Trabalho prático 3 - Organização de computadores

Janderson Glauber Mendes dos Santos - 2020054544
Kleber Junior Alves Pereira - 2020054625
Laura Godinho Barroso - 2020425100

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) - Belo Horizonte - MG-Brasil.

1.INTRODUÇÃO

O intuito deste trabalho é familiarizar os envolvidos com a Linguagem de Descrição de HardWare Verilog juntamente com os conceitos da matéria de Organização de Computadores aprendidos em sala. Para a execução desse TP será usado o Google colab para executar as etapas exigidas em um arquivo .ipynb que possui a implementação do RISC-V em Verilog.

Neste trabalho temos como objetivo alterar o caminho de dados fornecidos a fim de incluir mais operações e módulos. Serão implementados os arquivos Verilog e os códigos de teste em assembly das instruções.

Por meio desse relatório será explicitado também as modificações implementadas no arquivo disponibilizado para o trabalho. Essas novas implementações foram feitas com o intuito de solucionar os problemas propostos. Vale destacar que para a conclusão de tal TP foi importante compreender claramente o funcionamento da linguagem Verilog e do caminho de dados implementado.

Os problemas propostos e solucionados serão apresentados nos tópicos seguintes.

2.XORI | XOR Immediate

Para implementar o XORI, foram necessárias as seguintes alterações:

- **ALU CONTROL:**

  4'd4: _funct = 4'd13. Como o funct3 do xori é 100, alteramos esse valor de entrada.

```
always @(*) begin
        case(funct[3:0])

            //Modificação Questao 1: mudamos de 4'd6 para 4'd4, que é o valor correto do func3 do xori
            4'd4:  _funct = 4'd13;  /* xor */

            default: _funct = 4'd0;
        endcase
    end
```

- **CONTROLE :**

Como o opcode do XORI é o mesmo do ADDI (0010011), implementamos um case que verifica o func3 e diferencia as instruções, addi:000, xori:100

Alteramos o valor do aluop para 2, que é o respectivo do xori.

```
case (opcode)
    /* Modificação Questao 1: o opcode do xori é o mesmo opcode do addi (0010011). Por isso, aqui faz um switch case
    pelo func3, que é o que diferencia os dois. A única alteração é mudar o aluop. O aluop do addi é 0. O aluop do xori é 2. */
    7'b0010011: begin   /* addi ou xori */
        case (f3)   /* addi */
            3'b000: begin
                //regdst   <= 1'b0; // rt or rd (only mips)
                aluop[1] <= 1'b0;
                alusrc   <= 1'b1;
                ImmGen   <= {{20{inst[31]}},inst[31:20]};
            end
            3'b100: begin /* xori */
                aluop[1:0] <= 2'b10;
                alusrc   <= 1'b1;
        ImmGen   <= {{20{inst[31]}},inst[31:20]};
            end
        endcase
    end
```

- **TESTES E VERIFICAÇÕES:**

Código de teste exemplo testando um caso de xor lógico:

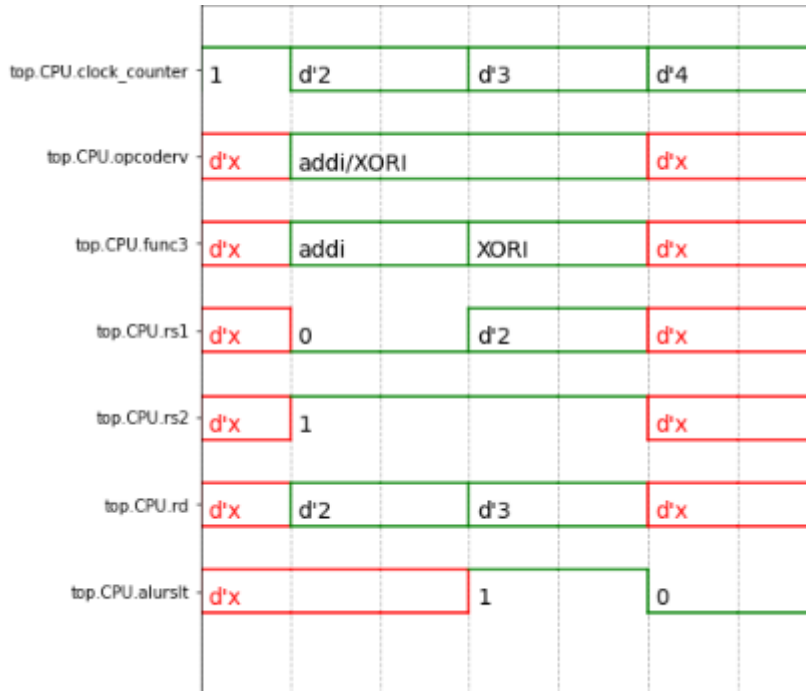
```
addi x2, x0, 1
xori x3, x2, 1
```

Saídas dos registradores. Abaixo estão representados os registradores x0,x1,x2 e x3.

```
!cat reg.data

// 0x00000000
00000000
00000001
00000001
00000000
```

Saídas em formato de ondas:



Abaixo serão mostrados os casos de teste para o XORI bit a bit:

```
addi x2, x0, 3
xori x3, x2, 4
```

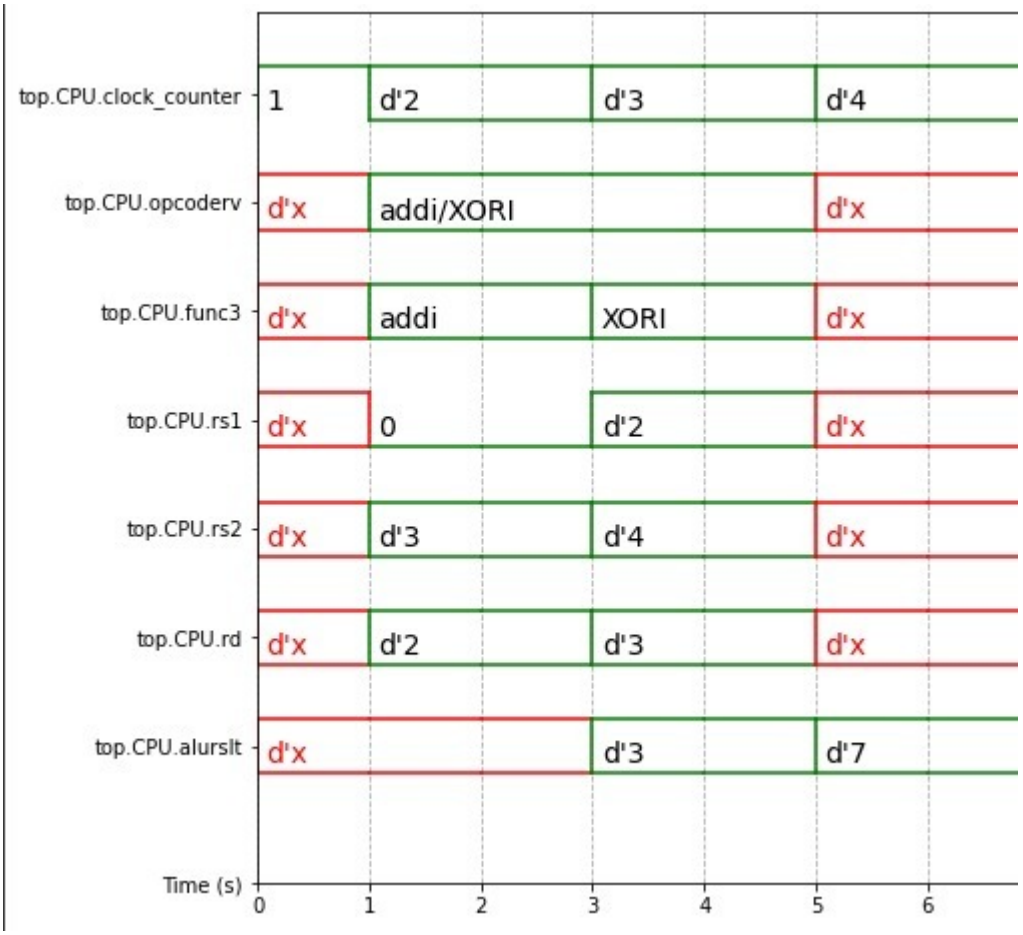
Saídas dos registradores. Abaixo estão representados os registradores x0,x1,x2 e x3:

```
[ ]  !cat reg.data

     // 0x00000000
     00000000
     00000001
     00000003
     00000007
```

Os valores de saída dos registradores estão em decimal mas mesmo assim é possível analisar a corretude da instrução XORI. Temos 3 = 011 e 4 = 100. Logo, 011 XORI 100 é igual a 111 = 7.

