ID: ******

Desenvolvimento de Sistemas Computacionais
Laboratório de Arquitetura de Computadores
PC2

São José dos Campos - Brasil

10 de Outubro de 2020

ID: ******

Desenvolvimento de Sistemas Computacionais
Laboratório de Arquitetura de Computadores
PC2

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.

Docentes: Prof. Dr. Tiago de Oliveira - Prof. Sérgio Ronaldo Barros dos Santos

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

10 de Outubro de 2020

Resumo

Esse relatório tem o objetivo de relatar o desenvolvimento de um processador baseado em mips 32 bits, capaz de processar um conjunto de instruções especifico. Indicando a função de cada uma dessas instruções, a forma com que elas foram codificadas, como foi efetuado o desenvolvimento em verilog de cada uma das estruturas da CPU, e como ocorreu os testes em Waveform para verificar seu correto funcionamento.

Palavras-chaves: mips. FPGA. CPU. Arquitetura e organização de computadores.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

Segundo Paterson (2), "A tarefa que o projetista de computador desempenha é complexa: determinar quais atributos são importantes para um novo computador, depois projetar um computador para maximizar o desempenho enquanto permanece dentro das restrições de custo, potência e disponibilidade. Essa tarefa possui muitos aspectos, incluindo o projeto do conjunto de instruções, a organização funcional, o projeto lógico e a implementação."(Paterson, 2017)

Uma dessas arquiteturas são os computadores do tipo RISC(Reduced instruction set computers), que ganharam popularidade buscando aumentar a performance sobre processadores do tipo CISC(Complex Instruction set Computer) ao possuir um design mais simples com número reduzido de instruções de tamanho fixo e registradores mais generalizados. Um dos processadores baseados no RISC que ganharam o mercado é a arquitetura de propósito geral MIPS, com sua primeira versão desenvolvida em 1985, pela empresa de John LeRoy Hennessy.

Em um outro âmbito, o FPGA, circuito lógicos programável, é cada vez mais utilizado por ser um dispositivo reprogramável em nível de hardware podendo se adaptar as mais diversas e específicas aplicações. (4)

Unindo essa capacidade de programação em hardware dos FPGAs e lógica das arquiteturas Risc vem esse projeto de implementação de um processador seguindo as diretrizes RISC em FPGA

2 Objetivos

2.1 Geral

O relatório busca o desenvolvimento de um processador baseado no MIPS 32 bits e sua implementação em uma plataforma de hardware, o kit FPGA DE2-115 Cyclone IV, que será programado através do Software Quartus II utilizando a linguagem de hardware Verilog. A CPU deverá ao final da implementação, ser capaz de receber um conjunto específico de instruções e dados, tanto em sua memória interna, quanto das entradas do kit FPGA , corretamente executá-los e mostra-los no kit.

2.2 Específico

Inicialmente foi selecionado um conjunto de instruções que executasse os principais algorítimos computacionais, foi criado um formato para essas instruções e planejado o esquemático completo do projeto.

Em seguida foram desenvolvidas e programadas em verilog os seguintes componentes e etapas:

- Desenvolvimento da ULA
- Desenvolvimento dos Registradores
- Desenvolvimento e integração das memórias e program conter
- Desenvolvimento e integração da ULA com a CPU
- Integração com o FPGA

Por fim ocorreu simulações na CPU usando waveform.

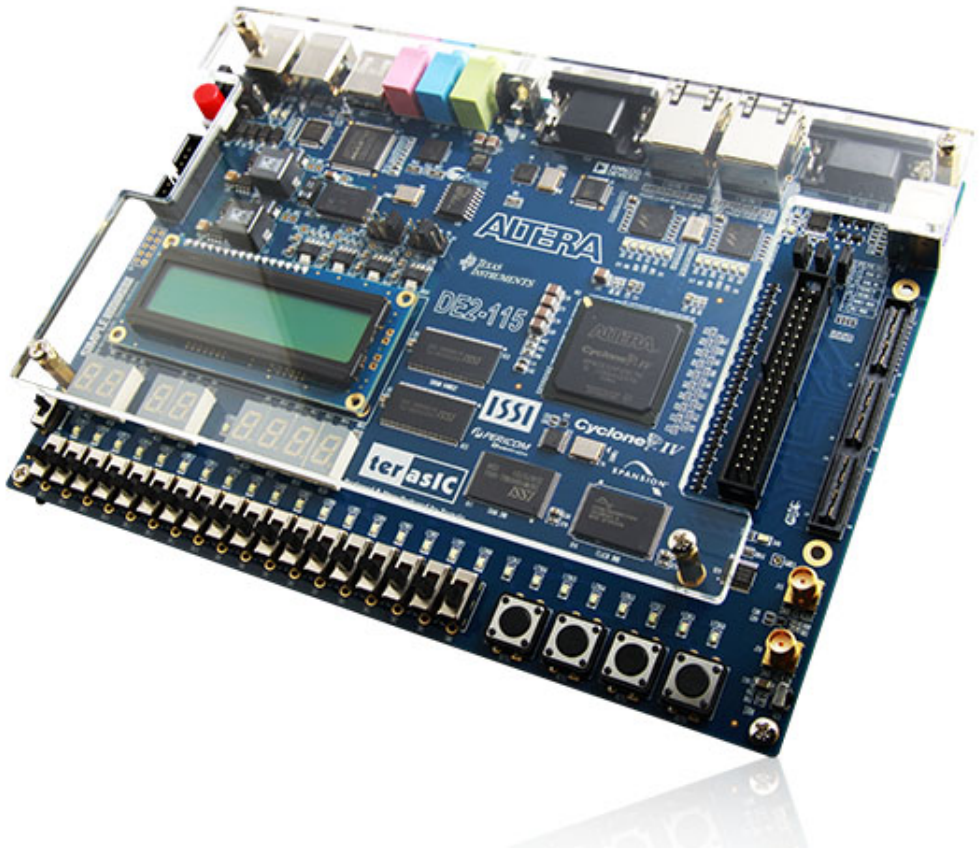
3 Fundamentação Teórica

3.1 FPGA

FPGA (Field programmable gate array) é um circuito integrado passível de configuração em nivel de hardware ao possuir um conjunto de componentes lógicos que podem ser programados e interconectados de diferentes formas, possibilitando os mais diferentes circuitos lógicos. (4)

