RA: 133536

Processador 32 bits desenvolvido em HDL Verilog, baseado na arquitetura MIPS - RISC

São José dos Campos - Brasil

Agosto de 2020

RA: 133536

Processador 32 bits desenvolvido em HDL Verilog, baseado na arquitetura MIPS - RISC

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

Aluno: Álvaro Cardoso Vicente de Souza

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

Agosto de 2020

Resumo

O projeto proposto no laboratório de arquitetura e organização de computadores consiste na implementação de um processador baseado na arquitetura MIPS (*Microprocessor Without Interlocked Pipeline Stages*) do tipo RISC (*Reduced Instruction Set Computer*), desenvolvido em HDL Verilog. O processador deve ser implememententado para executar diversas instruções com o objetivo de rodar os algoritmos propostos pelo professor, como por exemplo, a sequência de *Fibonacci*. Para que isso seja possível, é necesssário a implementação de um conjunto de instruções, com seus respectivos modos de endereçamento, capazes de realizar a tarefa demandada. Para isso, utilizando o *Software Intel Quartus Prime*, implementou-se, na linguagem HDL Verilog, e gerou-se as *Waveforms* de todos os módulos que compõem o *datapath* que os dados devem percorrer contendo uma unidade lógica e aritmética, memórias de instrução e de dados, unidade de controle, extensores de bits e um banco de 32 registradores. Além disso, deseja-se implementar um Módulo de entrada e saída responsável pela comunicação com a plataforma física FPGA, *Field Programmable Gate Array - Intel/Altera* modelo DE2-115.

Palavras-chaves: Processador, 32 Bits, MIPS, RISC, FPGA.

Lista de ilustrações

Lista de tabelas

Lista de códigos

Sumário

1 Introdução

A civilização humana já sofreu inúmeras transformações ao longo dos anos, atualmente, habitamos uma sociedade marcada pela presença de novas tecnologias e ferramentas tecnológicas que promovem avanços, nunca antes imagináveis,nas mais diversas áreas de atuação e, por consequência, estão cada vez mais presentes no nosso cotidiano. Podemos atribuir a esses componentes tecnológicos algumas características em comum como, por exemplo, a presença dos sistemas digitais, responsáveis por todo o controle dos sinais elétricos que garantem o funcionamento correto de mais de uma miríade de dispositivos eletrônicos.

Seguindo as definições de TOCCI, 2011 (1), podemos considerar os sistemas digitais como combinações de dispositivos integrados que são desenvolvidos com o intuito de manipular informações lógicas ou quantidades físicas representadas em um formato digital podendo assumir apenas valores discretos e, na maioria dos casos, são sistemas mais fáceis de serem projetados.

Com esse pensamento em mente, foi possível associar o funcionamento desses sistemas digitais a uma linguagem de descrição de *hardware* denominada Verilog. Possibilitando a implementação de projetos digitais, analógicos e ainda possibilitam a combinação de ambos. Assim nos permite abstrair alguns dos conceitos de portas lógicas e álgebra booleana, relacionando-se com as boas práticas e a lógica de programação.

Como já mencionado, atualmente existem uma infinidade de sistemas digitais empregados nas mais diversas aplicações desde simples calculadoras a satélites que orbitam o globo terrestre porém, neste projeto vamos focar especificamente na aplicação da linguagem Verilog, para a descrição de um Processador de 32 Bits baseado na arquitetura MIPS - RISC.

2 Objetivos

2.1 Geral

Desenvolver e implementar um processador de 32 Bits baseado na arquitetura MIPS - RISC em HDL Verilog, através da implementação das Unidades de Processamento, Controle e, se possível, o Módulo de Entrada e Saída para comunicação com o FPGA.

2.2 Específicos

- Projetar e descrever a Unidade Lógica e Aritmética em Verilog;
- Projetar e descrever a Memória de Dados em Verilog;
- Projetar e descrever a Memória de Instruções em Verilog;
- Projetar e descrever o Banco de Registradores em Verilog;
- Projetar e descrever o *Program Counter* em Verilog;
- Projetar e descrever o extensor de bits em Verilog;
- Realizar o *link* entre os módulos que compõem a unidade de processamento;
- Obter e analisar os resultados e suas respectivas *Waveforms* geradas pelo *software Intel Quartus Prime*.

3 Fundamentação Teórica

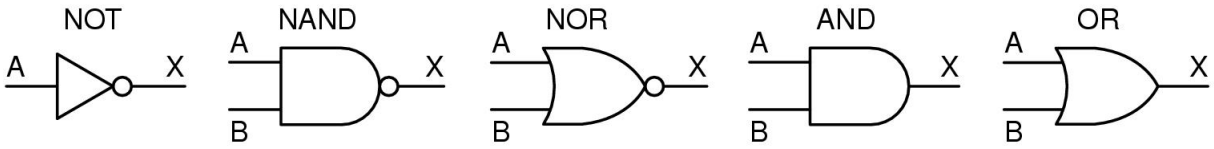
3.1 Álgebra Booleana, Portas Lógicas e Circuitos Digitais

Para que se tenha um bom entendimento do projeto como um todo, precisa -se de uma base teórica sólida e para alcançarmos esse objetivo alguns conceitos e definições se fazem necessários, como é o caso da álgebra booleana, das portas lógicas e dos circuitos digitais. Podemos considerar a Álgebra Booleana como a base matemática e lógica por trás das portas lógicas e dos circuitos digitais em si.

A Álgebra Booleana foi introduzida por George Boole no século XIX e com o passar do tempo ganhou popularidade e se tornou extremamente vital o estudo, desenvolvimento e representação de circuitos digitais.

Ela trabalha principalmente com a implementação de Tabelas Verdade que descrevem Funções Booleanas que possuem os valores de entrada e saída representados de forma binária (0-1) nas colunas da tabela. Desta maneira, podemos associar algumas Funções Booleanas e suas respectivas tabelas verdades a portas lógicas, que são representações gráficas dessas funções mais comuns utilizadas na formulação de circuitos digitais, mas também são empregadas na construção de circuitos reais onde essas portas são formadas por diversos transistores e resistores.

Figura 1 – Portas lógicas e suas tabelas verdades A,B e X representam as duas entradas e uma saída respectivamente.



A	X
0	1
1	0

(a)

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

(b)

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

(c)

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

(d)

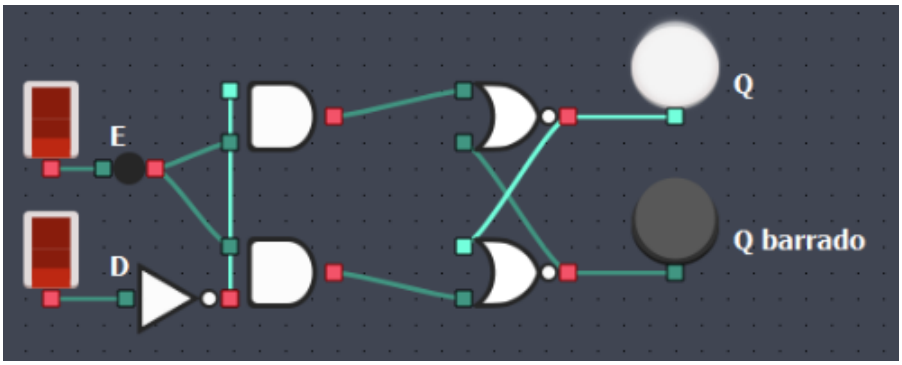
A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

(e)

Fonte: http://www.dpi.inpe.br/ carlos/Academicos/Cursos/ArqComp/3-2.jpg.

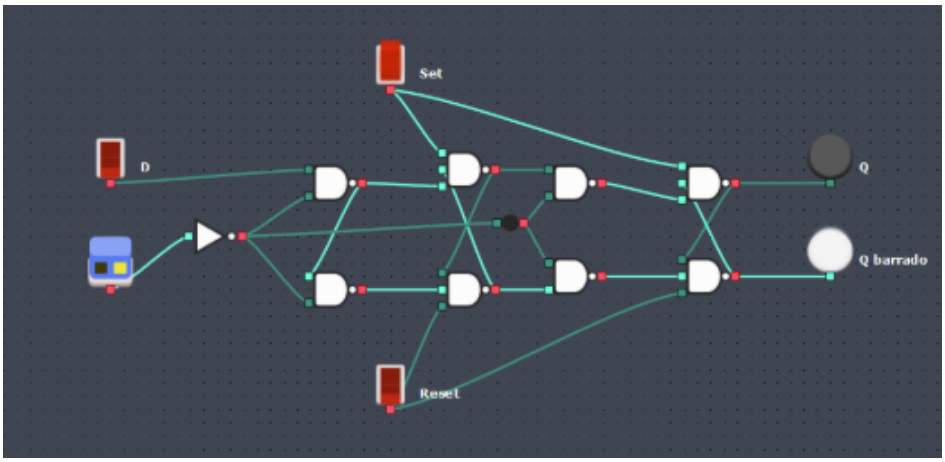
Ao falarmos dos circuitos digitais, podemos dividí-los em duas categorias principais: os circuitos sequênciais e os combinacionais. Os circuitos combinacionais são aqueles onde as saídas dependem somente dos valores de entrada e os circuitos sequênciais, por sua vez, contam com a presença de elementos de memórias chamados de *Flip-Flops* ou, em alguns casos podem conter elementos mais rudimentares denominados *Latches*.

Figura 2 – *R*epresentação de um *D-Latch* utilizando a plataforma *wiRed Panda*.



Fonte: Autor

Figura 3 – Representação de um *Flip-Flop D* com *set-reset* utilizando a plataforma *wiRed Panda*.

