outras partes descritas usando processos. É utilizado sinais para comunicar instâncias e processos. Pode ser escrito um modelo híbrido incluindo ambos componentes de instância e processos no corpo de uma *architecture*. Estas instruções são coletivamente chamadas de instruções concorrentes, desde que os processos correspondentes executem de maneira concorrente quando o modelo é simulado[8]. No quadro abaixo uma descrição concorrente adaptada de [5].

```
architecture structure of count2 is
  component t_flipflop
```

```
3       port (ck : in bit; q : out bit);
4     end component;
5     component inverter
6       port (a : in bit; y : out bit);
7     end component;
8     signal ff0, ff1, inv_ff0 : bit;
9   begin
10    bit_0 : t_flipflop port map (ck => clock, q => ff0);
11    inv : inverter port map (a => ff0, y => inv_ff0);
12    bit_1 : t_flipflop port map (ck => inv_ff0, q => ff1);
13    q0 <= ff0;
14    q1 <= ff1;
15  end structure;
```

VHDL tem uma grande dualidade entre instruções concorrentes e sequenciais. Se uma ação é implementada usando sinais concorrentes há um equivalente para fazer a mesma coisa usando instruções sequenciais. Também é permitido chamadas de funções sequenciais dentro de programas concorrentes. Um exemplo de dualidade e mostrado no quadro abaixo[6].

```
1  Concorrente
2
3  output <= true when input='1' else false;
4
5  Sequencial
6
7  process(input)
8  begin
9    if input='1' then
10     output<=true;
11   else
12     output<=false;
13   end if;
14 end process;
```

Os principais conceitos da estrutura de VHDL foram apresentados e servirão como base para a descrição dos componentes e a interconexão entre eles para constituir o processador. Detalhes sobre comandos e tipos de dados da linguagem

podem ser encontrados em [5] e [8], as mesmas referências em que a revisão sobre a estrutura para descrição de *hardware* com VHDL foi baseada.

2.2 Circuitos Digitais

Neste capítulo será apresentado alguns conceitos de circuitos digitais dividido em circuitos combinacionais e sequenciais necessários para compreensão do projeto.

2.2.1 Circuitos Combinacionais

Um circuito digital é dito combinacional quando em um instante de tempo a saída depende exclusivamente das combinações das variáveis de entrada, não armazenando valores em memória, ou seja, seu estado atual das entradas e saídas independe dos estados anteriores. Seu fluxograma é composto de situação, tabela da verdade, expressão lógica e circuito. O circuito combinacional executa uma expressão lógica através da interligação de suas portas internas[9].

Um multiplexador é um circuito combinacional com n entradas e 1 saída, controlado por sinais de controle. De acordo com os sinais de controle é selecionada uma saída. Um demultiplexador é um circuito combinacional com 1 entrada e n saídas, com sinais de controle que selecionam para qual saída o dado de entrada será copiado enquanto as demais assumem um valor lógico[10].

O decodificador é um circuito digital que detecta a presença de uma combinação específica de bits em suas entradas indicando a presença desse código através de um nível de saída especificado. Normalmente tem-se n linhas para manipular n bits e de uma a 2^n linhas para indicar uma ou mais combinações de n bits. O codificador é um circuito lógico que realiza a função inversa do decodificador, aceitando um nível ativo em uma das entradas representando um dígito e convertendo em uma saída codificada.[11].

2.2.2 Circuitos Sequenciais

Circuitos sequenciais possuem suas saídas dependentes das variáveis de entrada e/ou de seus estados anteriores que permanecem armazenados, sendo, geralmente impulsionados por um sinal denominado *clock*. O *flip-flop* é o mais básico circuito sequencial. Ele possui basicamente dois estados de saída e para assumir um deles é necessário que haja uma combinação das variáveis e do pulso de controle. Após este pulso, o *flip-flop* permanecerá neste estado até a chegada de um novo pulso de *clock* e, então, de acordo com as variáveis de entrada, mudará ou não de estado[12].

Registradores são *flip-flops* arranjados de uma tal maneira que uma cadeia de bits pode ser armazenada por diversos registradores, onde cada registrador armazenará um bit. A cada ciclo de relógio os bits que se encontram na entrada dos registradores serão propagados alternando para uma nova cadeira de bits.

Contadores são *flip-flops* organizados de uma maneira síncrona ou assíncrona que modificam seu estado e o de outro *flip-flop* comportando-se como um contador. Contadores também podem ser usados como geradores de máquinas de estados combinados com lógica combinacional para voltar ao estado inicial.

Registradores de deslocamento são um tipo de circuito lógico parecido com os contadores digitais. Os registradores são usados principalmente no armazenamento de dados digitais e não possuem características de sequência de estados como contadores.

As memórias são em tese pequenos registradores agrupados. Cada elemento de armazenamento da memória pode reter um nível 1 ou 0 e é denominado de célula. As memórias são formadas por arranjos de células que podem ser acessados de acordo com sua linha e coluna. As células são arranjadas por linhas para formar endereços de um tamanho fixo de bits[11].

Nesta seção foram descritos conceitos sobre circuitos digitais que servirão como base para implementação dos componentes com a linguagem VHDL.

2.3 Conceitos de Testes

Neste trabalho os conceitos de teste de *software* serão aplicados à validação dos componentes do processador.

Teste de software é um processo pelo qual os sistemas são executados de maneira controlada, sendo analisadas as conformidades e as funcionalidades de acordo com as especificações do projeto de desenvolvimento[13].

Segundo Neto[14] "teste de *software* é o processo de execução de um produto para determinar se ele atingiu suas especificações e funcionou corretamente no ambiente para o qual foi projetado" e simplifica dizendo que "[...]testar um software significa verificar através de uma execução controlada se o seu comportamento corre de acordo com o especificado".

Um programa pode ser observado como sendo uma função que descreve uma relação entre um elemento de entrada e um elemento de saída e o processo de teste é utilizado para assegurar que o programa realize fielmente esta função consistindo em obter um valor válido do seu domínio funcional, determinar o comportamento esperado do programa para o valor válido escolhido, executar o programa, observar seu comportamento e, finalmente, comparar este comportamento com o esperado[15].

Para verificar o comportamento do *software* existem as técnicas de teste que são classificadas de acordo com a origem das informações utilizadas para estabelecer os requisitos de teste. Elas contemplam diferentes perspectivas do software e impõe-se a necessidade de se estabelecer uma estratégia de teste que contemple as vantagens e os aspectos complementares dessas técnicas. As técnicas existentes são: técnica funcional (ou teste caixa-preta) e estrutural (ou teste caixa-branca)[16].

Teste estrutural é uma técnica que avalia o comportamento interno do componente de *software* e o teste funcional é uma técnica que o componente de *software* é testado como se fosse uma caixa-preta, ou seja, não se considera a estrutura e comportamento interno do mesmo. Este envolve dois testes principais: identificar as funções que o software deve realizar e criar casos de teste capazes de checar se essas funções estão sendo realizadas pelo software[17].

O objetivo dos casos de teste é descobrir erros, utilizando-se de um critério com um mínimo esforço e tempo. As descrições devem identificar: estados do sistema antes da execução do teste, funções a serem testadas, valores de parâmetros para o teste e resultados esperados do teste. Um aspecto funcional de um caso de teste refere-se à definição dos resultados esperados, estes também devem ser planejados para auxiliar na identificação de entradas inválidas e inesperadas[7].

O teste de *software* é composto por uma gama de técnicas e estágios, e esta abordagem teórica especifica apenas o que será utilizado por este trabalho.

Capítulo 3

Desenvolvimento de um Modelo para Arquitetura MIPS

O processador MIPS é construído sobre a arquitetura *Harvard* e sua principal característica é ter os sinais e armazenamento da memória de programa e dados separadas. Assim, é possível acessar a memória de programa e memória de dados simultaneamente. Normalmente a memória de programa é de apenas leitura e a memória de dados de leitura e escrita, tornando impossível que a memória de programa seja alterada pelo próprio programa. Outra característica deste processador são seus 32 registradores com tamanho de 32 bits, detalhados na Seção 3.1. No MIPS este conjunto de 32 bits de dados é chamado de *word* para facilitar sua identificação nas instruções.

Nesta seção será apresentado os registradores e sua descrição, a descrição das instruções implementadas, a descrição dos componentes do processador e a implementação nas abordagens multiciclo e *pipeline*.

3.1 Registradores no MIPS

O processador MIPS possui em sua arquitetura um conjunto de 32 registradores, e na Tabela 3.1 temos a identificação e descrição de cada registrador.

Na próxima seção é detalhada as instruções implementadas neste trabalho.

Tabela 3.1: Identificação e descrição dos registradores.Adaptado de Patterson[1].

Nome	**Número**	**Descrição**
$zero	0	Valor constante 0
$at	1	Utilizado pelo montador de programas
$v0-$v1	2-3	Valores para resultados e avaliação de expressões
$a0-$a3	4-7	Argumentos de procedimentos
$t0-$t7	8-15	Registradores temporários
$s0-$s7	16-23	Valores salvos
$t8-$t9	24-25	Valores temporários
$k0-$k1	26-27	Para utilização do sistema operacional
$gp	28	*Global pointer*
$sp	29	*Stack pointer*
$fp	30	*Frame pointer*
$ra	31	Endereço de retorno

3.2 Instruções MIPS

O processador MIPS possui cerca de 120 instruções em sua arquitetura. Este trabalho implementa 15 instruções baseado na proposta de Patterson[1] que justifica sua abordagem devido a popularidade das instruções MIPS para os *benchmarks* de inteiro e ponto flutuante SPEC2000. A Tabela 3.2 adaptada de Patterson[1] mostra o quantitativo em que estas instruções são utilizadas.

Tabela 3.2: Utilização das instruções nos *benchmarks* SPEC 2000.

Núcleo MIPS	**Nome**	**Inteiro**	**PF**
adição sem sinal	addu	7%	21%
adição imediata sem sinal	addiu	12%	2%
subtração sem sinal	subu	3%	2%
operação lógica 'e'	and	1%	0%
operação lógica 'ou'	or	7%	2%
deslocamento lógico à esquerda	sll	1%	1%
carregar palavra(*load word*)	lw	24%	15%
armazenar palavra(*store word*)	sw	9%	2%
branch se igual	beq	6%	2%
branch se não igual	bne	5%	1%
jump and link	jal	1%	1%
jump para registrador	jr	1%	1%
setar menor que	slt	2%	0%
setar menor que imediato	slti	1%	0%
jump	j	<1%	<1%

Neste trabalho optamos por implementar as instruções *add*, *addi* e *sub* que realizam adição e subtração de números com sinais e são uma generalização das instruções *addu*, *addiu* e *subu*, o que nos leva a considerar a mesma porcentagem

de utilização nos *benchmarks*. Assim alcançamos uma cobertura de 81% das instruções utilizadas pelos *benchmarks* SPEC 2000 para programas com operações sobre inteiros. Instruções para ponto flutuante não serão implementadas pois exigiriam um maior grau de tratamento de erros como verificação de estouro aritmético e a implementação de exceções.

As instruções que serão suportadas possuem formato fixo com tamanho de 32 bits e são divididas em três tipos: R, I e J.

3.2.1 Instruções do Tipo R

O formato R possui seis campos que estão descritos na Tabela 3.3.

Tabela 3.3: Campos do formato R do processador MIPS.

Sigla	**Descrição**	**Tamanho**
op	operação da instrução	6 bits
rs	registrador do primeiro operando de origem	5 bits
rt	registrador do segundo operando de origem	5 bits
rd	registrador do operando de destino	5 bits
shamt	quantidade de deslocamento	5 bits
funct	função que representa uma variante da opção	6 bits

As operações do tipo R são para instruções aritméticas que utilizam três registradores. O campo operação tem tamanho fixo. O campo rs é o valor do registrador à esquerda do operador, o campo rt é o valor do registrador à direita do operador e o campo rd é o valor do registrador onde será armazenado o resultado da operação(*e.g.*, rd = rs+rt). O campo *shamt* é utilizado para operações de deslocamento de bits, indicando a quantidade de bits que serão deslocados. Por fim, o campo *funct* é utilizado para selecionar uma variação das operações do tipo R.

As instruções do tipo R que serão implementadas estão descritas a seguir.

Add $t0,$t0,$t1: esta instrução soma o conteúdo do registrador $t0 com o registrador $t1 armazenando o resultado em $t0.

Sub $t0,$t1,$t2: esta instrução subtrai o conteúdo do registrador $t1 do registrador $t2 armazenando o resultado em $t0.

And $t0,$t1,$t2: esta instrução realiza a operação *and* bit a bit do conteúdo

do registrador $t1 com o registrador $t2 armazenando o resultado em $t0.

Or $t0,$t1,$t2: esta instrução realiza a operação *or* bit a bit do conteúdo do registrador $t1 com o registrador $t2 armazenando o resultado em $t0.

Jr $ra: é uma instrução chamada *jump register* usada no retorno à funções chamadoras, utilizando o endereço armazenado no registrador $ra.

Slt $t0,$t1,$t2: é uma instrução de comparação onde se $t1 for menor que $t2, $t0 é igual a 1, caso contrário é igual a 0(zero).

Sll $t0,$t1,N: é uma instrução de deslocamento de bits, onde é deslocado N bits de $t1 armazenando o resultado em $t0.

3.2.2 Instruções do Tipo I

O formato I possui 4 campos e é utilizado pela instruções imediatas e de transferência de dados. O detalhe dos campos é apresentado na Tabela 3.4.

Tabela 3.4: Campos do formato I do processador MIPS.

