ID: .....

Desenvolvimento de um processador em lógica programável: NorMIPS

São José dos Campos - Brasil

Junho de 2017

ID: .....

Desenvolvimento de um processador em lógica programável: NorMIPS

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

Junho de 2017

Resumo

O objetivo deste relatório é mostrar o planejamento e desenvolvimento de uma arquitetura de processador completamente funcional utilizando lógica programável e operando em um *kit* FPGA. O projeto foi desenvolvido em 4 etapas: elaboração do **Conjunto de Instruções** e escolha da **Arquitetura Base**, desenvolvimento da **Unidade de Processamento**, construção da **Unidade de Controle** e dos sinais de controle, criação do módulo de entrada-saída e **testes no FPGA**. A arquitetura MIPS (*Microprocessor without Interlocked Pipeline Stages*) serviu de suporte teórico e inspiração para a composição da arquitetura base idealizada com um conjunto de instruções possui 19 instruções binárias que foram utilizadas em testes. A unidade de processamento é composta de *Program Counter*, Banco de Registradores, Unidade Lógica Aritmética (ULA) e um dispositivo de Memória, composto por uma Memória de Instrução e uma Memória de Dados. A unidade de controle (UC) foi implementada como uma Máquina de Estados (método *hardwire*) que tem seus sinais de controle determinados pelos sinais de entrada em questão. O produto final foi submetido por uma série de testes e avaliado pelo docente da disciplina Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores.

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

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.

Este trabalho mostra todas as etapas do desenvolvimento de uma arquitetura de processador, desde o planejamento de seu conjunto de instruções até o funcionamento no *kit* FPGA.

Este relatório foi organizado da seguinte maneira: em **Objetivos** é exposto os objetivos que o trabalho busca alcançar, na **Fundamentação Teórica** discute-se sobre os principais conceitos para entendimento do trabalho por completo. A terceira seção, **Desenvolvimento**, mostra e detalha todas as etapas do desenvolvimento do projeto: Planejamento do Conjunto de Instruções e escolha da arquitetura Base, implementação da Unidade de Processamento e dispositivos de memória, elaboração da Unidade de Controle, criação do módulo de entrada-saída e implementação do modelo na FPGA. Em **Resultados Obtidos e Discussões** discute-se sobre os testes feitos ao decorrer do projeto, onde são expostos os resultados em forma de onda (*Waveform*), para mostrar o funcionamento correto das instruções, e fotos da FPGA em funcionamento executando dois algoritmos de testes: Fibonacci e um teste lógico. Por fim, em **Considerações Finais**, é feito uma discussão sobre a realização do projeto, comentando as dificuldades enfrentadas no decorrer de sua elaboração e os próximos passos de seu desenvolvimento.

2 Objetivos

Desenvolvimento e implementação de um processador com lógica programável completamente funcional, utilizando a linguagem de descrição de *hardware* Verilog, e deverá funcionar em um dispositivo FPGA.

3 Fundamentação Teórica

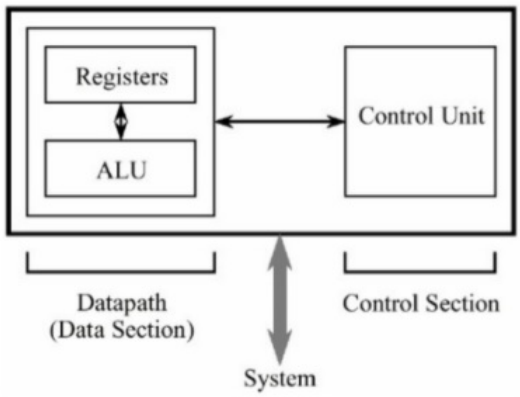
Este tópico expõe o estudo fundamental para a elaboração e efetuação do trabalho, auxiliando o leitor a compreender este relatório em sua totalidade.

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.

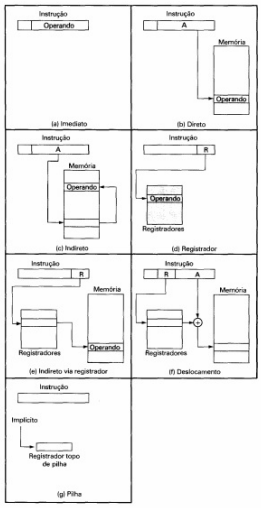
3.3.1.1 Endereçamento Imediato

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

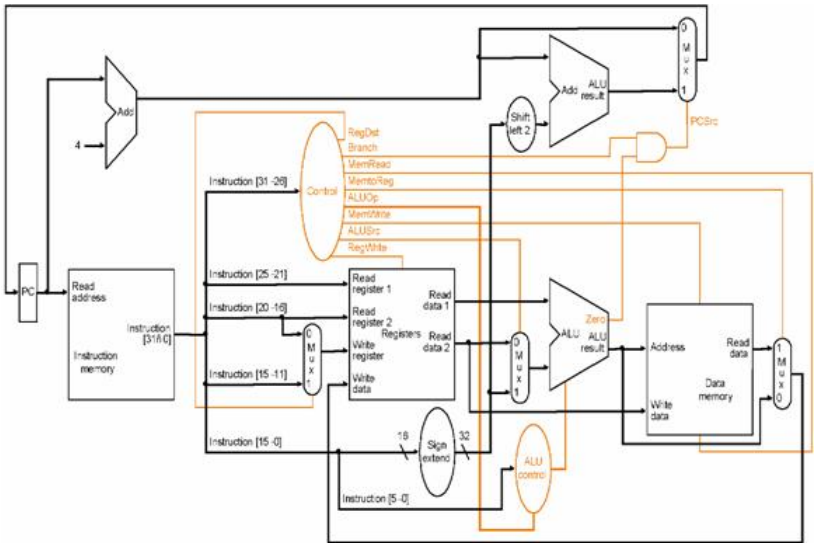
No modo de Endereçamento por Registrador, é referenciado o respectivo registrador no Campo de Endereço e nele contém o operando. Não há acesso a memória.

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 Máquina de Estados Finitos

Uma máquina de estados finitos, ou autômato finito, é uma modelagem de um comportamento composto por estados, transições e ações. A máquina está em apenas um estado por vez, chamado de estado atual. Em um estado é contido informações sobre o estado passado. Uma transição indica uma mudança de estado e é descrita por uma condição que precisa ser realizada para que a transição ocorra. Uma ação é a descrição de uma atividade que deve ser realizada em determinado momento.

3.6 Unidade de Controle

Unidade de Controle é o componente presente nos processadores, responsável por conduzir os sinais que controlam as operações nos outros módulos da CPU. É importante para o controle do processador, em que controla a ordem que as operações nos módulos são executadas e também para a generalização do projeto, pois sem ela o processador executaria apenas uma única função.

Os dois principais métodos de implementação da Unidade de Controle são: **Unidade de Controle por Microprogramação**, em que a UC possui um circuito decodificador que decodifica as instruções e as usam em um microprograma interno; **Unidade de Controle *Hardwire*** onde é dada como uma Máquina de Estados Finitos, em que os estados são determinados pela instrução em execução. (5)

3.7 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* (6).

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

3.8 Circuitos Integrados

Circuitos integrados são circuitos eletrônicos que reúne diversas peças, como transistores, diodos, resistores e capacitores. Estes elementos são introduzidos em uma lâmina de silício. Essa lâmina ("chip") é montado em um bloco, de plástico ou cerâmica, conectando os terminais dos seus componentes por fios condutores.

3.8.1 FPGA

FPGA ("*Field Programmable Gate Array*", em português "Arranjo de Portas Programável em Campo") é um tipo de circuito integrado projetado para configuração própria pelo usuário, seja ele um simples consumidor ou um projetista. Funciona como um dispositivo lógico programável que suporta implentação de circuitos digitais. A partir do *software* Quartus é gerado um arquivo binário com a configuração da FPGA. Esse arquivo binário contém as informações necessárias para especificar a função de cada unidade lógica e da seleção de *switches* e botões a serem utilizado no projeto. Para desenvolvimento do trabalho, fora utilizado o FPGA modelo EP4CE115F29C7, da Altera.

4 Desenvolvimento

O processador **NorMIPS** 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 lógico-aritméticas, dois **somadores** e quatro **multiplexadores**, para controle, além de um ***Program Counter***, para definir o endereço da instrução a ser executada. A memória principal é composta de uma **Memória de Instruçoes**, que armazena as instruções a serem executadas, e uma **Memória Principal**, para armazenar informações que não estão presentes no Banco de Registradores. Para controlar a ordem que as operações nos módulos são executadas, foi implementada uma **Unidade de Controle** e como interface para interação com o usuário, foi desenvolvido um módulo de **Entrada-Saída**.

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

4.1 Conjunto de Instruções

Para a arquitetura NorMIPS, foi planejada um conjunto de instruções contendo 18 instruções, seguindo o formato de instrução do MIPS, com tipos R (para operações lógico-aritméticas), I (para operações com imediatos e *branches*) e J (*jumps*), e adicionado um formato para instruções de entrada-saída (*input* e *output*) e transferência de dados, o I/O.

4.1.1 Formato das Instruções

Como já dito, a arquitetura NorMIPS segue o modelo de formato de instrução do MIPS, contendo três tipos de formatos:

- *R-type*: Para instruções que fazem operações lógico-aritméticas, como **ADD** e **AND**. É composta do endereço de três registradores, RS, RT e RD, em que os dois primeiros fornecem os operandos e o último é o registrador destino. O resto da instrução não é utilizada, portanto deve ser preenchida por "0".

- *I-type*: Para instruções que fazem operações de *branches* ou que utilizam de valores imediatos. É composta do endereço de dois registradores e uma parte para valor Imediato. Utilizada em instruções como **BEQ** (*branch-on-equal*) e *ADDI*.

- *J-type*: Para instruções de ***Jump***. Composta por apenas uma parte (fora o *opcode*), que deve conter o valor imediato de 26 *bits*. ***JUMP***, **HALT** e **NOP** são as instruções que possuem este formato de instrução.

- *I/O-type*: Para instruções de Entrada-Saída e transferência de dados. Sua composição é similar ao *I-type*, mas a posição do primeiro registrador não é utilizada, sendo geralmente definida com cinco zeros (00000). As instruções de I/O, *Input* e *Output*, e de transferência de dados, **LW** (*Load Word*) e **SW** (*Store Word*), possuem este tipo de formato.

Na Figura 4 é mostrado a organização dos formatos de instruções citados.

Figura 4 – Formatos de Instrução de NorMIPS

			NorMIPS			
Formato	op	rs	rt	rd	shamt	funct
Tipo R	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Formato	op	rs	rt		endereço	
Tipo I	6 bits	5 bits	5 bits		16 bits	
Formato	op			endereço		
Tipo J	6 bits			26 bits		
Formato	op	NULL	rt		endereço	
Tipo I/O	6 bits	5 bits	5 bits		16 bits	

4.1.2 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 que 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. Para melhor compreensão da tabela, logo abaixo está a descrição de cada componente citado:

- RD: Endereço de 5 bits do registrador destino;

- RS: Endereço de 5 bits do registrador de informação;

- RT: Endereço de 5 bits que, em diferentes casos, atua como registrador de informação e registrador de destino.