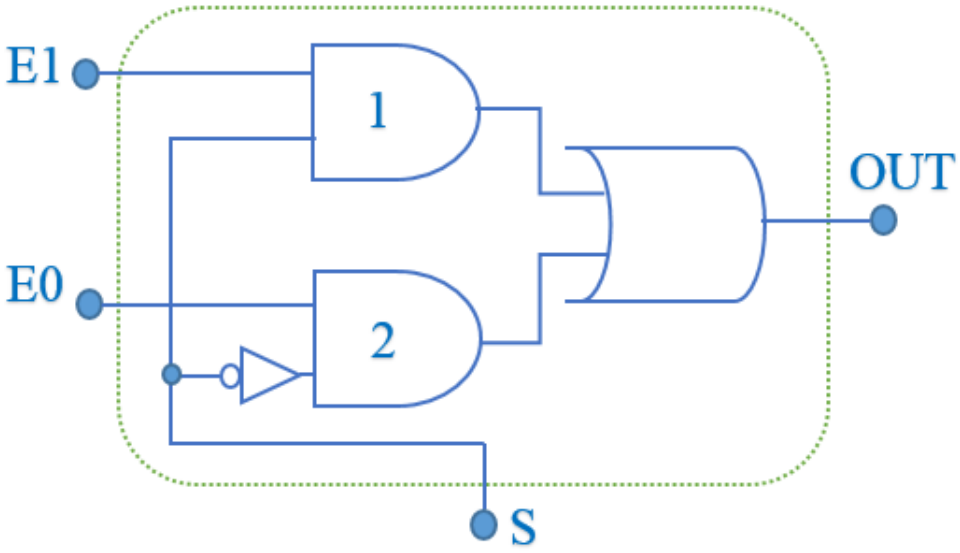


Fonte: Autor

3.2 Multiplexadores

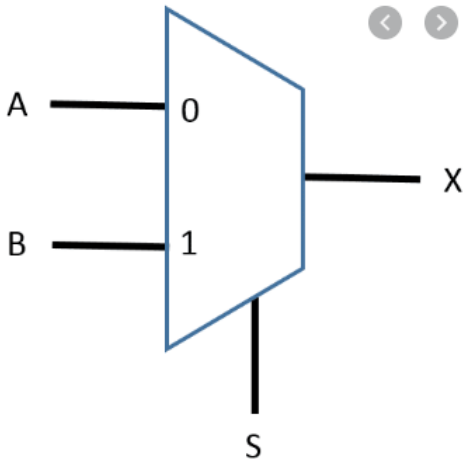
Esse circuito, também chamado de circuito seletor, é o circuito responsável pela seleção de um sinal, dentre múltiplos sinais de entrada, para ser propagado na saída em um tempo específico. Ele recebe os sinais de entrada e através de ums sinal de seleção determina a saída propagada.

Figura 4 – Multiplexador de duas entradas



Fonte: https://eletronworld.com.br/eletronica/multiplexadores/

Figura 5 – Representação do Multiplexador



Fonte: https://www.embarcados.com.br/mux/

3.3 Arquiterura e Organização de Computadores

Podemos dividir o processo de desenvolvimento de um processador em duas partes principais, a arquitetura e a organização. Ao falarmos de arquitetura, nos referimos a parte vísual voltada aos atributos utilizados diretamente pelo programador, normalmente ralaciona-se ao *software* e ao conjunto de instruções que serão utilizadas durante o processamento de dados, normalmente trabalhadas em linguagens de baixo nível, como o Assembly, por exemplo. Por outro lado, ao nos referirmos a organização, estamos falando

da maneira que o sistema em si é estruturado, levando em conta o *hardware* utilizado e como a Unidade de Processamento e a unidade de Controle se interligam descrevendo a maneira que as instruções são realizadas e os dados processados.

3.4 Números Binários, Decimais e representação BCD

Durante nosso cotidiano, costumamos utilizar os números decimais, que utilizam os numerais de 0-9, para fazer todas as tarefas, desde operações mais simples como a adição e subtração até as mais complexas como integrais e derivadas. Porém ao lidarmos com circuitos digitais, trabalhamos com um sistema diferente onde existem apenas com duas possibilidades 0 ou 1. Denominamos a representação que consiste apenas nessas duas possibilidades de representação binária. Como podemos imaginar essa diferença de representação pode causar inúmeros problemas já que seria o mesmo que tentar realizar uma conversa onde uma das partes só fala alemão e a outra árabe. Para contornarmos essas diferenças devemos realizar as conversões utilizando o sistema BCD *(binary-coded decimal)* onde cada dígito de 0-9 tem seu correspondente binário.

Tabela 1 – Algarismos decimais e seus respectivos códigos BCD

Algarismo	Codificação
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Fonte: https://mathwithtom.files.wordpress.com/2018/10/cc3b3digo-bcd.jpg?w=656

3.5 Placa FPGA *Field Programmable Gate Array*

De acordo com FERDJALLAH, 2011 (2) a rápido avanço na tecnologia VLSI - *Very Large Scale Integration* (processo de criação de circuitos integrados combinando vários transistores em um só *chip*) criou *chips* especiais que podem ser programados pelo usuário final para funcionar como diferentes circuitos lógicos.

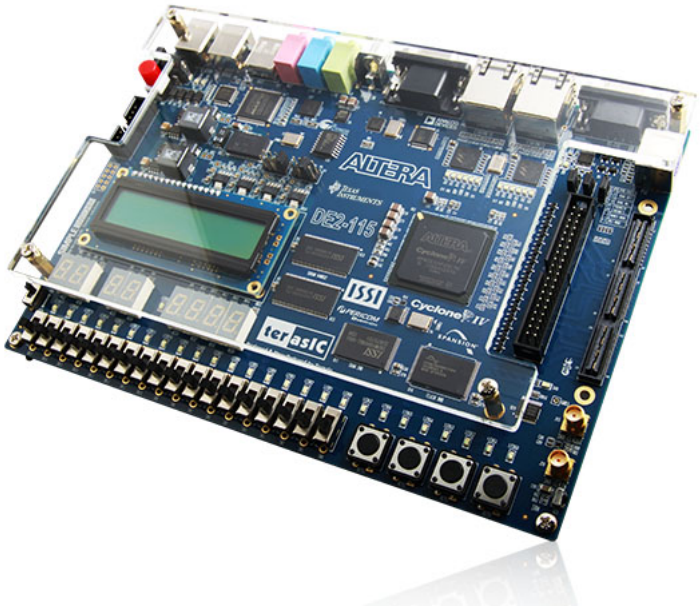
Ainda segundo FERDJALLAH, 2011 (2), esses *chips* denominados PDL's - *programmable logic devices* - possuem diversas funções customizáveis, desde os switches até LED's coloridos. Sendo um dos principais od denominados FPGA's. Segundo a própria fabricante Intel, o FPGA é um circuito integrado, semicondutor, onde a maior parte da

funcionalidade elétrica interna do dispositivo pode ser customizada criando circuitos que podem funcionar de diversas maneiras como, por exemplo, um processador, uma GPU (unidade gráfica), entre outras.

A placa pode ser "programada"(descrita) de diversas maneiras as mais comuns são utilizando as linguagens denominadas *Hardware Descriptive Languages* (HDL's) como é o caso da Verilog, porém implementações com outras linguagens (exemplo: MATLAB) também são possíveis.

Essa maleabilidade, a facilidade de "programação"(descrição do *Hardware*) e o fato dela possuir diversos da placa a tornam extremamente atrativa para os profissionais da meio tecnológico que as utilizam nos mais diversos dispositivos.

Figura 6 – Placa FPGA Intel/Altera DE2 - 115



Fonte: https://www.macnicadhw.com.br/produtos/kits/altera-de2115

3.6 Memória ROM - *Read Only Memory*

A memória ROM ((*Read Only Memory*), é uma memória não-volátil, ou seja, ela mantém os dados salvos sem uma fonte de energia. Como o próprio nome sugere, as memórias ROM's , são memórias de somente leitura e não permitem escrita de dados após serem iniciadas. O grande atrativo das memória ROM's se encontra no fato dos dados estarem de forma permanete na memória principal e não necessitam serem carregados de uma memória secundária (STALLINGS, 2010)(3).

3.7 Memória RAM - *Random Access Memory*

A memória RAM (*Random Access Memory*), é uma memória volátil, ou seja, ela mantém os dados salvos somente enquanto existe um suprimento constante de energia, se o suprimento de energia é interrompido os dados são apagados. O grande atrativo das memória RAM's se encontra no fato dela poder ler e escrever dados de maneira relativamente fácil e veloz. Ambas as operações de leitura e escrita são realizadas através de sinais elétricos (STALLINGS, 2010)(3).

3.8 RISC - *Reduced Instruction Set Computer*

Como o próprio nome já diz, a arquitetura RISC conta com um conjunto instruções reduzidas que não precisam de um interpretador. A grande vantagem da arquitetura RISC é o fato das instruções serem de execução rápida porém, em contra partida, são instruções simples e acabam sendo necessárias diversas instruções para realiazar uma tarefa simples, já que cada instrução descreve um comportamento específico básico, tarefas essas que,em outras arquiteturas, como a CISC (*Complex Instruction Set Computer*) poderiam ser executadas em apenas uma única instrução.

3.9 MIPS - *Microprocessor Without Interlocked Pipeline Stages*

A arquitetura MIPS, desenvolvida em 1984 por graduandos da Universidade de Stanford, trata-se de uma arquitetura sem a presença de *pipelines*(segmentção de intruções) baseada em um conjunto de intruções reduzidas(RISC). Essa arquitetura é usualmente implementada com cada instrução realizada a cada ciclo de clock, conhecida como monociclo.