Sigla	**Descrição**	**Tamanho**
op	operação da instrução	6 bits
rs	registrador do primeiro operando de origem	5 bits
rt	registrador do operando de destino	5 bits
	constante ou endereço	16 bits

Os campos op e rs têm a mesma função da instrução do tipo R. O campo rt é o valor do registrador utilizado como destino do resultado das operações. O último campo funciona como constante para operações imediatas(*e.g.*, rt = rs+100) e comparações(*e.g.*, rt = rs<100) ou como endereço para operações de leitura e escrita na memória.

As instruções do tipo I que serão implementadas estão descritas a seguir.

Addi $t0,$t0,N: esta instrução soma o conteúdo do registrador $t0 com o valor da constante N.

Beq $t0,$t1,N: compara se o valor em $t0 é igual ao valor em $t1. Se verdadeiro o contador de programas é desviado em relação ao próximo endereço do contador de programas mais N posições(PC+1+N), caso contrário realiza a pró-

xima instrução.

Bne $t0,$t1,N: compara se o valor em $t0 é diferente do valor em $t1. Se verdadeiro o contador de programas é desviado em relação ao próximo endereço do contador de programas mais N posições(PC+1+N), caso contrário realiza a próxima instrução.

Lw $t0,N($t1): transfere dados da memória para o registrador utilizando o endereço base da memória $t1 somado com o *offset* N e armazenando os dados no registrador $t0.

Sw $t0,N($t1): transfere dados do registrador para memória utilizando os dados do registrador $t0 e armazenando no endereço base da memória $t1 somado com o *offset* N.

Slti $t0,$t1,N: compara se $t1 é menor que o valor de N, se verdadeiro $t0 é igual 1 se não $t0 igual a 0.

3.2.3 Instruções do Tipo J

O formato J possui 2 campos e é utilizado por instruções de desvio incondicional. O detalhe dos campos é apresentado na Tabela 3.5.

Tabela 3.5: Campos do formato J do processador MIPS.

Sigla	**Descrição**	**Tamanho**
op	operação da instrução	6 bits
	endereço de destino	26 bits

O campo op é utilizado para o código da operação. O campo endereço de destino indica a quantas palavras está a próxima instrução a ser executada.

As instruções que serão implementadas do tipo J estão descritas a seguir.

J N: onde N é a posição da *word* de instrução relativa à primeira instrução do programa.

Jal N: onde o endereço do contador de programa mais um(PC+1) é armazenado no registrador $ra e o contador de programa aponta para o endereço da nova instrução.

A seção seguinte apresenta a descrição dos componentes utilizados na implementação do processador.

3.3 Componentes do processador

Esta seção descreve o funcionamento dos componentes que serão implementados com a linguagem de descrição de *hardware* VHDL e exibe o bloco funcional com detalhe das portas de entrada e saída e *flags* de controle.

Os componentes são utilizados na implementação das duas abordagens do processador, onde as modificações que caracterizam cada abordagem são identificadas pela conexão entre os componentes e pelo controle das *flags* e serão explicadas nas Seções 3.4 e 3.5.

3.3.1 Contador de Programa

O contador de programa(CP) é um registrador que aponta o endereço da instrução a ser executada.

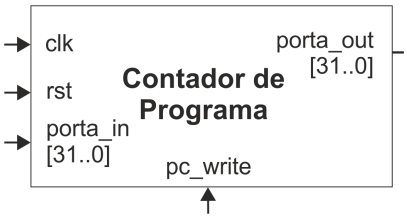

Figura 3.1: Bloco funcional do contador de programa.

Na Figura 3.1 temos o bloco funcional do contador de programa. Na porta *clk* é conectado o pulso de relógio, o CP opera na borda de subida do pulso e é reiniciado quando a porta *rst* é colocada em nível baixo. Na *porta_ in* é conectado uma entrada(e.g., multiplexador, *saída da ALU*) que dá origem aos valores do CP e a *porta_ out* propaga o valor da entrada da *porta_ in* quando a *flag* de controle *pc_ write* é ativada(nível alto).

3.3.2 Memória

A memória armazena as instruções do programa, valores de constantes e variáveis. Sua utilização ocorre de maneira diferenciada nas duas abordagens que serão explicadas nas Seções 3.4 e 3.5.

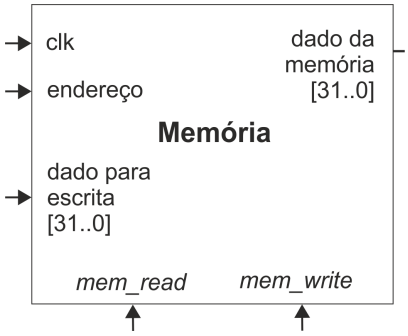

Figura 3.2: Bloco funcional da memória.

Na Figura 3.2 temos o bloco funcional da memória. Na porta *clk* é conectado o pulso de relógio. A porta *endereço* é conectada ao CP que envia o endereço da posição de memória onde ocorrerá uma leitura ou escrita. A porta *dado para escrita* recebe os dados para escrita na memória e funciona em conjunto com a *flag* de controle *mem_read* ativada em nível alto, escrevendo o dado na subida do pulso. A porta *dado da memória* coloca o dado lido de uma posição de memória quando a *flag* de controle *mem_read* é ativada em nível alto.

Esta memória foi descrita para funcionar com 256 endereços de 8 bits cada, sendo possível armazenar 64 palavras de 32 bits, o equivalente a 256 bytes de dados. Esta limitação ocorre por conveniência do projeto.

3.3.3 Registrador de Instrução

O registrador de instrução armazena os bits transferidos da memória de dados e separa em suas saídas os bits dos campos de operação (op), registradores (rs, rt) e os 16 bits menos significativos da instrução.

Na Figura 3.3 temos o bloco funcional do registrador de instrução. Na porta *clk* é conectado o pulso de relógio. A porta *dado_in* está conectada à saída da memória que transfere uma palavra de dados(*word*). O registrador de instrução

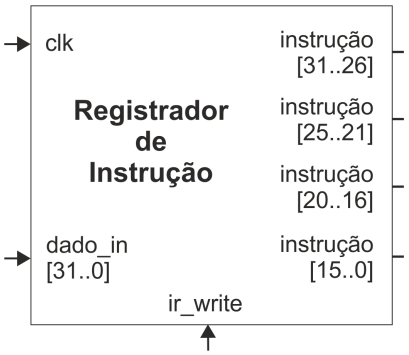

Figura 3.3: Bloco funcional do registrador de instrução.

armazena a palavra na subida do pulso quando a *flag* de controle *ir_write* está ativada(nível alto). As portas de saída fornecem os campos separados e têm seus valores modificados quando um novo dado é armazenado no registrador.

3.3.4 Registrador de Dados da Memória (MDR)

O registrador de dados da memória armazena uma palavra para ser escrita em um registrador do banco de registradores. É utilizado para permitir que o registrador de instruções possa determinar o endereço do registrador que o dado será armazenado, e fornecer o dado a ser escrito.

Figura 3.4: Bloco funcional de um registrador.

Na Figura 3.4 temos o bloco funcional de um registrador. Na porta *clk* é conectado o pulso de relógio. A porta *porta_in* recebe os dados que são armazenados na subida do pulso. A porta *porta_out* fornece o dado armazenado no registrador.

3.3.5 Banco de Registradores

O banco possui 32 registradores de 32 bits, de acordo com a arquitetura do processador MIPS, e é utilizado para armazenar constantes, variáveis e dados que

são utilizados pelo processador durante a execução de um programa. A função de cada registrador está detalhada na Seção 3.1.

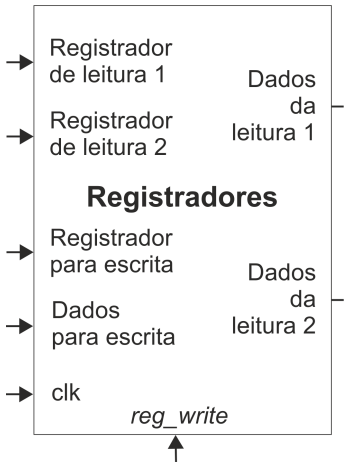

Figura 3.5: Bloco funcional do banco de registradores.

Na Figura 3.5 temos o bloco funcional do banco de registradores. Na porta *clk* é conectado o pulso de relógio. As portas *registrador de leitura 1* e *registrador de leitura 2* estão conectadas ao registrador de instrução que transfere cinco bits referentes ao endereço dos registradores rs e rt respectivamente, colocando os dados contidos nestes registradores nas portas *dados da leitura 1* e *dados da leitura 2*. A porta *registrador para escrita* recebe o endereço de um registrador e escreve os dados da porta *dados para escrita* na subida do pulso quando a *flag* de controle *reg_ write* está ativada em nível alto.

3.3.6 Extensor de Sinal

O componente para extensão de sinal modifica um dado de 16 bits para 32 bits propagando o bit de sinal.

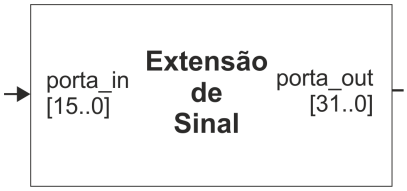

Figura 3.6: Bloco funcional do extensor de sinal.

Na Figura 3.6 temos o bloco funcional do extensor de sinal. A extensão é independente de pulso de relógio e uma alteração do dado na porta *porta_ in* é

convertida para 32 bits na porta *porta_ out*.

3.3.7 Registrador de Deslocamento

O registrador de deslocamento é utilizado para multiplicar um número por quatro ao deslocar dois bits à esquerda. As instruções de desvio incondicional utilizam este componente para calcular o novo endereço do CP.

Figura 3.7: Bloco funcional do registrador de deslocamento.

Na Figura 3.7 temos o bloco funcional do registrador de deslocamento. O registrador de deslocamento é independente de pulso de relógio e uma alteração do dado na porta *porta_ in* é transferida para porta *porta_ out*.

3.3.8 Multiplexador

O mux(multiplexador) é utilizado na arquitetura do processador para reduzir o número de componentes utilizados. As informações que possuem o mesmo destino são selecionadas por *flags* de controle e transmitidas ao componente.

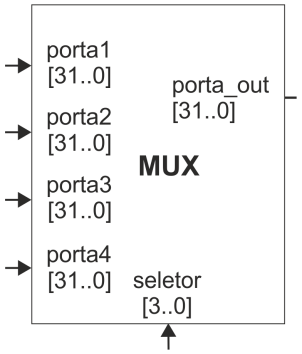

Figura 3.8: Bloco funcional do multiplexador.

Na Figura 3.8 temos o bloco funcional do mux que é independente de pulso de relógio. A porta *seletor* seleciona entre as portas *porta[i]* uma saída para porta *porta_ out* através das *flags* de controle do processador.

3.3.9 Unidade Lógica e Aritmética

A ULA(unidade lógica e aritmética) é utilizada por todas as classes de instruções, exceto *jump*, após a leitura dos registradores. As operações de cálculo de endereços, comparações lógicas e aritmética para desvios e operações entre registradores (adição, subtração, e, ou e menor que) são realizadas pela ULA.

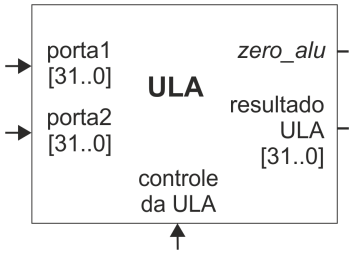

Figura 3.9: Bloco funcional da ULA.

Na Figura 3.9 temos o bloco funcional da ULA. A ULA é independente de pulso de relógio e sua saída na porta *resultado ULA* é modificada de acordo com a alteração de *porta1* e *porta2* e da porta *controle da ULA* que determina qual operação será realizada pela ULA. A *flag zero_ alu* é utilizada por outros componentes para comparações de valores entre registradores.

Tabela 3.6: Operações implementadas na ULA.

Instrução	**Controle da ULA**	**Operação**
and	0000	porta1 and porta2
or	0001	porta1 or porta2
add, addi, lw, sw, j, jr	0010	porta1 + porta2
sub, beq	0110	porta1 - porta2 , zero_alu=1 ou 0
slt, slti	0111	porta1 < porta2
bne	1000	porta1 - porta2 , zero_alu=0 ou 1
sll	1001	porta2 << 2
mul	1010	porta1[15..0] * porta2[15..0]

Na Tabela 3.6 temos as operações suportadas pela ULA implementada. Nas instruções *beq* e *bne* a *flag zero_ alu* é alterada conforme o resultado da subtração. A operação de multiplicação é feita sobre os 16 *bits* dos dados pois seu resultado é de 32 *bits*.

3.3.10 Controle da ULA

O controle da ULA é utilizado para determinar qual operação será realizada analisando os campos da instrução.

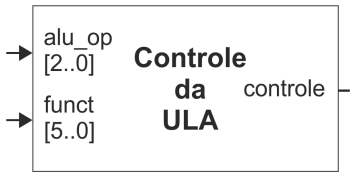

Figura 3.10: Bloco funcional do controle da ULA.

Na Figura 3.10 temos o bloco funcional do controle da ULA. O controle da ULA é independente de pulso de relógio. A porta *alu_ op* recebe bits do componente controlador principal(Seção 3.4.1). A porta *funct* tem os bits de função de uma instrução. A porta *controle* é conectada a ULA para selecionar a operação que será realizada.

Tabela 3.7: Síntese do controle da ULA.

instruçao	**alu_op**	**funct**	**controle**
lw, sw, addi	000	xxxxxx	0010
beq	001	xxxxxx	0110
add	010	100000	0010
sub	010	100010	0110
and	010	100100	0000
or	010	100101	0001
slt	010	101010	0111
jr	010	001000	0010
sll	010	000000	1001
slti	011	xxxxxx	0111
bne	100	xxxxxx	1000
mul	101	xxxxxx	1010

Na Tabela 3.7 é detalhado o funcionamento do controle da ULA com a saída de controle determinada pelos campos alu_op e *funct* da instrução.