- IMED: Valor imediato de tamanho 16 bits;

- PC: *Program Counter*;

- SWITCHES(E/S): *Switches* de entrada de dados. Como são utilizados 16 alavancas, tem tamanho 16 bits;

- DISPLAY(E/S): *Display* de 7 segmentos presente na FPGA;

- MEM[AD]: Endereços referentes aos dispositivos de memória;

Tabela 1 – Mapeamento das Instruções

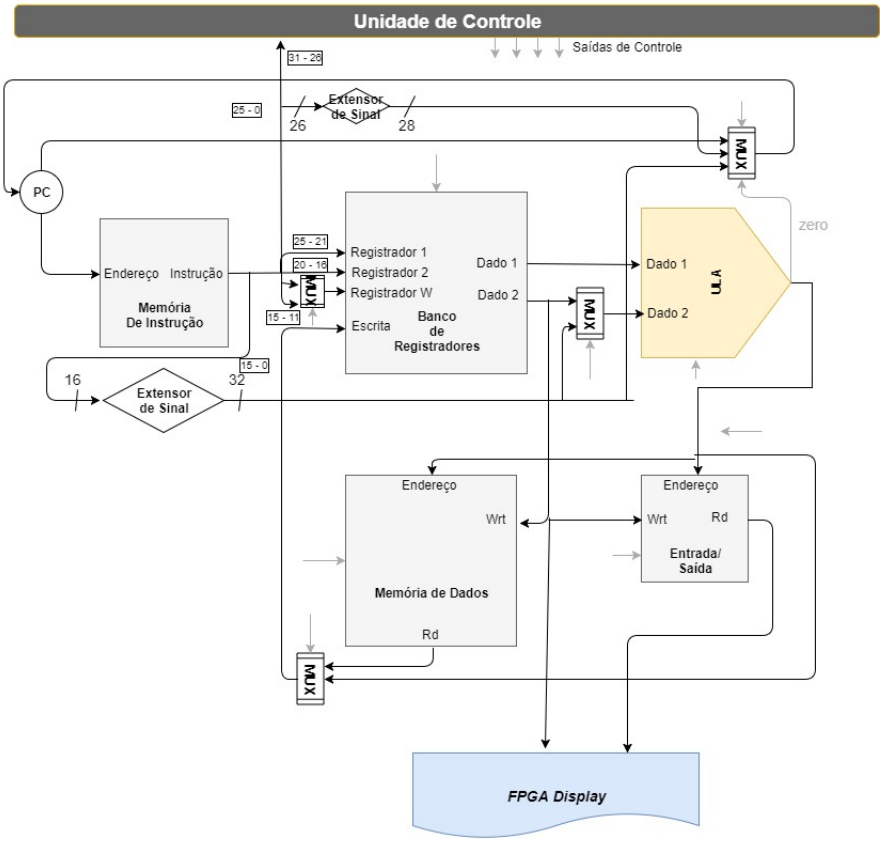
Instrução	**OpCode**	**Modo de Endereçamento**	**Operação**	**Tipo**
ADD	100000	Por Registrador	RD = RS + RT	R
ADDI	010000	Imediato	RT = RS + IMED	I
SUB	100001	Por Registrador	RD = RS - RT	R
SUBI	010001	Imediato	RT = RS - IMED	I
AND	100010	Por Registrador	RD = RS & RT	R
OR	100011	Por Registrador	RD = RS | RT	R
SLT	100100	Por Registrador	RD = (RS <RT)	R
MOV	010100	Por Registrador	RT = RS	R
BEQ	010110	Relativo ao PC	condicional ==	I
BNQ	010111	Relativo ao PC	condicional !=	I
SW	011110	Direto	Mem[AD] = RT	I/O
LW	011111	Direto	RT = Mem[AD]	I/O
LOADI	011001	Imediato	RT = IMED	I/O
J	111111	Absoluto	Jump	J
NOP	000000		PC + 1	J
HALT	111110		Espera ENTER	J
IN	000111		RT = SWITCHES(E/S)	I/O
OUT	111000		DISPLAY(E/S) = RT	I/O

4.2 Unidade de Processamento

A arquitetura base utilizada foi inspirada na arquitetura MIPS monociclo. A Unidade de Processamento de NorMIPS é composta de um Banco de Registradores, que contém registradores que armazenarão as informações em operação, uma Unidade Lógica Aritmética, que realizará as operações lógico-aritmética, *Program Counter*, que armazenará o endereço da próxima instrução, um dispositivo de Memória, composta de Memória de Dados e Memória de Instruções, extensores de sinal e multiplexadores. A organização da arquitetura pode ser vista na Figura 5.

A Unidade de Controle será melhor discutida em uma outra seção, onde será explicada sua funcionalidade e seus sinais de controle.

Figura 5 – Esquematico



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 registrador de 32 bits (endereco_atual), o qual indica o endereço da próxima instrução. O código do *Program Counter* pode ser visto logo abaixo.

```
1  module programCounter(proximo_endereco, endereco_atual, clock, reset, halt);
2
3          // input
4          input [31:0] proximo_endereco;
5          input clock, reset, halt;
6
7          // output
8          output reg [31:0] endereco_atual;
9
10         // registradores
11         //reg [31:0] registrador_pc;
12
13
14         always @ (posedge clock) begin
15
16                 if(reset) begin
17                         endereco_atual = 0;
18                 end
19                 else if(halt) begin
20
21                 end
```

```
22                  else begin
23                          endereco_atual = proximo_endereco;
24                  end
25          end
26  endmodule
```

A próxima instrução a ser executada é definida antes de chegar ao PC, no MUX_4. O PC serve simplesmente para repassar a Memória de Instrução qual a próxima instrução a se executar na subida do *Clock*, verificando a ocorrência de *Reset* e HALT:

- Caso o botão *Reset* for acionado, o PC indicará o endereço 0 à Memória de Instrução;
- Caso o processador entre em estado de HALT, nada mais será indicado;
- Caso não ocorra nenhuma das condições acima, ele repassará o endereço normalmente.

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.

```
1   module bancoRegistradores (endereco_read1, endereco_read2, endereco_wrt, dado_wrt,
        saida_RD, saida_RS, saida_RT, controle /* Controle de Escrita */, clock);
2
3           // input
4           input [4:0] endereco_read1;
5           input [4:0] endereco_read2;
6           input [4:0] endereco_wrt;
7           input [31:0] dado_wrt;
8           input clock, controle;
9
10          // registradores
11          reg [31:0] registradores_banco[31:0];
12
13          // output
14          output [31:0] saida_RD;
15          output [31:0] saida_RS;
16          output [31:0] saida_RT;
17
18
19          always @ (posedge clock) begin
20
21                  if(controle == 1) registradores_banco[endereco_wrt] = dado_wrt;
22          end
23
24          // Resultados Saidas
25          assign saida_RD = registradores_banco[endereco_wrt];
26          assign saida_RS = registradores_banco[endereco_read1];
27          assign saida_RT = registradores_banco[endereco_read2];
28  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 (controle_BANCOREG) 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	MOV	dado1 = dado2
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 @ (controle or dado1 or dado2) begin
13
14                 case(controle[2:0])
15
16                         3'b000: saida = dado1 + dado2;  // ADD
17                         3'b001: saida = dado1 - dado2;  // SUB
18                         3'b010: saida = dado1 & dado2;          // AND
19                         3'b011: saida = dado1 | dado2;  // OR
20                         3'b100: saida = dado1;          // MOV
21                         3'b101: saida = dado1 >> 1;             // SHIFT RIGHT 1
22                         3'b110: begin
                               // SET LESS THAN
23                                 if (dado1[31] != dado2[31]) begin
24                                         if (dado1[31] > dado2[31]) begin
```

```
25                                                  saida = 1;
26                                          end
27                                          else begin
28                                                  saida = 0;
29                                          end
30                                  end
31                                  else begin
32                                          if (dado1 < dado2) begin
33                                                  saida = 1;
34                                          end
35                                          else begin
36                                                  saida = 0;
37                                          end
38                                  end
39                          end
40                          3'b111: saida = ~dado1;                         // NOT
41                  endcase
42          end
43
44          assign sinal_ZERO = (saida == 0);
45          assign sinal_NEG = (($signed(saida) < 0));
46  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 (sinal_ZERO) e quando o resultado é negativo (sinal_NEG). **sinal_ZERO** será ligado ao MUX_4 (multiplexador responsável por determinar qual próximo endereço de instrução será executado), sendo fundamental para instruções do tipo *branch* (determina se ocorrerá ou não o pulo).

4.2.4 Multiplexadores

Foram desenvolvidos 5 multiplexadores, colocados na arquitetura para escolher uma dentre várias entradas, fazendo com que o processador funcione da maneira correta. As entradas são escolhidas a partir do sinal de controle recebido pelo componente. Foram desenvolvidos três multiplexadores controlados por um sinal de controle de 1 bit (MUX_1, MUX_2 e MUX_5) e dois controlados por 2 bits (MUx_3 e MUX_4).

A Tabela 3 indica as entradas e as respectivas saídas para cada sinal de controle em "**MUX 1**", "**MUX 2**"e "**MUX 5**.

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_5	Instrução[15-0]	SWITCHES

O **MUX_1** está posicionado antes da entrada do Banco de Registradores e é responsável por determinar qual o endereço do registrador de escrita, RT (20:16) ou RD (15:11). O **MUX_2** está depois do Banco de Registradores e indica qual será o dado de segunda entrada da ULA, se é o dado2 que sai do Banco de Registradores ou o extendido da instrução. O **MUX_5** determina qual será o dado a ser extendido de 16 bits para 32 bits, podendo ser a parte 15:0 da instrução ou a seleção dos *switches*.

O **MUX_3** tem função de escolher qual dado que será escrito no Banco de Registradores: o resultado da ALU (como ocorre na instrução ADD), a saida da Leitura da Memória de Dados (instrução LW) ou o extendido (IPT). A Tabela 4 mostra as saídas do MUX_3 para cada sinal de controle.

Tabela 4 – Saídas do MUX_3

Controle	**Saída**
01	Saída MEM DADOS
10	EXTENDIDO
default	Resultado ALU

O "**MUX 4**"é responsável por indicar ao *Program Counter* qual o endereço da próxima instrução a ser executada. Os sinais de controle indicam quando é para apenas incrementar o endereço do PC, quando for uma instrução do tipo *branch on equal*, ocorrendo a verificação do sinal_ZERO, quando for uma instrução do tipo *branch on not equal*, e quando for uma instrução do tipo *jump*, tendo como saída o endereço do *jump*. A Tabela 5 indica as saídas para MUX_4.

Tabela 5 – Saídas do MUX_4

Controle	**Saída**
01	Endereco - BEQ ou PC + 1
10	Endereco - BNQ ou PC + 1
00	PC + 1
11	Endereco - JUMP

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, como campo de endereço de instruções do tipo *branch* ou *Load Word* e seleção das alavancas na instrução de *Input*. 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 o valor absoluto de 26 bits em 32 bits.

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[70:0];
9
10         // output
11         output [31:0] instrucao;
12
13         // inicializador
14
15         always @ (posedge clock) begin
16
17
18         end
19
20         assign instrucao = registradores_instrucao[endereco];
21 endmodule
```

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 Con-

trole), sinal_MEMDADOS 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[20:0];
10
11
12         // output
13         output [31:0] saida_dado;
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**".