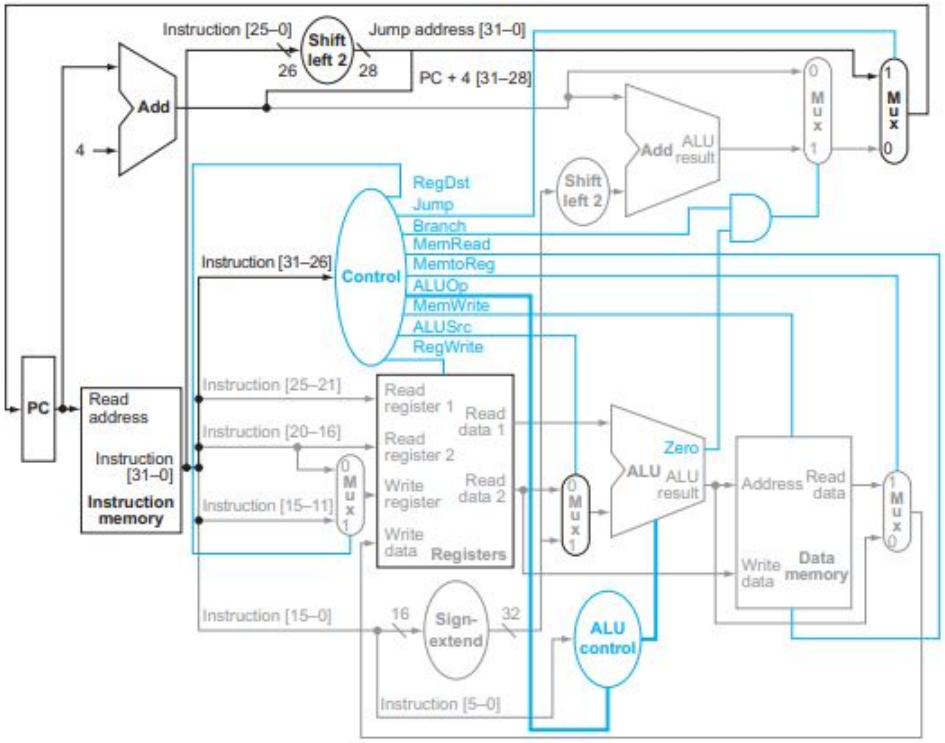
3.9.1 Principais Elemetos

Segundo PATTERSON, 2014 (4), os principais elementos do MIPS presentes no *datapath* são:

- **PC (*Program Counter*)** - Registarador que guarda o endereço da instrução atual;
- **PC *Adder*** - Somador que incrementa o PC para o endereço da próxima instrução;
- **Memória de Instrução** - Memórias onde são guardadas as instruções a serem realizadas durante o processamento;
- **Banco de Registradores** - Armazena os dados obtidos na memória que serão utilizados nas operações das intruções especificadas para o processamento;

- **Memória de Dados** - Memória onde os dados que serão utilizados durante o processamento ficam guardados;
- **Unidade Lógica e Artimética(ULA)** - Segundo PATTERSON, 2014 (4), a ULA é o músculo(*brawn*) do computador responsável pelas operações artitméticas, como a adição/subtração e as operações lógicas, como AND, OR, NOT, entre outras;
- **Unidade de Controle** - Responsável por receber a instrução e, a partir dela, determinar qual caminho de processamento os dados devem seguir, ou seja, codifica a instrução e gera os sinais de controle para os elementos supracitados e os multiplexadores de seleção.
- **Extensores de Bits** - Utilizados para extender a instrução para ocupar os 32 Bits.

Figura 7 – *datapath* MIPS



Fonte:PATTERSON, 2014 (4)

3.9.2 Sinais de Controle

Como mencionado no sub-capítulo acima, a unidade de controle é a responsável por gerar os sinais de controles que determinam o caminho dos dados de cada instrução. No MIPS existem os seguintes sinais de controle:

- **RegDst** - Determina o registrador destino;
- **Jump** - Habilita endereço de salto no PC;
- **Branch** - Testa com uma condição booleana para habilitar e carregar o salto para o endereço especificado do PC;
- **MemRead** - Habilita a leitura da memória;
- **MentoReg** - Determina de onde o valor de escrita vem;
- **AluOP** - Especifica a operação que a ULA deve realizar;
- **MemWrite** - Habilita a escrita na memória;
- **AluSrc** - Seleciona o segundo operando;
- **RegWrite** - Habilita a escrita em um registrador;

3.9.3 Conjunto de Instruções

Como mencionado anteriormente, para o funcionamento do processador, necessita-se de um conjunto de instruções que determinam o que o processador e capaz de realizar com os dados. Essas instruções podem variar em diferentes tipos, cada um com seu formato específico. No MIPS temos o seguinte Conjunto de Instruções:

Tabela 2 – Conjunto de Instruções MIPS

Category	Instruction	Example
Arithmetic	add	add $s1,$s2,$s3
	subtract	sub $s1,$s2,$s3
	add immediate	addi $s1,$s2,20
Data transfer	load word	lw $s1,20($s2)
	store word	sw $s1,20($s2)
	load half	lh $s1,20($s2)
	load half unsigned	lhu $s1,20($s2)
	store half	sh $s1,20($s2)
	load byte	lb $s1,20($s2)
	load byte unsigned	lbu $s1,20($s2)
	store byte	sb $s1,20($s2)
	load linked word	ll $s1,20($s2)
	store condition. word	sc $s1,20($s2)
	load upper immed.	lui $s1,20
Logical	and	and $s1,$s2,$s3
	or	or $s1,$s2,$s3
	nor	nor $s1,$s2,$s3
	and immediate	andi $s1,$s2,20
	or immediate	ori $s1,$s2,20
	shift left logical	sll $s1,$s2,10
	shift right logical	srl $s1,$s2,10
Conditional branch	branch on equal	beq $s1,$s2,25
	branch on not equal	bne $s1,$s2,25
	set on less than	slt $s1,$s2,$s3
	set on less than unsigned	sltu $s1,$s2,$s3
	set less than immediate	slti $s1,$s2,20
	set less than immediate unsigned	sltiu $s1,$s2,20
Unconditional jump	jump	j 2500
	jump register	jr $ra
	jump and link	jal 2500

Fonte:PATTERSON, 2014 (4)(Adaptado)

Tabela 3 – Formato de Instruções MIPS

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

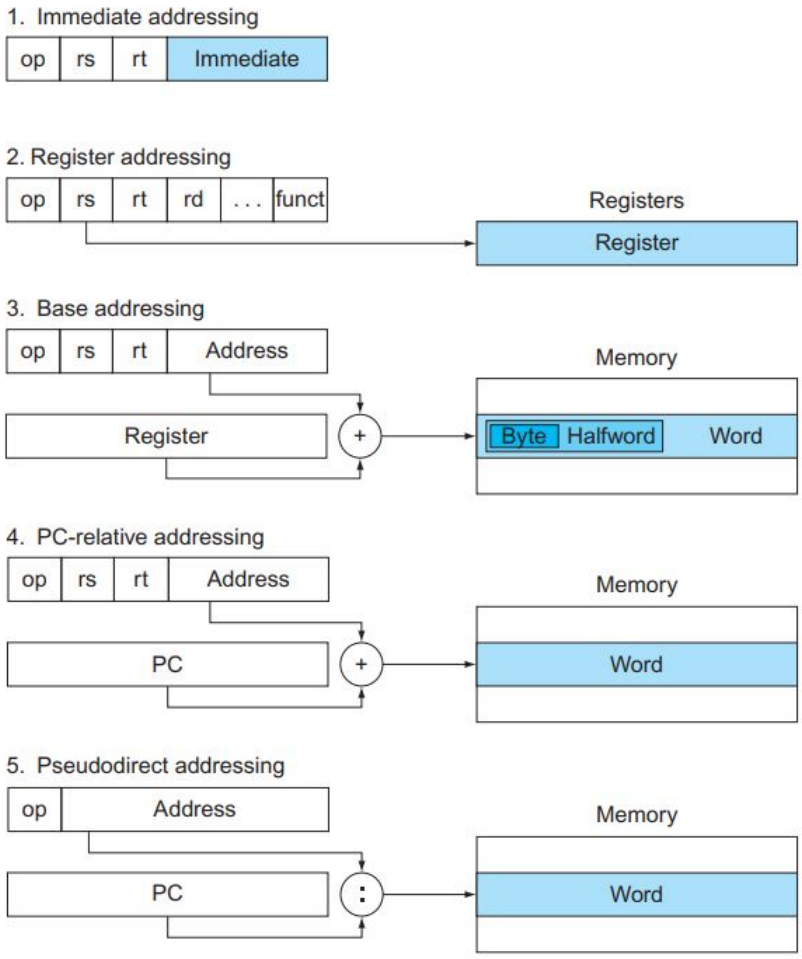
Fonte:PATTERSON, 2014 (4)

3.9.4 Modos de Endereçamento

Segundo PATTERSON, 2014 (4) o MIPS conta com cinco modos de endereçamento:

- **Imediato/Direto** - Operando é uma constante dentro da própria instrução;
- **Registrador/Indireto** - Operando é um registrador;
- **Base - Deslocamento** - Operando está no endereço de memória composto pela soma do registrador e uma constante presente na instrução;
- **Relativo ao PC** - Onde o endereço do *Branch* é a soma do PC e uma constante presente na instrução;
- **Pseudo-direto** - Onde o endereço do *Jump* é composto pelos 26 bits da instrução concatenados com os bits superiores do PC.