Saídas em formato de ondas:



3.AND | AND LÓGICO

Para implementar o AND, foram necessárias as seguintes alterações:

- **ALU CONTROL**

A operação NOR estava implementada com o func do AND, realizamos essa mudança,
4'd7: _funct = 4'd0;
func3 do AND = 111.

```
always @(*) begin
    case(funct[3:0])

        //Modificação Questao 2: O nor estava recebendo o func3 do and então atribuimos 4'd7 ao AND
        //4'd7:  _funct = 4'd12;    /* nor */
        4'd7: _funct = 4'd0; /*and*/

    endcase
end
```

- **CONTROLE**:

Como o opcode do AND é o mesmo do ADD (0110011), implementamos um case que verifica o func3 e diferencia as instruções, and:111, add:000.
O aluop das duas instruções é 2.

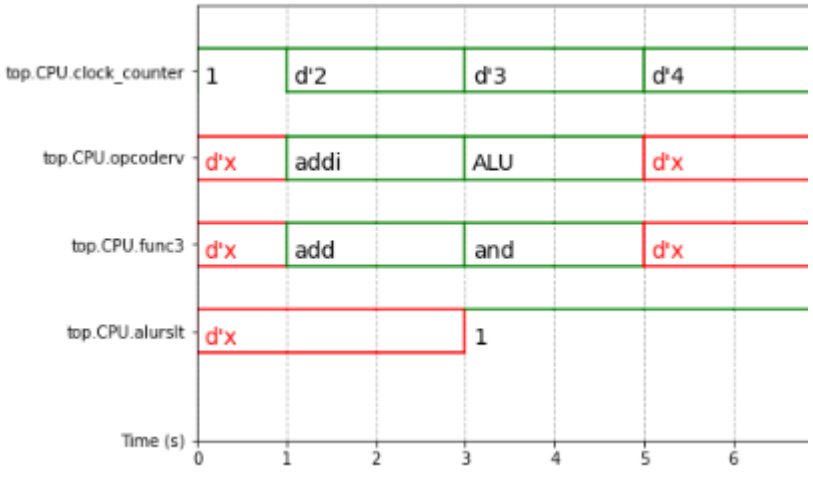
```
case (opcode)
    //Modificação Questao 2: o opcode do add é o mesmo opcode do and (b0110011).
    //Por isso, aqui faz um switch case pelo func3, que é o que diferencia os dois.
    7'b0110011: begin
        case (f3)
            3'b111: begin /* and */
                aluop[1:0] <= 2'b10;
                 //Modificação: o alusrc passa a ser 0 , vai ser o registrador
                alusrc     <= 1'b0;
                end
                //Modificação:
            3'b000: begin /* add */
                aluop[1:0] <= 2'b11;
                alusrc     <= 1'b0;
                end
        endcase
    end
```

- **TESTES E VERIFICAÇÕES:**

Código de teste exemplo:

```
addi x1, x0, 1
and x2, x1, x1
```

Teste em forma de ondas:



Apesar de no TP o and estar especificado como and lógico esse também funciona como um and bit a bit.