3.4 Arquitetura multiciclo

A abordagem multiciclo é caracterizada por executar as instruções em cinco etapas utilizando cinco ciclos de relógio, permitindo cada componente operar mais de uma vez durante uma instrução contanto que em ciclo de relógio diferente. Isto

reduz a quantidade de *hardware* e permite o compartilhamento dos componentes dentro da execução da instrução. Nesta seção será apresentada a arquitetura do processador MIPS implementado na sua abordagem multiciclo, o componente de controle principal que é responsável pela sincronização do processador e o controle de *flags* com a descrição da função de cada *flag* de controle.

3.4.1 O Controle Principal

O controle principal tem a função de sincronizar o funcionamento dos componentes do processador através das *flags* de controle. A cada pulso de relógio um passo de uma instrução é executado e na abordagem multiciclo são consumidos de 3 a 5 pulsos de relógio por instrução.

A implementação pode ser feita com máquina de estados ou microprogramação. Neste trabalho é utilizada a abordagem com máquina de estados, onde cada estado determina um passo da execução das instruções. A implementação está separada entre dois componentes: o controle de estados e o controle das *flags*.

Na Figura 3.11 temos o bloco funcional do controle de estados.

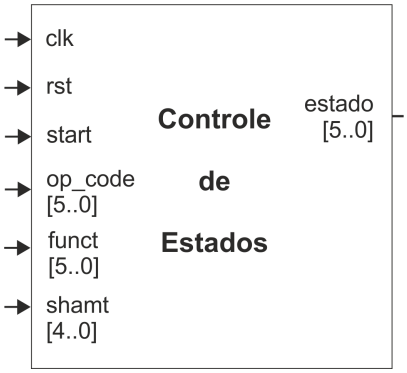

Figura 3.11: Bloco funcional do controle de estados.

O controle de estados é sensível à subida do pulso de relógio que está conectado à porta *clk*. As portas *rst* e *start* são conectadas a sinais externos para reiniciar e inciar respectivamente a execução das instruções de um programa armazenado na memória. A porta *op_ code* recebe o código de operação da instrução. A porta *funct* tem os bits de função de uma instrução que são utilizados para diferenciar os estados em instruções do tipo R. A porta *shamt* tem os bits do campo *shamt* para instrução

de deslocamento de bits. A porta *estado* transfere o estado atual para o controle de *flags* ativar os componentes.

Na Figura 3.12 temos o bloco funcional do controle de *flags*.

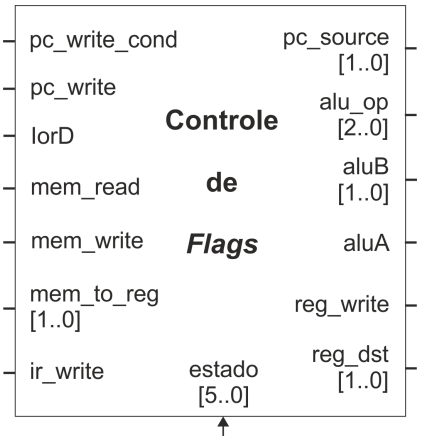

Figura 3.12: Bloco funcional do controle de flags.

O controle de flags é independente de pulso de relógio e suas alterações ocorrem com a mudança dos estados que determinam o comportamento dos componentes e seu modo de operação.

A função das portas do controle de *flags* estão descritas a seguir.

A flag *pc_write_cond* determina a escrita no CP quando uma instrução de desvio condicional é executada. Esta ação é concluída se a porta *zero_alu* do componente ULA está ativada.

A flag *pc_write* ativa a escrita de um endereço de instrução no CP. Sua origem pode ser de desvio condicional, incondicional ou do incremento do CP por 4 para buscar a próxima instrução.

A flag *IorD* define o endereço que será acessado na memória de dados tem origem no CP ou no componente *saída da ULA*. No caso do CP temos uma nova instrução e da *saída da ULA* temos uma busca ou escrita de dados na memória.

As flags *mem_write* e *mem_read* determinam qual operação é realizada pela memória de dados. Quando *mem_write* está ativada, a entrada de dados para escrita é armazenada no endereço de memória. Quando *mem_read* está ativada a saída da memória recebe os dados apontados pelo endereço.

A flag *mem_to_reg* determina se os dados a serem escritos nos registradores terão origem na MDR ou na *saída da ULA*.

A flag *ir_write* determina a escrita do dado da memória no registrador de instrução.

A flag *pc_source* controla um mux determinando qual dado será escrito no CP. Os dados tem origem no resultado da ULA, na *saída da ULA* e no registrador de deslocamento em caso de instrução *jump*.

A flag *aluOp* tem três bits que determina qual operação será realizada pela ULA. Esta definição utiliza também os bits do campo *funct* da instrução.

A flag *aluB* define qual dado será utilizado pelo registrador B. Temos a saída do registrador com a porta *dados de leitura 2*, o valor quatro para cálculo da próxima instrução, os 32 bits de dados extendidos ou o registrador de deslocamento.

A flag *aluA* define se o registrador A receberá o valor do CP ou a saída da porta *dados de leitura 1*.

A flag *reg_write* permite o banco de registradores escrever um dado em um registrador.

A flag *reg_dst* define qual a origem do endereço de registrador para escrita, podendo ser os registradores rt ou rd.

Os estados e as *flags* controlam o funcionamento do processador e toda dinâmica da execução das instruções estão nestes componentes.

3.4.2 Etapas de Execução Multiciclo

Na abordagem multiciclo as etapas de uma instrução ocorrem a cada ciclo de relógio. Esta seção detalha o valor das *flags* de controle na execução de uma instrução, que ocorre de três a cinco etapas dependendo do tipo da instrução.

Na primeira etapa a instrução é buscada ativando as *flags mem_read* e *ir_write* e *IorD*=0, armazenando a instrução apontada pelo CP no registrador de instruções. O endereço para próxima instrução é calculado com a *flag aluA*=0, *aluB*=01

e *aluOp*=000 e para o CP armazenar o endereço da próxima instrução a *flag pc_source*=00 e *pc_write*=1. A Tabela 3.8 exibe o resumo das *flags*.

Tabela 3.8: Resumo do controle de *flags* da etapa de busca de instrução.

mem_write	ir_write	IorD	aluA	aluB	aluOp	pc_source	pc_write
1	1	0	0	01	000	00	1

Na segunda etapa os valores dos registradores rs e rt são armazenados nos registradores A e B. O endereço de desvios é calculado com as *flags aluA*=0, *aluB*=11 e *aluOp*=000 armazenando o resultado na *saída da ULA*. A Tabela 3.9 exibe o resumo do controle de *flags*.

Tabela 3.9: Resumo do controle de *flags* da etapa de decodificação da instrução e busca dos registradores.

aluA	aluB	aluOp
0	11	000

A terceira etapa pode ter execução, cálculo do endereço de memória ou conclusão do desvio.

Para referência à memória, a ULA calcula o endereço com as *flags aluA*=1, *aluB*=10 e *aluOp*=000 somando o conteúdo registrador rs aos 32 bits do extensor de sinal. A Tabela 3.10 exibe o resumo do controle de *flags*.

Tabela 3.10: Resumo do controle de *flags* para referência à memória.

aluA	aluB	aluOp
1	10	000

Na execução de uma instrução lógica ou aritmética a ULA faz uma operação entre os registrados rs e rt colocando as *flags aluA*=1, *aluB*=00 e *aluOp*=010 e utiliza o campo *funct* definir a operação. A Tabela 3.11 exibe o resumo do controle de *flags*.

Tabela 3.11: Resumo do controle de *flags* para execução de operações lógicas e aritméticas.

aluA	aluB	aluOp
1	00	010

Em desvios a ULA compara os registradores e ativa a *flag "zero da ULA"* caso sejam iguais e as *flags aluA*=1, *aluB*=00 e *aluOp*=001. Se a saída for ativada

as *flags pc_ write_ cond* é ativada e *pc_ source*=01. A Tabela3.12 exibe o resumo do controle de *flags*.

Tabela 3.12: Resumo do controle de *flags* para desvios.

aluA	aluB	aluOp	pc_source	pc_write_cond	zero da ULA
1	00	001	01	1	1

Na quarta etapa as instruções para acesso à memória ou conclusão de instruções do tipo R são finalizadas. As instruções *load* e *store* acessam a memória, e uma instrução de operação lógica ou aritmética escreve seu resultado no registrador rd.

Na referência à memória a instrução *load* lê a memória de dados e escreve na MDR colocando a *flag mem_ read*=1. Na instrução *store* o dado é escrito na memória de dados colocando a *flag mem_ write*. A referência de endereço está calculada na *saída da ULA* e é utilizado colocando a *flag IorD*=1. A Tabela 3.13 exibe o resumo do controle de *flags*.

Tabela 3.13: Resumo do controle de *flags* para conclusão de uma instrução *store* e leitura de dados para uma instrução *load*.

mem_write ou mem_read	IorD
1	1

Para instrução lógica ou aritmética a *flag reg_ dst*=1 para selecionar o campo rd como registrador de destino. A *flag reg_ write*=1 e *mem_ to_ reg*=0 para os dados da *saída da ULA* como dados de escrita. A Tabela 3.14 exibe o resumo do controle de *flags*.

Tabela 3.14: Resumo do controle de *flags* para conclusão de instruções do tipo R.

reg_dst	reg_write	mem_to_reg
1	1	0

A quinta etapa é usada para conclusão da leitura de memória. A instrução *load* completa sua execução escrevendo os dados da MDR no registrador de destino colocando as *flags mem_ to_ reg*=1, *reg_ write*=1 e *reg_ dst*=0. A Tabela 3.15 exibe o resumo do controle de *flags*.

Com a implementação dos componentes do processador, o controle principal da abordagem multiciclo e com as etapas de execução descritas o funcionamento

Tabela 3.15: Resumo do controle de *flags* para conclusão da instrução *load*.

mem_to_reg	reg_write	reg_dst
1	1	0

do processador é finalizado com a conexão entre os componentes. A seção seguinte mostra a arquitetura do processador MIPS na abordagem multiciclo.

3.4.3 Modelo multiciclo

O modelo multiciclo é mostrado na Figura 3.13 para detalhar as conexões. Alguns componentes foram modificados para permitir que mais instruções(*jump register, mul, addi e slti* fossem implementadas devido não serem descritas ou suportadas pela arquitetura proposta por Patterson[1]. Estas instruções são importantes para executar os programas que definimos para validar a implementação do processador.

As modificações podem ser observadas no multiplexador *mem_to_reg* que possui uma entrada com o valor 31 para indicar o registrador $ra utilizado em instruções *jump register* e *jump and link*. O controle da ULA tem como entrada a *flag aluOp* para permitir mais operações de instruções com valores imediatos(*addi,slti* e aritméticas(*mul e sll*). Os demais componentes possuem o funcionamento apresentado no projeto de Patterson[1].

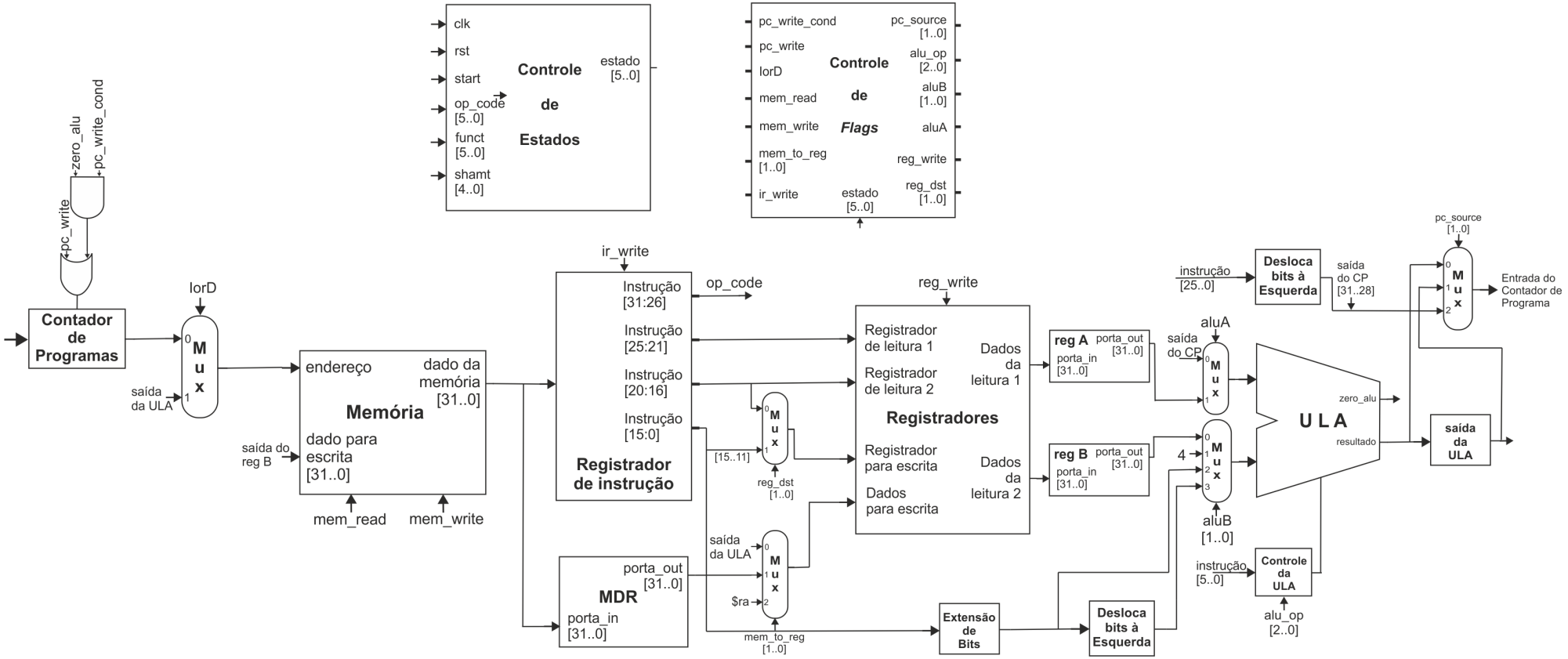

Figura 3.13: Arquitetura do processador MIPS multiciclo implementado.