Figura 8 – Modos de Endereçamento MIPS



Fonte:PATTERSON, 2014 (4)

4 Desenvolvimento

A arquitetura do processador desenvolvido baseia-se na implementação básica do MIPS realizada por PATTERSON, 2014 (4) e na arquitetura RISC, com as instruções sendo realizadas em um ciclo de *clock* e sem a presença de *pipelines*. Para isso, criou-se um conjunto de instruções baseados nas instruções do MIPS, com seus respectivos formatos e modos de endereçamento. Além disso, descreveu-se toda a implementação dos principais elementos do MIPS, citados anteriormente no sub-capítulo 3.9.1 em HDL Verilog, utilizando a plataforma *Intel Quartus Prime*.

4.1 Conjunto de Instruções

Para a implementação do processador se faz necessário um conjunto de instruções capaz de realizar os algoritmos propostos pelo professor. Tendo isso em mente, foram utilizadas as seguintes instruções:

Tabela 4 – Conjunto de Instruções

Conjunto de Instruções		
TIPO	OpCode-B	INSTRUÇÃO
I	00001	ADDI - Adição com Imediato
I	00010	SUBI - Subtração com Imediato
I	00011	NOT - Função Lógica NOT
I	00100	BEQ - Desvio se Igual
I	00101	BNQ - Desvio se não Igual
I	00110	SL - Deslocamento a Esquerda
I	00111	SR - Deslocamento a Direita
I	01000	LD - Load
I	01001	ST - Store
R	01010	ADD - Adição
R	01011	SUB - Subtração
R	01100	MUL - Multiplicação
R	01101	DIV - Divisão
R	01110	MOD - Resto da Divisãoi
R	01111	AND - Função Lógica AND
R	10000	OR - Função Lógica OR
R	10001	XOR - Função Lógica XOR
R	10010	JR - Salto para Registrador
R	10011	SLT - Set Less Than
R	10100	SGT - Set Greater Than
J	10101	JMP - Salto Incondicional
J	10110	JAL - Salto e Link
I	10111	IN - Input
I	11000	OUT - Output
X	11111	HLT - Halt (Parada)
X	00000	NOP - Instrução Vazia

Fonte:Autor

Vale ressaltar que as instruções **IN** e **OUT** são as instruções que realizam a comunicação com o módulo de entrada e saída para que seja possível a implementação no FPGA mais a frente para os próximos laboratórios.

4.2 Formato das Instruções

Como podemos perceber na Tabela - 4 temos quatro tipos de instruções onde cada uma tem um formato diferente, como demonstrado abaixo:

Tabela 5 – Formato das Instruções

Formato das Instruções						
	OpCode-A 32-31	OpCode - B 30-26	RD 25-20	RS 19-14	IMEDIATO 13-1	
Tipo - I	00	#####	######	######	##############	
	OpCode-A 32-31	OpCode - B 30-26	RD 25-20	RS 19-14	RT 13 - 8	Blank 7-1
Tipo - R	01	#####	######	######	######	00000000
	OpCode-A 32-31	OpCode - B 30-26	RD 25-20	IMEDIATO 19-1		
Tipo - J	10	#####	######	###################		
	OpCode-A 32-31	OpCode - B 30-26	IMEDIATO 25-1			
Tipo - X	11	#####	########################			

Fonte:Autor

Como pode-se observar as instruções possuem campos diferentes de acordo com cada tipo, onde podemos observar:

- **Tipo I** - OpCode A codifica o tipo da instrução I, OpCode B codifica a instrução em si, RD é a posição no banco de registradores do registrador destino, RS é a posição no banco de registradores do registrador com o operando, Imediato é o operando em si;
- **Tipo R** - OpCode A codifica o tipo da instrução R, OpCode B codifica a instrução em si, RD é a posição no banco de registradores do registrador destino, RS é a posição no banco de registradores do registrador com o primeiro operando, RT é a posição no banco de registradores do registrador com o segundo operando, *Blank* responsável por completar os 32 bits;
- **Tipo J** - OpCode A codifica o tipo da instrução J, OpCode B codifica a instrução em si, RD é a posição no banco de registradores do registrador destino(nesse caso guarda a posição para o *Jump and Link*), Imediato é o valor utilizado para o *Jump*;
- **Tipo X** - OpCode A codifica o tipo da instrução X, OpCode B codifica a instrução em si, Imediato apenas completa os 32 bits da instrução, pois apenas precisamos saber se é a instrução **HLT** ou **NOP**.

4.3 Modos de endereçamento

Como mencionada na sub-seção 3.9.4, o MIPS conta com cinco modos de endereçamento prinpais. Para esse implementação, serão utilizados os seguintes modos de instrução, descritos na Figura 8:

- **Imediato/Direto** - Operando é uma constante dentro da própria instrução(ADDI, SUBI, NOT, BEQ, BNQ, SL, SR, JUMP, JAL, IN, OUT) ou o campo imediato da instrução se refere diretamente ao endereço na memória (LD, ST);
- **Registrador/Indireto** - Operando é um registrador (ADD, SUB, MUL, DIV, MOD, AND, OR, XOR, JR, SLT, SGT).

4.4 Sinais de Controle

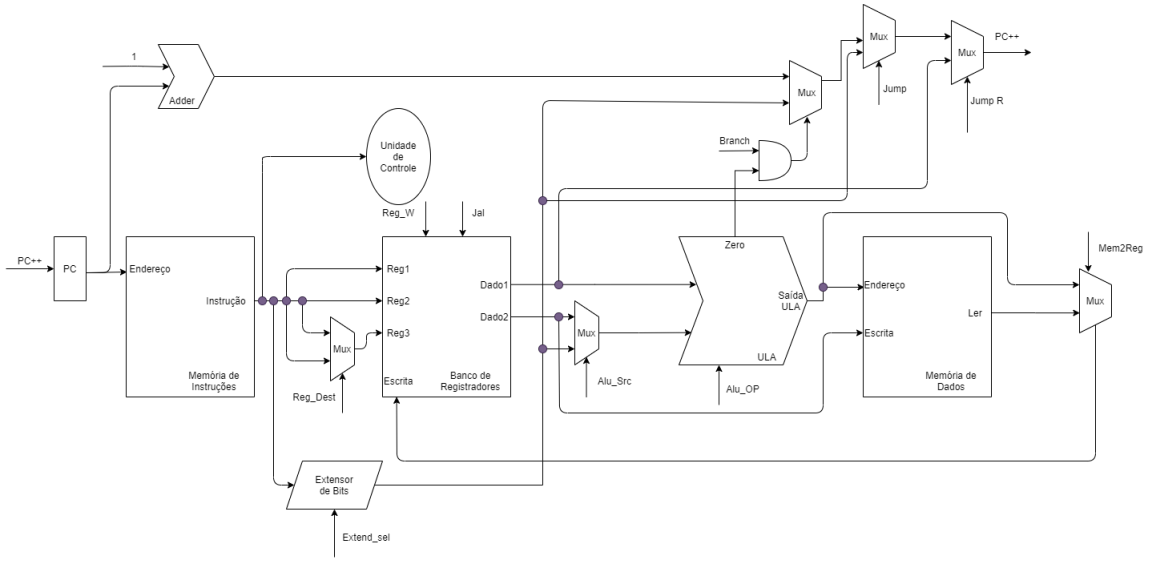
Os sinais de controle utilizados na implementação serão semelhantes aos sinais do MIPS, com a adição de um sinal de *HALT* e dois sinais para extensão de bits.

- **RegDst** - Determina o registrador destino;
- **AluSrc** - - Seleciona o segundo operando;
- **Jump** - Habilita endereço de salto no PC;
- **JumpR** - Habilita endereço de salto no Jump Reg;
- **Jal** - Habilita endereço de salto no PC e guarda o PC + 1 antes do salto em um registrador dedicado;
- **Branch** - Testa com uma condição booleana para habilitar e carregar o salto para o endereço especificado do PC;
- **Men2Reg** - Determina de onde o valor de escrita vem;
- **AluOP** - Especifica a operação que a ULA deve realizar;
- **MemW** - Habilita a escrita na memória;
- **RegW** - Habilita a escrita em um registrador;
- **Halt** - Sinaliza a parada;
- **Extend** - Sinaliza que será necessária uma extensão de bits na instrução.
- **Extend-sel** - Determina o tamanho da extensão a ser realizada em determinada instrução.

