Organização de Computadores I

Trabalho Prático 2

João Vitor Ferreira — 2021039654

Lucas Roberto Santos Avelar — 2021039743

Luiza de Melo Gomes — 2021040075

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte — MG — Brasil

Introdução

Este trabalho prático cogita mostrar a relação entre a Organização de Computadores I e a Linguagem de Descrição de Hardware Verilog no que se refere ao conteúdo passado em sala de aula. O Google Colab é usado para executar um arquivo ".ipynb" que implementa RISC-V em Verilog. A partir disto é proposto que realizemos alterações neste código com o intuito de acrescentar funcionalidades a este código.

Estamos modificando o caminho de dados adicionando mais componentes e módulos ao código. O código Verilog e os códigos de teste serão implementados como instruções de montagem Assembly.

Este relatório também detalha as modificações incorporadas ao arquivo final do Trabalho para atender às questões nele abordadas.

É importante entender a linguagem Verilog e o caminho de dados incluído em um TP geral ao concluir esse projeto.

Para cada alteração realizada no código e documentada aqui há uma imagem que mostra o como o código ficou após a alteração ser realizada, para isso as imagens foram numeradas, e o número delas foi colocado na legenda de cada uma. Ao final de cada explicação de alteração há (Imagem **) que denomina qual imagem que representa essa mudança. Estas imagens também estarão disponíveis na pasta "IMG" junto ao ".zip" enviado.

Implementações

Nesta seção descreveremos todas as alterações realizadas no código original para a implementação das funções requeridas

Problema 1: XORI — XOR Immediate

Para podermos realizar a implementação do XORI, tivemos de realizar as seguintes alterações:

- Na ALU
  - Alteramos o valor do funct3 para a operação, originalmente ele estava implementado como 4'd6 (0110) valor que representa a operação de XOR, como queríamos implementar o XORI, este valor foi 4'd4 (0100). (Imagem 1)
- Na Unidade de Controle
  - Realizamos alterações no bloco que possui o opcode 7'b0010011, este responsável pelas operações de ADDI, ANDI, SLRI, em geral, operações com imediato. Nele tivemos de alterar o aluop para receber o valor 2 (10). Esta mudança foi feita com o intuito de induzir a unidade de controle a realizar a operação de XORI, designando o aluop como 2. Os demais valores se mantiveram inalterados. (Imagem 2)

```
always @(*) begin
  case(funct[3:0])
    4'd0:  _funct = 4'd2; /* add */
    4'd8:  _funct = 4'd6; /* sub */
    4'd5:  _funct = 4'd1; /* or */
    //4'd6:  _funct = 4'd13;  /* xor */

    /* Foi alterado o funct3 de 6 para 4. Esta alteração é referente a mudança da operação XOR para XORI*/
    4'd4: _funct = 4'd13;

    /* Foi removido o funct3 da operação NOR */
    //4'd7:  _funct = 4'd12;  /* nor */

    4'd10: _funct = 4'd7; /* slt */

    /* Por ter sido removido o funct3 do NOR alteramos para podermos codificar o AND */
    4'd7: _funct = 4'd0;

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

Imagem 1

```
    7'b0010011: begin /* addi */
      //regdst   <= 1'b0; // rt or rd (only mips)

      /* Alterado o ALUOP para 10 */
      aluop[1] <= 1'b1; /* O segundo bit alterado para 1 */
      aluop[0] <= 1'b0; /* Inicializacao do bit 0 do ALUOP */

      alusrc   <= 1'b1;
      ImmGen   <= {{20{inst[31]}},inst[31:20]};
    end
```

Imagem 2

Problema 2: AND — AND Logic

Para podermos realizar a implementação do AND, tivemos de realizar as seguintes alterações:

- Na ALU
  - Removemos o código do funct3 que representa o NOR, 4'd7 (0111) que recebia o valor 4'd12 (1100), referente ao valor de funct3 do NOR, passado para agora ser codificado para d'0 (0000), que representa o valor da função AND. (Imagem 3)
- Na Unidade de Controle
  - Realizamos alterações no bloco que possui o opcode 7'b0011011, este responsável pelas operações de ADD, AND, SLR, ou seja, operações lógicas sem o imediato. Nela adicionamos para o aluop receber 2'd2 (10) e o alusrc 1'b0 (0), portanto desativando o alusrc e designado o aluop como 2. (Imagem 4)

