RA: 136124

Confecção da Unidade de Processamento de um Sistema Computacional descrito em Verilog HDL

São José dos Campos - Brasil

Setembro de 2020

RA: 136124

Confecção da Unidade de Processamento de um Sistema Computacional descrito em Verilog HDL

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

Setembro de 2020

Resumo

Este projeto tem como finalidade a confecção e descrição de uma unidade de processamento de um sistema computacional projetado com características baseados em MIPS (*Microprocessor without interlocked pipeline stages*). Seus módulos foram desenvolvidos em Verilog HDL a fim de serem simulados em formas de ondas e futuramente, em uma placa FPGA (*Field Programmable Gate Array*) contida no ambiente de teste remoto. Essa unidade deve conseguir suportar e trabalhar com a arquitetura e conjunto de instruções idealizado no último ponto de checagem.

Palavras-chaves: Unidade de Processamento. Sistema Computacional. Verilog HDL.

Lista de ilustrações

Figura 1 – Portas Lógicas 13
Figura 2 – *Latch* Tipo D 14
Figura 3 – *Flip-Flops* 15
Figura 4 – Representação de um Microprocessador 15
Figura 5 – Arquitetura do MIPS 17
Figura 6 – Estrutura de uma DRAM 18
Figura 7 – Placa Altera DE2-115 18
Figura 8 – *Display* de 7 segmentos 20
Figura 9 – Arquitetura planejada 25
Figura 10 – Esquema da Unidade de Processamento 34
Figura 11 – Testes das operações da ULA 1 35
Figura 12 – Testes das operações da ULA 2 36
Figura 13 – Testes dos Registradores 36
Figura 14 – Teste do *Reset* dos Registradores 36
Figura 15 – Simulação da Memória de Dados 37
Figura 16 – Simulação da Memória de Instrução 37
Figura 17 – Testes do PC 38
Figura 18 – Testes do Mux 38
Figura 19 – Simulação do Extensor 38
Figura 20 – Teste do Módulo de Saída de Dados 39

Lista de tabelas

Tabela 1 – Operações em Verilog HDL 19
Tabela 2 – Conjunto de Instruções R 22
Tabela 3 – Conjunto de Instruções I 22
Tabela 4 – Conjunto de Instruções J 22
Tabela 5 – Conjunto de Instruções O 23
Tabela 6 – Formato das Instruções por Conjunto 23
Tabela 7 – Simplificação das Operações do Tipo R 23
Tabela 8 – Simplificação das Operações do Tipo I 24
Tabela 9 – Codificação das Operações da ULA 26

Lista de códigos

Código 1 Variáveis, sintaxe do *always* e exemplo de algumas operações da ULA . 27
Código 2 Banco de Registradores 28
Código 3 Memória de Dados 29
Código 4 Memória de Instruções 29
Código 5 Program Counter 29
Código 6 Incremento de endereço para a próxima instrução 30
Código 7 Multiplexador para dados 30
Código 8 Multiplexador para endereço de registradores 31
Código 9 Extensor 31
Código 10 Extensor para instrução de *Jump* 32
Código 11 Saída de dados 33
Código 12 Valores de inicialização da memória de instrução para simulação 37
Código 13 Unidade Lógica e Aritmética 45
Código 14 Template de Memória RAM de dados 49
Código 15 Template de Memória ROM de instruções 49
Código 16 Número Negativo 51
Código 17 Casas decimais 51
Código 18 Decodificador BCD 52
Código 19 Unidade de Processamento 53

Sumário

1 INTRODUÇÃO 9

2 OBJETIVOS 11
2.1 Geral 11
2.2 Específico 11

3 FUNDAMENTAÇÃO TEÓRICA 13
3.1 Elementos básicos de Circuitos Digitais 13
3.1.1 Portas Lógicas 13
3.1.2 Elementos de Memória 14
3.2 Processador 15
3.2.1 Arquitetura MIPS 16
3.3 Memórias 16
3.3.1 Memória Principal 17
3.4 *Field Programmable Gate Array* 18
3.5 Linguagem de Descrição de Hardware 19
3.6 *Display* de 7 segmentos 19

4 DESENVOLVIMENTO 21
4.1 Conjunto de Instruções 21
4.2 Arquitetura do sistema 24
4.3 Descrição dos módulos em Verilog HDL 26
4.3.1 Unidade Lógica e Aritmética 26
4.3.2 Banco de Registradores 27
4.3.3 Memórias 28
4.3.4 *Program Counter* 29
4.3.5 Multiplexadores 30
4.3.6 Extensores 31
4.3.7 Saída de Dados 32
4.4 Unidade de Processamento 33

5 RESULTADOS OBTIDOS E DISCUSSÕES 35
5.1 Formas de Onda para a ULA 35
5.2 Formas de Onda para o Banco de Registradores 36
5.3 Formas de Onda para a Memória de Dados 37
5.4 Formas de Onda para a Memória de Instruções 37

5.5 Formas de Onda para o PC 37
5.6 Formas de Onda para o Mux 38
5.7 Formas de Onda para o Extensor 38
5.8 Formas de Onda para a Saída de Dados 38

6 CONSIDERAÇÕES FINAIS 41

REFERÊNCIAS 43

APÊNDICE A – ULA.V 45

APÊNDICE B – TEMPLATE USADO PARA AS MEMÓRIAS . . . 49

APÊNDICE C – CONJUNTO DE MÓDULOS PARA A APRESENTAÇÃO DE DADOS 51

APÊNDICE D – UNIDADE DE PROCESSAMENTO 53

1 Introdução

O estudo de circuito digitais é de suma importância no mundo atual em que vivemos, nessa era tecnológica, uma grande parte de tudo o que fazemos em nosso dia a dia tem relação com algum aparelho eletrônico. Apesar de todos que usam tais dispositivos saberem quais são suas funcionalidades, muitos não sabem de que forma essa função é executada. A lógica por trás de todas essas diretrizes, são feitas por componentes chamados transistores que formam juntamente a outros aparatos elétricos, os circuitos de *hardware*. Grande quantidade de pessoas optaram por aprofundar seus conhecimentos nesta área ainda tão desconhecida para muitos, portanto a experiência ao se obter com a pesquisa, é tamanha ao ponto de poder desenvolver projetos e dispositivos que possam revolucionar a sociedade.

O computador, por exemplo, foi um grande marco, possuindo uma enorme quantidade de funções, ele é um aparelho indispensável nos dias de hoje. O que não sabemos é que houve uma grande escalada dos *hardwares* para chegarem ao que são na atualidade. Em meados dos anos 70, foi possível implementar todos os circuitos lógicos e elétricos em um único chip, chamado de microprocessador (BROWN; VRANESIC, 2009). Conhecido normalmente por CPU (*Central Processing Unit*), esta unidade foi uma evolução para a tão pouca conhecida, área eletrônica, é um componente que pode processar diversas instruções por segundo e processar uma enorme quantidade de dados.

Dentro do contexto de *hardwares* e computadores, temos a arquitetura que é a peça imprescindível para o entendimento do funcionamento das máquinas computacionais, esse termo se refere ao projeto e a definição da estrutura operacional desses sistemas, sendo as revolucionárias nessa área as arquiteturas RISC (*Reduced Instruction Set Computer*) que maximizaram o padrão do desempenho, forçando as anteriores a acompanhar este padrão ou a desaparecer (HENNESSY; PATTERSON, 2014). A MIPS (*Microprocessor without interlocked pipeline stages*) era uma delas, na época foi uma grande inovação e é lembrada até nos dias atuais, possui características de fáceis entendimentos, mas é muito eficiente, por isso é uma das primeiras a serem estudadas nas disciplinas de Arquitetura e Organização de Computadores.

A confecção de um sistema computacional é um grande objetivo e pode ser alcançado com muito preparo e cuidado. Usando os conceitos aprendidos e pesquisados, e também a arquitetura base e conjunto de instruções definido no último ponto de checagem, este relatório visa a descrição dos componentes da unidade de processamento usando uma linguagem de descrição de *hardware*.

2 Objetivos

2.1 Geral

Este projeto tem como finalidade a implementação da unidade de processamento do sistema computacional projetado através da linguagem de descrição de *hardware* Verilog para ser desenvolvido no Software Quartus.

2.2 Específico

- Descrição da ULA e banco de registadores;
- Confecção das unidades de memória de dados e de instruções;
- Implementação do PC, extensores de bits e multiplexadores;
- Descrição do módulo de saída;
- Simulação dos componentes separadamente;
- Integração dos módulos em uma única unidade.

3 Fundamentação Teórica

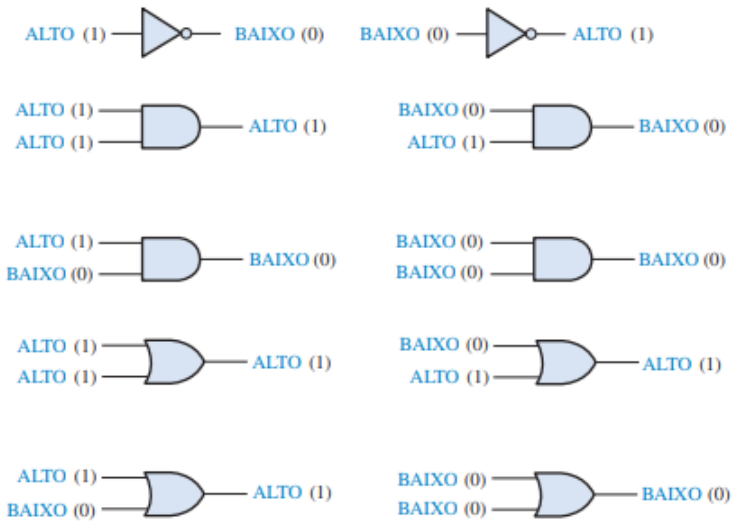
3.1 Elementos básicos de Circuitos Digitais

3.1.1 Portas Lógicas

Os circuitos digitais são compostos de diversos elementos que produzem a lógica necessária para que, o meio que o aplique, funcione corretamente. Podendo ser do tipo combinacional e sequencial, cada um possui suas características mas o que difere um do outro é que dados armazenados de saídas anteriores podem alterar as próximas.

As portas lógicas são as bases para todos os circuitos lógicos, são elas que fazem com que o nosso dispositivo funcione através de um fundamento. Como sabemos que muitos dos aparelhos utilizados funcionam com dados binários, ou seja, sinal alto e baixo, temos algumas operações que podemos fazer com os mesmos.

