Andrew Medeiros de Campos
RA: 111775

Implementação em Verilog de uma unidade de processamento RISC multiciclo

(Laboratório de sistemas computacionais: Arquitetura e Organização de Computadores)

São José dos Campos - Brasil

Maio de 2019

Andrew Medeiros de Campos
RA: 111775

Implementação em Verilog de uma unidade de processamento RISC multiciclo

(Laboratório de sistemas computacionais: Arquitetura e Organização de Computadores)

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

Docente: Prof. Dr. Fábio Augusto Menocci Cappabianco

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Maio de 2019

Resumo

No relatório em questão constam as ideias trabalhadas no projeto da disciplina Laboratório de Arquitetura e Organização de Computadores, onde a proposta foi desenvolver a arquitetura e a organização de um processador, contando com conjunto de instruções, modos de endereçamento e caminho de dados do mesmo. O processador será baseado no principio de processamento multiciclo, contendo um conjunto de instruções RISC e arquitetura Von Neumann, ou seja, apenas uma unidade de memória para manter tanto os dados quanto as instruções, usando como base a arquitetura MIPS (*Microprocessor without Interloked Pipeline Stages*). Neste relatório serão apresentados, além da fundamentação teórica, todo o desenvolvimento e projeto descrevendo o funcionamento do processador.

Palavras-chaves: Processador. Computadores. Instruções. Arquitetura. Organização. RISC. Von Neumann. Multiciclo.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

Desde a criação do primeiro computador à válvulas em 1946, equipamentos eletrônicos tem sido cada vez mais presentes no dia a dia das pessoas, seja em forma de computadores de mesa, de telefones celulares ou até como dispositivos mais simples como calculadoras. A maioria dos eletrônicos que temos contato contam com uma unidade de processamento central, seja ela um microcontrolador ou um microprocessador, que realiza todos os cálculos necessários para o funcionamento do dispositivo. À partir disso pode-se notar que as unidades de processamento são os componentes mais valiosos presentes em um dispositivo eletrônico, uma vez que as mesmas são as principais responsaveis por diferenciar o funcionamento dos equipamento por serem as responsáveis pelo processamento dos dados de entrada e saída.

O estudo de unidades de processamento é extremamente importante, uma vez que estamos em contato com eles frequentemente em nosso dia a dia, e nossa realidade atual sem eles seria inimaginável. Sistemas computacionas cada vez mais se espalham pelos diferentes setores da sociedade, tanto com a finalidade de aumentar o conforto e a praticidade dos usuários, quanto como centrais multimída em automóveis e sistemas embarcados presentes em eletrodomésticos que se conectam à Internet, ou seja, para automatizar processos, aumentar a precisão em operações delicadas, entre outras diversas aplicações, como braços robóticos de montadoras e robôs-cirurgiões que auxiliam médicos em operações.

Visto toda a necessidade de processadores no dia a dia, esse projeto visa estudar e desenvolver uma unidade de processamento completa contando com memória de dados e instruções, registradores parar operações, unidades lógicas e aritméticas e módulos de entrada e saída. Para o desenvolvimento do projeto foi optado por uma abordagem multiciclo devivo à menor quantidade de *hardware* necessária em comparação com a abordagem monociclo, visto a limitação de memória dos *kits* FPGA's, porém sem perda de capacidade de processamento e com um desempenho ainda melhor comparado também ao monociclo. Além disso com a abordagem multiciclo é possível optar pela arquitetura Von Neumann para a memória de instrução e de dados, que será explicada mais a frente. Será utilizado como base o modelo de processador multiciclo MIPS de 32 bits.

2 Objetivos

2.1 Objetivos Gerais

A proposta do projeto descrito neste relatório é desenvolver o esquema de funcionamento de uma unidade de processamento completa e implementá-la na linguagem de descrição de *hardware* Verilog, e futuramente testada-la em um *kit* FPGA. À principio apenas o caminho de dados básico será apresentado, juntamente com os modos de endereçamento, o conjunto inicial de instruções e os formatos de instrução, além do código Verilog de cada módulo. A unidade de controle ainda não será desenvolvida pois será incluida em uma próxima etapa do projeto.

2.2 Objetivos Específicos

Para alcançar o objetivo geral, sete etapas devem ser concluídas. As mesmas estão citadas abaixo:

- Idealizar o diagrama do caminho de dados do processador,
- Estabelecer os tipos de instrução e seu formato,
- Elaborar o conjunto de instruções com as instruções básicas para o funcionamento,
- Escolher os modos de endereçamento que serão utilizados para alcançar os dados de memória,
- Decidir sobre o modo de transferência do módulo de entrada e saída,
- À partir do *datapath* estabelecido desenvolver os módulos necessários em Verilog e
- Unir todos os módulos em um arquivo final.

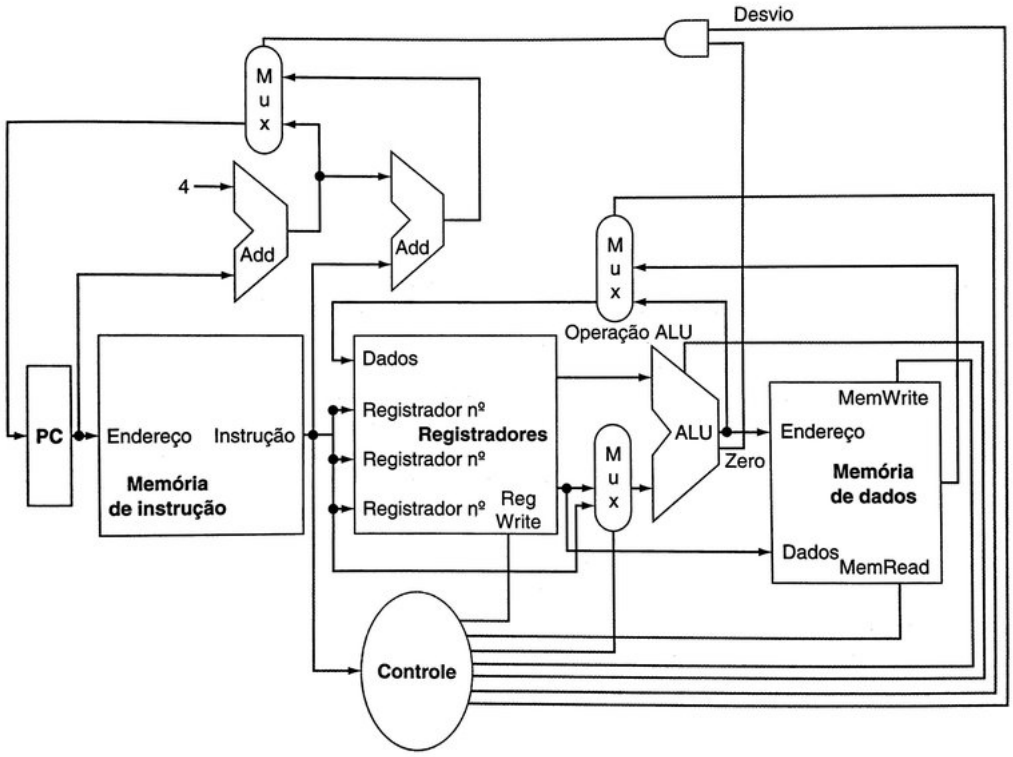
3 Fundamentação Teórica

Esta seção abordará os conceitos utilizados para o desenvolvimento do projeto apresentando uma síntese sobre unidades de processamento, explicações sobre as diferenças entres as arquiteturas (Monociclo e Multiciclo, Harvard e Von Neumann, etc.), os modos de endereçamento, os tipos de instrução e a arquitetura MIPS multiciclo 32 bits.

3.1 Processadores

A unidade de processamento nada mais é do que o cérebro de qualquer computador já que é a responsável por coletar todos os dados recebidos dos dispositivos de entrada e transforma-los em alguma forma de saída. Todo processador conta com algumas unidades básicas para o seu funcionamento sendo elas a um *Program Counter*, unidades de memória, unidades lógicas e aritméticas, registradores de dados e uma unidade de controle. Na Figura 1 pode ser visto um esquemático básico de um processador MIPS com as quatro estruturas citadas acima, entre algumas outras.

Figura 1 – Diagrama Básico de um Processador MIPS



Fonte: *Organização e Projeto de Computadores*(1)

Seguindo a Figura, é possível explicar um por um dos módulos básicos para o funcionamento de um processador, sendo o *Program Counter* o que pode ser analizado primeiro. O mesmo tem a função de guardar o endereço da instrução que está sendo processada, ou seja, nada mais é do que um registrador único com o propósito de guiar o processamento. Logo após o *program counter* geralmente pode-se encontrar uma unidade mémória, onde serão guardados todas as instruções ou alguns dados menos utilizados no processamento, que no caso do esquemático MIPS da Figura 1 é apenas uma das duas unidades presentes. O próximo módulo é o banco de registradores. Esse módulo se trata de um aglomerado de registradores que guardam resultados de operações recém realizadas ou operandos frequentemente utilizados, uma vez que o acesso ao banco de registradores é muito mais rápido do que o acesso à memória de dados devido ao seu tamanho reduzido. Em seguida temos uma Unidade Lógica e Aritimética (ou ALU - *Arithmetic Logic Unit*) que é a responsável pela realização de todas as operações do processador, uma vez que conta com circuitos para resolver tanto operações aritiméticas como soma e subtração, quanto operações lógicas como *and, or* e *not*.

Por último, a unidade de controle pode ser vista na parte de baixo da figura. Essa unidade é a mais importante de todo o processador pois ela comanda todos os outros módulos através de seus sinais de controle, guiando a escrita e a leitura dos registradores, escolhendo se devem guardar informações ou apenas exibir a informação já retida, instruindo a ALU para qual tipo de operação deve ser feita e orientando o PC se deve ou não mudar para a próxima instrução.

3.2 Conjunto de Instruções

Após explicado sobre a organização de um processador é necessário entender sua arquitetura. A arquitetura de um processador nada mais é do que os aspectos relacionados diretamente à implementação de programas e algoritmos, como o conjunto de instruções que o mesmo pode realizar, contando com as operações, os desvios, instruções de controle, entre outras. O conjunto de instruções define o que o processador pode ou não fazer e cada instrução dá as informações necessárias para as operações serem concluidas, como o endereço dos dados dos operandos, endereço de desvio e o código da operação.

A arquitetura de processadores pode ser dividida em duas categorias: a arquitetura RISC (*Reduced Instruction Set Computing* e a arquitetura CISC (*Complex Instruction Set Computing*). A grande diferença entre os dois tipos de arquitetura é a eficiência, uma vez que a arquitetura RISC tem um desempenho maior, porém conta com menos instruções e menos formatos de instruções comparada à arquitetura CISC. Atualmente visando o ganho de desempenho e a facilidade de implementação de estruturas externas como compiladores, os processadores contam com um conjunto de instruções híbirdo, tendo uma unidade de

processamento que apenas interpreta instruções simples e uma unidade de conversão que transforma instruções complexas em várias instruções simples que podem ser interpretadas pelo processador.

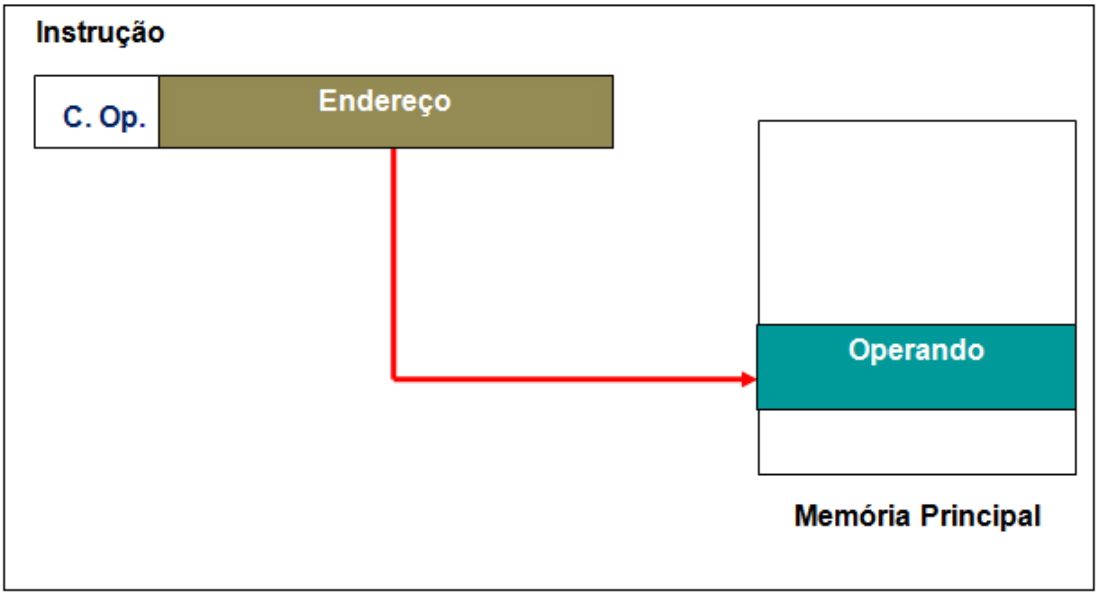
3.3 Modos de Endereçamento

Alguns formatos de instrução contam com um ou mais campos referente à endereços de dados, campos que às vezes não tem o tamanho necessário para abranger todas as possibilidades de endereços. Para resolver esse problema são necessários os diversos modos de endereçamento. Na arquitetura MIPS são utilizados seis tipos de endereçamento sendo eles imediato, direto, indireto, por registrador, indireto por registrador e por deslocamento.

