UNIVERSIDADE FEDERAL DE SÃO PAULO

BACHARELADO EM CIÊNCIA E TECNOLOGIA

CARLOS GUILHERME MORAES

140386

Desenvolvimento de um Processador RISC Implementado

São José dos Campos - Brasil

Julho de 2021

CARLOS GUILHERME MORAES

140386

Desenvolvimento de um Processador RISC Implementado

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

Julho de 2021

Lista de Ilustrações

Lista de Tabelas

Sumário

1 Introdução

A tecnologia, com o seu enorme avanço e desenvolvimento tem sido cada vez mais essencial para a vida humana, uma vez que a mesma está presente em diversos setores como indústria, saúde e locomoção, caminhando cada vez mais para a era digital e suprindo ainda mais as necessidades humanas. Em seu desenvolvimento e aplicação estão presentes os processadores que são capazes de guiá-las e recebem o título de coração da tecnologia, por ser responsável por reger todos os processos e aplicações. Para a disciplina de Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores será feita a implementação do mesmo no qual visa oferecer ao aluno experiência a fim de proporcionar oportunidades que poderão inserí-lo nesse desenvolvimento. Neste relatório será apresentado a segunda parte da implementação, tratando-se do segundo dos três pontos de checagem.

2 Objetivos

2.1 Gerais

Realizar a descrição de um processador através da implementação em *verilog* das partes a serem desenvolvidas nos Pontos de Checagem do projeto.

2.2 Específicos

Determinar elementos fundamentais para o desenvolvimento do projeto referentes ao segundo Ponto de Checagem. Implementação em verilog dos componentes referentes a unidade de processamento, assim como a adequação de instruções que precisam de certas mudanças em relação ao datapath e por fim o teste através das formas de ondas obtidas com as instruções requisitadas. Temos assim, um roteiro para os desenvolvimentos em verilog:

- Desenvolver o *Program Counter*;
- Desenvolver a *ROM*;
- Desenvolver o banco de Registradores;
- Desenvolver a *ULA*;
- Desenvolver a *RAM*;
- Desenvolver os *Mux*s;
- Desenvolver o Somador;
- Desenvolver a unidade que fará a extensão do sinal;
- Desenvolver a unidade responsável por fazer o deslocamento;
- Desenvolver o *top-level*;

3 Fundamentação Teórica

Antes de aprofundarmos nos conceitos que serão apresentados para o processador precisamos ter em mente alguns conceitos básicos e teóricos em relação a ele.

3.1 Arquiteturas

A Arquitetura de processador dita a sua descrição, ela pode ser classificada como CISC ou RISC.

A arquitetura CISC (*Complex Instruction Set Computing*), apresenta um conjunto de instruções amplo e complexo que tem um tempo de execução mais lento. Já a arquitetura RISC (*Reduced Instruction Set Computing*), apresenta poucas intruções sendo elas mais simples e com uma execução mais rápida e constante. Existem diversas estruturas que fazem uso dessas arquiteturas, para esse processador foi utilizada a MIPS (*Microprocessor without interlocked pipeline stages*) baseado na arquitetura RISC, o MIPS utiliza-se de registradores para a execução das instruções e ele pode ser monociclo, indicando que as instruções serão executadas em um ciclo de clock necessitando que o mesmo seja longo o suficiente para acomodar a instrução mais complexa ou multiciclo que divide as instruções em ciclos, ele também pode utilizar o formato little-endian que, ao contrário do formato big-endian, os bytes de menor ordem do número são armazenado na memória nos menores endereços, e ter tamanhos em bits para as intruções de 32 ou 64 bits.

Além do MIPS, existem, também, arquiteturas como x86, ARM e até mesmo processadores baseados em pilhas, cada qual com as suas vantagens e desvantagens.

3.2 Instruções

Uma instrução de um processador é o meio pelo qual é transferida a informação de uma operação para diferentes componentes do processador. Como já vimos, cada arquitetura e processador apresenta um conjunto de instruções com tamanho de 32 ou 64

bits e cada instrução está ligada a específicas operações se diferenciando em três tipos: R, I e J.

Tabela 1 - Tipos de instruções 32bits.

Tipo	32 bits					
R	opcode 6 bits	$rs 5 bits	$rt 5 bits	$rd 5 bits	Shamt 5 bits	Funct 6 bits
I	opcode 6 bits	$rs 5 bits	$rt 5 bits	Immediate 16 bits		
J	opcode 6 bits	Target 26 bits				

Fonte: O autor.

Como podemos ver, todos os três tipos apresentam opcode de 6 bits, o opcode é uma referência à instrução para ser realizada alguma operação, para instruções do tipo R, o Funct de 6 bits será responsável por indicar qual operação R será feita, além disso, as instruções do tipo R apresentam 3 registradores com tamanho de endereço de 5 bits e com papéis diferentes. O registrador $rs é o encarregado de obter o resultado da operação que será realizado entre os outros dois registradores $rt e $rd. Por fim, instruções do tipo R apresentam o Shamt, com tamanho de 5 bits, que é utilizado nas instruções de deslocamento ao invés do $rt tornando o hardware mais simples.

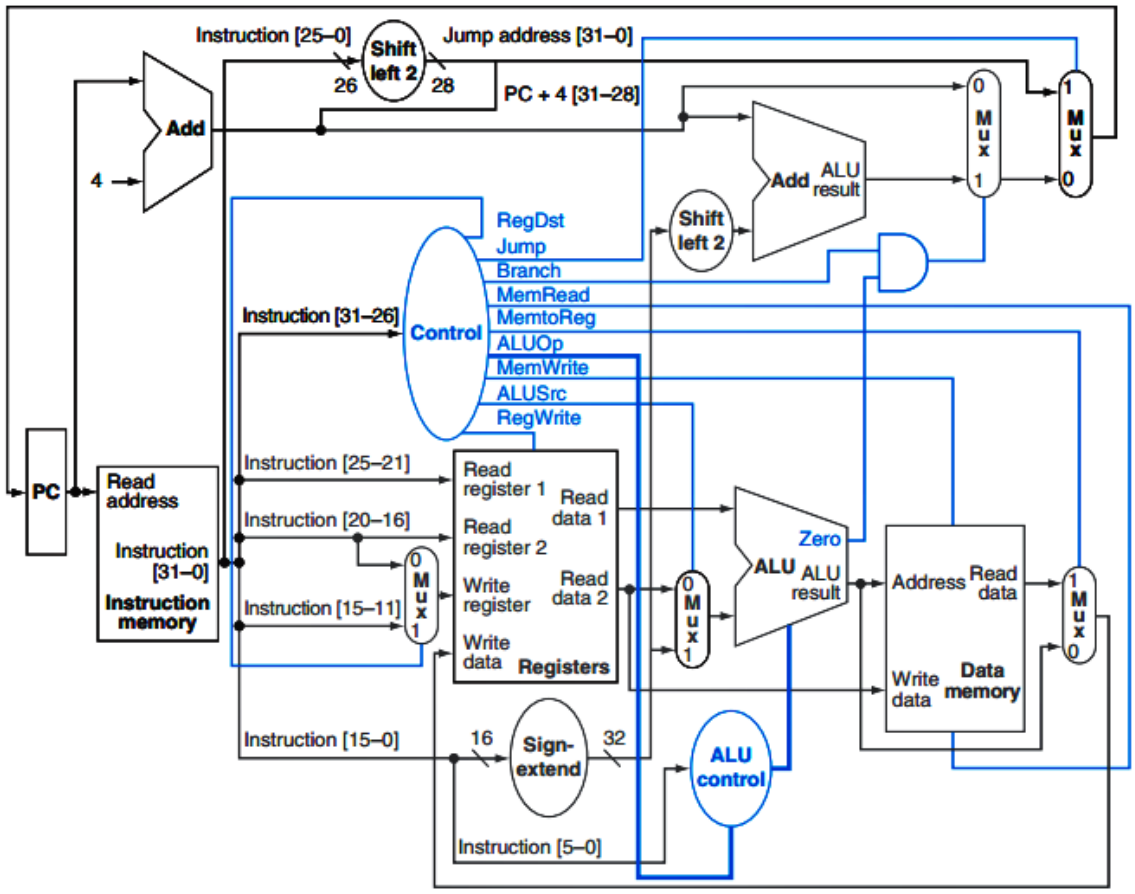
As intruções do tipo I, além do opcode, aprensentam, também, registradores, com uma unidade a menos em relação ao R, e o Immediate de 16 bits referente a um valor imediato. O valor do Immediate fará a operação com o registrador $rt e o resultado, novamente, será acomodado no registrador $rs.

Por último, temos as instruções do tipo J referente a salto, nelas estão presentes o opcode e um Target de 26 bits que representa a instrução que deseja ser realizada. Instruções do tipo J realizam um “salto” para uma outra instrução através do Target.

3.3 Caminho dos Dados

Um Caminho de Dados (*Datapath*) é uma coleção de unidades funcionais, como unidades lógicas aritméticas ou multiplicadores que executam operações de processamento de dados, registros e barramentos. Alguma das unidades funcionais básicas que podemos analisar são:

Figura 1 - MIPS Datapath.



Fonte: Executing push $reg using one instruction on single cycle datapath. [1]

- Unidade de Controle (*Control*) responsável por controlar todos os membros que compõem a unidade de processamento;
- Contador de Programa (*PC*) responsável por identificar qual a próxima instrução que

será executada;

- Memória de Instruções (*Instruction Memory*) responsável por acomodar todos endereços de instruções disponíveis da CPU;
- Banco de Registradores (*Registers*) responsável por acomodar todos os registradores disponíveis da CPU;
- Unidade Lógica Aritmética (*ALU*) responsável pelas operações aritméticas como soma e subtração e também lógicas como and, not e or;
- Memória de dados (*Data Memory*) responsável por fazer a operação de leitura e escrita de dados na memória;
- Extensão de Sinal (*Sign Extend*) responsável por estender o tamanho do imediato de 16 bits para 32 bits;
- Multiplexador (*MUX*) responsável por escolher qual entrada será executada juntamente com uma resposta da unidade de controle.
- Somadores (*ADD*), como o nome já sugere, responsável por realizar a soma das entradas..
- 2 Deslocamentos para a Esquerda (*Shift left 2*) responsável por fazer o equivalente a multiplicar por 4 a entrada.