Figura 1 – Portas Lógicas



Fonte: Sistemas Digitais (FLOYD, 2007)

As três operações mais simples e mais usadas são a NOT, que faz com que o sinal de entrada se inverta na saída, a porta AND, que só possui sinal alto se ambas as entradas estejam em estado alto, e a OR, que tem sinal alto quando qualquer uma das duas possuem sinal alto. Podemos fazer um paralelo dessas lógicas com a matemática que conhecemos e então aplicá-las em uma álgebra chamada de Booleana, que nos ajudará a resolver e

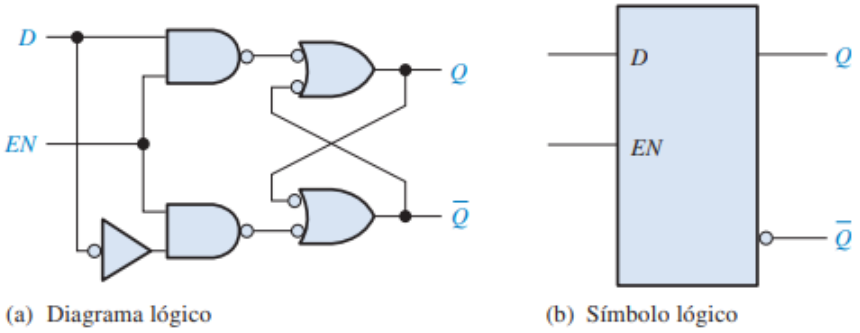
também simplificar circuitos muito complexos. A porta AND, indicada por (·), pode ser relacionada com uma multiplicação e a OR, indicada por (+), é comparado a soma.

3.1.2 Elementos de Memória

Outro tipo de dispositivo muito usado são aqueles que armazenam valores, chamados de memória e também mas conhecidos como *Latches* e *Flip-Flops*. Ambos têm a capacidade de armazenar um bit que normalmente são usados para realimentar a entrada e formar os conhecidos circuitos sequenciais, que são aqueles que as saídas interferem na lógica inicial.

O *Latch* é o componente de memória mas simples entre os dois, o seu funcionamento é devido a um circuito combinacional compacto porém com uma capacidade de armazenar um determinado bit, e o seu funcionamento ocorre segunda a ativação de sua chave de *enable*. O mesmo pode ser encontrado em diversos tipos, onde os mais vistos são o SR, que possui modos de setar (definir sinal alto na saída) ou resetar (definir sinal baixo), e também o tipo D, que será explicado juntamente ao próximo elemento.

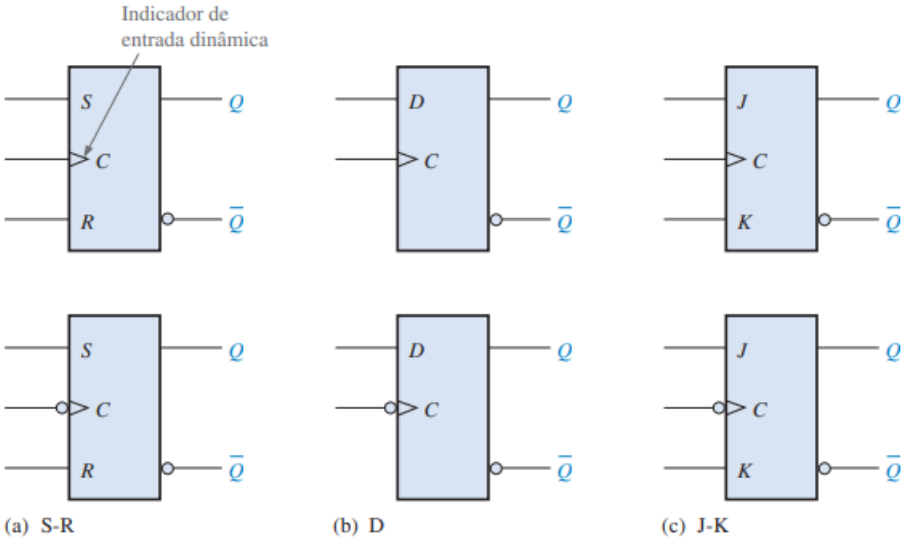
Figura 2 – *Latch* Tipo D



Fonte: Sistemas Digitais (FLOYD, 2007)

O mais conhecido e também mais usado são os *Flip-Flops*, que são parecidos com os *Latches*, porém seu funcionamento é sensível a uma entrada que chamamos de *clock*, que nada mais é uma onda que assume valores altos e baixos determinados por uma frequência que o mesmo possui. A sua ativação pode ser na subida (baixo para alto) ou na descida (alto para baixo) desses valores e além disso, os *Flip-Flops* existem de diversas funcionalidades. O do tipo D conduz a entrada para saída, ou seja, quando a entrada for 0 teremos esse valor na saída e o mesmo acontece para o bit 1, e também temos o do tipo SR que segue a mesma lógica do *Latch*. Já o JK, é mais completo, possuindo as funções de manter, setar e resetar, e também a de *Toggle* que também dá nome ao do tipo T, que nada mais é que a alternação do valor o mesmo possui.

Figura 3 – *Flip-Flops*

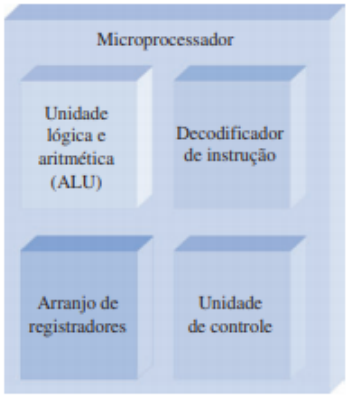


Fonte: Sistemas Digitais (FLOYD, 2007)

3.2 Processador

A CPU (*central processing unit*) é o "cérebro"do computador; ela supervisiona tudo que o computador faz (FLOYD, 2007). Suas principais funções são a de buscar e executar instruções armazenada na memória além de controlar todas as demais unidades que compõem um sistema computacional. Todos os programas e *softwares* usados são executados nesse componente. A simples instrução de soma de uma série de números é armazenada em formato de dados binários e processado pelo microprocessador. A Figura 4 representa o conteúdo de uma CPU que consiste basicamente de três unidades, uma unidade lógica e aritmética, a ULA, um banco de registradores e uma unidade de controle.

Figura 4 – Representação de um Microprocessador



Fonte: Sistemas Digitais (FLOYD, 2007)

Como o próprio nome do componente já diz, é uma unidade em que se pode fazer operações lógicas, como por exemplo AND e OR, e aritméticas, como somas e subtrações. A ULA em sistema computacional recebe os operandos para ser trabalhados de um banco de registradores e é controlada pela unidade de controle de uma CPU. Algumas das instruções podem requerer alguns sinais para que possam ser executadas, os códigos de condição, como a multiplicação, que usa operações sucessivas e deslocamento.

O banco de registradores de uma máquina é um componente em que se guardam dados que vão ser usados várias vezes e que necessitam de um acesso mais rápido. A memória de um sistema computacional possui um acesso mais restrito e mais lento que um registrador. Essa unidade possui um pequeno número de memórias para atender os requerimentos do sistema.

A unidade de controle é encarregada de manejar os demais componentes no processamentos de instruções, é ela que provê a cada unidade o que deve ser ativo e faz a sincronização do processo.

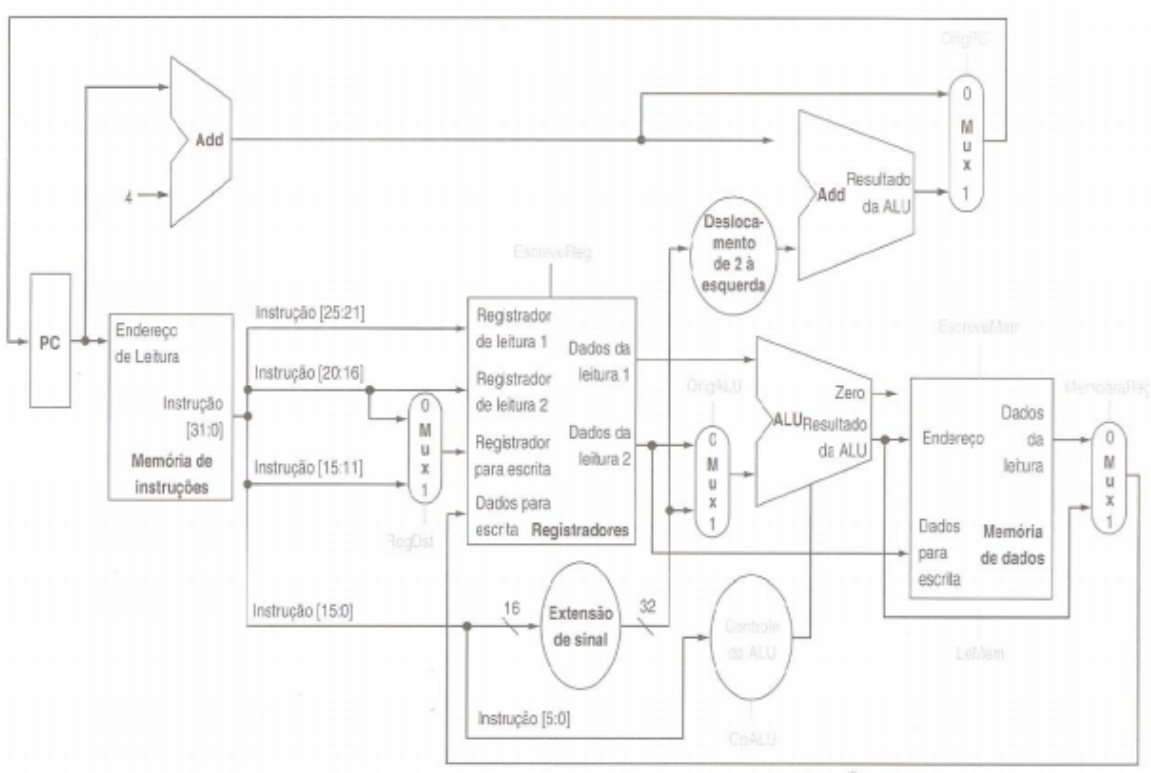
3.2.1 Arquitetura MIPS

As arquiteturas de computadores podem ser classificadas em dois tipos distintos, a RISC que possui um conjunto simples e menor de instruções, é mais rápido porém menos versátil, e a CISC (*Complex Instruction Set Computer*) que é capaz de executar uma vasta gama de operações complexas e por isso é muito versátil. A Arquitetura MIPS vista na Figura 5 é uma das mais básicas e de fácil entendimento, esta é baseada em registradores e trabalha com dados de 32 bits. Sua estrutura foi feita para adequar-se à instruções que trabalham em um único ciclo de *clock*, a mesma também pode ser estruturada em multiciclo, mas não será utilizada nesse projeto.