O trabalho será realizado no dispositivo FPGA DE2-115 Cyclone IV Figura 1, um kit FPGA educativo da Altera (1)

Figura 1 – DE2-115 Cyclone IV



Fonte: Altera (1)

Para programar o FPGA será utilizado o *Software* Quartus II com linguagem de descrição *Hardware* Verilog.

3.2 RISC

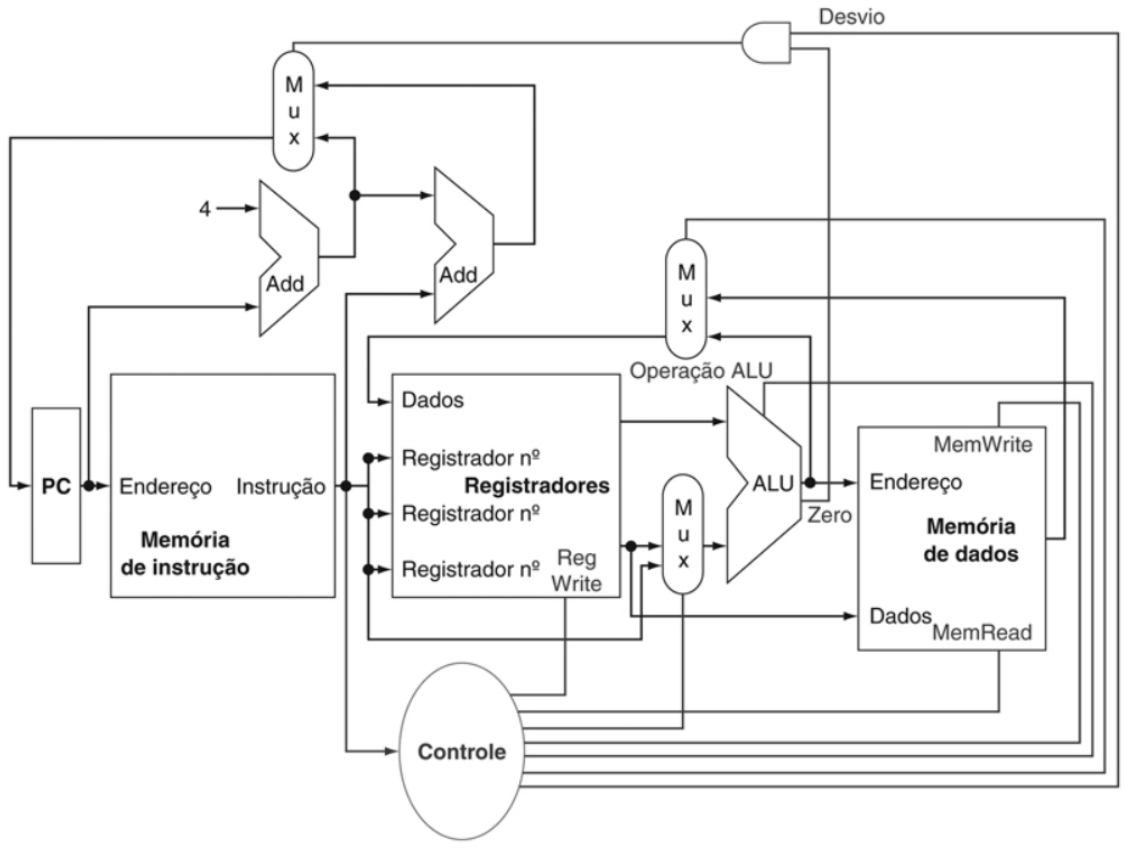
Risc, é uma arquitetura de microcomputadores que utiliza um número reduzido de instruções. Os primeiros projetos dessa arquitetura vieram por volta de 1975 buscando se contrapor ao consolidado CISC, que possuía um conjunto complexo de instruções, ao passo que o risc tem uma arquitetura mais simples e por consequência mais rápida em execução. (3)

3.3 MIPS

A arquitetura MIPS é um conhecido exemplo de processador RISC. Apesar de ser multi cíclica a trataremos como monocíclica ou seja consiste em executar cada uma das funções a cada ciclo do clock, que apesar de ser mais lenta facilita o seu compreendimento. O processadores mips original tinha arquitetura de 32 bits com estruturas tamanho fixo, 32 registradores de propósito geral. E nele será inspirado o projeto desse relatório. (2)

Observamos na Figura 2 o caminho dos dados bem como o controle de um processador mips básico:

Figura 2 – Arquitetura MIPS



Fonte: Patterson 2007 (2)

As funções dos componentes presentes são as seguintes:

- Memória de instruções: Local onde ficam armazenados as instruções produzidas pelo compilador e que devem ser executadas.
- PC (Program Counter): É um registrador responsável por manter o dado de qual a próxima instrução que será executada.
- Banco de Registradores: Armazenamento da CPU, mais rápido que a memória principal,é o local onde são guardados dados atualmente utilizados no processamento
- Memória de dados: Local onde ficam armazenados dados que serão futuramente utilizados
- ULA (Unidade Logica Aritmética) Responsável por realizar as operações lógicas e aritméticas
- Unidade de controle: Responsável por coordenar o caminho dos dados de acordo com o tipo de instrução a ser executado
- MUX(Multiplexador): Responsável por selecionar entre diversas entradas qual sera a saída baseado em um sinal controle.
- ADD (Adder): realiza a operação aritmética de Adição.