3.4 Verilog

É uma linguagem, como VHDL, largamente usada para descrever sistemas digitais e utilizada universalmente. Inicialmente, Verilog era uma linguagem proprietária e desenvolvida pela empresa Gateway. Verilog foi desenvolvida nos anos 1980 e foi inicialmente usada para modelar dispositivos ASIC.[2] Em 1990, Verilog caiu no domínio público e agora está sendo padronizada como IEEE 1364. Temos como operadores básicos de Verilog:

Aritméticos:

+ -> Adição

- -> Subtração

* -> Multiplicação

/ -> Divisão

% -> Módulo

Lógicos:

~ -> Negação

& -> Conjunção

| -> Disjunção Inclusiva

A estrutura de um código em Verilog é da seguinte forma:

module <nome_módulo>(<lista_portas>);

<declarações>

<elementos_módulo>

endmodule

E assim, podemos, por exemplo, construir um código capaz de apresentar o modelo comportamental do AND entre dois valores de entrada:

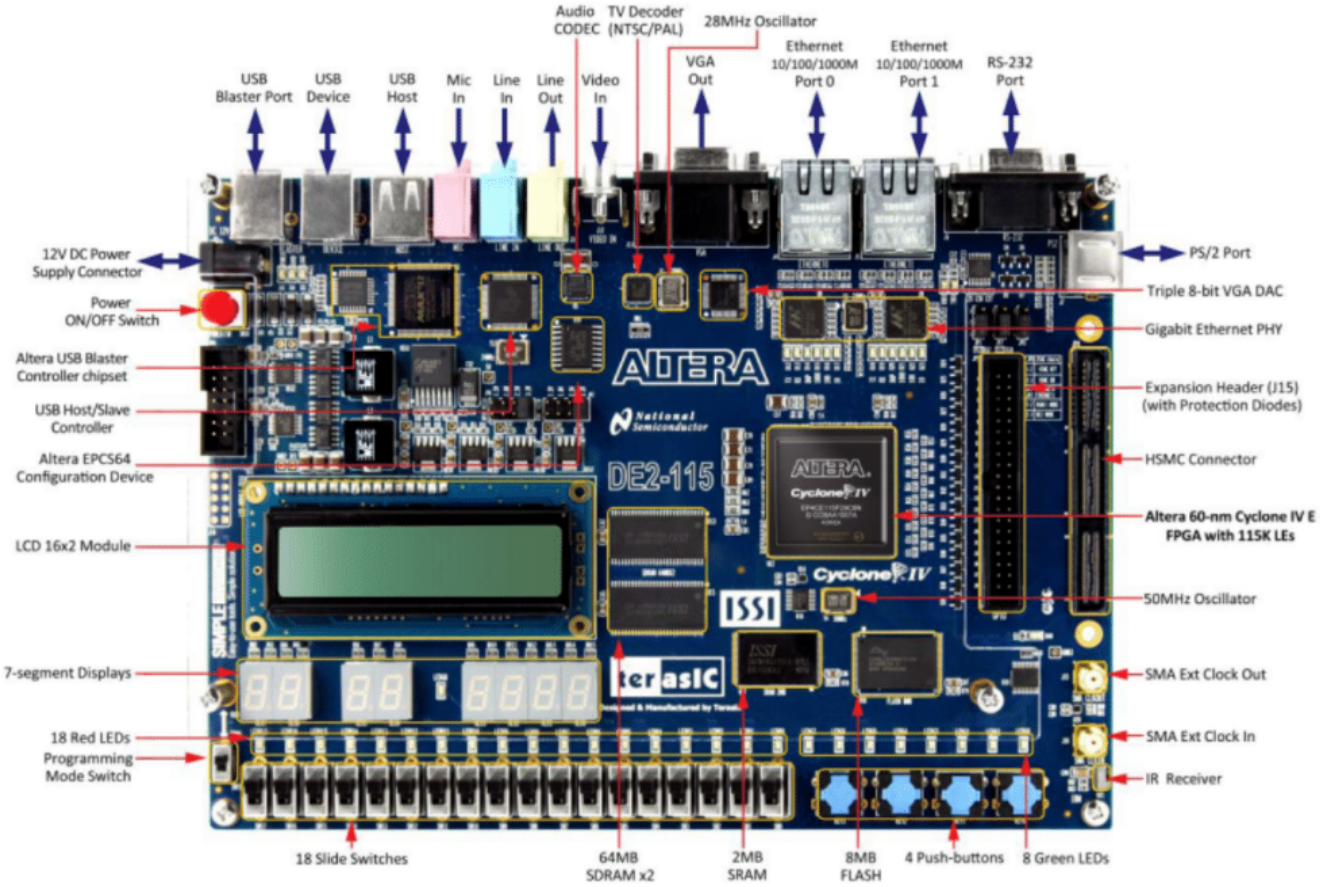
```
module AND(ini1, ini2, out);
    input ini1, ini2;
    output out;
    assign out = ini1 & ini2; //Conjunção
endmodule
```

Verilog pode ser usado para fazer diversas operações envolvendo sistemas digitais como uma simples operação lógica AND como vimos, até um processador capaz de fazer diversas operações mais complexas no qual é a finalidade desse projeto.

3.5 FPGA

O projeto será implementado utilizando um software capaz de compilar códigos em verilog chamado Quartus Prime e será simulado em um dispositivo FPGA DE2 - 115 Cyclone IV. FPGA ou field-programmable gate array (ou ainda matriz de portas programáveis) é um dispositivo lógico programável que suporta a implementação de circuitos digitais. A placa foi criada para uso em laboratórios de colégios e universidades. Ela é adequada para muitos tipos de exercícios em cursos de lógica digital e organização de computadores, com ferramentas simples que ilustram conceitos básicos para projetos avançados.[3]

Figura 2 - Recursos da Placa DE2-115.



Fonte: Altera.[4]

4 Desenvolvimento do Trabalho

Podemos agora definir os passos para o desenvolvimento do projeto, para este projeto, foi escolhido o MIPS baseado na arquitetura RISC. O MIPS será de 32 bits, monociclo utilizando o formato little-endian com 19 instruções disponíveis.

O MIPS fará operações com registradores e imediatos, para isso, dentro das 19 instruções que serão utilizadas, haverá instruções do tipo R, I e J, as do tipo R obterão os valores digitados pelo programador através dos registradores $rt e $rd, como já vimos, as instruções do tipo I, terá um dos valores acomodado ao registrador $rt, o outro valor será inserido diretamente pelo programador sem necessitar de um registrador e esse será o imediato já mencionado algumas vezes. As instruções do tipo J usará do Target mencionado caso sejam executadas.

As 19 instruções que serão utilizadas são:

Tabela 2 - Conjunto de Instruções.

Instrução	**Formato**		**Instrução**	**Formato**
Operações aritméticas			**Transferência de dados**	
1. add $rs, $rt, $rd	R		**12.** lw $rs, Immediate	I
2. addi $rs, $rt, Immediate	I		**13.** sw $rs, Immediate	I
3. sub $rs, $rt, $rd	R			
4. subi $rs, $rt, Immediate	I		**Operações de salto condicional**	
5. mult $hi/$lo, $rt, $rd	R		**14.** beq $rs, $rt, Immediate	I
6. div $hi/$lo, $rt, $rd	R		**15.** bne $rs, $rt, Immediate	I
7. sll $rs, Shamt	R		**16.** bgt $rs, $rt, Immediate	I

8.	srl $rs, Shamt	R		**17.** blt $rs, $rt, Immediate	I
Operações lógicas				**Operações de salto incondicional**	
9.	and $rs, $rt, $rd	R		**18.** j Immediate	J
10.	or $rs, $rt, $rd	R		**19.** jal Immediate	J
11.	not $rs, $rt	I			

Fonte: O autor.

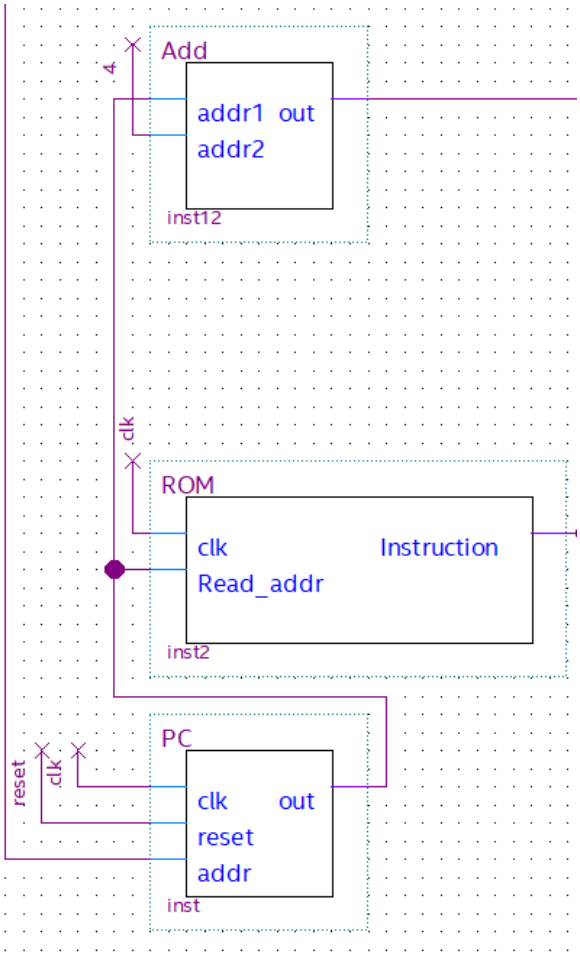
Com as instruções definidadas é possível, então, definir os modos de enderaçamento das instruções de tipo R, I e J. Para isso, usaremos a seguinte tabela:

Tabela 3 - Modo de endereçamento das instruções.

Opcode	Funct	Instrução		Opcode	Instrução
Tipo R				**Tipo I**	
000000	100100	and		001000	addi
	100101	or		001010	subi
	100000	add		001100	not
	100010	sub		100011	lw
	011000	mult		101011	sw
	011010	div		000100	beq
	000000	sll		000101	bne
	000001	srl		000110	blt
				000111	bgt
Tipo J					
000010	X	j			
000011	X	jal			

4.1 Desenvolver o *Program Counter*

Figura 3 - Esquemático *PC.*



Fonte: O autor.

Com as instruções e o modo de enderaçamento em mãos, será feito então a implementação em verilog das unidades que compõem a unidade de processamento de um processador, para isso, iniciaremos, desta forma, pelo PC cuja tarefa é identificar a instrução que será executada.