3.3 Memórias

Um computador pode possuir diversos tipos de memórias, cada uma com funções específicas e características diferentes. Porém como o nome já diz, todos elas servem para o armazenamento de dados e informações, a sua capacidade de armazenamento e velocidade de acesso são particularidades que as define, temos por exemplo, os registradores que estão no mais baixo nível e localizadas mais próximas as CPU, por isso são mais rápidas mas com um baixa capacidade de guardar dados. Quanto mais alto o seu nível, maior seu armazenamento e menor sua velocidade, tendo em seguida as caches, a principal e a secundária.

Figura 5 – Arquitetura do MIPS



Fonte: Organização e Projeto de Computadores (PATTERSON, 2005)

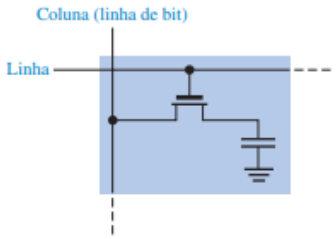
3.3.1 Memória Principal

Esta é a que será utilizada no sistema computacional para o armazenamento de dados e instruções. Uma delas é a RAM (*Random-Access Memory*), uma memória volátil, significando que a informação é perdida se a alimentação for desligada ou faltar energia (FLOYD, 2007). Os dados podem ser escritos e lidos a qualquer momento.

São dividas em dois grupo, as SRAM (*Static RAM*) e as DRAM (*Dynamic* RAM). A memória estática usa *flip-flops*, pode ser visto na Figura 2, para o armazenamento de bits e podem ser lidas seguindo certas condições lógicas, já as dinâmicas, fazem a utilização de capacitores para o armazenamento, vide Figura 6, caracterizando-a com um simples circuito, porém na leitura de dados, é necessário o que chamamos de *refresh*, uma técnica para re-armazenar o valor lido uma vez que ele está em formato de sinal elétrico.

Outro tipo de memória principal utilizada é a ROM (*Read-Only Memory*), que é uma memória não volátil, ou seja, não perde os dados armazenados na ausência de energia. Dados também podem ser lidos e escritos a qualquer momento.

Figura 6 – Estrutura de uma DRAM



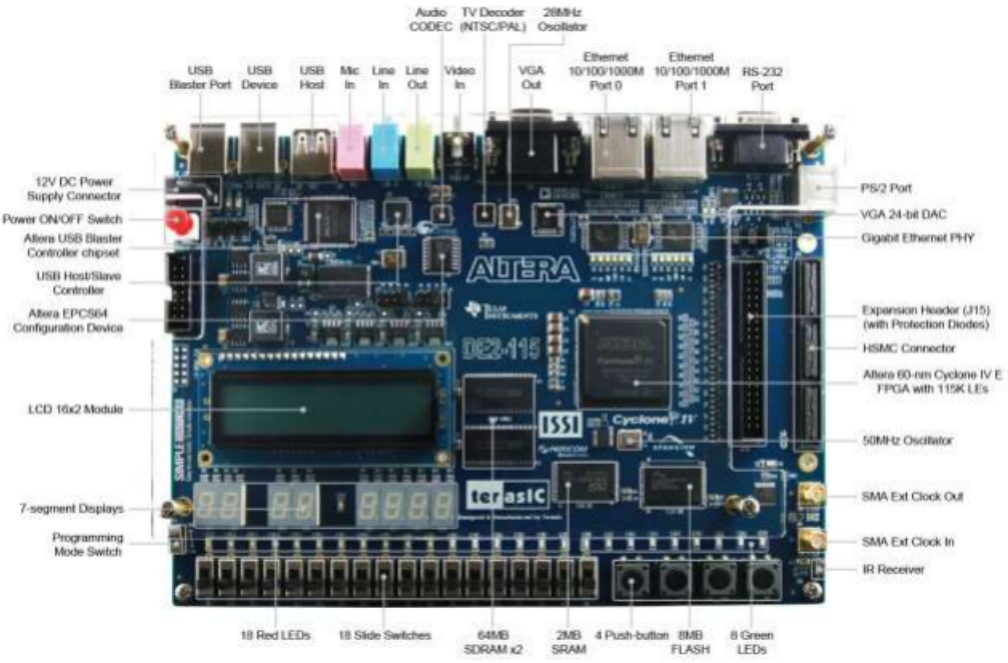
Fonte: Sistemas Digitais (FLOYD, 2007)

3.4 *Field Programmable Gate Array*

O *Field Programmable Gate Array* ou FPGA é um circuito integrado que pode ser programado e configurado por um usuário. O mesmo suporta uma grande quantidade de circuitos lógicos e tem como finalidade diversas aplicações que está a disposição do programador, podendo ser encontrado acompanhado de diversos componentes utilitários, tais como chaves, botões, LEDs, entre outros, para o teste dos seus circuitos.

Pode então ser comercializado em um formato de placa, contendo vários tipos de aparelhos que podem ser utilizados para fins educacionais, como usado em nossa universidade. Nela podemos utilizar o FPGA da Altera, modelo DE2-115 como visto na Figura 7.

Figura 7 – Placa Altera DE2-115



Fonte: Altera DE2-115 User Manual (ALTERA..., 2012)

3.5 Linguagem de Descrição de Hardware

A HDL é um tipo de linguagem que permite a construção de um circuito lógico em código, possuindo sua própria sintaxe e aplicação. Muito usado para modelar circuitos de grande porte, que se forem feitos em esquemático dariam mais trabalho e podem muitas vezes ser mais complexo.

Dentro desse conceito, esta linguagens possuem diversas ramificações tais como, o VHDL e o Verilog HDL. Para este projeto utilizaremos o Verilog HDL que possui sua própria sintaxe e pode nos ajudar na implementação da Máquina de Estados.

A linguagem descreve toda a lógica usada em circuitos digitais em regras chamadas de sintaxe, podemos ver alguns exemplos de operações lógicas na Tabela 1.

Tabela 1 – Operações em Verilog HDL

OPERAÇÕES	VERILOG
AND	&
OR	|
NOT	~
NAND	~&
NOR	~|
XOR	^
XNOR	~^

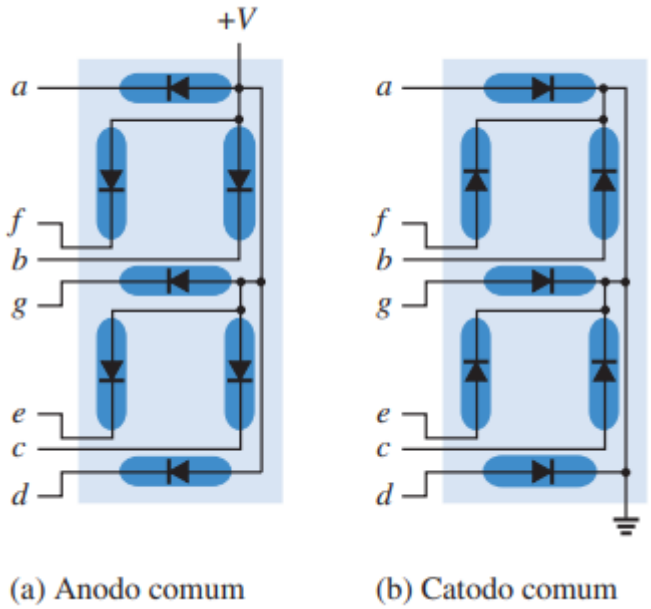
Fonte: Autor

O Verilog também possui semelhanças com algumas linguagens de programação, podemos então traçar uim paralelo, de alguns dos casos, como por exemplo uma comparação dada por *if-else* ou então *cases* são a mesma, porém em casos de *loop* o Verilog HDL possue a sintaxe de *always*.

3.6 *Display* de 7 segmentos

O *Display* de 7 segmentos é conjunto de LEDs que juntos podem formar números e letras de acordo com uma certa entrada. Possui 7 segmentos e 1 ponto, o que faz com que a variedade de informação a ser projetada seja grande. O display pode ser do tipo cátodo ou ânodo comum, o que significa que ou os LEDs estão com os terminais positivos em comum ou então os negativos, portanto para cada tipo, o valor de entrada para ligá-los é diferente.

Figura 8 – *Display* de 7 segmentos



(a) Anodo comum (b) Catodo comum

Fonte: Sistemas Digitais (FLOYD, 2007)

4 Desenvolvimento

A arquitetura projetada será baseada em um MIPS, que é caracterizado como um sistema RISC, possuindo então uma gama de instruções menor e menos complexas. Estas ações devem ser processadas em um único ciclo de *clock* para que haja uma facilidade na manipulação da unidade de controle a ser desenvolvida. O sistema computacional também trabalhará com números de 32 bits de tamanho, tanto de dados quanto de instruções. Os únicos modos de endereçamentos usados, serão o de registrador, imediato e direto.

4.1 Conjunto de Instruções

As instruções planejadas devem ser adequadas para a conversão e utilização de uma linguagem de programação com quaisquer necessidades, tais como problemas de repetições e até o Fibonacci. O formato das mesmas foram ajustados para que possam abranger diversas características que possam ter, juntamente a arquitetura que foi estruturada para suportá-la.

Estas foram divididas em conjuntos que as representam e para que possamos classificá-las. O conjunto R, trabalha com instruções que utilizam apenas registradores como operandos, a do tipo I, também faz o uso de registradores e de valores imediatos ou de endereços. Já o J operará apenas com o endereço de uma certa instrução e o conjunto O será designado para operações diversas.

Para o processador entender a instrução que ele deve executar, cada uma possui uma identificação que caracteriza a sua operação chamado de *opcode*. Cada um dos conjuntos e seus respectivos detalhes foram separados para melhor entendimento. Para o tipo R, veja a Tabela 2, para o I, veja a Tabela 3, para o J, veja a Tabela 4, e para o O, veja a Tabela 5.

Tabela 2 – Conjunto de Instruções R

Opcode	Instrução	Exemplo
00000	ADD	RD <= RS + RT
00001	SUB	RD <= RS - RT
00010	MUL	RD <= RS * RT
00011	DIV	RD <= RS / RT
00100	NOT	RD <= ¬RS
00101	AND	RD <= RS · RT
00110	OR	RD <= RS + RT
00111	XOR	RD <= RS ⊕ RT
01000	SLT	RD <= RS < RT
01001	SGT	RD <= RS > RT
01010	JR	PC <= RD
01011	IN	RD <= INPUT
01100	OUT	OUTPUT <= RD

Tabela 3 – Conjunto de Instruções I

Opcode	Instrução	Exemplo
01101	ADDI	RD <= RS + IM17
01110	SUBI	RD <= RS - IM17
01111	BEQ	PC <= ENDEREÇO SE RD = RS
10000	BNEQ	PC <= ENDEREÇO SE RD ≠ RS
10001	LW	RD <= DADOS[ENDEREÇO]
10010	SW	DADOS[ENDEREÇO] <= RD
10011	SR	RD <= DESLOC(RS) P/ DIREITA
10100	SL	RD <= DESLOC(RS) P/ ESQUERDA

Tabela 4 – Conjunto de Instruções J

Opcode	Instrução	Exemplo
10101	J	PC <= ENDEREÇO
10110	JAL	PC <= ENDEREÇO

Tabela 5 – Conjunto de Instruções O

Opcode	Instrução	Exemplo
10111	NOP	NENHUMA OPERAÇÃO
11000	HLT	PARAR CPU

O formato das instruções e a quantidade de bits necessárias para cada membro está descrito na Tabela 6. Cada conjunto possui uma certa quantidade de possibilidades para atender a todas as possíveis operações a serem feitas.