4 Desenvolvimento

A arquitetura escolhida para processador é trazida do conceito RISC seção 3.2 baseado no processador MIPS seção 3.3 visto seu conjunto reduzido de instruções com execução monocíclica, ou seja, serão completamente executadas a cada ciclo do clock. Possui uma palavra de 32 bits e um banco de registradores com capacidade para 32 palavras. Apesar de seguir a arquitetura MIPS o processador a ser desenvolvido possui inicialmente um conjunto menor de instruções bem como mudanças em seu endereçamento.

4.1 Conjunto de instruções

Possui as seguintes instruções agrupadas por tipo de operação:

Operações aritiméticas: Soma (ADD), Soma Imediata (ADDi), Subtração (SUB) e Subtração imediata (SUBi)

Operações Lógicas: Negação lógica (NOT), E lógico (AND), E lógico Imediato (ANDi), OU lógico (OR), OU lógico imediato (ORi), deslocamento de bits para a esquerda (SL), deslocamento de bits para a direita(SR)

Operações Condicionais: Set on Less Than(SLT) e Set on Less Than Imediato(SLTi)

Branch Condicionais: Branch equal(BEQ), Branch not equal (BNQ)

Saltos incondicionais: Jump (J) e Jump Imediato (Ji)

Escrita e leitura da memória: Carregamento no registrador (Load), Carregamento Imediato no registrador (Loadi) e Carregamento na memória (Store)

Nenhuma operação(NOP) e Parar processamento (HLT)

Integração com FPGA: Receber entrada (In) e mostrar saida (Display)

Essas instruções combinadas permitem a realização das principais operações matemáticas e de lógica de programação necessários, sendo endereçadas de 4 formas como dado na Tabela 1.

Tabela 1 – Modos de Endereçamento

R	op	rd	rs	rd	0
I	op	rd	rs	IM16	
J	op	Addr26			
K	op	Rd	Addr21Addr21		

Fonte: Autor

A Figura 3 é uma tabela com todas as instruções do processador, suas respectivas funções e seu formato quando armazenada na memória de instrução

Figura 3 – Instruções

		31-26	25-21	20-16	15-11	10-0
Operação		6 bits OP	5 bits	5 bits	5 bits	11 bits
NOP	No-Op	000000	0			
ADD	Rd = Rs + Rt	000001	Rd	Rs	Rt	0
ADDi	Rd = Rs + IM16	000010	Rd	Rs	IM16	
SUB	Rd = RS – RT	000011	Rd	Rs	Rt	0
SUBi	Rd = RS – IM16	000100	Rd	Rs	IM16	
NOT	Rd= ~Rs	000101	Rd	Rs	0	
AND	Rd=Rs & Rt	000110	Rd	Rs	Rt	0
ANDi	Rd = Rs & IM16	000111	Rd	Rs	IM16	
OR	Rd = Rs | Rt	001000	Rd	Rs	Rt	0
ORi	Rd = Rs | IM16	001001	Rd	Rs	IM6	
SL	Rd = Rs << SHIFT16	001010	Rd	Rs	SHIFT16	
SR	Rd = Rs >> SHIFT16	001011	Rd	Rs	SHIFT16	
SLT	Rd = (Rs ± < Rt ±) ? 1 : 0	001100	Rd	Rs	Rt	0
SLTi	Rd = (Rs ± < IM16 ±) ? 1 : 0	001101	Rd	Rs	IM16	
BEQ	IF RS = RT, PC += OFF16	011110	Rs	Rt	Off16(mi)	
BNQ	IF RS ≠ RT, PC += OFF16	011111	Rs	Rt	Off16(mi)	
J	PC = Rs :: Addr32(mi)	010000	Rs	0		
Ji	PC = Addr26(mi)	010001	Addr26(MEMinst)			
Load	Rd = MEMdata[Addr21]	010010	Rd	Addr21(MEMdata)		
Loadi	Rd = MEMdata[IM21]	010011	Rd	IM21		
STORE	MEMdata[Addr21] = Rd	010100	Rd	Addr21(MEMdata)		
HLT	Parar processamento	010101	0			
In	Rd = SW18	010110	Rd	000	SW18(FPGA)	
Display	Saída para displays	010111	Rd	Display		

Fonte: Autor

- No-Op refere-se a não ocorrer nenhuma operação,
- Rd, Rs e Rt são o registrador destino e origens
- IM16 é um valor imediato de 16 bits inserido na própria instrução
- SHIFT16 é um valor imediato de 16 bits referente ao numero de vezes que ocorrerá o deslocamento de bits
- PC refere-se ao registrador do *Program Counter*
- OFF16 é um valor imediato de 16 bits que será somado ao PC para deslocá-lo de posição

- Addr32 é um endereço da memória de instruções guardado em Rs que será substituindo no *Program Counter*

- Addr26 é um endereço imediato da memória de instruções inserido na própria instrução que será substituido no *Program Counter*

