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

Organização de Computadores
Trabalho Prático 02

Gabriel Sacoman Teixeira Silva - Matrícula: 2019054560
Lucca Silva Medeiros - Matrícula: 2019054773
Ricardo Furbino Marques do Nascimento - Matrícula: 2020420630

1. Introdução

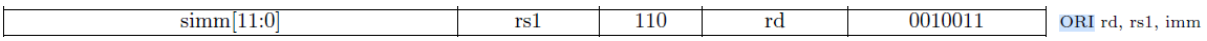
Temos como intuito neste relatório explicar as modificações feitas no arquivo .ipynb - arquivo este foi passado nas instruções deste trabalho. Essas mudanças foram realizadas para solucionar os problemas propostos e que serão mais bem descritos durante esse relatório.

Importante dizer que o objetivo deste trabalho é fazer com que nós, alunos, colocássemos em prática a implementação da linguagem Verilog. Para isso, temos a contextualização da linguagem e a elaboração de uma representação do caminho de dados do RISC-V de 32 bits. Para encontrarmos a solução para as questões, tivemos que entender o caminho de dados previamente implementado para, depois, fazermos as alterações necessárias com o propósito de tornar as operações solicitadas viáveis.

Para tornar a explicação mais didática, vamos mostrar as alterações de forma sequencial dos problemas de forma cumulativa. Isso porque algumas modificações feitas em uma questão também auxiliam na solução das próximas, mas, para evitar o retrabalho na explicação de uma mudança, deve-se assumir que uma modificação feita em uma questão anterior influencia nas seguintes.

2. Problema 1: ORI - Bitwise or immediate

A operação lógica ORI executa a operação OR bit a bit no registrador rs1 e no imediato de 12-bits imediato que vem junto com a operação. Feito isso, armazena o resultado no registrador de destino. Segue uma imagem da representação em linguagem de máquina da instrução, vinda da própria documentação do trabalho.



Para realizar essa operação, tivemos que realizar as seguintes alterações no RISC-V:

1. Tirar a atribuição que deixa o bit[1] do aluop sempre setado como 1 para permitir o funcionamento da ALU para outras operações sem ser a “add”.

```
case (opcode)
 7'b0000011: begin // lw == 3
      alusrc   <= 1'b1;
      aluop[1:0]  <= 2'b00;
      memtoreg <= 1'b1;
      regwrite <= 1'b1;
      memread  <= 1'b1;
      ImmGen   <= {{20{inst[31]}},inst[31:20]};
    end
  7'b0010011: begin /* addi */
    //regdst   <= 1'b0; // rt or rd (only mips)
    //Modificação [Questão 1]: Não deixa o bit[1] do aluop setado como 1 para
    //permitir o funcionamento da ALU para outras operações sem ser a add
    //aluop[1] <= 1'b0;
    alusrc   <= 1'b1;
    ImmGen   <= {{20{inst[31]}},inst[31:20]};
  end
```

Como o controle da ALU está setado ao _funct, a ALU conseguirá realisar novas operações.

```
always @(*) begin
  case(aluop)
    2'd0: aluctl = 4'd2;  /* add */
    2'd1: aluctl = 4'd6;  /* sub */
    2'd2: aluctl = _funct;
    2'd3: aluctl = 4'd2;  /* add */
    default: aluctl = 0;
  endcase
```

2. Adaptar o _funct das operações xor e or para os seus valores do funct3 no RISC-V.
   Código OR e XOR no RISC-V:

00000	00	rs2	rs1	110	rd	0110011	OR rd, rs1, rs2
00000	00	rs2	rs1	100	rd	0110011	XOR rd, rs1, rs2

   Código Verilog:

```
always @(*) begin
  case(funct[3:0])
    4'd0:  _funct = 4'd2; /* add */
    //Modificação [Questão 2]: Definindo o um reconhecedor para a sll no ALU Control
    4'd1:  _funct = 4'd4; /* sll */
    4'd8:  _funct = 4'd6; /* sub */
    //Modificação [Questão 1]: Modificando o reconhecedor para or no ALU Control
    4'd6:  _funct = 4'd1; /* or */
    //Modificação [Questão 1]: Modificando o reconhecedor para xor no ALU Control
    4'd4:  _funct = 4'd13;  /* xor */
    4'd7:  _funct = 4'd12;  /* nor */
    4'd10: _funct = 4'd7; /* slt */
    default: _funct = 4'd0;
  endcase
end
```

3. Problema 2: SLLI - Shift Left Logical Immediate

A operação SSLI muda para esquerda o conteúdo de rs1 conforme o que foi deixado no imediato e armazena o resultado no registrador de resultado. Na prática, como será visto nos testes, SSLI multiplica rs1 por 2 elevado ao número do imediato. Segue uma imagem da representação em em linguagem de máquina da instrução, vinda da própria documentação do trabalho.