4.5 *Datapath*

Implementação simplificada do *datapath* (PC2), sem o módulo de entrada e saída:

Figura 9 – *datapath* desenvolvido

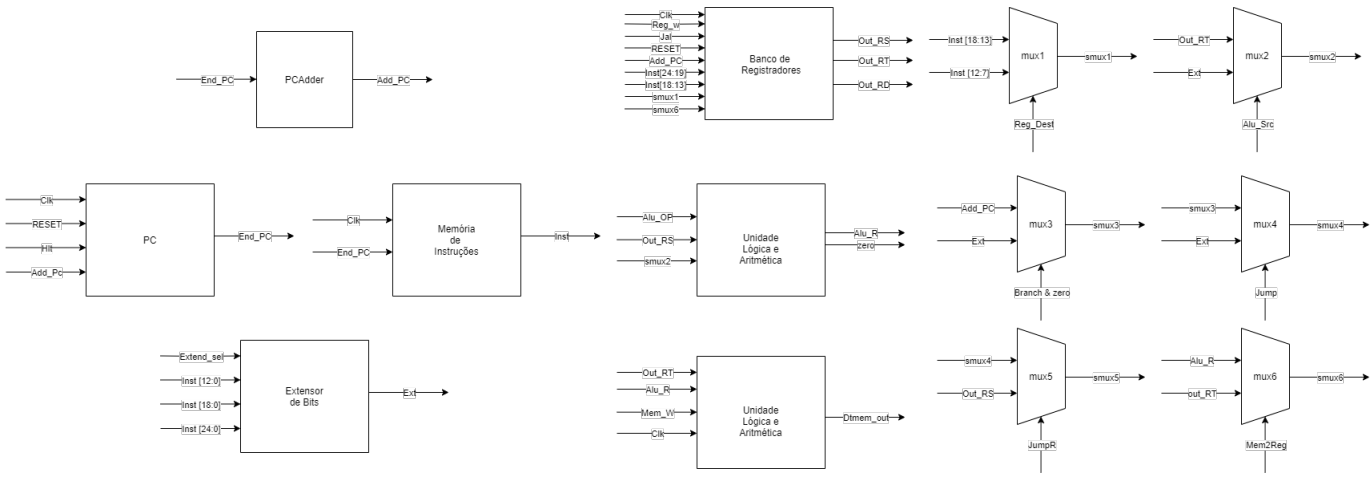


Fonte:Autor

4.6 Unidade de Processamento

A unidade de processamento consiste basicamente no "*link*"dos módulos que compõem a CPU(*central processing unity*) seguindo o esquemático descrito pela Figura 10. Podemos observa que a CPU vai receber os dados através dos 17 *switches* para a entrada de dados e depois tem como saidas os valores para o dysplay de 7 segmentos.

Figura 10 – *link* dos módulos da CPU



Fonte:Autor

A CPU possui a seguinte implementação em Verilog:

```
1  module CPUx (input Clk, input [17:0]Enter_Dt, input RESET, output Dsp1, output Dsp2,
       output Dsp3, output Dsp4, output Dsp5, output Dsp6, output Dsp7);
2
3  wire [31:0] End_PC; //Endereco PC
4  wire [31:0] Add_PC; //PC ++
5  wire [31:0] Inst; //Inst
6  // sinais de controle
7  wire [4:0]Alu_OP;
8  wire Alu_Src;
9  wire JumpR;
10 wire Jal;
11 wire Hlt;
12 wire Jump;
13 wire Reg_W;
14 wire Mem2Reg;
15 wire Mem_W;
16 wire Reg_Dest;
17 wire Branch;
18 wire [1:0]Extend_sel; //sinal de selec para o extensor
19 wire zero; // zero para o branch
20 wire [31:0]Alu_R; // saida ula
21 wire [31:0]Out_RS; // saida rs
22 wire [31:0]Out_RT; // saida rt
23 wire [31:0]Out_RD; // saida rd
24 wire [31:0]Dtmem_out; // saida mem
25 // saida multiplexadores
26 wire [5:0] smux1;
27 wire [31:0] smux2;
28 wire [31:0] smux3;
29 wire [31:0] smux4;
30 wire [31:0] smux5;
31 wire [31:0] smux6;
32 wire [31:0] Ext;// saida extendida
33
34 // Link dos modulos de acordo com o datapath
35 PC PCx (.Clk(Clk), .RESET(RESET), .Hlt(Hlt), .PC_Add(Add_PC), .PC(End_PC));
36 PCAdder PCAdderx (.PC(End_PC), .PC_Add(Add_PC));
37 MemInst MemInstx (.PCin(End_PC), .Clk(Clk), .ins(Inst));
38 RegBk RegBkx (.Clk(Clk), .Reg_W(Reg_W), .Jal(Jal), .RESET(RESET), .PCin(Add_PC), .RS(Inst
       [24:19]), .RT(Inst[18:13]), .RD(smux1), .DataW(smux6), .Out_RS(Out_RS), .Out_RT(
       Out_RT), .Out_RD(Out_RD));
39 BitExt BitExtx (.Extend_sel(Extend_sel), .Case13(Inst[12:0]), .Case19(Inst[18:0]), .
       Case25(Inst[24:0]), .Extended(Ext));
40 ALU ALUx (.Alu_OP(Alu_OP), .D1(Out_RS), .D2(smux2), .R(Alu_R), .z(zero));
41 MemDados MemDadosx (.dado(Out_RT), .mem_end(Alu_R), .Mem_W(Mem_w), .Clk(Clk), .saida(
       Dtmem_out)); // Mem Dados
42 Mux1 Mux1x (.Reg_Dest(Reg_Dest), .RT(Inst[18:13]), .RD(Inst[12:7]), .R_Out(smux1));
43 Mux2 Mux2x (.Alu_Src(Alu_Src), .D1(out_RT), .D2(Ext), .D_Out(smux2));
44 Mux3 Mux3x (.Branch(Branch & zero), .D1_PC(Add_PC), .D2_Branch(Ext), .D_Out(smux3));
45 Mux4 Mux4x (.Jump(Jump), .D1(smux3), .D2(Ext), .D_Out(smux4));
46 Mux5 Mux5x (.Jump_R(JumpR), .D1(smux4), .D2(out_RS), .D_Out(smux5));
47 Mux6 Mux6x (.Mem2Reg(Mem2Reg),.Alu_res(Alu_R), .D_Read(out_RT), .D_Out(smux6));
48
49 endmodule
```

Código 1 – Implementação em Verilog da CPU

4.6.1 *Program Counter*

O PC foi dividido em duas partes, a primeira descreve o PC em si, onde ele é incializado com 32 bits zerados, possui um sinal de controle para o **RESET**, que reinicia o valor do PC para a primeira posição e quando os sinais de controle **HLT e RESET** não estão ativos ele é atualizado com o valor da proxima posição que ele recebe do PCAdder.

O PC possui a seguinte implementação em Verilog:

```
1  module PC (input Clk, input RESET, input Hlt, input [31:0]PC_Add, output reg [31:0]PC);
       //Atualiza o PC para PC++, HALT e RESET
2
3          always @(posedge Clk)
4
5                  begin
6
7                          if (RESET == 1)
8                          begin
9
10                         PC <= 32'b0;
11
12                         end
13
14                         else if (Hlt == 0 && RESET == 0)
15                         begin
16
17                         PC <= PC_Add;
18
19                         end
20                 end
21
22 endmodule
```

Código 2 – Implementação em Verilog do PC

```
1  module PCAdder (input [31:0]PC, output reg [31:0]PC_Add); //PC++
2
3          always @(PC)
4
5                  begin
6                  PC_Add <= PC + 1;
7                  end
8
9  endmodule
```

Código 3 – Implementação em Verilog do PCAdder

4.6.2 Memória de Instruções - ROM

A memória de instruções foi implementada utilizando o *template* do próprio *software* **Intel Quartus Prime** de memórias ROM(*Read Only Memory*), esse tipo de memória não aceita escrita e é inicializada através de arquivos de texto contendo as instruções para o programa a ser realizado durante o processamento. Ela recebe o endereço do PC e

retorna a instrução na posição designada. Nessa implementação, a memória de instrução conta com $2^{12} = 4096$ posições de 32 bits.

A Memória de Instruções - ROM possui a seguinte implementação em Verilog:

```
module MemInst
#(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=12)
(
        input [(ADDR_WIDTH-1):0] PCin,
        input Clk,
        output reg [(DATA_WIDTH-1):0] ins
);


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

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

        always @ (posedge Clk)
        begin
                ins <= rom[PCin];
        end

endmodule
```

Código 4 – Implementação em Verilog da Memória de Instruções - ROM

4.6.2.1 Arquivo de Iniciação dos Valores na Memória de Instruções

Para a iniciação dos valores na memória foi utilizado o argumento *initial* Código 5, que inicia as posições da memória com os valores em cada linha do arquivo de texto Código 6.

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

Código 5 – Bloco de Iniação da Memória de Instruções

```
10000000000000000000000000000001
00000000000000000000000000000010
00000000000000000000000000000011
00000000000000000000000000000100
00000000000000000000000000000101
00000000000000000000000000000110
00000000000000000000000000000111
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

```
00000000000000000000000000000000
00000000000000000000000000000000
11111111111111111111111111111111
```

Código 6 – Valores Iniciados na Memória de Instruções

4.6.3 Banco de Registradores

O banco de registradores conta com 32 registradores de 32 bits, onde 31 registradores são os chamados, registradores gerais, variando da posição 0 a 30 e um registrador específico utilizado para guardar o endereço quando é realizada a instrução *Jump and Link*(registrador 31).

O Banco é sensível aos sinais de controle de **RegW, Jal e RESET**. O sinal **RESET** faz com que os valores salvos em cada registrador sejam preenchidos com 32 números 0. O **RegW**, por sua vez, é responsável por sinalizar quando irá ocorrer escrita no banco de registradores e o sinal **Jal** significa que o registrador 31 irá receber o endereço do PC atualizado. A leitura por sua vez é sempre realizada nos ciclos de clock, nas posições passadas como parâmetro **RS, RT, RD**.

O Banco de Registradores possui a seguinte implementação em Verilog:

```
module RegBk (input Clk, input Reg_W, input Jal, input RESET, input [31:0]PCin, input
    [5:0]RS, input [5:0]RT, input [5:0]RD, input [31:0]DataW, output [31:0]Out_RS, output
     [31:0]Out_RT, output [31:0]Out_RD);

integer i;

reg [31:0] Reg_Mem[31:0]; //Reg_Mem[31] reservado para o jal

        always @(posedge Clk)

                begin
                        if (RESET == 1)
                        begin

                                        for (i=0; i<32; i = i+1)
                                                begin
                                                        Reg_Mem[i]  <= 32'b0; //Reseta os
                                                             valores dos 32
                                                            Registaradores
                                                end