Figura 4 - Implementação em Verilog do *Program Counter.*

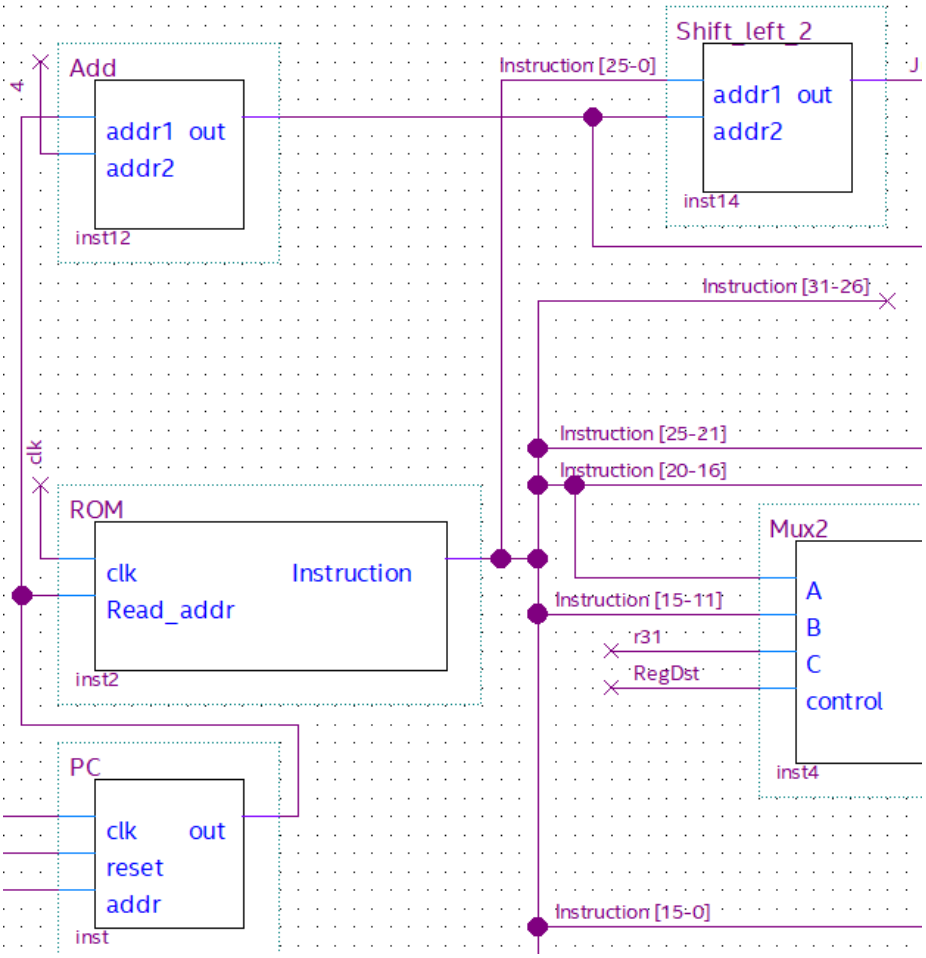
```
module PC
#(parameter ADDR_WIDTH=5)
(
   input clk, reset,
   input [2**ADDR_WIDTH-1:0] addr,
   output reg [2**ADDR_WIDTH-1:0] out
);

   always @ (posedge clk, negedge reset)
   begin
      if (!reset) out <= 0;
      else out <= addr;
   end
endmodule
```

Fonte: O autor.

O PC, apresentado em verilog, recebe o valor de *clock*, representado pelo *input clk*, no qual dita quando uma operação deve ser feita juntamente com o *reset*. O *addr* de 32 bits é responsável por obter o endereço de memória da instrução a ser realizada e o *out*, de 32 bits também, é responsável por passar esse endereço adiante. Quando o *clock* estiver em curva de subida ou o *reset* em curva de descida, o bloco representado pelo *always* é acionado e então será analisado se o *reset* está em nível baixo, caso esteja, o *out* passará adiante o valor inicial da memória reiniciando todo o processo, caso não esteja, o *out* receberá o valor do endereço de memória da instrução atual no qual seguirá para a ROM.

4.2 Desenvolver a *ROM*

Figura 5 - Esquemático *ROM.*



Fonte: O autor.

Em seguida, como a saída do PC passará adiante o endereço de memória da instrução, precisamos implementar a unidade que será responsável por gerir o banco de instruções, a fim de, com o endereço de memória, identificar a instrução desejada. Para essa unidade temos o código em verilog:

Figura 6 - Implementação em Verilog da *ROM.*

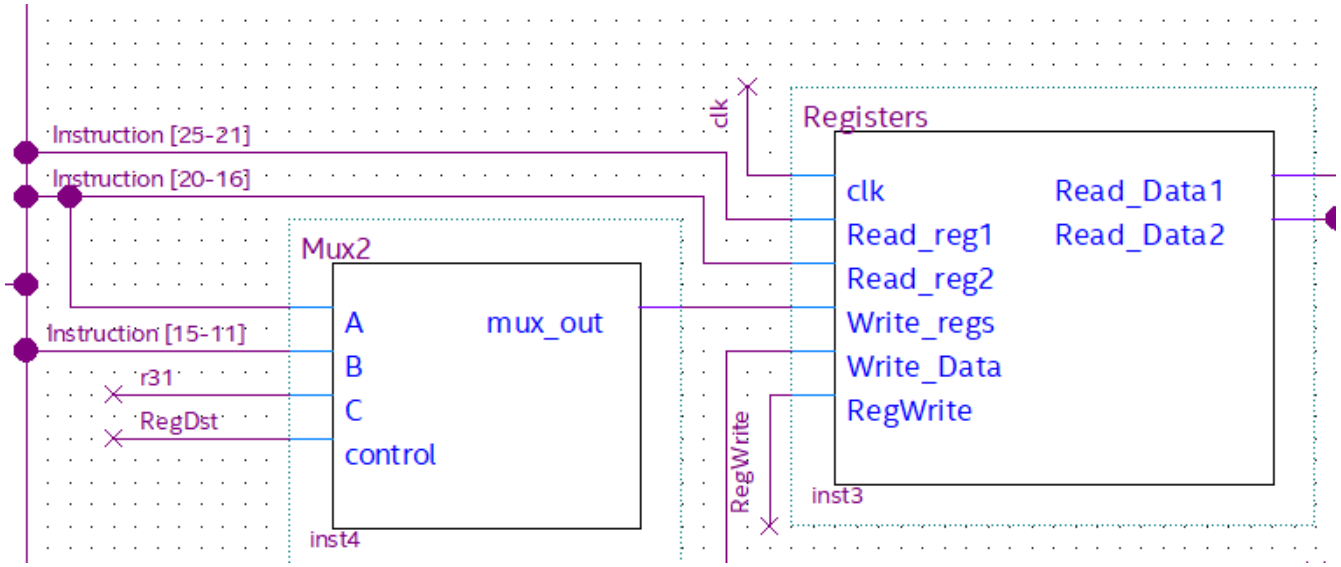
```
module ROM
#(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=5)
(
   input clk,
   input [(ADDR_WIDTH-1):0] Read_addr,
   output reg [(DATA_WIDTH-1):0] Instruction
);

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

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

   always @ (posedge clk)
   begin
      Instruction <= rom[Read_addr];
   end

endmodule
```

Fonte: O autor.

Novamente, como visto na implementação do PC, a *ROM* recebe o *clock* a fim de indicar quando deve acontecer a operação (curva de subida), também recebe o endereço de memória da instrução representado por *Read_addr*, porém este apresenta somente 5 bits indicados pela posição de 2 a 6 do endereço de saída do PC, isso acontece pois os dois bits menos significativos do endereçamento não é tão importante e os próximos 5 bits são os mais importantes para o endereçamento, e a *ROM* também apresenta um *reg* de saída que é própriamente a instrução de 32 bits referenciado por *Instruction*. Além das variáveis de entrada e saída, também temos um outro *reg rom* com tamanho de 32 bits por 32 bits cuja finalidade é armazenar todas as instruções que são lidas de um arquivo txt. Com isso, após a curva de subida do *clock*, o *Instruction* recebe a instrução que está contida em *rom* na posição referenciada pelo endereço de memória, *Read_addr.* Assim, *Instruction* passará adiante a instrução buscada.

4.3 Desenvolver o banco de Registradores

Figura 7 - Esquemático banco de registradores.



Fonte: O autor.

Com a instrução em mãos, devemos, agora, implementar o banco de registradores a fim de localizar qual registrador será utilizado através dos endereços de memória que a instrução carrega. Assim, temos a implementação:

Figura 8 - Implementação em Verilog do banco de Registradores.

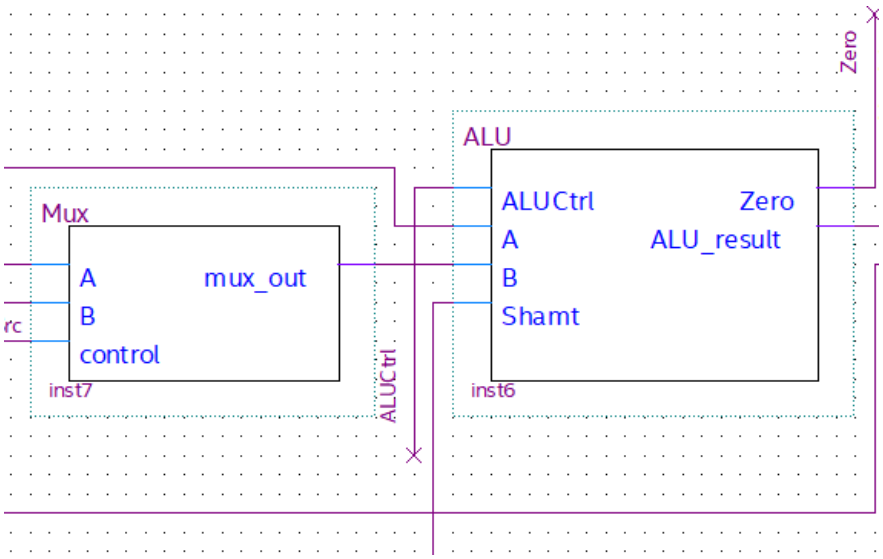
```
module Registers
#(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=5)
(
 input clk,
 input [(ADDR_WIDTH-1):0] Read_reg1, Read_reg2, Write_regs,
 input [(DATA_WIDTH-1):0] Write_Data,
 output [(DATA_WIDTH-1):0] Read_Data1, Read_Data2,
 input RegWrite
);

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

 assign Read_Data1 = registers[Read_reg1];
 assign Read_Data2 = registers[Read_reg2];

 always @(posedge clk)
 begin
    registers[Write_regs] <= (RegWrite) ? Write_Data:registers[Write_regs];
 end
endmodule
```

Fonte: O autor.

O banco também fará a operação a partir da curva de subida do *clock*, nele contém três *inputs* referentes à memória dos registradores, o primeiro *Read_reg1* contém o endereço do registrador *$rs*, o *Read_reg2* contém o endereço do registrador *$rt* e o *Write_regs* contém o endereço ou do *$rt* ou do *$rd*, quem decide é um *Mux,* juntamente com uma *flag* da unidade de controle e o *opcode*, que terá sua implementação apresentada mais à frente. Todos os três *inputs* apresentam 5 bits. Além disso, também se encontra um *input Write_Data* que obterá o valor de uma operação realizada e apresenta 32 bits, dois *outputs* de 32 bits, *Read_Data1* e *Read_Data2*, que carregarão o valor contido dentro dos registradores buscados e um último *input RegWrite* referenciado por uma *flag* da unidade de controle que indicará se será feita a escrita em um registrador. Com isso, é criado um *reg registers* de 32 bits por 32 bits que, semelhante ao *rom* já visto, servirá de armazenamento dos valores dos registradores e, assim, o *Read_Data1* recebe o valor contido no endereço de memória *Read_reg1* e o *Read_Data2* recebe o valor contido no endereço de memória *Read_reg2*. Após a curva de subida do *clock*, se o *RegWrite* estiver em nível alto o valor de *Write_Data* será atribuído ao endereço *Write_regs* caso contrário, nada será feito mais.