4.4 Unidade de Controle

A Unidade de Controle de NorMIPS foi implementada pelo método *hardwire*, em que a UC é dada como uma Máquina de Estados Finitos. São emitidos 8 sinais de controle para controlar cada componente da CPU que podem ser vistos na Tabela 6.

Tabela 6 – Sinais de Controle

sinal	**tamanho**	**sinal**	**tamanho**
controle_BANCOREG	1 bit	controle_MUX2	1 bit
controle_ALU	3 bits	controle_MUX3	2 bits
controle_MEMDADOS	1 bit	controle_MUX4	2 bits
controle_MUX1	1 bit	controle_MUX5	1 bit

O sinal **controle_BANCOREG** é responsável por determinar se ocorrerá escrita no Banco de Registradores (1 - Escrita, 0 - Leitura). **controle_ALU** determina qual operação será executada na Unidade Lógica Aritmética, os sinais para cada operação da ULA podem ser vistos na Tabela 7. **controle_MEMDADOS** tem função semelhante a de controle_BANCOREG, determinando se ocorrerá escrita na Memória de Dados (1 - Escrita, 0 - Leitura). **controle_MUX1**, **controle_MUX2** e **controle_MUX5** são sinais que controlam a saídas dos multiplexadores respectivos (MUX1, MUX2 e MUX5, no caso). **controle_MUX3** é o sinal de controle que indica qual dado será escrito no banco de Registradores.**controle_MUX4** é responsável por determinar o próximo endereço da instrução que será colocado no *Program Counter*.

Tabela 7 – Sinais de Controle da ULA

SINAL	OPERAÇÃO	SINAL	OPERAÇÃO
000	A + B	**100**	A = B
001	A - B	**101**	A »1
010	A & B	**110**	if (A <B)
011	A | B	**111**	∼A

4.4.1 Sinais de Controle para instruções do Tipo R

São as instruções lógico-aritméticas. Se caracterizam por utilizarem 2 registradores que fornecem dados para as operações e um para destino. A maioria dos sinais de controle são os mesmos para estas instruções, não ocorrendo escrita na memória de dados (controle_MEMDADOS = 0) mas permitindo a escrita no banco de registradores (controle_BANCOREG = 1). Os estados da Unidade de Controle podem ser vistos na Figura 6.

4.4.2 Sinais de Controle para instruções do Tipo I

São as instruções que utilizam imediatos e *branches*. As instruções de *branch* (BEQ e BNQ) são as únicas do Conjunto de Instrução que fazem uso do sinal_NEG emitido pela ALU. Há muitas variações nos sinais para este formato de instrução, inclusive no sinal controle_MUX4, que indica as diferenças de *branches* ou dá apenas PC + 1. Os sinais de controle pode ser visto na Figura 7.

4.4.3 Sinais de Controle para instruções do Tipo J

São as instruções de *Jump* e aquelas que modificam o estado do Processador. Na instrução JUMP, quase tudo é dado como 0 ou "*don't care*"(x), pois apenas a operação de pulo de instrução ocorrerá. O NOP tem seu foco no sinal controle_MUX4, pois dará o

Figura 6 – Sinais de Controle para Instruções do tipo R

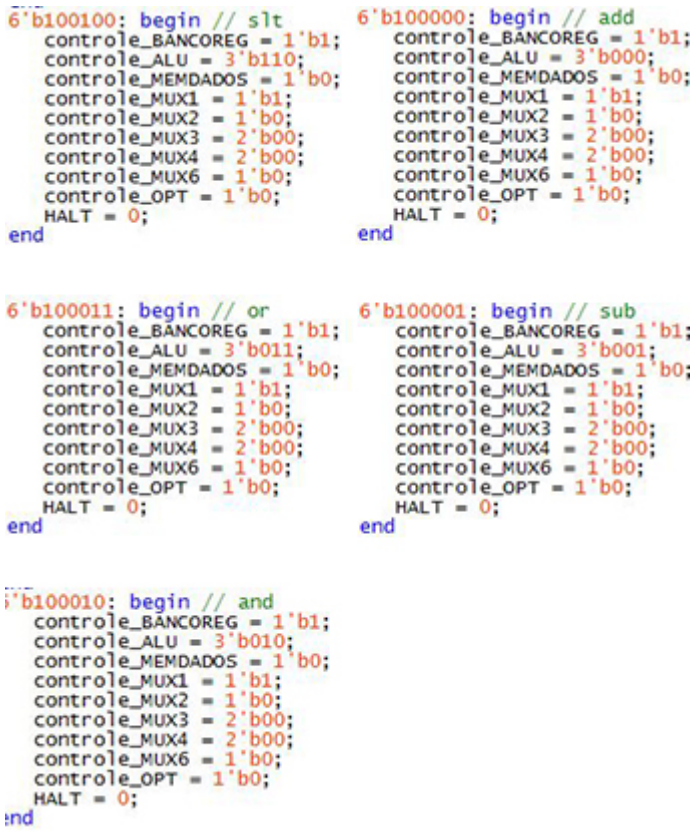
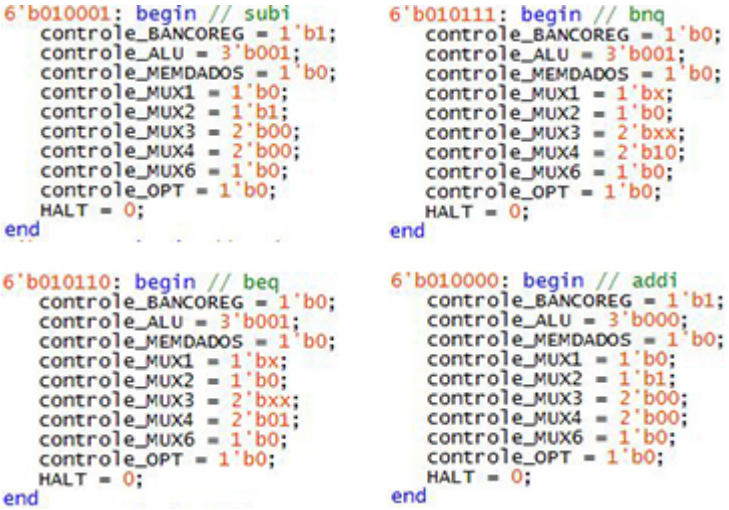


Figura 7 – Sinais de Controle para instruções do tipo I



PC + 1 e o HALT no sinal HALT, que dará o estado de parada ao sistema. Os sinais de controle para estas instruções podem ser vistas na Figura 8

É importante ressaltar que a instrução HALT sempre verifica o pressionamento do botão ENTER, para voltar a operar modificando o sinal HALT = 0.

Figura 8 – Sinais de Controle para instruções do tipo J

```
6'b000000: begin // nop
   controle_BANCOREG = 1'b0;
   controle_ALU = 3'bxxx;
   controle_MEMDADOS = 1'b0;
   controle_MUX1 = 1'bx;
   controle_MUX2 = 1'bx;
   controle_MUX3 = 2'bxx;
   controle_MUX4 = 2'b00;
   controle_MUX6 = 1'bx;
   controle_OPT = 1'b0;
   HALT = 0;
end

6'b111111: begin // jump
   controle_BANCOREG = 1'b0;
   controle_ALU = 3'bxxx;
   controle_MEMDADOS = 1'b0;
   controle_MUX1 = 1'bx;
   controle_MUX2 = 1'bx;
   controle_MUX3 = 2'bxx;
   controle_MUX4 = 2'b11;
   controle_MUX6 = 1'bx;
   controle_OPT = 1'b0;
   HALT = 0;
end
```

```
6'b111110: begin // HALT
   if(botao) begin
      HALT = 0;
      controle_MUX4 = 2'b00;
   end
   else if (clock) begin
      controle_BANCOREG = 1'b0;
      controle_ALU = 3'bxxx;
      controle_MEMDADOS = 1'b0;
      controle_MUX1 = 1'bx;
      controle_MUX2 = 1'bx;
      controle_MUX3 = 2'bxx;
      controle_MUX4 = 2'b11;
      controle_MUX6 = 1'bx;
      controle_OPT = 1'b0;
      HALT = 1;
   end
end
```

4.4.4 Sinais de Controle para instruções do Tipo I/O

São as instruções que ativam a interface de entrada-saída e as que fazem transferência de dados. Aqui há a ocorrência de sinais que são dados como 0 nos outros tipos de instrução: controle_OPT (controle de *Output* e controle_MEMDADOS. Como ocorre a transferência de dados, há a necessidade de permissão de escrita na Memória de Dados. A instrução de LOADI tem sinais semelhantes a instruções do tipo R. Os sinais de controle podem ser vistos na Figura 9.

Figura 9 – Sinais de Controle para instruções do tipo I/O

```
6'b011001: begin // loadi
   controle_BANCOREG = 1'b1;
   controle_ALU = 3'bxxx;
   controle_MEMDADOS = 1'b0;
   controle_MUX1 = 1'b0;
   controle_MUX2 = 1'bx;
   controle_MUX3 = 2'b10;
   controle_MUX4 = 2'b00;
   controle_MUX6 = 1'b0;
   controle_OPT = 1'b0;
   HALT = 0;
end

6'b011110: begin // sw
   controle_BANCOREG = 1'b0;
   controle_ALU = 3'bxxx;
   controle_MEMDADOS = 1'b1;
   controle_MUX1 = 1'bx;
   controle_MUX2 = 1'bx;
   controle_MUX3 = 2'bxx;
   controle_MUX4 = 2'b00;
   controle_MUX6 = 1'b0;
   controle_OPT = 1'b0;
   HALT = 0;
end

6'b111000: begin // out
   controle_BANCOREG = 1'b0;
   controle_ALU = 3'bxxx;
   controle_MEMDADOS = 1'b0;
   controle_MUX1 = 1'bx;
   controle_MUX2 = 1'bx;
   controle_MUX3 = 2'bxx;
   controle_MUX4 = 2'b00;
   controle_MUX6 = 1'bx;
   controle_OPT = 1'b1;
   HALT = 0;
end
```

```
6'b000111: begin // in
   if(botao) begin
      HALT = 0;
      controle_MUX4 = 2'b00;
   end
   else if (clock) begin
      controle_BANCOREG = 1'b1;
      controle_ALU = 3'bxxx;
      controle_MEMDADOS = 1'b0;
      controle_MUX1 = 1'b0;
      controle_MUX2 = 1'bx;
      controle_MUX3 = 2'b10;
      controle_MUX4 = 2'b00;
      controle_MUX6 = 1'b1;
      controle_OPT = 1'b0;
      HALT = 1;
   end

end

6'b011111: begin // lw
   controle_BANCOREG = 1'b1;
   controle_ALU = 3'b000;
   controle_MEMDADOS = 1'b0;
   controle_MUX1 = 1'b0;
   controle_MUX2 = 1'b1;
   controle_MUX3 = 2'b01;
   controle_MUX4 = 2'b00;
   controle_MUX6 = 1'b0;
   controle_OPT = 1'b0;
   HALT = 0;
end
```

Na instrução de IPT, há a ocorrência de um estado de parada (HALT), tendo seu tratamento semelhante a instrução de HALT.