- MEMdata[ ] é o valor contido na memória de dados em um determinado endereço

- addr21 é um endereço da memória de dados inserido na própria instrução

- SW18 é o correspondente em 18 bits do estado dos 18 *switches* do FPGA

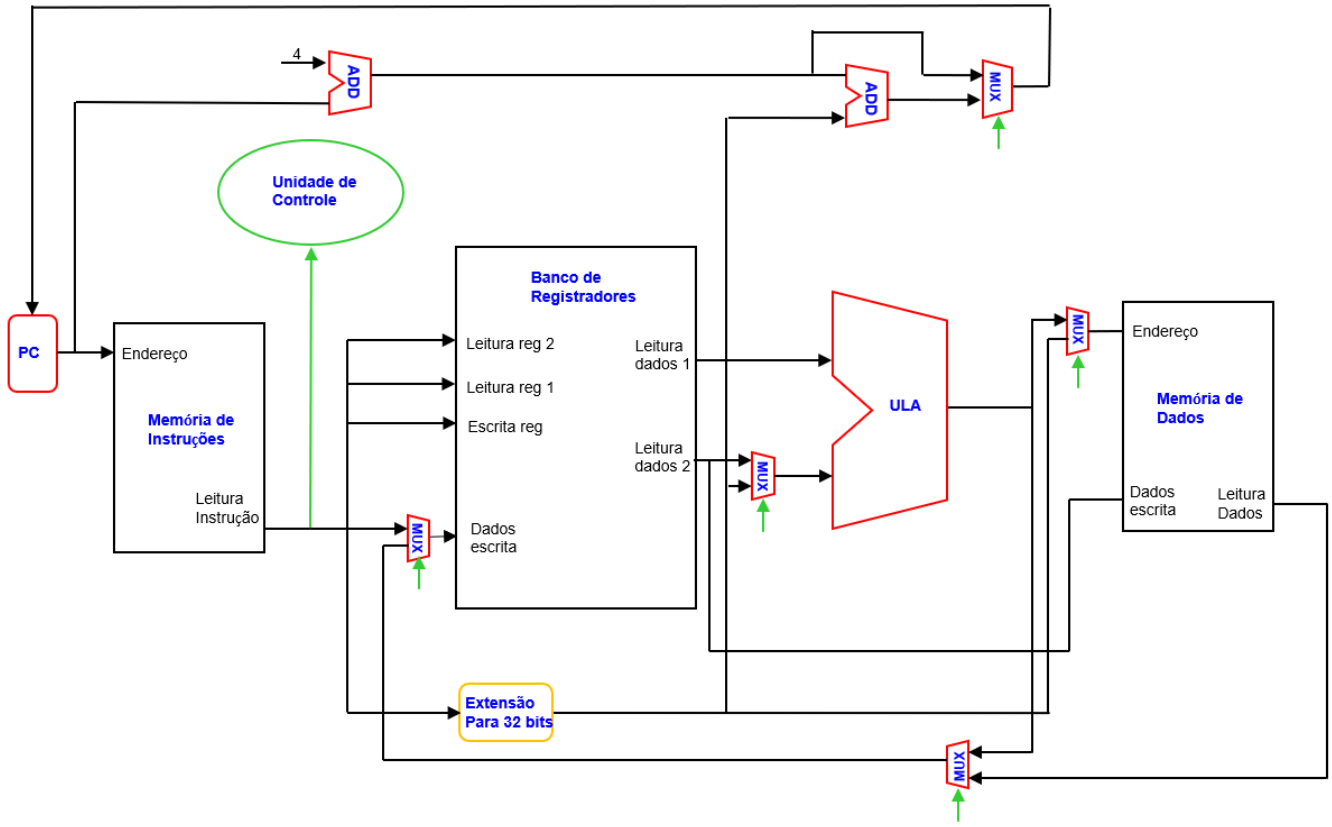
Vale ressaltar que a as operações da segunda coluna são realizadas com os valores dos registradores e das memórias já nas instruções os bits inseridos são os endereços desses dados

Os primeiros 6 bits de cada uma das instruções estarão reservados para o *opcode* da instrução, ditando e codificando de forma única para a Unidade de controle qual operação deverá ser executada. Vale ressaltar que o *opcode* poderia ser incorporado em apenas 5 bits, porem visto a dificuldade em reestruturar todas as instruções caso haja a necessidade de expandir além de 32 instruções foi optado em reservar 6 bits.

4.2 Arquitetura

Para que o conjunto de instruções possa ser executado por completo foi pensada o seguinte esquemático Figura 4 baseado na organização do MIPS em 32 bits

Figura 4 – Esquemático



Fonte: Autor

Ver os componentes mips na seção 3.3 para a definição de cada.

As memórias estão separas em memória de dados e memória de instruções aceitando 1000 palavras, cada uma poderia ter um tamanho teórico de 2^{32} porem foi optado em limita-las para reduzir o tempo de compilação.

O banco de registradores possui 32 registradores de 32 bits O componente denominado extensão para 32 bits estenderá os valores imediatos da instrução para 32 bits para que possam ser processados ou armazenados corretamente

A unidade de controle decide usando os multiplexadores a partir de cada tipo de instrução qual sera o caminho dos dados bem como quais componentes serão utilizados para executá-los. Para facilitar a visualização do esquemático suas conexões foram simplificadas.

O extensor para 32 bits faz a extensão dos bits dos valores imediatos das instruções para o tamanho de 32 bits.

O *Program Counter* mantêm registro da próxima instrução da memória de instruções a ser executada, este pode incrementar, sofrer um *branch* ou receber um valor específico dependendo da instrução executada por ser controlado pela Unidade de Controle.