```
always @(*) begin
  case(funct[3:0])
    4'd0:  _funct = 4'd2; /* add */
    4'd8:  _funct = 4'd6; /* sub */
    4'd5:  _funct = 4'd1; /* or */
    //4'd6:  _funct = 4'd13;  /* xor */

    /* Foi alterado o funct3 de 6 para 4. Esta alteração é referente a mudança da operação XOR para XORI*/
    4'd4: _funct = 4'd13;

    /* Foi removido o funct3 da operação NOR */
    //4'd7:  _funct = 4'd12;  /* nor */

    4'd10: _funct = 4'd7; /* slt */

    /* Por ter sido removido o funct3 do NOR alteramos para podermos codificar o AND */
    4'd7: _funct = 4'd0;

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

Imagem 3

```
/* Mudanças para as funcoes do Tipo R, mais precisamente visando o AND */
7'b0110011: begin /* tipo - R */
  aluop <= 2'd2;
  alusrc <= 1'b0;
end
```

Imagem 4

Problema 3: J — Jump

Para podermos realizar a implementação do JUMP, tivemos de realizar as seguintes alterações:

- Na Unidade de Controle
  - Realizamos alterações no bloco que possui o opcode 7'b1101111, este responsável pela operação de jump. Este valor está inicialmente errado como 6'b000010, porém esta alteração já era esperada, pois foi reportada como um erro e corrigida. Neste opcode tivemos de alterar o controle do jump ao passar para ele que iremos realizar uma operação de jump designando-o com o valor 1'b1 (1). (Imagem 5)
- No Código de Decodificação
  - Alteramos o valor que o jaddr_s2 recebe originalmente ele estava calculando o valor do jump da forma em que é implementada no MIPS. Mudamos totalmente ela, pois o RISC-V calcula diferentemente este "pulo". (Imagem 6)

```
/* Mudanças para a realização do JUMP */
7'b1101111: begin /* j jump */
  jump <= 1'b1;
end
```

Imagem 5

```
wire [5:0]  opcode; wire [6:0]  opcoderv;
  wire [4:0]  rs;     wire [4:0]  rs1;
  wire [4:0]  rt;     wire [4:0]  rs2;
  wire [4:0]  rd;
                      wire [6:0] func7; wire [2:0] func3;
  wire [15:0] imm;
  wire [4:0]  shamt;
  wire [31:0] jaddr_s2;
  wire [31:0] seimm;  // sign extended immediate
  //
  assign opcode   = inst_s2[31:26];  assign opcoderv   = inst_s2[6:0];
  assign rs       = inst_s2[25:21];  assign rs2       = inst_s2[24:20];
  assign rt       = inst_s2[20:16];  assign rs1       = inst_s2[19:15];
  assign rd       = inst_s2[11:7];
                                     assign func7       = inst_s2[31:25];
                                     assign func3       = inst_s2[14:12];
  assign imm      = inst_s2[15:0];
  assign shamt    = inst_s2[10:6];

  /* Alteração referente ao JUMP */
  assign jaddr_s2 = {32{inst_s2[31], inst_s2[19:12], inst_s2[20], inst_s2[30:21]}} + pc + 4;
  assign seimm  = {{16{inst_s2[15]}}, inst_s2[15:0]};
```

Imagem 6

Problema 4: BLTU — Branch Less Then Unsigned

Para podermos realizar a implementação do BLTU, tivemos de realizar as seguintes alterações:

- Na Unidade de controle
  - Criamos um registrador de saída nomeado Branch_ltu, responsável pelo Branch Less Then Unsigned. (Imagem 7).
  - Diante da criação deste novo registrador tivemos que inicializá-lo com o valor 1'b0, toda vez que este módulo é executado. (Imagem 8).
  - Mudamos o bloco que possui o opcode 7' b1100011, este responsável pelas operações de BRANCH, JUMP e JAL. Nela adicionamos a decodificação para o novo registrador que foi criado. (Imagem 9)
- No Código de Decodificação
  - Criamos um fio para poder transmitir os dados do BLTU, este sendo denominado Branch_ltu_s2, a fim de seguir com a coerência já encontrada na implementação do processador. (Imagem 10)
  - Com isso tivemos de enviar este fio para o control. (Imagem 11)
  - Nesta parte, criamos o fio para receber a saída do estágio 2 e armazenar os valores no estágio 3. (Imagem 12)
  - Da mesma forma que foi realizado logo acima, fizemos o mesmo para a passagem do estágio 3 para o 4. (Imagem 13)

- Por fim tivemos de adicionar o caso do BLTU no pcsrc, para que ele possa obter o valor retornado da ALU. (Imagem 14)

```
module control(
    input  wire [6:0] opcode,
    output reg      branch_eq, branch_ne, branch_lt, branch_ltu, /* adicao branch_ltu */
    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
```

Imagem 7

```
  always @(*) begin
    /* defaults */
    aluop[1:0]  <= 2'b10;
    alusrc    <= 1'b0;
    branch_eq <= 1'b0;
    branch_ne <= 1'b0;
    branch_ltu  <= 1'b0; /* Inicializacao default do BLTU */
    memread   <= 1'b0;
    memtoreg  <= 1'b0;
    memwrite  <= 1'b0;
    regdst    <= 1'b1;
    regwrite  <= 1'b1;
    jump    <= 1'b0;
```

Imagem 8

```
   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;

       /* Adicao da decodificação do BLTU */
       branch_ltu <= (f3 == 3'b110) ? 1'b1 : 1'b0;

    end
```

Imagem 9

```
    // control (opcode -> ...)
  wire    regdst;
  wire    branch_eq_s2;
  wire    branch_ne_s2;
  wire    branch_lt_s2;
  wire    branch_ltu_s2; /* Criação do fio para o BLTU*/
  wire    memread;
  wire    memwrite;
  wire    memtoreg;
  wire [1:0]  aluop;
  wire    regwrite;
  wire    alusrc;
  wire    jump_s2;
  wire [31:0] ImmGen;  // RISCV
  //
```

Imagem 10

```
//agora passa blt para o control
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), /* Adição do Branch Less Than Unsigned */
     .memread(memread),
     .memtoreg(memtoreg), .aluop(aluop),
     .memwrite(memwrite), .alusrc(alusrc),
     .regwrite(regwrite), .jump(jump_s2), .ImmGen(ImmGen), .inst(inst_s2));
```

Imagem 11

```
/* Criação do fio do BLTU para o estagio 3 */
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}));
```

Imagem 12

```
 /* Criação do fio do BLTU para o estagio 4 */
 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}));
```

Imagem 13

```
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];

    /* Adição do caso do BLTU no pcsrc */
    branch_ltu_s4: pcsrc <= alurslt_s4[31];


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

Imagem 14

Conclusão

Ao realizar as alterações necessárias para que os objetivos propostos neste Trabalho Prático o grupo pode compreender melhor o funcionamento de um processador em RISC-V, além de aumentar o entendimento e familiaridade com a linguagem Verilog a qual este trabalho está apresentado e a linguagem Assembly para RISC-V a qual foi utilizada para poder realizar a criação de teste que comprovem que o objetivo geral do TP foi alcançado e as funções XORI, AND, JUMP e BLTU estão implementadas e funcionais. O que foi testado com o auxílio da ferramenta https://venus.kvakil.me/ que comprova que o resultado dos testes em Assembly criados e executados pelo grupo tem o resultado correto dado que a relação dos valores encontrados nos registradores deste trabalho e no da ferramenta apresentam o mesmo valor.

Com isso temos que este trabalho foi de suma importância e contribuição para inferimos os conteúdos tratados neste módulo em sala de aula pela disciplina de Organização de Computadores I, já que foi trabalhado com a arquitetura RISC V — incluindo o pipeline e o caminho de dados — em detalhes.