                        end

                        else if (Reg_W == 1) //Escreve o dado
                        begin
                                        Reg_Mem[RD] <= DataW;

                                        if (Jal == 1)
                                                        Reg_Mem[31] <= PCin;
                        end
                end
```

```
28
29 assign Out_RS = Reg_Mem[RS]; //Leitura R1
30 assign Out_RT = Reg_Mem[RT]; //Leitura R2
31 assign Out_RD = Reg_Mem[RD]; //Leitura RD
32
33 endmodule
```

Código 7 – Implementação em Verilog do Banco de Registradores

4.6.4 Unidade Lógica e Aritmética - ULA

Como o próprio nome denota, a Unidade Lógica e Aritmética, é a responsável por realizar as operações que envolvem lógica e aritmética presentes no processador. Nessa implementação podemos observar que o sinal **ALUOP** é o responsável por selecionar a operações que a ULA deve realizar de acordo com a instrução em um determinado ciclo, temos também a *flag* **zero** que é responsável por sinalizar os casos que ocorre um *branch* como demosntrado na Figura 9. A ULA é responsável pelas seguintes operações:

- **ADD** - Realiza a adição do Dado 1 com o Dado 2;
- **SUB** - Realiza a subtração do Dado 1 com o Dado 2;
- **MUL** - Realiza a multiplicação do Dado 1 com o Dado 2;
- **DIV** - Realiza a divisão do Dado 1 pelo Dado 2;
- **MOD** - Realiza o resto da divisão do Dado 1 pelo Dado 2;
- **AND** - Realiza a função lógica AND;
- **OR** - Realiza a função lógica OR;
- **NOT** - Realiza a função lógica NOT
- **XOR** - Realiza a função lógica XOR;
- **SR** - Realiza o deslocamentos dos bits à direita;
- **SL** - Realiza o deslocamentos dos bits à esquerda;
- **SLT** - Realiza a comparação do Dado 1 com o Dado 2 e retorna uma flag quando o Dado 1 $<$ Dado 2;
- **SGR** - Realiza a comparação do Dado 1 com o Dado 2 e retorna uma flag quando o Dado 1 $>$ Dado 2;
- **NOP** - Realiza a operação nula.

A Unidade Lógica e Aritmética (ULA) possui a seguinte implementação em Verilog:

```
1  module ALU (input [4:0]Alu_OP, input [31:0]D1, input [31:0]D2, output reg [31:0]R, output
       reg z);
2
3          always @(Alu_OP)
4
5                              begin
6                                              z = 0;
7
8                                              case(Alu_OP)
9
10                                                             0:begin
11                                                             R <= D1 + D2; //Add
12                                                             end
13
14                                                             1:begin
15                                                             R <= D1 - D2; //Sub
16                                                             end
17
18                                                             2:begin
19                                                             R <= D1 * D2; //Mul
20                                                             end
21
22                                                             3:begin
23                                                             R <= D1 / D2; //Div
24                                                             end
25
26                                                             4:begin
27                                                             R <= D1 % D2; //Mod
28                                                             end
29
30                                                             5:begin
31                                                             R <= D1 & D2; //And
32                                                             end
33
34                                                             6:begin
35                                                             R <= D1 | D2; //Or
36                                                             end
37
38                                                             7:begin
39                                                             R <= ~ D1; //Not
40                                                             end
41
42                                                             8:begin
43                                                             R <= D1 ^ D2; //Xor
44                                                             end
45
46                                                             9:begin
47                                                             R <= D1 >> D2; //Sr
48                                                             end
49
50                                                             10:begin
51                                                             R <= D1 << D2; //Sl
52                                                             end
53
54                                                             11:begin
55                                                                     if (D2 > D1) begin
56                                                                     R <= 1; //Slt
```

```
57                                                              z <= 1;
58                                                              end
59                                                              else R <= 0;
60                                                      end
61
62                                                      12:begin
63                                                              if (D1 > D2) begin
64                                                              R <= 1; //Sgt
65                                                              z <= 1;
66                                                              end
67                                                              else R <= 0;
68                                                      end
69
70                                                      13:begin
71                                                      R <= 0; //Nop
72                                                      end
73
74                                                      default:begin
75                                                                      R <= 0;
76                                                                      z <= 0;
77                                                                      end
78
79                                      endcase
80
81                              end
82
83  endmodule
```

Código 8 – Implementação em Verilog da Unidade Lógica e Aritmética (ULA)

4.6.5 Memória de Dados - RAM

A memória de instrução foi implementada utilizando o *template* do próprio *software* **Intel Quartus Prime** de memórias RAM (*Random Access Memory*), diferente da memória ROM, esse tipo de memória aceita escrita, porém foi implementa a possibilidade dela ser inicializa com valores através e um arquivo de texto para satisfazer as necessidades do projeto.

Como essa memória será utilizada de fato para realizar a escrita de dados, precisamos de um sinal de controle responsável por sinalizar quando ocorre escrita de dados na memória, esse é o propósito da sinal **MemW**. No caso da leitura não houve a implemetação de um sinal de controle, assim, a saída sempre contém o dado que está salvo no endereço de memória especificado pelo **mem-end**. Nessa implementação, a memória de dados, assim como a memória de instruções, conta com $2^{12} = 4096$ posições de 32 bits.

A Memória de Dados - RAM possui a seguinte implementação em Verilog:

```
module MemDados
#(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=12)
(
        input [(DATA_WIDTH-1):0] dado,
        input [(ADDR_WIDTH-1):0] mem_end,
        input Mem_W, Clk,
        output [(DATA_WIDTH-1):0] saida
);


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


        reg [ADDR_WIDTH-1:0] mem_end_reg;

        initial
        begin : INIT
                $readmemb("data_ini.txt", ram);
        end

        always @ (posedge Clk)
        begin
                // Write
                if (Mem_W)
                        ram[mem_end] <= dado;

                mem_end_reg <= mem_end;
        end

        assign saida = ram[mem_end_reg];

endmodule
```

Código 9 – Implementação em Verilog da Memória de Dados - RAM

4.6.5.1 Arquivo de Iniciação dos Valores na Memória de Dados

Para a iniciação dos valores na memória foi utilizado o argumento *initial* Código 10, que inicia as posições da memória com os valores em cada linha do arquivo de texto Código 11.

```
        initial
        begin : INIT
                $readmemb("data_ini.txt", ram);
        end
```

Código 10 – Bloco de Iniação da Memória de Dados

```
10000000000000000000000000000001
00000000000000000000000000000010
00000000000000000000000000000011
00000000000000000000000000000100
00000000000000000000000000000101
00000000000000000000000000000110
00000000000000000000000000000111
```

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
11111111111111111111111111111111
```

Código 11 – Valores Iniciados na Memória de Dados

4.7 Extensor de Bits

Como mencionado anteriormente, a arquitetura do processador funciona com 32 Bits. tendo isto em mente, existem casos onde se fazem necessários a utilização dos chamados extensores de Bits, que são os responsáveis por completar os Bits necessários para satisfazer os 32 Bits da arquitetura projetada. O número de Bits que devem ser adicionados variam de acordo o formato das instruções, Tabela 5, que estão sendo realizadas no ciclo de processamento em questão. Como podemos perceber pela Tabela 5, para esse projeto necessitamos de 3 extensores de Bits selecionados por meio do sinal de controle Extend-sel:

- **Extensor de 13 Bits (00)** - Adiciona 19 Bits 0's para completar os 32, utilizado nas instruções do **Tipo I**;
- **Extensor de 19 Bits (01)** - Adiciona 13 Bits 0's para completar os 32, utilizado nas instruções do **Tipo J**;
- **Extensor de 25 Bits (10)** - Adiciona 7 Bits 0's para completar os 32, utilizado nas instruções do **Tipo X**.

O Extensor de Bits possui a seguinte implementação em Verilog:

```
module BitExt (input [1:0]Extend_sel, input [12:0]Case13, input [18:0]Case19, input
    [24:0]Case25, output reg [31:0]Extended);

        always @(Extend_sel)

                        begin

                                        case (Extend_sel)

                                                        0:begin
                                                        Extended <= {{19{1'b0}} , Case13
                                                            };
                                                        end

```

```
13                                                  1:begin
14                                                  Extended <= {{13{1'b0}} , Case19
                                                        };
15                                                  end
16
17                                                  2:begin
18                                                  Extended <= {{7{1'b0}} , Case25};
19                                                  end
20
21                                      default:    Extended <= 32'b1;
22
23                                      endcase
24                          end
25  endmodule
```

Código 12 – Implementação em Verilog do Extensor de Bits

4.8 Multiplexadores

Como mencionado anteriormente, os multiplexadores são circuitos seletores de entradas. Para esse projeto foram utilizados multiplexadores de duas entradas. Para o funcionamento correto do multiplexador ele necessita de um terceiro sinal denominado sinal seletor, para essa implementação, seguindo o esquemático descrito na Figura 9, eles são os respectivos sinais de controle:

- **RegDst** - Seletor do Multiplexador 1, determina qual registrador sera utilizado no Reg3 (**RT, RD**);
- **AluSrc** - Seletor do Multiplexador 2, determina qual dado sera utilizado na ULA (**Dado2, Dado Extendido**);
- **Branch AND zero** - Seletor do Multiplexador 3, determina se vai ou não ocorrer o *Branch* (**PC++, Dado Extendido**);
- **Jump** - Seletor do Multiplexador 4, determina se vai ou não ocorrer o *Jump* (**Saída Mux 3, Dado Extendido**);
- **JumpR** - Seletor do Multiplexador 5, determina se vai ou não ocorrer o *Jump* (**Saída Mux 4, Dado1**);
- **Mem2Reg** - Seletor do Multiplexador 6, determina o que será escrito no banco de registradores (**Ler[MemDados], Saída ULA**).