A forma de endereçamento mais simples dentre todas as citadas acima é o endereçamento por imediato, onde o operando é dado diretamente na palavra de instrução. Esse método é comumente utilizado para definir valores de constantes, inicializar variáveis ou realizar operações com constantes e tem entre suas vantagens a facilidade de obtenção do valor, porém o campo do operando tem um tamanho muito limitado não possibilitando o uso de imediatos de valor muito alto.

O modo de endereçamento citado em seguida é o direto, cujo valor do operando se encontra na memória principal e seu endereço é dado diretamente na palavra de instrução como ilustrado na Figura 2. Entre suas vantagens pode-se citar a maior diponibilidade de bits para armazenar o valor do operando, uma vez que a memória principal tem a capacidade de armazenar valores maiores do que um campo de operando, porém o custo para buscar um dado nela é muito maior.

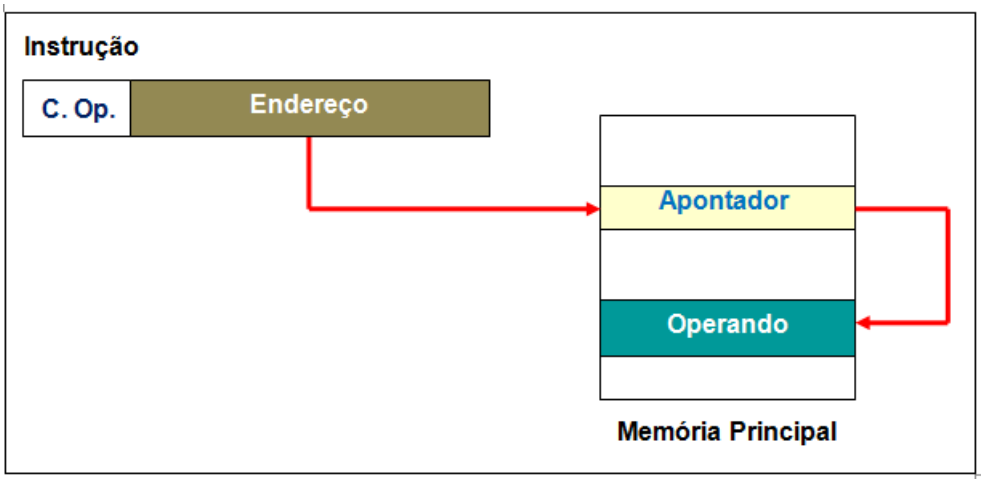
Figura 2 – Endereçamento Direto



Fonte: *Modos de Endereçamento e Conjunto de Instruções* (2)

Em seguida é citado o endereçamento indireto, onde o endereço do operando também se encontra na memória principal porém o endereço contido na palavra de instrução não é a localização do operando e sim a localização de um apontador que contém o endereço real do operando. A vantagem dessa abordagem é conseguir uma maior abrangência de endereços em que os valores de operandos podem ser armazenados uma vez que o campo de operando tem o tamanho menor do que o campo da memória, porém o custo de acesso do operando é ainda maior que o modo direto.

Figura 3 – Endereçamento Indireto

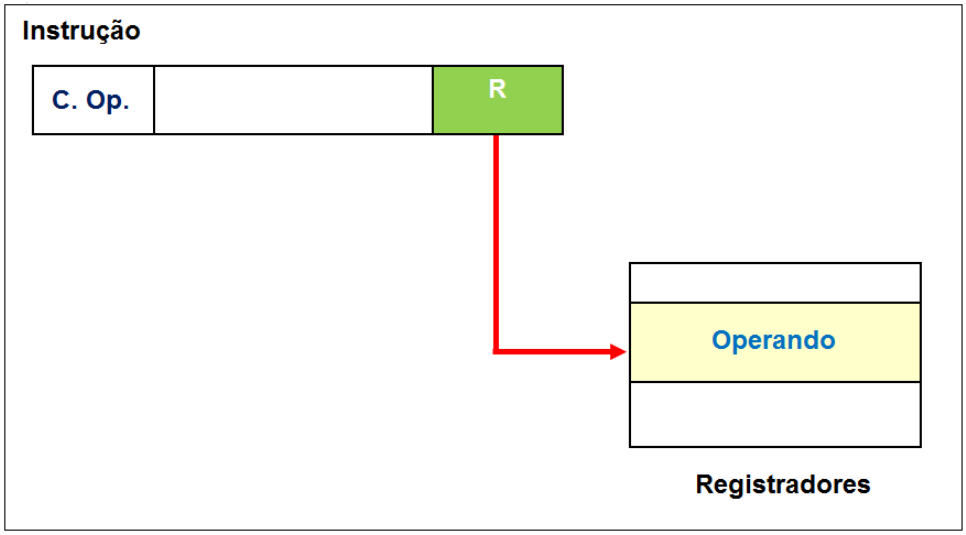


Fonte: *Modos de Endereçamento e Conjunto de Instruções* (2)

Saindo do âmbito da memória principal tem-se o endereçamento por registrador,

onde o operando é armazenado em um registrador do banco e seu endereço é contido em um campo específico da instrução. A vantagem desse modo de endereçamento em relação ao modo direto e indireto é a velocidade de acesso, uma vez que o custo de acessar um dado no banco de registradores é muito menor do que o custo de acessar um dado na memória principal. A desvantagem é que geralmente os bancos de registradores tem um tamanho muito menor do que o tamanho da memória principal, resultando em uma quantidade de dados armazenados menor.

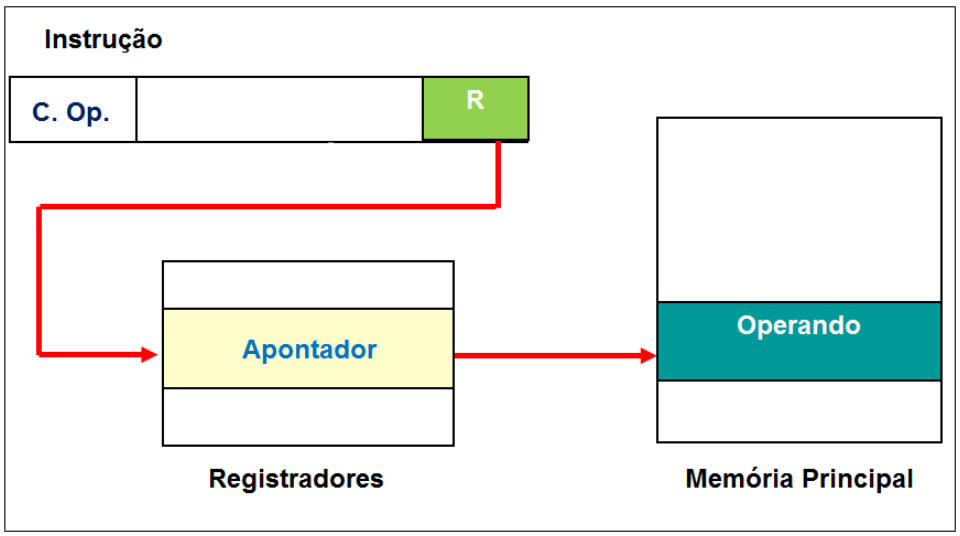
Figura 4 – Endereçamento por Registrador



Fonte: *Modos de Endereçamento e Conjunto de Instruções* (2)

O próximo modo de endereçamento é uma mistura entre o endereçamento indireto e o endereçamento por registradores. A diferença desse modo para o modo indireto é que o apontador nesse caso não está presente na memória principal e sim no banco de registradores. Esse modo de endereçamento tem as mesmas vantagens que o modo indireto, porém é um pouco menos custoso já que é necessário acessar a memória principal apenas uma vez, porém ainda é mais custoso que o endereçamento por registrador.

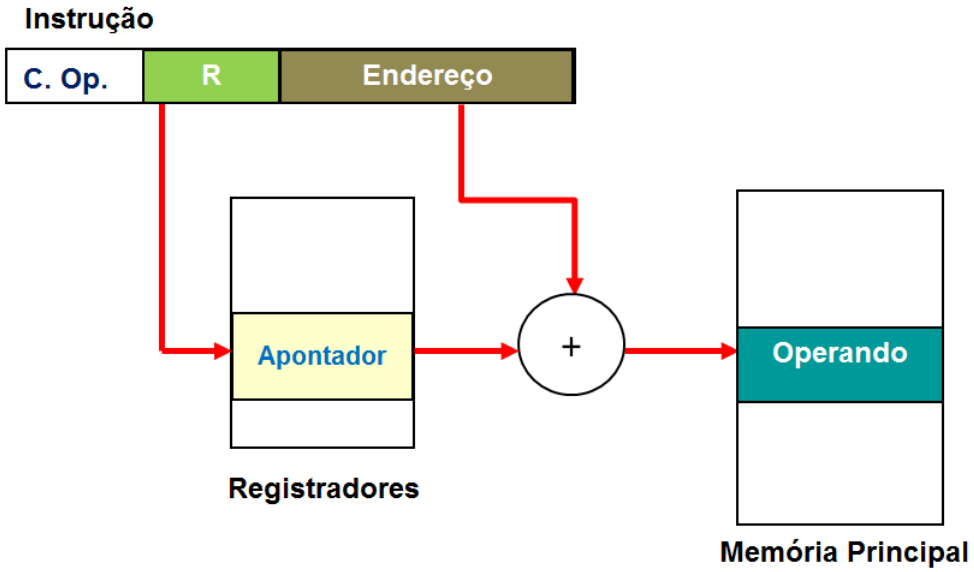
Figura 5 – Endereçamento Indireto por Registrador



Fonte: *Modos de Endereçamento e Conjunto de Instruções* (2)

O último modo de endereçamento citado é o endereçamento por deslocamento, que consiste no endereço de um registrador e um imediato na palavra de instrução sendo que o endereço real do operando na memória é dado pela soma entre o imediato e o valor armazenado no registrador. Esse tipo de endereçamento é o mais poderoso pois permite uma enorme gama de endereços que podem ser utilizados à um custo não tão alto como o endereçamento indireto.

Figura 6 – Endereçamento por Deslocamento



Fonte: *Modos de Endereçamento e Conjunto de Instruções* (2)

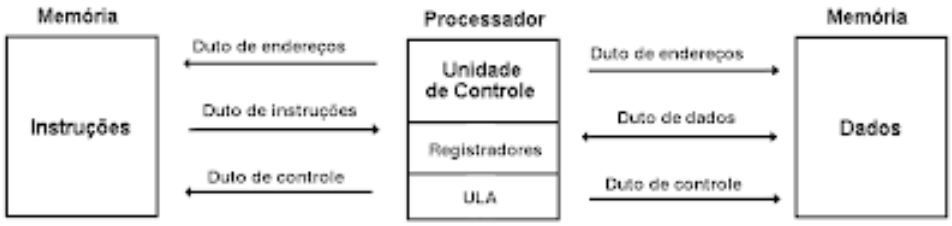
Por fim um outro modo de endereçamento não presente na estrutura original do

MIPS pode ser citado: o endereçamento por pilha, ou endereçamento implicito. Esse modo de endereçamento consiste em uma pilha de endereços onde para acessar o endereço do dado basta aplicar um comando *pop* na pilha. Esse modo de endereçamento é frequentemente utilizado em situações de recursão uma vez que existe uma ordem de acesso à endereços de instruções que pode ser simulada por uma pilha.

3.4 Harvard e Von Neumann

As arquituras Harvard e Von Neumann são simplificações da estrutura de processadores, modelos que unidades de processamento se baseiam para ser desenvolvidos. A grande diferença entre essas arquiteturas é o módulo de memória, que pode ser único e dividido entre dados e instruções, modelo usado pela arquitetutra Von Neumann, ou dividido em dois módulos sendo um a memória de instruções e o outro a memória de dados, usado pela arquitetura Harvard.

Figura 7 – Diagrama Arquitetura Harvard



Fonte: Site *Trabalho de A.C.: Arquitetura Harvard* (3)

Figura 8 – Diagrama Arquitetura Von Neumann

Fonte: Site *Trabalho de A.C.: Arquitetura Harvard* (3)

O modelo MIPS mostrado na Figura 1 é um exemplo de arquitetura Harvard, por ser composto por memória de instruções, banco de registradores, ALU e memória de dados, porém nada impede um modelo baseado na arquitetura MIPS usar o modo de memória de Von Neumann, são necessárias apenas alterações na estrutura do processador, como pode ser visto na Seção 3.5.