00000	0	shamt[5:0]	rs1	001	rd	0010011	SLLI rd, rs1, imm

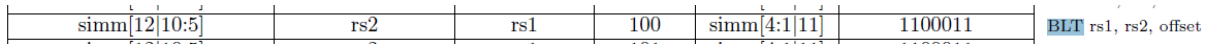
Para realizar essa operação, tivemos que realizar as seguintes alterações no RISC-V:

1. Adicionar à ALU a operação sll. Isso foi feito adicionando-a no Controle da ALU com o funct3 igual a 001, ou seja, _funct = 4'd4 em Verilog.

```
always @(*) begin
  case(funct[3:0])
    4'd0:  _funct = 4'd2; /* add */
    //Modificação [Questão 2]: Definindo o um reconhecedor para a sll no ALU Control
    4'd1:  _funct = 4'd4; /* sll */
    4'd8:  _funct = 4'd6; /* sub */
    //Modificação [Questão 1]: Modificando o reconhecedor para or no ALU Control
    4'd6:  _funct = 4'd1; /* or */
    //Modificação [Questão 1]: Modificando o reconhecedor para xor no ALU Control
    4'd4:  _funct = 4'd13;  /* xor */
    4'd7:  _funct = 4'd12;  /* nor */
    4'd10: _funct = 4'd7; /* slt */
    default: _funct = 4'd0;
  endcase
end
```

4. Problema 3: BLT - Branch on Less Than

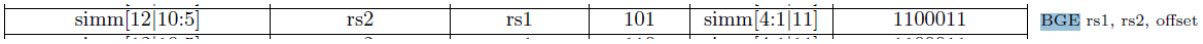
A operação BLT desvia para certa instrução caso o valor de rs1 seja menor que o de rs2. Segue uma imagem da representação em em linguagem de máquina da instrução, vinda da própria documentação do trabalho.



Felizmente para nós, ao estudarmos a implantação no estágio de modificações que estávamos, ou seja, o RISC-V com as modificações já realizadas nos exercícios anteriores, e testarmos a operação, não foi necessário realizarmos mais nenhuma alteração para BLT funcionar.

5. Problema 4: BGE - Branch on Greater Than or Equal

A operação BGE é a oposta da operação anterior (BLT): desvia para certa instrução caso o valor de rs1 não seja menor que o de rs2, ou seja, caso o valor de rs1 seja maior ou igual ao de rs2. Segue uma imagem da representação em em linguagem de máquina da instrução, vinda da própria documentação do trabalho.



Para realizar essa operação, tivemos que realizar as seguintes alterações no RISC-V:

1. Criar, no Controle do RISC-V, o wire para a operação BGE.

```
%%writefile control.v
module control(
    input  wire [6:0] opcode,
    //Modificação [Questão 4]: Adicionando operação de Branch on Greater
    //Than or Equal no controle - if(R[rs1]>=R[rs2)
    output reg      branch_eq, branch_ne, branch_lt, branch_ge,
    output reg [1:0]  aluop,
    output reg      memread, memwrite, memtoreg,
    output reg      regdst, regwrite, alusrc,
    output reg      jump,
    output reg [31:0] ImmGen,
    input [31:0] inst);
    wire[2:0] f3 = inst[14:12]; //funct3 para diferenciar as instruções de branch
  always @(*) begin
    /* defaults */
    aluop[1:0]  <= 2'b10;
    alusrc    <= 1'b0;
    branch_eq <= 1'b0;
    branch_ne <= 1'b0;
    //Modificação [Questão 4]: Setando wire = 0 para Branch on Greater
    //Than or Equal no controle - if(R[rs1]>=R[rs2)
    branch_ge <= 1'b0;
    memread   <= 1'b0;
    memtoreg  <= 1'b0;
    memwrite  <= 1'b0;
    regdst    <= 1'b1;
    regwrite  <= 1'b1;
    jump    <= 1'b0;
```

