ID: .....

Desenvolvimento e Implementação da Unidade de Processamento e dispositivo de Memória Principal

São José dos Campos - Brasil

Maio de 2017

ID: .....

Desenvolvimento e Implementação da Unidade de Processamento e dispositivo de Memória Principal

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

Docente: Prof. Dr. Tiago de Oliveira

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Maio de 2017

Resumo

O objetivo deste relatório é mostrar o planejamento e desenvolvimento de uma Unidade de Processamento funcional em lógica programável, composta de *Program Counter*, Banco de Registradores, Unidade Lógica Aritmética e um dispositivo de Memória Principal, composta de Memória de Instruções e Memória de Dados. A arquitetura MIPS (*Microprocessor without Interlocked Pipeline Stages*) serviu de suporte teórico e inspiração para a composição da arquitetura base idealizada. O Conjunto de Instruções fora definido em um momento anterior, no 1º Ponto de Checagem, e foi adaptada durante este 2º Ponto de Checagem. O projeto final é desenvolver uma CPU (*Central Processor Unit*) funcional em lógica programável que opere em um circuito FPGA (*Field Programable Gate Array*). A próxima etapa será a construção da Unidade de Controle que coordenará a Unidade de Processamento produzida neste PC2.

Palavras-chaves: processador. unidade de processamento. lógica programável. memória principal. MIPS. conjunto de instruções. FPGA.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

Na era das máquinas, o computador (ou os sistemas computacionais em um geral) está presente no dia-dia da sociedade, fazendo falta caso se passe um dia distante do dono. Enquanto seus usuários transitam por amigáveis interfaces, interagindo com textos e caixas de entrada, não sabem a profunda complexidade do que está acontecendo dentro do aparelho. A **CPU** (*Central Processor Unit*, em português, Unidade Central de Processamento) está por trás de todas as operações que os sistemas são submetidos. Sendo representado até como um cérebro em algumas interpretações. É ela que captura os dados, processa-os e imprime o que foi pedido pelo usuário. Com os avanços tecnológicos e progressos nos estudos arquitetura e organização de computadores, o desempenho dos processadores vem evoluindo, encontrando atualmente computadores pessoais com CPUs de acima de 2GHz.

Neste 2º Ponto de Checagem, foi desenvolvida, em lógica programável, a Unidade de Processamento do processador, sendo ela composta por Unidade Lógica Aritmética, *Program Counter*, Banco de Registradores, Somadores e Multiplexadores, além de um dispositivo de memória formada por Memória de Instruções e Memória de Dados. A implementação das componentes foi toda feita em *Verilog*.

Também foi testada a execução de algumas instruções e demonstrada em Forma de Onda, mostrando o funcionamento de todas as partes da Unidade de Processamento projetada.

2 Objetivos

2.1 Geral

O objetivo geral é o desenvolvimento e implementação de um processador com lógica programável, utilizando a linguagem de descrição de hardware *Verilog*, que contenha uma CPU e um dispositivo de Memória Principal. Ao final, o mesmo deve ser funcionar em um dispositivo FPGA.

2.2 Específico

O objetivo deste PC2 é desenvolver e implementar em lógica programável a Unidade de Processamento, incluindo Banco de Registradores, *Program Counter*, somadores e multiplexadores, Unidade Lógica Aritmética. Também deve conter a Memória Principal, composta por Memória de Instruções e Memória de Dados.

3 Fundamentação Teórica

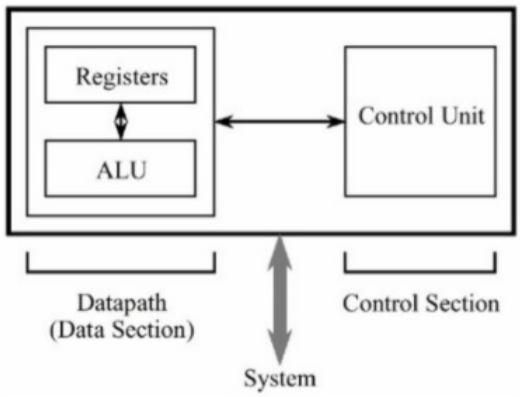
Este tópico expõe o estudo fundamental para a elaboração e efetuação do trabalho, auxiliando o leitor a compreender este relatório em sua totalidade.

Em primeiro momento, discute-se sobre processadores e seus componentes e a memória principal e seus componentes, além dos modos de endereçamento. Posteriormente, uma breve explicação sobre o *clock*, responsável por fazer o processador fluir da maneira desejada, discute-se sobre a arquitetura MIPS, o qual inspirou a escolha da arquitetura, e a linguagem Verilog, ferramenta utilizada para a elaboração dos componentes.

3.1 Processadores

Processadores são os componentes cruciais dos sistemas computacionais. Funcionando como o "cérebro"do computador, é ele que realiza as instruções de um programa, como operações de aritmética, lógica, entrada e saída de dados. Basicamente, sua estrutura é formada por três itens: a Unidade Lógica Aritmética (ou ULA), responsável por efetuar as operações lógicas e aritméticas; a Unidade de Controle, responsável por controlar os pontos de execução e desvios; e os Registradores, que armazenam os dados de processamento (1). A Figura 1 ilustra como estes componentes se interagem na Unidade de Processamento.

Figura 1 – Esquemático de uma Unidade de Processamento



Fonte: Principles of computer architecture (2)

O processador é responsável por executar as operações determinadas por um programador, operações essas que devem pertencer ao conjunto de instruções que a CPU suporta.

3.1.1 Conjunto de Instruções

Conjunto de Instruções são todas as operações que uma CPU admite e consegue executar. A arquitetura escolhida para um processador delimitará o seu conjunto de instruções, podendo ser uma arquitetura do tipo RISC ou CISC.

3.1.1.1 Arquiteturas RISC e CISC

RISC e CISC são tipos de arquiteturas de processadores, tomando uma abordagem de projeto do conjunto de instruções. RISC, ou *Reduced Instruction Set Computer* (Computador de Conjunto de Instruções Reduzidos), é uma arquitetura que visa o menor conjunto de instruções possível. Sua implementação (a nível de *hardware*) tende a ser feita de maneira simples, dado que é pouco o número de instruções. Já a arquitetura CISC, ou *Complex Instruction Set Computer* (Computador de Conjunto de Instruções Complexo) possuem um conjunto de instruções maior (3). Dessa forma, sua organização se constrói de maneira complexa. Porém, para o programador, o desenvolvimento do *software* é mais simples, já que o grande número de instruções abrange vastas funcionalidades.

3.2 *Clocks*

Clocks são sinais usados em circuitos digitais responsáveis por coordenar os componentes de um circuito. Os constituintes de um circuito pode responder ao pulso de *clock* quando este está em alta, ou seja, tem valor 1 e dizemos que ele é "ativo em alta", ou quando é em baixa, sendo de valor 0, e o diz-se que o componente é "ativo em baixa".