A ULA efetua todas as operações logicas e aritméticas escolhidas em seção 4.1 podendo efetuar operações entre registradores ou entre registrador e imediato de acordo com o que for ditado pela Unidade de Controle

4.2.1 Program Counter

O modulo PC em verilog contem tanto o registrador addr que mantêm gravado qual instrução está atualmente sendo executada bem como os seus controladores que podem causar um branch, somando um valor *bOff* ao *addr*, ou causar um jump, fazendo *addr* receber *jVal*, de acordo com o estado dos flags *branch* e *jump* respectivamente.

Código do arquivo PC.v

```
1  module PC(clk, jump, branch, jVal, bOff, addr);
2
3          input clk;
4          input jump, branch;
5
6          input [15:0] bOff;
7          input [31:0] jVal;
8
9          output reg [31:0] addr;
10
11         always @ (posedge clk )
12         begin
13
14                 if (jump)
15                         addr <= jVal;
16                 else if (branch)
17                         addr <= addr + $signed(bOff);
18                 else
19                         addr <= addr + 1;
20
21         end
22 endmodule
```

4.2.2 Registradores

O modulo *registers* contem 32 registradores de tamanho 32, entradas de dados, entradas de endereço para leitura e escrita nesses registradores e saída de dados. O flag *RegWrite* irá indicar se ocorrerá escrita em um dos registradores

Código do arquivo registers.v Código do arquivo registers.v

```
1  module registers(clk, RegWrite, addrR_reg1, addrR_reg2, addrR_reg3, addrW_reg, write_reg,
       read_reg1, read_reg2, read_reg3);
2
3          input clk, RegWrite;
4          input [4:0] addrR_reg1, addrR_reg2, addrR_reg3, addrW_reg;
5          input [31:0] write_reg;
6          output [31:0] read_reg1, read_reg2, read_reg3;
7          integer Spulse = 0;
8
9
```

```
10          reg [31:0] regs[31:0]; //32 registradores de tamanho 32
11
12
13          always @ (posedge clk )
14          begin
15
16                  if (Spulse==0)
17                  begin
18
19                          regs[0] <= 32'd0;
20
21                  Spulse = 1;
22                  end
23
24                  if(RegWrite)
25                  begin
26                          regs[addrW_reg] <= write_reg;
27                  end
28          end
29
30          assign read_reg1 = regs[addrR_reg1];
31          assign read_reg2 = regs[addrR_reg2];
32          assign read_reg3 = regs[addrR_reg3];
33
34  endmodule
```

4.2.3 ULA

o módulo *ULA* receberá 2 entradas de 32 bits, A e B, realizará uma das operações escolhidas codificadas por *sel* e disponibilizará a saída em *out*. Vale ressaltar que o calculo de igualdade da instrução branch também ira ocorrer dentro desse módulo enviando a resposta pelo *output branch*

Código do arquivo ULA.v Código do arquivo ULA.v

```
1   module ULA(sel, A, B, branch, out);
2
3           output reg branch;
4           input [3:0] sel;
5           input signed [31:0] A, B;
6           output reg [31:0] out;
7
8
9           always @(sel or A or B)
10          begin
11
12                  branch = 0;
13                  case (sel)
14                          4'b0000: out =  32'b0;
15                          4'b0001: out = A + B;
16                          4'b0010: out = A - B;
17                          4'b0011: out = ~A   ;
18                          4'b0100: out = A & B;
19                          4'b0101: out = A | B;
20                          4'b0110: out = A << B;
21                          4'b0111: out = A >> B;
22                          4'b1000: out = B + 0; //Loadi
```

```
                        4'b1001: out = (A < B) ? 32'b1 : 32'b0;//SLT
                        4'b1010:        branch = (A == B) ? 1 : 0;//BEQ
                        4'b1011:        branch = (A != B) ? 1 : 0;//BNE
                        default : out =  32'b0;
                endcase
        end
endmodule
```

4.2.4 Memória de Instruções

O modulo *inst mem* recebe o endereço *addr* vindo do *PC* acessa esse endereço em um dos 100 registradores e coloca sua instrução para leitura em *read*

Código do arquivo inst mem.v Código do arquivo inst mem.v

```
module inst_mem(clk, addr, read);

        input clk;
        input [31:0] addr;
        output [31:0] read;

        integer Spulse = 0;

        reg [31:0] mem[100:0]; // 100 de tamanho 32


        always @ (posedge clk )
        begin

                if (Spulse==0)
                begin
                        //$readmemb("./instructions.txt", mem);

                        mem[0] <= 32'b00000000000000000000000000000000;

                        Spulse = 1;
                end
        end

        assign read = mem[addr];

endmodule
```

4.2.5 Memória de Dados

O módulo *data mem* contem 100 registradores de tamanho 32, uma entrada de dados, uma entrada de endereços para leitura e escrita e uma saída de dados. O flag *MemWrite* irá indicar se ocorrerá escrita na memória.