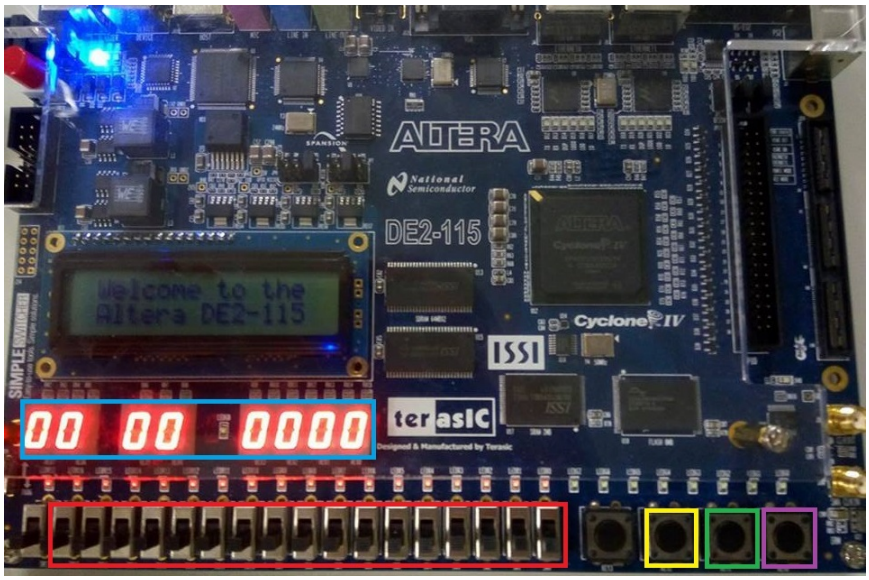
O código completo da Unidade de Controle pode ser encontrado em Código completo da Unidade de Controle.

4.5 Entrada e Saída

O módulo de Entrada-Saída é responsável pela intercomunicação do processador implementado na FPGA com o usuário. Ela é responsável por receber as informações das alavancas (*switches*) e salvar no registrador durante a instrução IPT (*Input*) e por emitir de forma visual nos *displays* de 7 segmentos da FPGA durante a instrução OPT (*Output*).

A Figura 10 mostra os componentes da interface de entrada e saída do processador.

Figura 10 – Fotografia da FPGA com indicações de seus componente de Entrada-Saída



Na Figura 10, a área circulada em vermelho representa são as 16 alavancas de dados utilizadas na instrução de *input*. Os botões circulados em amarelo e verde representam o botão de *Reset* e *Enter*, respectivamente. Os *displays* circulados em azul são os utilizados na instrução de *output*. O botão em roxo representa o *Clock*, utilizado para a fase de testes, Foi substituído por um Temporizador.

4.5.1 Entrada

A entrada de dados foi organizada de maneira bem simples: recebe-se o valor das alavancas e armazena-se no registrador destino (RT). O valor das alavancas será escolhido pelo MUX_5 para ir ao extensor e esse extendido será escolhido no MUX_3 para ser escrito no Banco de Registradores. Quando é acionada a instrução de *input*, o processador

entra em estado de HALT, ou seja, não executará mais operação. Ele só sairá do estado de HALT, quando o botão ENTER for acionado. Fica a cargo da Unidade de Controle fazer esta verificação.

4.5.2 Saída

A saída de dados é feita pela instrução *Output* utilizando do módulo "output_normips"(Saída). Ele recebe um binário de 32 bits do Banco de Registradores e faz com que cada *display* receba o seu valor correspondente. O módulo "binario_bcd"(Saída) é responsável por fazer a conversão do dado binário para uma notação decimal, suportada pelos *displays*. "sete_segmentos"o (Saída) recebe o dado convertido por "binario_bcd"e imprime os seus LEDS formando um número no *display*. Declarou-se um "sete_segmentos"para cada *display*".

```
1  module output_normips (binario, unidade, dezena, centena, milhar, d_milhar, c_milhar,
       milhao, d_milhao, clock, controle);
2          input clock, controle;
3          input [31:0] binario;
4
5          reg [31:0] aux_binario;
6
7          output [6:0] unidade, dezena, centena, milhar, d_milhar, c_milhar, milhao,
               d_milhao;
8
9          wire [3:0] bin_unidade, bin_dezena, bin_centena, bin_milhar, bin_dmilhar,
               bin_cmilhar, bin_milhao, bin_dmilhao;
10
11         binario_bcd binario_bcd(binario, bin_unidade, bin_dezena,
12                                                                  bin_centena,
                                                                        bin_milhar,
                                                                        bin_dmilhar,
13                                                                  bin_cmilhar,
                                                                        bin_milhao,
                                                                        bin_dmilhao);
14
15         sete_segmentos displayA(bin_unidade, unidade[0], unidade[1],
16                                                                  unidade[2],
                                                                        unidade[3],
                                                                        unidade[4],
17                                                                  unidade[5],
                                                                        unidade[6],
                                                                        clock,
                                                                        controle);
18
19         sete_segmentos displayB(bin_dezena, dezena[0], dezena[1],
20                                                                  dezena[2], dezena
                                                                        [3], dezena
                                                                        [4],
21                                                                  dezena[5], dezena
                                                                        [6], clock,
                                                                        controle);
22
23         sete_segmentos displayC(bin_centena, centena[0], centena[1],
24                                                                  centena[2],
                                                                        centena[3],
```

```
                                                                centena[4],
25                                                            centena[5],
                                                                centena[6],
                                                                clock,
                                                                controle);
26
27          sete_segmentos displayD(bin_milhar, milhar[0], milhar[1],
28                                                            milhar[2], milhar
                                                                [3], milhar
                                                                [4],
29                                                            milhar[5], milhar
                                                                [6], clock,
                                                                controle);
30
31          sete_segmentos displayE(bin_dmilhar, d_milhar[0], d_milhar[1],
32                                                            d_milhar[2],
                                                                d_milhar[3],
                                                                d_milhar[4],
33                                                            d_milhar[5],
                                                                d_milhar[6],
                                                                clock,
                                                                controle);
34
35          sete_segmentos displayF(bin_cmilhar, c_milhar[0], c_milhar[1],
36                                                            c_milhar[2],
                                                                c_milhar[3],
                                                                c_milhar[4],
37                                                            c_milhar[5],
                                                                c_milhar[6],
                                                                clock,
                                                                controle);
38
39          sete_segmentos displayG(bin_milhao, milhao[0], milhao[1],
40                                                            milhao[2], milhao
                                                                [3], milhao
                                                                [4],
41                                                            milhao[5], milhao
                                                                [6], clock,
                                                                controle);
42
43          sete_segmentos displayH(bin_dmilhao, d_milhao[0], d_milhao[1],
44                                                            d_milhao[2],
                                                                d_milhao[3],
                                                                d_milhao[4],
45                                                            d_milhao[5],
                                                                d_milhao[6],
                                                                clock,
                                                                controle);
46
47  endmodule
```

```
1   module binario_bcd(binario, unidade, dezena, centena, milhar, d_milhar, c_milhar, milhao,
        d_milhao);
2
3           input [31:0] binario;
4
5           output reg [3:0] unidade, dezena, centena, milhar, d_milhar, c_milhar, milhao,
                d_milhao;
6
7           integer i;
```

```
8
9         always @ (binario) begin
10                unidade = 4'b0;
11                dezena = 4'b0;
12                centena = 4'b0;
13                milhar = 4'b0;
14                d_milhar = 4'b0;
15                c_milhar = 4'b0;
16                milhao = 4'b0;
17                d_milhao = 4'b0;
18
19                for(i = 31; i >= 0; i = i - 1) begin
20                        if(unidade >= 5)
21                                unidade = unidade + 3;
22                        if(dezena >= 5)
23                                dezena = dezena + 3;
24                        if(centena >= 5)
25                                centena = centena + 3;
26                        if(milhar >= 5)
27                                milhar = milhar + 3;
28                        if(d_milhar >= 5)
29                                d_milhar = d_milhar + 3;
30                        if(c_milhar >= 5)
31                                c_milhar = c_milhar + 3;
32                        if(milhao >= 5)
33                                milhao = milhao + 3;
34                        if(d_milhao >= 5)
35                                d_milhao = d_milhao + 3;
36
37                        d_milhao = d_milhao << 1;
38                        d_milhao[0] = milhao[3];
39                        milhao = milhao << 1;
40                        milhao[0] = c_milhar[3];
41                        c_milhar = c_milhar << 1;
42                        c_milhar[0] = d_milhar[3];
43                        d_milhar = d_milhar << 1;
44                        d_milhar[0] = milhar[3];
45                        milhar = milhar << 1;
46                        milhar[0] = centena[3];
47                        centena = centena << 1;
48                        centena[0] = dezena[3];
49                        dezena = dezena << 1;
50                        dezena[0] = unidade[3];
51                        unidade = unidade << 1;
52                        unidade[0] = binario[i];
53                end
54        end
55
56 endmodule
```

```
1  module sete_segmentos(numero, A, B, C, D, E, F, G, clock, controle);
2
3         input clock, controle;
4         input [3:0] numero;
5
6         output A, B, C, D, E, F, G;
7
8         reg [6:0] resultado;
9
10        always @ (posedge clock) begin
```

```
11                if(controle) begin
12                        case(numero)
13                                4'b0000: resultado = 7'b1111110;
14                                4'b0001: resultado = 7'b0110000;
15                                4'b0010: resultado = 7'b1101101;
16                                4'b0011: resultado = 7'b1111001;
17                                4'b0100: resultado = 7'b0110011;
18                                4'b0101: resultado = 7'b1011011;
19                                4'b0110: resultado = 7'b1011111;
20                                4'b0111: resultado = 7'b1110000;
21                                4'b1000: resultado = 7'b1111111;
22                                4'b1001: resultado = 7'b1111011;
23                                default: resultado=  7'b0000000;//0000001
24                        endcase
25                end
26                else begin
27                        resultado = 7'b0000001;
28                end
29        end
30
31        assign {A, B, C, D, E, F, G} = ~resultado;
32
33 endmodule
```

Observação: Quando o processador não está executando uma instrução de *Output*, por *default*, fora implementado de forma que apenas o LED "G"(do meio) ficasse aceso, formando um traço.