3.3 Memória Principal

Memória Principal é a memória fundamental para o computador funcionar. É responsável por guardar informações úteis do processador, como dados e instruções (STTALINGS,2010). Há dois tipos de memória principal, as memórias voláteis, como a Memória RAM (*Random Access Memory*), e as memórias não-voláteis, como as ROM (*Read Only Memory*).

As memórias voláteis são memórias que precisam de energia elétrica, para armazenar dados. Quando há falta de energia, os dados são perdidos. Nos computadores em geral, a memória RAM faz o trabalho de memória principal. As memórias não-voláteis não necessitam de energia para armazenar dados, porém, geralmente armazenam mais dados que a memória volátil.

3.3.1 Modos de Endereçamento

A memória RAM é composta de diversas partes de memória que podem ser acessadas. As maneiras de como se acessam os dados na memória são conhecidas como Modos de Endereçamento.

Discutiremos três tipos de endereçamento: o Endereçamento Imediato, o Endereçamento Direto e o Endereçamento por Registrador. Estes serão os utilizados para o projeto e podem ser ilustradas na Figura 2.

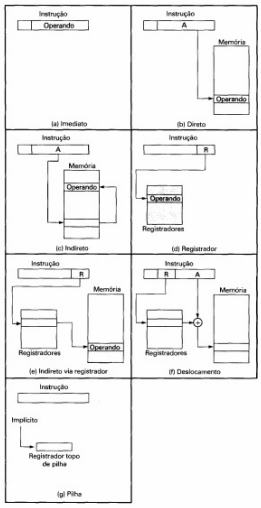
3.3.1.1 Endereçamento Imediato

No modo de Endereçamento Imeditado, o operando (uma constante) faz parte da instrução, sendo especificado justamente no Campo de Endereço. O acesso a memória só ocorre para busca de uma instrução.

3.3.1.2 Endereçamento Direto

No modo de Endereçamento Direto, o endereço desejado é o endereço mencionado no Campo de Endereço. Faz-se referência direta a memória.

Figura 2 – Modos de Endereçamento



Fonte: Arquitetura e Organização de Computadores (1)

3.3.1.3 Endereçamento por Registrador

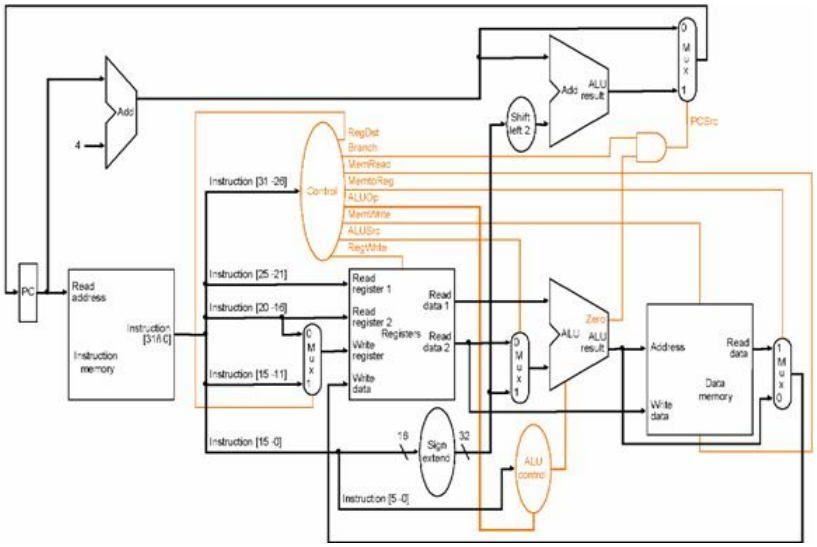
No modo de Endereçamento por Registrador, é referenciado o respectivo registrador no Campo de Endereço e nele contém o operando. Não há acesso a memória.

3.4 Arquitetura MIPS

O MIPS (*Microprocessor without Interlocked Pipeline Stages*) é uma arquitetura RISC elaborada pela MIPS *Computer Systems*. Possui um conjunto de instruções contendo 39 instruções reais e 8 pseudo-instruções (instruções que podem ser traduzidos em instruções reais-múltiplas). A arquitetura foi desenvolvido por uma equipe da Universidade de Berkeley conduzido por David Patterson e Carlos Séquin. (4)

A arquitetura MIPS é estruturada por 5 componentes: *Program Counter*, Banco de Registradores, Unidade Lógica Aritmética e uma Memória Principal, repartida Memória de Instruções e Memória de Dados. A Figura 3 retrata o esquemático da arquitetura MIPS.

Figura 3 – Esquemático da Arquitetura MIPS



Fonte: Organização e Projeto de Computadores (3)

O *Program Counter* é um registrador responsável por indicar qual a próxima execução a ser executada. Após instruções que não indicam um pulo de endereço, como instruções de "*jump*", o registrador é simplesmente incrementado.

O Banco de Registradores é responsável por armazenar os dados que a CPU irá processar. Estes dados são lidos na memória principal, na parte da memória de dados. É composto por 32 registradores de 32 bits.

Unidade Lógica Aritmética é o componente responsável por fazer as operações lógicas, como AND e OR, e aritméticas, como soma e subtração. Os dados são recebidos do Banco de Registradores e seus resultados são armazenados na Memória de Dados.

A Memória de Instruções é a seção da Memória Principal responsável por armazenar as instruções recebidas. Recebe do *Program Counter* o endereço da próxima instrução a ser executada, através de um código binário.

A Memória de Dados é a outra parte da Memória Principal que armazena os dados processados pela CPU.

A arquitetura MIPS monociclo (utilizada no projeto) faz com que o processador realize uma instrução por ciclo de *clock*.

3.5 Verilog

Verilog é uma linguagem de descrição de *hardware* (HDL), utilizada para projetar sistemas eletrônicos. Seu objetivo é especificar a descrição do comportamento de um circuito que eventualmente será implementado em *hardware* (5).

A seguir está a sintaxe básica de *Verilog*, para melhor compreensão do leitor aos códigos que serão vistos posteriormente.

- **Tipos de portas:** São meios fundamentais de comunicação com um módulo.
  - ***input*:** Porta de entrada;
  - ***output*:** Porta de saída;
  - ***inout*:** Porta bidirecional
- **Operadores aritméticos:** O *Verilog* trata de operações como soma, subtração, multiplicação e divisão. Os valores negativos são armazenado na configuração de Complemento de 2, ou seja, o complemento de bits em relação a 2^n.
  - **'+' :** Executa a operação de soma;
  - **'-' :** Executa a operação de subtração, negação.
  - **'*' :** Executa a operação de multiplicação;
  - **'/' :** Executa a operação de divisão;
- **Operadores relacionais:** Utilizados para comparar valores.
  - **'>' :** Compara se um valor é maior que o outro;
  - **'<' :** Compara se um valor é menor que o outro;
  - **'>=' :** Compara se um valor é maior ou igual a outro;
  - **'<=' :** Compara se um valor é menor ou igual a outro
- **Operadores de igualdade:** Também utilizados para comparar valores
  - **'==' :** Confere igualdade entre valores;
  - **'!=' :** Confere diferença entre valores

- **Tipos de dados *registers* e *wires*:** Fiações ou nós.
  - ***reg*:** Variável do tipo de dados *register*. São drivers que guardam valores.
  - ***wire*:** Variável do tipo de dados *Net*. São drivers que apenas conectam dois pontos

4 Desenvolvimento

O processador foi baseado na arquitetura MIPS monociclo, em que cada instrução só pode ser executada em um ciclo de *clock*. Sua unidade de processamento contém como mencionado um **Banco de Registradores**, para armazenar os dados a processar, uma **Unidade Lógica Aritmética**, para realizar as operações, dois **somadores** e quatro **multiplexadores**, para determinados usos, além de um ***Program Counter***, para definir o endereço da instrução a ser executada. A memória principal contém uma **Memória de Instrução**, que armazena as instruções a serem executadas, e uma **Memória de Dados**, para armazenar as informações definidas.