Tabela 6 – Formato das Instruções por Conjunto

	Bits				
Tipo	31 - 27	26 - 22	21 - 17	16 - 12	11 - 0
R	Opcode	RD / 0	RS / 0	RT / 0	0
I	Opcode	RD	RS / ENDEREÇO / 0	IM17 / ENDEREÇO / 0	
J	Opcode	ENDEREÇO			
O	Opcode	0			

Cada instrução possui uma limitação ao uso de operandos, para cada conjunto possui alguns subconjuntos em que se separa o que cada conteúdo da instrução leva. Para a do tipo R, há algumas operações que fazem o uso de todos os 3 registradores, enquanto outras levam 2 ou apenas 1, vide Tabela 7.

Tabela 7 – Simplificação das Operações do Tipo R

	Bits				
Subconjunto	31 - 27	26 - 22	21 - 17	16 - 12	11 - 0
ADD, SUB, MUL, DIV, AND, OR, XOR, SLT, SGT	Opcode	RD	RS	RT	0
NOT	Opcode	RD	RS	0	
JR, IN, OUT	Opcode	RD	0		

Para o tipo I, as operações aritméticas trabalham com 2 registradores e um imediato de 17 bits, as de verificação de igualdade tem 2 registradores e um endereço de 17 bits, as de deslocamento possuem apenas 2 registradores e as que usam a memória de dados, apenas 1 registrador e um endereço para a mesma de 22 bits. Essa simplificação pode ser vista de melhor forma na Tabela 8.

Tabela 8 – Simplificação das Operações do Tipo I

	Bits				
Subconjunto	31 - 27	26 - 22	21 - 17	16 - 12	11 - 0
ADDI, SUBI	Opcode	RD	RS	IM17	
BEQ, BNEQ	Opcode	RD	RS	ENDEREÇO	
SR, SL	Opcode	RD	RS	0	
LW, SW	Opcode	RD	ENDEREÇO		

Para as instruções do tipo J e O, ambas as operações de cada conjunto operam da forma descrita na Tabela 6.

4.2 Arquitetura do sistema

Atendendo o requerido para a criação da arquitetura, todo o planejamento de instruções e dos dados foram aplicados para criar a representação do sistema apresentado na Figura 9.

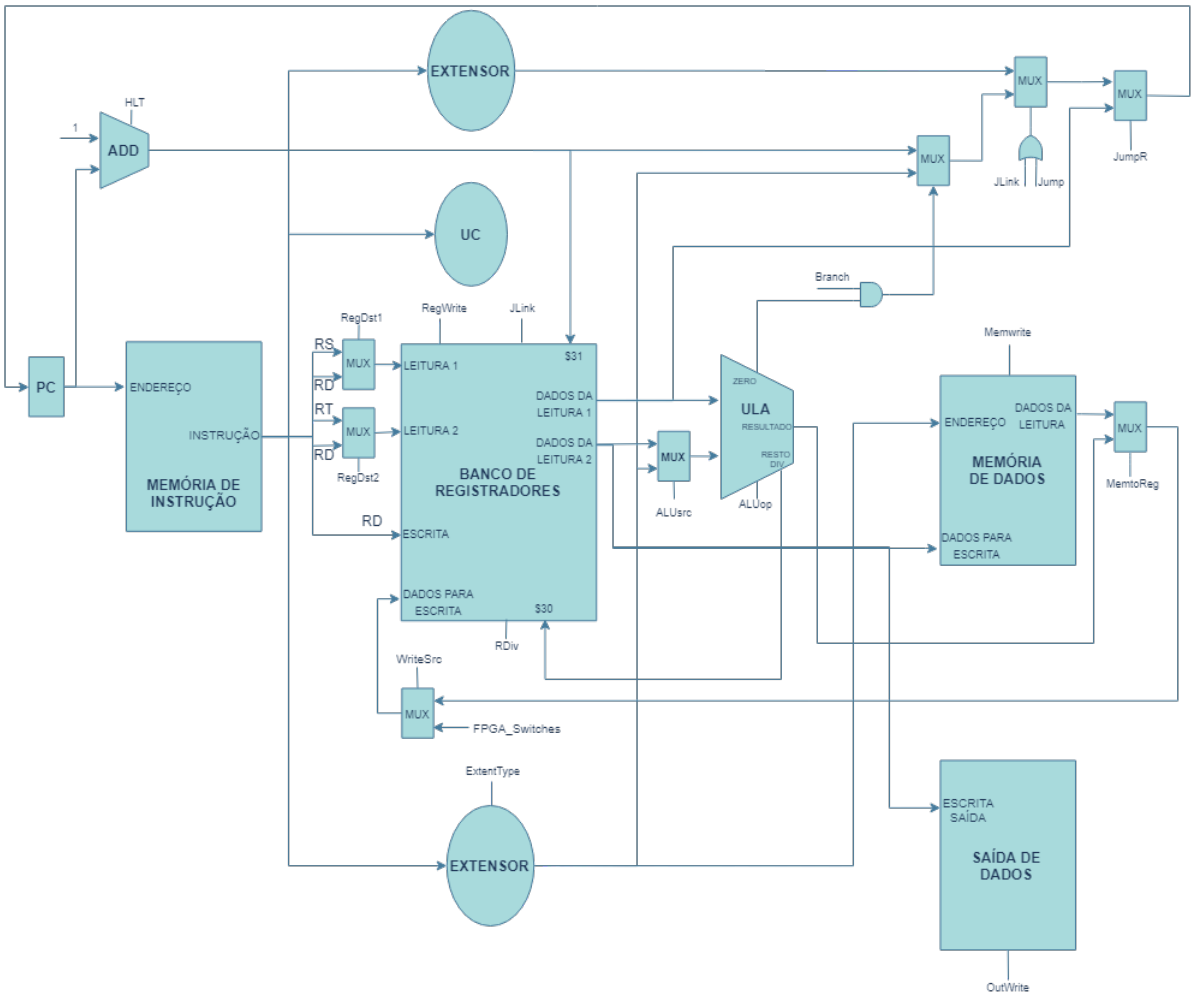
Cada unidade possui uma função dependendo da operação a ser realizada. O PC é um registrador que armazena o endereço da instrução que está para ser processada, e para que possa avançar para a próxima é necessário incrementar seu valor, porém há outras maneiras de mudar de endereço que é caso haja alguma instrução que altere o mesmo.

A memória RAM está divida em duas, uma para as instruções e outra para os dados, a de instruções que armazenam todas as operações da CPU e poderão ser escritas com um arquivo .txt, e a de dados que armazenam valores e podem ser acessadas através de ações que podem ocasionar em leituras e escritas.

O banco de registradores é uma unidade onde se pode armazenar valores e que possuam maior facilidade e agilidade de acesso do que a RAM. Ela possui um total de 32 registradores que são nada mais que *Flip-Flops* do tipo D , sendo que o primeiro registrador possuirá sempre o valor zero, o de número 31 será para o uso da instrução DIV e o registrador 32 para a utilização da operação JAL que armazenará o endereço da instrução a seguir. Todos os dados trabalhados por esse componentes e outros estão em um formato de 32 bits.

A ULA que fará as operações lógicas e aritméticas com dados que virão dos registradores ou diretamente das instruções. Para o uso de imediatos e endereços, deverão passar por um extensor de bits que ajusta a quantidade necessária de dígitos para que possam ser trabalhados. A ULA também possui campos que produzirão certos resultados que ajudarão em outros casos, como o zero no caso das instruções BEQ e BNEQ e o resto da divisão para o DIV.

Figura 9 – Arquitetura planejada



Fonte: Autor, usando a plataforma (DIAGRAMS. . . , 2020)

A entrada de dados no sistema é feita através de *Slides-Switches*, esse valor é armazenada em um registrador passado usando a instrução IN, como a quantidade de chaves é limitada a 18, ocasiona uma limitação na entrada de dados. A apresentação de dados através da interface de comunicação com o FPGA será feita pela unidade de saída de dados com a operação OUT, que utilizará quatro *displays* de sete segmentos, com o primeiro display mostrando o sinal de negativo caso necessário e outros três para dígitos, limitando assim os números a um intervalo de 0 a 999, ou seja 10 bits correspondentes a 10 chaves de entrada no FPGA.

Para a manipulação de todas essas unidades, temos a de controle que ficará encarregada em decidir os sinais dos demais componentes para que executem corretamente determinada instrução, além de escolher as saídas de cada multiplexador e portas lógicas.

4.3 Descrição dos módulos em Verilog HDL

Após a criação da arquitetura e definição de todo o conjunto de instruções e modos de endereçamento a ser empregados, a descrição de cada módulo da unidade de processamento do sistema computacional começou a ser produzido, seguindo as especificações e necessidades do projeto. Veja as subseções de 4.3.1 a 4.3.7 para a melhor compreensão de cada componente e sua implementação.

4.3.1 Unidade Lógica e Aritmética

Esta unidade provê uma vasta gama de operações lógicas e aritméticas para com os dados do sistema, além de fornecer informações para tomadas de decisões dentre outras para análise das instruções. A ULA possui um total de 14 ações que podem ser feitas, vide Tabela 9, portanto, para sua codificação foram necessários 4 bits, chamados de ALUop, estes que serão comandados pela Unidade de Controle.