2. Adicionar comportamento deste wire para operações com o opcode de branch. Esse comportamento se baseia em "ligar" o wire, ou seja, passar 1 no fio, para quando temos o funct3 correspondente à operação de BGE.

```
7'b1100011: begin // beq == 99
    aluop  <= 2'b1;
    ImmGen   <= {{19{inst[31]}},inst[31],inst[7],inst[30:25],inst[11:8],1'b0};
    regwrite  <= 1'b0;
    //branch_eq <= 1'b1;
    branch_eq <= (f3 == 3'b0) ? 1'b1 : 1'b0;
    branch_ne <= (f3 == 3'b1) ? 1'b1 : 1'b0;
    branch_lt <= (f3 == 3'b100) ? 1'b1 : 1'b0;
    //Modificação [Questão 4]: Setando wire do BGE para operação com opcode b1100011
    branch_ge <= (f3 == 3'b101) ? 1'b1 : 1'b0;
```

3. Adicionar esse wire nos registradores entre estágios.
4. Adicionar a passagem do wire de BGE registradores entre estágios ID/EX.

```
    // control (opcode -> ...)
wire    regdst;
wire    branch_eq_s2;
wire    branch_ne_s2;
wire    branch_lt_s2;
//Modificação [Questão 4]: Adicionando wire do BGE
wire    branch_ge_s2;
wire    memread;
wire    memwrite;
wire    memtoreg;
wire [1:0]  aluop;
wire    regwrite;
wire    alusrc;
wire    jump_s2;
wire [31:0] ImmGen;  // RISCV
//
//agora passa blt para o control
//Modificação [Questão 4]: Adicionando a passagem do wire de BGE 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_ge(branch_ge_s2),
      .memread(memread),
      .memtoreg(memtoreg), .aluop(aluop),
      .memwrite(memwrite), .alusrc(alusrc),
      .regwrite(regwrite), .jump(jump_s2), .ImmGen(ImmGen), .inst(inst_s2));
```

5. Adicionar wire do BGE registradores entre estágios ID/EX para registrador entre estágios EX/MEM.

```
// transfer the control signals to stage 3
wire    regdst_s3;
wire    memread_s3;
wire    memwrite_s3;
wire    memtoreg_s3;
wire [1:0]  aluop_s3;
wire    regwrite_s3;
wire    alusrc_s3;
// A bubble is inserted by setting all the control signals
// to zero (stall_s1_s2).
regr #(.N(8)) reg_s2_control(.clk(clk), .clear(stall_s1_s2), .hold(1'b0),
    .in({regdst, memread, memwrite,
        memtoreg, aluop, regwrite, alusrc}),
    .out({regdst_s3, memread_s3, memwrite_s3,
        memtoreg_s3, aluop_s3, regwrite_s3, alusrc_s3}));
//Modificação [Questão 4]: Adicionando wire do BGE registradores entre estágios ID/EX para registrador entre estágios EX/MEM
wire branch_eq_s3, branch_ne_s3, branch_lt_s3, branch_ge_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_ge_s2}),
      .out({branch_eq_s3, branch_ne_s3,branch_lt_s3,branch_ge_s3}));
```

6. Adicionar wire do BGE do registrador entre estágios EX/MEM para registrador entre estágios MEM/WB.

```
// pass to stage 4
regr #(.N(5)) reg_wrreg(.clk(clk), .clear(flush_s3), .hold(1'b0),
      .in(wrreg), .out(wrreg_s4));
//Modificação [Questão 4]: Adicionando wire do BGE do registrador entre estágios EX/MEM para registrador entre estágios MEM/WB
wire branch_eq_s4, branch_ne_s4, branch_lt_s4, branch_ge_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_ge_s3}),
      .out({branch_eq_s4, branch_ne_s4,branch_lt_s4,branch_ge_s4}));
```

7. Definir o comportamento de BGE. Como dito, é o inverso de BLT.

```
// branch
reg pcsrc;
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] - BGE resulta no contrário da operação BLT
    branch_ge_s4: pcsrc <= ~(alurslt_s4[31]);
```