O Conjunto de Instruções fora definido no 1º Ponto de Checagem, que neste relatório será demonstrado o mapeamento destas instruções.

Os códigos em Verilog podem ser vistos no subseção 4.2.1, presente no final deste relatório.

4.1 Mapeamento das Instruções e Modos de Endereçamento

Cada instrução apresenta um ***OPCODE*** (*Operation Code*, ou em português Código de Operação). Esse *opcode* será adquirido pela Unidade de Controle (ainda não projetada) e organizará a arquitetura do processador com seus sinais de controle para que a mesma funcione como desejado. Além disso, também foi determinada o modo de endereçamento de cada instrução. As instruções, os *opcodes* e os modos de endereçamento podem ser vistos na Tabela 1.

4.2 Unidade de Processamento

A arquitetura base já foi definida no 1º Ponto de Checagem e pode ser vista na Figura 4:

Para este Ponto de Checagem, apenas a Unidade de Controle não foi implementada. Para os testes, os sinais de controle foram indicados manualmente direto nos componentes. Todos os componentes foram implementados em *Verilog*.

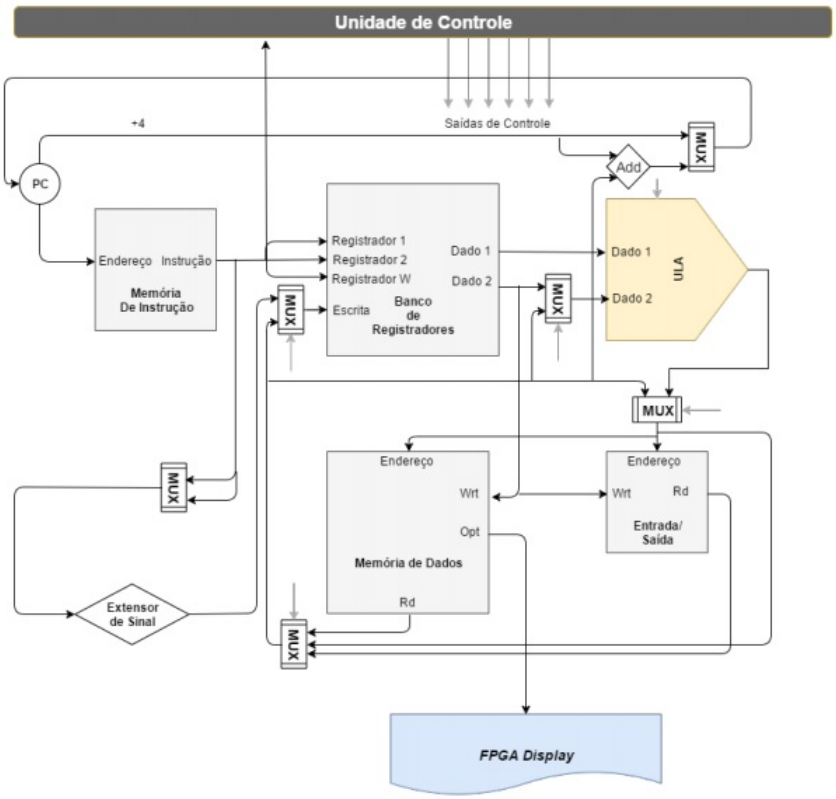
4.2.1 *Program Counter*

O ***Program Counter*** é responsável por indicar a Memória de Instrução qual a próxima instrução a ser executada. Nesta arquitetura ela é composta por apenas um

Tabela 1 – Mapeamento das Instruções

Instrução	OpCode	Modo de Endereçamento
add	100000	Por Registrador
addi	010000	Imediato
sub	100001	Por Registrador
subi	010001	Imediato
and	100010	Por Registrador
or	100011	Por Registrador
blank	100010	Por Registrador
neg	011000	Por Registrador
slt	100100	Por Registrador
slti	010100	Imediato
beq	010110	Relativo ao PC
bnq	010111	Relativo ao PC
sw	011110	Direto
lw	011111	Direto
j	111111	Absoluto
reset	000000	Não é necessário
in	000111	Não definido
out	111000	Não definido

Figura 4 – Esquematico



registrador de 8 bits, o qual indica o endereço da próxima instrução. O código do *Program Counter* pode ser visto logo abaixo.

```
module programCounter(proximo_endereco, endereco_atual, clock);

        // input
        input [31:0] proximo_endereco;
        input clock;

        // output
        output reg [31:0] endereco_atual;

        // registradores
        //reg [31:0] registrador_pc;

        integer inicializa_pc = 1;

        always @ (posedge clock) begin

                if(inicializa_pc == 1) begin
                        endereco_atual = 32'b0;
                        inicializa_pc = 0;
                end
                else begin
                        endereco_atual = proximo_endereco;
                end
        end
endmodule
```

A variável inteira "inicializa pc"tem função de inicializar o PC para que, no primeiro ciclo de *clock*, o endereço de saída seja o endereço binário 0000 0000. Nos próximos ciclos de *clock* a saída será o endereço recebido por "proximo endereco". No esquemático, o PC pode ser visto como "**programCounter**".

4.2.2 Banco de Registradores

O **Banco de Registradores** foi desenvolvido para reduzir os acessos na Memória de Dados e aprimorar o desempenho do processador. Ele é constituído de 32 registradores de 32 bits que armazenam os dados que a CPU irá processar. O código do Banco de Registradores pode ser visto a seguir.

```
module bancoRegistradores (endereco_read1, endereco_read2, endereco_wrt, dado_wrt,
    saida_RS, saida_RT, controle /* Controle de Escrita */, clock);

        // input
        input [4:0] endereco_read1;
        input [4:0] endereco_read2;
        input [4:0] endereco_wrt;
        input [31:0] dado_wrt;
        input clock, controle;

        // registradores
        reg [31:0] registradores_banco[31:0];

        // output
        output [31:0] saida_RS;
        output [31:0] saida_RT;

```

```
17          // inicializador
18          integer inicia_banco = 1;
19
20          always @ (posedge clock) begin
21                  if(controle == 1) begin // Controle de Escrita
22
23                          registradores_banco[endereco_wrt] = dado_wrt;
24                  end
25          end
26
27          // Resultados Saidas
28          assign saida_RS = registradores_banco[endereco_read1];
29          assign saida_RT = registradores_banco[endereco_read2];
30  endmodule
```

O Banco de Registradores recebe os campos da instrução destinados ao endereçamento dos registradores. As entradas "endereco read1"e "endereco read"referem-se aos campos RS e RT, respectivamente, enquanto "endereco wrt"refere-se ao campo RD. Ele é orientado por um sinal de controle de escrita recebido da Unidade de Controle. Caso este sinal seja igual ao valor binário 1, deve-se escrever os dados recebidos em "dado wrt", se for 0, apenas ler os registradores dos endereços delimitados.