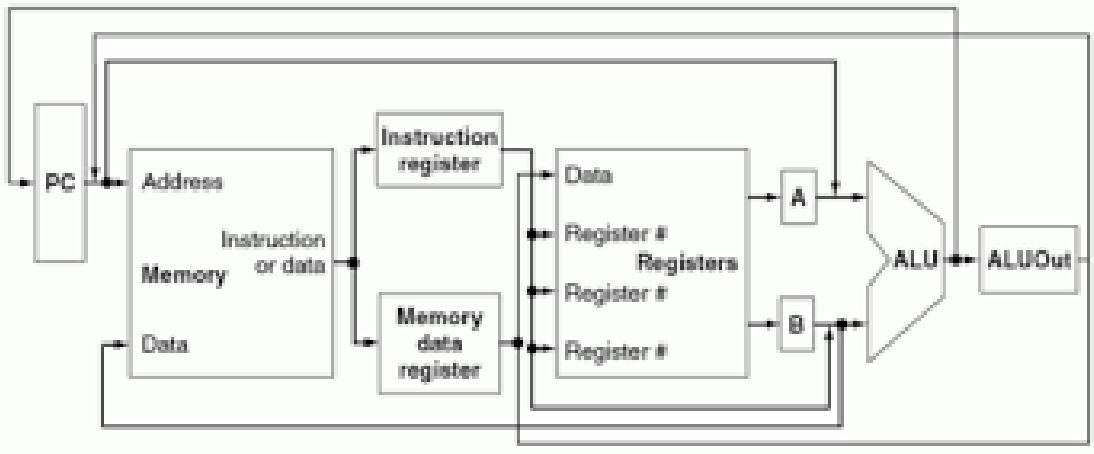
3.5 Monociclo e Multiciclo

A definição de processadores monociclo e multiciclo vem da quantidade de ciclos de *clock* necessários para se completar uma instrução. Processadores monociclo, como pode-se prever pelo nome, necessitam de apenas um ciclo de *clock* para processar completamente uma instrução enquanto processadores muticiclo precisam de dois ou mais ciclos. Um processador multiciclo não necessariamente completa todas as instruções com uma mesma quantidade de ciclos, ou seja, o mesmo pode ter uma instrução que necessita de três ciclos para ser completa enquato outra necessita de quatro ciclos.

Ao contrário do que parece, geralmente processadores multiciclo tem um desempenho melhor do que processadores monocilco, isso se da pelo fato de processadores monociclo terem o período do *clock* ditado pelo tempo necessário para o processamento da instrução mais lenta, enquanto em processadores multiciclo tem seu período de *clock* ditado pelo tempo de processamento de sua unidade mais lenta. Dessa forma comparando a mesma instrução entre os processadores é possível notar a melhora no tempo de processamento do processador multiciclo.

Outra vantagem que o multiciclo tem sobre o monociclo é a diminuição do número de módulos custosos, como por exemplo a memória. O processador MIPS mostrado na Figura 1 é um processaor monociclo baseado na arquitetura Harvard, com um módulo de memória para dados e outro para instruções. Isso ocorre devido ao fato dos das instruções serem acessadas no mesmo ciclo de *clock* dos dados, impossibilitando o uso de uma arquitetura do tipo Von Neumann. Porém no caso de um processador multiciclo esse problema não existe, uma vez que as instruções podem ser acessadas independentemente dos dados, mesmo usando a mesma estrutura para armazenar ambos os dados. Abaixo na Figura 9 é possível ver um esquemático básico de um processador do tipo MIPS multiciclo.

Figura 9 – Diagrama MIPS Multiciclo



Fonte: Organização e Projeto de Computadores(1)

Como pode ser visto na Figura acima, outra vantagem da abordagem multiciclo em comparação à monociclo é a quantidade reduzida de ALUs. Isso se deve ao fato da implementação multiciclo poder realizar todas as operações necessárias, tanto as operações ditadas pelas instruções quanto cálculos de desvio e endereço das próximas instruções.

As desvantagens do modelo multiciclo em relação ao monociclo é sua unidade de controle muito mais complexa comparada com a do monociclo e a necessidade de registradores intermediários entre os módulos do processador, já que é necessário guardar as informações processadas entre os ciclos de *clock*. Porém essa necessidade não influencia no desempenho do processador pelo fato dos registradores serem estruturas muito simples.

4 Desenvolvimento

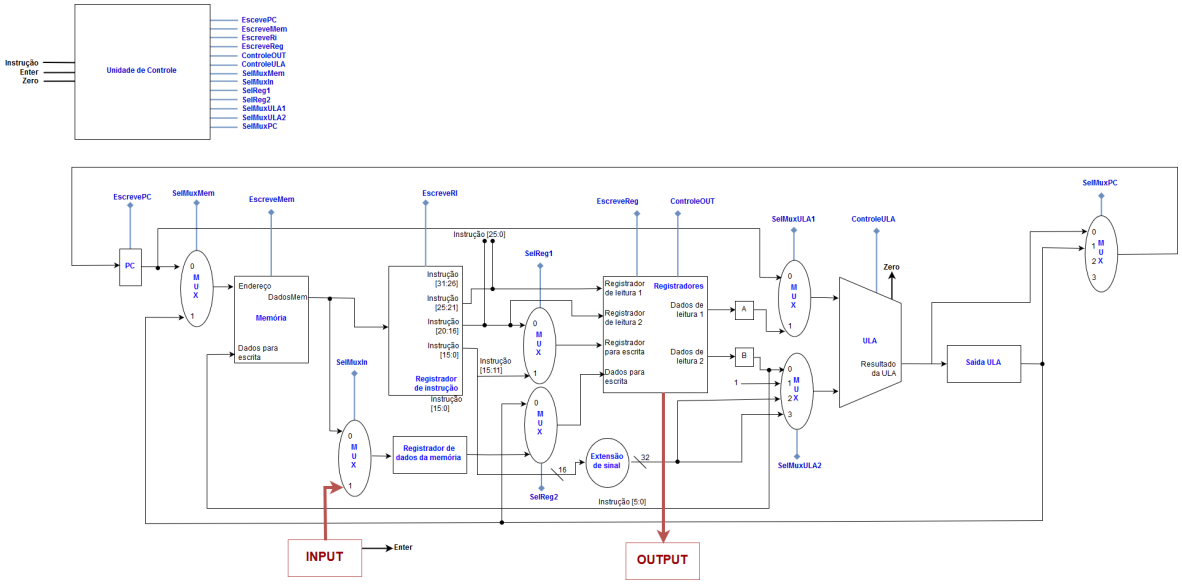
O intuito deste capítulo é descrever aplicação de toda a introdução teórica no desenvolvimento o projeto, explicando os modelos escolhidos e os esquemáticos. O objetivo desse projeto foi desenvolver o esquema de um processador completo com memória, registradores e ALU, porém sem unidade de controle, e implementá-los m Verilog.

4.1 Arquitetura Escolhida

Para a realização do projeto foi escolhida a abordagem multiciclo devido seu melhor desempenho e menor quantidade de módulos como explicado na Seção 3.4. O projeto foi baseado na arquitetura MIPS multiciclo presente no livro *Organização e Projeto de Computadores: A Interface Hardware/Software*(1) de David A. Patterson e John L. Henessy. O caminho de dados(*datapath*) do processador pode ser visto na Figura 10.

Outras estruturas presentes no *datapath* do projeto que não estavam na Figura 9 são os módulos de entrada, saída e extensor de sinal. O módulo extensor de sinal replica o bit mais significativo à esquerda do número de entrada até o mesmo atingir o comprimento de 32 bits, já o módulo de saída (*Output*) retira o valor indicado no Banco de Registradores, o converte em um código BCD e decodfica para o display de sete segmentos e por fim a entrada é simplesmente o sinal vindo dos *switches* do kit FPGA que são salvas no Registrador de Dados de Memória para poderem ser armazenados no Banco de Registradores.

Figura 10 – *Datapath* do Processador



Fonte:O Autor

4.2 Conjunto de Instruções

Para esse projeto foi optado por um conjunto de instruções do tipo RISC, com instruções não muito complexas e poucos formatos de instrução. O conjunto de instruções também foi baseado no conjunto presente no processador MIPS de *Organização e Projeto de Computadores* (1), com algumas alterações como novas instruções e novos formatos de instrução.

O conjunto de instruções escolhido para o projeto pode ser dividido em quatro formato de instruções sendo eles tipo R para instruções com operações que utilizam apenas operandos com seus valores guardados no banco de registradores, tipo I para instruções que utilizam além de valores presentes em registradores um valor imediato, tipo J para instruções de *Jump* e tipo *IN/OUT* para instruções específicas para os módulos de E/S (entrada e saída), sendo o ultimo tipo uma adição não presente na arquitetura MIPS. O formato das palavras de instrução citadas acima podem ser vistas abaixo nas Tabelas 1, 2, 3 e 4.

Tabela 1 – Formato da Instrução Tipo R

Opcode	**R1**	**R2**	**RF**	***Don't Care***	***Funct***
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Fonte: O Autor

Tabela 2 – Formato da Instrução Tipo I

Opcode	**R1**	**RF**	**Imediato**
6 bits	5 bits	5 bits	16 bits

Fonte: O Autor

Tabela 3 – Formato da Instrução Tipo IN/OUT

Opcode	***Don't Care***	**R**	***Don't Care***
6 bits	5 bits	5 bits	16 bits

Tabela 4 – Formato da Instrução Tipo J

Opcode	**Endereço**
6 bits	26 bits

Como pode ser visto nas tabelas existem campos comuns entre as instruções e alguns campos peculiares para cada instrução. O primeiro campo comum que pode ser notado é o campo de *Opcode*, com o tamanho de 6 bits. Esse campo é o responsável pela diferenciação entre as instruções do conjunto, cada *Opcode* codifica uma instrução, exceto no caso das instruções do tipo R que tem praticamente o mesmo *Opcode* para todas as instruções e sua difererenciação se dá pelo campo *Funct*.

Outro campo que pode ser visualizado em mais de um formato de instrução é o campos Rx, como por exemplo R1, R2 e RF das instruções R. Esses campos são responsáveis por armazenar o endereço de regitradores, sejam eles registradores de origem dos dados ou de destino.

Nas instruções dos tipos I e J pode-se notar a presença de campos com valores imediatos, sendo eles *"Imediato"* e *"Endereço"*, que são responsáveis por carregarem valores a serem usados diretamente em operações, ou seja, não tem a necessidade de buscar os valores dos operandos ou endereços no banco de registradores.

Os campos *Don't Care* das instruções IN/OUT e R são grupos de bits que não interferem na operação realizada pelo procesador.

Apresentadados os formatos de instrução pode-se começar a discutir as instruções do conjunto. As mesmas podem ser vistas abaixo na Tabela 5.

Esse conjunto de instruções, como citado acima, conta com algumas instruções adicionais quando comparado ao conjunto de instruções padrão do MIPS convencional, sendo elas *Entrada* e *Saída*. Essas instruções servem para controlar a entrada e a saída de dados.

Tabela 5 – Conjunto de Instruções

Nome	**Tipo**	**Abreviação**
Soma	R	*add*
Subtraçao	R	*sub*
Multiplicação	R	*mult*
Divisão	R	*div*
And	R	*and*
Or	R	*or*
Nand	R	*nand*
Nor	R	*nor*
Set less or equal than	R	*sle*
Set less than	R	*slt*
Soma imediato	I	*addi*
Subtração imediato	I	*subi*
Multiplicação imediato	I	*multi*
Divisão imediato	I	*divi*
And imediato	I	*andi*
Or imediato	I	*ori*
Nand imediato	I	*nandi*
Nor imediato	I	*nori*
Set less or equal than imediato	I	*slei*
Set less than imediato	I	*slti*
Store Word	I	*sw*
Load Word	I	*lw*
Halt	I	*hlt*
Branch on equal	I	*bre*
Branch greater than	I	*bgt*
Branch less than	I	*blt*
Branch on not equal	I	*bne*
Entrada	IN/OUT	*in*
Saída	IN/OUT	*out*
Jump	J	*jmp*

Fonte: O Autor

4.3 Modos de Endereçamento

Para definir os modos de endereçamento que serão utilizados no projeto do processador foi analizado o comportamento das instruções que estarão presentes no mesmo e a necessidade de cada uma. Após essa análise foram definidos dois modos de endereçamento a serem utilizados: endereçamento imediato e endereçamento por registrador. O endereçamento imediato será utilizado nas instruções do tipo I uma vez que existe um campo *Imediato* que carrega o valor de um operando para a realização de uma operação, e na instrução de Jump uma vez que o endereço de desvio será armazenado inteiramente no campo imediato e as instruções do tipo IN/OUT e R utilizam o endereçamento por registrador já que o valor requisitado está sempre presente em um registrador.