Código do arquivo data mem.v

```
module data_mem(clk, MemWrite, addr, write, read);

        input clk, MemWrite;
        input [20:0] addr;
```

```
5         input [31:0] write;
6         output [31:0] read;
7         integer Spulse = 0;
8
9         reg [31:0] mem[100:0]; // 100 de tamanho 32
10
11
12        always @(posedge clk)
13        begin
14                if (Spulse==0)
15                begin
16                        mem[0] <= 32'd0;
17
18                Spulse = 1;
19                end
20
21                if(MemWrite)
22                begin
23                        mem[addr] <= write;
24                end
25        end
26
27        assign read = mem[addr];
28
29 endmodule
```

4.2.6 extensores

Os extensores convertem os diferentes tamanhos de dados imediatos para o padrão de 32 bits utilizado pelos módulos, para isso foi utilizado a inserção em registrador utilizando *signed*, que converte o dado de inserção para o tamanho do destino, porem somente ocorrerá corretamente se o valor estiver escrito em complemento de dois, o que sempre ocorrerá visto que todas as operações do processador são feitas levando em conta essa notação.

Código do arquivo data sign ext 16.v

```
1  module sign_ext_16(in16, out32);
2
3         input [15:0] in16;
4
5         output reg [31:0] out32;
6
7         always @(in16)
8         begin
9
10        out32 <= $signed(in16);
11
12        end
13 endmodule
```

4.2.7 Multiplexadores

Foi utilizado a mega função parametrizada BUSMAX inclusa no *Quartus* que tem a função de um multiplexador para barramentos de tamanhos variados.

4.2.8 Unidade de Controle

O módulo *UC* recebe o *Opcode* dado pela instrução e retorna o correto estado de cada um dos flags e seletores permitindo a integração entre os diversos componentes. Como o modulo ainda esta incompleto foram mostrados no código apenas as 2 primeiras instruções.

Código do arquivo data UC.v

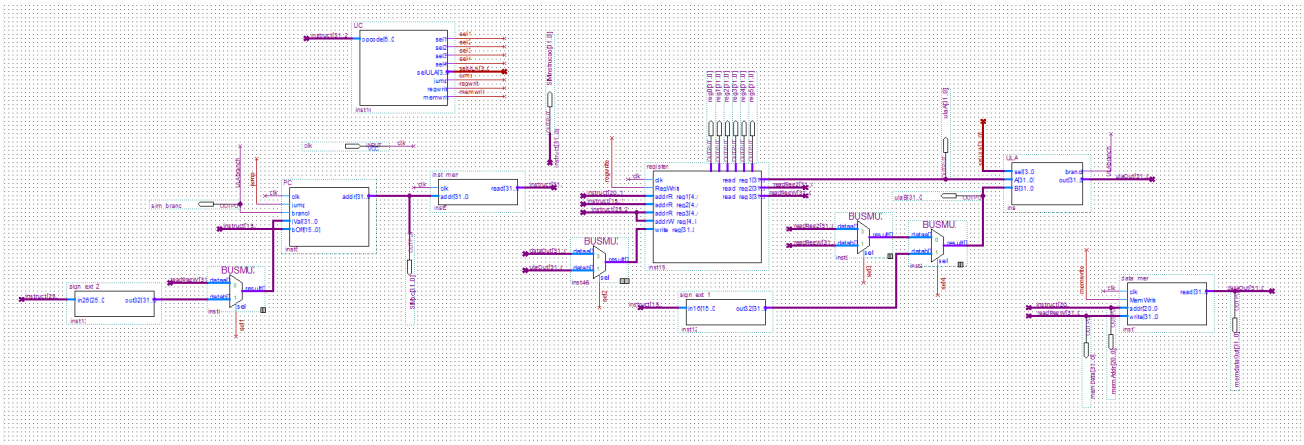
```
1  module UC(opcode, sel1, sel2, sel3, sel4, selULA, jump, regwrite, memwrite);
2
3          input [5:0] opcode;
4          output reg sel1, sel2, sel3, sel4, jump, regwrite, memwrite;
5          output reg [3:0] selULA;
6
7
8
9
10 always @(opcode)
11 begin
12
13                 case(opcode)
14
15                         6'b000001: // ADD
16                         begin
17                                 sel1 =     1'b0;
18                                 sel2 =     1'b1;
19                                 sel3 =     1'b0;
20                                 sel4 =     1'b0;
21                                 selULA =   4'b0001;
22                                 regwrite = 1'b1;
23                                 memwrite = 1'b0;
24                                 jump =     1'b0;
25                         end
26
27                         6'b000010: // ADDi
28                         begin
29                                 sel1 =     1'b0;
30                                 sel2 =     1'b1;
31                                 sel3 =     1'bX;
32                                 sel4 =     1'b1;
33                                 selULA =   4'b0001;
34                                 regwrite = 1'b1;
35                                 memwrite = 1'b0;
36                                 jump =     1'b0;
37                         end
38
39                         default:
40                         begin
41                                 sel1 =     1'b0;
42                                 sel2 =     1'b0;
```

```
43                                  sel3 =      1'b0;
44                                  sel4 =      1'b0;
45                                  selULA =    4'b0000;
46                                  regwrite =  1'b0;
47                                  memwrite =  1'b0;
48                                  jump =      1'b0;
49                          end
50                  endcase
51  end
52  endmodule
```

4.2.9 Interligação entre módulos

A interligação entre módulos foi feita inicialmente em forma de diagrama de blocos, para facilitar a manipulação e depuração do processador, cada um dos módulos em verilog foram transformados em blocos e corretamente interligados da forma mostrada na Figura 5:

Figura 5 – Diagrama de blocos



Fonte: Autor

5 Resultados Obtidos e Discussão

Após o desenvolvimento e interligação dos blocos foram efetuados simulações: Na primeira linha teremos o sinal do clock com um ciclo de 20 ns para melhor visualização. Na segunda linha teremos o endereço do *Program Counter* mostrado em decimal, seguido pela respectiva instrução da memória de instruções representada em binário. Por fim é mostrado os valores armazenados nos 6 primeiros registradores do banco de registradores.