6. Testes para validação do funcionamento

O grupo desenvolveu um teste para cada problema a fim de verificar o funcionamento da nova instrução implementada, segue os testes e os resultados obtidos:

▾ Problema 1: ORI -Bitwise or immediate

```
[ ]  %%writefile simple.s
     # x2=0 or imediato=1 = x3=1
     addi x2, x0, 0
     ori x3, x2, 1

     # x4=1 or imediato=1 = x5=1
     addi x4, x0, 1
     ori x5, x4, 1

     # x6=0 or imediato=0 = x7=0
     addi x6, x0, 0
     ori x7, x6, 0
```

```
# Final Register file
!cat reg.data
```

```
// 0x00000000
00000000
00000001
00000000
00000001
00000001
00000001
00000000
00000000
00000008
00000009
0000000a
0000000b
0000000c
0000000d
0000000e
0000000f
```

Problema 2: SLLI -Shift Left Logical Immediate

```
[ ] %%writefile simple.s
    #x2 = 1
    addi x2, x0, 1

    #x3 = 1 shift 1 vez = 2
    slli x3, x2, 1

    #x4 = 1 shift 2 vezes = 4
    slli x4, x2, 2
```

```
# Final Register file
!cat reg.data
```

```
// 0x00000000
00000000
00000001
00000001
00000002
00000004
00000005
00000006
00000007
00000008
00000009
0000000a
0000000b
0000000c
0000000d
0000000e
0000000f
```

Problema 3: BLT -Branch on Less Than

```
[ ] %%writefile simple.s
    # TESTE 1
    addi x2, x0, 1
    addi x3, x0, 5
    addi x4 ,x0, 10

    # Caso 5<10 seja verdadeiro -> x2 permanece 1
    blt x3, x4, teste1

    # Caso 5<10 seja falso -> x2 = 0
    addi x2, x0, 0
    teste1:

    # Resposta x2 = 1

    # TESTE 2
    addi x5, x0, 1
    addi x3, x0, 10
    addi x4 ,x0, 5

    # Caso 10<5 seja verdadeiro -> x5 permanece 1
    blt x3, x4, teste2

    # Caso 10<5 seja falso -> x5 = 0
    addi x5, x0, 0
    teste2:

    # Resposta x5 = 0
```

```
# Final Register file
!cat reg.data
```

```
// 0x00000000
00000000
00000001
00000001
0000000a
00000005
00000000
00000006
00000007
00000008
00000009
0000000a
0000000b
0000000c
0000000d
0000000e
0000000f
```

Problema 4: BGE -Branch on Greater Than or Equal

```
[ ]  %%writefile simple.s
     # TESTE 1
     addi x2, x0, 1
     addi x3, x0, 5
     addi x4 ,x0, 10

     # Caso 5>=10 seja verdadeiro -> x2 permanece 1
     bge x3, x4, teste1

     # Caso 5>=10 seja falso -> x2 = 0
     addi x2, x0, 0
     teste1:

     # Resposta x2 = 0

     # TESTE 2
     addi x5, x0, 1
     addi x3, x0, 10
     addi x4 ,x0, 5

     # Caso 10>=5 seja verdadeiro -> x5 permanece 1
     bge x3, x4, teste2

     # Caso 10>=5 seja falso -> x5 = 0
     addi x5, x0, 0
     teste2:

     # Resposta x5 = 1

     # TESTE 3
     addi x6, x0, 1
     addi x3, x0, 10
     addi x4 ,x0, 10

     # Caso 10>=10 seja verdadeiro -> x6 permanece 1
     bge x3, x4, teste3

     # Caso 10>=10 seja falso -> x6 = 0
     addi x6, x0, 0
     teste3:

     # Resposta x6 = 1
```

```
# Final Register file
!cat reg.data
```

```
// 0x00000000
00000000
00000001
00000000
0000000a
0000000a
00000001
00000001
00000007
00000008
00000009
0000000a
0000000b
0000000c
0000000d
0000000e
0000000f
```

7. Conclusão

Conseguimos, por meio desse trabalho, compreender o funcionamento da linguagem Verilog aplicando os conhecimentos adquiridos em aula sobre Organização de Computadores, especificamente a Arquitetura RISC-V.