Devido ao tamanho limitado da memória não foi preciso usar enderecamentos de base ou deslocamento, uma vez que apenas os campos de imediatos comportam todos os endereços da memória.

4.4 Entrada e Saída

O módulos de entrada e saída foram planejados para funcionar acoplados ao Registrador de Dados de Memória e ao banco de registradores respectivamente (estruturas que podem ser vistas na Figura 10), de modo que o valor de entrada seja primeiramente carregado no *registrador de dados de memória* para então ser carregado no banco de registradores no endereço referente ao valor de R na palavra de instrução *in*. Já referente ao módulo de saída ao ser processada, a instrução *out* carregará o dado armazenado no registrador R presente em sua palavra de instrução no módulo de Saída para que assim o valor possa ser tratado pela mesma.

A unidade de entrada trabalhará com a interrupção do processador até a ativação de uma chave de controle para poder guardar o valor de entrada no registrador.

4.5 Implementação em Verilog

Após todo desenvolvimento teórico do funcionamento do processador, o próximo passo é implementar os módulos idealizados em Verilog. Observando o *daatapath* presente na Figura 10 é possivel contar 21 unidades funcionais que teóricamente teriam que ser implementadas, porém existe muita redundância entre essas unidades como por exemplo entre os registradores de A, B e de Instrução. Apesar de serem três unidades diferentes, todos são registradores simples de 32 bits com um sinal de controle para escrita, o que possibilita serem implementados em Verilog apenas uma vez como um registrador de 32 bits e declarados várias vezes no momento de unir todas as unidades.

Sabendo desse fato é possível resumir os 21 módulos em apenas 9 unidades que serão implementadas em Verilog, sendo elas: Banco de Registradores, Extensor de Sinal, Unidade de Memória, Multiplexador de duas saídas de 5 bits, Multiplexadore de duas de 32 bits, Multiplexador de quatro saídas de 32 bits, Registrador de 32 bits, Deslocamento à esquerda e Unidade Lógica e Aritmética.

4.5.1 Banco de Registradores

O banco de registradores à princípio é uma estrutura que conta com 32 registradores de 32 bits, endereçados por um número de 5 bits. O Banco de Registradores é a primeira unidade não genérica da lista de uniaddes presentes no processador. À princípio existem um registrador de propósito específico em todo banco sendo ele o registrador *$rr* usado para guardar o resto de uma operação de divisão, os demais 31 registradores são todos de propósito geral.

O Banco de Registradores recebe como entrada 3 registradores de 5 bits referentes aos registradores 1 e 2 de leitura e um terceiro registrador para escrita, um sinal de controle de 3 bits que controlam respectivamente a leitura dos registradores reg1 e reg2 e a escrita no registrador regF e u número de 32 bits a ser escrito no registrador regF. Como saída tem apenas dois números de 32 bits que serão guardados nos registradores A e B antes de serem processados na ULA.

O código Verilog do banco pode ser conferido abaixo:

```
1  module bancoReg(controle, reg1, reg2, regF, dados, A, B, clk);
2
3          input clk;
4          input [2:0]controle;
5          input [4:0] reg1, reg2, regF;
6          input   [31:0]dados;
7          output reg [31:0] A, B;
8          reg [31:0]registradores[31:0];
9
10 always @(posedge clk) begin
11
12         if(controle[0] == 1'b1)
13                 A = registradores[reg1];
14
15         if(controle[1] == 1'b1)
16                 B = registradores[reg2];
17
18         if(controle[2] == 1'b1)
19                 registradores[regF] = dados;
20
21 end
22 endmodule
```

4.5.2 Extensor de Sinal

O Extensor de Sinal é o módulo responsável por transformar a entrada de 16 bits vinda da palavra de instrução em um número de 32 bits que será processado pela ULA.

Esse módulo é relativamente simples, conta apenas com uma entrada de 16 bits (sinal) e uma saída de 32 bits (sinal_ext), onde o bit mais significativo da entrada é replicado até a mesma atingir o tamanho de 32 bits.

```
1  module extensor(sinal, sinal_ext);
2
3          input [15:0]sinal;
4          output reg [31:0]sinal_ext;
5
6  always @(*)begin
7
8          if(sinal[15] == 1'b0)
9                  sinal_ext = {16'b0000000000000000,sinal};
10         else
11                 sinal_ext = {16'b1111111111111111,sinal};
12
13 end
14 endmodule
```

4.5.3 Memória

O módulo de Memória consiste em registradores endereçados por um número de 32 bits chamado 'endereço', 2 bits de controle responsáveis rescpectivamente pela leitura e escrita de um dado e uma saída de 32 bits. Caso fosse usada a capacidade total de endereçamento, a memória teria o tamanho de 4294967296 registradores, porém graças à limitação física das FPGAs foi optado por utilizar apenas 100 registradores.

```
1
2  module memoria(dado, endereco, write, wclk, rclk, saida);
3
4          input [21:0] dado;
5          input [6:0] endereco;
6          input write, wclk, rclk;
7          output reg [31:0] saida;
8          reg [31:0]ram[127:0];
9
10         initial begin
11
12                         ram[7'd0]   = {6'b101000,5'd0,5'd1,16'd0}; // out
13                         ram[7'd1]   = {6'b100001,5'd0,5'd1,16'd0}; // out
14                         ram[7'd2]   = {6'b111111,26'd0}; // hlt
15         end
16         always @ (posedge wclk)
17         begin
18                 if (write == 1'b1)
19                         ram[endereco] <= dado;
20         end
```

```
21
22 always @ (negedge rclk) saida <= ram[endereco];
23
24 endmodule
```

4.5.4 Multiplexadores

Um multiplexador é responsável por alternar um sinal de entrada entre várias saídas dependendo de um sinal de controle. Como pode ser visto na Figura 10, no projeto foram utilizados 3 tipos de multiplexadores sendo um com duas saídas de 5 bits, um com duas saídas de 32 bits e um com quatro saídas de 32 bits. Os códigos Verilog dos mesmos podem ser vistos abaixo:

```
1  module mux2_5b(seletor, entrada1, entrada2, saida);
2          input seletor;
3          input [4:0]entrada1, entrada2;
4          output reg [4:0]saida;
5
6  always @(*) begin
7          if(seletor == 0)
8                  saida = entrada1;
9          else
10                 saida = entrada2;
11
12 end
13 endmodule
```

```
1  module mux2_32b(seletor, entrada1, entrada2, saida);
2          input seletor;
3          input [31:0]entrada1, entrada2;
4          output reg [31:0]saida;
5
6  always @(*) begin
7          if(seletor == 0)
8                  saida = entrada1;
9          else
10                 saida = entrada2;
11
12 end
13 endmodule
```

```
1  module mux4_32b(seletor, entrada1, entrada2, entrada3, entrada4, saida);
2          input [1:0] seletor;
3          input [31:0] entrada1, entrada2, entrada3, entrada4;
4          output reg [31:0]saida;
5
6  always @(*) begin
7          case (seletor)
8                  2'b00:
9                          saida = entrada1;
10                 2'b01:
11                         saida = entrada2;
```

```
12                  2'b10:
13                          saida = entrada3;
14                  2'b11:
15                          saida = entrada4;
16          endcase
17  end
18  endmodule
```

4.5.5 Registradores

Esse módulo engloba todos os registradores únicos presentes no *datapath* como PC, A, B, Saída ULA, entre outros. Os registradores recebem como entrada um sinal de contole de 1 bit para controlar a escrita e um sinal 'entrada' de 32 bits, referente ao dado que será armazenado no registrador, e como saída um valor de 32 bits referente ao valor que está salvo no registrador.

```
1   module registrador32b(controle, set, clk, entrada, saida);
2                   input [31:0] entrada;
3                   input controle, clk, set;
4                   output reg [31:0] saida;
5
6   always @(posedge clk)begin
7
8           if(controle == 1'b1) saida <= entrada;
9
10  end
11
12  endmodule
```

4.5.6 ULA

A ULA (Unidade Lógica e Aritimética) vai ser a responsável por processar os dados dos registradores A e B e apresentar seu resultado, realizando operações de soma, subtração, divisão, multiplicação, or, and, nor ou nand ou ainda instruções como *set less than* (menor que), *set less or equal than* (menor ou igual), entre outras. Ela recebe como entrada três valores, os operandos de 32 bits vindos dos registradores A e B e um sinal de controle vindo do módulo de controle da ULA que selecionará o tipo de operação que será feita.

```
1   module ULA(A, B, clk, controle, saida, overflow, zero);
2
3           input [4:0] controle;
4           input [31:0] A, B;
5           input clk;
6           output reg [31:0]saida;
7           output reg overflow, zero;
8
9   always @(posedge clk) begin
10
```

```
11          case(controle)
12                  5'd0: begin//add
13                          saida = A + B;
14                          overflow = 1'b0;
15                          zero = 1'b0;
16                  end
17                  5'd1: begin//sub
18                          saida = A - B;
19                          overflow = 1'b0;
20                          zero = 1'b0;
21                  end
22                  5'd2: //mult
23                  begin
24                          if(A[16] == 1 && B[16] == 1)
25                                  overflow = 1'b1;
26                          else
27                                  overflow = 1'b0;
28                          zero = 1'b0;
29                          saida = A * B;
30                  end
31
32                  5'd3: begin //div
33                          saida = A / B;
34                          overflow = 1'b0;
35                          zero = 1'b0;
36                  end
37                  5'd4: begin //and
38                          saida = A & B;
39                          overflow = 1'b0;
40                          zero = 1'b0;
41                  end
42                  5'd5: begin //or
43                          saida = A | B;
44                          overflow = 1'b0;
45                          zero = 1'b0;
46                  end
47                  5'd6: begin //nand
48                          saida = ~(A & B);
49                          overflow = 1'b0;
50                          zero = 1'b0;
51                  end
52                  5'd7: begin //nor
53                          saida = ~(A | B);
54                          overflow = 1'b0;
55                          zero = 1'b0;
56                  end
57                  5'd8: //beq
58                  begin
59                          saida = B;
60                          overflow = 1'b0;
61                          if( A == B)
62                                  zero = 1'b1;
63                          else
64                                  zero = 1'b0;
65                  end
66
67                  5'd9: begin//bne
68                          saida = B;
69                          overflow = 1'b0;
70                          if(A != B)
```

```
71                                      zero = 1'b1;
72                              else
73                                      zero = 1'b0;
74                      end
75
76                      5'd10: //bgt
77                      begin
78                              saida = B;
79                              overflow = 1'b0;
80                              if(A > B)
81                                      zero =  1'b1;
82                              else
83                                      zero = 1'b0;
84                      end
85
86                      5'd11: //slt
87                      begin
88                              overflow = 1'b0;
89                              zero = 1'b0;
90                              if(A < B)
91                                      saida = 1'b1;
92                              else
93                                      saida = 1'b0;
94                      end
95
96                      5'd12: //sle
97                      begin
98                              overflow = 1'b0;
99                              zero = 1'b0;
100                             if(A > B)
101                                     saida = 1'b0;
102                             else
103                                     saida = 1'b1;
104                     end
105
106                     5'd13: //blt
107                     begin
108                             if(A < B)
109                                     zero = 1'b1;
110                             else
111                                     zero = 1'b0;
112                             overflow = 1'b0;
113                             saida = B;
114                             end
115
116                     5'd31: begin
117                             zero = 1'b0;
118                             overflow = 1'b0;
119                             saida = B;
120                     end
121
122                     default: begin
123                             saida = saida;
124                             overflow = overflow;
125                             zero = zero;
126                     end
127
128             endcase
129     end
130     endmodule
```