4.4 Desenvolver a ULA

Figura 9 - Esquemático *ULA*.



Fonte: O autor.

Agora, com os valores obtidos dos registradores armazenados em *Read_Data1* e *Read_Data2* podemos implementar a *ULA* para ser feita algumas operações. Desta forma temos:

Figura 10 - Implementação em Verilog da *ULA*.

```
module ALU
#(parameter CONTROL_WIDTH=4, parameter DATA_WIDTH=32)
(
 input [(CONTROL_WIDTH-1):0] ALUCtrl,
 input [4:0] Shamt,
 input [(DATA_WIDTH-1):0] A, B,
 output Zero,
 output reg [(DATA_WIDTH-1):0] ALU_result
);
 reg signed [(DATA_WIDTH*2)-1:0] HiLo;

 initial
  begin
   HiLo = 0;
  end

 always @ (A, B, ALUCtrl)
 begin
    case(ALUCtrl)
     4'd0: ALU_result <= A&B;            //and
     4'd1: ALU_result <= A|B;            //or
     4'd2: ALU_result <= A+B;            //add
     4'd3: ALU_result <= A-B;            //sub
     4'd4: ALU_result <= A<B ? 1:0;      //blt
     4'd5: ALU_result <= A>B ? 1:0;      //bgt
     4'd6: ALU_result <= ~A;             //not
     4'd7: HiLo <= A*B;                  //mult
     4'd9: ALU_result <= A << Shamt;     //sll
     4'd10: ALU_result <= A >> Shamt;    //srl
     4'd8:                               //div
     begin
      HiLo[31:0] <= A / B;
      HiLo[63:32] <= A % B;
     end
    default: ALU_result <= 0;
    endcase
 end

 assign Zero = (ALU_result==0);
endmodule
```

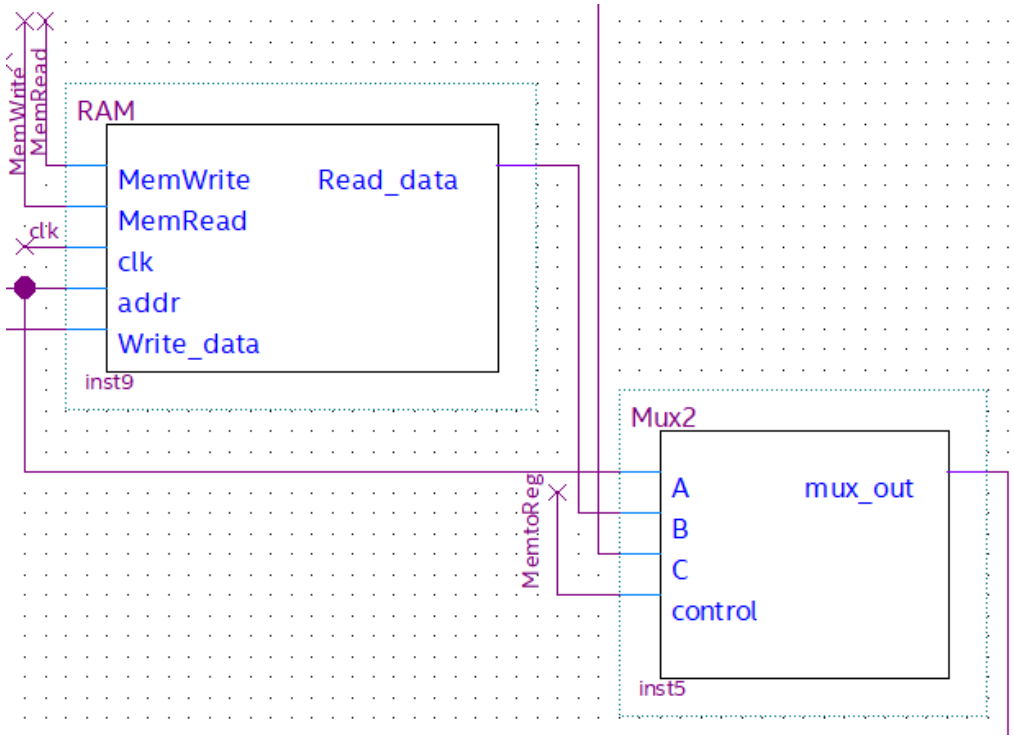
Fonte: O autor.

Diferente das outras unidades já apresentadas, a *ULA* não obtém o valor de entrada do *clock*, por não haver necessidade, assim, as entradas são: *ALUCtrl* de 4 bits, referenciado por uma *flag* da unidade de controle da *ULA* no qual indica qual operação será feita, o *Shamt* de 5 bits que carrega o valor do deslocamento para instruções como *sll* ou *srl*, *A* e *B* de 32 bits que são os valores nos quais estão relacionados com a operação. Já as saídas são: *Zero* sendo uma *flag* indicada para a realização de instruções *Branch* onde, quando 0,

indicará que os valores são iguais para *beq*, caso contrário os valores não são iguais, e um *ALU_result* de 32 bits que carregará o resultado da instrução. Em seguida se encontra um bloco *always* que é acionado quando *A, B* ou *ALUCtrl* apresentar uma variação e dentro do bloco, *ALU_result* receberá o valor da operação desejada através do *ALUCtrl*. Com isso, caso o *ALU_result* seja 0, o *output Zero* receberá o valor de 1 para a utilização de instruções *Branch* já mecionadas.

4.5 Desenvolver a *RAM*

Figura 11 - Esquemático *RAM.*



Fonte: O autor.

Com a saída da *ULA*, temos então a implementação da *RAM* para operações que exigem uma leitura ou escrita da memória.

Figura 12 - Implementação em Verilog da *RAM.*

```
module RAM
#(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=5)
(
   input clk,
   input [(ADDR_WIDTH-1):0] addr,
   input [(DATA_WIDTH-1):0] Write_data,
   input MemWrite, MemRead,
   output [(DATA_WIDTH-1):0] Read_data
);

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

   always @ (posedge clk)
   begin
      if (MemWrite) ram[addr] <= Write_data;
   end

   assign Read_data = ram[addr];
endmodule
```

Fonte: O autor.

A *RAM*, novamente, terá acionamento através da curva de subida do *clock*, além disso, há 4 *inputs*, sendo *addr* de 32 bits o endereço de memória recebido da saída da *ULA*, *Write_data* de 32 bits também referente ao valor de *Read_Data2*, saída do banco de registradores, *MemWrite* e *MemRead* referentes às *flags* da unidade de controle que servem de indicadores para leitura ou escrita na *RAM*. A saída será o *Read_data* de 32 bits que carregará o valor, em instrução como *lw*, até o banco de registradores para ser escrito em um registrador. Assim como a *ROM* e o banco de registradores, a *RAM* apresenta um *reg ram* de 32 bits por 32 bits encarrecado de armazenar o dado ou atribuí-lo. No bloco *always* após a curva de subida do *clock* a *reg ram* receberá o valor de *Write_data* no endereço referente ao *addr* caso *MemWrite* indique que a operação é referente a uma instrução de escrita na memória, como o *sw*, caso contrário a *reg ram* no endereço *addr* será atribuído ao *Read_data.* Como o *Mux*, em seguida, indica qual valor irá para o banco de registradores, *Read_data* sempre receberá o valor caso a *flag MemWrite* não esteja em nível alto.

4.6 Desenvolver os Muxs

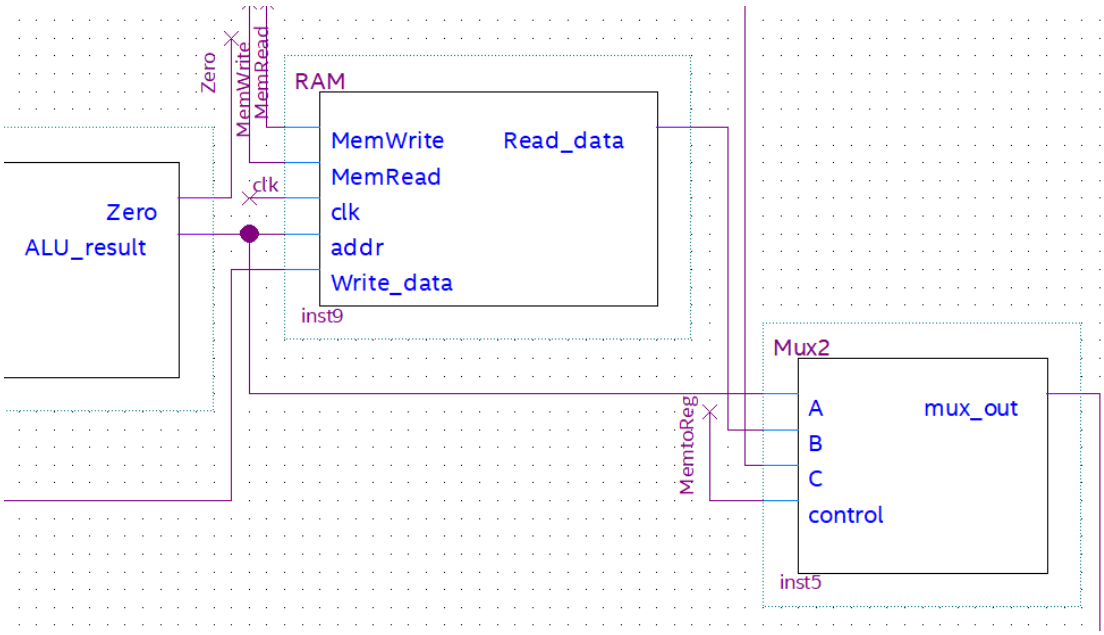
A esse nível, as principais unidades já foram implementadas, dasta forma, podemos fazer a implementação dos *Mux*s. No projeto há 5 *Mux*s, diferenciando-se em 2, um para 2 entradas, sem contar as *flags* de controle, e o outro para 3, e diferenciando-se entre eles pelo tamanho das entradas e saída. Como a última implementação vista foi da *RAM*, vejamos então a implementação do *Mux* que vem a seguir:

Figura 13 - Implementação em Verilog do *Mux* de 3 entradas.

```
module Mux2_N
#(parameter N=32)
(
 input [N-1:0] A, B, C,
 input [1:0] control,
 output [N-1:0] mux_out
);

 assign mux_out = (control==2'b00) ? A: (control==2'b01) ? B:C;
endmodule
```

Fonte: O autor.