4.6 Debounce

O *Debounce* é o módulo responsável por intervir no efeito "*Bouncing*"causado ao apertar um botão. Foi utilizado no botão de *Clock*, para que os testes fossem realizados. O código do Debounce está no Apêndice A. (7)

4.7 Junção dos Módulos

Os módulos foram organizados e conectados de maneira que seu funcionamento seja o mais correto possível, baseando-se na arquitetura base (Figura 5). O código da CPU completa está descrito logo abaixo:

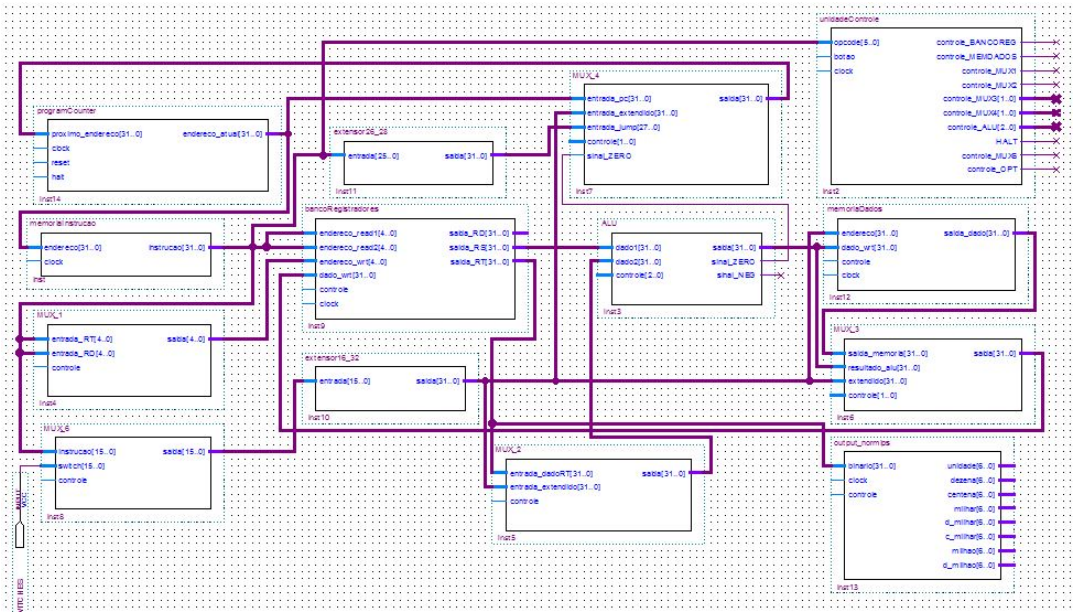
```
1  module normips (CLOCK, BTN, RESET, BOTAO, SWITCH, UNIDADE, DEZENA,
2                                          CENTENA, MILHAR, D_MILHAR, C_MILHAR, MILHAO,
                                               D_MILHAO, ALU_RESULT_TEST, INSTRUCAO_TEST,
3                                          SAIDA_RT_TEST, MEMDADOS_SAIDA_TEST);
4
5         input CLOCK, RESET;
6         input BTN;
7         input [15:0] SWITCH;
8         input BOTAO;
9
10        wire CLOCK;
11        wire [31:0] ENDERECO_INSTRUCAO;
```

```
12          wire [31:0] INSTRUCAO;
13          wire BANCOREG_CONTROL, MEMDADOS_CONTROL, MUX1_CONTROL, MUX2_CONTROL, SINAL_ZERO,
                SINAL_NEG;
14          wire [1:0] MUX3_CONTROL;
15          wire [4:0] SAIDA_MUX1;
16          wire [31:0] SAIDA_MUX3;
17          wire [31:0] SAIDA_RD;
18          wire [31:0] SAIDA_RS;
19          wire [31:0] SAIDA_RT;
20          wire [31:0] EXTENDIDO;
21          wire [31:0] SAIDA_MUX2;
22          wire [31:0] ALU_RESULT;
23          wire [2:0] ALU_CONTROL;
24          wire [31:0] MEMDADOS_SAIDA;
25          wire [31:0] EXTENDIDO2;
26          wire [31:0] EXTENDIDO_DESLOCADO;
27          wire [31:0] SAIDA_MUX4;
28          wire [1:0] MUX4_CONTROL;
29          wire HALT;
30          wire MUX6_CONTROL;
31          wire [15:0] SAIDA_MUX6;
32          wire OPT_CONTROL;
33          wire [31:0] SAIDA_MUX7;
34
35
36          output [6:0] UNIDADE, DEZENA, CENTENA, MILHAR, D_MILHAR, C_MILHAR, MILHAO,
                D_MILHAO;
37
38          debouncer debouncer(CLOCK1, ~BTN, CLOCK);
39
40          programCounter programCounter(SAIDA_MUX4, ENDERECO_INSTRUCAO, CLOCK, RESET, HALT)
                ;
41
42          memoriaInstrucao memoriaInstrucao(ENDERECO_INSTRUCAO, INSTRUCAO, CLOCK);
43
44          unidadeControle unidadeControle(INSTRUCAO[31:26], BOTAO, CLOCK,
45          BANCOREG_CONTROL, MEMDADOS_CONTROL,
46          MUX1_CONTROL, MUX2_CONTROL, MUX3_CONTROL,
47      MUX4_CONTROL,ALU_CONTROL, HALT,
48          MUX6_CONTROL, OPT_CONTROL);
49
50          MUX_1 MUX_1(INSTRUCAO[20:16], INSTRUCAO[15:11], SAIDA_MUX1, MUX1_CONTROL);
51
52          bancoRegistradores bancoRegistradores(INSTRUCAO[25:21], INSTRUCAO[20:16],
53          SAIDA_MUX1, SAIDA_MUX3,
54          SAIDA_RD, SAIDA_RS, SAIDA_RT,
55          BANCOREG_CONTROL, CLOCK);
56
57          MUX_6 MUX_6(INSTRUCAO[15:0], SWITCH, SAIDA_MUX6, MUX6_CONTROL);
58
59          extensor16_32 extensor16_32(SAIDA_MUX6, EXTENDIDO);
60
61          MUX_2 MUX_2(SAIDA_RT, EXTENDIDO, SAIDA_MUX2, MUX2_CONTROL);
62
63          ALU ALU(SAIDA_RS, SAIDA_MUX2, ALU_RESULT, ALU_CONTROL, SINAL_ZERO, SINAL_NEG);
64
65          memoriaDados memoriaDados(EXTENDIDO, SAIDA_RT, MEMDADOS_SAIDA,
66          MEMDADOS_CONTROL, CLOCK);
67
68          MUX_3 MUX_3(MEMDADOS_SAIDA, ALU_RESULT, EXTENDIDO, SAIDA_MUX3, MUX3_CONTROL);
```

```
69
70          extensor26_28 extensor26_28(INSTRUCAO[25:0], EXTENDIDO2);
71
72          MUX_4 MUX_4(ENDERECO_INSTRUCAO, EXTENDIDO, EXTENDIDO2,
73                                                      SAIDA_MUX4, MUX4_CONTROL,
                                                            SINAL_ZERO);
74
75          output_normips output_normips(SAIDA_RT, UNIDADE, DEZENA, CENTENA, MILHAR,
76          D_MILHAR, C_MILHAR, MILHAO,
77          D_MILHAO, CLOCK, OPT_CONTROL);
78
79  endmodule
```

A Figura 11 mostra o esquemático de blocos da CPU.

Figura 11 – Esquemático de Blocos da CPU



5 Resultados Obtidos e Discussões

Neste capítulo será demonstrado os resultados obtidos da aplicação de diversos testes na arquitetura de processador desenvolvida. Os testes foram divididos em duas partes: Testes das Instruções, em que será demonstrado as simulações em forma de onda para verificação do funcionamento de cada instrução, e Testes de Algoritmos, o qual será demonstrado as formas de onda e o funcionamento da FPGA para o algoritmo de Fibonacci e um lógico.

Todos os algoritmos deste capítulo estão presentes no Apêndice.

5.1 Testes das Instruções

Utilizando-se do *software* Quartus Prime 16.1 *Lite Edition*, foram feitos testes do funcionamento das instruções.

5.1.1 Testes de Instruções do tipo R

As instruções do tipo R são: ADD, SUB, AND, OR e SLT. Para que as operações ocorra, as seguintes linhas de código foram executadas previamente:

- LOADI REG(0) IMED 07;
- LOADI REG(1) IMED 09;

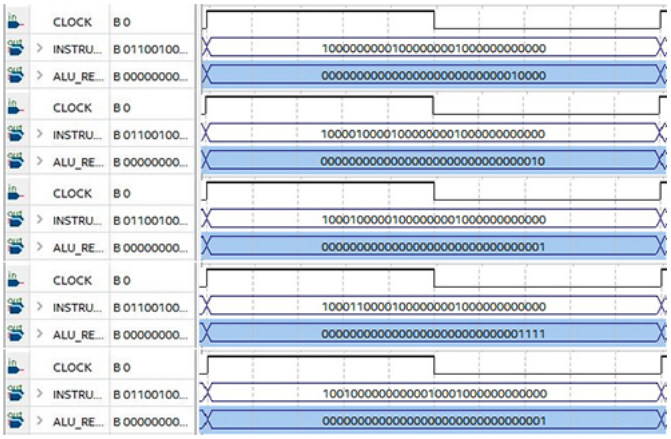
Basicamente está sendo aplicada a instrução LOADI (*Load Imediato*) para carregar os registradores de endereço 0 e 1 com os valores imediatos 07 e 09, respectivamente. Esses registradores serão utilizados para fornecerem os dados para cada instrução do tipo R:

- ADD REG(1) REG(0) REG(2);
- SUB REG(1) REG(0) REG(2);
- AND REG(1) REG(0) REG(2);
- OR REG(1) REG(0) REG(2);
- SLT REG(0) REG(1) REG(2);

A Figura 20 mostra o algoritmo acima em linguagem de máquina.

A primeira forma de onda representa a instrução ADD. Como os dados colocados nos REG(0) e REG(1) foram 7 e 9, respectivamente, temos que a ALU_RESULT (*wire*

Figura 12 – Forma de Onda do teste para instruções do tipo R



do resultado da ALU) resultou em 10000 (16 em binário). O mesmo ocorreu corretamente para as formas de onda das outras instruções. No caso da instrução de SLT (*set-less-than*), ultima forma de onda, temos que o ALU_RESULT foi 1, pois REG(0) é subitamente menor que REG(1) ($7 < 9$).