Tabela 9 – Codificação das Operações da ULA

ALUop	Instrução referente
0000	ADD / ADDI
0001	SUB / SUBI
0010	MUL
0011	DIV
0100	NOT
0101	AND
0110	OR
0111	XOR
1000	SLT
1001	SGT
1010	BEQ
1011	BNEQ
1100	SR
1101	SL

A ULA trabalha com dados de tamanho de 32 bits e tem como resultados também números com 32 bits, além da saída das operações, existe um campo chamado zero, que ajuda na tomada de decisão de instruções do tipo *branch*, e um campo chamado de RESTODiv que armazena o resto da operação de divisão e a guarda no registrador de número 31 conforme necessidade. As instruções serão realizadas sempre que o código da operação ou os dados sofrerem alguma alteração. Um trecho de sua descrição pode ser vista no Código 1, e seu complemento pode ser encontrado no Apêndice A.

```
1  module ULA(ALUop, D1, D2, RESULTADO, ZERO, RESTOdiv);
2
3          input [3:0] ALUop;
4          input [31:0] D1, D2;
5          output reg [31:0] RESULTADO, RESTOdiv;
6          output reg ZERO;
7
8          always @(ALUop or D1 or D2) begin
9                  case(ALUop[3:0])
10                         4'b0000: // ADD/ADDI
11                         begin
12                                 RESULTADO = D1 + D2;
13                                 RESTOdiv = 0;
14                                 ZERO = 0;
15                         end
16
17                         4'b0001: // SUB/SUBI
18                         begin
19                                 RESULTADO = D1 - D2;
20                                 RESTOdiv = 0;
21                                 ZERO = 0;
22                         end
23
24                         4'b0010: // MUL
25                         begin
26                                 RESULTADO = D1 * D2;
27                                 RESTOdiv = 0;
28                                 ZERO = 0;
29                         end
30
31                         4'b0011: //DIV
32                         begin
33                                 RESULTADO = D1 / D2;
34                                 RESTOdiv = D1 - (D1 / D2) * D2;
35                                 ZERO = 0;
36                         end
```

Código 1 – Variáveis, sintaxe do *always* e exemplo de algumas operações da ULA

4.3.2 Banco de Registradores

O banco de registradores é uma pequena memória que foi projetada para armazenar 32 dados de 32 bits, sua descrição pode ser encontrada no Código 2. Na arquitetura desenvolvida poderão ser lidos dois registradores simultaneamente de forma contínua e ser escrito em apenas um, conforme o *clock* do sistema. Quando inicializada, possuem valor 0 do tamanho de 32 bits, e podem ser alteradas conforme necessidades das instruções. Um certo dado pode sobrescrever o conteúdo de um registrador quando seu endereço de escrita está selecionado e o sinal de controle RegWrite comandado pela Unidade de Controle possui um sinal alto. Os sinais de controle RDiv e JLink, servem para o armazenamento de informação da instrução DIV e JAL, respectivamente.

Como foram projetados 32 registradores, será necessário um total de 5 bits para a representação do endereço de cada um, sendo que o primeiro registrador, 00000 em binário,

sempre armazenará o valor zero, o trigésimo primeiro, 11110 em binário, para guardar o resto da divisão da respectiva instrução e o trigésimo segundo, 11111 em binário, para guardar o endereço de *link* da instrução JAL. O banco também encontra uma função *reset* para a reinicialização do conteúdo dos registradores.

```
1  module BancoDeRegs(RegLeit1, RegLeit2, RegEscrita, DadoEscrita, R30, R31, RegWrite, RDiv,
       JLink, CLK, Reset, DadoLeit1, DadoLeit2);
2
3          input [4:0] RegLeit1, RegLeit2, RegEscrita;
4          input [31:0] DadoEscrita, R30, R31;
5          input CLK, RegWrite, RDiv, JLink, Reset;
6          reg [31:0] REG [31:0];
7          output [31:0] DadoLeit1, DadoLeit2;
8          integer i;
9
10         initial begin
11                 for(i = 0; i < 32 ; i = i + 1)
12                         REG[i] = 0;
13         end
14
15         always @(posedge CLK) begin
16                 if(Reset == 1)
17                 begin
18                         for(i = 0; i < 32 ; i = i + 1)
19                         REG[i] = 0;
20                 end
21
22                 if(RegWrite == 1)
23                         REG[RegEscrita] = DadoEscrita;
24
25                 if(RDiv == 1)
26                         REG[30] = R30;
27
28                 if(JLink == 1)
29                         REG[31] = R31;
30         end
31
32         assign DadoLeit1 = REG[RegLeit1];
33         assign DadoLeit2 = REG[RegLeit2];
34 endmodule
```

Código 2 – Banco de Registradores

4.3.3 Memórias

A memória é um unidade em que se guarda dados para que possam ser usados no processamento de instruções com uma vasta gama de posições para o armazenamento. A mesma deve poder receber um receber um endereço e retornar um dado contido naquele espaço, além de ter a possibilidade de poder sobrescrever algum possível valor já nele contido ao comando de um sinal de controle MemWrite. Toda essas ações serão reproduzidas conforme o *clock* do sistema.

A memória de dados foi projetada para suportar números de até 32 bits e possui 2^{10} posições de armazenamento que são inicializados como 0. Vista no Código 3, é um módulo em que instancia-se uma memória RAM de um *template* disponível no Quartus com os parâmetros que foram definidos, que pode ser encontrada no Apêndice B, que possa ler e escrever dados em posições da memória.

```
1 module DMem(Endereco, DadoEscrita, MemWrite, DadoLeit, CLK);
2
3         input [31:0] Endereco, DadoEscrita;
4         input MemWrite, CLK;
5         output [31:0] DadoLeit;
6
7         RAM_Dados RAM_Dados(.data(DadoEscrita), .addr(Endereco[9:0]), .we(MemWrite), .clk
              (CLK), .q(DadoLeit));
8
9 endmodule
```

Código 3 – Memória de Dados

A memória de instruções possui as mesmas especificações que a de dados, porém ela não possibilitará sobrescrever dados. Os endereços da memória serão inicializados com dados de um arquivo .txt que conterão as instruções necessárias para realização de algum programa. O Código 4 contém a descrição deste módulo que faz o uso do *template* de memória ROM também presente no Apêndice B.

```
1 module IMem(Endereco, Instrucao, CLK);
2
3         input [31:0] Endereco;
4         input CLK;
5         output [31:0] Instrucao;
6
7         ROM_Instrucao ROM_Instrucao(.addr(Endereco[9:0]), .clk(CLK), .q(Instrucao));
8
9 endmodule
```

Código 4 – Memória de Instruções

4.3.4 *Program Counter*

O *Program Counter* ou apenas PC, descrito no Código 5 é um módulo que se guarda o endereço da instrução atual da memória de instruções. A cada subida do *clock*, essa contagem é atualizada conforme a necessidade das operações realizadas. Possui também um sinal de *Reset* caso seja necessário uma reinicialização do sistema.

```
1 module PC(Prox, Atual, Reset, CLK);
2
3         input [31:0] Prox;
4         input Reset, CLK;
```

```
5         output reg [31:0] Atual;
6
7         initial begin
8                 Atual = 0;
9         end
10
11        always @(posedge CLK) begin
12                Atual = Prox;
13
14                if(Reset == 1)
15                        Atual = 0;
16
17        end
18
19  endmodule
```

Código 5 – Program Counter

Na descrição presente no Código 6, possui a transição para o próximo endereço que, caso não haja uma instrução de *jump* ou *branch*, apenas incrementa 1. Esse módulo que também trabalha com a instrução HLT, parando o processo.

```
1   module SomaPC(Endereco, HLT, Saida);
2         input [31:0] Endereco;
3         input HLT;
4         output reg [31:0] Saida;
5
6         always@(Endereco) begin
7                 if(HLT==1)
8                         Saida = Endereco;
9                 else
10                        Saida = Endereco + 1;
11        end
12
13  endmodule
```

Código 6 – Incremento de endereço para a próxima instrução

4.3.5 Multiplexadores

Os multiplexadores, ou apenas mux, foram separados em dois tipos, um para a seleção de dados de tamanho de 32 bits e outro para a seleção de endereçamento de registradores. Os mesmos serão controlados pela unidade de controle respectivos sinais de comandos. O Código 7 apresenta a descrição do mux de dados e o Código 8 apresenta a descrição do mux de registradores.

```
1   module Mux(Entrada0, Entrada1, Selecao, Saida);
2
3         input [31:0] Entrada0, Entrada1;
4         input Selecao;
5         output reg [31:0] Saida;
6
```

```
7         always @(*) begin
8                 case (Selecao)
9                         0: Saida = Entrada0;
10                        1: Saida = Entrada1;
11                endcase
12        end
13
14endmodule
```

Código 7 – Multiplexador para dados

```
1 module MuxReg(Entrada0, Entrada1, Selecao, Saida);
2
3         input [4:0] Entrada0, Entrada1;
4         input Selecao;
5         output reg [4:0] Saida;
6
7         always @(*) begin
8                 case (Selecao)
9                         0: Saida = Entrada0;
10                        1: Saida = Entrada1;
11                endcase
12        end
13
14endmodule
```

Código 8 – Multiplexador para endereço de registradores

4.3.6 Extensores

As instruções do tipo imediato, podem passar operandos que possuem uma quantidade de bits menor do que a trabalhada pelos dados e endereços do sistema, por esse motivo o módulo extensor produzido, tem como finalidade ajustar a quantia de bits para cada tipo de dado.

O Código 9 descreve o extensor usado para instruções do tipo I, possuindo dois casos distintos divididos de acordo com o tamanho do número, que serão selecionados por sinais de controle da unidade de controle, para decisão da quantia de zeros a serem adicionados. Também devemos levar em conta os números negativos, visto que para não alterar suas informações , deve-se adicionar uns à esquerda.

```
1 module Extensor(Entrada0, Entrada1, ExtentType, Saida);
2
3         input [16:0] Entrada0;
4         input [21:0] Entrada1;
5         input ExtentType;
6         output reg [31:0] Saida;
7
8         always @(*) begin
9                 case(ExtentType)
10                0: begin
11                                Saida = Entrada0;
12                                if(Entrada0[16] == 1)
```

```
13                                          Saida = Saida + 32'
                                                b11111111111111100000000000000000;
14                          end
15                  1: begin
16                                  Saida = Entrada1;
17                                  if(Entrada1[21] == 1)
18                                          Saida = Saida + 32'
                                                b11111111110000000000000000000000;
19                          end
20                  endcase
21          end
22
23  endmodule
```

Código 9 – Extensor

O sistema computacional também possui um extensor para os endereços passados pelas instruções do tipo J que possui a mesma funcionalidade do extensor descrito acima. Como visto no Código 10, os dados possuem apenas a quantia de 27 bits, por isso, não se faz necessário um sinal de controle de seleção.

```
1   module ExtensorJump(Entrada, Saida);
2
3           input [26:0] Entrada;
4           output reg [31:0] Saida;
5
6           always @(*) begin
7                   Saida = Entrada;
8                   if(Entrada[26] == 1)
9                           Saida = Saida + 32'b11111000000000000000000000000000;
10          end
11
12  endmodule
```

Código 10 – Extensor para instrução de *Jump*

4.3.7 Saída de Dados

O módulo de saída de dados corresponde a instrução OUT, que apresenta em 4 *display* de 7 segmentos do tipo ânodo comum presentes no FPGA de testes, um dado contido em um endereço do banco de registradores. Ao processador computar a instrução de saída de dados, mandará um sinal de controle OutWrite que fará com que o dado recebido possa ser mostrado nos *displays*. O dado então passa por um módulo que verifica o sinal do número e atribui ao primeiro display o sinal de negativo ou não e também o converte do complemento de dois. O módulo de divisão de casas, separa os dígitos a serem apresentados usando um algoritmo de conversão de binário para BCD (*binary-coded decimal*), um número em binário que representa um único dígito em decimal. E por fim, as casas decimais codificadas em binário são decodificadas em um binário para sua representação no *display* de 7 segmentos. O Código 11 apresenta a descrição desta unidade

e os módulos necessários para sua implementação podem ser encontrados no Apêndice C.