Com este diagrama concluímos o desenvolvimento da abordagem multiciclo e na próxima seção será apresentada a implementação da abordagem *pipeline*.

3.5 Arquitetura *pipeline*

O MIPS *pipeline* é implementado utilizando a técnica de *pipelining* onde mais de uma instrução executa ao mesmo tempo. Isso é possível pois a construção do processador é definida por estágios que realizam operações independentes nos componentes, proporcionando um aumento no desempenho quando comparado à implementação do processador MIPS multiciclo.

Este aumento do desempenho ocorre pela quantidade de instruções executadas e não pela redução do tempo para executar uma instrução. No multiciclo as instruções são executadas entre três e cinco ciclos de relógio, e a próxima instrução só iniciará após o término da instrução anterior. No *pipeline* as instruções iniciam uma após a outra a cada ciclo de relógio reduzindo a quase um ciclo de relógio por instrução.

O *pipeline* neste projeto possui cinco estágios:

1 Buscar instrução da memória - IF(*instruction fet*)

2 Decodificar instrução e ler registradores - ID(*instruction decode*)

3 Calcular endereço para desvios ou executar uma operação - EX(*execute*)

4 Executar uma escrita ou leitura na memória de dados - MEM(*memory*)

5 Escrever em um registrador - WB(*write back*)

A comunicação entre estágios ocorre através de registradores e sua identificação determina entre quais estágios o registrador está. Assim, entre o primeiro e o segundo estágios está o registrador IF/ID, segundo e terceiro estágios o registrador ID/EX, terceiro e quarto estágios EX/MEM, quarto e quinto estágios o registrador MEM/WB. No quinto estágio não há dado para um estágio posterior realizando

apenas a escrita no banco de registradores quando necessário. Estes registradores fazem a transição dos dados a cada ciclo de relógio sincronizadamente.

A implementação do processador foi dividida em modelos simples, intermediário e de perigos. O modelo simples executa as instruções do tipo R(*add, sub, and, or, slt e sll*) e do tipo I(*addi, lw, sw, beq, bne e slti*). O modelo intermediário executa desvios incondicionais(*jump register* do tipo R, *j* e *jal* do tipo J). Por fim, o modelo de perigos(ou *hazards* na literatura) que trata conflitos que podem prejudicar e falhar a execução do processador.

3.5.1 Modelo simples

No primeiro estágio(IF) o multiplexador define qual a origem do endereço da instrução na memória. O endereço vindo do somador seleciona a próxima instrução, somando quatro ao endereço atual do contador de programa e o endereço vindo do somador no estágio 3 é resultado de uma operação de desvio condicional. A memória de instruções possui o programa armazenado e, diferente do processador multiciclo, é independente da memória de dados. Isto acontece porque no estágio do *pipeline* cada componente pode ser lido ou escrito apenas uma vez, evitando conflitos durante a execução do processador.

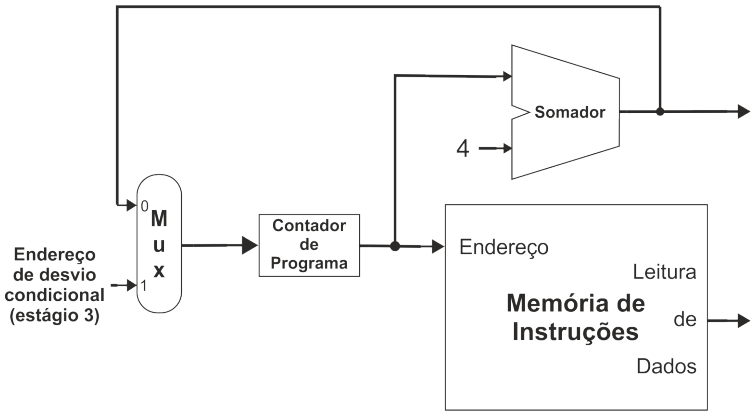

Figura 3.14: Estágio 1 da implementação do modelo simples.

No segundo estágio a instrução é decodificada em campos(opcode, rs, rt e os 16 bits). O banco de registradores tem em suas entradas o endereço dos registradores de leitura rs e rt. Na entrada para o registrador de escrita o multiplexador define a origem no campo rt ou rd da instrução. O dado escrito tem origem no estágio 5

definido pelo multiplexador que seleciona a origem de uma leitura da memória ou de um resultado da ULA. O componente de extensão de sinal tem como entrada os 16 bits menos significativos da instrução, sendo útil caso tenhamos uma instrução do tipo I. Esta operação é realizada neste estágio pois não há custo de utilização do *hardware*.

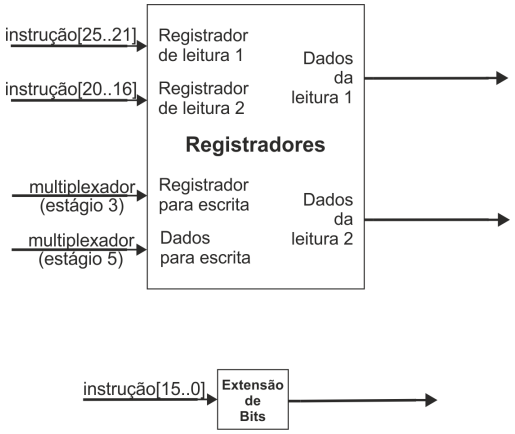

Figura 3.15: Estágio 2 da implementação do modelo simples.

No terceiro estágio o controle da ULA define qual operação será relizada analisando o campo opcode. As entradas da ULA são o registrador rs e o multiplexador que tem o campo rt e os 16 bits extendidos para instruções do tipo I. A ULA possui na saída a *flag zero* utilizada para identificar quando um desvio será realizado, direcionando o endereço calculado para o contador de programa, com origem na saída do somador deste mesmo estágio. O registrador de deslocamento multiplica o endereço por quatro para intruções de desvio. Como a memória implementada é de 8 bits por endereço e o desvio é referido a quantidade de palavras, é necessário a conversão para uma posição efetiva de memória. O somador realiza a operação entre o endereço da próxima instrução, que tem origem no contador de programa, e os 16 bits deslocados que apontam para o desvio.

No quarto estágio a memória de dados é acessada para operações de leitura ou escrita. A porta *and* verifica a *flag zero* da ULA e a *flag branch* para escrever o endereço de desvio no contador de programa.

No quinto estágio o multiplexador seleciona uma leitura realizada na memória ou um resultado de uma operação realizada na ULA para escrever no banco de registradores. Este estágio encerra um ciclo de execução de uma instrução no

pipeline.

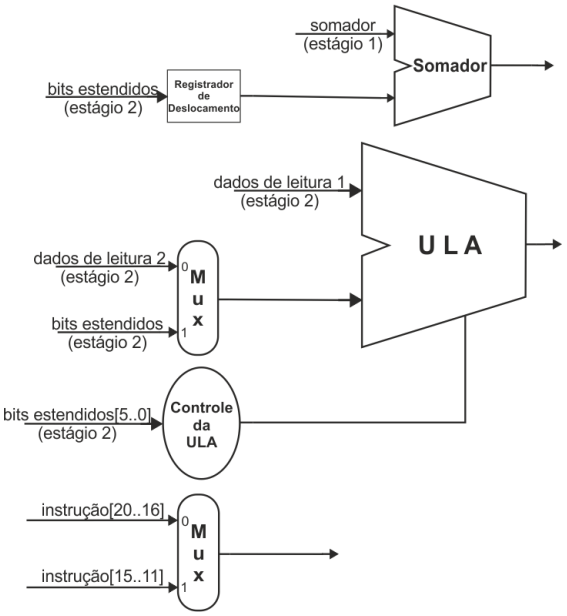

Figura 3.16: Estágio 3 da implementação do modelo simples.

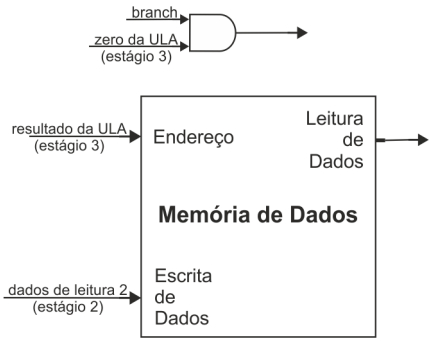

Figura 3.17: Estágio 4 da implementação do modelo simples.

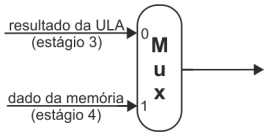

Figura 3.18: Estágio 5 da implementação do modelo simples.

3.5.2 Modelo intermediário

O modelo intermediário implementa as instruções de desvios incondicionais e é necessário alterar a estrutura do multiplexador conectado ao contador de programas para quatro entradas(somador, endereço para j e jal, e jr).

O endereço das instruções *j* e *jal* é calculado no estágio 3 com os quatro bits mais significativos da saída do somador(estágio 1) mais os 26 bits deslocados do campo de endereço em instruções do tipo J. O endereço é então constituído pelos bits [31..28](somador), [27..2](campo da constante deslocado) e [1..0](deslocamento). Este endereço é transferido ao multiplexador do contador de programa.

A instrução *jal* escreve o endereço da próxima instrução no registrador $ra e realiza o desvio. O multiplexador do estágio 5 é alterado para três entradas incluindo o cálculo de endereço do somador(estágio 1) propagado nos registradores do *pipeline*. O multiplexador para seleção do registrador de destino no estágio 3 também é modificado para três entradas incluindo o valor de endereço do registrador $ra(31).

A instrução de *jump register*(jr) altera o estágio 3 incluindo uma conexão da saída de leitura 1 para o multiplexador do contador de programa. Na execução da instrução *jr* o valor do registrador $ra é lido do banco de registradores e escrito no contador de programa para retornar à instrução seguinte ao desvio.

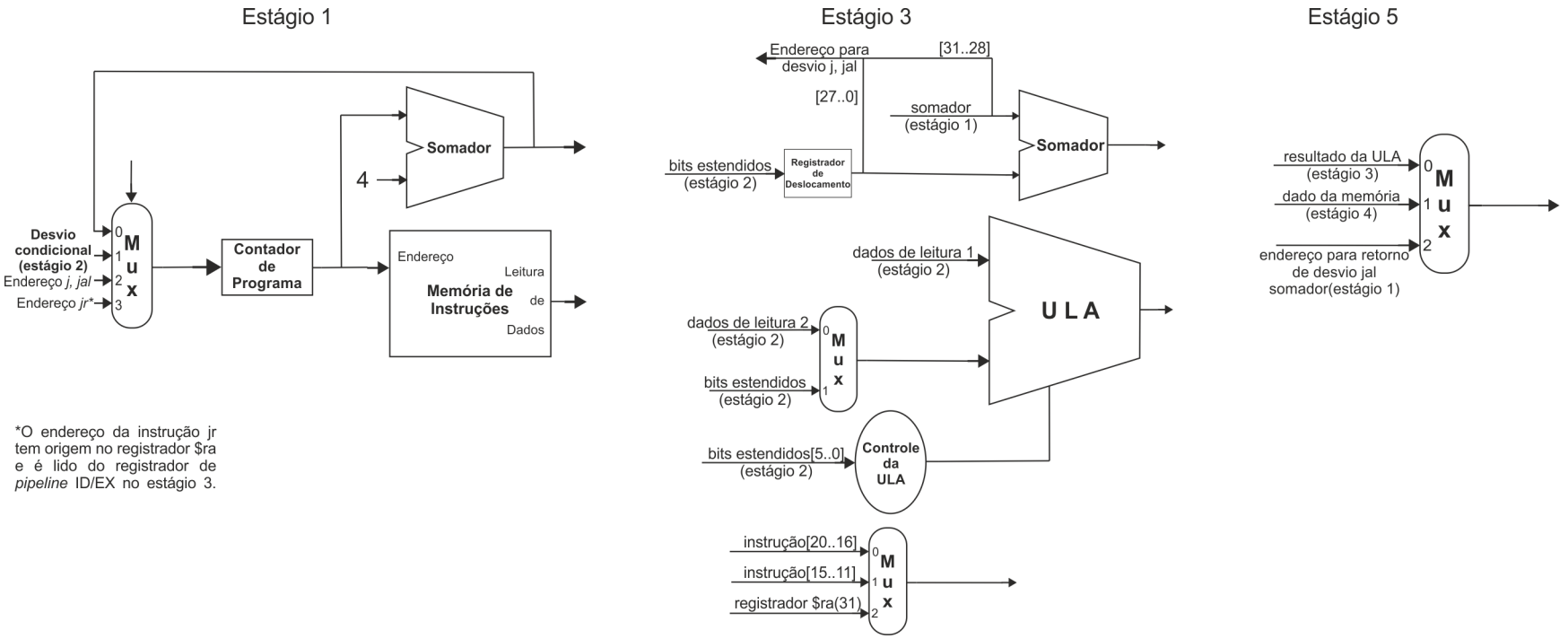

Figura 3.19: Alterações do modelo intermediário.