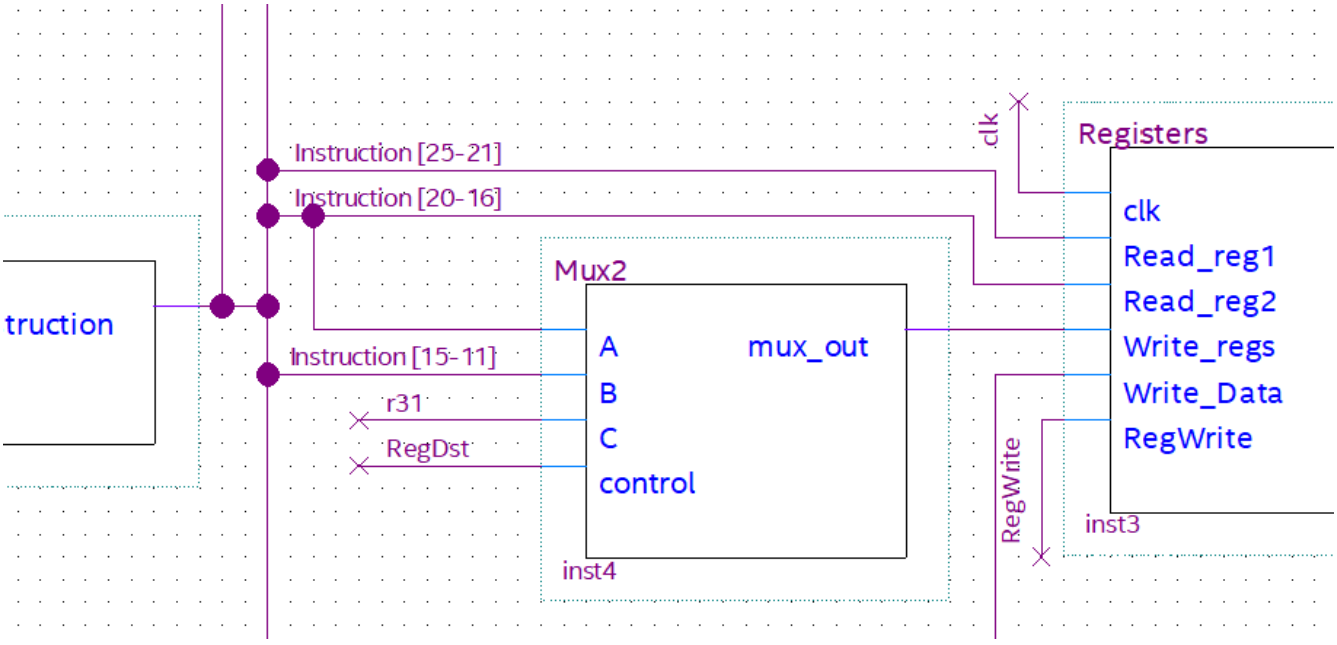
Figura 14 - Esquemático *Mux* de 3 entradas 1.



Fonte: O autor.

Neste *Mux*, é passado como parametro ao declarar a *blackbox* no *top-level*, o valor de 32 que indica a quantidade de bits das entradas, com exceção da *flag* de controle, e da saída. As entradas *A*, *B* e *C* receberão os valores de *ALU_result*, saída da *ULA*, *Read_data*, saída da *RAM* e um *out*, saída de um somador que veremos mais à frente. O *control* de 2 bits é referente à *flag* de controle da unidade de controle, *MemtoReg*, e *mux_out* é a saída que carrega o valor escolhido das entradas. Caso *control* seja 0, o valor de *ALU_result* será atribuído à saída indicando que não foi necessário a utilização da *RAM*, geralmente acontece em instruções do tipo R, caso seja 1 o valor de *Read_data* será atribuído à saída indicando uma instrução *lw*, como já vimos e por último, caso seja 2 o valor da saída do somador, *out*, será atríbuído à saída como *Mux* indicando a instrução *jal*.

Figura 15 - Esquemático *Mux* de 3 entradas 2.



Fonte: O autor.

O outro *Mux* que também apresenta 3 entradas, com exceção da *flag* de controle, e que se diferencia do anterior somente em tamanho de bits das entradas e saídas é o *Mux* que se encontra entre a *ROM* e o banco de registradores, neste, será atribuído o parametro de 5 bits para o tamanho de entradas e saída e as entradas serão o endereço de memória

do registrador *$rt*, o endereço de memória do registrador *$rd* e o valor 31, 11111, referente à um registrador *$r31*. O *control*, neste caso, será controlado pela *flag RegDst*, e caso seja 0, a saída obterá o valor do endereço de *$rt*, indicando que o mesmo receberá o valor de escrita de *Write_Data* no banco de registradores, caso seja 1, o endereço de *$rd* que será atribuído e o mesmo que receberá o valor de *Write_Data*, caso seja 2, o valor 31 será atribuído a saída e o registrador *$r31* receberá o valor de *Write_Data* indicando uma instrução *jal.*

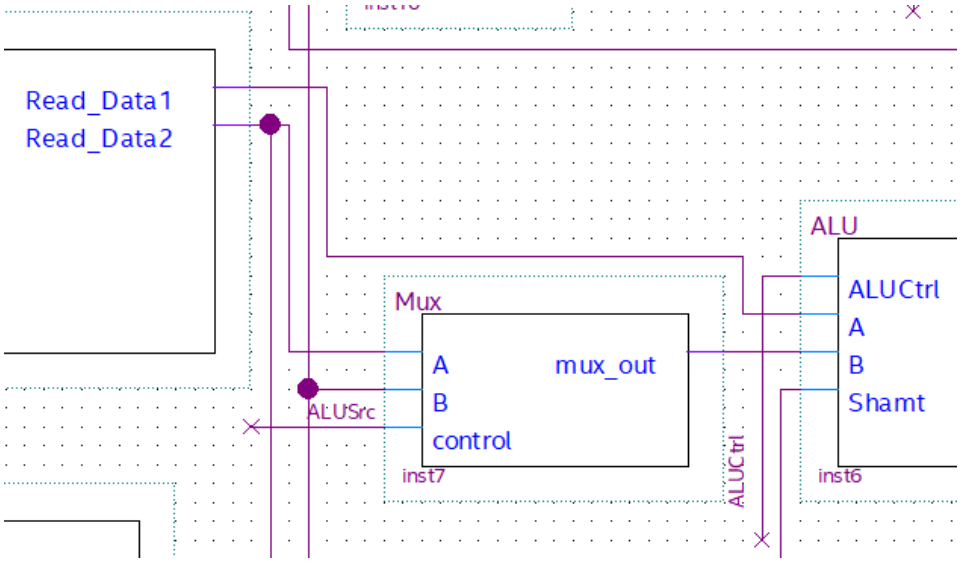
Os outros 3 *Muxs* apresentam 2 entradas somentes e se diferenciam somente no tamanho de bits. Vejamos a implementação deles:

Figura 16 - Implementação em Verilog do *Mux* de 2 entradas.

```
module Mux_N
#(parameter N=32)
(
 input [N-1:0] A, B,
 input control,
 output [N-1:0] mux_out
);

 assign mux_out = (control==0) ? A:B;
endmodule
```

Fonte: O autor.

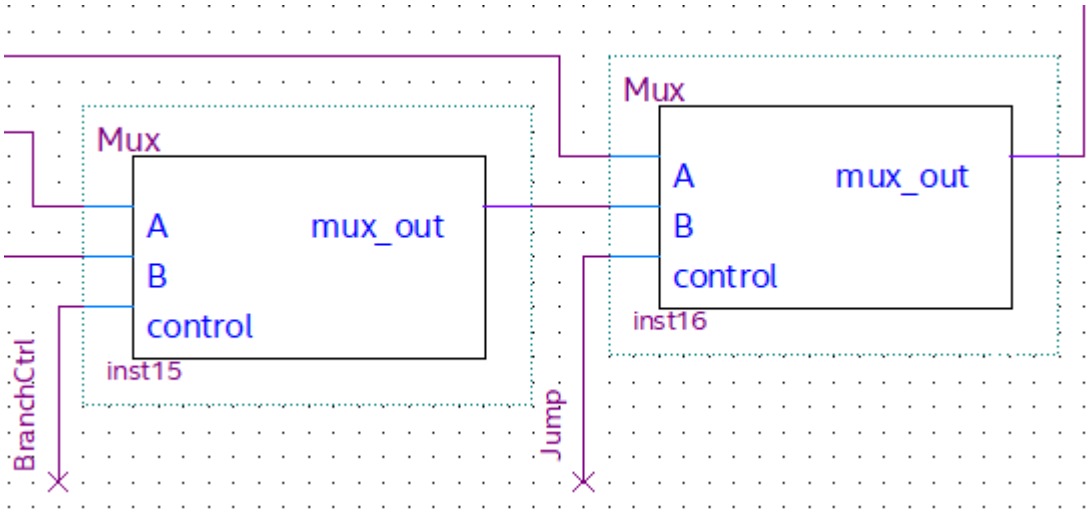
Figura 17 - Esquemático *Mux* de 2 entradas 1.



Fonte: O autor.

Para este tipo de *Mux*, temos como exemplo, o *Mux* entre o banco de registradores e a *ULA*, este *Mux* receberá como entrada, *Read_Data2*, saída do banco de registradores, e o *addr* referente à saída da unidade que fará extensão do sinal no qual veremos mais à frente, a *flag* de controle atribuída a essa *Mux* é a *ALUSrc* e caso seja 0, a saída do *Mux* terá como atribuição o valor de *Read_Data2*, indicando uma instrução do tipo R, por exemplo, e caso seja 1, o valor de saída será da saída da unidade responsável por estender o sinal de entrada, indicando uma instrução do tipo I, por exemplo.

Figura 18 - Esquemático *Mux* de 2 entradas 2 e 3.



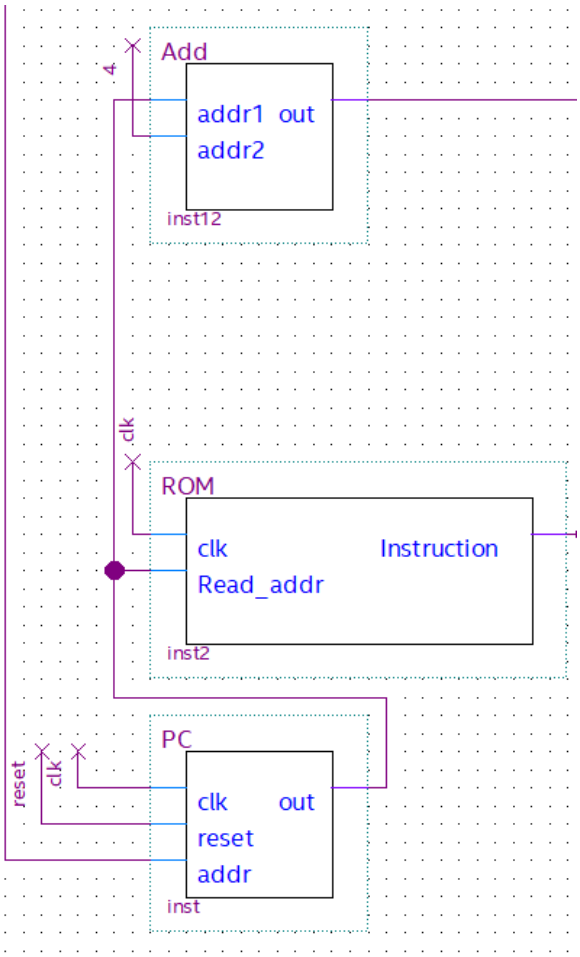
Fonte: O autor.