5.1.2 Testes de Instruções do tipo I

As instruções do tipo I são: ADDI, SUBI, BEQ e BNQ. O algoritmo de teste para estas instruções está logo abaixo:

- LOADI REG(0) IMED 05;
- LOADI REG(2) IMED 10;
- ADDI REG(0) REG(1) IMED 05;
- BEQ REG(1) REG(2) ENDEREÇO 10;
- SUBI REG(2) REG(1) IMED 05;
- BNQ REG(1) REG(0) ENDEREÇO 0;

A Figura 21 mostra o algoritmo acima em linguagem de máquina.

São carregados os valores imediatos 05 e 10 para os registradores de endereço 0 e 2, para que as operações ADDI, faz-se a verificação de *branch*, pula para a instrução de endereço 10 (SUBI), faz-se novamente a verificação de *branch*, mas agora verificando se é diferente. A Figura 21 mostra a Forma de Onda para instruções do tipo I em tempo contínuo, a Figura 14 mostra a Forma de Onda de cada instrução.

Na Figura 14, a instrução de ADDI mostra o resultado 10 (ALU_RESULT), o que é o esperado, e ao chegar na instrução seguinte (BEQ), temos que o ALU_RESULT é

Figura 13 – Forma de Onda completa para instruções do tipo I

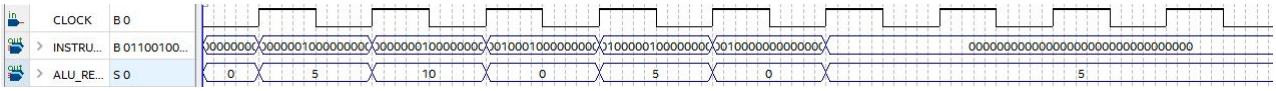
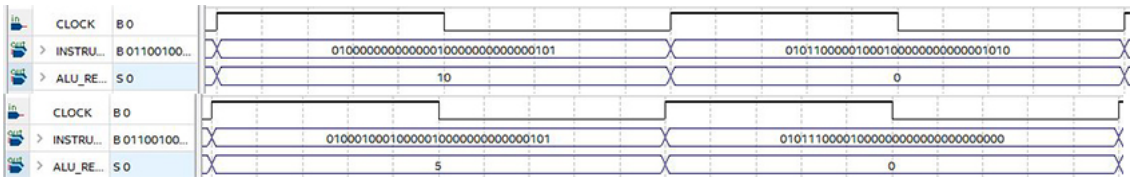


Figura 14 – Forma de Onda das Instruções do tipo I



0, ou seja, irá ocorrer o *branch*. O SUBI também funcionou corretamente e na Figura 21 vê-se que não ocorreu o segundo *branch*, pois o ALU_RESULT é igual a zero.

5.1.3 Testes de Instruções do tipo J

As instruções do tipo J são: J, HALT e NOP. Para o teste destas instruções foi utilizado o seguinte algoritmo:

- JUMP ENDEREÇO 5
- NOP
- HALT

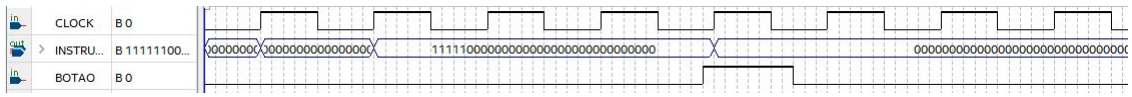
A Figura 22 mostra o algoritmo acima em linguagem de máquina.

As instruções NOP e HALT estão no endereço 5 e 6 da memória de instrução, respectivamente. O sistema deve simplesmente pular para a instrução no endereço 5 e entrar em estado de HALT. A simulação em forma de onda da Figura 15 representa as instruções do tipo J, em que não ocorreu o pressionamento do botão ENTER.

Figura 15 – Forma de Onda das Instruções do tipo J

Da Figura 15, vê-se que o sistema funcionou como o esperado, ao chegar na instrução HALT, deixou de operar. A Figura 16 mostra o sistema saindo do estado de parada devido a ocorrência do ENTER.

Figura 16 – Forma de Onda de Instruções do Tipo J com o pressionamento do botão ENTER



O ENTER é representado na Figura 16 como BOTAO.

5.1.4 Testes de Instruções do tipo I/O

As instruções do tipo I/O são: IPT, OPT, SW, LW e LOADI. Não foram testadas em formas de onda as instruções de IPT e OPT, pois é interessante ver o funcionamento delas direto na FPGA. A instrução LOADI foi indiretamente testada nos algoritmos de outros formatos e mostrou-se estar funcionando corretamente, então os testes serão focados nas instruções SW (*Store Word*) e LW (*Load Word*). O algoritmo de teste:

- LOADI REG(0) IMED 10;
- SW REG(0) ENDEREÇO 02;
- LW REG(1) ENDEREÇO 02;

A Figura 23 mostra o algoritmo acima em linguagem de máquina.

O algoritmo basicamente carrega um valor imediato 10 no registrador de endereço 0 e manda guarda na memória de dados no endereço 02 para logo após carregar este valor ao registrador de endereço 1. A forma de onda está presenta na Figura 23.

Figura 17 – Forma de Onda das instruções do tipo I/O

A SAIDA_RT_TEST representa a saída do registrador RT no Banco de Registradores. Durante a instrução de SW o dado do registrador 2 está sendo emitido corretamente como esperado. A MEMDADOS_SAIDA_TEST representa a saída de dados da Memória de Dados e durante a instrução de LW, está emitindo o dado do registrador de endereço da memória 2.

5.2 Testes de Algoritmos

Afim de demonstrar o funcionamento de NorMIPS, foram implementados dois algoritmos de teste: o Algoritmo de Fibonacci, que a partir de um número X dado como entrada, imprimirá nos *displays* até a posição Xº da sequência de Fibonacci, e um teste lógico simples, para testar as instruções não utilizadas na de Fibonacci.

O algoritmo de Fibonacci é o seguinte:

- IPT REG(0);
- LOADI REG(1) IMED 0;
- LOADI REG(2) IMED 0;

- LOADI REG(3) IMED 1;
- BEQ REG(0) REG(3) ENDEREÇO 12;
- SUBI REG(0) REG(0) IMED 1;
- ADD REG(2) REG(3) REG(4);
- MOV REG(3) REG(2);
- MOV REG(4) REG(3);
- ADDI REG(1) IMED 1;
- OPT REG(4);
- BNQ REG(0) REG(1) ENDEREÇO 6;
- JUMP ENDEREÇO 14;
- LOADI REG(4) IMED 1;
- SW REG(4) ENDEREÇO 1;
- LW REG(7) ENDEREÇO 7;
- OPT REG(7);

O algoritmo verifica se a entrada das alavancas tem valor 1, caso tenha ele irá direto para a instrução de impressão (que está organizada em SW, LW e OPT, para testar as instruções de *store* e *load*) e no *display* aparecerá 1. Caso seja maior que 1, ele entrará na lógica da sequência de Fibonacci (c = a + b; a = b; b = c) até o contador (representado como REG(1)) ser igual ao valor de entrada subtraído de 1 (desconsiderando o primeiro valor da sequência). Para os testes, a entrada foi de 1001 (9).

5.2.1 Testes em Forma de Onda

A simulação em Forma de Onda do algoritmo de Fibonacci pode ser visto na Figura 18.

Vê-se que na Figura 18 não dá para distinguir bem os estados de cada saída. Isso ocorreu devido a frequência colocada no *clock* ser de 20ns, porém, dá para perceber a variação das saídas que ligam o *display*, quando ocorre a instrução OPT. As fotos da FPGA na próxima seção dá para melhor se ver o funcionamento do algoritmo.

Figura 18 – Forma de Onda do algoritmo de Fibonacci

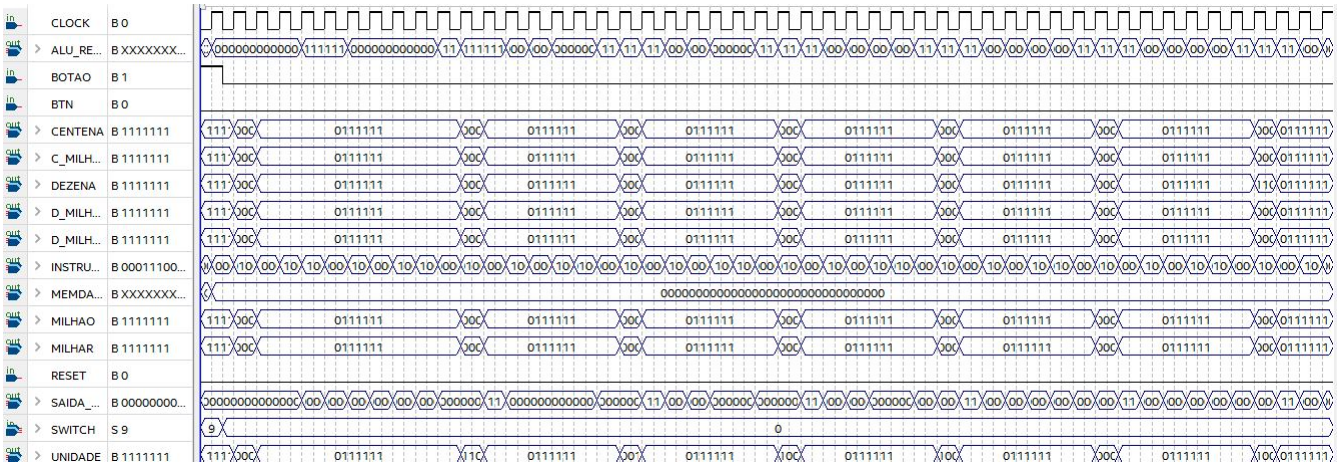
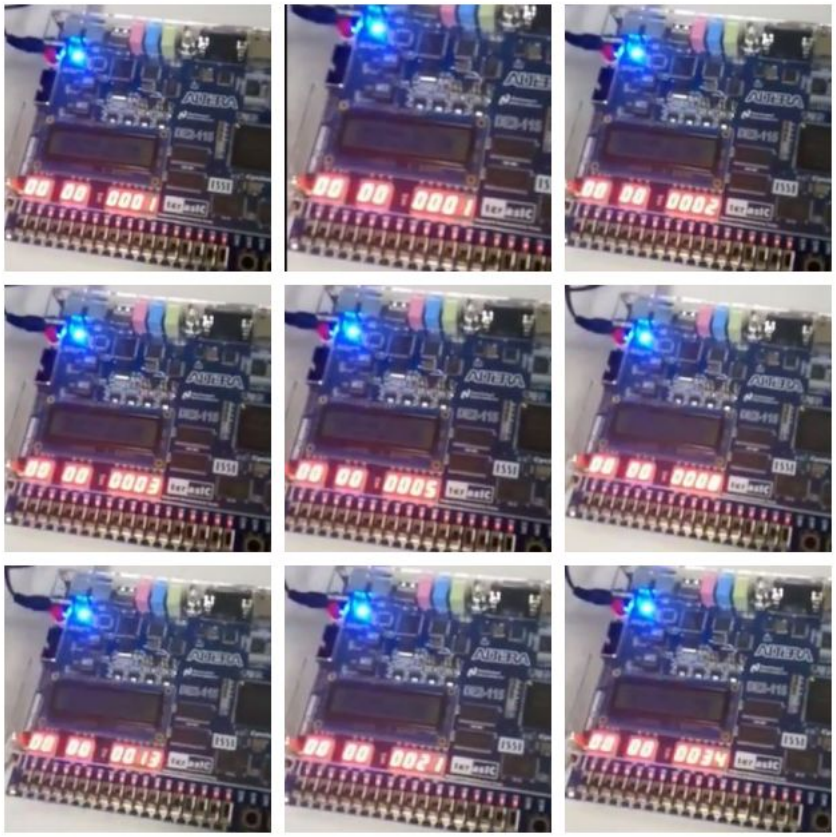


Figura 19 – Fotografias da FPGA mostrando o funcionamento do Teste de Fibonacci



5.2.2 Testes na FPGA

O funcionamento da FPGA para o algoritmo de Fibonacci pode ser visto na Figura 19.