4.2.3 Unidade Lógica Aritmética

A **Unidade Lógica Aritmética** é responsável por fazer as operações de lógico-aritméticas. Um sinal de controle é recebido pela ULA que determinará qual operação deve ser feita. Os sinais de controle e suas respectivas operações podem ser vistas na Tabela 2. Logo abaixo, o código da Unidade Lógica Aritmética é apresentado.

Tabela 2 – Sinais de Controle para as Operações da ALU

Controle	**Operação**	**Descrição**
000	SOMA	dado1 + dado2
001	SUBT	dado1 - dado2
010	AND	dado1 & dado2
011	OR	dado1 | dado2
100	SHIFT LEFT	dado1 «1
101	SHIFT RIGHT	dado1 »1
110	BLANK	dado1
111	NEG	∼dado1

```
1   module ALU (dado1, dado2, saida, controle /*ALU Control*/, sinal_ZERO, sinal_NEG);
2
3           // input
4           input [31:0] dado1;
5           input [31:0] dado2;
6           input [2:0] controle;
7
8           // output
9           output reg [31:0] saida;
```

```
10          output sinal_ZERO, sinal_NEG;
11
12                  always @ (*) begin
13
14                  case(controle[2:0])
15
16                          3'b000: saida = dado1 + dado2;
17                          3'b001: saida = dado1 - dado2;
18                          3'b010: saida = dado1 & dado2;
19                          3'b011: saida = dado1 | dado2;
20                          3'b100: saida = dado1 << 1;
21                          3'b101: saida = dado1 >> 1;
22                          3'b110: saida = dado1;
23                          3'b111: saida = ~dado1;
24                  endcase
25          end
26
27          assign sinal_ZERO = (saida == 0);
28          assign sinal_NEG = (($signed(saida) < 0));
29  endmodule
```

A Unidade Lógica Aritmética recebe os dados do Banco de Registradores e com eles são feitos as operações determinadas pelo sinal de controle. Além de emitir o resultado da operação, a ULA deve emitir como saída dois tipos de sinais: quando o resultado é igual a 0 e quando o resultado é negativo. Essas condições são importantes para instruções do tipo *branch*.

4.2.4 Multiplexadores

Foram desenvolvidos 4 multiplexadores, posicionados na arquitetura para escolher uma dentre várias entradas. As entradas são escolhidas a partir do sinal de controle recebido pelo componente. A tabela 3 indica as entradas e as respectivas saídas para cada sinal de controle em "**MUX 1**", "**MUX 2**"e "**MUX 3**.

Tabela 3 – Saídas dos Multiplexadores

MUX	**Controle = 0**	**Controle = 1**
MUX_1	Instrução [20-16]	Instrução [15-11]
MUX_2	Dado2	Instrução [15-0] Extendido
MUX_3	Resultado ALU	Saida Memória de Dados

O "**MUX 4**"é responsável por indicar ao *Program Counter* qual o endereço da próxima instrução a ser executada. O sinal de controle é maior do que dos outros multiplexadores, sendo composto por 4 valores. Os sinais de controle indicam quando é para apenas incrementar o endereço do PC (sinal de controle = 00), quando for uma instrução do tipo *branch on equal* (sinal de controle = 01), quando for uma instrução do tipo *branch on not equal* (sinal de controle = 10), e quando for uma instrução do tipo *jump* (sinal de controle = 11).

Os códigos em Verilog dos multiplexadores podem ser encontrados no Apêndice A.

4.2.5 Extensor de Sinal

As unidades do processador trabalham apenas com dados de 32 bits, devido sua arquitetura. A função do Extensor de Sinal é converter dados com menos de 32 bits em 32 bits. No caso, o **Extensor de Sinal** será utilizado para converter dados de 16 bits, campo endereço de instruções do tipo *branch*. No projeto, o extensor de sinal pode ser encontrado como: "**extensor16 32**".

Há também um Extensor de Sinal especial que será utilizado nas instruções do tipo *jump*. Ele extenderá um dado de 26 bits em 28 bits, fazendo um deslocamento de 2 bits para a esquerda, concatentado dois valores 0 em seus bits menos significativos.

Os códigos em Verilog dos Extensores de Sinal podem ser encontrados no Apêndice A.

4.3 Memória Principal

A memória principal foi dividida em duas partes: Em Memória de Instrução (ou Memória de Programa) e a Memória de Dados. A primeira é responsável por armazenar as instruções a serem executadas, enquanto a última é encarregada de armazenar os dados das operações da unidade de processamento.

4.3.1 Memória de Instrução

A **Memória de Instrução** foi desenvolvida para este processador com 100 registradores de 32 bits. Cada registrador é responsável por armazenar uma instrução que será executada. Ela recebe do *Program Counter* o endereço (em valor binário) da próxima instrução a executar e sua saída é a instrução a ser executada. Não há orientação por sinal de controle. O código da Memória de Instrução está a seguir.

```
1  module memoriaInstrucao(endereco, instrucao, clock);
2
3          // input
4          input [31:0] endereco;
5          input clock;
6
7          // registradores
8          reg [31:0] registradores_instrucao[256:0];
9
10         // output
11         output [31:0] instrucao;
12
13         // inicializador
14         integer inicializa_memoria = 1;
15
```

```
16          always @ (posedge clock) begin
17
18
19                  if(inicializa_memoria == 1) begin
20
21                          inicializa_memoria = 0;
22                  end
23          end
24
25          assign instrucao = registradores_instrucao [endereco];
26  endmodule
```

A variável inteira "inicializa memoria"é responsável por inserir as instruções nos registradores no primeiro ciclo de *clock* e os próximos ciclos são responsáveis pelo envio das instruções. No esquemático, a memória de instruções pode ser vista como "**memoriaInstrucao**".