Os outros dois *Muxs* restantes também apresentam 2 entradas e têm o mesmo tamanho de bits, 32, o primeiro, recebe como entrada o *out* referente à saída do somador que soma 4 ao endereço da instrução, e um outro *out* referente a um outro somador que faz a soma do endereço mais 4 juntamente com o sinal estendido e delocado dois bits, a *flag* de controle é acionada quando há uma operação do tipo *Branch* e caso seja 0, o valor de saída será o endereço mais 4 indicando a próxima instrução que deverá ser seguida e caso seja 1, será a soma do endereço mais 4 juntamente com o sinal estendido e deslocado dois bits indicando que o *Branch* é verdadeiro saltando para uma instrução que não, necessariamente, é a seguida.

O último *Mux* está em seguida do anterior e tem como entradas o endereço da próxima instrução em seguida e a saída do *Mux* anterior. A *flag* de controle é a *Jump* e caso seja 0, a saída será o resultado do *Mux* anterior indicando que pode ter havido *Branch* ou não e caso seja 1, a saída será o endereço da instrução que virá em seguida.

4.7 Desenvolver o Somador

Figura 19 - Esquemático Somador.



Fonte: O autor.

A unidade de Somador nada mais fará do que somar as duas entradas e atribuí-las à saída.

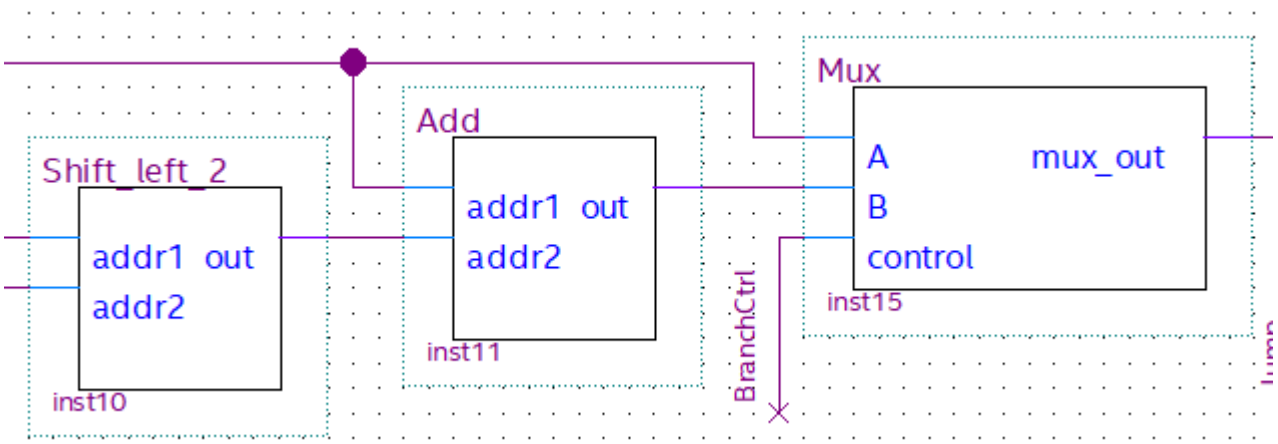
Figura 20 - Implementação em Verilog do *Somador.*

```
module Add
#(parameter DATA_WIDTH=32)
(
 input [(DATA_WIDTH-1):0] addr1, addr2,
 output [(DATA_WIDTH-1):0] out
);

 assign out = addr1 + addr2;
endmodule
```

Fonte: O autor.

O primeiro Somador reberá a instrução e o valor 4 como entrada e terá como saída a sua soma indicado como o endereço para a próxima instrução.

Figura 21 - Esquemático Somador 2.

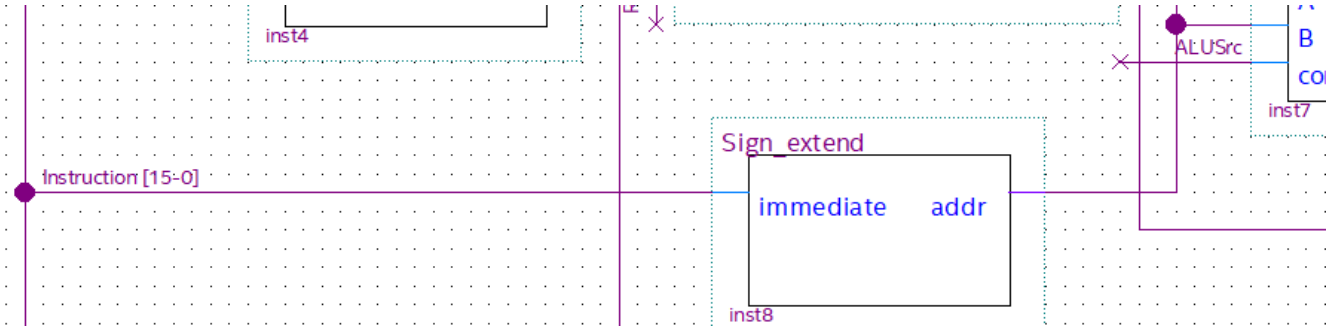


Fonte: O autor.

Já o segundo Somador, terá como entrada a saída do outro Somador e o sinal estendido deslocado 2 bits, a saída será a soma indicando uma operação que necessite pular para uma instrução que não necessariamente está em seguida.

4.8 Desenvolver a unidade que fará a extensão do sinal

Figura 22 - Esquemático unidade responsável por fazer a extensão do sinal.



Fonte: O autor.

Para instruções que precisam de um target ou immediate, o valor, inicialmente, apresenta 16 bits e precisa ser estendido para 32 bits para que se prossiga a execução, assim, a unidade que faz a extensão do sinal se torna necessária.

Figura 23 - Implementação em Verilog da unidade responsável por fazer a extensão do sinal.

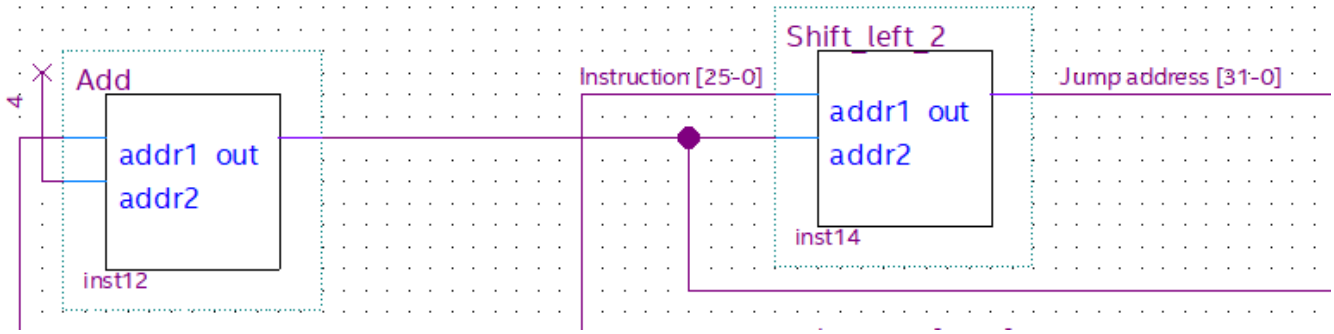
```
module Sign_extend
#(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=16)
(
 input [(ADDR_WIDTH-1):0] immediate,
 output reg [(DATA_WIDTH-1):0] addr
);

 always @(immediate)
 begin

  if(immediate[(ADDR_WIDTH-1)]==1)
   addr[(DATA_WIDTH-1):ADDR_WIDTH] = 16'b1111111111111111;
  else
   addr[(DATA_WIDTH-1):ADDR_WIDTH] = 16'b0000000000000000;

  addr[(ADDR_WIDTH-1):0] = immediate[(ADDR_WIDTH-1):0];
 end
endmodule
```

Fonte: O autor.

A unidade receberá o valor do target ou imediato de 16 bits e a saída será o sinal estendido para 32 bits. O bloco *always* será ativado quando o imediato tiver alguma variação e, assim, a saída será o sinal, por fim, estendido.

4.9 Desenvolver a unidade responsável por fazer o deslocamento

Figura 24 - Esquemático unidade responsável por fazer o deslocamento de 2 bits 1.



Fonte: O autor.

Em seguida temos a unidade responsável por fazer o deslocamento do endereço, justamente pois o valor do endereçamento mais importante é do intervalo de 2 a 6 como já mencionado, assim, o deslocamento é feito em 2 bits para o bit menos significativo estar na posição 2.

Figura 25 - Implementação em Verilog da unidade responsável por fazer o deslocamento.

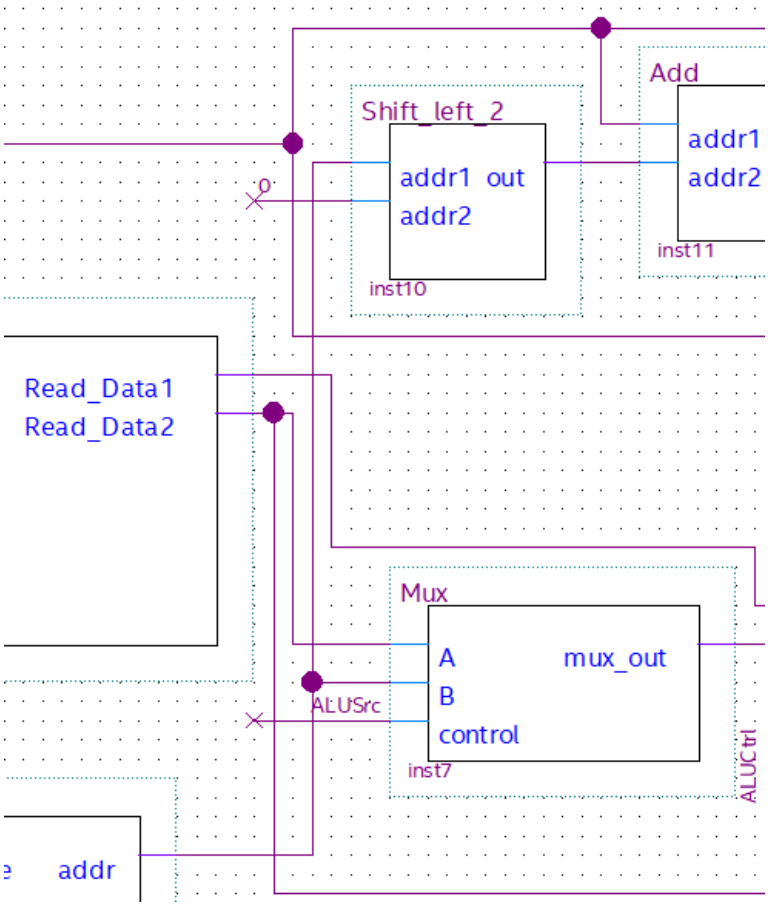
```
module Shift_left_2
#(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=1)
(
 input [(DATA_WIDTH-1):0] addr1,
 input [(ADDR_WIDTH-1):0] addr2,
 output reg [31:0] out
);

 always @(addr1, addr2)
 begin

  out <= addr1 << 2'd2;
  out[31:(32-ADDR_WIDTH)] <= (ADDR_WIDTH != 1) ? addr2[(ADDR_WIDTH-1):0]:out[31:(32-ADDR_WIDTH)];

 end
endmodule
```

Fonte: O autor.

