Relatório Semana 12 a 13 - ARQ1

1st Enzo Dezem Alves - 801743
Departamento de Computação
Universidade Federal de São Carlos
São Carlos, Brazil
enzodezem@estudante.ufscar.br

2nd Vinícius Marto da Veiga - 821252
Departamento de Computação
Universidade Federal de São Carlos
São Carlos, Brasil
viniciusveiga@estudante.ufscar.br

I. Especificação e Objetivos do Problema

A. Proposta dos Experimentos/Exercícios Propostos

A proposta envolve a criação de novas instruções no processador monociclo, como segue a imagem:

Versão 2 – acrescentar:			
		sll	bne
		slli	blt
		srl	bge
		srli	
		andi	
		xor	
		xori	

Fig. 1. Intruções novas

Devemos:

- Implementar as Instruções da Figura 1
- Testar cada instrução individualmente
- Fazer dois programas com 15 instruções cada testando as instruções

B. Objetivos em Termos de Aprendizado do Aluno

Os principais objetivos de aprendizado desses experimentos são:

- **Aplicar a teoria**: Aplicar a Teoria aprendida no problema proposto mostrando que aprendemos como funciona o processador
- **Melhorar Habilidades no Verilog**: Desenvolver habilidades práticas na implementação de circuitos lógicos em Verilog.
- **Melhora na Compreensão do Processador**: Melhorar o entendimento no funcionamento do processador Monociclo.

II. Metodologia

Os experimentos foram conduzidos utilizando o Quartus como ferramenta de design. As principais etapas foram:

A. Implementação das Intruções

mudança na lógica de Verilog para implementação das lógicas novas, as mudanças foram feitas nos Modulos Ula, ULADec, MainDec, e controller.

B. Criação de Arquivos .WVF para Teste Isolado

Elaboração de arquivos .WVF para testar cada módulo individualmente. Verificação dos resultados esperados.

C. Teste dos Modulos com Instruções

Foi feito dois programas com 15 instruções ou mais cada e testado no novo processador.

III. Experimentos Práticos

- **Implementação**: Começamos alterando a ula.sv , apenas reutilizando os bits de controle não utilizados ainda e criando um sinal novo para o controller:

```
module alu(input  logic [31:0] a, b,
           input  logic [2:0]  alucontrol,
           output logic [31:0] result,
           output logic        Zero,
           output logic        Menor);

  logic [31:0] condinvb; // pega o b, se alucontrol[0] for 0 e pega o b~ se for 1
  logic [31:0] sum;      //  aqui é o resultado de a + b + o bit de carry in
  logic        v;        // overflow
  logic        isAddSub;     // true when is add or subtract operation

  assign condinvb = alucontrol[0] ? ~b : b;
  assign sum = a + condinvb + alucontrol[0];
  assign isAddSub = ~alucontrol[2] & ~alucontrol[1] | //testando se foi soma ou sub
                    ~alucontrol[1] & alucontrol[0];

  always_comb
    case (alucontrol)
      3'b000:  result = sum;         // add
      3'b001:  result = sum;         // subtract
      3'b010:  result = a & b;       // and
      3'b011:  result = a | b;       // or
      3'b101:  result = sum[31] ^ v; // slt , pega o ultimo bit para ver se deu negativo(A - B , A<B), s
      3'b100:  result = a << b;   // sll
      3'b110:  result = a >> b;   // srl
      3'b111:  result = a ^ b;    //xor
      default: result = 32'bx;
    endcase

  assign Zero = (result == 32'b0); //se todos os bits derem 0 o sinal zero sai como 1.
  assign Menor = (sum[31] ^ v);  //1 se A < B
  assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) & isAddSub; //verifica se é overflow.

endmodule
```

Fig. 2. Ula.sv

O novo sinal menor, é apenas 1 quando A for menor do que B, sendo bastante util para comparação das instruções novas do tipo B, as instruções sll, srl, e xor foram implementadas sem dificudades(Implementadas na Figura 2).

Apos isso fomos para a implementação na ALUDec, que tirou o sinal do Branch do MainDec e assumiu essa responsabilidade de diferenciar os sinais para cada Branch. utilizamos um sinal de 4 bits para o Branch , sendo o Branch = 0000 quando não é uma instrução do Tipo-B, o Branch = 0001 quando for **beq** , Branch = 0010 quando for **blt**, Branch = 0100 quando for **bne** e Branch