4.3.2 Memória de Dados

A **Memória de Dados** foi desenvolvida para este processador com 250 registradores de 32 bits. Cada registrador é responsável por guardar os resultados feitos pela unidade de processamento. Ela é orientada por um sinal de controle (oriundo da Unidade de Controle) que indica se é para escrever ou ler um dado. Este componente recebe um endereço (indicado pela instrução executada), um dado para escrever, e o sinal de controle. Caso o sinal recebido for igual a '1', deve escrever o dado recebido; caso for igual a '0', apenas ler o registrador do endereço recebido. O código da Memória de Dados está a seguir.

```
1   module memoriaDados (endereco, dado_wrt, saida_dado, controle /* Controle de Escrita e
        Leitura */, clock);
2
3           // input
4           input [31:0] endereco;
5           input [31:0] dado_wrt;
6           input controle, clock;
7
8           // registradores
9           reg [31:0] registradores_dados[100:0];
10
11          // output
12          output [31:0] saida_dado;
13          integer inicializa_memD = 1;
14
15          always @ (posedge clock) begin
16
17                  if(controle == 1) begin // Controle = 1 => Escreve <<>> Controle = 0 =>
                        Le, apenas
18                          registradores_dados[endereco] = dado_wrt;
19                  end
20          end
21
22          assign saida_dado = registradores_dados[endereco];
23  endmodule
```

A saída de dados é dada de forma contínua, ou seja, em todo ciclo de clock, emitirá a informação contida no registrador do endereço considerado. No esquemático, a memória de dados pode ser vista como "**memoriaDados**".

5 Resultados Obtidos e Discussões

Neste capítulo será demonstrado os resultados obtidos da aplicação de diversos testes na Unidade de Processamento construída. Os componentes serão testados de maneira isolada, verificando se suas operações estão funcinando corretamente, e conectados entre si, executando uma soma de dois números (instrução *add*).

5.1 Testes com Componentes Isolados

Nesta seção, os componentes da Unidade de Processamento foram testados isoladamente, ou seja, não estão se intercomunicando.

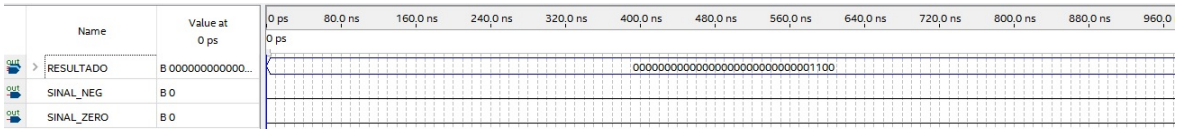
5.1.1 Teste da Unidade Lógica Aritmética

A ULA foi testada para verificar dois de seus tipos de operações: **SOMA** e **AND**.

Para a operação de **SOMA**, foi imposta a seus *inputs* dois dados de 32 bits com valores binários 00000000000000000000000000000101 (5, em decimal) e 00000000000000000000000000000111 (7, em decimal) e um sinal de controle de valor binário 000, que é responsável por fazer a operação de adição desses dois dados e emitir o resultado. O código Verilog pode ser visto abaixo. Para verificar suas saídas, foram instanciadas variáveis *output* para verificar o resultado, o valor do sinal de ZERO e NEGATIVO emitidos pela ULA. A forma de onda pode ser vista na Figura 5

```
1  module teste_ALU(RESULTADO, SINAL_ZERO, SINAL_NEG);
2
3          output [31:0] RESULTADO;
4          output SINAL_ZERO, SINAL_NEG;
5
6          ALU ALU(32'b00000000000000000000000000000101, // dado 1 = 5
7                              32'b00000000000000000000000000000111, // dado 2 = 7
8                              RESULTADO,
9                              3'b010, // sinal de controle para AND
10                             SINAL_ZERO,
11                             SINAL_NEG);
12
13 endmodule
```

Figura 5 – Forma de Onda da ULA para a operação de SOMA

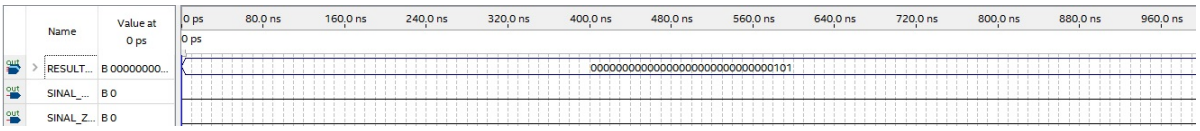


Como pode se ver na Figura 5 o *output* do resultado foi o valor binário 00000000000000000000000000001100, o qual é 12 em decimal, sendo o resultado esperado para soma de 5 e 7. O sinal de ZERO e NEGATIVO foram ambos de valor binário 0, como desejado.

Para a operação de **AND**, foram impostas as mesmas entrada de dados que para a operação de SOMA, variando apenas o sinal de controle que terá valor binário 010. O código está descrito abaixo e a Figura 6 é a forma de onda simulada.

```
1  module teste_ALU_and(RESULTADO, SINAL_ZERO, SINAL_NEG);
2
3          output [31:0] RESULTADO;
4          output SINAL_ZERO, SINAL_NEG;
5
6          ALU ALU(32'b00000000000000000000000000000101, // dado 1 = 5
7                              32'b00000000000000000000000000000111, // dado 2 = 7
8                              RESULTADO,
9                              3'b010, // sinal de controle para AND
10                             SINAL_ZERO,
11                             SINAL_NEG);
12
13 endmodule
```

Figura 6 – Forma de Onda da ULA para operação de AND



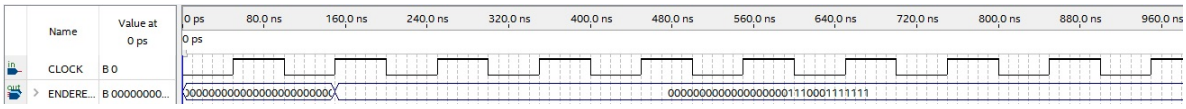
Para entradas de valores binários 00000000000000000000000000000101 e 00000000000000000000000000000111, o resultado 00000000000000000000000000000101 é o esperado. Os sinais ZERO e NEGATIVO são de valor 0, como na operação de SOMA e dentro da expectativa.

5.1.2 Teste do *Program Counter*

O teste do *Program Counter* consistiu em impôr um novo endereço ao componente que deveria fornecer este endereço. O código do teste e a forma de onda podem ser vistos abaixo (Figura 7).

```
1  module teste_PC(ENDERECO_ATUAL, CLOCK);
2
3          output [31:0] ENDERECO_ATUAL;
4          input CLOCK;
5
6          programCounter programCounter(
7                                                  32'b00000000000000000001110001111111,//
                                                       proximo endereco
8                                                  ENDERECO_ATUAL, // endereco atual
9                                                  CLOCK);
10
11 endmodule
```

Figura 7 – Forma de Onda do PC



Na Figura 7 pode se ver que no primeiro ciclo de *clock* o endereço de saída é todo 0, o que é esperado, já que o primero ciclo é usado para inicializar o componente. Para um endereço de entrada (próximo endereço) 00000000000000000001110001111111, teve que o endereço de saída era o mesmo, como desejado, já que o próximo endereço é calculado antes de entrar no PC, que só irá indicar qual a próxima instrução na Memória de Instrução.

