Organização de Computadores I

Trabalho Prático 2

Leonardo Bhering Damasceno — 2020006728

Pedro Henrique E.Dalla-Lana - 2020420613

Vinicius Trindade Dias Abel — 2020007112

1. Introdução

Este trabalho prático se refere a Linguagem de Descrição de Hardware Verilog. Neste trabalho o Google Colab e o Visual Studio Code foram usados em conjunto para executar um arquivo “.ipynb” que implementa RISC-V em Verilog previamente disponibilizado e suas alterações, utilizando sempre o original como referência para evitar alterações catastróficas e sensíveis.

A partir disto realizamos alterações no código com o intuito de acrescentar funcionalidades solicitadas no enunciado, modificando o caminho de dados e adicionando mais componentes.

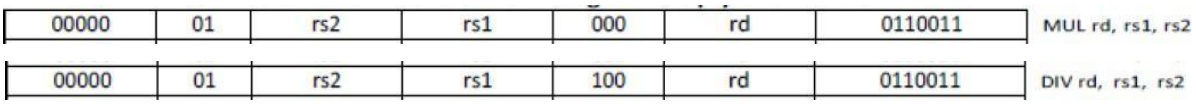
O código Verilog e de teste serão implementados por via de instruções de montagem Assembly. Para cada alteração realizada no código há uma imagem correspondente que mostra o como o código após a alteração.

---

2. Implementação

2.1 MUL
2.2 DIV

Como o código a ser alterado é basicamente o mesmo para as duas instruções, por as instruções MUL e DIV serem aritméticas, reunimos ambas em um só bloco.



Observamos que as instruções são basicamente idênticas, porém há a peculiaridade de ser necessário receber um bit extra (mais precisamente o [5] do funct7) que será mencionado em seguida.

Inicialmente o código original comporta um problema que é o funct só receber três bits, e para implementarmos as operações devemos alterar o sinal funct para que ele receba também a informação de f7[0], então sendo necessário receber mais bits, isso faz com que todos os outros decimais se alterem para comportar a mudança (o único permanecendo inalterado é o add, pois 7'b0 = 4'b0 = 7'd0)

```
module alu_control(
  /* input wire [3:0] funct */
  input wire [5:0] funct, // Atualizado
  input wire [1:0] aluop,
  output reg [3:0] aluctl
);

  reg [3:0] _funct;
  reg [3:0] _functi;

  always @(*) begin
    case(funct)
      6'd0:   _funct = 4'd2; /* add */
      6'd64:  _funct = 4'd6; /* sub */
      6'd32:  _funct = 4'd1; /* or */
      6'd20:  _funct = 4'd13;  /* xor */
      6'd28:  _funct = 4'd7; /* slt */
      6'd48:  _funct = 4'd0; /* and */
      6'd8:   _funct = 4'd5;  /* sll */
      6'd40:  _funct = 4'd8;  /* srl */

      6'd72:  aluctl = 4'd10; /* mul */
      6'd76:  aluctl = 4'd11; /* div */
```

Dessa forma, o código fica assim:

Além disso, é necessário alterar o pipelineDecExec_s2 Incluindo o bit do sinal que está sendo ligado ao seu campo funct extra sendo passado para .out(func_s3), para a operação ser reconhecida.

```
/* regr #(.N(5)) func7_3_s2(.clk(clk), .clear(stall_s1_s2 || flush_s2), .hold(1'b0),
          .in({func7[5],func3}), .out(func_s3));*/

regr #(.N(6)) func7_3_s2(.clk(clk), .clear(stall_s1_s2 || flush_s2), .hold(1'b0),
          .in({func7[6], func7[5], func3}), .out(func_s3));
```