4.5.7 Módulo de Saída

Como explicado acima, a saída do processador é dada diretamente do Banco de Registradores para o módulo de saída. O valor de saída é mostrado em 4 displays de 7 segmentos no formato decimal, para isso é necessário converter o valor vindo do Banco de Registradores de binário para BCD e então converter o número BCD para ser mostrado nos displays. Para isso foi usada a seguinte lógica: o número de entrada primeiramente é divido por dez e seu resto é salvo em uma variável, então resultado é novamente dividido por dez e o resto armazenado em outra variável. O processo é feito 4 vezes de forma que ao final teremos um número dividido por dez mil e quatro números entre 0 e 9. Caso o número restante for maior que zero, os quatro displays mostram a letra 'E', identificando que o numero alvo é maior do que a capacidade de saída do sistema. Caso o número seja igual a zero os quatro números de 0 a 9 são enviados para outro módulo onde serão decodificados para o display. Os códigos desses módulos podem ser conferidos abaixo.

```
1  module moduloSaida(entrada, saida4, saida3, saida2, saida1,clk);
2
3          input clk;
4          input [31:0] entrada;
5          output [6:0] saida1, saida2, saida3, saida4;
6          reg [31:0] n1,  n2, n3, n4;
7          reg [31:0] temp;
8
9  initial begin
10
11         n1 = 32'd15;
12         n2 = 32'd15;
13         n3 = 32'd15;
14         n4 = 32'd15;
15
16 end
17
18 always @(*) begin
19         temp = entrada;
20
21                 if(temp / 32'd10000 > 0) begin
22                         n1 <= 32'd14;
23                         n2 <= 32'd14;
24                         n3 <= 32'd14;
25                         n4 <= 32'd14;
26                 end
27                 else begin
28                         n1 <= temp % 32'd10;
29                         temp = temp / 32'd10;
30                         n2 <= temp % 32'd10;
31                         temp = temp / 32'd10;
32                         n3 <= temp % 32'd10;
33                         temp = temp / 32'd10;
34                         n4 <= temp % 32'd10;
35                 end
36         //end
37 end
38 // decodfica display 1
```

```
39 decodDisplay dispay1(n1[3:0],saida1);
40 // decodfica display 2
41 decodDisplay dispay2(n2[3:0],saida2);
42 // decodfica display 3
43 decodDisplay dispay3(n3[3:0],saida3);
44 // decodfica display 4
45 decodDisplay dispay4(n4[3:0],saida4);
46
47 endmodule
```

```
1  module decodDisplay(in,segmentos);
2          input [3:0] in;
3          output reg [6:0] segmentos;
4
5  always@(*) begin
6          case (in)
7                  4'b0000: segmentos=7'b0000001;
8                  4'b0001: segmentos=7'b1001111;
9                  4'b0010: segmentos=7'b0010010;
10                 4'b0011: segmentos=7'b0000110;
11                 4'b0100: segmentos=7'b1001100;
12                 4'b0101: segmentos=7'b0100100;
13                 4'b0110: segmentos=7'b0100000;
14                 4'b0111: segmentos=7'b0001111;
15                 4'b1000: segmentos=7'b0000000;
16                 4'b1001: segmentos=7'b0000100;
17                 4'b1110: segmentos=7'b0110000;
18                 default: segmentos = 7'b1111111;
19         endcase
20 end
21
22 endmodule
```

4.5.8 Unidade de Controle

Por fim, o módulo mais importante do processador é a unidade de controle. É a responsável por sinalizar as operações a serem realizadas em cada unidade do processador por meio de sinais de controle.

A unidade de controle é baseada em uma máquina de estados finitos de Moore dependente do *opcode* da instrução para mudar seu estado e a maioria dos sinais de controle são definidos pelo estado atual. Apesar disso, no caso da unidade de controle do projeto em questão foi optado por em sinais pontuais o *opcode* ser também um dos critérios para definir seu valor.

```
1  module ctrl_undd(opcode,
2                                          funct,
3                                          enter,
4                                          clk,
5                                          estado,
6                                          EscrevePC,
7                                          EscreveRI,
```

```
8                                       EscreveReg,
9                                       EscreveMem,
10                                      SelMuxMem,
11                                      SelMuxReg1,
12                                      SelMuxReg2,
13                                      SelMuxUlaA,
14                                      SelMuxUlaB,
15                                      SelMuxPC,
16                                      zero,
17                                      controleULA,
18                                      controleOUT,
19                                      SelMuxIn);
20
21      input clk, zero, enter;
22      input [5:0] opcode;
23      input [5:0] funct;
24      output reg [3:0] estado;
25      reg [3:0] prox_estado;
26
27      //sinais de controle de memoria
28      output reg EscrevePC, EscreveRI, EscreveReg, EscreveMem, controleOUT;
29
30      //seletores de multiplexadores
31      output reg SelMuxMem, SelMuxReg1, SelMuxReg2, SelMuxUlaA, SelMuxIn;
32      output reg [1:0] SelMuxUlaB, SelMuxPC;
33
34      //sinal de controle da ULA
35      reg [1:0]OpULA;
36      output [4:0] controleULA;
37
38      parameter ESTADO0=4'b0000, ESTADO1=4'b0001, ESTADO2=4'b0010, ESTADO3=4'b0011,
39                          ESTADO4=4'b0100, ESTADO5=4'b0101, ESTADO6=4'b0110,
                                ESTADO7=4'b0111,
40                          ESTADO8=4'b1000, ESTADO9=4'b1001, ESTADO10=4'b1010,
                                ESTADO11=4'b1011,
41                          ESTADO12=4'b1100, ESTADO13=4'b1101, ESTADO14=4'b1110,
                                ESTADO15=4'b1111;
42
43      ULA_ctrl ctrlULA(.opcode(opcode),
44                                          .funct(funct),
45                                          .opULA(OpULA),
46                                          .controle(controleULA),
47                                          .clk(clk));
48
49      always @(negedge clk) begin
50
51              case(estado)
52              ESTADO0: begin //carrega RI
53              // controle
54                      EscrevePC   <=  1'b0;
55                      EscreveRI   <=  1'b1;
56                      EscreveReg  <=  1'b0;
57                      EscreveMem  <=  1'b0;
58                      controleOUT <=  1'b0;
59                      OpULA       <= 2'b01;
60              // mux
61                      SelMuxPC    <= 2'b00;
62                      SelMuxMem   <=  1'b0;
63                      SelMuxReg1  <=  1'b0;
64                      SelMuxReg2  <=  1'b0;
```

```
65                      SelMuxUlaA   <=  1'b0;
66                      SelMuxUlaB   <= 2'b01;
67                      SelMuxIn     <=  1'b1;
68                      prox_estado <= ESTADO1;
69              end
70
71              ESTADO1: begin //decodifica instrucao
72              // controle
73                      if(opcode == 6'b111111) EscrevePC <= 1'b0;
74                      else EscrevePC   <=  1'b1;
75                      EscreveRI    <=  1'b0;
76                      EscreveReg   <=  1'b0;
77                      EscreveMem   <=  1'b0;
78                      if(opcode == 6'b100001) controleOUT <=  1'b1;
79                      else controleOUT <=  1'b0;
80                      OpULA <= 2'b00;
81              // mux
82                      SelMuxPC     <= 2'b00;
83                      SelMuxMem    <=  1'b0;
84                      SelMuxReg1   <=  1'b0;
85                      SelMuxReg2   <=  1'b0;
86                      SelMuxUlaA   <=  1'b0;
87                      SelMuxUlaB   <= 2'b01;
88                      SelMuxIn     <=  1'b0;
89
90                      case(opcode)
91                              6'b000110: begin
92                                      prox_estado <= ESTADO2; //sw
93                              end
94                              6'b000111: begin
95                                      prox_estado <= ESTADO2; //lw
96                              end
97                              6'b000000: begin
98                                      prox_estado <= ESTADO6; //R
99                              end
100                             6'b010000: begin
101                                     prox_estado <= ESTADO8; //btl
102                             end
103                             6'b100000: begin
104                                     prox_estado <= ESTADO8; //bgt
105                             end
106                             6'b110000: begin
107                                     prox_estado <= ESTADO8; ///beq
108                             end
109                             6'b111000: begin
110                                     prox_estado <= ESTADO8; //bne
111                             end
112
113                             6'b111110: begin
114                                     prox_estado <= ESTADO9; //jmp
115                             end
116                             6'b111100: begin
117                                     prox_estado <= ESTADO9; //jal
118                             end
119
120                             6'b101000: begin
121                                     prox_estado <= ESTADO12; //in
122                             end
123
124                             6'b100001: begin
```

```
125                             prox_estado <= ESTADO0; //out
126                     end
127
128                     6'b111111: begin
129                             prox_estado <= ESTADO14; //hlt
130                     end
131
132                     default: begin
133                             prox_estado <= ESTADO11; // I
134                     end
135             endcase
136     end
137
138     ESTADO2: begin //carrega B na saida da ULA (lw)
139     // controle
140             EscrevePC   <=  1'b0;
141             EscreveRI   <=  1'b0;
142             EscreveReg  <=  1'b0;
143             EscreveMem  <=  1'b0;
144             controleOUT <=  1'b0;
145             OpULA       <= 2'b11;
146     // mux
147             SelMuxPC    <= 2'b00;
148             SelMuxMem   <=  1'b1;
149             SelMuxReg1  <=  1'b0;
150             SelMuxReg2  <=  1'b0;
151             SelMuxUlaA  <=  1'b1;
152             SelMuxUlaB  <= 2'b11;
153             SelMuxIn    <=  1'b0;
154
155             case(opcode)
156                     6'b000111: prox_estado <= ESTADO3; //lw
157
158                     6'b000110: prox_estado <= ESTADO5; //sw
159
160                     default: prox_estado <= ESTADO0;
161             endcase
162     end
163
164     ESTADO3: begin //busca valor de dado dentro da memoria
165     // controle
166             EscrevePC   <=  1'b0;
167             EscreveRI   <=  1'b0;
168             EscreveReg  <=  1'b0;
169             EscreveMem  <=  1'b0;
170             controleOUT <=  1'b0;
171             OpULA       <= 2'b00;
172     // mux
173             SelMuxPC    <= 2'b00;
174             SelMuxMem   <=  1'b1;
175             SelMuxReg1  <=  1'b0;
176             SelMuxReg2  <=  1'b1;
177             SelMuxUlaA  <=  1'b0;
178             SelMuxUlaB  <= 2'b00;
179             SelMuxIn    <=  1'b0;
180             prox_estado <= ESTADO4;
181     end
182
183     ESTADO4: begin //salva dado do registrador de dados no banco
184     // controle
```

```
185                     EscrevePC   <=  1'b0;
186                     EscreveRI   <=  1'b0;
187                     EscreveReg  <=  1'b1;
188                     EscreveMem  <=  1'b0;
189                     controleOUT <=  1'b0;
190                     OpULA       <= 2'b00;
191             // mux
192                     SelMuxPC    <= 2'b00;
193                     SelMuxMem   <=  1'b1;
194                     SelMuxReg1  <=  1'b0;
195                     SelMuxReg2  <=  1'b1;
196                     SelMuxUlaA  <=  1'b0;
197                     SelMuxUlaB  <= 2'b00;
198                     SelMuxIn    <=  1'b0;
199                     prox_estado <= ESTADO13;
200             end
201
202             ESTADO5: begin //salva valor na memoria
203             // controle
204                     EscrevePC   <=  1'b0;
205                     EscreveRI   <=  1'b0;
206                     EscreveReg  <=  1'b0;
207                     EscreveMem  <=  1'b1; // salva valor na memoria
208                     controleOUT <=  1'b0;
209                     OpULA       <= 2'b11;
210             // mux
211                     SelMuxPC    <= 2'b00;
212                     SelMuxMem   <=  1'b1;
213                     SelMuxReg1  <=  1'b0;
214                     SelMuxReg2  <=  1'b0;
215                     SelMuxUlaA  <=  1'b1;
216                     SelMuxUlaB  <= 2'b00;
217                     SelMuxIn    <=  1'b0;
218                     prox_estado <= ESTADO15;
219             end
220
221             ESTADO6: begin //faz operacao entre A e B
222                     EscrevePC   <=  1'b0;
223                     EscreveRI   <=  1'b0;
224                     EscreveReg  <=  1'b0;
225                     EscreveMem  <=  1'b0;
226                     controleOUT <=  1'b0;
227                     OpULA       <= 2'b00;
228             // mux
229                     SelMuxPC    <= 2'b00;
230                     SelMuxMem   <=  1'b0;
231                     SelMuxReg1  <=  1'b1;
232                     SelMuxReg2  <=  1'b0;
233                     SelMuxUlaA  <=  1'b1;
234                     SelMuxUlaB  <= 2'b00;
235                     SelMuxIn    <=  1'b0;
236                     prox_estado <= ESTADO7;
237             end
238
239             ESTADO7: begin //salva o resultado no banco
240                     EscrevePC   <=  1'b0;
241                     EscreveRI   <=  1'b0;
242                     EscreveReg  <=  1'b1;
243                     EscreveMem  <=  1'b0;
244                     controleOUT <=  1'b0;
```

```
245                     OpULA       <= 2'b00;
246             // mux
247                     SelMuxPC    <= 2'b00;
248                     SelMuxMem   <=  1'b0;
249                     if(opcode == 6'b000000)SelMuxReg1  <=  1'b1;
250                     else SelMuxReg1 <= 1'b0;
251                     SelMuxReg2  <=  1'b0;
252                     SelMuxUlaA  <=  1'b1;
253                     if(opcode == 6'b000000)SelMuxUlaB  <= 2'b00;
254                     else SelMuxUlaB <= 2'b11;
255                     SelMuxIn    <=  1'b0;
256                     prox_estado <= ESTADO0;
257             end
258
259             ESTADO8: begin //calcula endereco de branch
260                     prox_estado <= ESTADO10;
261                     EscrevePC <= 1'b0;
262                     EscreveRI   <=  1'b0;
263                     EscreveReg  <=  1'b0;
264                     EscreveMem  <=  1'b0;
265                     controleOUT <=  1'b0;
266                     OpULA       <= 2'b00;
267             // mux
268                     SelMuxPC    <= 2'b01;
269                     SelMuxMem   <=  1'b0;
270                     SelMuxReg1  <=  1'b0;
271                     SelMuxReg2  <=  1'b0;
272                     SelMuxUlaA  <=  1'b1;
273                     SelMuxUlaB  <= 2'b00;
274                     SelMuxIn    <=  1'b0;
275             end
276
277             ESTADO9: begin //faz o jump
278                     EscrevePC   <=  1'b1;
279                     EscreveRI   <=  1'b0;
280                     EscreveReg  <=  1'b0;
281                     EscreveMem  <=  1'b0;
282                     controleOUT <=  1'b0;
283                     OpULA       <= 2'b11;
284             // mux
285                     SelMuxPC    <= 2'b00;
286                     SelMuxMem   <=  1'b0;
287                     SelMuxReg1  <=  1'b0;
288                     SelMuxReg2  <=  1'b0;
289                     SelMuxUlaA  <=  1'b1;
290                     SelMuxUlaB  <= 2'b11;
291                     SelMuxIn    <=  1'b0;
292                     prox_estado <= ESTADO13;
293             end
294
295             ESTADO10: begin //atualiza pc
296                     if(zero == 1'b1)EscrevePC   <=  1'b1;
297                     else EscrevePC   <=  1'b0;
298                     EscreveRI   <=  1'b0;
299                     EscreveReg  <=  1'b0;
300                     EscreveMem  <=  1'b0;
301                     controleOUT <=  1'b0;
302                     OpULA       <= 2'b11;
303             // mux
304                     SelMuxPC    <= 2'b10;
```

```
305                     SelMuxMem    <=  1'b0;
306                     SelMuxReg1   <=  1'b0;
307                     SelMuxReg2   <=  1'b0;
308                     SelMuxUlaA   <=  1'b0;
309                     SelMuxUlaB   <= 2'b00;
310                     SelMuxIn     <=  1'b0;
311                     prox_estado <= ESTADO0;
312                 end
313
314                 ESTADO11:begin //faz operacao entre A e Imm
315                     EscrevePC    <=  1'b0;
316                     EscreveRI    <=  1'b0;
317                     EscreveReg   <=  1'b0;
318                     EscreveMem   <=  1'b0;
319                     controleOUT  <=  1'b0;
320                     OpULA        <= 2'b00;
321                 // mux
322                     SelMuxPC     <= 2'b00;
323                     SelMuxMem    <=  1'b0;
324                     SelMuxReg1   <=  1'b0;
325                     SelMuxReg2   <=  1'b0;
326                     SelMuxUlaA   <=  1'b1;
327                     SelMuxUlaB   <= 2'b11;
328                     SelMuxIn     <=  1'b0;
329                     prox_estado <= ESTADO7;
330                 end
331
332                 ESTADO12: begin //aguarda entrada de dados
333                     EscrevePC    <=  1'b0;
334                     EscreveRI    <=  1'b0;
335                     EscreveReg   <=  1'b1;
336                     EscreveMem   <=  1'b0;
337                     controleOUT  <=  1'b0;
338                     OpULA        <= 2'b00;
339                 // mux
340                     SelMuxPC     <= 2'b00;
341                     SelMuxMem    <=  1'b0;
342                     SelMuxReg1   <=  1'b0;
343                     SelMuxReg2   <=  1'b0;
344                     SelMuxUlaA   <=  1'b0;
345                     SelMuxUlaB   <= 2'b00;
346                     SelMuxIn     <=  1'b1;
347                     if(enter) prox_estado <= ESTADO15;
348                     else prox_estado <= prox_estado;
349                 end
350
351                 ESTADO13: begin // finaliza jmp, sw e lw
352                     if(opcode == 6'b111110) EscrevePC    <=  1'b1;
353                     else EscrevePC <= 1'b0;
354                     EscreveRI    <=  1'b0;
355                     if(opcode == 6'b000111) EscreveReg   <=  1'b1;
356                     else EscreveReg <= 1'b0;
357                     EscreveMem <= 1'b0;
358                     controleOUT  <=  1'b0;
359                     OpULA        <= 2'b00;
360                 // mux
361                     SelMuxPC     <= 2'b00;
362                     SelMuxMem    <=  1'b0;
363                     SelMuxReg1   <=  1'b0;
364                     SelMuxReg2   <=  1'b1;
```

```
365                         SelMuxUlaA   <=  1'b0;
366                         SelMuxUlaB   <= 2'b00;
367                         SelMuxIn     <=  1'b0;
368                         prox_estado <= ESTADO0;
369                 end
370
371                 ESTADO14: begin //paraliza o processador
372                         EscrevePC    <=   1'b0;
373                         EscreveRI    <=  1'b0;
374                         EscreveReg   <=  1'b0;
375                         EscreveMem   <=  1'b0;
376                         controleOUT <=  1'b0;
377                         OpULA        <=  2'b00;
378                 // mux
379                         SelMuxPC     <= 2'b00;
380                         SelMuxMem    <=  1'b0;
381                         SelMuxReg1   <=  1'b0;
382                         SelMuxReg2   <=  1'b0;
383                         SelMuxIn     <=  1'b0;
384                         SelMuxUlaA   <=   1'b0;
385                         SelMuxUlaB   <=  2'b00;
386                         prox_estado <= ESTADO14;
387
388                 end
389
390                 ESTADO15: begin
391                 // controle
392                         EscrevePC    <=  1'b0;
393                         EscreveRI    <=  1'b0;
394                         if(opcode == 6'b000110) EscreveReg   <=  1'b0;
395                         else EscreveReg   <=  1'b1;
396                         EscreveMem   <=  1'b0;
397                         controleOUT <=  1'b0;
398                         OpULA        <= 2'b00;
399                 // mux
400                         SelMuxPC     <= 2'b00;
401                         SelMuxMem    <=  1'b0;
402                         SelMuxReg1   <=  1'b0;
403                         SelMuxReg2   <=  1'b1;
404                         SelMuxUlaA   <=  1'b0;
405                         SelMuxUlaB   <= 2'b00;
406                         SelMuxIn     <=  1'b1;
407                         prox_estado <= ESTADO0;
408                 end
409                 endcase
410         end //fim always
411
412 always @(posedge clk) begin
413
414                 estado <= prox_estado;
415
416         end//fim always
417 endmodule
```