Da Figura 19, vê-se que na nona foto foi mostrado o 9º número da sequência de Fibonacci. Já que definimos que na instrução IPT que, pelas alavancas, o valor 9, foi ocorrido como esperado. Este último teste, demonstrou o funcionamento corretamente do processador com todos os módulos (Unidade de Processamento, Unidade de Controle e

Entrada-Saída) conectados.

6 Conclusão e Considerações Finais

O objetivo da construção de um processador funcional em lógica programável e funcionando em um FPGA foi alcançado com sucesso. Durante o semestre, muitos desafios referentes ao desenvolvimento do projeto foram encontrados e resolvidos, ocasionando um amadurecimento da ideia do que é ser um Engenheiro da Computação e os problemas enfrentados por ele.

O conhecimento obtido pela teoria sobre o que é um processador e como ele funciona, se tornou algo diminuto com o conhecimento obtido pela prática. As dificuldades em dar um ponta pé no desenvolvimento e a inexperiência com lógica programável para este nível de projeto, foram as barreiras mais difíceis encontradas. Porém, com o auxílio do professor e da monitoria, estes problemas foram sanados. A integração do processador em Verilog com o dispositivo FPGA não funcionou como o esperado na primeira vez, mas um problema que foi relativamente resolvido de maneira rápida, com uma melhor elaboração do módulo de entrada-saída.

O processador NorMIPS ainda tem muito a melhorar. Ele foi desenvolvido pensando no básico e funcional, para que problemas futuros não sejam oriundos do desenvolvido neste laboratório. Seus próximos passos são a implementação de mais instruções e, consequentemente, aumentar a complexidade do processador. Além disso, com o decorrer dos laboratórios da graduação, o processador será modificado e aprimorado.

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 UNIDADE de Controle. 2012. Disponível em: <https://pt.wikipedia.org/wiki/Unidade_de_controle>. Citado na página 15.

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

7 DEBOUNCE Verilog. 2013. Disponível em: <https://eewiki.net/pages/viewpage.action?pageId=13599139>. Citado na página 36.

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 [4:0] saida;

        always @ (*) begin

                case(controle)
                        1'b0: saida = entrada_RT;
                        1'b1: saida = entrada_RD;
                endcase
        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

                case(controle)
                        1'b0: saida = entrada_dadoRT;
                        1'b1: saida = entrada_extendido;
                endcase
        end
endmodule
```

```
module MUX_3 (saida_memoria, resultado_alu, extendido, saida, controle);

        // input
        input [31:0] saida_memoria;
        input [31:0] resultado_alu;
        input [31:0] extendido;
        input [1:0] controle;

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

        always @ (*) begin

```

```
14                case(controle)
15                        default: saida = resultado_alu;
16                        2'b01: saida = saida_memoria;
17                        2'b10: saida = extendido;
18
19                endcase
20        end
21endmodule
```

```
1 module MUX_4 (entrada_pc, entrada_extendido, entrada_jump, saida, controle, sinal_ZERO);
2
3         // input
4         input [31:0] entrada_pc;
5         input [31:0] entrada_extendido;
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 ON EQUAL
19                                if(sinal_ZERO == 1) begin
20                                        saida = entrada_extendido;
21                                end
22                                else begin
23                                        saida = entrada_pc + 1; // PC + 1
24                                end
25                        end
26                        2'b10: begin // BRANCH NOT EQUAL
27                                if(sinal_ZERO == 0) begin
28                                        saida = entrada_extendido;
29                                end
30                                else begin
31                                        saida = entrada_pc + 1; // PC + 1
32                                end
33                        end
34                        2'b11: begin // JUMP
35                                saida = entrada_jump;
36                        end
37                endcase
38        end
39
40endmodule
```

```
1 module MUX_6 (instrucao, switch, saida, controle /*Controle MUX6*/);
2
3         // input
4         input [15:0] instrucao;
5         input [15:0] switch;
6         input controle;
7
8         // output
9         output reg [15:0] saida;
10
```

```
11          always @ (*) begin
12
13                  case(controle)
14                          1'b0: saida = instrucao;
15                          1'b1: saida = switch;
16                  endcase
17          end
18  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'b0000000000000000, entrada};
10  endmodule
```

```
1   module extensor26_28 (entrada, saida);
2
3           // input
4           input [25:0] entrada;
5
6           // output
7           output [31:0] saida;
8
9           assign saida = {6'b0, entrada};
10  endmodule
```

```
1   // DeBounce_v.v
2
3
4   ////////////////////////// Button Debounceer ///////////////////////////////////////
5   //*********************************************************************
6   // FileName: DeBounce_v.v
7   // FPGA: MachXO2 7000HE
8   // IDE: Diamond 2.0.1
9   //
10  // HDL IS PROVIDED "AS IS." DIGI-KEY EXPRESSLY DISCLAIMS ANY
11  // WARRANTY OF ANY KIND, WHETHER EXPRESS OR IMPLIED, INCLUDING BUT NOT
12  // LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
13  // PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL DIGI-KEY
14  // BE LIABLE FOR ANY INCIDENTAL, SPECIAL, INDIRECT OR CONSEQUENTIAL
15  // DAMAGES, LOST PROFITS OR LOST DATA, HARM TO YOUR EQUIPMENT, COST OF
16  // PROCUREMENT OF SUBSTITUTE GOODS, TECHNOLOGY OR SERVICES, ANY CLAIMS
17  // BY THIRD PARTIES (INCLUDING BUT NOT LIMITED TO ANY DEFENSE THEREOF),
18  // ANY CLAIMS FOR INDEMNITY OR CONTRIBUTION, OR OTHER SIMILAR COSTS.
19  // DIGI-KEY ALSO DISCLAIMS ANY LIABILITY FOR PATENT OR COPYRIGHT
20  // INFRINGEMENT.
21  //
22  // Version History
23  // Version 1.0 04/11/2013 Tony Storey
24  // Initial Public Release
25  // Small Footprint Button Debouncer
26
27  `timescale 1 ns / 100 ps
28  module  DeBounce
29          (
```

```
30          input                   clk, n_reset, button_in,
                // inputs
31          output reg      DB_out
                                                    // output
32          );
33  //// ---------------- internal constants --------------
34          parameter N = 11 ;              // (2^ (21-1) )/ 38 MHz = 32 ms debounce time
35  ////---------------- internal variables ---------------
36          reg  [N-1 : 0]  q_reg;                                                  // timing
                 regs
37          reg  [N-1 : 0]  q_next;
38          reg DFF1, DFF2;
                // input flip-flops
39          wire q_add;
                            // control flags
40          wire q_reset;
41  //// -------------------------------------------------------
42
43  ////contenious assignment for counter control
44          assign q_reset = (DFF1  ^ DFF2);                // xor input flip flops to look
                for level chage to reset counter
45          assign  q_add = ~(q_reg[N-1]);                  // add to counter when q_reg msb
                is equal to 0
46
47  //// combo counter to manage q_next
48          always @ ( q_reset, q_add, q_reg)
49                  begin
50                          case( {q_reset , q_add})
51                                  2'b00 :
52                                                  q_next <= q_reg;
53                                  2'b01 :
54                                                  q_next <= q_reg + 1;
55                                  default :
56                                                  q_next <= { N {1'b0} };
57                          endcase
58                  end
59
60  //// Flip flop inputs and q_reg update
61          always @ ( posedge clk )
62                  begin
63                          if(n_reset ==  1'b0)
64                                  begin
65                                          DFF1 <= 1'b0;
66                                          DFF2 <= 1'b0;
67                                          q_reg <= { N {1'b0} };
68                                  end
69                          else
70                                  begin
71                                          DFF1 <= button_in;
72                                          DFF2 <= DFF1;
73                                          q_reg <= q_next;
74                                  end
75                  end
76
77  //// counter control
78          always @ ( posedge clk )
79                  begin
80                          if(q_reg[N-1] == 1'b1)
81                                          DB_out <= DFF2;
82                          else
```

```
83                                          DB_out <= DB_out;
84                  end
85
86          endmodule
```

APÊNDICE B – Códigos Auxiliares da Fase de Testes

Figura 20 – Algoritmo para teste do Tipo R

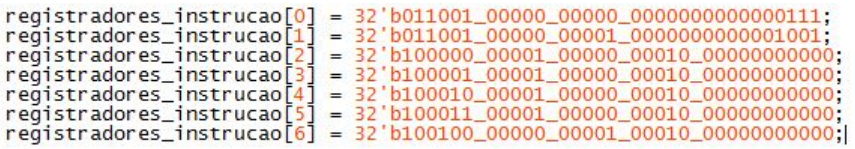


Figura 21 – Algoritmo para teste do tipo I

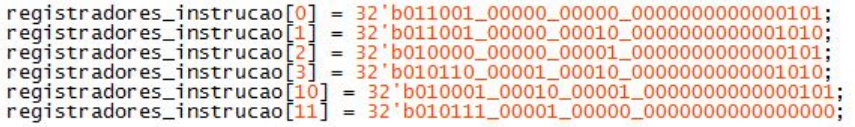


Figura 22 – Algoritmo para teste do tipo J

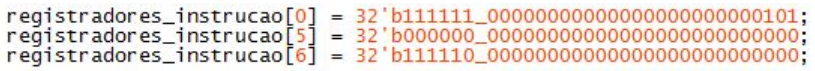
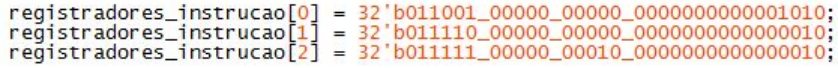


Figura 23 – Algoritmo para teste do tipo I/O



APÊNDICE C – Código completo da Unidade de Controle