Os Multiplexadores, com entradas (**E1, E2**) e saída (**S**), possuem a seguinte implementação em Verilog:

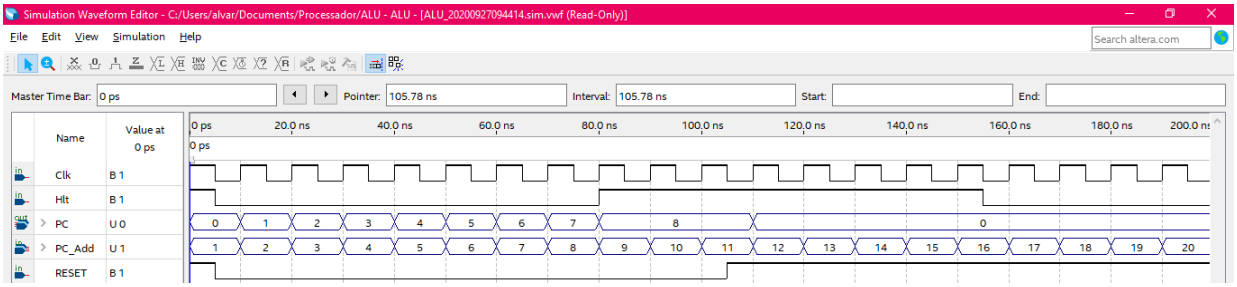
```
1  module Mux4 (input Seletor, input E1, input E2, output reg S);
2
3  always @ (Seletor)
4
5          begin
6
7                  if (Seletor == 0)
8                                  S <= E1;
9
10                 else if (Seletor == 1)
11                                 S <= E2;
12
13         end
14
15 endmodule
```

Código 13 – Implementação em Verilog de um Multiplexador de Duas Entradas

5 Resultados Obtidos e Discussões

Para o PC2 foram realizadas as simulações do funcionamento de cada bloco que compõem a Unidade de Processamento, utilizando o *Software Intel Quartus Prime* através das *Waveforms descritas abaixo*, na maioria dos casos utilizou-se a representação decimal sem o sinal para melhor entendimento.

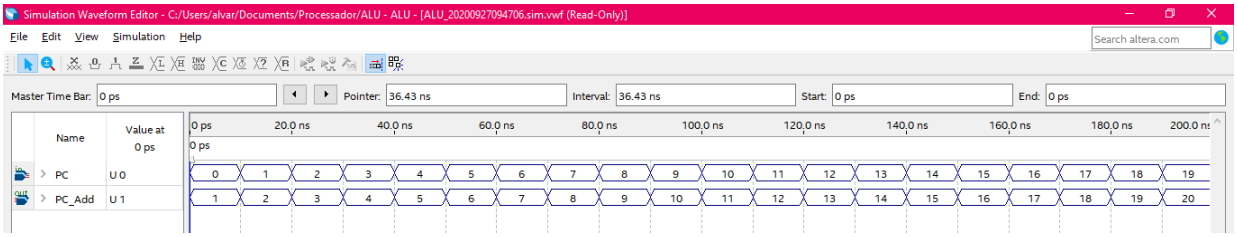
Figura 11 – *Waveform* do PC



Fonte: Autor

Podemos perceber na Figura 11, que quando os sinais de **RESET e HALT** estão desabilitados o PC é atualizado com o valor do PCAdder. Quando habilitamos o **RESET** o valor do PC volta pra primeira posição, nesse caso o PCAdder não foi resetado, pois foi simulado o bloco sozinho e a entrada do PCAdder foi uma contagem que aumenta a cada cilclo completo de *clock*, porém durante o processamento completo o PCAdder recebe o PC + 1, Figura 12, logo quando o PC volta para a primeira posição o PCAdder deverá acompanhar. No caso do **HALT** quando ele é habilidado o endereço do PC não é atualizado e o PCAdder deve acompanhar, quando falamos do processamento completo.

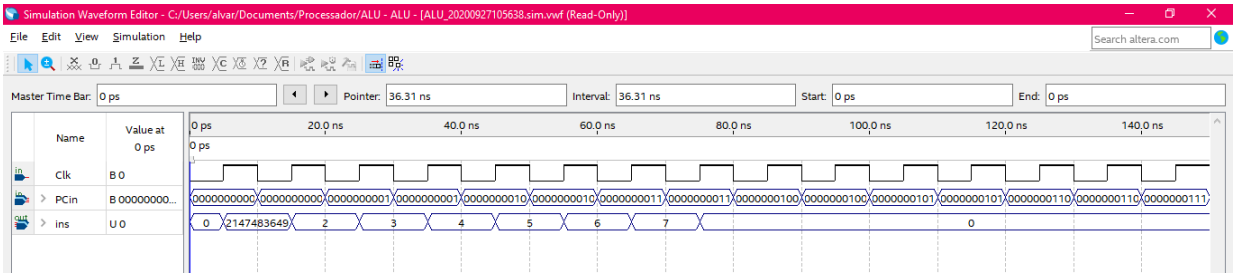
Figura 12 – *Waveform* do PCAdder



Fonte: Autor

Podemos perceber na Figura 12 que o bloco PCAdder soma um ao endereço do PC, assim quando o PC sofrer um *RESET/HALT*, no processamento completo, o mesmo acontece com o PCAdder.

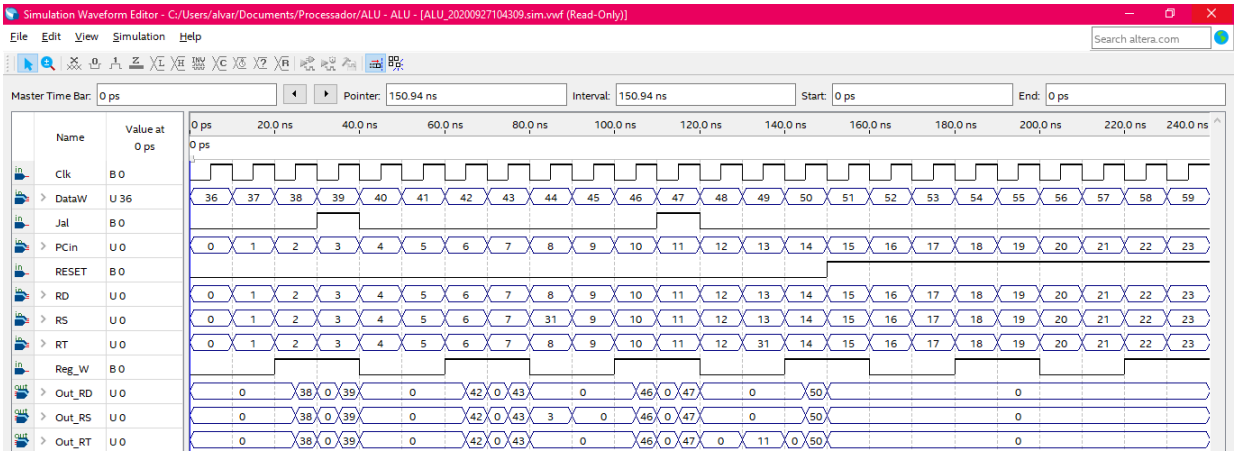
Figura 13 – *Waveform* da Memória de Instruções



Fonte: Autor

Podemos perceber na Figura 13 que a memória de instruções recebe o endereço do PC e a cada ciclo de *clock* vai percorrendo as posições da memória e tem como saída os valores iniciado na memória de instruções presentes no arquivo de texto, Código 6, na representação decimal.

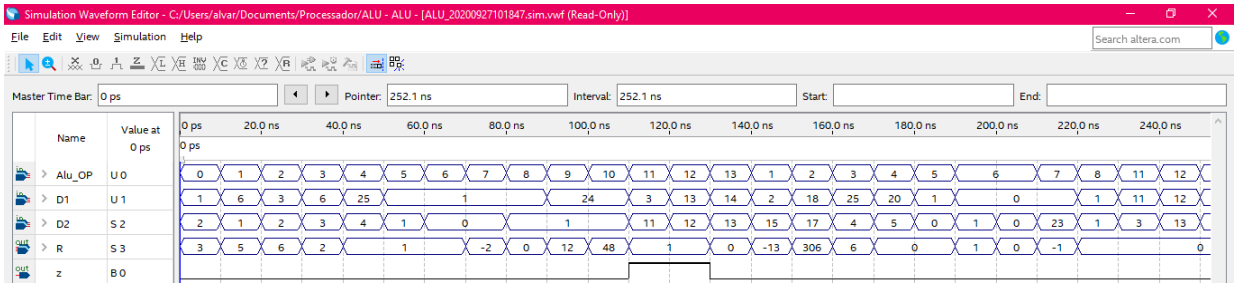
Figura 14 – *Waveform* do Banco de Registradores



Fonte: Autor

Podemos perceber na Figura 14, que o banco de registradores recebe o dado, o endereço do PC e as posições dos registradores **RS, RT, RD**. Quando o sinal de controle **RegW** está ativo, o banco é sucetível a escrita do dado na posição **RD**, a leitura por sua vez, ocorre sempre nas posições dos registradores **RS, RT, RD**. Temos também, o sinal de controle que sinaliza a instrução *Jump and Link*(**Jal**), quando ele está ativo o registrador na posição 32 recebe o endereço do PCin. O sinal **RESET**, por sua vez, apaga os dados em todas as posições no banco de registradores.