3.5.3 Implementando os perigos

Perigos são situações em que determinadas sequências de instruções não podem ser executadas corretamente devido a conflitos no *pipeline*(*e.g.*, uma instrução *load word* carrega o valor de uma variável em um registrador no estágio cinco e instruções que tentarem acessar este registrador antes disso utilizarão o valor antigo). Existem três tipos de perigos que serão apresentados com detalhes: estrutural, de dados e controle.

O perigo estrutural ocorre quando um mesmo componente é acessado por duas instruções diferentes ao mesmo tempo. Por exemplo, a memória sendo escrita por uma instrução *store* com uma leitura de instrução ocorrendo. O MIPS soluciona este perigo utilizando dois tipos de memória(dados e instruções) e toda sua arquitetura é própria para não ocorrer perigo estrutural. Nas seções seguintes é detalhado os perigos de dados e controle.

3.5.3.1 Perigo de Dados

O perigo de dados ocorre quando uma instrução depende do resultado de outra para executar corretamente. É o caso de instruções *load* seguidas por outras que utilizam o registrador que está sendo escrito. Neste caso é necessário ocasionar uma espera na execução do processador, o que é chamado *stall* ou bolha. As bolhas são instruções que não executam nenhuma operação, atrasando a execução do programa em um ou mais ciclos para que este conflito dos dados não ocorram.

Uma das técnicas utilizadas para evitar que bolhas ocorram é chamada de *forwarding* ou *bypass*. Esta técnica consiste em adiantar o valor do registrador de destino para o estágio em que será utilizado pela próxima instrução. Ainda assim existem combinações de instruções que causam bolhas na execução, e isto será tratado ainda nesta seção.

Na implementação para detectar o perigo de dados utilizando *forwarding*, insere-se a unidade de *forwarding* que possui em suas entradas dados dos estágios 3, 4 e 5.

Os dados do estágio 3 são dos campos rs e rt da instrução, propagados pelo

registrador ID/EX, que são comparados com o campo rd do estágio 4. Se iguais o dado utilizado no estágio 3 tem origem no registrador EX/MEM do estágio 4, mesmo dado que será escrito na memória ou no banco de registradores.

No estágio 4 é utilizado o valor do campo rd e caso no estágio 3 seja igual ao campo rs ou rt do registrador ID/EX, este dado é passado à instrução antes de ser escrito no banco de registradores.

A unidade de *forwarding* também trata uma situação em que um dado de instrução *load* é escrito com *store* na instrução seguinte.

A unidade de *forwarding* possui duas saídas que são *flags* para controlar dois multiplexadores que são adicionados ao estágio 3. Os multiplexadores são adicionados entre o registrador ID/EX e a ULA, sendo a saída do multiplexador A conectada a uma entrada da ULA e a saída do multiplexador B conectada ao multiplexador que seleciona se o dado que vai pra ULA tem origem no multiplexador de *forwarding* ou no campo da constante para instruções do tipo I. As entradas do multiplexador A são a saída da leitura de registrador do banco de registradores, a saída do multiplexador do estágio 5 que fornece dados para escrita no banco de registradores e o dado de uma operação realizada na ULA saindo do registrador EX/MEM no estágio 4. O multiplexador B tem como entradas os dados de leitura 2 do banco de registradores e as outras entradas são as mesmas do multiplexador A.

No perigo de dados é implementado um componente de detecção de perigo, que verifica se uma instrução *load* irá escrever um dado em um registrador que será utilizado para uma operação na instrução seguinte e caso isto ocorra é inserido uma bolha no *pipeline* para evitar o erro. Este componente possui em suas entradas os registradores rs e rt do estágio 2 e o registrador rt do estágio 3. Suas saídas são *flags* de controle que determinam quando a bolha será inserida, assim desativando a escrita do contador de programa(permanece na instrução atual), do registrador de *pipeline* IF/ID e do multiplexador do estágio 2 zerando as *flags* de controle(sem operações).

O modelo com estas alterações é exibido na Figura 3.20.

3.5.3.2 Perigo de Controle

O perigo de controle ocorre em instruções de desvio. Na arquitetura atual a execução de desvio é finalizada somente no estágio 4, causando bolhas no *pipeline* devido à próxima instrução não ser executada até a decisão ser tomada.

Implementando o perigo de controle ou de desvios, deslocamos o somador e o deslocador de bits do estágio 3 para o estágio 2 e adicionamos um comparador às saídas do banco de registradores substituindo as comparações feitas na ULA, assim reduzindo em um ciclo de relógio o custo para instruções de desvios condicionais. É necessário tratar o perigo de dados que pode ocorrer com a comparação dos registradores do estágio 2 adicionando dois multiplexadores(C e D) entre o banco de registradores e o comparador, que possui uma entrada para o código da operação para diferenciar entre instruções *beq* e *bne*. Para controlar estes multiplexadores utilizamos a unidade de forwarding com duas *flags* de controle.

Nos desvios condicionais é utilizada a previsão estática que considera que os desvios não serão tomados. Caso o desvio seja realizado, uma alteração na implementação do registrador IF/ID inclui uma *flag* de controle *flush*, ocasionando uma instrução de "não operação"(nop) aos outros estágios, permitindo que o programa realize o desvio sem executar instruções incorretamente.

Nos desvios incondicionais para *jumps*(j e jal) definimos o endereço de desvio no estágio 2. Para *jump register* conectamos a saída da leitura de registradores diretamente ao multiplexador do contador de programa, definindo o endereço de desvio.

O modelo com estas alterações é exibido na Figura 3.20. Na próxima seção é explicado o funcionamento do *pipeline*.

3.5.4 Controle Principal do *Pipeline*

No controle do *pipeline* não é necessário máquina de estados pois todas as instruções passam pelos cinco estágios, resumindo o controle à sincronização das *flags* de acordo com o *opcode* da instrução. Nos estágios 1 e 2 não é utilizado

flag pois executam a busca e decodificação da instrução. As *flags* são introduzidas a partir do estágio 3 e são propagadas aos outros estágios pelos registradores do *pipeline*.

Além das *flags* utilizadas pela unidade de detecção de perigos e da unidade de *forwarding*, o *pipeline* utiliza *reg_dst*, *mem_read*, *mem_write*, *reg_write*, *mem_to_reg*, *alu_op*, *pc_source*, *alu_src* e *branch*. A *flag alu_src* controla um mux no estágio 3 e determina se o dado para ULA tem origem no mux B do *forwarding* ou nos *bits* estendidos no estágio 2 para instruções que utilizam o operador imediato. A *flag branch* determina se um desvio será realizado. As outras *flags* tem a mesma função descrita na Seção 3.4.1.

A Tabela 3.16 exibe o valor das *flags* em relação à instrução.

Tabela 3.16: Resumo das *flags* de controle do *pipeline*.

***flags*/instrução**	**R**	**addi**	**slti**	**beq**	**bne**	**jal**	**jr**	**j**	**lw**	**sw**	**mul**
reg_dst	01	00	00	00	00	10	01	00	00	00	01
alu_src	0	1	1	0	0	0	0	0	1	1	0
branch	0	0	0	1	1	0	0	0	0	0	0
mem_read	0	0	0	0	0	0	0	0	1	0	0
mem_write	0	0	0	0	0	0	0	0	0	1	0
reg_write	1	1	1	0	0	1	0	0	1	0	1
mem_to_reg	00	00	00	00	00	10	00	00	01	01	00
alu_op	010	000	011	001	100	000	010	000	000	000	101
pc_src	00	00	00	00	00	10	11	10	00	00	00

Cada estágio do *pipeline* representa uma etapa quando comparado à execução multiciclo e combinado com o controle das *flags* temos a implementação do processador.

A Figura 3.20 exibe a arquitetura do processador MIPS *pipeline* com as *flags*, registradores do *pipeline*, unidade de detecção de perigos e a unidade de *forwarding*.

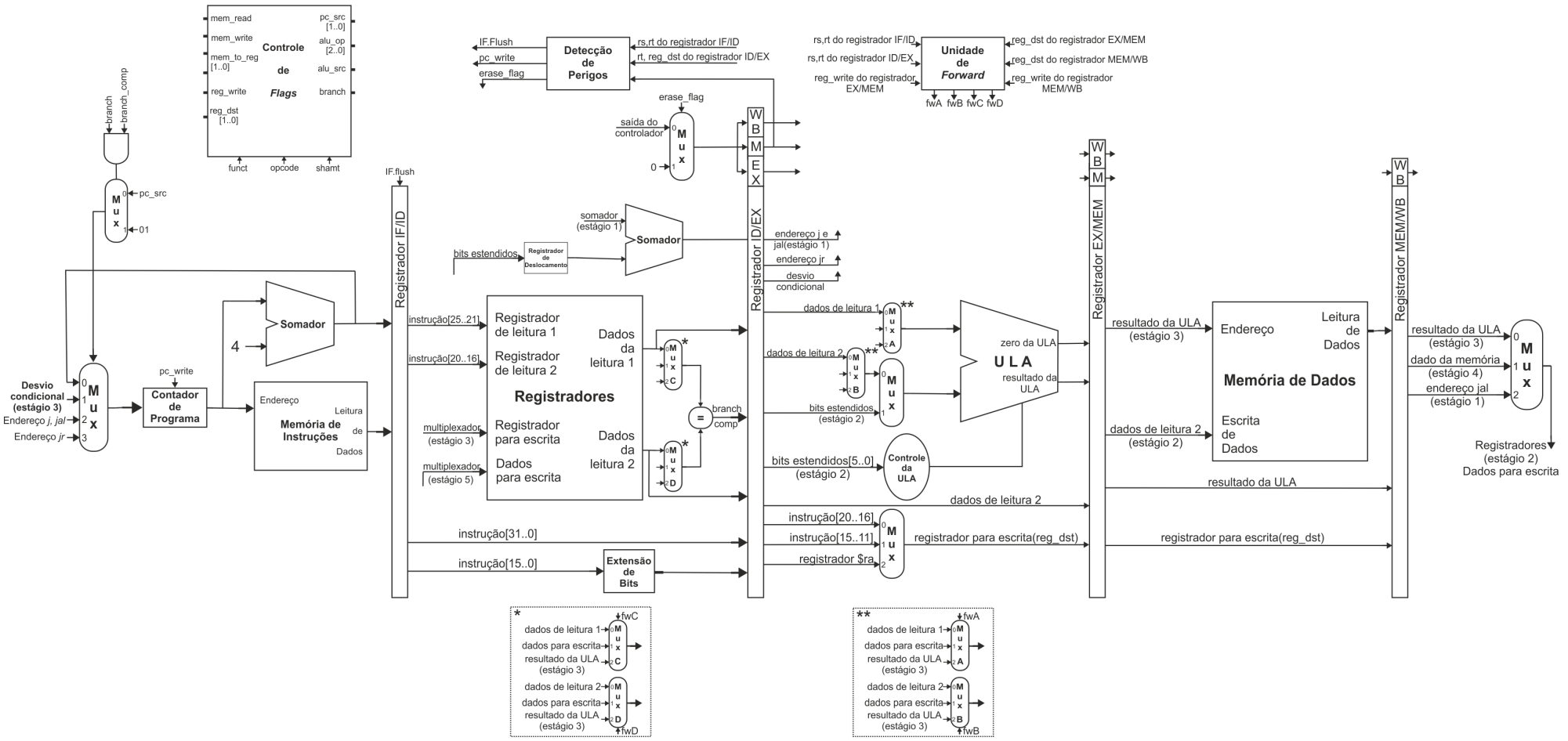

Figura 3.20: Modelo do processador MIPS *pipeline*.

Assim é concluída a implementação dos processadores MIPS e o capítulo a seguir apresenta os resultados obtidos com validação da execução dos processadores utilizando dois *benchmarks* e teste de funcionamento dos componentes.

Capítulo 4

Validação e Resultados

Neste capítulo será apresentado os testes de validação dos componentes implementados e sua integração utilizando o *software* Qsim[2] desenvolvido pela Altera, mostrando em uma *timeline* o comportamento do *hardware* ao ser submetido a entradas determinadas. Nos resultados será exibido o processamento de dois *benchmarks* utilizando os recursos da simulação com o Qsim e os processador implementado será embarcado em uma placa de FPGA do modelo Stratix[4], desenvolvida pela Altera, para certificar seu funcionamento. Por fim será feita uma análise sobre os resultados.

4.1 Teste Unitário dos Componentes

Nesta seção utilizaremos o software QSim desenvolvido pela Altera para visualizarmos a *timeline* que exibe a forma de onda da resposta dos componentes quando submetidos a determinadas entradas.

4.1.1 Contador de Programas

No contador de programas a saída é igual a entrada quando temos uma transição na subida do ciclo de relógio e a *flag pc_ write* está ativada(valor alto).

Caso de teste 1:

1) Entrada do sistema: pulso de relógio na transição de subida, porta de entrada

com o endereço 0x2A, *flag pc_write* desativada.

2) O passo a passo da execução dos testes : simular a utilização do contador de programas conforme a Figura 4.1.

3) Saída esperada do sistema : valor 0x0 na porta de saída.

4) Saída real do sistema : valor 0x0 na porta de saída.

Caso de teste 2:

1) Entrada do sistema: pulso de relógio na transição de subida, porta de entrada com o endereço 0x2A, *flag pc_write* ativada

2) O passo a passo da execução dos testes : simular a utilização do contador de programas conforme a Figura 4.1

3) Saída esperada do sistema : valor 0x2A na porta de saída

4) Saída real do sistema : valor 0x2A na porta de saída

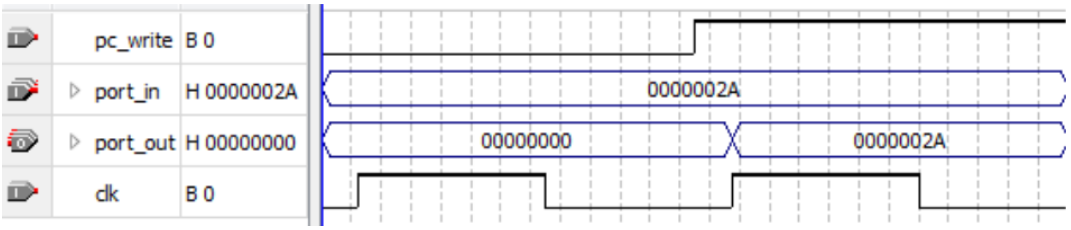

Figura 4.1: Teste do contador de programa gerado pelo QSim.