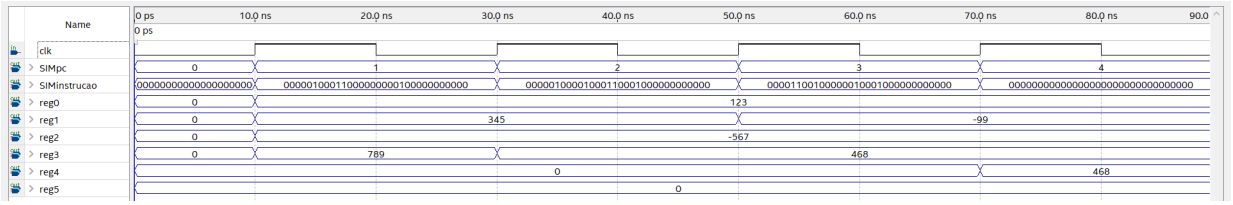
Para a primeira simulação buscaremos verificar o funcionamento efetivo das operações aritméticas da ULA bem como funcionamento do banco de registradores

Os seguintes valores para os registradores e Instruções serão inseridos na primeira subida do clock:

```
1  regs[0] <= 32'd123;
2  regs[1] <= 32'd345;
3  regs[2] <= -32'd567;
4  regs[3] <= 32'd789;
5
6  mem[1] <= 32'b000001_00010_00000_00001_00000000000; //SOMA rd[2]=rs[0] + rt[1]
7  mem[2] <= 32'b000011_00011_00001_00010_00000000000; //SOMA rd[3]=rs[1] - rt[2]
8  mem[3] <= 32'b000010_00011_00001_0000000000010101; //SOMA rd[3]=rs[1] + imm16(21)
```

A simulação é decorrida até o tempo 90ns na Figura 6

Figura 6 – Simulação 1



Fonte: Autor

Observamos que na primeira instrução é efetuado corretamente a adição 123+345 resultando em 468 que é inserido no reg3. Na segunda instrução somamos 468 com -567 resultando em -99 que é inserido em reg1.Por fim na terceira instrução subtraímos -99 de -567, que resulta em 468 que é inserido em reg4.

Para a segunda simulação buscaremos verificar o funcionamento efetivo das operações lógicas da ULA.

Os seguintes valores para os registradores e Instruções serão inseridos na primeira subida do clock:

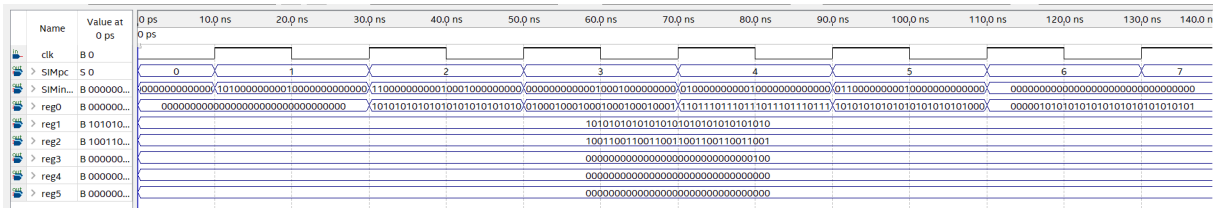
```
1  regs[0] <= 32'b0;
```

```
regs[1] <= 32'b10101010101010101010101010101010;
regs[2] <= 32'b10011001100110011001100110011001;
regs[3] <= 32'd4;

mem[1] <= 32'b000101_00000_00001_00000_00000000000; //NOT rd[0]= ~rs[1]
mem[2] <= 32'b000110_00000_00001_00010_00000000000; //AND rd[0]=rs[1] & rt[2]
mem[3] <= 32'b001000_00000_00001_00010_00000000000; //OR rd[0]=rs[1] | rt[2]
mem[4] <= 32'b001010_00000_00001_0000000000000101; //SL rd[0]=rs[1] << imm(5)
mem[5] <= 32'b001011_00000_00001_0000000000000101; //SR rd[0]=rs[1] >> imm(5)
```

A simulação é decorrida até o tempo 140ns na Figura 7

Figura 7 – Simulação 2



Fonte: Autor

Na primeira instrução foi realizado corretamente a negação lógica do reg1, para a segunda instrução foi feito o AND lógico entre reg1 e reg2, para terceira instrução foi realizado o OR lógico entre r1 e r2, para a quarta e quinta instrução foram realizados shifts lógicos de 5 posições para a esquerda e direita respectivamente.

Para a terceira simulação buscaremos verificar o funcionamento efetivo das instruções *branch* e SLT.

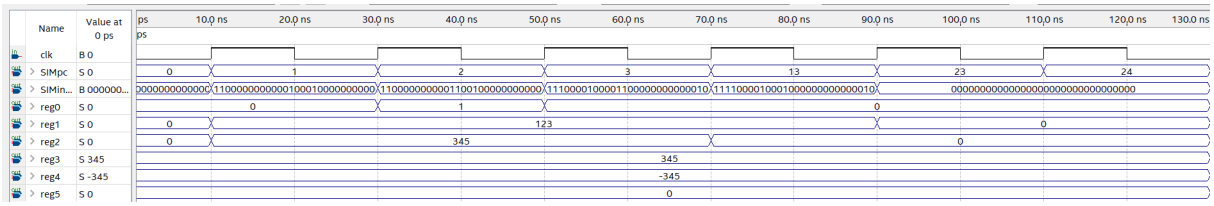
Os seguintes valores para os registradores e Instruções serão inseridos na primeira subida do clock:

```
regs[0] <= 32'd0;
regs[1] <= 32'd123;
regs[2] <= 32'd345;
regs[3] <= 32'd345;
regs[4] <= -32'd345;

mem[1] <= 32'b001100_00000_00001_00010_00000000000; //SLT rd[0]= (rs[1] < rt[2])? 1 : 0
mem[2] <= 32'b001100_00000_00011_00100_00000000000; //SLT rd[0]= (rs[3] < rt[4])? 1 : 0
mem[3] <= 32'b001110_00010_00011_0000000000001010; //BEQ IF(rs[2] = rt[3]), PC+=off16(10)
mem[13] <= 32'b001111_00001_00010_0000000000001010; //BNQ IF(rs[1] = rt[2]), PC+=off16(10
```