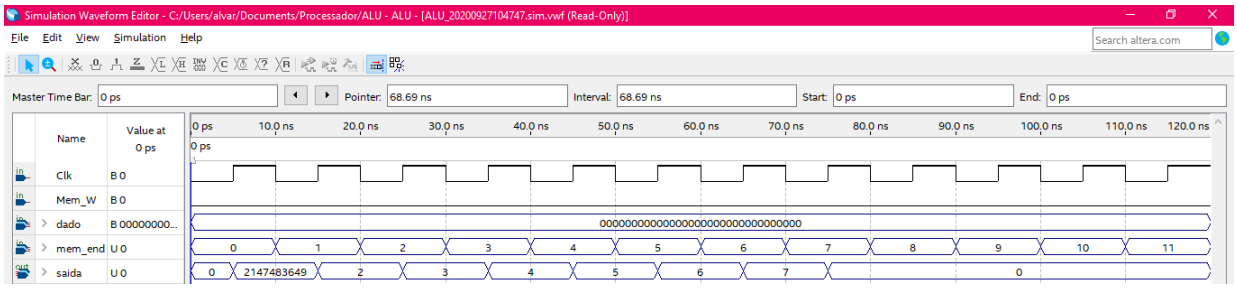
Figura 15 – *Waveform* da ULA



Fonte: Autor

Podemos perceber na Figura 15 o funcionamento de todas as operações realizadas pela ULA, subseção 4.6.4, determinadas pelos valores do **AluOP**, com seus respectivos resultados. Assim como o marcador **zero(z)** para ser utilizado no AND com o sinal **Branch**.

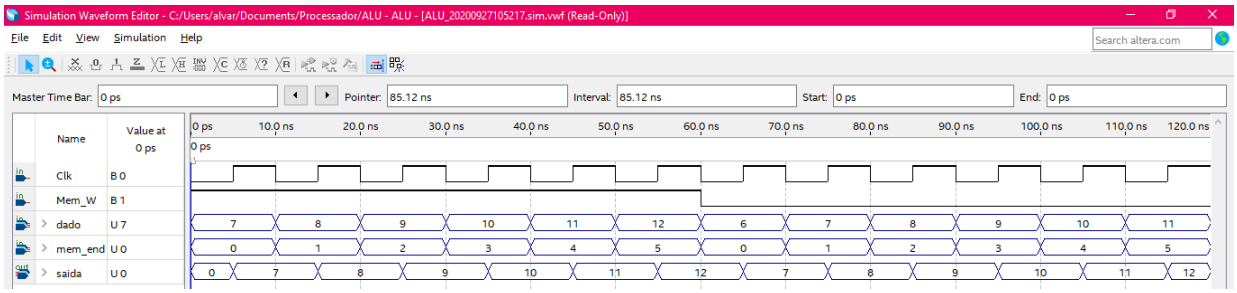
Figura 16 – *Waveform* da Memória de Dados - Leitura dos Dados Iniciados no Arquivo de Texto



Fonte: Autor

Podemos perceber na Figura 16, que a memória de dados recebe o dado a ser escrito e o endereço de escrita. A cada ciclo de *clock* vai percorrendo as posições da memória e tem como saída os valores iniciado na memória de instruções presentes no arquivo de texto, Código 6, na representação decimal.

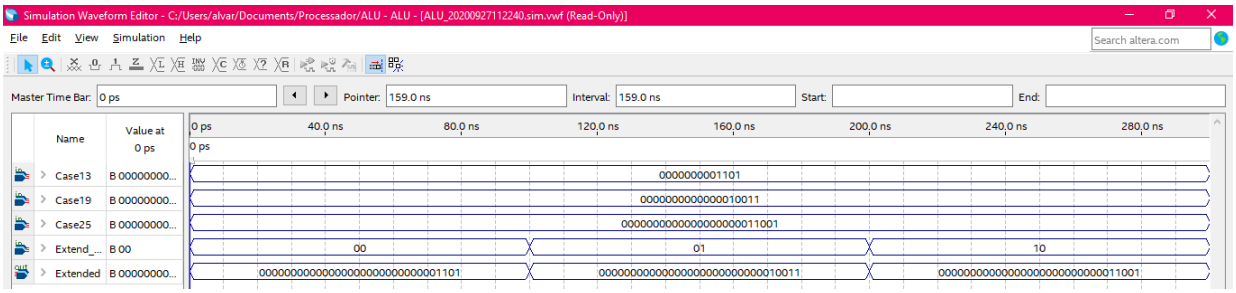
Figura 17 – *Waveform* da Memória de Dados



Fonte: Autor

Podemos perceber nas Figura 17, que a memória de dados recebe o dado a ser escrito e o endereço de escrita. A cada ciclo de *clock* vai percorrendo as posições da memória e tem como saída os valores, na representação decimal, dos dados que foram passados quando o sinal de controle que habilita a escrita, **MemW**, estava ativo.

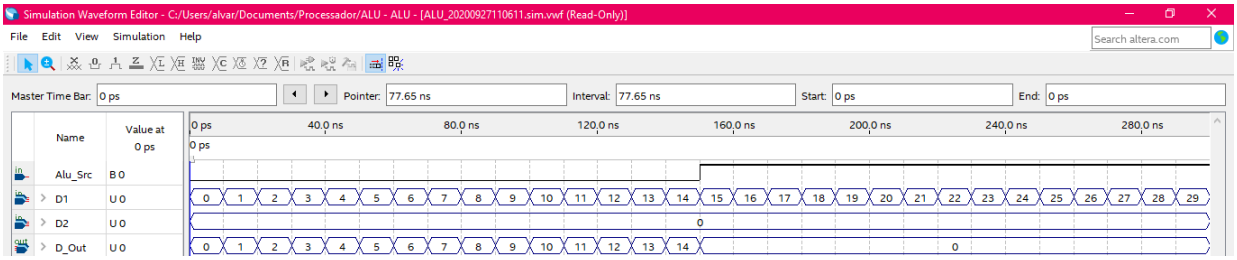
Figura 18 – *Waveform* do Extensor de Bits



Fonte: Autor

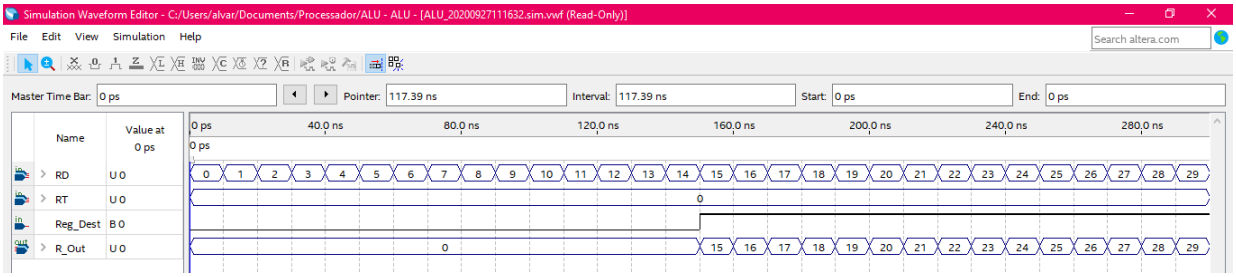
Podemos perceber na Figura 18, que o extensor de Bits recebe um sinal com 13 Bits, um sinal com 19 Bits e um Sinal com 25 Bits para serem extendidos. O sinal de controle **Extend-sel** realiza a seleção do número de Bits a ser concatenado para completar os 32 Bits de acordo com o tamanho original do dado.

Figura 19 – *Waveform* do Multiplexador de duas entradas variação 1



Fonte: Autor

Figura 20 – *Waveform* do Multiplexador de duas entradas variação 2



Fonte: Autor

Podemos perceber nas Figuras 19 e 20 as variações utilizadas no projeto dos multiplexadores de 2 entradas. Onde eles são sensíveis ao respectivos sinais de controle, subseção 4.8, e tem entradas de 32 Bits ou 6 Bits (Registradores). Quando os sinais de seleção estão ativos ele "passa"a segunda entrada e quando eles não estão ativos ele "passa"a primeira entrada.

6 Considerações Finais

Com a descrição dos módulos que compõem a unidade de controle em Verilog, somado as *Waveforms* de cada módulo, obtem-se uma maior clareza no funcionamento da unidade de processamento como um todo. Assim, alguns conceitos que antes eram abstratos se tornaram concretos, como a implemetação das memórias de dados e instruções.

Além disso, devemos manter em mente o processo de desenvolvimento do módulo de entrada e saída que terá extrema importancia para os laboratórios seguintes e, provavelmente, trará consigo alguns desafios, principalmente pelo fato de o acesso aos FPGAs pelo laboratório remoto ser limitado.

Assim, até o presente momento o projeto está de acordo com as expectativas. O próximo PC irá abranger a implentação da unidade de controle e a finalização do projeto em si.

Referências

1 TOCCI, R.; WIDMER, N.; MOSS, G. *Sistemas Digitais: Princípios e Aplicações*. 11ª edicao. ed. São Paulo: Pearson, 2011. Citado na página 9.

2 FERDJALLAH, M. *Introduction to Digital Systems: Modeling, Synthesis, and Simulation Using VHDL*. 1st edition. ed. New Jersey: John Wiley and Sons, 2011. Citado na página 16.

3 STALLINGS, W. *Arquitetura e Organização de Computadores*. 8th edition. ed. São Paulo/Sp Brasil: Pearson, 2010. Citado 2 vezes nas páginas 17 e 18.

4 PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design: The Hardware/Software Interface*. 5th edition. ed. [S.l.]: Morgan Kaufman, 2014. Citado 5 vezes nas páginas 18, 19, 21, 22 e 23.