```
module unidadeControle(opcode, botao, clock, controle_BANCOREG, controle_MEMDADOS,
    controle_MUX1, controle_MUX2, controle_MUX3, controle_MUX4, controle_ALU, HALT,
    controle_MUX6, controle_OPT);

        input [5:0] opcode;
        input botao;
        input clock;

        output reg [1:0] controle_MUX4;
        output reg [1:0] controle_MUX3;
        output reg [2:0] controle_ALU;
        output reg controle_BANCOREG, controle_MEMDADOS, controle_MUX1, controle_MUX2,
            controle_MUX6, controle_OPT, HALT;

        always @ (opcode or botao) begin

                case(opcode)

                        // ENTRADA, SAIDA E HALT
                        6'b000111: begin // in
                                if(botao) begin
                                        HALT = 0;
                                        controle_MUX4 = 2'b00;
                                end
                                else if (clock) begin
                                        controle_BANCOREG = 1'b1;
                                        controle_ALU = 3'bxxx;
                                        controle_MEMDADOS = 1'b0;
                                        controle_MUX1 = 1'b0;
                                        controle_MUX2 = 1'bx;
                                        controle_MUX3 = 2'b10;
                                        controle_MUX4 = 2'b00;
                                        controle_MUX6 = 1'b1;
                                        controle_OPT = 1'b0;
                                        HALT = 1;
                                end

                        end
                        6'b111000: begin // out
                                controle_BANCOREG = 1'b0;
                                controle_ALU = 3'bxxx;
                                controle_MEMDADOS = 1'b0;
                                controle_MUX1 = 1'bx;
                                controle_MUX2 = 1'bx;
                                controle_MUX3 = 2'bxx;
                                controle_MUX4 = 2'b00;
                                controle_MUX6 = 1'bx;
                                controle_OPT = 1'b1;
                                HALT = 0;
                        end
                        6'b000000: begin // nop
```

```
49                              controle_BANCOREG = 1'b0;
50                              controle_ALU = 3'bxxx;
51                              controle_MEMDADOS = 1'b0;
52                              controle_MUX1 = 1'bx;
53                              controle_MUX2 = 1'bx;
54                              controle_MUX3 = 2'bxx;
55                              controle_MUX4 = 2'b00;
56                              controle_MUX6 = 1'bx;
57                              controle_OPT = 1'b0;
58                              HALT = 0;
59                      end
60                      6'b111110: begin // HALT
61                              if(botao) begin
62                                      HALT = 0;
63                                      controle_MUX4 = 2'b00;
64                              end
65                              else if (clock) begin
66                                      controle_BANCOREG = 1'b0;
67                                      controle_ALU = 3'bxxx;
68                                      controle_MEMDADOS = 1'b0;
69                                      controle_MUX1 = 1'bx;
70                                      controle_MUX2 = 1'bx;
71                                      controle_MUX3 = 2'bxx;
72                                      controle_MUX4 = 2'b11;
73                                      controle_MUX6 = 1'bx;
74                                      controle_OPT = 1'b0;
75                                      HALT = 1;
76                              end
77                      end
78
79                      // OPERACOES
80                      6'b111111: begin // jump
81                              controle_BANCOREG = 1'b0;
82                              controle_ALU = 3'bxxx;
83                              controle_MEMDADOS = 1'b0;
84                              controle_MUX1 = 1'bx;
85                              controle_MUX2 = 1'bx;
86                              controle_MUX3 = 2'bxx;
87                              controle_MUX4 = 2'b11;
88                              controle_MUX6 = 1'bx;
89                              controle_OPT = 1'b0;
90                              HALT = 0;
91                      end
92                      6'b100000: begin // add
93                              controle_BANCOREG = 1'b1;
94                              controle_ALU = 3'b000;
95                              controle_MEMDADOS = 1'b0;
96                              controle_MUX1 = 1'b1;
97                              controle_MUX2 = 1'b0;
98                              controle_MUX3 = 2'b00;
99                              controle_MUX4 = 2'b00;
100                             controle_MUX6 = 1'b0;
101                             controle_OPT = 1'b0;
102                             HALT = 0;
103                     end
104                     6'b010000: begin // addi
105                             controle_BANCOREG = 1'b1;
106                             controle_ALU = 3'b000;
107                             controle_MEMDADOS = 1'b0;
108                             controle_MUX1 = 1'b0;
```

```
109                             controle_MUX2 = 1'b1;
110                             controle_MUX3 = 2'b00;
111                             controle_MUX4 = 2'b00;
112                             controle_MUX6 = 1'b0;
113                             controle_OPT = 1'b0;
114                             HALT = 0;
115                         end
116                         6'b100001: begin // sub
117                             controle_BANCOREG = 1'b1;
118                             controle_ALU = 3'b001;
119                             controle_MEMDADOS = 1'b0;
120                             controle_MUX1 = 1'b1;
121                             controle_MUX2 = 1'b0;
122                             controle_MUX3 = 2'b00;
123                             controle_MUX4 = 2'b00;
124                             controle_MUX6 = 1'b0;
125                             controle_OPT = 1'b0;
126                             HALT = 0;
127                         end
128                         6'b010001: begin // subi
129                             controle_BANCOREG = 1'b1;
130                             controle_ALU = 3'b001;
131                             controle_MEMDADOS = 1'b0;
132                             controle_MUX1 = 1'b0;
133                             controle_MUX2 = 1'b1;
134                             controle_MUX3 = 2'b00;
135                             controle_MUX4 = 2'b00;
136                             controle_MUX6 = 1'b0;
137                             controle_OPT = 1'b0;
138                             HALT = 0;
139                         end
140                         6'b100010: begin // and
141                             controle_BANCOREG = 1'b1;
142                             controle_ALU = 3'b010;
143                             controle_MEMDADOS = 1'b0;
144                             controle_MUX1 = 1'b1;
145                             controle_MUX2 = 1'b0;
146                             controle_MUX3 = 2'b00;
147                             controle_MUX4 = 2'b00;
148                             controle_MUX6 = 1'b0;
149                             controle_OPT = 1'b0;
150                             HALT = 0;
151                         end
152
153                         6'b100011: begin // or
154                             controle_BANCOREG = 1'b1;
155                             controle_ALU = 3'b011;
156                             controle_MEMDADOS = 1'b0;
157                             controle_MUX1 = 1'b1;
158                             controle_MUX2 = 1'b0;
159                             controle_MUX3 = 2'b00;
160                             controle_MUX4 = 2'b00;
161                             controle_MUX6 = 1'b0;
162                             controle_OPT = 1'b0;
163                             HALT = 0;
164                         end
165                         6'b100100: begin // slt
166                             controle_BANCOREG = 1'b1;
167                             controle_ALU = 3'b110;
168                             controle_MEMDADOS = 1'b0;
```

```
169                             controle_MUX1 = 1'b1;
170                             controle_MUX2 = 1'b0;
171                             controle_MUX3 = 2'b00;
172                             controle_MUX4 = 2'b00;
173                             controle_MUX6 = 1'b0;
174                             controle_OPT = 1'b0;
175                             HALT = 0;
176                         end
177                         6'b010100: begin // slti
178                             controle_BANCOREG = 1'b1;
179                             controle_ALU = 3'b110;
180                             controle_MEMDADOS = 1'b0;
181                             controle_MUX1 = 1'b0;
182                             controle_MUX2 = 1'b1;
183                             controle_MUX3 = 2'b00;
184                             controle_MUX4 = 2'b00;
185                             controle_MUX6 = 1'b0;
186                             controle_OPT = 1'b0;
187                             HALT = 0;
188                         end
189                         6'b100101: begin // mov
190                             controle_BANCOREG = 1'b1;
191                             controle_ALU = 3'b100;
192                             controle_MEMDADOS = 1'b0;
193                             controle_MUX1 = 1'b0;
194                             controle_MUX2 = 1'b0;
195                             controle_MUX3 = 2'b00;
196                             controle_MUX4 = 2'b00;
197                             controle_MUX6 = 1'b0;
198                             controle_OPT = 1'b0;
199                             HALT = 0;
200                         end
201                         6'b011111: begin // lw
202                             controle_BANCOREG = 1'b1;
203                             controle_ALU = 3'b000;
204                             controle_MEMDADOS = 1'b0;
205                             controle_MUX1 = 1'b0;
206                             controle_MUX2 = 1'b1;
207                             controle_MUX3 = 2'b01;
208                             controle_MUX4 = 2'b00;
209                             controle_MUX6 = 1'b0;
210                             controle_OPT = 1'b0;
211                             HALT = 0;
212                         end
213                         6'b011110: begin // sw
214                             controle_BANCOREG = 1'b0;
215                             controle_ALU = 3'bxxx;
216                             controle_MEMDADOS = 1'b1;
217                             controle_MUX1 = 1'bx;
218                             controle_MUX2 = 1'bx;
219                             controle_MUX3 = 2'bxx;
220                             controle_MUX4 = 2'b00;
221                             controle_MUX6 = 1'b0;
222                             controle_OPT = 1'b0;
223                             HALT = 0;
224                         end
225                         6'b010110: begin // beq
226                             controle_BANCOREG = 1'b0;
227                             controle_ALU = 3'b001;
228                             controle_MEMDADOS = 1'b0;
```

```
229                                 controle_MUX1 = 1'bx;
230                                 controle_MUX2 = 1'b0;
231                                 controle_MUX3 = 2'bxx;
232                                 controle_MUX4 = 2'b01;
233                                 controle_MUX6 = 1'b0;
234                                 controle_OPT = 1'b0;
235                                 HALT = 0;
236                         end
237                         6'b010111: begin // bnq
238                                 controle_BANCOREG = 1'b0;
239                                 controle_ALU = 3'b001;
240                                 controle_MEMDADOS = 1'b0;
241                                 controle_MUX1 = 1'bx;
242                                 controle_MUX2 = 1'b0;
243                                 controle_MUX3 = 2'bxx;
244                                 controle_MUX4 = 2'b10;
245                                 controle_MUX6 = 1'b0;
246                                 controle_OPT = 1'b0;
247                                 HALT = 0;
248                         end
249                         6'b011001: begin // loadi
250                                 controle_BANCOREG = 1'b1;
251                                 controle_ALU = 3'bxxx;
252                                 controle_MEMDADOS = 1'b0;
253                                 controle_MUX1 = 1'b0;
254                                 controle_MUX2 = 1'bx;
255                                 controle_MUX3 = 2'b10;
256                                 controle_MUX4 = 2'b00;
257                                 controle_MUX6 = 1'b0;
258                                 controle_OPT = 1'b0;
259                                 HALT = 0;
260                         end
261                 endcase
262         end
263
264 endmodule
```