4.1.2 Memória

A memória tem uma porta de entrada para endereço e outra para dados para escrita, duas *flags* de controle e uma porta para saída de dados. A porta de endereço seleciona um dado para porta de saída quando a *flag mem_read* está ativada. Quando a *flag mem_write* está ativada a porta de endereço seleciona em qual posição de memória o dado da porta de dados para escrita será gravado.

Caso de teste 1:

1) Entrada do sistema: pulso de relógio na transição de subida, *flag mem_read* ativada, *flag mem_write* desativada, endereço 0x10, dado para escrita 0xFFFB1A, valor armazenado no endereço 0x10 é 0xAC020000

2) O passo a passo da execução dos testes : simular a utilização da memória conforme a Figura 4.2

3) Saída esperada do sistema : valor 0xAC020000 na porta de saída

4) Saída real do sistema : valor 0xAC020000 na porta de saída

Caso de teste 2:

1) Entrada do sistema: pulso de relógio na transição de subida, *flag mem_read* desativada, *flag mem_write* ativada, endereço 0x10, dado para escrita 0xFFFB1A, valor armazenado no endereço 0x10 é 0xAC020000

2) O passo a passo da execução dos testes : simular a utilização da memória conforme a Figura 4.2

3) Saída esperada do sistema : valor 0xFFFB1A no endereço 0x10

4) Saída real do sistema : valor 0xFFFB1A no endereço 0x10

Caso de teste 3:

1) Entrada do sistema: pulso de relógio na transição de subida, *flag mem_read* ativada, *flag mem_write* desativada, endereço 0x10, dado para escrita 0xFFFB1A, valor armazenado no endereço 0x10 é 0xFFFB1A

2) O passo a passo da execução dos testes : simular a utilização da memória conforme a Figura 4.2

3) Saída esperada do sistema : valor 0xFFFB1A na porta de saída

4) Saída real do sistema : valor 0xFFFB1A no na porta de saída

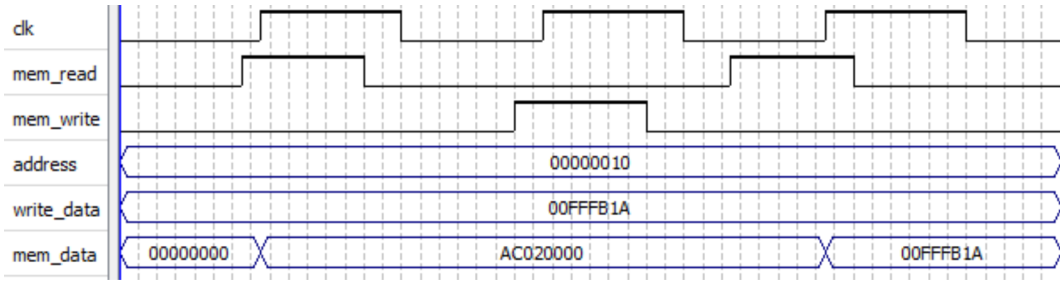

Figura 4.2: Teste da memória gerado pelo QSim.

4.1.3 Registrador de Instrução

O registrador de instrução opera na transição de subida do ciclo de relógio decodificando o valor de entrada e propagando para as portas de saída quando a *flag ir_write* está ativada.

Caso de teste 1:

1) Entrada do sistema: pulso de relógio na transição de subida, *flag ir_ write* ativada, dado "10010101010101000101110101010101"em binário na porta de entrada

2) O passo a passo da execução dos testes : simular o registrador de instrução conforme a Figura 4.3

3) Saída esperada do sistema :

valor "100101"no campo de instrução[31..26]

valor "01010"no campo de instrução[25..21]

valor "10100"no campo de instrução[20..16]

valor "0101110101010101"no campo de instrução[15..0]

4) Saída real do sistema :

valor "100101"no campo de instrução[31..26]

valor "01010"no campo de instrução[25..21]

valor "10100"no campo de instrução[20..16]

valor "0101110101010101"no campo de instrução[15..0]

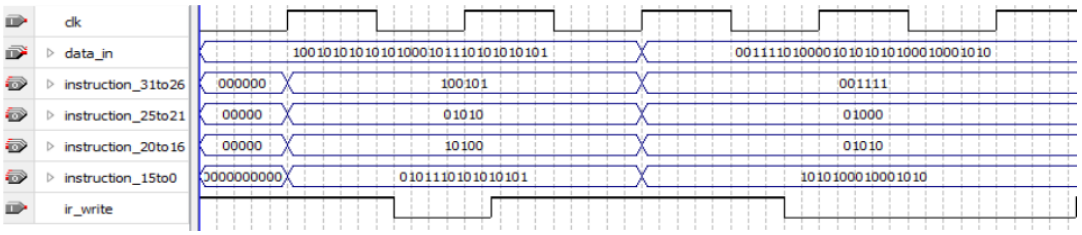

Figura 4.3: Teste do registrador de instruções gerado pelo QSim.

4.1.4 Registrador

O teste do registrador mostra o comportamento de todos os registradores utilizados no projeto(registrador de dados da memória, registrador A e B na saída do banco de registradores e o registrador *saída da ULA*. O comportamento esperado do registrador propaga o dado da entrada para saída somente na subida do ciclo de relógio.

Caso de teste 1:

1) Entrada do sistema: pulso de relógio na transição de subida, valor 0x010F32FD na porta de entrada

2) O passo a passo da execução dos testes : simular o registrador conforme a Figura

4.4

3) Saída esperada do sistema : valor 0x010F32FD na porta de saída

4) Saída real do sistema : valor 0x010F32FD na porta de saída

Caso de teste 2:

1) Entrada do sistema: pulso de relógio na transição de subida, valor 0x245BC21A na porta de entrada

2) O passo a passo da execução dos testes : simular o registrador conforme a Figura 4.4

3) Saída esperada do sistema : valor 0x245BC21A na porta de saída

4) Saída real do sistema : valor 0x245BC21A na porta de saída

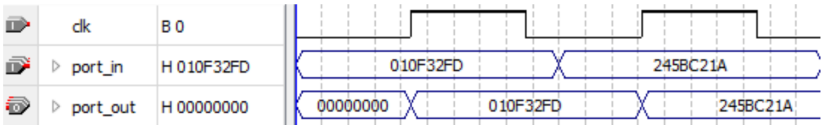

Figura 4.4: Teste do registrador gerado pelo QSim.

4.1.5 Banco de Registradores

O teste realizado sobre o banco de registradores simula a escrita e leitura em um endereço. O comportamento esperado na leitura é a mudança de valor quando o endereço dos registradores a serem lidos forem alterados e na escrita é possível visualizar o funcionamento correto quando escrevemos em um mesmo endereço que está sendo lido e o valor da porta de saída referente ao registrador é alterado. A escrita ocorre na transição de subida do ciclo de relógio quando a *flag reg_ write* está ativada.

Caso de teste 1:

1) Entrada do sistema: pulso de relógio na transição de subida, *flag reg_ write* ativada, valor 0x0D para o endereço do registrador de escrita, valor 0x14DF23 na entrada de dados para escrita, valor 0x0D no endereço da porta do registrador de leitura 1, valor 0x05 no endereço da porta do registrador de leitura 2

2) O passo a passo da execução dos testes : simular o banco de registradores conforme a Figura 4.5

3) Saída esperada do sistema : valor 0x14DF23 na porta de saída de dados 1, valor

0x0 na porta de saída de dados 2

4) Saída real do sistema : valor 0x14DF23 na porta de saída de dados 1, valor 0x0 na porta de saída de dados 2

Caso de teste 2:

1) Entrada do sistema: pulso de relógio na transição de subida, *flag reg_write* ativada, valor 0x05 para o endereço do registrador de escrita, valor 0xF3AB124 na entrada de dados para escrita, valor 0x0D no endereço da porta do registrador de leitura 1, valor 0x05 no endereço da porta do registrador de leitura 2

2) O passo a passo da execução dos testes : simular o banco de registradores conforme a Figura 4.5

3) Saída esperada do sistema : valor 0x14DF23 na porta de saída de dados 1, valor 0xF3AB124 na porta de saída de dados 2

4) Saída real do sistema : valor 0x14DF23 na porta de saída de dados 1, valor 0xF3AB124 na porta de saída de dados 2

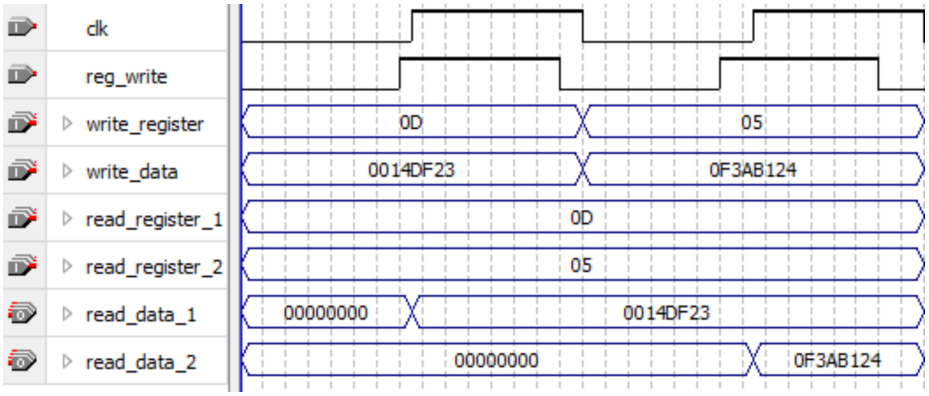

Figura 4.5: Teste do banco de registradores gerado pelo QSim.

4.1.6 Extensor de Sinal

O comportamento esperado do extensor de sinal é ter o dado de entrada com 16 bits estendido na saída para 32 bits de acordo com seu sinal.

Caso de teste 1:

1) Entrada do sistema: valor 0x04AB na porta de entrada

2) O passo a passo da execução dos testes : simular o extensor de sinal conforme a Figura 4.6

3) Saída esperada do sistema : valor 0x000004AB na porta de saída

4) Saída real do sistema : valor 0x000004AB na porta de saída

Caso de teste 2:

1) Entrada do sistema: valor 0xF01C na porta de entrada

2) O passo a passo da execução dos testes : simular o extensor de sinal conforme a Figura 4.6

3) Saída esperada do sistema : valor 0xFFFFF01C na porta de saída

4) Saída real do sistema : valor 0xFFFFF01C na porta de saída

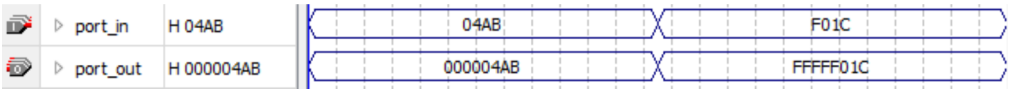

Figura 4.6: Teste do extensor de sinal.

4.1.7 Deslocador de Bits

O componente de deslocamento de bits está configurado para deslocar dois bits à esquerda.

Caso de teste 1:

1) Entrada do sistema: valor "00010010110011010000000100101010"na porta de entrada

2) O passo a passo da execução dos testes : simular o deslocador de bits conforme a Figura 4.7

3) Saída esperada do sistema : valor "01001011001101000000010010101000"na porta de saída

4) Saída real do sistema : valor "01001011001101000000010010101000"na porta de saída

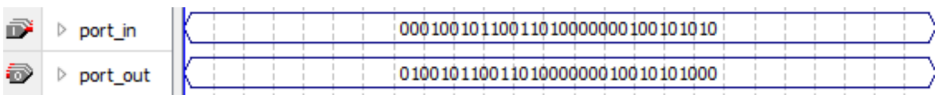

Figura 4.7: Teste do deslocador de bits gerado pelo QSim.

4.1.8 Multiplexador

O multiplexador possui uma *flag* seletora que seleciona qual entrada será propagada para saída.