A simulação é decorrida até o tempo 130ns na Figura 8

Figura 8 – Simulação 3



Fonte: Autor

Na primeira instrução é verificado a condição (reg1<reg2) que foi verdadeira fazendo reg0 valer 1. Na segunda instrução é verificado a condição (reg3<reg4) que foi falsa fazendo reg0 valer 0. Na terceira instrução foi verificado a condição verdadeira (reg2==reg3) fazendo o PC ser acrescentado em 10, valendo então 13. Na quarta instrução, de endereço 13, foi verificado a condição falsa (reg1==reg2) fazendo o PC ser acrescentado novamente em 10, passando a valer 23.

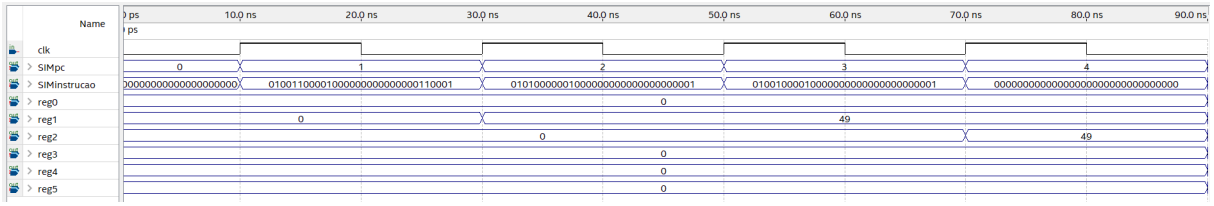
Para a quarta simulação buscaremos verificar o funcionamento efetivo de inserções e leituras na memória

Os seguintes valores para os registradores e Instruções serão inseridos na primeira subida do clock:

```
1 regs[0] <= 32'd0;
2 mem[1] <= 32'b010011_00001_00000_0000000000110001; //LOADi RD[1]=imm16(49)
3 mem[2] <= 32'b010100_00001_00000000000000000001; //STORE MEMdata[1] = Rd[1]
4 mem[3] <= 32'b010010_00010_00000000000000000001; //LOAD Rd[2]=MEMdata[1]
```

A simulação é decorrida até o tempo 90ns na Figura 9

Figura 9 – Simulação 4



Fonte: Autor

Na primeira instrução inserimos o valor imediato 49 no reg1. Na segunda instrução armazenamos esse valor no endereço 1 da memória de dados. Na terceira instrução inserimos esse valor da memória de dados novamente no banco de registrados em reg2.

Com as quatro simulações feita podemos verificar o correto funcionamento dos diferentes blocos do processador bem como as suas integrações entre si. Ao verificar que os diferentes grupos de instruções realizam de forma esperada mudanças de dados nos registradores comprovamos o funcionamento do processador.

6 Considerações Finais

Foi desenvolvido corretamente em Verilog cada um dos componentes da CPU bem como suas integrações. Foi possível verificar através de simulações em waveform o correto funcionamento do processador integralizado. Uma das principais dificuldades foi a integração entre os diferentes componentes da CPU que se comportavam de forma esperada separadamente porem em conjunto geravam interferências entre si. Outra dificuldade foi ampliar o conhecimento da linguagem Verilog, que possui características únicas a linguagens de descrição de hardware. Podemos efetuar em seguida a integração com o kit FPGA, desenvolvimento das instruções de entrada e saída de dados, como display, *resets* e entrada de *switch*. Outro ponto importante seria converter o esquemático em blocos para verilog, assim todo o processador estaria contido em arquivos de mesmo formato.

Referências

1 ALTERA. San Jose, California, USA: [s.n.], 2005. Disponível em: <https://www.intel.com/content/www/us/en/products/programmable.html>. Acesso em: 29.08.2020. Citado na página 11.

2 PATTERSON, D. A.; HENNESY, J. L. *Computer Organization and Design*. 5th edition. ed. Waltham/MA, EUA: Morgan Kaufmann, 2007. Citado 2 vezes nas páginas 7 e 12.

3 WEBER, Raul Fernando. *Fundamentos de arquitetura de computadores*. 4th edition. ed. Porto Alegre: recurso online (Livros didáticos informática UFRGS 8, 2012. WEBER, Raul Fernando. Fundamentos de arquitetura de computadores. 4. Citado na página 12.

4 PIMENTA, Tales Cleber. *Circuitos digitais : análise e síntese lógica: aplicações em FPGA*. Rio de Janeiro GEN LTC 2016, recurso online ISBN 9788595156586. Citado 2 vezes nas páginas 7 e 11.

5 Romero dos Santos, Helder Celso. *Desenvolvimento de um Sistema Computacional Composto por Processador, Memória e Interface de Comunicação em FPGA* . , São José dos Campos, 2014. Nenhuma citação no texto.