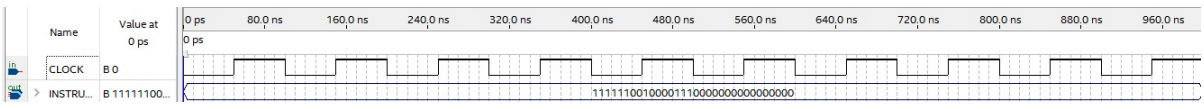
5.1.3 Teste da Memória de Instrução

Um endereço foi passado para a Memória de Instrução, o objetivo era que o componente emitisse uma saída com a instrução alocada no registrador do determinado endereço. Dentro da Memória de Instrução, uma instrução foi alocada a priori, para a realização do teste. Os código do teste pode ser visto abaixo, a Figura 14 é a parte do código da Memória de Instrução que indica qual instrução em qual registrador está alocado os valores de teste e a Figura 8 mostra a simulação em Forma de Onda.

```
1  module teste_MEMI(INSTRUCAO, CLOCK);
2
3          output [31:0] INSTRUCAO;
4          input CLOCK;
5
6          memoriaInstrucao memoriaInstrucao(
7                                  32'b00000000000000000000000000001100,
8                                  INSTRUCAO,
9                                  CLOCK);
10
11 endmodule
```

O registrador 12 da memória de instrução recebe a instrução 11111100100001110000000000000000 quando a memória é inicializada no primeiro ciclo de *clock*. O endereço passado como parâmetro para a Memória de Instrução é 00000000000000000000000000001100 (em decimal, 12).

Figura 8 – Forma de Onda da Memória de Instrução



Pela Figura 8, a Memória de Instrução está respondendo corretamente, emitindo como saída a instrução 11111100100001110000000000000000, que está alocada no registrador do endereço passado como parâmetro. Como a função da Memória de Instrução é apenas armazenar e retornar as instruções em registradores, não foram necessários mais testes.

5.1.4 Teste do Banco de Registradores

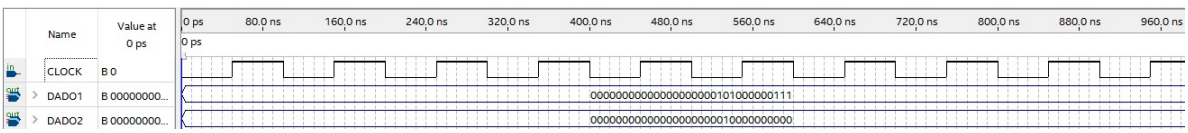
O teste submetido ao Banco de Registradores foram dois: um para apenas leitura de dados e outro para escrita. Os códigos de ambos testes estão logo abaixo e as Figuras 9 e 10 são suas respectivas simulações em Forma de Onda.

O Banco de Registradores recebeu três endereços, RS, RT e RD, que indicam em qual registradores cada um irá acessar. O endereço RS passado foi 00100 (4, em decimal) e RT foi 00111 (7, em decimal). No Banco de Registradores, os registradores 4 e 7 irão receber os valores 00000000000000000000101000000111 e 00000000000000000000010000000000, para serem usados como saída do teste. Como o objetivo desta parte é apenas leitura, o sinal de controle indicado foi 0.

```
1  module teste_BR_read(DADO1, DADO2, CLOCK);
2
3          output [31:0] DADO1;
4          output [31:0] DADO2;
5          input CLOCK;
6
7          bancoRegistradores bancoRegistradores(
8                                                  5'b00100,
9                                                  5'b00111,
10                                                 5'b01111,
11                                               32'b00000000000000000000000000000001,
12                                                 DADO1,
13                                               DADO2,
14                                                 1'b0,
15                                                 CLOCK);
16
17 endmodule
```

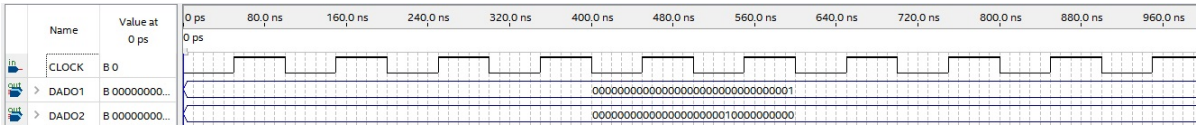
```
1  module teste_BR_wrt(DADO1, DADO2, CLOCK);
2
3          output [31:0] DADO1;
4          output [31:0] DADO2;
5          input CLOCK;
6
7          bancoRegistradores bancoRegistradores(
8                                           5'b01111,
9                                           5'b00111,
10                                  5'b01111,
11                                          32'b00000000000000000000000000000001,
12                                          DADO1,
13                                          DADO2,
14                                          1'b1,
15                                          CLOCK);
16
17 endmodule
```

Figura 9 – Forma de Onda da Leitura no Banco de Registradores



Vendo a Forma de Onda simulada para quando só há leitura, vê-se que as saídas são os valores 00000000000000000000101000000111 e 00000000000000000000010000000000, que estão alocados nos registradores 4 e 7, respectivamente, portanto, dentro do esperado, para uma simples leitura.

Figura 10 – Forma de Onda da Escrita no Banco de Registradores



Quando há operação de Escrita, o endereço RS será agora 00111 e o sinal de controle será 1. As saídas deve ser o dado 00000000000000000000000000000001 que será inscrito nesse endereço e o valor alocado no registrador RT. Vendo a simulação de onda para este caso, Figura 10, o esperado foi ocorrido.

5.1.5 Teste da Memória de Dados

A Memória de Dados teve as duas mesmas operações que o Banco de Registradores, um teste par apenas leitura da memória e outra para escrita. A instanciação do teste foram tiveram o mesmo endereço para leitura e para escrita, o sinal de controle de entrada, responsável pelo controle de escrita é, no primeiro, valor 0 e no outro, 1. Os códigos podem ser vistos abaixo.

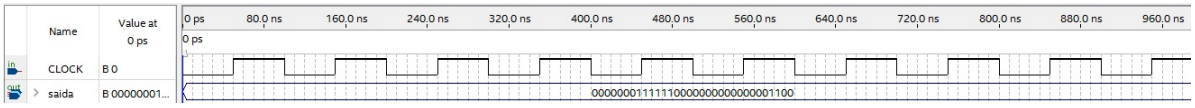
```
1  module teste_MEMD_read(saida, CLOCK);
2
3          output [31:0] saida;
4          input CLOCK;
5
6          memoriaDados memoriaDados(
7                          32'b00000000000000000000000000110010,
8                          32'b00000001111111000000000000001100,
9                          saida,
10                         1'b0,
11                         CLOCK);
12
13 endmodule
```

```
1  module teste_MEMD_wrt(saida, CLOCK);
2
3          output [31:0] saida;
4          input CLOCK;
5
6          memoriaDados memoriaDados(
7                          32'b00000000000000000000000000110010,
8                          32'b11110001111110011101100000111111,
9                          saida,
10                         1'b1,
11                         CLOCK);
12
13 endmodule
```

A Memória de Dados é inicializada com o registrador 50 recebendo o valor de teste 00000001111110000000000000001100, como pode ser visto na Figura 16.