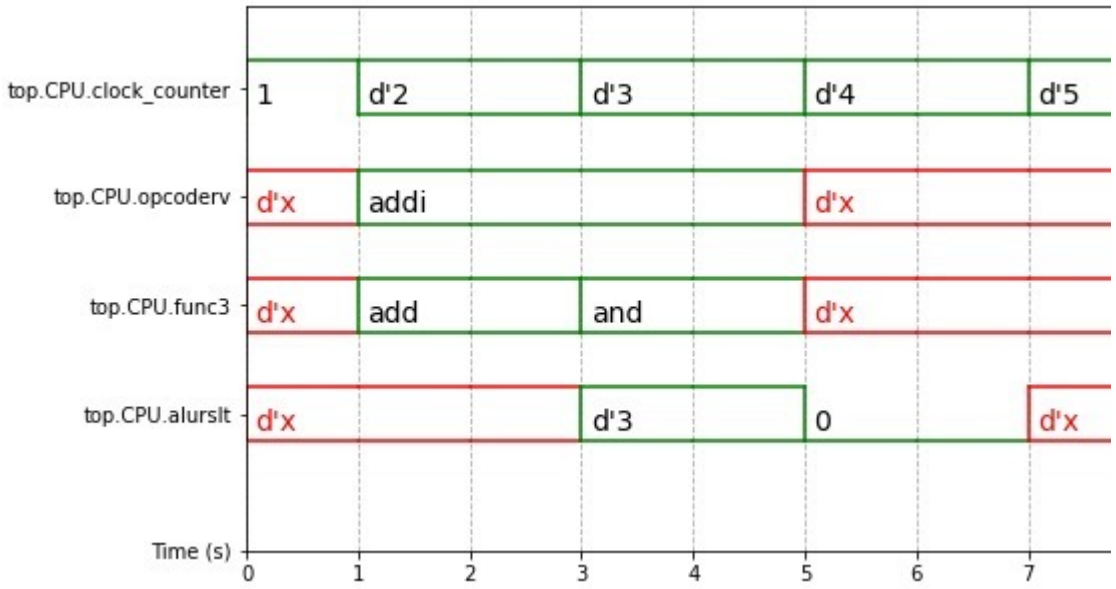
```
addi x1, x0, 3
and  x2, x1, 4
```

Saída dos registradores:

```
[94] !cat reg.data

     // 0x00000000
     00000000
     00000003
     00000000
     00000003
     00000004
     00000005
     00000006
     00000007
```

Saída em formato de onda:



Nesse caso temos 011 e 100, fazendo o and bit a bit temos 000. O resultado tanto nos registradores quanto na saída de onda está como 0.

3. JUMP

Para implementar o Jump, foram necessárias as seguintes alterações:

- **CÓDIGO DE DECODIFICAÇÃO:**

Adicionamos o valor de pc4 + ImmGen a variável jadd_s2, que após a subida do clock verifica se a instrução é um jump e atribui jadd_s2 ao novo valor do PC.

```
wire [31:0] jaddr_s2;

//Modificação Questao 3: atribuindo o valor de pc + immediate ao jaddr para realizar o jump
assign jaddr_s2 = pc4_s2 + ImmGen;
```

- **CONTROLE:**

Definimos os bits do regwrite e do ImmGen dentro do case Jump

```
case (opcode)

    //Modificação Questão 3: setando regWrite, ImmGen e ligando o wire do Jump
    7'b1101111: begin   /* j jump */
        regwrite <= 1'b0;
        ImmGen <= {{25{inst[31]}},inst[31],inst[20],inst[30:21],inst[19:12],1'b1};
        jump <= 1'b1;
    end
endcase
```

- **TESTES E VERIFICAÇÕES:**

CÓDIGO TESTE EXEMPLO:

```
addi x1, x0, 0
j pulei
addi x1,x1,2
pulei:
addi x1,x1,3
```