Para a primeira unidade a entrada será os 26 primeiros bits da instrução representado pelo *addr1* e os 4 últimos bits da saída do primeiro somador que faz a soma do endereço

mais 4 representado pelo *addr2*, a saída, por sua vez, é um *reg* de 32 bits com o sinal estendido contido nele. O bloco *always* é ativado quando há variação em *addr1* ou *addr2* e assim, a saída recebe o valor deslocado 2 bits, nesta unidade em específica, além do deslocamento, o valor dos 4 últimos bits da saída do somador é substituído dos 4 últimos bits do valor deslocado.

Figura 26 - Esquemático unidade responsável por fazer o deslocamento de 2 bits 2.



Fonte: O autor.

A segunda unidade tem como entrada o sinal estendido, saída de 32 bits da unidade responsável por fazer a extensão do sinal, e a saída será justamente o sinal deslocado 2 bits somente.

4.10 Desenvolver o top-level

Por fim, temos então a implementação do *top-level* significando a junção das unidades apresentadas em forma de blackbox.

Figura 27 - Implementação em Verilog do *top-level* parte 1.

```
module CPU
(
    input CLK, reset,
    output Clock,
    output wire [31:0] ALU_result,Read_Data1,
    output wire [31:0] Read_Data2,Address_in,
    output wire [31:0] Address_out,Address_4,
    output wire [31:0] Instruction,RAM_Read_Data,
    output wire [31:0] Instruction_Left
);
    wire Jump,Branch,Bne,ALUSrc,RegWrite,MemWrite,
    wire MemRead,Zero,BranchCtrl,BEQ_out,BNE_out;
    wire [1:0] RegDst,MemtoReg;
    wire [2:0] ALUop;
    wire [3:0] ALUCtrl;
    wire [4:0] r31;
    wire [5:0] Write_register;
    wire [31:0] Write_Data,Sign,Sign_Left,ALU_Mux,
    wire [31:0] ALU_result_Add,Branch_or_normal;

    assign r31 = 5'b11111;

    Clock #(0) Unit0(CLK,
                     reset,
                     Clock);
    PC Unit1(Clock,
             reset,
             Address_in,
             Address_out);
    Add Unit2(Address_out,
              32'd4,
              Address_4);
    ROM Unit3(Clock,
              Address_out[6:2],
              Instruction);
    Mux2_N #(5) Unit4(Instruction[20:16],
                      Instruction[15:11],
                      r31,
                      RegDst,
                      Write_register);
    Registers Unit5(Clock,
                    Instruction[25:21],
                    Instruction[20:16],
                    Write_register,
                    Write_Data,
                    Read_Data1,
                    Read_Data2,
                    RegWrite);
    Sign_extend Unit6(Instruction[15:0],
                      Sign);
```

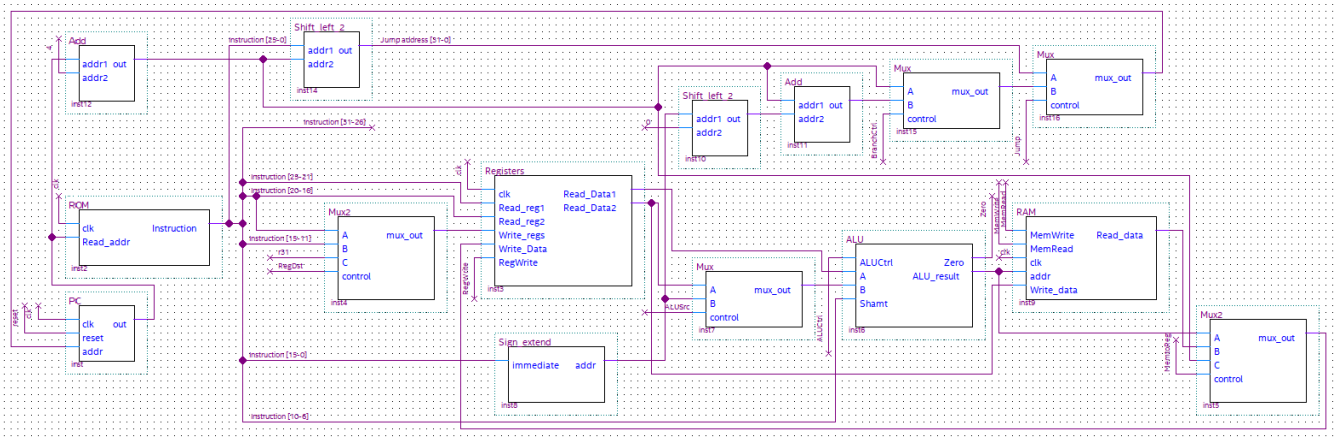
Figura 28 - Implementação em Verilog do *top-level* parte 2.

```
    Mux_N #(32) Unit7(Read_Data2,
                      Sign,
                      ALUSrc,
                      ALU_Mux);
    ALU Unit8(ALUCtrl,
              Instruction[10:6],
              Read_Data1,
              ALU_Mux,
              Zero,
              ALU_result);
    RAM Unit9(Clock,
              ALU_result[6:2],
              Read_Data2,
              MemWrite,
              MemRead,
              RAM_Read_Data);
    Mux2_N #(32) Unit10(ALU_result,
                        RAM_Read_Data,
                        Address_4,
                        MemtoReg,
                        Write_Data);
    Shift_left_2 Unit11(Sign,
                        1'b0,
                        Sign_Left);
    Add Unit12(Address_4,
               Sign_Left,
               ALU_result_Add);
    and(BEQ_out,
       Branch,
       Zero);
    and(BNE_out,
       Bne,
       ~Zero);
    or(BranchCtrl,
       BEQ_out,
       BNE_out);
    Mux_N #(32) Unit13(Address_4,
                       ALU_result_Add,
                       BranchCtrl,
                       Branch_or_normal);
    Shift_left_2 #(26,4) Unit14(Instruction[25:0],
                                Address_4[31:28],
                                Instruction_Left);
    Mux_N #(32) Unit15(Branch_or_normal,
                       Instruction_Left,
                       Jump,
                       Address_in);

endmodule
```

Fonte: O autor.

E também a apresentação do esquemático de todo o projeto já criado até o momento juntamente com todas as ligações feitas.

Figura 29 - Esquemático *top-level*.



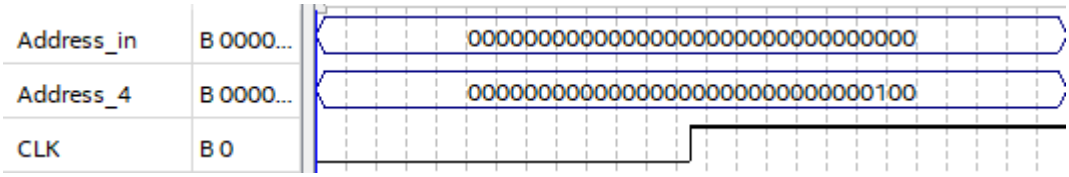
Fonte: O autor.

5 Resultados e Discussões

Depois de todas as implementações em verilog, podemos analisar as formas de onda dos componentes.

Começando pela unidade do *PC*, para este, foi analisada a forma de onda do mesmo juntamente com o somador.

Figura 30 - Forma de onda do *PC* junto com o Somador.

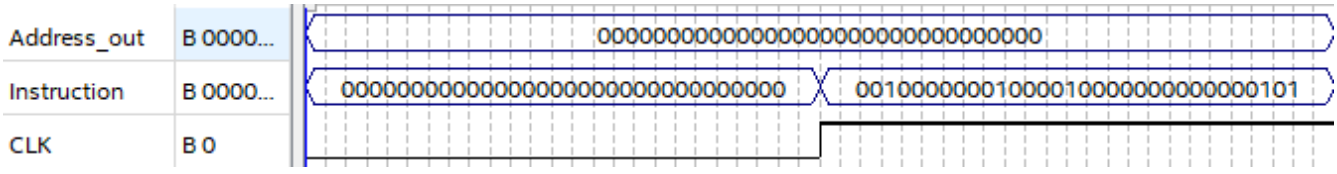


Fonte: O autor.

Os valores binários identificados são, respectivamente, o endereço de memória que entra na unidade do *PC* e o endereço de memória que sai do somador, cujo objetivo é fazer a soma de mais 4 ao endereço. Como foi o primeiro endereço a ser visto a forma de onda, logo o endereço analisado foi o 0 com 32 bits, e assim, ao passar pelo somador, a saída deveria ser exatamente o valor 4 com 32 bits representado pelo *Address_4*.

Em seguida, foi obtida a forma de onda da *ROM*, a mesma deveria buscar a instrução a partir do endereço informado.

Figura 31 - Forma de onda da *ROM*.

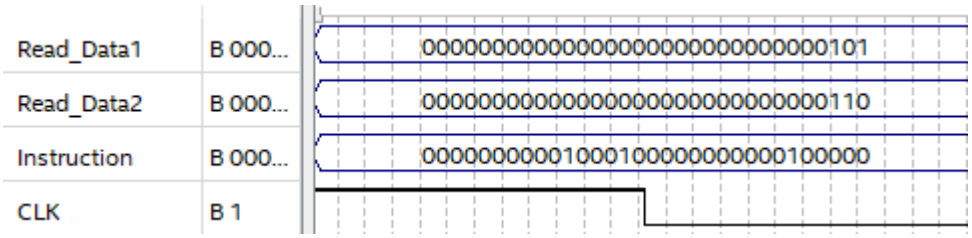


Fonte: O autor.

Novamente, o endereço passado foi exatamente o 0 com 32 bits, logo, a partir do pulso de *clock*, a *ROM* encontrou a instrução localizada na primeira posição da memória.

Adiante, foi obtida a forma de onda do banco de registradores, porém, para este, precisou ser feito a inserção de dois valores (5 e 6) em dois registradores cujos endereços de memória foram 1 e 2 respectivamente.

Figura 32 - Forma de onda do banco de registradores.

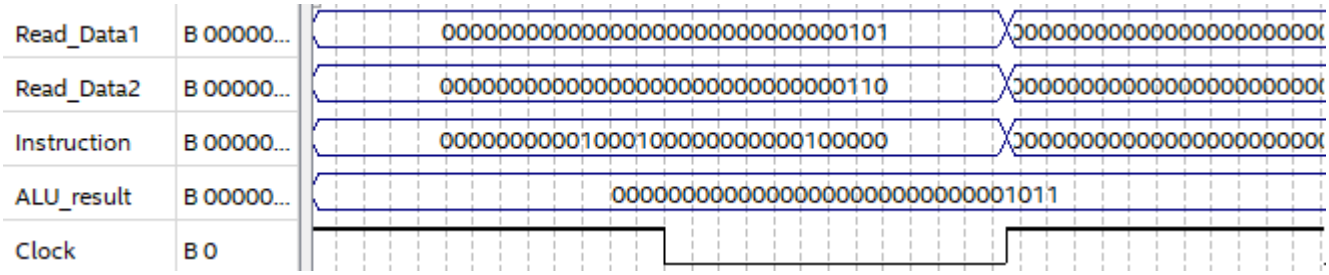


Fonte: O autor.