É passado como parâmetro o endereço 00000000000000000000000000110010 (50, em decimal) para acesso a memória e um dado, para escrita, de 00000001111110000000000000001100. A simulação em Forma de Onda para apenas a operação de leitura pode ser vista na Figura 11, abaixo.

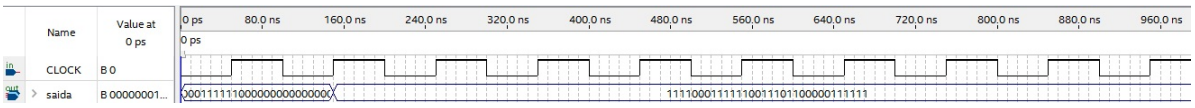
Figura 11 – Forma de Onda da Leitura na Memória de Dados



O dado contido no endereço 00000000000000000000000000110010 foi concedido na saída, como o esperado para um acesso a memória sem escrita.

No caso em que há acesso a memória para escrita, o dado a se escrever é 11110001111110011101100000111111 que será escrito no endereço 00000000000000000000000000110010. A simulação em Forma de Onda pode ser visto na Figura 12.

Figura 12 – Forma de Onda da Escrita na Memória de Dados



Vê-se na Forma de Onda que a saída é o valor 11110001111110011101100000111111. Porém, o valor iniciado na memória de dados neste endereço é o valor 00000000000000000000000000110010, ou seja, o processo de escrita está ocorredo no registrador está acontecendo, que é o desejado.

5.2 Teste com os Componentes Conectados

Foi feito um teste com os componentes da Unidade de Processamento e o dispositivo de Memória Principal conectados. Se utilizou uma instrução do tipo *add*, para somar dois números. Os sinais de controle foram indicados manualmente.

O código do teste com os componentes conectados pode ser visto abaixo:

```
1  module normips(clock,
2                                          RESULTADO_ALU,
3                                          DADO_RS,
4                                          DADO_RT,
5                                          PROXIMA_INSTRUCAO,
6                                          INSTRUCAO_EXECUTADA);
7
8
9          wire [31:0] MUX4_PC;
```

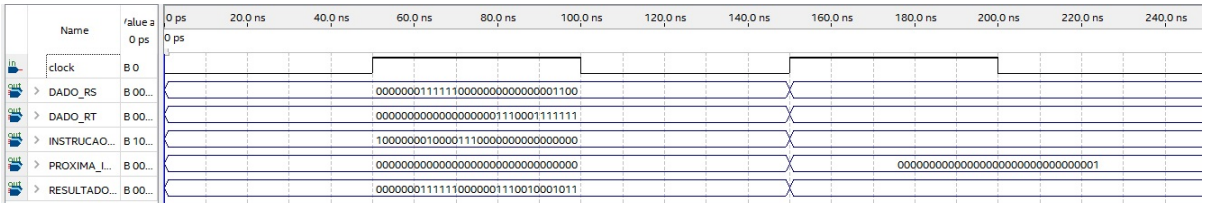
```
10          input clock;
11          wire [31:0] END_ATUAL;
12          wire [31:0] INSTRUCAO;
13          wire [4:0] MUX1_BR;
14          wire [31:0] MUX3_BR;
15          wire [31:0] dadoRS;
16          wire [31:0] dadoRT;
17          wire [31:0] dadoRD;
18          wire [31:0] EXTENDIDO;
19          wire [31:0] MUX2_ALU;
20          // wire [31:0] RESULTADO;
21          wire [31:0] sinal_ALU_ZERO;
22          wire [31:0] sinal_ALU_NEG;
23          wire [31:0] MEMD_MUX3;
24          wire [31:0] EXTENDIDO_DESLOCADO;
25          wire [27:0] END_JUMP;
26          wire[31:0] RESULTADO;
27
28          output [31:0] RESULTADO_ALU;
29          output [31:0] DADO_RS;
30          output [31:0] DADO_RT;
31          output [31:0] PROXIMA_INSTRUCAO;
32          output [31:0] INSTRUCAO_EXECUTADA;
33
34          programCounter programCounter(MUX4_PC,
35                                                                  END_ATUAL
                                                                        ,
36                                                                  clock
                                                                        )
                                                                        ;
37
38
39          memoriaInstrucao memoriaInstrucao(END_ATUAL,
40                                                                        INST
41                                                                        cloc
42
43
44          MUX_1 MUX_1(INSTRUCAO[20:16],
45                                      INSTRUCAO[15:11],
46                                      MUX1_BR,
47                                      /*controle_MUX1*/1'b0);
48
49          bancoRegistradores bancoRegistradores(
50                      INSTRUCAO[25:21],
51                      INSTRUCAO[20:16],
52                      MUX1_BR,
53                      MUX3_BR,
54                      dadoRS,
55                      dadoRT,
56                  /*controle_BR*/1'b0,
57                      clock);
58
59          extensor16_32 extensor16_32(INSTRUCAO[15:0],
```

```
60                                                                          EXTENDIDO
                                                                                );
61
62          deslocamento_2 deslocamento_2(EXTENDIDO,
63                                                                                  EXTENDIDO_DESLOCAD
                                                                                        )
                                                                                        ;
64
65          MUX_2 MUX_2(dadoRT,
66                                      EXTENDIDO,
67                                      MUX2_ALU,
68                                      /*controle_MUX2*/1'b0);
69          ALU ALU(dadoRS,
70                          MUX2_ALU,
71                          RESULTADO,
72                          /*controle_ALU*/3'b000,
73                          sinal_ALU_ZERO,
74                          sinal_ALU_NEG);
75
76          memoriaDados memoriaDados(RESULTADO,
77                                                                  dadoRT,
78                                                                  MEMD_MUX3,
79                                                                  /*controle_MEMD
                                                                        */1'b0,
80                                                                  clock);
81
82          MUX_3 MUX_3(MEMD_MUX3,
83                                      RESULTADO,
84                                      MUX3_BR,
85                                      /*controle_MUX3*/1'b0);
86
87          extensor26_28 extensor26_28(INSTRUCAO[25:0],
88                                                                          END_JUMP
                                                                                );
89
90          MUX_4 MUX_4(END_ATUAL,
91                                      EXTENDIDO_DESLOCADO,
92                                      END_JUMP,
93                                      MUX4_PC,
94                                      /*controle_MUX4*/2'b00,
95                                      sinal_ALU_ZERO);
96
97          assign RESULTADO_ALU = RESULTADO;
98          assign DADO_RS = dadoRS;
99          assign DADO_RT = dadoRT;
100         assign PROXIMA_INSTRUCAO = END_ATUAL;
101         assign INSTRUCAO_EXECUTADA = INSTRUCAO;
102
103 endmodule
```

Os sinais de controle de cada componente estão indicados no código. A simulação de forma de onda pode ser visto na Figura 13.