Como pôde ser verificado no código acima, existe um módulo dentro da unidade de controle chamado 'ULActrl', esse módulo é voltado apenas para designar os sinais de controle da ULA e funciona apenas como um decodficador, dependente do *opcode*, do campo *funct* e da Opula.

```
module ULA_ctrl(opcode,funct,opULA,controle,clk);
        input [5:0] funct, opcode;
        input [1:0] opULA;
        input clk;
        output reg [4:0] controle;

always @(*) begin

        if(opcode == 6'd0) begin
                case (funct)
                        6'b000000: controle = 5'd0; // add
                        6'b000001: controle = 5'd1; // sub
                        6'b000010: controle = 5'd2; // mult
                        6'b000011: controle = 5'd3; // div
                        6'b100000: controle = 5'd4; // and
                        6'b100001: controle = 5'd5; // or
                        6'b100010: controle = 5'd6; // nand
                        6'b100011: controle = 5'd7; // nor
                        6'b110000: controle = 5'd12;// sle
                        6'b110001: controle = 5'd11;// slt
                        default: controle = controle;
                endcase
        end
        if(opcode != 6'd0) begin
                case (opcode)
                        6'b000001: controle = 5'd0;  // addi
                        6'b000010: controle = 5'd1;  // subi
                        6'b000011: controle = 5'd3;  //divi
                        6'b000100: controle = 5'd2;  //multi
                        6'b001001: controle = 5'd7;  //nori
                        6'b001010: controle = 5'd5;  //ori
                        6'b001011: controle = 5'd4;  //andi
                        6'b010000: controle = 5'd13; // blt
                        6'b011100: controle = 5'd12; // slei
                        6'b011110: controle = 5'd11; // slti
                        6'b100000: controle = 5'd13; // bgt
                        6'b110000: controle = 5'd8;  // beq
                        6'b111000: controle = 5'd9;  // bne
                        default: controle = controle;
                endcase
        end//else

        if(opULA != 2'b00) begin
                case(opULA)
                        2'b01: controle = 5'd0;
                        2'b10: controle = 5'd33;
                        2'b11: controle = 5'd31;
                        default: controle = controle;
                endcase
        end
end//always
endmodule
```

4.5.9 Unidade de Processamento

Após definidos todos os módulos presentes no processadoro passo final é juntá-lo todos em em um arquivo principal e fazer suas respectivas ligações, definindo os sinais de

entrada e saida globais do processador. No caso deste projeto, os sinais de entrada são 16 bits vindos das chaves presentes no FPGA que são referentes aos números dados pelo usuário na execução de um programa, o sinal de *clock* vindo diretamente do FPGA e um bit chamado 'Enter' respectivo ao ultimo *switch* (SW17) do FPGA, usado para definir a entrada. Já como saida tem-se 27 bits separados em 4 saídas de 7 bits (saida1, saida2, saida3 e saida4) referentes aos segmentos do display da FPGA.

```
1  module processador(dadosIN,
2                                                  enter,
3                                                  realClk,
4                                                  estado,
5                                                  disp4,
6                                                  disp3,
7                                                  disp2,
8                                                  disp1);
9
10 input realClk;
11 input enter;
12 input [8:0] dadosIN;
13 output [3:0] estado;
14 output [0:6] disp4, disp3, disp2, disp1;
15
16 wire [31:0] sregB, sMEM, saidaEXT, sregA, ULA1, ULA2, sULA,  valorPC, toOUT, carregaDados
       , sValorPC, sregULA, endereco, dadosMEM, mem, instr, dadosEscrita, sA, sB;
17 wire [4:0] regEscrita;
18
19 // SINAIS DE CONTROLE
20 //sinais de controle de memoria
21 wire EscrevePC, SetPC, LeMem, EscreveRI, EscreveReg, controleOUT, EscreveMem, ovrflw,
       zero;
22
23 //seletores de multiplexadores
24 wire SelMuxMem, SelReg1, SelReg2, SelMuxUlaA, SelMuxIn;
25 wire [1:0] SelMuxUlaB, SelMuxPC;
26
27 //sinal de controle da ULA
28 wire [4:0]controleULA;
29
30
31 // MODULOS
32
33 divisor divFreq(.clk(realClk),
34                                          .div_clk(clk));
35
36 registrador32b PC(.controle(EscrevePC),
37                                                .clk(clk),
38                                                .entrada(valorPC),
39                                                .saida(sValorPC)); // PC
40
41
42 mux2_32b MuxMem(.seletor(SelMuxMem),
43                                     .entrada1(sValorPC),
44                                     .entrada2(sULA),
45                                     .saida(endereco)); // para 'endereco' mem
46
47
```

```
48  memoria MEM(.dado(sregB),
49                                  .endereco(endereco[6:0]),
50                                  .write(EscreveMem),
51                                  .wclk(clk),
52                                  .rclk(clk),
53                                  .saida(carregaDados)); // memoria (mem)
54
55
56  mux2_32b MuxIn(.seletor(SelMuxIn),
57                                     .entrada1(carregaDados),
58                                     .entrada2({23'd0,dadosIN}),
59                                     .saida(sMEM)); // para r-mem
60
61
62  registrador32b Rmem(.controle(1'b1),
63                                                  .clk(clk),
64                                                  .entrada(sMEM),
65                                                  .saida(dadosMEM)); // registrador de
                                                        dados da memoria (r-mem)
66
67
68  registrador32b ri(.controle(EscreveRI),
69                                                .clk(clk),
70                                                .entrada(carregaDados),
71                                                .saida(instr)); // registrador de instr (
                                                    ri)
72
73
74  mux2_5b MuxReg1(.seletor(SelReg1),
75                                           .entrada1(instr[20:16]),
76                                           .entrada2(instr[15:11]),
77                                           .saida(regEscrita)); // para 'registrador de
                                               escrita b-reg
78
79
80  mux2_32b MuxReg2(.seletor(SelReg2),
81                                            .entrada1(sregULA),
82                                            .entrada2(dadosMEM),
83                                            .saida(dadosEscrita)); // para 'dados para
                                                escrita' b-reg
84
85
86  bancoReg banco(.escreve(EscreveReg),
87                                         .out(controleOUT),
88                                         .clk(clk),
89                                         .reg1(instr[25:21]),
90                                         .reg2(instr[20:16]),
91                                         .regF(regEscrita),
92                                         .dados(dadosEscrita),
93                                         .A(sA),
94                                         .B(sB),
95                                         .toOUT(toOUT)); // banco de registradores (b-reg)
96
97
98  moduloSaida ModOUT(.entrada(toOUT),
99                                                 .saida1(disp1),
100                                                .saida2(disp2),
101                                                .saida3(disp3),
102                                                .saida4(disp4),
103                                                .clk(clk)); // modulo de saida (out)
```

```
104
105
106 registrador32b A(.controle(1'b1),
107                                         .clk(clk),
108                                         .entrada(sA),
109                                         .saida(sregA)); // registrador A (a)
110
111
112 registrador32b B(.controle(1'b1),
113                                         .clk(clk),
114                                         .entrada(sB),
115                                         .saida(sregB)); // registrador B (b)
116
117
118 extensor EXT(.sinal(instr[15:0]),
119                                  .sinal_ext(saidaEXT)); // extensor de sinal (ext)
120
121
122 mux2_32b MuxUlaA(.seletor(SelMuxUlaA),
123                                         .entrada1(endereco),
124                                         .entrada2(sregA),
125                                         .saida(ULA1)); // para ula-1
126
127
128 mux4_32b MuxUlaB(.seletor(SelMuxUlaB),
129                                         .entrada1(sregB),
130                                         .entrada2(32'd1),
131                                         .entrada3(saidaEXT),
132                                         .entrada4(saidaEXT),
133                                         .saida(ULA2)); // para ula-2
134
135
136 ULA ALU(.A(ULA1),
137                   .B(ULA2),
138                   .clk(clk),
139                   .controle(controleULA),
140                   .saida(sULA),
141                   .overflow(ovrflw),
142                   .zero(zero)); // ULA
143
144
145 registrador32b saidaUla(.controle(1'b1),
146                                                                 .clk(~clk),
147                                                                 .entrada(sULA),
148                                                                 .saida(sregULA)); //
                                                                     Saida ULA (s-ula)
149
150
151 mux4_32b MuxPC(.seletor(SelMuxPC),
152                                    .entrada1(sULA),
153                                    .entrada2(sregULA),
154                                    .entrada3({26'b0, instr[6:0]}),
155                                    .entrada4(sregULA),
156                                    .saida(valorPC)); // para PC
157
158 //CONTROLE
159 ctrl_undd Controle(.opcode(instr[31:26]),
160                    .funct(instr[5:0]),
161                                                .zero(zero),
162                                                .enter(enter),
```