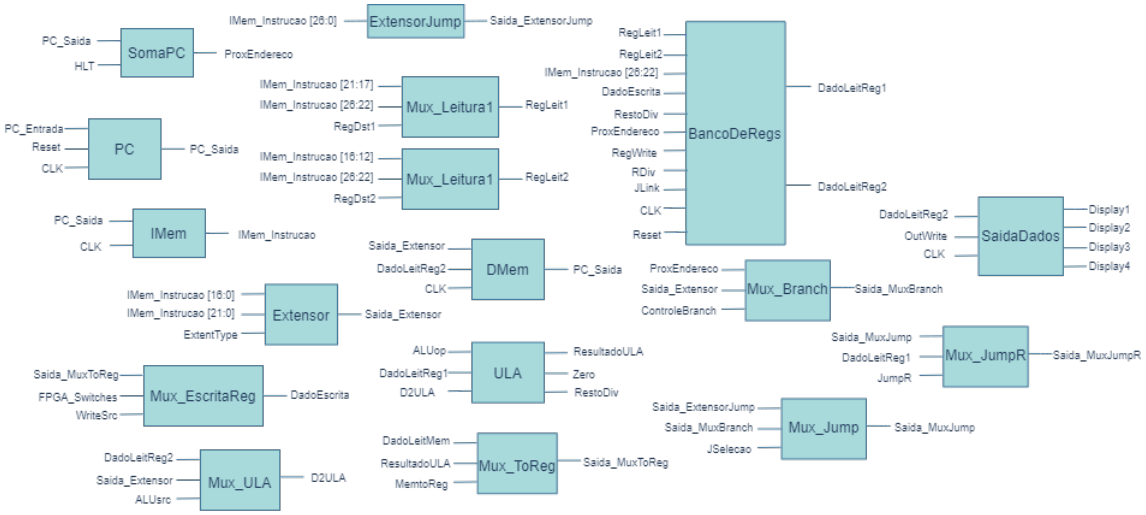
```
1  module SaidaDados(Dado, OutWrite, CLK, Display1, Display2, Display3, Display4);
2
3          input [31:0] Dado;
4          wire [3:0] Centena, Dezena, Unidade;
5          wire [6:0] HEXc, HEXd, HEXu, HEXn;
6          wire [31:0] BIN;
7          input OutWrite, CLK;
8          output reg [6:0] Display1, Display2, Display3, Display4;
9
10         initial begin
11                 Display3 = 7'b1111111;
12                 Display2 = 7'b1111111;
13                 Display1 = 7'b1111111;
14                 Display4 = 7'b1111111;
15         end
16
17         Negativo Negativo(.NUM(Dado), .BIN(BIN), .HEX(HEXn));
18         DivisaoCasas DivisaoCasas(.NUM(BIN[9:0]), .C(Centena), .D(Dezena), .U(Unidade));
19         DecodBCD DecodBCD_Centena (.NUM(Centena), .HEX(HEXc));
20         DecodBCD DecodBCD_Dezena (.NUM(Dezena), .HEX(HEXd));
21         DecodBCD DecodBCD_Unidade (.NUM(Unidade), .HEX(HEXu));
22
23         always @(posedge CLK) begin
24                 if (OutWrite == 1) begin
25                         Display3 = HEXc;
26                         Display2 = HEXd;
27                         Display1 = HEXu;
28                         Display4 = HEXn;
29                 end
30         end
31
32 endmodule
```

Código 11 – Saída de dados

4.4 Unidade de Processamento

A junção de todos os módulos para a descrição da unidade de processamento podem ser encontradas no Apêndice D. Nela constam a instanciação e conexão de todas as unidades descritas na Seção 4.3 da arquitetura projetada. A Figura 10 representa a esquematização e as ligações da unidade de processamento.

Figura 10 – Esquema da Unidade de Processamento



Fonte: Autor, usando a plataforma (DIAGRAMS. . . , 2020)

5 Resultados Obtidos e Discussões

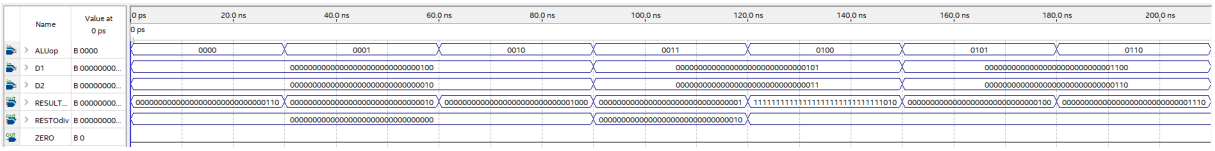
Com o projeto concluído pode-se então fazer alguns testes no Quartus Prime usando as formas de ondas, para a verificação da funcionalidade de cada módulo e da unidade de processamento.

5.1 Formas de Onda para a ULA

Presente na Figura 11, pode-se perceber os seguintes resultados para as operações:

- 0000 (ADD/ADDI): 100 + 10 = 110;
- 0001 (SUB/SUBI): 100 - 10 = 10;
- 0010 (MUL): 100 * 10 = 1000
- 0011 (DIV): 101 / 11 = 1, com RestoDIV = 10;
- 0100 (NOT): (101) = 010, com os zeros a esquerda negados a 1;
- 0101 (AND): 1100 & 110 = 100;
- 0110 (OR): 1100 | 110 = 1110.

Figura 11 – Testes das operações da ULA 1

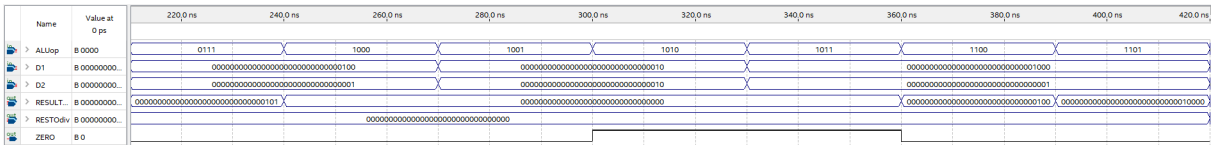


Os testes para as demais operações podem ser encontradas na Figura 12, conseguindo os seguintes resultados:

- 0111 (XOR): 100 $\oplus$ 1 = 101;
- 1000 (SLT): 100 < 1 = 0;
- 1001 (SGT): 10 > 10 = 0;
- 1010 (BEQ): 10 = 10 = 1 (Sinal de saída ZERO);
- 1011 (BNEQ): 1000 != 1 = 1 (Sinal de saída ZERO);

- 1100 (SR): SR(1000) um bit = 100;
- 1101 (SL): SL(1000) um bit = 10000.

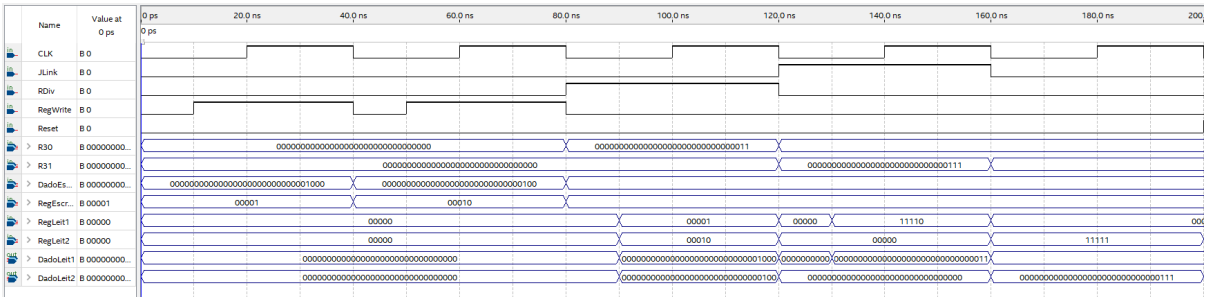
Figura 12 – Testes das operações da ULA 2



5.2 Formas de Onda para o Banco de Registradores

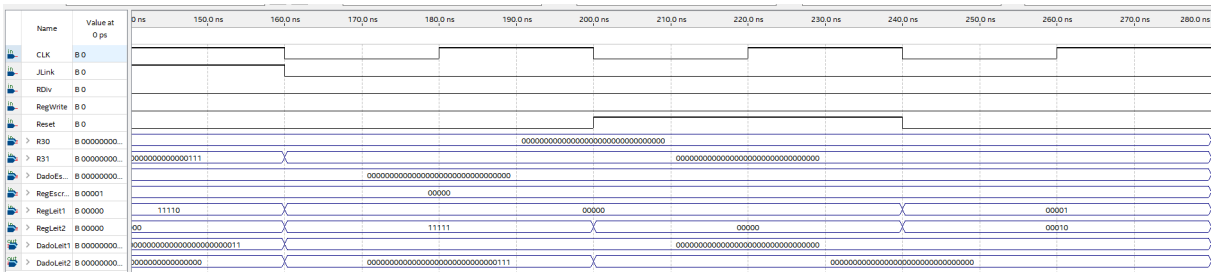
A Figura 13 apresenta a simulação do banco de registradores. Entre o tempo de 0 a 80 ns, o valor 8 em binário é escrito no registrador número 1 na primeira subida de *clock* e o valor 4 em binário é escrito no registrador 2 na segunda subida de *clock*. Na próxima subida do *clock* os registradores 1 e 2 são lidos e apresentados pelos dados de leitura 1 e 2 respectivamente, nesta mesma subida de *clock*, o registrador 31 recebe o valor 3 ao sinal de comando RDiv que está em sinal alto. Entre o tempo de 120 e 160 ns, o registrador de número 31 é lido apresentando o valor armazenado, e é escrito no registrador 32, o valor 7 com o sinal de JLink com 1. E por fim, o registrador 32 é lido constando o valor 7.

Figura 13 – Testes dos Registradores



Entre o tempo de 200 e 240 ns, como visto na Figura 14, o sinal de *reset* está em valor alto, e no próximo *clock*, ao ler novamente os registradores 1 e 2, seus valores voltaram ao valor 0.