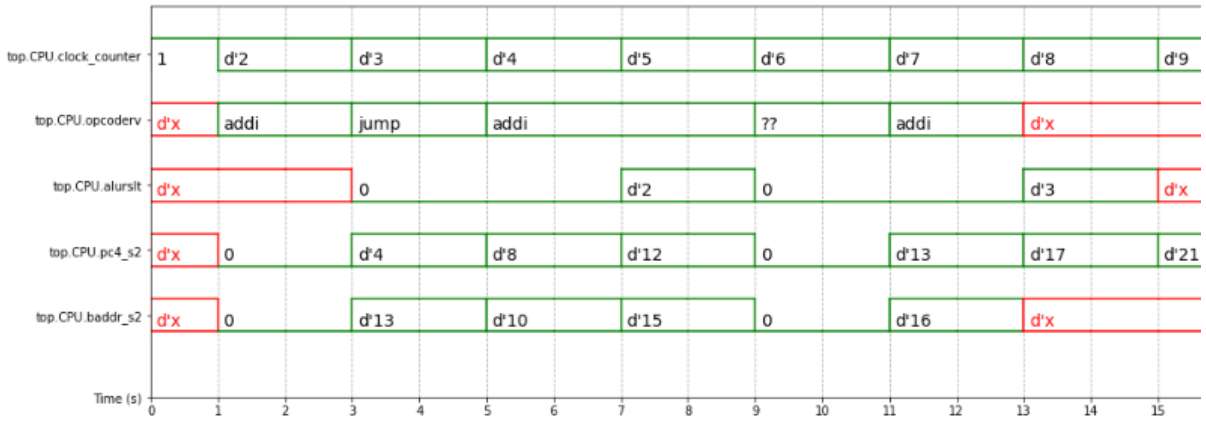
Saídas nos registradores:

```
[522] !cat reg.data

// 0x00000000
00000000
00000003
```

-O salto é realizado, a operação addi x1,x1,2 é ignorada e somente a instrução após o label ‘pulei’ é realizada e armazena o valor 3 no registrador x1.

Saídas em formato de onda:



4. BLTU - BRANCH LESS THAN UNSIGNED

Para implementar o BLTU, foram necessárias as seguintes alterações:

- **CONTROLE:**

Criamos a saída do BLTU que é associado de acordo com o opcode fornecido na entrada
Definimos como 0 o seu valor, para que somente seja ativado quando for executar sua instrução e atenda a condição.
Identificamos o case do BLTU por meio do seu func3 que o diferencia (110).

```
module control(
        input  wire [6:0]   opcode,

        //Modificação Questao 4:
        output reg          branch_eq, branch_ne, branch_lt, branch_ltu,
```

```
always @(*) begin
        /* defaults */
        aluop[1:0]  <= 2'b10;
        alusrc      <= 1'b0;
        branch_eq   <= 1'b0;
        branch_ne   <= 1'b0;

        //Modificação Questao 4: Setando wire = 0 para Branch less then - if(R[rs1]<R[rs2)
        branch_lt   <= 1'b0;
        branch_ltu  <= 1'b0;
```

```
case (opcode)
    7'b1100011: begin

        //Modificação Questão 4: Setando wire do BLTU para operação com opcode b1100011
        branch_ltu <= (f3 == 3'b110) ? 1'b1 : 1'b0;
    end
```

- **CÓDIGO DE DECODIFICAÇÃO:**

Adicionamos mais um fio que irá passar o sinal através dos estágios do processador
Realizamos essa passagem do wire entre os estágios IX/EX

```
// control (opcode -> ...)
    wire        regdst;
    wire        branch_eq_s2;
    wire        branch_ne_s2;
  wire      branch_lt_s2;

    //Modificação Questão 4: Adicionando wire do BLTU
  wire      branch_bltu_s2;
```

```
//Modificação para Questão 4: Adicionando a passagem do wire de Bltu registradores entre estágios ID/EX
    control ctl1(.opcode(opcoderv), .regdst(regdst),
                .branch_eq(branch_eq_s2), .branch_ne(branch_ne_s2), .branch_lt(branch_lt_s2), .branch_ltu(branch_ltu_s2),
                .memread(memread),
                .memtoreg(memtoreg), .aluop(aluop),
                .memwrite(memwrite), .alusrc(alusrc),
                .regwrite(regwrite), .jump(jump_s2), .ImmGen(ImmGen), .inst(inst_s2));
```