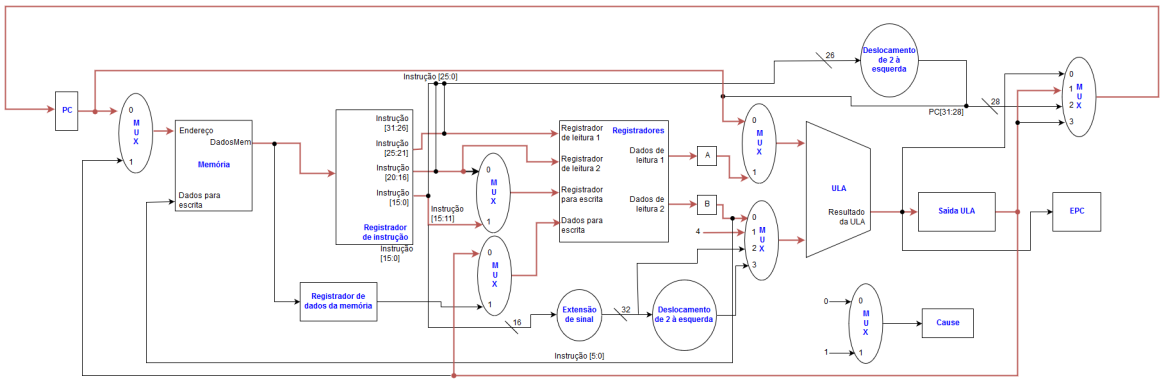
```
163                                         .clk(clk),
164                                         .estado(estado),
165                                         .SelMuxPC(SelMuxPC),
166                                         .EscrevePC(EscrevePC),
167                                         .SelMuxMem(SelMuxMem),
168                                         .EscreveMem(EscreveMem),
169                                         .EscreveRI(EscreveRI),
170                                         .SelMuxReg1(SelReg1),
171                                         .SelMuxReg2(SelReg2),
172                                         .EscreveReg(EscreveReg),
173                                         .SelMuxUlaA(SelMuxUlaA),
174                                         .SelMuxUlaB(SelMuxUlaB),
175                                         .controleULA(controleULA),
176                                         .SelMuxIn(SelMuxIn),
177                                         .controleOUT(controleOUT));
178 endmodule
```

4.6 Caminhos de Dados das Instruções

O último passo é definir como se comportarão as intruções, para isso é preciso saber o caminho que cada instrução ou conjunto de instruções vão seguir dentro dos módulos da unidade de processamento.

Começando pelo conjunto de instruções R, temos a Figura 11 que ilustra todos os caminhos que a instrução *add* toma durante seu processamento.

Figura 11 – *Datapath* da instrução *add*



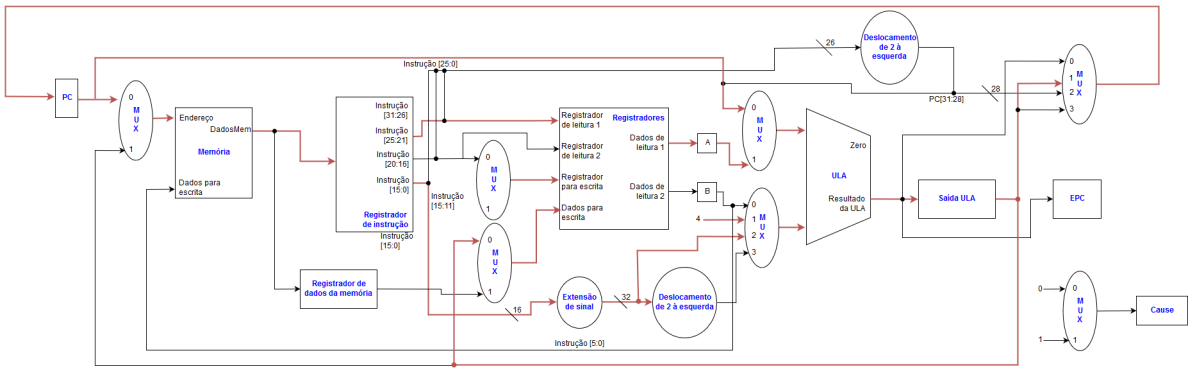
Fonte:O Autor

Para entender o processamento das instruções é necessário dividir os caminhos em ciclos, que são referentes aos ciclos de clock do processador. No caso da instrução *add* o primeiro ciclo se dá entre o PC e o Registrador de Instrução onde o endereço contido no PC é escolhido pelo multiplexador para ser buscado na Memória permitindo que a instrução possa ser escrita no Registrador de Instrução. Simulâneamente o valor do PC é enviado para a ULA para o endereço da próxima instrução ser calculado e salvo novamente no PC no início do próximo ciclo. O segundo ciclo é processado entre o Registrador de

Instrução e os registradores auxiliares A e B, onde a instrução guardada no registrador é desmembrada nas informações presentes na Tabela 1. Os endereços dos operandos e do registrador de destino são enviados para o Banco de Registradores possibilitando os valores dos operandos serem carregados nos registradores A e B.. Durante o quarto ciclo os operandos salvos nos registradores A e B são enviados à ULA e o resultado da operação é salvo no registrador Saída ULA. Por fim no quinto e último ciclo o consiste em apenas salvar o valor presente em Saída ULA no Banco de Registradores.

Como todas as instruções do tipo R são operações na ULA entre dois operandos presentes no Banco de Registradores e referenciados por endereços na palavra de instrução, o caminho de dados delas é o mesmo. Quanto à instruções do tipo I, todas as instruções lógicas, como *ori, andi,* e outras, e as instruções aritiméticas, como *addi, subi,* e outras, tem o caminho de dados muito parecido às instruções do tipo R sendo a única diferença estando no segundo ciclo do clock que ao invés de carrregar um valor no registrador B, o imediato é enviado diretamente à entrada do segundo operando da ULA, passando pelo extensor de sinal. Esse caminho pode ser visto na Figura 14.

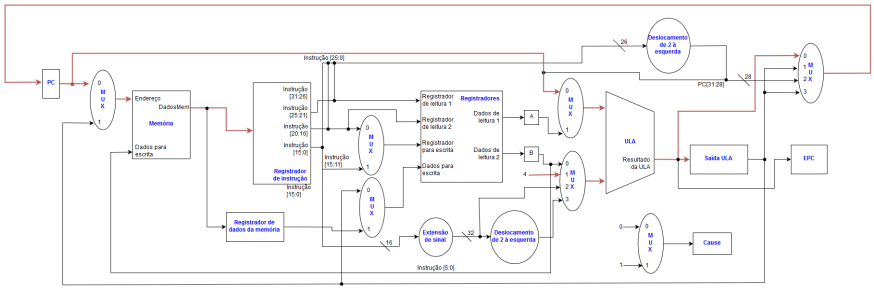
Figura 12 – *Datapath* da instrução *addi*



Fonte:O Autor

As intruções do tipo IN/OUT contam com apenas o primeiro ciclo, que calcula o valor da próxima instrução, pois após isso a Unidade de Controle manda um sinal de *halt* para o PC para que os dados de entrada ou saída possam ser processados pela unidade E/S.

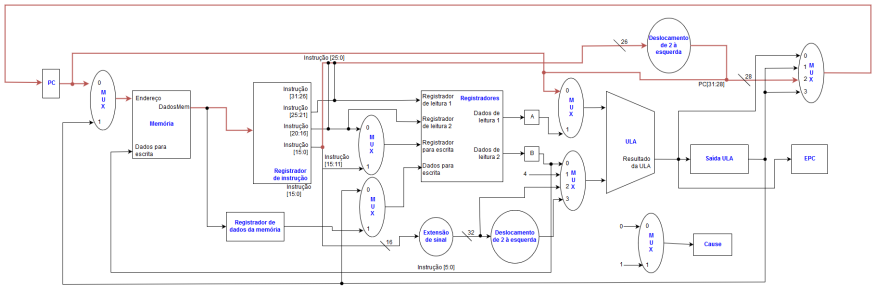
Figura 13 – *Datapath* da instrução *in*



Fonte:O Autor

E por fim tem-se as instruções do tipo J, que também contam com apenas dois ciclos, sendo o primeiro responsável por buscar as instruções na Memória e carregá-la no Registrador de Instruções como as outras instruções e o segudo grava o endereço do desvio no PC concatenando o valor de 28 bits presente na palavra de instrução com os quatro bits mais significativos do PC no inicio e dois bits 0 no fim.

Figura 14 – *Datapath* da instrução *jmp*



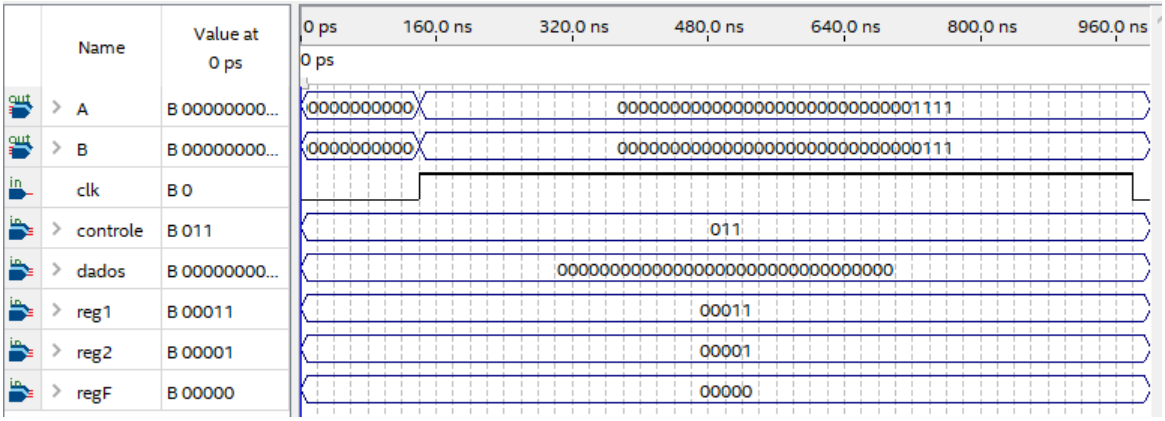
Fonte:O Autor

5 Resultados e Discussões

Unindo todas as unidades apresentadas no Capítulo 4 de *Desenvolvimento* é possivel ter uma boa idéia do funcionamento previsto do processador a ser desenvolvido, as unidades desenvolvidas em verilog foram testadas separadamente com valores de controle, entrada e saída pré-definidos usando o software Quartus Prime e é possivel verificar seu funcionamento correto à partir das formas de onda geradas pelo *software*.

Seguindo a ordem em que os códigos foram apresentados na Seção 4.5 primeiramente foi testado o Banco de Registradores salvando o valor 00000000000000000000000000001111 no registrador 00011 e o valor 00000000000000000000000000000111 no registrador 00001. É possível ver na forma de onda presente na Figura 15 que como os registradores 00001 e 000011 selecionados em reg1 e reg2 e com o sinal de controle habilitando a leitura, na subida do *clock* os valores salvos são mostrados corretamente em A e B.