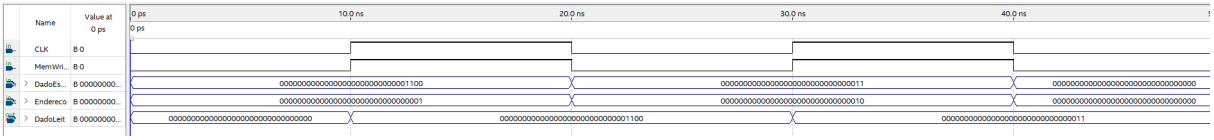
Figura 14 – Teste do *Reset* dos Registradores



5.3 Formas de Onda para a Memória de Dados

A simulação da memória de dados pode ser encontrada na Figura 15, para sua realização, na primeira subida de *clock* o valor 12 é alocado no endereço 1 da memória e lido ao mesmo tempo. O mesmo acontece na segunda subida de *clock*, onde é simulado o número 3 sendo armazenado na posição 2 e lido simultaneamente.

Figura 15 – Simulação da Memória de Dados



5.4 Formas de Onda para a Memória de Instruções

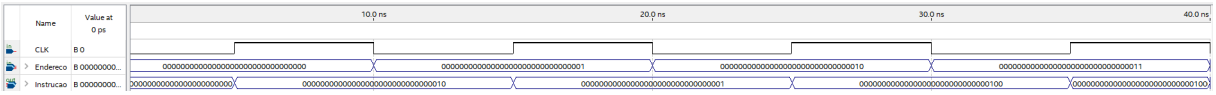
Para o teste de funcionalidade da memória de instrução, os dados presentes no Código 12 foram usados para a simulação, os mesmos são armazenados na ordem de posicionamento, ou seja, a primeira linha é alocada na primeira posição, a segunda na próxima, e assim por diante.

```
1 00000000000000000000000000000010
2 00000000000000000000000000000001
3 00000000000000000000000000000100
4 00000000000000000000000000001000
```

Código 12 – Valores de inicialização da memória de instrução para simulação

Na Figura 16, pode-se visualizar que a cada subida de *clock*, com uma contagem dos endereços de 0 até 3 em binário, o conteúdo do arquivo de inicialização estão neles contidos.

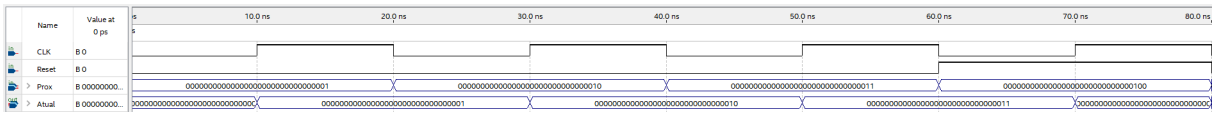
Figura 16 – Simulação da Memória de Instrução



5.5 Formas de Onda para o PC

A simulação do *Program Counter*, encontrada na Figura 17, consta de um contador para verificação se a cada subida de *clock*, o sinal que armazena o número atual é atualizado. A função de *reset* pode ser visualizada ao fim da onda, quando o atual volta ao valor 0.

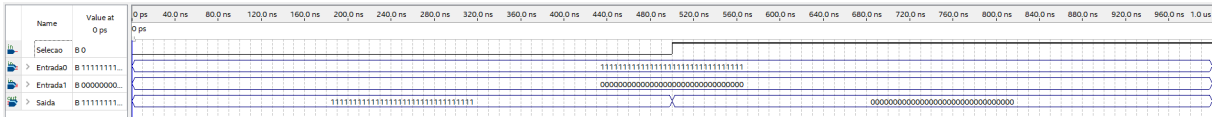
Figura 17 – Testes do PC



5.6 Formas de Onda para o Mux

O teste para os multiplexadores, vide Figura 18, consiste em variar o sinal de seleção e verificar qual valor é passado para a saída.

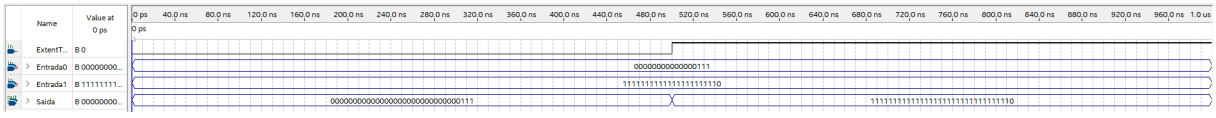
Figura 18 – Testes do Mux



5.7 Formas de Onda para o Extensor

Na Figura 19, pode-se visualizar que dependendo do sinal de tipo de extensão, a saída corresponde as especificações, onde verifica-se a utilização do extensão do complemento de dois.

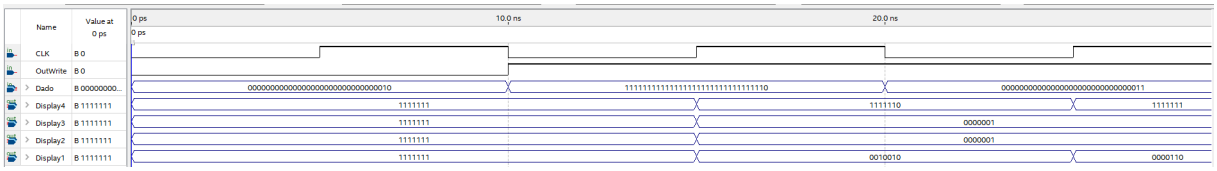
Figura 19 – Simulação do Extensor



5.8 Formas de Onda para a Saída de Dados

Para a simulação de valores sendo apresentados nos *displays* de 7 segmentos, foram escolhidos 3 números, para o primeiro *clock* tem-se o número 2 em binário, para o seguinte, o número -2 representado em complemento de dois e por fim o número 3. Como visto na Figura 20, na primeira subida de *clock*, o sinal de controle que escreve os números está em valor 0, logo nada é escrito, e devido os *displays* serem do tipo ânodo comum, devem possuir valor 1 para todos os bits. Na segunda subida de *clock*, o valor de escrita está em alto, logo o valor -2 será mostrado, o display 4 representa o sinal de negativo, e no display 1 consta a casa da unidade do número decodificado. Para o último *clock*, o número 3 é mostrado, com o sinal de negativo e os demais *displays* de centena e dezenas apagados e o de unidade decodificado com o valor 3.

Figura 20 – Teste do Módulo de Saída de Dados



6 Considerações Finais

Com a finalização da descrição da unidade de processamento e todos os módulos que a formam, foi possível obter um entendimento melhor em relação a arquitetura e organização de computadores. Os caminhos que cada instrução e seus dados tomam devem ser levados em conta para a projeção dos componente em Verilog. Consegue-se perceber através das formas de ondas, que cada um deles estão funcionando conforme o planejado. Para o próximo ponto de checagem deve ser possível visualizar o sistema computacional completo e o seu funcionamento, devido a acoplagem da unidade de controle e visualização no ambiente de testes do FPGA.

Referências

ALTERA DE2-115 User Manual. 2012. Disponível em: <https://www.intel.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-1404062209-de2-115-user-manual.pdf>. Citado na página 18.

BROWN, S. D.; VRANESIC, Z. *Fundamentals of Digital Logic with VHDL Desgin*. 3rd edition. ed. New York: McGraw-Hill, 2009. Citado na página 9.

DIAGRAMS.NET. 2020. Disponível em: <https://app.diagrams.net>. Citado 2 vezes nas páginas 25 e 34.

FLOYD, T. L. *Sistemas Digitais: Fundamentos e Aplicações*. 9ª edicao. ed. Porto Alegre: Bookman, 2007. Citado 6 vezes nas páginas 13, 14, 15, 17, 18 e 20.

HENNESSY, J. L.; PATTERSON, D. A. *Organização e Projeto de Computadores: Uma Abordagem Quantitativa*. 5ª edicao. ed. [S.l.]: Elsevier, 2014. Citado na página 9.

PATTERSON, D. A. *Organização e Projeto de Computadores, A interface hardware/software*. 3ª edicao. ed. [S.l.: s.n.], 2005. Citado na página 17.

APÊNDICE A – ULA.v

```
1  module ULA(ALUop, D1, D2, RESULTADO, ZERO, RESTOdiv);
2
3          input [3:0] ALUop;
4          input [31:0] D1, D2;
5          output reg [31:0] RESULTADO, RESTOdiv;
6          output reg ZERO;
7
8          always @(ALUop or D1 or D2) begin
9                  case(ALUop[3:0])
10                         4'b0000: // ADD/ADDI
11                         begin
12                                 RESULTADO = D1 + D2;
13                                 RESTOdiv = 0;
14                                 ZERO = 0;
15                         end
16
17                         4'b0001: // SUB/SUBI
18                         begin
19                                 RESULTADO = D1 - D2;
20                                 RESTOdiv = 0;
21                                 ZERO = 0;
22                         end
23
24                         4'b0010: // MUL
25                         begin
26                                 RESULTADO = D1 * D2;
27                                 RESTOdiv = 0;
28                                 ZERO = 0;
29                         end
30
31                         4'b0011: //DIV
32                         begin
33                                 RESULTADO = D1 / D2;
34                                 RESTOdiv = D1 - (D1 / D2) * D2;
35                                 ZERO = 0;
36                         end
37
38                         4'b0100: // NOT
39                         begin
40                                 RESULTADO = ~D1;
41                                 RESTOdiv = 0;
42                                 ZERO = 0;
43                         end
44
45                         4'b0101: // AND
46                         begin
47                                 RESULTADO = D1 & D1;
48                                 RESTOdiv = 0;
49                                 ZERO = 0;
50                         end
51
52                         4'b0110: // OR
53                         begin
54                                 RESULTADO = D1 | D2;
```

```
55                              RESTOdiv = 0;
56                              ZERO = 0;
57                      end
58
59                      4'b0111: //XOR
60                      begin
61                              RESULTADO = D1 ^ D2;
62                              RESTOdiv = 0;
63                              ZERO = 0;
64                      end
65
66                      4'b1000: // SLT
67                      begin
68                              if(D1 < D2)
69                                      RESULTADO = 1;
70                              else
71                                      RESULTADO = 0;
72                              RESTOdiv = 0;
73                              ZERO = 0;
74                      end
75
76                      4'b1001: // SGT
77                      begin
78                              if(D1 > D2)
79                                      RESULTADO = 1;
80                              else
81                                      RESULTADO = 0;
82                              RESTOdiv = 0;
83                              ZERO = 0;
84                      end
85
86                      4'b1010: // BEQ
87                      begin
88                              if(D1 == D2)
89                                      ZERO = 1;
90                              else
91                                      ZERO = 0;
92                              RESTOdiv = 0;
93                              RESULTADO = 0;
94                      end
95
96                      4'b1011: // BNEQ
97                      begin
98                              if(D1 != D2)
99                                      ZERO = 1;
100                             else
101                                     ZERO = 0;
102                             RESTOdiv = 0;
103                             RESULTADO = 0;
104                     end
105
106                     4'b1100: // SR
107                     begin
108                             RESULTADO = D1 >> 1;
109                             RESTOdiv = 0;
110                             ZERO = 0;
111                     end
112
113                     4'b1101: // SL
114                     begin
```

```
115                                 RESULTADO = D1 << 1;
116                                 RESTOdiv = 0;
117                                 ZERO = 0;
118                             end
119
120                     endcase
121
122             end
123
124     endmodule
```