= 1000 quando for **bge**, utilizamos essa metodologia de cada bit para cada instrução apenas para facilitar o trabalho e a didatica de mostrar como foi feito, poderia ter sido feito com 3 bits. Cada Branch é enviado quando o Opcode é de Tipo-B e a func3 diferencia eles. Alem da implementação das instruções Branch foi implementado as instruções novas sll, srl e xor em ALUDec, como mostra a figura 3 e 4:

```
always_comb
  case(ALUOp)
    2'b00:
          begin
             ALUControl = 3'b000; // addition for lw,sw
             Branch = 4'b0000;
          end
    2'b01:
             case(funct3)
                3'b000:
                      begin
                        ALUControl = 3'b001;
                        Branch = 4'b001; // subtraction for beq
                      end
                3'b100:
                      begin
                        ALUControl = 3'b001;
                        Branch = 4'b0010; //sub for blt
                      end
                3'b001:
                      begin
                        ALUControl = 3'b001;
                        Branch = 4'b0100;  //sub for bne
                      end
                3'b101:
                      begin
                        ALUControl = 3'b001;
                        Branch = 4'b1000; // sub for bge
                      end
               default:
                  begin
                     ALUControl = 3'bxxx;
                     Branch = 3'bxxx;
                  end
             endcase
```

Fig. 3. ALUDec implementação

```
             endcase
      default: case(funct3) // R-type or I-type ALU
                 3'b000:  if (RtypeSub)
                            begin ALUControl = 3'b001; // sub
                            Branch = 4'b0000; end
                          else
                            begin ALUControl = 3'b000; // add, addi
                            Branch = 4'b0000; end
                 3'b010:    begin ALUControl = 3'b101; // slt, slti
                            Branch = 4'b0000; end
                 3'b110:    begin ALUControl = 3'b011; // or, ori
                            Branch = 4'b0000; end
                 3'b111:    begin ALUControl = 3'b010; // and, andi
                            Branch = 4'b0000; end
                 3'b001:    begin ALUControl = 3'b100; //sll ,slli
                            Branch = 4'b0000; end
                 3'b101:    begin ALUControl = 3'b110; //srl , srli
                            Branch = 4'b0000; end
                 3'b100:    begin ALUControl = 3'b111; //xor , xori
                            Branch = 4'b0000; end
                 default:   begin ALUControl = 3'bxxx;
                            Branch = 4'bxxxx; end   // ???
               endcase
    endcase
endmodule
```

Fig. 4. ALUDec implementação

Depois faltou apenas implementar a lógica combinacional para ativar o PCSrc de maneira correta, deixamos assim:

```
module controller(input  logic [6:0] op,
                  input  logic [2:0] funct3,
                  input  logic       funct7b5,
                  input  logic       Zero,
                  input  logic       menor,
                  output logic [1:0] ResultSrc,
                  output logic       MemWrite,
                  output logic       PCSrc, ALUSrc,
                  output logic       RegWrite, Jump,
                  output logic [1:0] ImmSrc,
                  output logic [2:0] ALUControl,
                  output logic [3:0] Branch2,
                  output logic [1:0] ALUOp2,
                  output logic       Bne,
                  output logic       Bge,
                  output logic       Blt,
                  output logic       Beq);

  logic [1:0] ALUOp;
  logic [3:0] Branch;

  assign Branch2 = Branch;
  assign ALUOp2 = ALUOp;

  assign Bne = (Branch[2] & ~Zero);
  assign Bge = ((~menor | Zero) & Branch[3]);
  assign Blt = (Branch[1] && menor);
  assign Beq = (Branch[0] & Zero);

  maindec md(op, ResultSrc, MemWrite,ALUSrc, RegWrite, Jump, ImmSrc, ALUOp);
  aludec  ad(op[5], funct3, funct7b5, ALUOp, ALUControl,Branch);

  assign PCSrc = (((~menor | Zero) & Branch[3])||(Branch[1] & menor)||(Branch[2] & ~Zero)||(Branch[0] &
endmodule
```

Fig. 5. Controller implementação

Utilizamos o sinal Zero, Zero negado, Menor e Menor negado, sendo os mesmos vindo da ula.
Implementamos o código 1 no processador:

semano12_1.asm | riscv2.asm | risc

```
1  main:
2  addi x1 , zero , 16
3  addi x2 , zero , 4
4  srli x1,x1,1
5  beq x1,x2,done #nao tomado
6  sw x1, 0(x2) #8
7  done:
8  blt x1,x2,done2 #nao tonado
9  slli x2,x2,1 #x2=8
10 done2:
11 bge x1,x2,done3 #tomado x1=x2
12 add x1,x1,x2  #pulada
13 and x2,x2,x1  #pulada
14 andi x1,x2,4    #pulada
15 done3:
16 xor x1,x1,x2 #x1 =0
17 bne x1,x2,done4 #tomado
18 add x1,x2,x1  #pulado
19 done4:
20 addi x1,x1,-1
21 sw x1, 4(x2) #-1
22 sw x2, 8(x2)#8
```