Os *outputs* foram instanciados através das saídas de cada componente. DADO RS e DADO RT são os registradores do Banco de Registradores, RESULTADO ALU é o resultado da operação da ULA, INSTRUCAO é a instrução executada da Memória de

Figura 13 – Forma de Onda de uma instrução add



Instrução e PROXIMA INSTRUCAO é o endereço da proxima instrução indicada pelo PC.

Vê-se somando DADO RT e DADO RS é igual ao RESULTADO ALU, por isso, dá pra se concluir que o Banco de Registradores e a ULA estão se comunicando corretamente. A PROXIMA INSTRUCAO, no primeiro ciclo de *clock* indica o endereço 0 e no próximo ciclo indicou o endereço 1, portanto, o MUX 4 fez a operação de PC+1, que é o esperado para uma instrução add.

6 Considerações Finais

O objetivo da construção de uma Unidade de Processamento com um dispositivo de Memória Principal foi alcançado. Os componentes foram mostrados funcionando para operações que possam ocorrer durante a execução de operações e quando foram conectados, todos funcionaram de maneira correta.

A maneira de indicar os sinais de controle manualmente fez com que o as operações ocorressem como se esperava, porém, testar diferentes tipos de instrução em uma mesma compilação se torna algo trabalhoso. Isso ocorre pois a parte responsável por coordenar os diferentes sinais de controle para cada instrução é a Unidade de Controle. Além do mais, fazer os testes diversas vezes, faz com que a construção da Unidade de Controle se torne algo mais simples, pois já se sabe quais valores de sinais de controle devem sair e em quais momentos.

Com a Unidade de Processamento já pronta e o Conjunto de Instruções mapeado, o desenvolvimento da Unidade de Controle (objetivo do 3º Ponto de Checagem) não parece vislumbrar algo de trabalho complicado, já que os sinais de controle estavam sendo orientados de maneira manual. Porém, juntar todos os componentes do processador e fazê-lo com que seja funcional será ser o trabalho mais complicado.

Referências

1 STTALINGS, W. *Arquitetura e Organização de Computadores*. 5th edition. ed. Waltham/MA, EUA: Pearson, 2003. Citado 2 vezes nas páginas 11 e 13.

2 FARAHAT, A. *Datapath and control: basics of the microarchitecture*. 2015. Disponível em: <http://8051-microcontrollers.blogspot.com.br/2015/01/datapath-and-control-basic-s-of.html#.WQ-PKYjyu00>. Citado na página 11.

3 PATTERSON, D. A.; HENNESY, J. L. *Organização e Projeto de Computadores*. 3th edition. ed. Waltham/MA, EUA: Elsevier, 2005. Citado 2 vezes nas páginas 12 e 14.

4 ARQUITETURA MIPS. 2008. Disponível em: <https://pt.wikipedia.org/wiki/Arquitetura_MIPS>. Citado na página 14.

5 CORPORATION, A. *Verilog HDL Basics*. [S.l.]: Altera, 2011. Citado na página 15.

Apêndices

APÊNDICE A – Códigos dos Componentes Auxiliares em Verilog

```
module MUX_1 (entrada_RT, entrada_RD, saida, controle /*Controle MUX1*/);

        // input
        input [4:0] entrada_RT;
        input [4:0] entrada_RD;
        input controle;

        // output
        output reg [31:0] saida;

        always @ (*) begin

                if(controle == 0) begin

                        saida = entrada_RT;
                end

                if(controle == 1) begin

                        saida = entrada_RD;
                end
        end
endmodule
```

```
module MUX_2 (entrada_dadoRT, entrada_extendido, saida, controle);

        // input
        input [31:0] entrada_dadoRT;
        input [31:0] entrada_extendido;
        input controle;

        // output
        output reg [31:0] saida;

        always @ (*) begin

                if(controle == 0) begin

                        saida = entrada_dadoRT;
                end

                if(controle == 1) begin

                        saida = entrada_extendido;
                end
        end
endmodule
```

```
module MUX_3 (entrada_memoria, resultado_alu, saida, controle);

        // input
```

```
4         input [31:0] entrada_memoria;
5         input [31:0] resultado_alu;
6         input controle;
7
8         // output
9         output reg [31:0] saida;
10
11        always @ (*) begin
12
13                if(controle == 1) begin
14
15                        saida = entrada_memoria;
16                end
17
18                if(controle == 0) begin
19
20                        saida = resultado_alu;
21                end
22        end
23 endmodule
```

```
1  module MUX_4 (entrada_pc, entrada_extendido_deslocado, entrada_jump, saida, controle,
       sinal_ZERO);
2
3         // input
4         input [31:0] entrada_pc;
5         input [31:0] entrada_extendido_deslocado;
6         input [27:0] entrada_jump;
7         input [1:0] controle;
8         input sinal_ZERO;
9
10        // output
11        output reg [31:0] saida;
12
13        always @ (*) begin
14
15                case(controle[1:0])
16
17                        2'b00: saida = entrada_pc + 1; // PC + 1
18                        2'b01: begin // BRANCH EQUAL
19                                if(sinal_ZERO == 1) begin
20                                        saida = entrada_pc + entrada_extendido_deslocado;
21                                end
22                        end
23                        2'b10: begin // BRANCH NOT EQUAL
24                                if(sinal_ZERO == 0) begin
25                                        saida = entrada_pc + entrada_extendido_deslocado;
26                                end
27                        end
28                        2'b11: begin // JUMP
29                                saida = {entrada_pc[31:28], entrada_jump};
30                        end
31                endcase
32        end
33 endmodule
```

```
1  module extensor16_32 (entrada, saida);
2
3         // input
4         input [15:0] entrada;
```

```
5
6           // output
7           output [31:0] saida;
8
9           assign saida = {16'b0, entrada};
10  endmodule
```

```
1   module extensor26_28 (entrada, saida);
2
3           // input
4           input [25:0] entrada;
5
6           // output
7           output [27:0] saida;
8
9           assign saida = {entrada, 2'b0};
10  endmodule
```

```
1   module deslocamento_2 (entrada, saida);
2
3           // input
4           input [31:0] entrada;
5
6           // output
7           output [31:0] saida;
8
9           assign saida = entrada << 2;
10  endmodule
```

APÊNDICE B – Códigos Auxiliares da Fase de Testes

Figura 14 – Parte do código da Memória de Instrução

```
always @ (posedge clock) begin

   if(inicializa_memoria == 1) begin
      registradores_instrucao[12] = 32'b11111100100001110000000000000000;
      inicializa_memoria = 0;
   end
end
```

Figura 15 – Parte do código do Banco de Registradores

```
   always @ (posedge clock) begin
      if(inicia_banco == 1) begin
         registradores_banco[4] = 32'b00000000000000000000101000000111;
         registradores_banco[7] = 32'b00000000000000000000010000000000;
         inicia_banco = 0;
      end
```

Figura 16 – Parte do código da Memória de Dados

```
  always @ (posedge clock) begin

     if(inicializa_memD == 1) begin
        registradores_dados[50] = 32'b00000001111110000000000000001100;
        inicializa_memD = 0;
     end
```