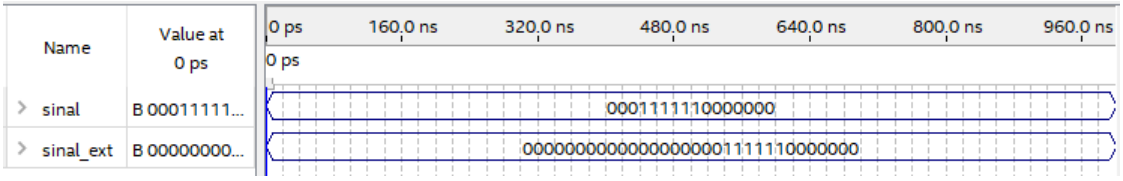
Figura 15 – Forma de onda do Banco de Registradores



Fonte:O Autor

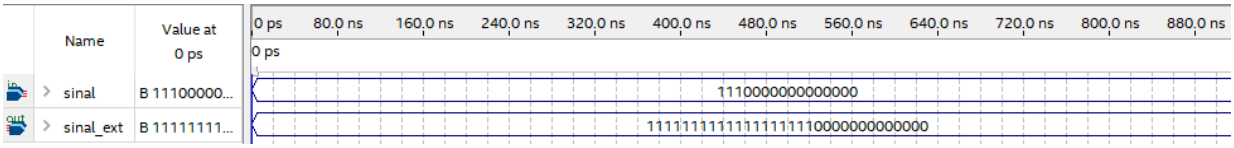
Em seguida o módulo testado foi o Extensor de sinal. É possível conferir nas Figuras 16 e 17 o funcionamento correto do mesmo, uma vez que com o bit mais significativo do sinal sendo 0, um sinal de 16 bits 0 é concatenado à ele, e o mesmo ocorre caso o bit mais significativo seja 1.

Figura 16 – Forma de onda do Extensor - Bit mais significativo 0



Fonte:O Autor

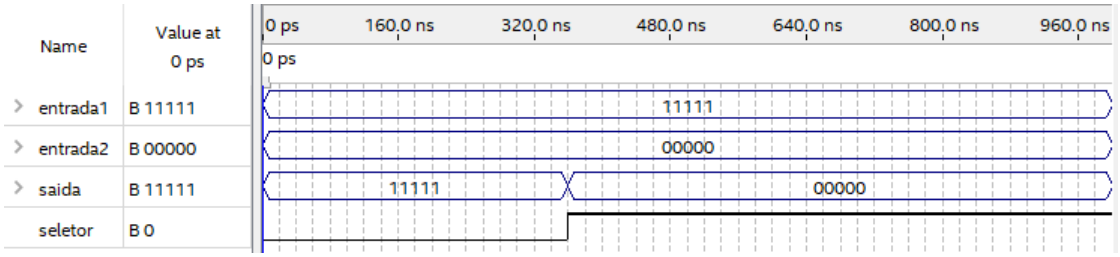
Figura 17 – Forma de onda do Extensor - Bit mais significativo 1



Fonte:O Autor

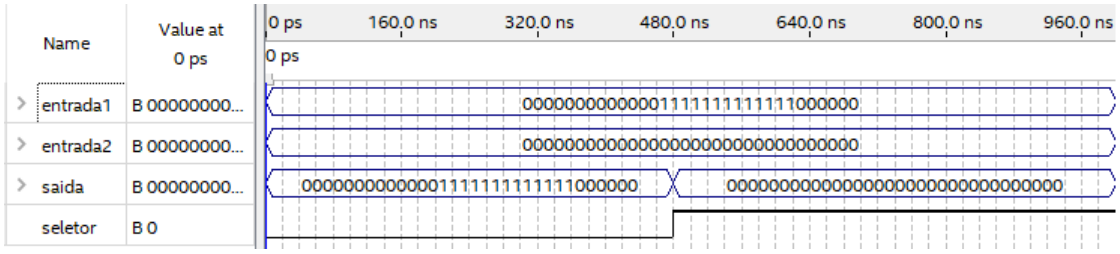
As Figuras 18, 19 e 20 contam com as formas de onda dos Multiplexadores e nela é visível a seleção dos sinais de entrada em função da chave seletora.

Figura 18 – Forma de onda do Multiplexador de 5 bits e 2 entradas



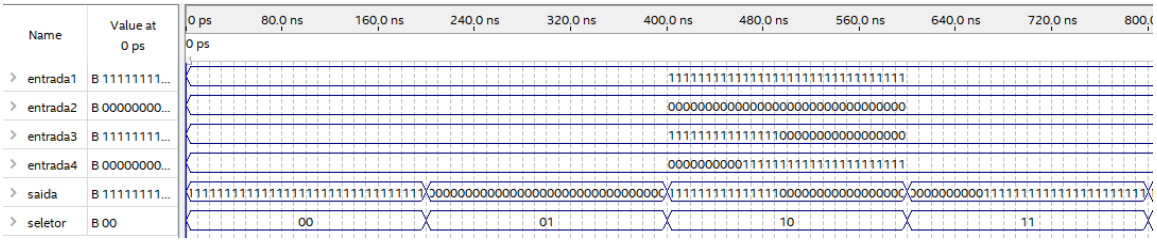
Fonte:O Autor

Figura 19 – Forma de onda do Multiplexador de 32 bits e 2 entradas



Fonte:O Autor

Figura 20 – Forma de onda do Multiplexador de 32 bits e 4 entradas

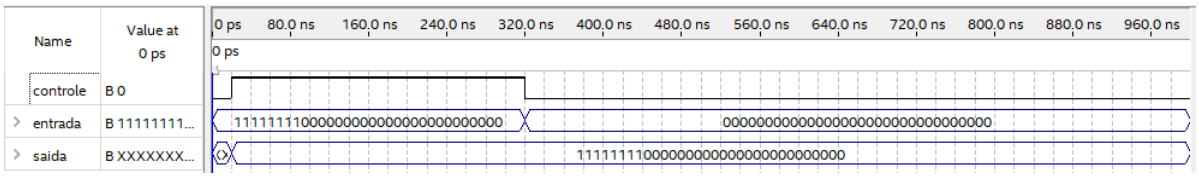


Fonte:O Autor

O módulo seguinte testado foi o do Registrador, que recebe primeirmente o valor 11111111000000000000000000000000 como entrada, e logo em seguida um pulso alto do

sinal de controle de escrita. É possivel ver na Figura 21 que na borda de subida do pulso de controle o valor presente na entrada é salvo no registrador e mesmo após a mudança do valor de entrada, o valor de saída não se altera.

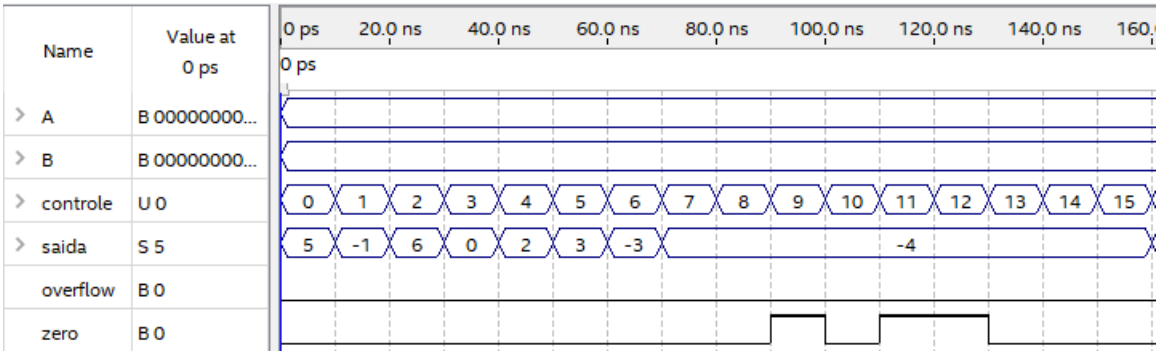
Figura 21 – Forma de onda do Registrador



Fonte:O Autor

Por útimo o módulo testado foi o da ULA inserindo como valor de controle uma contagem de 0 à 12 referente às instruções presentes na ULA, que podem ser vistas na Subseção 4.5.7. Também foram definidos os valores 3 para A e 2 para B. Os resultados podem ser vistos na Figura 22.

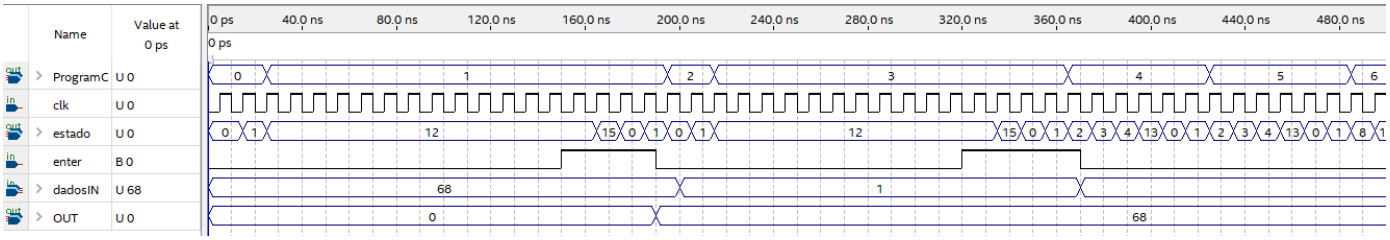
Figura 22 – Forma de onda da ULA



Fonte:O Autor

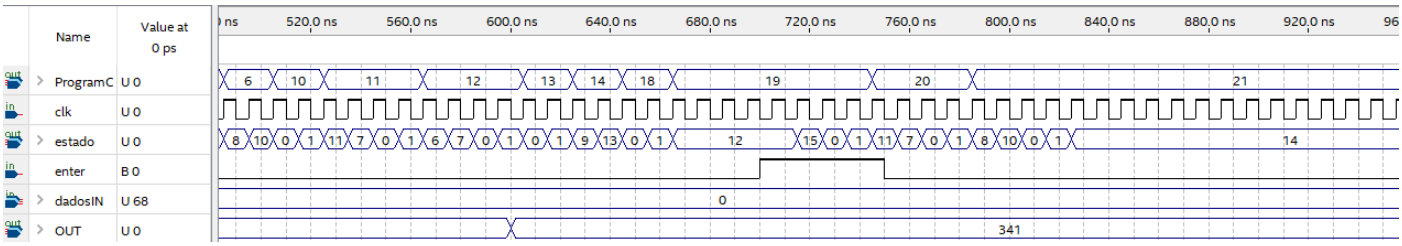
Depois do teste das unidades separadas foi feito o teste de todas as unidades juntas controladas pela Unidade de Controle. Nesse ponto foi possível verificar a funcionalidade completa do processador com uma sequência de instruções para teste, que tem o propósito de fazer a conversão de uma temperatura em graus Celcius para graus Fahrenheit ou Kelvin. Pela forma de onda, que pode ser vista nas Figuras 23 e 24, o funcionamento aconteceu como o previsto então o código foi carregado em um kit FPGA comprovando praticamente o funcionamento do processador que executou sem erros o algoritmo.

Figura 23 – Forma de onda 1 do código teste



Fonte:O Autor

Figura 24 – Forma de onda 2 do código teste



Fonte:O Autor

No teste acima, foi inserido o valor 68ºC e selecionado para a conversão de ºC para Kelvin, onde após todos os ciclos esperados, pode-se ver no campo de saída o valor , que é o correspondente correto em Kelvin para 69ºC.

6 Considerações Finais

Por fim, é possível afirmar que o projeto conseguiu alcançar completamente seu objetivo, que era de idealizar uma unidade de processamento completa contendo todos os módulos necessários para ser funcional, uma vez que o mesmo teve o compotamento esperado. As maiores dificuldades durante a realização se deram durante a fase de idealização do projeto, a escolha da arquitetura e das mudificações que precisariam ser feitas na mesma e a união e sincronização final dos módulos. Após realizadas as decisões críticas do inicio do projeto as demais etapas fluiram tranquilamente até o momento de sincronizar a unidade de controle com os outros módulos, onde foram encontrados diversos erros que custaram e ser corrigidos.

Apesar de completamente funcinal, ainda existem alterações que podem ser feitas no futuro, como a adição de instruções *Jump and Link* e *Jump Register*, juntas de uma pilha de recursão para executar tarefas recursivas e também inclusão dos registradores EPC e Cause para o controle de exceções.

Referências

1 PATTERSON, D. A.; HENNESSY, L. J. *Organização e Projeto de Computadores: A Interface Hardware/Software*. [S.l.]: Elsevier Editora Ltda., 2005. v. 3. Citado 4 vezes nas páginas 11, 18, 21 e 22.

2 CHICHOSZ, A. L. et al. *Modos de Endereçamento e Conjunto de Instruções*. Citado 3 vezes nas páginas 14, 15 e 16.

3 TRABALHO DE A.C. *Arquitetura Harvard*. Disponível em: <http://trabalhoac-tkv.blogspot.com/2010/11/arquitetura-harvard.html>. Acesso em: 13 de abril 2019. Citado na página 17.