Fig. 6. código 1 rars

```
initial
begin
  RAM[0]  = 32'h01000093;      // addi x1,x0 ,16       # x1=16 (main:)
  RAM[1]  = 32'h00400113;      // addi x2,x0,4         #x2 = 4
  RAM[2]  = 32'h0010d093;      // srli x1,x1,1         # x1 = 8
  RAM[3]  = 32'h00208463;      // beq x1,x2,done       #nao tomado
  RAM[4]  = 32'h00112023;      // sw x1, 0(x2)         #mem[4]=8
  RAM[5]  = 32'h0020c463;      // blt x1,x2,done2      #nao tonado
  RAM[6]  = 32'h00111113;      // slli x2,x2,1         #x2=8
  RAM[7]  = 32'h0020d863;      // bge x1,x2,done3      #tomado x1=x2
  RAM[8]  = 32'h0020d8b3;      // add x1,x1,x2         #pulada
  RAM[9]  = 32'h00117133;      // and x2,x2,x1         #pulada
  RAM[10] = 32'h00417093;      // andi x1,x2,4         #pulada
  RAM[11] = 32'h0020c0b3;      // xor x1,x1,x2         #x1 =0
  RAM[12] = 32'h00209463;      // bne x1,x2,done4      #tomado
  RAM[13] = 32'h001100b3;      // add x1,x2,x1         #pulado
  RAM[14] = 32'hfff08093;      // addi x1,x1,-1        # x1 = -1
  RAM[15] = 32'h00112223;      // sw x1, 4(x2)         #mem[12] = -1
  RAM[16] = 32'h00212423;      // sw x2, 8(x2)         #mem[16] = 8
end
```

Fig. 7. código 1 em Hexa na Memória

Depois implementamos esse código 2 com poucas

diferenças apenas para mostrar o funcionamento das novas instruções:

```
main:
addi x1 , zero , 8 #x1 = 8
addi x2 , zero , 4  #x2 = 4
srli x1,x1,1  # x1/2 = 4
beq x1,x2,done #tomado
sw x1, 0(x2) #pulado
done:
xori x1,x1,5 #x1 = 1
blt x1,x2,done2 #tomado
slli x2,x2,1  #pulada
add x1,x1,x2  #pulada
and x2,x2,x1  #pulada
andi x1,x2,3    #pulada
done2:
bge x1,x2,done3 #nao tomado
addi x1,x1,3 #x1=4
done3:
bne x1,x2,done4 #nao tomado x1=x2=4
add x1,x2,x1 #x1 = 8
done4:
addi x1,x1,-1 #x1 = 7
sw x1, 4(x2) # mem[8] = 7
sw x2, 8(x2)#mem[12] = 4
```

Fig. 8. código 2 rars

```
initial
begin
  RAM[0]  = 32'h00800093;      // addi x1,x0 ,8        # x1=16 (main:)
  RAM[1]  = 32'h00400113;      // addi x2,x0,4         #x2 = 4
  RAM[2]  = 32'h0010d093;      // srli x1,x1,1         # x1 = 4
  RAM[3]  = 32'h00208463;      // beq x1,x2,done       #tomado
  RAM[4]  = 32'h00112023;      // sw x1, 0(x2)         #pulado
  RAM[5]  = 32'h0050c093;      // xori x1,x1,5         #x1 = 1
  RAM[6]  = 32'h0020ca63;      // blt x1,x2,done2      #tomado
  RAM[7]  = 32'h00111113;      // slli x2,x2,1         #pulada
  RAM[8]  = 32'h002080b3;      // add x1,x1,x2         #pulada
  RAM[9]  = 32'h00117133;      // and x2,x2,x1         #pulada
  RAM[10] = 32'h00317093;      // andi x1,x2,4         #pulada
  RAM[11] = 32'h0020d463;      // bge x1,x2,done3      #nao tomado
  RAM[12] = 32'h00308093;      // addi x1,x2,3         #x1 = 4
  RAM[13] = 32'h00209463;      // bne  x1,x2,done4     #nao tomado x1=x2=4
  RAM[14] = 32'h001100b3;      // add x1,x2,x1         # x1 = 8
  RAM[15] = 32'hfff08093;      // addi x1,x1,-1        # x1 = 7
  RAM[16] = 32'h00112223;      // sw x1, 4(x2)         #mem[8] = 7
  RAM[17] = 32'h00212423;      // sw x2, 8(x2)         #mem[12] = 4
end
```

Fig. 9. código 2 em Hexa memoria

- **Verificação dos resultados por meio dos arquivos .WVF**: Após implementação das novas instruções verificamos seu funcionamento por meio dos testes de onda .WVF, onde pode ser averiguado o correto funcionamento das novas instruções. segue imagens:

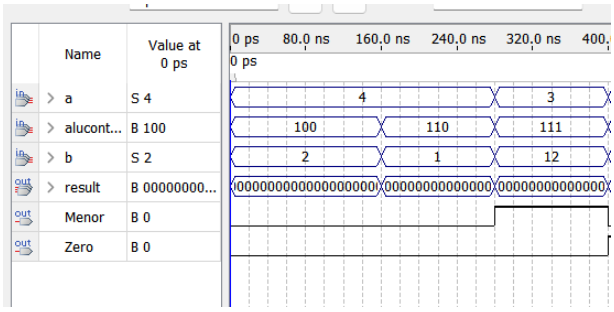

Fig. 10. WVF ula

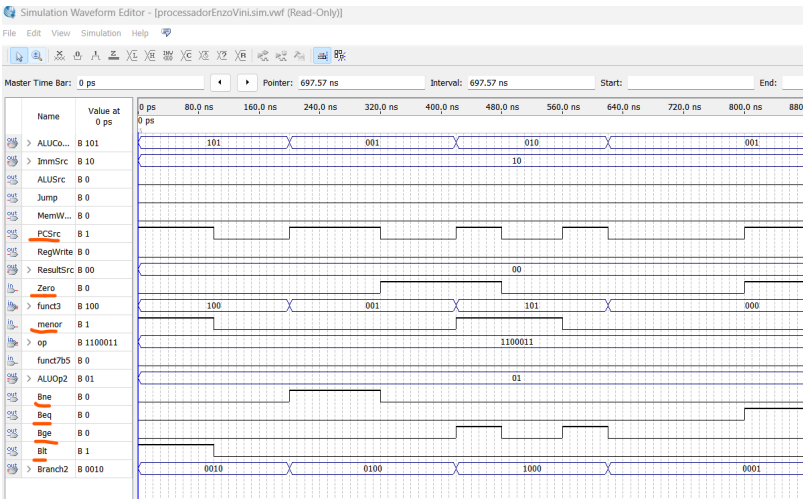

Fig. 11. WVF controler

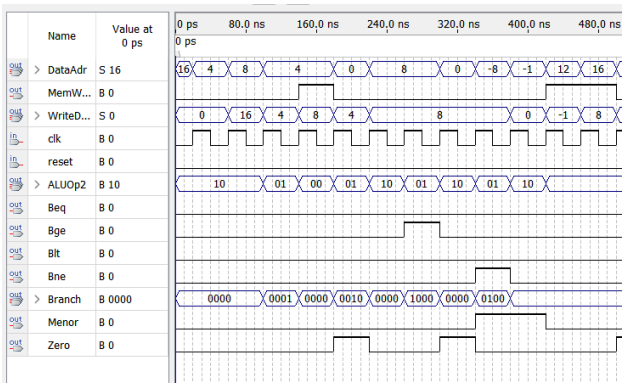

Fig. 12. código teste 1 rars

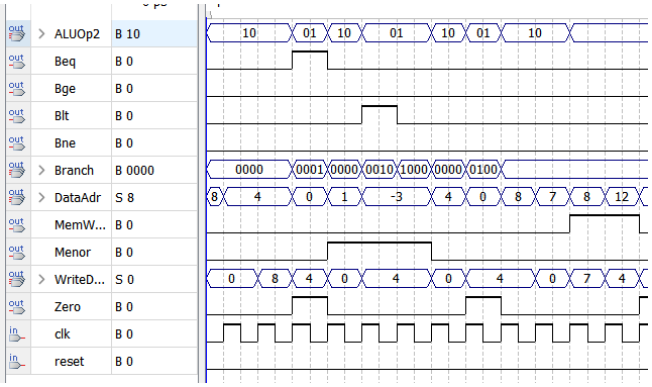

Fig. 13. código teste 2 rars

IV. Conclução

Avaliação dos Objetivos de Aprendizagem:

- Aprendemos melhor a Sintaxe do Verilog e a lógica para implementar mais instruções.
- Também adquirimos conhecimentos sobre a utilização do simulador para verificar os resultados e sobre a criação de projetos no Quartus. Como a propagação de variaveis

para averiguarmos o funcionamento da implementação

Dificuldades Encontradas:

- Sintaxe do Verilog é meio desagradavel algumas vezes.
- Averiguar os resultados com tantos modulos e variaveis.

Sugestões para Melhorias:

- não consegui pensar em nenhuma.

V. LINK DO VIDEO

LINK DO VIDEO

VI. BIBLIOGRAFIA

Brock J. LaMeres: Quick Start Guide to Verilog, Springer, 2019. https://zyedidia.github.io/notes/sv$_guide.pdf$