Caso de teste 1:

1) Entrada do sistema: valor 0xF8000000 na porta de entrada 1, valor 0X0003E000 na porta de entrada 2, valor 0x0000003F na porta de entrada 3, valor 0x00FFE000 na porta de entrada 4, valor 00 na porta seletora

2) O passo a passo da execução dos testes : simular o multiplexador conforme a Figura 4.8

3) Saída esperada do sistema : valor 0xF8000000 na porta de saída

4) Saída real do sistema : valor 0xF8000000 na porta de saída

Caso de teste 2:

1) Entrada do sistema: valor 0xF8000000 na porta de entrada 1, valor 0X0003E000 na porta de entrada 2, valor 0x0000003F na porta de entrada 3, valor 0x00FFE000 na porta de entrada 4, valor 01 na porta seletora

2) O passo a passo da execução dos testes : simular o multiplexador conforme a Figura 4.8

3) Saída esperada do sistema : valor 0003E000 na porta de saída

4) Saída real do sistema : valor 0003E000 na porta de saída

Caso de teste 3:

1) Entrada do sistema: valor 0xF8000000 na porta de entrada 1, valor 0X0003E000 na porta de entrada 2, valor 0x0000003F na porta de entrada 3, valor 0x00FFE000 na porta de entrada 4, valor 00 na porta seletora

2) O passo a passo da execução dos testes : simular o multiplexador conforme a Figura 4.8

3) Saída esperada do sistema : valor 0x0000003F na porta de saída

4) Saída real do sistema : valor 0x0000003F na porta de saída

Caso de teste 4:

1) Entrada do sistema: valor 0xF8000000 na porta de entrada 1, valor 0X0003E000 na porta de entrada 2, valor 0x0000003F na porta de entrada 3, valor 0x00FFE000

na porta de entrada 4, valor 00 na porta seletora

2) O passo a passo da execução dos testes : simular o multiplexador conforme a Figura 4.8

3) Saída esperada do sistema : valor 0x00FFE000 na porta de saída

4) Saída real do sistema : valor 0x00FFE000 na porta de saída

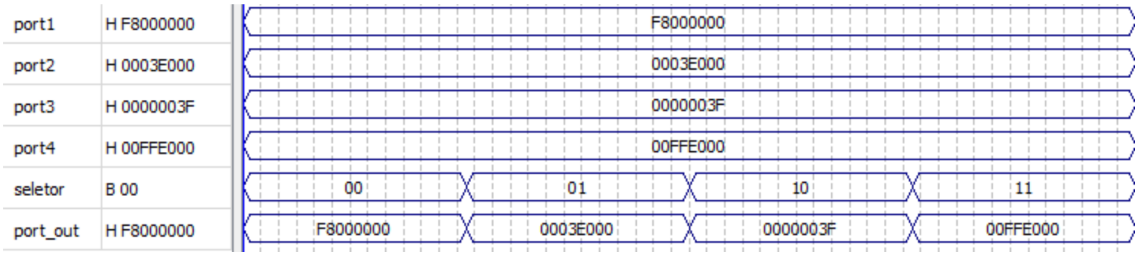

Figura 4.8: Teste do multiplexador gerado pelo QSim.

4.1.9 Unidade Lógica e Aritmética

No teste da ULA será observado o comportamento para oito operações, na sequência: *and*, *or*, *add*, *sub*, *slt*, *sub(bne)*, *shift* e *mult*. Na Figura 4.9 as entradas são as mesmas para cada operação e o valor observado nas saídas da ULA representa o resultado esperado na execução do componente.

Caso de teste 1:

1) Entrada do sistema: valor 0x3C3 na porta de entrada 1, valor 0X7C3 na porta de entrada 2, variar o valor do controle da ULA em 0000, 0001, 0010, 0110, 0111, 1000, 1001, 1010

2) O passo a passo da execução dos testes : simular o controlador da ULA conforme a Figura 4.9

3) Saída esperada do sistema : valores de acordo com a operação, 0x3C3, 0x7C3, 0xB86, 0xFFFFFC00, 0x01, 0xFFFFFC00, 0x01, 0xFFFFFC00 com a *flag zero_ alu* ativada, 0x1F0C, 0x1D3289

4) Saída real do sistema : valores 0x3C3, 0x7C3, 0xB86, 0xFFFFFC00, 0x01, 0xFFFFFC00, 0x01, 0xFFFFFC00 com a *flag zero_ alu* ativada, 0x1F0C, 0x1D3289

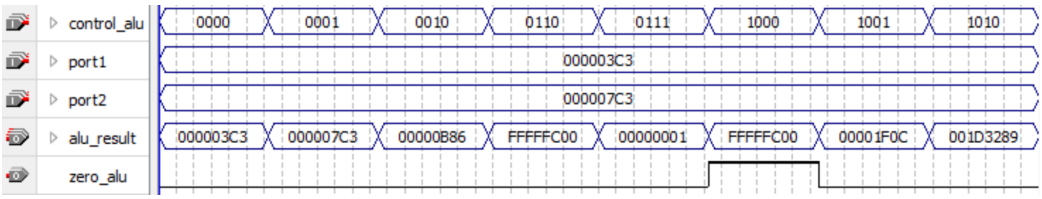

Figura 4.9: Teste da ULA gerado pelo QSim.

4.2 Simulação do Processador MIPS no *Software* QSim

Esta seção apresenta os resultados referente a simulação do processador MIPS implementado em suas abordagens multiciclo e *pipeline* executando sobre os algoritmos *sort* e fatorial. Para simulação foi utilizado o *software* QSim que exibe uma *timeline* com o valor de determinados sinais do processador. Os resultados serão analisados com base na saída esperada pela execução do algoritmo e o número de ciclos de relógio por instrução(CPI) será comparado entre as abordagens como métrica de desempenho.

4.2.1 Algoritmo *sort*

O algoritmo *sort* realiza a ordenação de vetores e foi escolhido por utilizar 12 das 15 operações suportadas pela implementação do processador. Além disso sua execução explora as instruções de leitura e escrita na memória que utilizam cinco e quatro ciclos de relógio respectivamente causando um valor alto de CPI para o processador multiciclo. Para o processador *pipeline* além das instruções de leitura e escrita na memória, os devios condicionais e incondicionais presentes neste algoritmo exigem desempenho dos componentes de tratamento de perigo. No quadro abaixo é observado o algoritmo da função *sort* e a função *swap* desenvolvidas na linguagem C.

```
1 void sort (int v[], int n) {
2       int i,j;
3       for (i=0;i<n;i+=1){
4         for(j=i-1;j>=0 && v[j]>v[j+1];j-=1){
5           swap(v,j);
```

```
6           }
7         }
8 }
9
10
11 void swap(int v[], int k){
12         int temp;
13         temp = v[k];
14         v[k] = v[k+1];
15         v[k+1] = temp;
16 }
```

Este algoritmo foi traduzido para a linguagem *assembly* do processador MIPS com o auxílio do *software* MARS(*MIPS Assembler and Runtime Simulator)* que simula a execução de algoritmos para o processador MIPS. Na Tabela 4.1 temos o código em *assembly* e em hexadecimal.

Para a simulação do algoritmo *sort* foi considerado o seguinte caso de teste:
1) Entrada do sistema: vetor de tamanho 4 com os elementos 22, 5, -2, 10
2) Execução do teste: o processador inicia a execução do algoritmo armazenado na memória
3) Saída esperada do sistema: o vetor com os valores -2, 5, 10, 22 nesta ordem
4) Saída real do sistema: vetor com os valores -2, 5, 10, 22 ordenados

Na Figura 4.10 é exibido o resultado da simulação pelo *software* Qsim na abordagem multiciclo, onde é observado o resultado correto da ordenação do vetor. Além disso, é visto que para processar este algoritmo foram necessários 479 ciclos de relógio em 122 instruções gerando um CPI igual a 3,92.

Na Figura 4.11 é exibido o resultado da simulação pelo *software* Qsim na abordagem *pipeline*, e também é observada a correta ordenação do vetor. Além disso, é visto que para processar este algoritmo foram necessários 161 ciclos de relógio em 149 instruções gerando um CPI igual a 1,08.

Tabela 4.1: Algoritmo *sort* desenvolvido na linguagem assembly.

Assembly do MIPS	**Instrução(hexadecimal)**
addi $a0,$0,160	200400A0
addi $a1,$0,10	2005000A
add $s2,$0,$a0	00049020
add $s3,$0,$a1	00059820
add $s0,$0,$0	00008020
for1tst: Slt $t0,$s0,$s3	0213402A
beq $t0,$0,exit	11000010
addi $s1,$s0,-1	2211FFFF
for2tst: slti $t0,$s1,0	2A280000
bne $t0,$0,exit2(11)	1500000B
sll $t1,$s1,2	00114880
add $t2,$s2,$t1	02495020
lw $t3,0($t2)	8D4B0000
lw $t4,4($t2)	8D4C0004
slt $t0,$t4,$t3	018B402A
beq $t0,$0,exit2(5)	11000005
add $a0,$0,$s2	00122020
add $a1,$0,$s1	00112820
jal swap(25)	0C000019
addi $s1,$s1,-1	2231FFFF
j for2tst(8)	08000008
exit2: addi $s0,$s0,1	22100001
j for1tst(5)	08000005
exit: sll $0,$0,0 (nop)	00000000
sll $0,$0,0 (nop)	00000000
swap: sll $t1,$a1,2	00054880
add $t1,$a0,$t1	00894820
lw $t0,0($t1)	8D280000
lw $t2,4($t1)	8D2A0004
sw $t2,0($t1)	AD2A0000
sw $t0,4($t1)	AD280004
jr $ra	03E00008

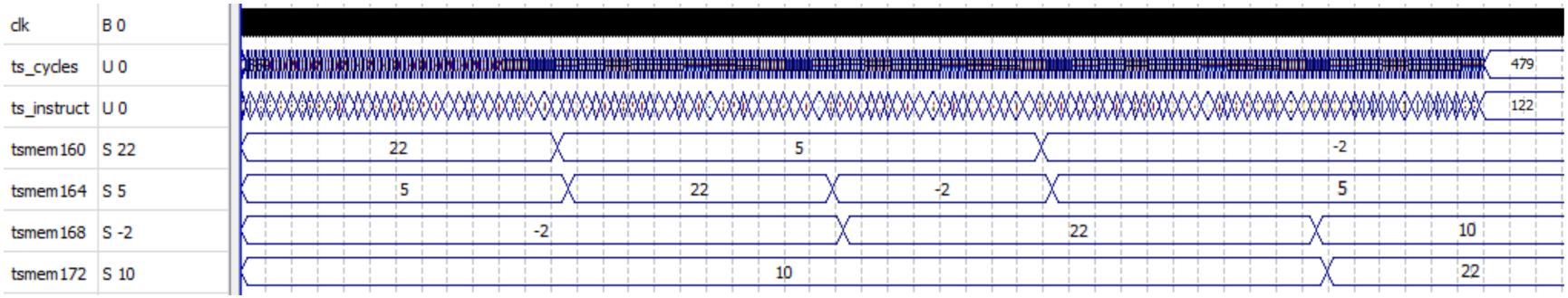

Figura 4.10: Simulação com o *software* Qsim do algoritmo *sort* na abordagem multiciclo.

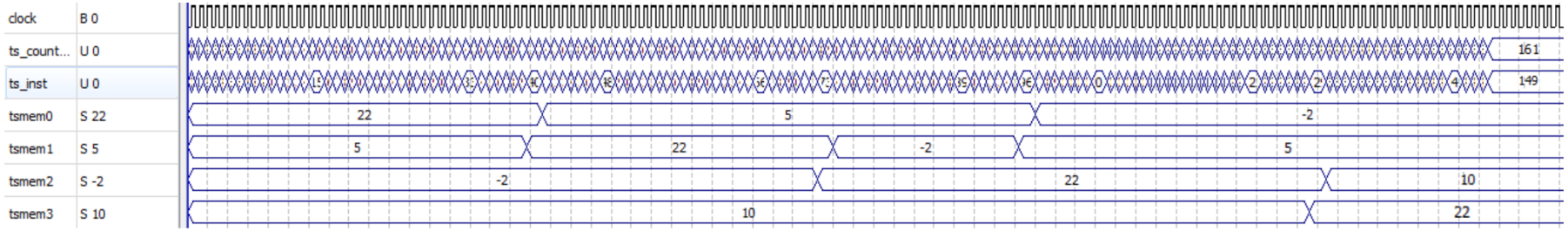

Figura 4.11: Simulação com o *software* Qsim do algoritmo *sort* na abordagem *pipeline*.

4.2.2 Algoritmo fatorial

O algoritmo fatorial utiliza 8 das 15 operações implementadas e sua execução explora o processamento da recursão que exige leitura e escrita na memória para salvar os parâmetros das chamadas anteriores e das instruções de *jump* para navegar no código. No quadro abaixo é observado o algoritmo fatorial desenvolvido na linguagem C.

```
1 int fatorial(int n){
2   if(n<1) return (1);
3   else return (n * fatorial(n-1));
4 }
```

Este algoritmo foi traduzido para a linguagem *assembly* do processador MIPS com o auxílio do *software* MARS(*MIPS Assembler and Runtime Simulator).*. Na Tabela 4.2 temos o código em *assembly* e em hexadecimal.

Tabela 4.2: Algoritmo fatorial desenvolvido na linguagem assembly.

Assembly do MIPS	**Instrução(hexadecimal)**
addi $sp,$0,252	201D00FC
addi $a0,$0,5	20040005
jal fact(5)	0C100005
sw $v0,0($s0)	AC020000
sll $0,$0,0 (nop)	00000000
fact: addi $sp,$sp,-8	23BDFFF8
sw $ra,4($sp)	AFBF0004
sw $a0,0($sp)	AFA40000
slti $t0,$a0, 1	28880001
beq $t0,$0,L1(3)	11000003
addi $v0,$0,1	20020001
addi $sp,$sp,8	23BD0008
jr $ra	03E00008
L1: addi $a0,$a0,-1	2084FFFF
jal fact(5)	0C100005
lw $a0,0($sp)	8FA40000
lw $ra,4($sp)	8FBF0004
addi $sp,$sp,8	23BD0008
mul $v0,$a0,$v0	70821002
jr $ra	03E00008

Para a simulação do algoritmo fatorial foi considerado o seguinte caso de teste:

1) Entrada do sistema: valor 5

2) Execução do teste: o processador inicia a execução do algoritmo armazenado na memória

3) Saída esperada do sistema: resultado do fatorial de 5 que e igual a 120

4) Saída real do sistema: valor 120

Na Figura 4.12 é exibido o resultado da simulação pelo *software* Qsim na abordagem multiciclo, onde é observado o resultado esperado na saída do sistema. Além disso, é visto que para processar este algoritmo foram necessários 295 ciclos de relógio em 73 instruções gerando um CPI igual a 4,04.

Na Figura 4.13 é exibido o resultado da simulação pelo *software* Qsim na abordagem *pipeline*, onde é observada a saída espera do sistema. Além disso, é visto que para processar este algoritmo foram necessários 102 ciclos de relógio em 79 instruções gerando um CPI igual a 1,29.