Código 13 – Unidade Lógica e Aritmética

APÊNDICE B – Template usado para as memórias

Template de memória RAM usado na unidade de memória de dados.

```
module RAM_Dados
#(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=10)
(
        input [(DATA_WIDTH-1):0] data,
        input [(ADDR_WIDTH-1):0] addr,
        input we, clk,
        output [(DATA_WIDTH-1):0] q
);
        reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

        reg [ADDR_WIDTH-1:0] addr_reg;

        initial
        begin : INIT
                integer i;
                for(i = 0; i < 2**ADDR_WIDTH; i = i + 1)
                        ram[i] = {DATA_WIDTH{1'b0}};
        end

        always @ (posedge clk)
        begin
                if (we)
                        ram[addr] <= data;

                addr_reg <= addr;
        end

        assign q = ram[addr_reg];

endmodule
```

Código 14 – Template de Memória RAM de dados

Template de memória RAM usado na unidade de memória de instruções.

```
module ROM_Instrucao
#(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=10)
(
        input [(ADDR_WIDTH-1):0] addr,
        input clk,
        output reg [(DATA_WIDTH-1):0] q
);

        reg [DATA_WIDTH-1:0] rom[2**ADDR_WIDTH-1:0];

        initial
        begin
                $readmemb("instructions.txt", rom);
        end
```

```
15
16          always @ (posedge clk)
17          begin
18                  q <= rom[addr];
19          end
20
21  endmodule
```

Código 15 – Template de Memória ROM de instruções

APÊNDICE C – Conjunto de módulos para a apresentação de dados

Conversão e atribuição do display de sinal para números negativos.

```
module Negativo(NUM,BIN,HEX);

        input [31:0] NUM;
        output reg [31:0] BIN;
        output reg [6:0] HEX;
        integer i;

        always@(*)begin
                BIN = NUM;
                if(BIN[31]==1)begin
                        for(i=0;i<31;i=i+1)begin
                                if(BIN[i]==1)
                                        BIN[i] = 0;
                                else
                                        BIN[i] = 1;
                        end
                        BIN = BIN + 1;
                        HEX = 7'b1111110;
                end
                else
                        HEX = 7'b1111111;
        end

endmodule
```

Código 16 – Número Negativo

Módulo com funcionalidade de dividir as casa decimais.

```
module DivisaoCasas(NUM, C, D, U);

        input [9:0] NUM;
        output reg [3:0] C, D, U;
        integer i;

        always @(NUM) begin
                C = 4'b0000;
                D = 4'b0000;
                U = 4'b0000;

                for(i = 9; i >= 0; i = i - 1) begin
                        if(C >= 5)
                                C = C + 3;
                        if(D >= 5)
                                D = D + 3;
                        if(U >= 5)
                                U = U + 3;

                        C = C << 1;
```

```
21                          C[0] = D[3];
22                          D = D << 1;
23                          D[0] = U[3];
24                          U = U << 1;
25                          U[0] = NUM[i];
26                  end
27          end
28
29  endmodule
```

Código 17 – Casas decimais

Decodificador de dígito decimal para o *display* de 7 segmentos.

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

Código 18 – Decodificador BCD

APÊNDICE D – Unidade de Processamento

Descrição da Unidade de Processamento.

```
1  module UnidadeProcessamento(Reset, CLK, HLT, RegDst, RegDst2, RegWrite, JLink, RDiv,
       WriteSrc, FPGA_Switches, ExtentType,
2                                                                    ALUsrc, ALUop,
                                                                         Branch, Jump,
                                                                          JumpR,
                                                                         MemWrite,
                                                                         MemtoReg,
                                                                         OutWrite,
                                                                         OpCode,
                                                                         Display4,
                                                                         Display3,
                                                                         Display2,
                                                                         Display1);
3
4          input [9:0] FPGA_Switches;
5          input Reset, CLK, HLT, RegDst, RegDst2, RegWrite, JLink, RDiv, WriteSrc,
               ExtentType, ALUsrc, Branch, Jump, JumpR, MemWrite, MemtoReg, OutWrite;
6          input [3:0] ALUop;
7          output [4:0] OpCode;
8          output [6:0] Display4, Display3, Display2, Display1;
9
10         // DECLARACAO DE PORTAS PARA A INSTANCIACAO
11
12                 // PROGRAM COUNTER
13                 wire [31:0] PC_Entrada, PC_Saida, ProxEndereco;
14
15                 // MEM DE INSTRUCAO
16                 wire [31:0] IMem_Instrucao;
17
18                 // EXTENSOR DE JUMP
19                 wire [31:0] Saida_ExtensorJump;
20
21                 // MUX LEITURA1
22                 wire [4:0] RegLeit1;
23
24                 // MUX LEITURA2
25                 wire [4:0] RegLeit2;
26
27                 // BANCO DE REGISTRADORES
28                 wire [31:0] DadoEscrita, DadoLeitReg1, DadoLeitReg2;
29
30                 // MEMORIA DE DADOS
31                 wire [31:0] DadoLeitMem;
32
33                 // MUX ULA
34                 wire [31:0] D2ULA;
35
36                 // ULA
37                 wire [31:0] ResultadoULA, RestoDiv;
38                 wire Zero;
39
40                 // MUX MEMTOREG
```

```
41                wire [31:0] Saida_MuxToReg;
42
43                // EXTENSOR IM/ENDERECO
44                wire [31:0] Saida_Extensor;
45
46                // MUX BRANCH
47                wire ControleBranch = Branch & Zero;
48                wire [31:0] Saida_MuxBranch;
49
50                // MUX JUMP E JLINK
51                wire JSelecao = Jump | JLink;
52                wire [31:0] Saida_MuxJump;
53
54                // MUX JUMPR
55                wire [31:0] Saida_MuxJumpR;
56
57        // INSTANCIACOES DOS MODULOS
58
59                // PROGRAM COUNTER
60                PC PC(.Prox(PC_Entrada), .Atual(PC_Saida), .Reset(Reset), .CLK(CLK));
61
62                // MEM DE INSTRUCAO
63                IMem IMem(.Endereco(PC_Saida), .Instrucao(IMem_Instrucao), .CLK(CLK));
64
65                // EXTENSOR DE JUMP
66                ExtensorJump ExtensorJump(.Entrada(IMem_Instrucao[26:0]), .Saida(
                      Saida_ExtensorJump));
67
68                // SAIDA OPCODE PARA A UC
69                assign OpCode = IMem_Instrucao[31:27];
70
71                // SOMA ENDERECO PC, DEVIDO INSTRUCAO HLT
72                SomaPC SomaPC(.Endereco(PC_Saida), .HLT(HLT), .Saida(ProxEndereco));
73
74                // MUX LEITURA1
75                MuxReg Mux_Leitura1(.Entrada0(IMem_Instrucao[21:17]), .Entrada1(
                      IMem_Instrucao[26:22]), .Selecao(RegDst), .Saida(RegLeit1));
76
77                // MUX LEITURA2
78                MuxReg Mux_Leitura2(.Entrada0(IMem_Instrucao[16:12]), .Entrada1(
                      IMem_Instrucao[26:22]), .Selecao(RegDst2), .Saida(RegLeit2));
79
80                // BANCO DE REGISTRADORES
81                BancoDeRegs BancoDeRegs(.RegLeit1(RegLeit1), .RegLeit2(RegLeit2), .
                      RegEscrita(IMem_Instrucao[26:22]), .DadoEscrita(DadoEscrita), .R30(
                      RestoDiv),
82                .R31(ProxEndereco), .RegWrite(RegWrite), .RDiv(RDiv), .JLink(JLink), .CLK
                      (CLK), .Reset(Reset), .DadoLeit1(DadoLeitReg1), .DadoLeit2(
                      DadoLeitReg2));
83
84                // MEMORIA DE DADOS
85                DMem DMem(.Endereco(Saida_Extensor), .DadoEscrita(DadoLeitReg2), .
                      MemWrite(MemWrite), .DadoLeit(DadoLeitMem), .CLK(CLK));
86
87                // MUX ESCRITA REG
88                Mux Mux_EscritaReg(.Entrada0(Saida_MuxToReg), .Entrada1(FPGA_Switches), .
                      Selecao(WriteSrc), .Saida(DadoEscrita));
89
90                // MUX ULA
```

```
91                  Mux Mux_ULA(.Entrada0(DadoLeitReg2), .Entrada1(Saida_Extensor), .Selecao(
                        ALUsrc), .Saida(D2ULA));
92
93                  // ULA
94                  ULA ULA(.ALUop(ALUop), .D1(DadoLeitReg1), .D2(D2ULA), .RESULTADO(
                        ResultadoULA), .ZERO(Zero), .RESTOdiv(RestoDiv));
95
96                  // MUX MEMTOREG
97                  Mux Mux_ToReg(.Entrada0(DadoLeitMem), .Entrada1(ResultadoULA), .Selecao(
                        MemtoReg), .Saida(Saida_MuxToReg));
98
99                  // EXTENSOR IM/ENDERECO
100                 Extensor Extensor(.Entrada0(IMem_Instrucao[16:0]), .Entrada1(
                        IMem_Instrucao[21:0]), .ExtentType(ExtentType), .Saida(Saida_Extensor
                        ));
101
102                 // MUX BRANCH
103                 Mux Mux_Branch(.Entrada0(ProxEndereco), .Entrada1(Saida_Extensor), .
                        Selecao(ControleBranch), .Saida(Saida_MuxBranch));
104
105                 // MUX JUMP E JLINK
106                 Mux Mux_Jump(.Entrada0(Saida_ExtensorJump), .Entrada1(Saida_MuxBranch), .
                        Selecao(JSelecao), .Saida(Saida_MuxJump));
107
108                 // MUX JUMPR
109                 Mux Mux_JumpR(.Entrada0(Saida_MuxJump), .Entrada1(DadoLeitReg1), .Selecao
                        (JumpR), .Saida(Saida_MuxJumpR));
110                 assign PC_Entrada = Saida_MuxJumpR;
111
112                 // SAIDA DE DADOSS
113                 SaidaDados SaidaDados(.Dado(DadoLeitReg2), .OutWrite(OutWrite), .CLK(CLK)
                        , .Display1(Display1), .Display2(Display2), .Display3(Display3),
114                 .Display4(Display4));
115
116 endmodule
```

Código 19 – Unidade de Processamento