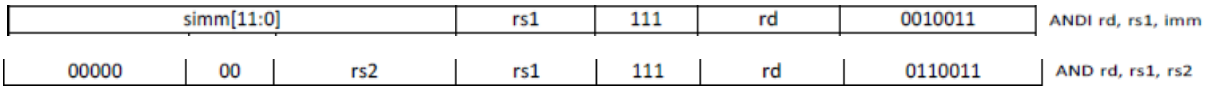
É necessário alterar também o case statement, colocando mul e div no final com seus códigos e operações correspondentes (identificadores arbitrários).

```
always @(*) begin
    case (ctl)
        4'd2:  out <= add_ab;              /* add */
        4'd0:  out <= a & b;               /* and */
        4'd12: out <= ~(a | b);            /* nor */
        4'd1:  out <= a | b;               /* or */
        4'd7:  out <= {{31{1'b0}}, slt};   /* slt */
        4'd6:  out <= sub_ab;              /* sub */
        4'd13: out <= a ^ b;               /* xor */
        4'd5: out <= a << b[4:0];  /* sll */
        4'd8: out <= a >> b[4:0];       /* srl */
        4'd10: out <= a * b;             /* mul */
        4'd11: out <= a / b;             /* div */

        default: out <= 0;
    endcase
```

2.3 ANDI

Podemos reparar que a função ANDI e AND são bastante similares, principalmente tendo em vista que a lógica para decodificá-la já está implementada



Alterando o código para adicionar a instrução ANDI à lógica da ALU, é necessário alterarmos o valor do funct para a operação, como alteramos o sinal para as funções anteriores, originalmente ele estava implementado no formato 4'dX, porém agora foi implementado como 7'd12

Como a operação é a mesma, só mudando as entradas, na nova instrução encaminhamos para o mesmo caminho que a instrução AND.

```
always @(*) begin
  case(f7_f3)
    7'd0:  _funct = 4'd2; /* add */
    7'd64: _funct = 4'd6; /* sub */
    7'd32: _funct = 4'd1; /* or */
    7'd20: _funct = 4'd13;  /* xor */
    7'd28: _funct = 4'd7; /* slt */
    7'd48: _funct = 4'd0; /* and */
    7'd8:  _funct = 4'd5;  /* sll */
    7'd40: _funct = 4'd8;  /* srl */

    7'd72:  aluctl = 4'd10; /* mul */
    7'd76:  aluctl = 4'd11; /* div */

    7'd12:  _funct = 4'd0;   /* andi */

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

2.4 BEQ

A única instrução do trabalho que não se encontra no bloco ALU. Para adicionar a instrução BEQ ao módulo control, precisamos primeiro adicionar um novo caso ao bloco case (opcode) à instrução, sendo esse o 7'b1100011: Tendo em vista que o registrador "branch_eq" já existe.

Foi gentilmente fornecido a dica do monitor que após os testes se mostrou operante, então indicamos que:

ImmGen <= {{19{inst[31]}},inst[31],inst[7],inst[30:25],inst[11:8],{1'b0}};

Sendo definido para extrair os bits relevantes da instrução.

Definimos a aluop como "2'd1" para realizar uma comparação na ALU.