Desta forma, com a instrução representando uma operação do tipo R, mais especificamente um *add* com endereços de registradores 1, 2 e 0, foi possível ver a saída do banco de registradores, representados por *Read_Data1* e *Read_Data2*, após o pulso de *clock* para o nível alto e que eles continham os valores 5 e 6, respectivamente.

Para a *ULA*, foram pegas as saídas do banco de registradores, que posteriormente foram as entradas da *ULA*, e verificando se estavam, de fato, acontecendo a soma orientada pela instrução.

Figura 33 - Forma de onda da *ULA*.

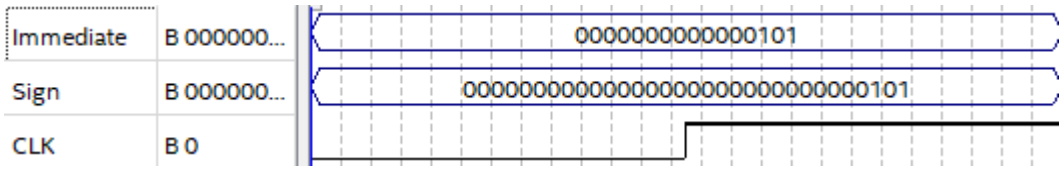


Fonte: O autor.

Como as entradas foram os valores 5 e 6 respectivamente e a instrução foi a *add*, logo o resultado obtido deveria ser a soma de ambos os valores respresentado pelo valor 11 em binário do *ALU_result*.

Prosseguindo, foi obtida, também, a forma de onda da unidade responsável por fazer a extensão do sinal, a entrada consistiu em um *Immediate* e a saída representada pelo *Sign*.

Figura 34 - Forma de onda da unidade responsável por fazer a extensão do sinal.

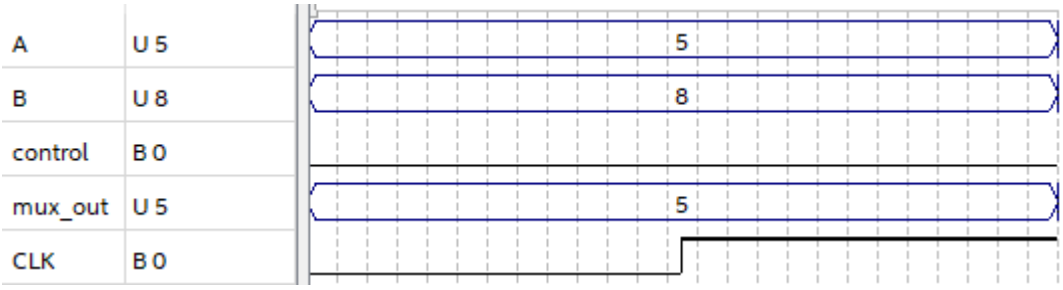


Fonte: O autor.

Podemos ver que na entrada o *Immediate* continha o valor 5 com somente 16 bits e ao final do processo o *Sign* passou a ter o valor 5 com 32 bits, como o esperado.

Também foram obtidas as formas de onda dos *Muxs*, para o primeiro foi utilizado o *Mux* de 2 entradas, com exceção da *flag* de controle, as entradas foram representadas por A, B e o *control* e a saída representada *mux_out*.

Figura 35 - Forma de onda do *Mux* de 2 entradas.

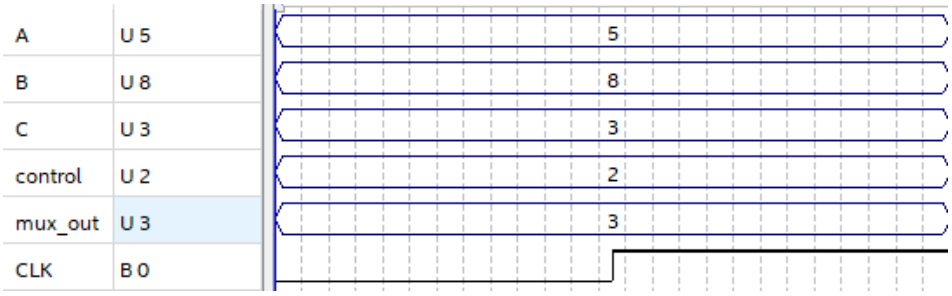


Fonte: O autor.

No *Mux*, quando a *flag* de controle está em nível baixo, o valor escolhido é o contido em A, os valores de A e B foram 5 e 8 respectivamente e como podemos ver analisando o *mux_out*, o valor escolhido foi o 5.

Para o *Mux* de 3 entradas, a *flag* terá 2 bits como já vimos, logo terá mais opção de escolha, as entradas para esse *Mux* foram A, B, C e *control*.

Figura 36 - Forma de onda do *Mux* de 3 entradas.

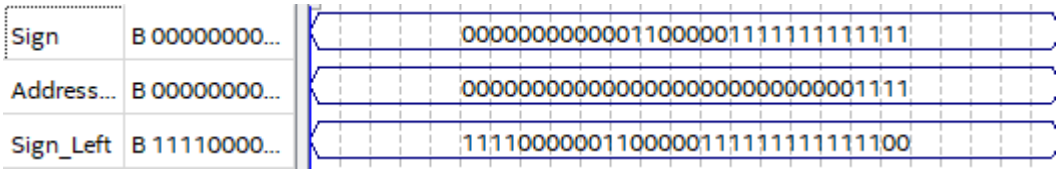


Fonte: O autor.

Os valores para as entradas A, B, C e *control* foram 5, 8, 3 e 2, logo, para o valor 2 da *flag*, o valor escolhido pelo *Mux* deveria ser o contido em C como podemos observar pelo *mux_out.*

A unidade responsável por fazer o deslocamento de 2 bits também foi analisada, ambos os dois casos. Para o primeiro, consideramos o caso onde o mesmo recebe os endereços onde o segundo é o endereço de saída do *PC* mais 4, nessa situação, além de haver o deslocamento do primeiro endereço, o 4 primeiros bits do segunda deve ser substituídos nos 4 últimos bits do endereço de saída.

Figura 37 - Forma de onda da unidade responsável por fazer o deslocamento de 2 bits 1.

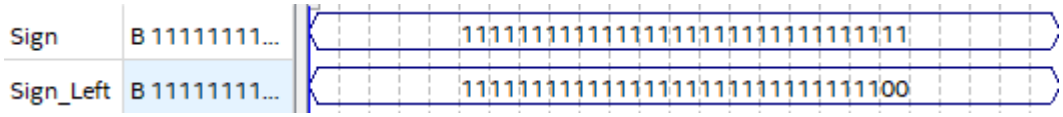


Fonte: O autor.

O endereço do *PC+4* apresenta 4 bits de valor 1 menos significativos e a saída, representada pelo *Sign_Left,* é semelhante ao primeiro endereço de entrada deslocados 2 bits e, além disso, os 4 bits mais significativos apresentam os 4 bits de valor 1 do *PC+4.*

Já a segunda unidade, como só consideramos somente uma entrada, então o único propósito é realizar o deslocamento.

Figura 38 - Forma de onda da unidade responsável por fazer o deslocamento de 2 bits 2.

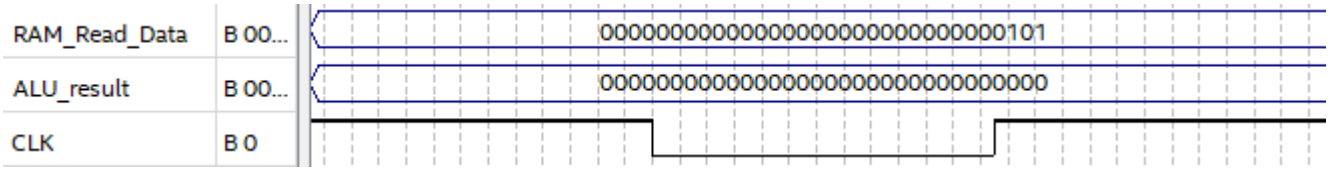


Fonte: O autor.

Como a entrada foram 32 bits de valores 1, a saída resultou em 30 bits com valores 1 mais significativos e 2 bits de valores 0 menos significativos indicando que foram deslocados como esperado.

Por último, foi obtido, também, a forma de onda da *RAM*, porém, semelhante à forma de onda do banco de registradores, foi necessário, antes da simulação, adicionar um valor para que pudesse ser visualizado a operação de busca na memória a partir do endereço de memória inserido.

Figura 39 - Forma de onda da *RAM*.



Fonte: O autor.

O valor representado pelo *ALU_result* foi a entrada, e como ele está todo zerado, o intuito foi buscar o valor 5 inserido na posição inicial da memória e, com o pulso de *clock* inicialmente em nível alto indicando que já houve um pulso, foi obtido na saída, *RAM_Read_Data*, o valor 5 em binário com 32 bits.

6 Considerações Finais

Em suma, o projeto ainda deve ter a implementação da unidade de controle, assim, podem haver modificações ao longo da implementação do projeto caso haja a necessidade. As estapas seguintes deverão ser, então, a implementação da unidade de controle, a elaboração de todas as instruções e a finalização do projeto. O projeto, até o momento, obteve os resultados esperados, porém, é inegável que foi investido muito tempo para que o mesmo chegasse a esse nível visto que o tempo de compilação do software utilizado, Quartus, foi o maior empecilho. Com a implementação do segundo ponto de chegacem, mais da metade do projeto já foi concluída, com isso, o restante para a finalização do projeto não deverá apresentar maiores dificuldades.

7 Referências

[1] Executing push $reg using one instruction on single cycle datapath. Acessado em junho de 2021 <https://electronics.stackexchange.com/questions/139307/executing-push-reg-using-one-instruction-on-single-cycle-datapath/139317>

[2] ALEXANDRO, DANIEL. DELGADO, RENIÊ. OGG, VANESSA. Sistemas Digitais - Linguagem Verilog. Acessado em julho de 2021 <https://www.cin.ufpe.br/~voo/sd/Aula6#:~:text=Verilog%20%C3%A9%20uma%20linguagem%2C%20como,sistemas%20digitais%20e%20utilizada%20universalmente%3B&text=Verilog%20foi%20desenvolvida%20nos%20anos,sendo%20padronizada%20como%20IEEE%201364.>

[3] ALTERA.This is a test entry of type @ONLINE. San Jose, California, USA: [s.n.], 2005.Disponível em:<http://www.altera.com/>. Acessado em julho de 2021.