- **MAIN:**

Realizamos essa passagem do wire entre os estágios EX/MEM e MEM/WB
Definimos no case quando houver um branch e for um BLTU, o pcsrc recebe o resultado da alu para completar o ciclo e endereçar ao PC o endereço do branch

```
//Modificação Questao 4: Adicionando wire do BLTU registradores entre estágios ID/EX para registrador entre estágios EX/MEM
    wire branch_eq_s3, branch_ne_s3, branch_lt_s3, branch_ltu_s3;
    regr #(.N(3)) branch_s2_s3(.clk(clk), .clear(flush_s2), .hold(1'b0),
                .in({branch_eq_s2, branch_ne_s2,branch_lt_s2, branch_ltu_s2}),
                .out({branch_eq_s3, branch_ne_s3,branch_lt_s3, branch_ltu_s3}));
```

```
//Modificação Questao 4: Adicionando wire do BLTU do registrador entre estágios EX/MEM para registrador entre estágios MEM/WB
   wire branch_eq_s4, branch_ne_s4, branch_lt_s4, branch_ltu_s4;
   regr #(.N(3)) branch_s3_s4(.clk(clk), .clear(flush_s3), .hold(1'b0),
               .in({branch_eq_s3, branch_ne_s3,branch_lt_s3, branch_ltu_s3}),
               .out({branch_eq_s4, branch_ne_s4,branch_lt_s4, branch_ltu_s4}));
```

```
always @(*) begin
    case (1'b1)
        branch_eq_s4: pcsrc <= zero_s4;
        branch_ne_s4: pcsrc <= ~(zero_s4);
        branch_lt_s4: pcsrc <= alurslt_s4[31];

        //Modificação Questão 4: BLTU
        branch_ltu_s4: pcsrc <= (alurslt_s4[31]);

        default: pcsrc <= 1'b0;
        endcase
    end
```

- **CÓDIGO DE DECODIFICAÇÃO:**

Código de teste exemplo:

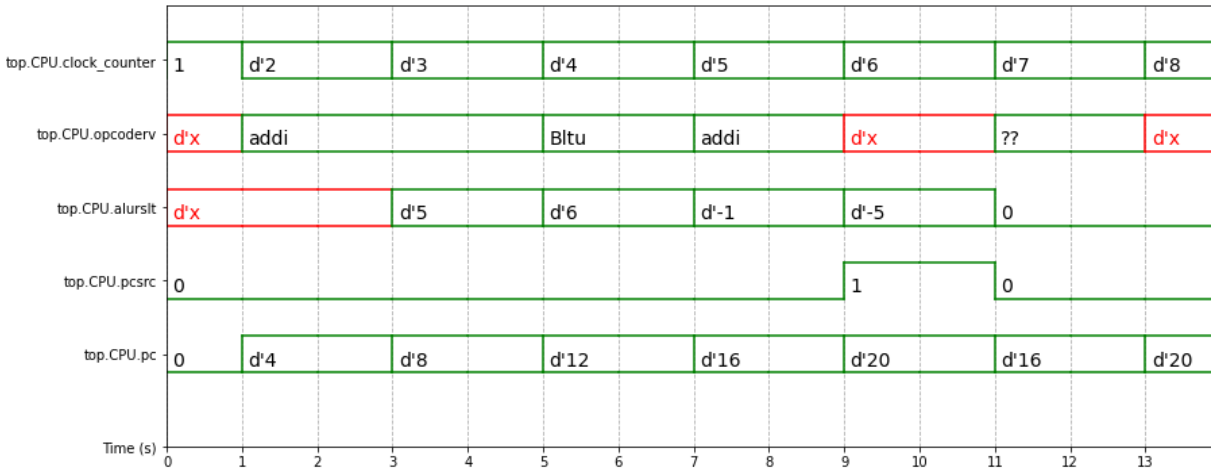
```
addi x2, x0, 5
addi x3, x0, 6
bltu x2, x3, pula
addi x2, x0, -5
pula:
```

Saída em registradores:

```
!cat reg.data

// 0x00000000
00000000
00000001
00000005
```

Saída em formato de ondas:



Tivemos dificuldade em implementar o BLTU. Em nosso trabalho temos o branch less than. No entanto , os sinais são considerados. Inicialmente buscamos implementar a ideia de adicionar um bit a mais nas entradas para que assim não houvesse mais o bit mais significativo e fosse possível fazer a implementação unsigned. No entanto , não foi possível executar essa ideia. Alcançamos apenas o branch comum.

CONCLUSÃO

Durante a execução deste trabalho foi possível que o grupo compreendesse o funcionamento da linguagem Verilog e, ao mesmo tempo, que aplicassem os conhecimentos adquiridos em sala de aula sobre a Organização de Computadores; especificamente sobre a Arquitetura RISC - V, pipeline e caminho de dados.