```
 7'b1100011: begin  /* BEQ */
    aluop <= 2'd1;
    alusrc <= 1'b0;
    regwrite <= 1'b0;
    branch_eq <= 1'b1;
    ImmGen <= {{19{inst[31]}}, inst[31], inst[7], inst[30:25], inst[11:8], 1'b0};
end
```

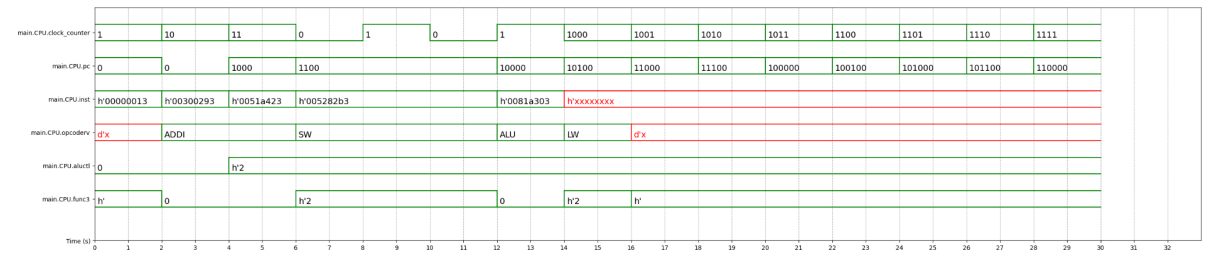
3. Waveforms

Em sequência, colocamos as waveforms dos sinais relevantes para cada operação trabalhada no enunciado, o código para MUL, DIV e BEQ foram reaproveitados do que já estava funcionando no código:

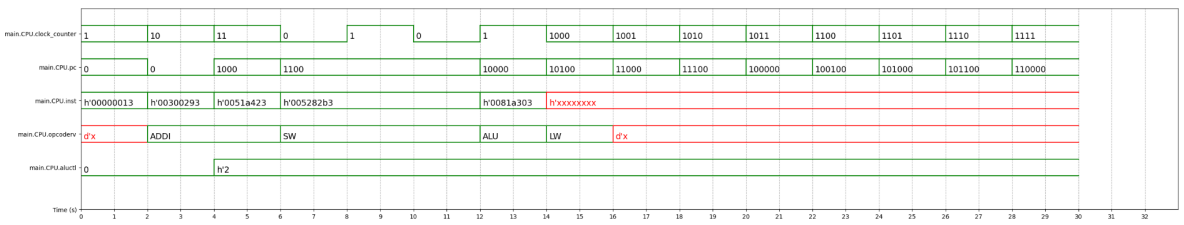
MUL

```
%%waveform test.vcd

op_dict = [{'11': 'LW', '1100011': 'BEQ', '110011': 'ALU', '100011':
'SW', '10011': 'ADDI', '10': 'MUL'}, {'10': 'sp', '111': 't2', '1001':
's1', '1000': 's0', '110': 't1', '100': 'tp', '1': 'ra', '11': 'gp',
'101': 't0'}]
sign_list = ['main.CPU.clock_counter, dec', 'main.CPU.pc, dec',
'main.CPU.inst', 'main.CPU.opcoderv, r[0]', 'main.CPU.aluctl',
'main.CPU.func3']
time_begin = 0
time_end = 32
base = 'hex'      # bin, dec, dec2, hex, octal
```



DIV

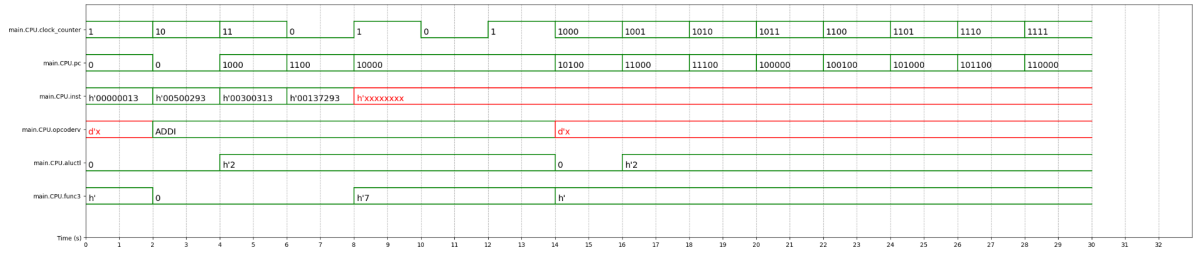


Nas waveforms listadas, mostramos além “aluctl” e o sinal do “funct3” do que era exibido previamente, já que são relevantes para a questão.

ANDI

código do teste:

```
00000013
00500293
00300313
00137293
```



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 e a linguagem Assembly para RISC-V, que foi utilizada para poder realizar a criação de testes que comprovem que o objetivo geral do TP foi alcançado e as funções MUL, DIV, ANDI e BEQ estão implementadas e funcionais.

Portanto, o 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.