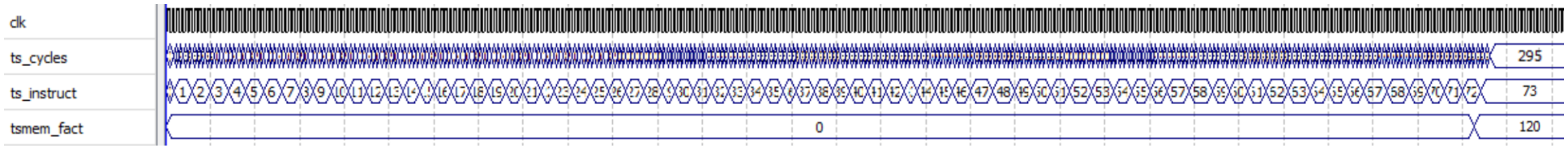

Figura 4.12: Simulação com o *software* Qsim do algoritmo fatorial na abordagem multiciclo.

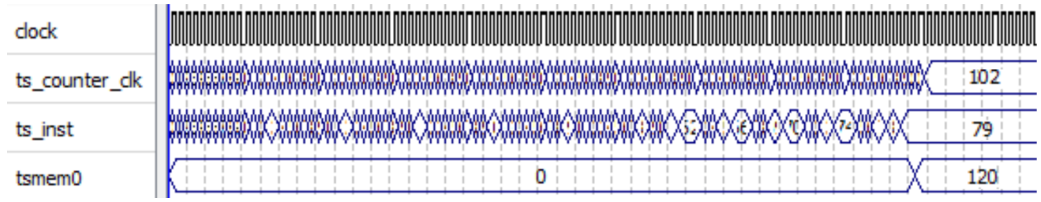

Figura 4.13: Simulação com o *software* Qsim do algoritmo fatorial na abordagem *pipeline*.

4.3 Processador MIPS embarcado na FPGA

Nesta seção serão exibidos os resultados da execução do processador embarcado na FPGA. Serão mostradas as configurações para utilizar a placa e os resultados obtidos nas duas abordagens implementadas.

4.3.1 Configurações para Embarcar o Processador na FPGA

Para o funcionamento do processador na placa de FPGA foi necessário configurar os pinos de entrada e saída de dados. A placa utilizada é do modelo Stratix EP1S10F780C6 que possui 8 *megabytes* de memória *flash*, 1 *megabyte* de memória RAM(*Random Access Memory)*, 41 pinos de entrada e saída e 50 *megahertz* de frequência de relógio[4]. Na tabela 4.3 é detalhada as configurações necessárias como os botões de *start* e *reset*, entrada do relógio e sinais de configuração para saída de dados para o visor de LCD(*Liquid Crystal Display*).

Tabela 4.3: Configuraçao de pinos na placa de FPGA Stratix EP1S10F780C6.

Porta	**Pino**
Relógio	K17
Start	W5
Reset	W6
E	K3
RS	M7
RW	M8
DB[7]	J4
DB[6]	L5
DB[5]	L6
DB[4]	H1
DB[3]	H2
DB[2]	L8
DB[1]	L7
DB[0]	H3

Com estas configurações realizadas podemos analisar os resultados obtidos pela execução do processador.

4.3.2 Processamento dos *Benchmarks*

O código *assembly*(Seções 4.2.1 e 4.2.2) dos *benchmarks* utilizados são salvos no arquivo de descrição da memória(para abordagem multiciclo, memória de dados para abordagem *pipeline*) e o processador é embarcado na placa.

A Figura 4.14(a) mostra o LCD com os elementos do vetor de entrada utilizado para testar o funcionamento do algoritmo *sort* embarcado na FPGA. Na Figura 4.14(b) é mostrado os elementos após o processamento. O resultado da ordenação foi o mesmo para a implementação multiciclo e *pipeline*.

(a) (b)

Figura 4.14: Em (a) o vetor com os elementos antes do processamento e em (b) o vetor com os elementos ordenados pelo algoritmo *sort*.

O processador multiciclo realizou a ordenação deste vetor em 2941 ciclos de relógio e 734 instruções, como mostrado na Figura 4.15(a), gerando um CPI igual a 4,00 enquanto o processador *pipeline* realizou em 941 ciclos de relógio e 869 instruções, como mostrado na Figura 4.15(b), gerando um CPI igual a 1,08.

(a) (b)

Figura 4.15: Em (a) o número de ciclos e instruções do multiciclo e em (b) *pipeline* do processamento do algoritmo *sort*.

No parâmetro do algoritmo fatorial foi colocado como valor o número 7. Devido a limitações da placa de FPGA não foi possível exibir no LCD o resultado da operação, pois o número de elementos lógicos ultrapassa o suportado pela placa. A validação do resultado foi feita com o simulador comparando número de instruções e ciclos de relógio utilizados por ambos na execução do algoritmo. O resultado da operação foi o mesmo para a implementação multiciclo e *pipeline*.

O processador multiciclo executou o cálculo fatorial de 7 em 393 ciclos de re-

lógio e 97 instruções, como mostrado na Figura 4.16(a), gerando um CPI igual a 4,05 enquanto o processador *pipeline* realizou em 135 ciclos de relógio e 105 instruções, como mostrado na Figura 4.16(b), gerando um CPI igual a 1,28.

(a) (b)

Figura 4.16: Em (a) o número de ciclos e instruções do multiciclo e em (b) *pipeline* do processamento do algoritmo fatorial.

4.4 Análise dos Resultados

Os valores de ciclos de relógio e instruções tiveram o mesmo resultado quando simulado no Qsim ou embarcado na FPGA, por isso foi decidido simular o algoritmo *sort* com um número de elementos diferente do algoritmo embarcado o que permitiu comparações de CPI para parâmetros de tamanhos diferentes. O mesmo foi realizado com o algoritmo fatorial que calcula o fatorial de 5 na simulação e o fatorial de 7 na FPGA.

A maior variação de CPI entre o Qsim e a FPGA pode ser observada no algoritmo *sort* multiciclo, quando variado o tamanho do vetor, que apresentou uma mudança de 3,92 para 4,00 no CPI. Isto ocorre devido à execução de instruções *load* e *store* que necessitam de cinco e quatro ciclos de relógio, respectivamente, para executar elevando o CPI. Os outros testes se mostraram estáveis quanto à execução com parâmetros diferentes.

Analisando o desempenho no Qsim o algoritmo *sort* multiciclo utiliza 3,62 vezes mais ciclos por instrução enquanto no fatorial a proporção é de 3,13. O desempenho do fatorial *pipeline* é mais lento quando comparado ao *sort* por executar a recursão, exigindo escrita e leitura de memória para armazenar os parâmetros entre as chamadas.

Analisando o desempenho na FPGA é observado que para o algoritmo *sort* a abordagem multiciclo utiliza 3,7 vezes mais ciclos por instrução em relação a

pipeline, e o fatorial multiciclo utiliza 3,16 vezes mais ciclos que o *pipeline*. Estes valores reforçam a diferença de desempenho obtida coma utilização da técnica de *pipeline*.

Na técnica de *pipeline* pode ser observado a proximidade das instruções para executar utilizando um ciclo por instrução, situação característica da técnica que quanto maior o número de instruções executadas menor o seu valor de CPI. O que mantém o valor do CPI diferente maior que um são as bolhas inseridas na execução dos algoritmos devido aos perigos explicados na Seção 3.5.3. A presença das bolhas pode ser observada na diferença da quantidade de instruções entre as abordagens multiciclo e *pipeline*. Em tese a execução de um algoritmo não poderia ser diferente em número de instruções, porém como no *pipeline* são inseridas bolhas que contam como instrução, um número maior de instruções é realizado.

Na Tabela 4.4 resume os resultados obtidos neste trabalho.

Tabela 4.4: Resultados da implementação do processador MIPS.

Algoritmo	**Qsim**			**FPGA**		
	Ciclos	Instruções	CPI	Ciclos	Instruções	CPI
sort multiciclo[1]	479	122	3,92	2941	734	4,00
sort pipeline[1]	161	149	1,08	941	869	1,08
fatorial multiciclo[2]	295	73	4,04	393	97	4,05
fatorial pipeline[2]	102	79	1,29	135	105	1,28

[1]No Qsim foi utilizado um vetor com 4 elementos e na FPGA um vetor com 10 elementos para simulação e execução do processador.

[2]No Qsim foi utilizado o fatorial de 5 e na FPGA fatorial de 7.

Capítulo 5

Conclusões

A arquitetura do processador MIPS foi detalhada desde sua implementação até o seu funcionamento. As características de registradores e instruções do processador foram respeitadas e o desenvolvimento foi baseado nas instruções que foram propostas para serem suportadas. O projeto da arquitetura multiciclo foi abordado considerando o caminho de dados que é executado pelas instruções apresentando as *flags* de controle que o controlador principal coordena por máquinas de estados, e o modelo foi ilustrado para simplificar o entendimento do que foi construído. A arquitetura *pipeline* foi descrita através de modelos incrementais para facilitar a adaptação dos componentes detectores e solucionadores dos perigos encontrados na execução das instruções. O controlador do pipeline também foi descrito e pode ser observado sua implementação mais simples devido atuar diretamente nas *flags* de controle e de todas as instruções propagadas através dos cinco estágios do *pipeline*.

Os componentes foram descritos na linguagem VHDL e o comportamento destes foi validado por testes que apresentaram os resultados esperados. A arquitetura *pipeline* apresentou suas diferenças estruturais com as unidades de tratamento e detecção de perigos além dos registradores de *pipeline* característicos. Além disso, a memória de dados é separada da memória de instruções por ser acessada por diferentes estágios ao mesmo tempo.

Os resultados de simulação e execução com FPGA certificaram o êxito na implementação proposta e pode ser avaliada a diferença de desempenho entre as

abordagens, utilizando as medidas de CPI para comparação entre processadores. Os valores mostraram o que era esperado na definição teórica das arquiteturas certificando o desempenho superior da abordagem *pipeline* nos *benchmarks* testados.

5.1 Considerações Finais

No desenvolvimento deste trabalho foram encontradas dificuldades para implementar instruções e situações que ocasionam perigos na execução dos processadores e não são ilustradas pela proposta de [1]. Além disso, a sincronização dos componentes quanto a transição do pulso de relógio exige bastante esforço na interconexão destes para definir uma arquitetura. Outro problema foi enfrentado na capacidade lógica limitada da placa Stratix EP1S10F760C6 utilizada para simulação dos *benchmarks* impedindo a exibição de alguns resultados gerados pelos algoritmos.

Com os resultados deste trabalho, é proposto para trabalhos futuros aumentar o suporte a instruções do *assembly* MIPS e executar o processador sobre uma maior gama de *benchmarks*. Utilizar placas de FPGA mais atuais para que seja possível o desenvolvimento de aplicações que executem no processador MIPS desenvolvido. Fornecer o suporte a previsão de desvios dinâmico para melhorar o desempenho do *pipeline* e a exceções além de modernizar a implementação da ULA para detecção de *overflow* aritmético. Por fim utilizar mais conceitos e técnicas de teste de *software* como a verificação formal de modelos para dar maior robustez e segurança ao *hardware* descrito.

Referências Bibliográficas

[1] PATTERSON DAVID A.; HENNESSY, J. L. *Organização e Projeto de Computadores:a interface hardware/software*. Rio de Janeiro: Elsevier, 2005.

[2] ALTERA Corporation. *Qsim Version 11.0 Build 208 Web Edition*. Copyright 1991-2011.

[3] ALTERA Corporation. *Quartus II Version 11.0 Build 208 Web Edition*. Copyright 1991-2011.

[4] ALTERA Corporation. *Nios Development Board Version 1.0, Stratix EP1S10F780C6 device*. Copyright 2003.

[5] ASHENDEN, P. J. *The VHDL Cookbook*. Australia: University of Adelaide, 1990.

[6] MORTON, D. P. *Hardware Modeling and Top-Down Design Using VHDL*. Dissertação (Mestrado), Cambridge, 1991.

[7] KRUG, M. R. *Aumento da Testabilidade do Hardware com Auxílio de Técnicas de Teste Software*. Tese (Doutorado) — Programa de Pós Graduação em Computação, Porto Alegre, 2007.

[8] ASHENDEN PETER J.; LEWIS, J. *The Designer's Guide to VHDL*. [S.l.]: Elsevier, 2008.

[9] TRAJANO, R. Circuitos combinacionais. *Notas de Aula*, IFPB, ParaÃba.

[10] VEGA, A. S. de la. *Apostila de Teoria para Circuitos Digitais*. Rio de Janeiro: Universidade Federal Fluminense, 2013.

[11] FLOYD, T. L. *Sistemas Digitais*: fundamentos e aplicações. Porto Alegre: Bookman, 2007.

[12] IDOETA IVAN V.; CAPUANO, F. G. *Elementos de Eletrônica Digital*. SÃ£o Paulo: Érica, 2008.

[13] BARTIÉ, A. *Garantia da Qualidade de Software*: Adquirindo maturidade organizacional. Rio de Janeiro: Elsevier, 2002.

[14] NETO, A. C. D. Introdução a teste de software. *Engenharia de Software Magazine*, Rio de Janeiro, p. 54–59, 2007.

[15] NETO JOÃO ROTTA; DOS SANTOS, M. C. N. Teste de software - uma introdução e exemplos. Artigo, 2001.

[16] MAIDASANI, D. *Software Testing*. [S.l.]: Laxmi Publications, 2007. 389 p.

[17] PRESSMAN, R. S. *Software Engineering*: A practitioner's approach. New York: McGraw-Hill, 2005.