Organização de Computadores
Trabalho prático 2

Daniel Oliveira Nascimento - 2020054293
Janderson Glauber Mendes dos Santos - 2020054544
Kleber Junior Alves Pereira - 2020054625

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. PROBLEMA 1: ANDI -Bitwise or immediate.

Para implementar o *ANDI*, foram necessárias as seguintes modificações:

- *ALUCtl:* alteração uso do $func3 = 7$ (b'111) que operava o nor para que passasse a operar o $andi$. Quando $func3 = 7$, a *ALUCtl* retornava *12* e a *ALU* executava o *nor*. Após a modificação, a *ALUCtl* recebe o valor *0*, o que faz a *ALU* operar o *and*.
- *Conrol*: alteração do bloco de código que controla as ações referentes ao *opcode 7'b0010011*, o qual é utilizado para referenciar as operações *addi, andi*

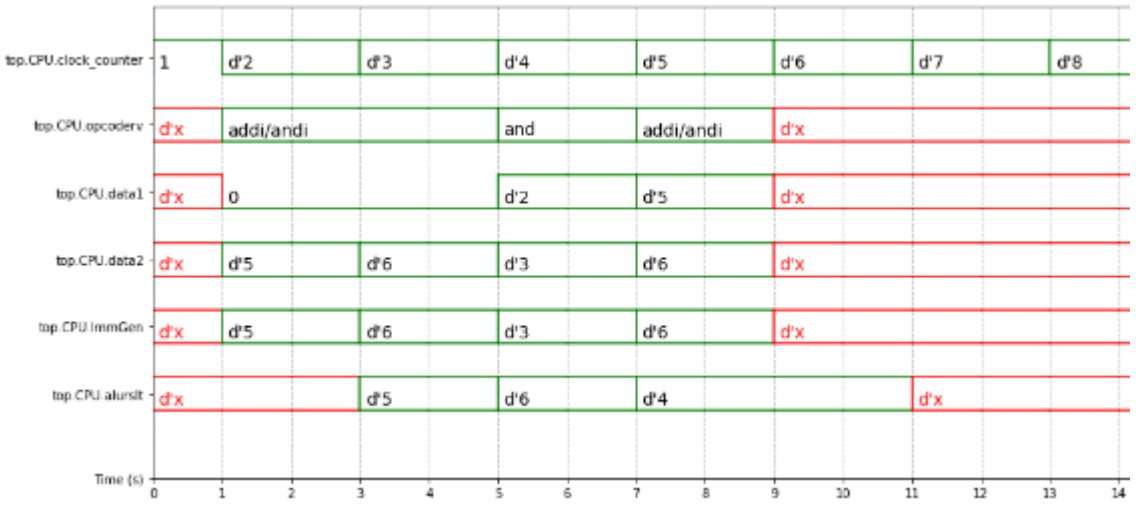
e *slri*. Neste bloco, foi necessário adicionar um operador ternário na atribuição de valores da *aluop*: no caso do *func3* determinar que a operação é um *add*, o *aluop* é *0* (adição), caso contrário, o *aluop* é *3* (operação irá depender do *func3* dentro da *ALU*);

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

  reg [3:0] _funct;

  always @(*) begin
    case(funct[3:0])
      4'd0:  _funct = 4'd2; /* add */
      4'd8:  _funct = 4'd6; /* sub */
      // MODIFIED: it doesn't apply to "or" anymore, but to "slri" instead
      4'd5:  _funct = 4'd3; /* slri */
      4'd6:  _funct = 4'd13;  /* xor */
      // MODIFIED: it doesn't apply to "nor" anymore, but to "andi" instead
      4'd7:  _funct = 4'd0; /* andi */
      4'd10: _funct = 4'd7; /* slt */
      default: _funct = 4'd0;
    endcase
  end
```

Através das formas de ondas, é possível comparar a função *and*, que opera com dois registradores, com a operação *andi*. Ambas retornam o mesmo resultado para os mesmos valores de entrada, no entanto, o segundo operando do *andi* é um imediato.



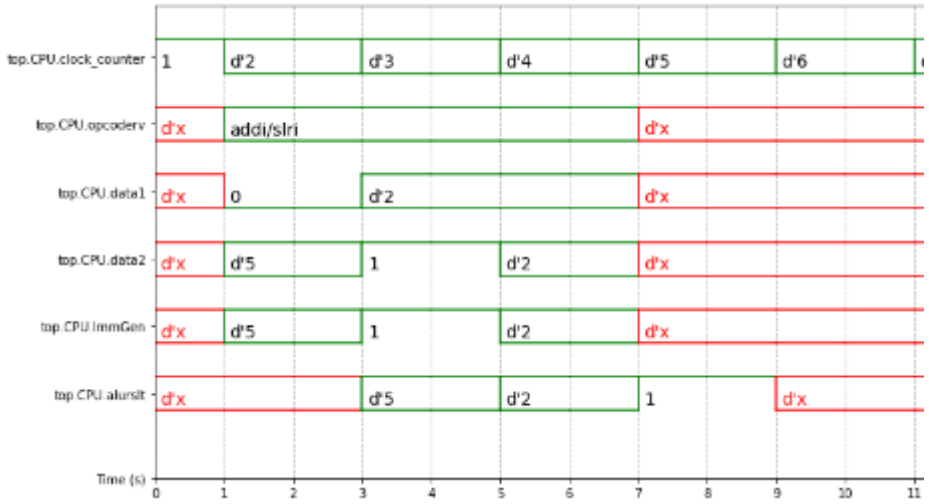
3. PROBLEMA 2: SRLI-shift Right Logical Immediate.

Para implementar o *SRLI*, foram necessárias as seguintes modificações:

- *ALUCtl*: Alteração do uso do *func3 = 5 (b'101)* que operava o *or* para que passasse a operar o *slri*. Quando *func3 = 5*, a *ALUCtl* retornava *1* e a *ALU* executava o *or*. Após a modificação, a *ALUCtl* retorna o valor *3*, que faz a *ALU* operar o *slri*.
- *ALU*: também foi necessário adicionar uma nova operação à *ALU*, a qual referencia o valor *3* da *ALUCtl*. Essa operação consiste em realizar o *shift right* entre os dois operandos.

```
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 */
    // MODIFIED: added "4'd3" case to solve shift right
    4'd3: out <= a >> b;        /* slr */
    default: out <= 0;
  endcase
end
```

Através das formas de ondas, é possível verificar o funcionamento através da realização de dois *shifts* no número *5*: um *shift 1* e um *shift 2*. O primeiro retorna o valor *2 (3'b101 >> 1 = 3'b010 = 2)*; já o segundo, o valor *1 (3'b101 >> 2 = 3'b001 = 1)*.



4. PROBLEMA 3: J-Jump.

Para implementar o *J*, foram necessárias as seguintes modificações:

- *Control*: Alteração do uso do opcode de *6'b000010* para *7'b1101111* no código. Isso porque a função *jump* já estava implementada, no entanto, o seu opcode estava incorreto.

No entanto, essa modificação não foi suficiente para que o *jump* operasse corretamente, possivelmente devido à utilização de partes incorretas da instrução no momento de atualizar o valor do *PC*: o código sempre ignora as instruções que sucedem o *jump*.

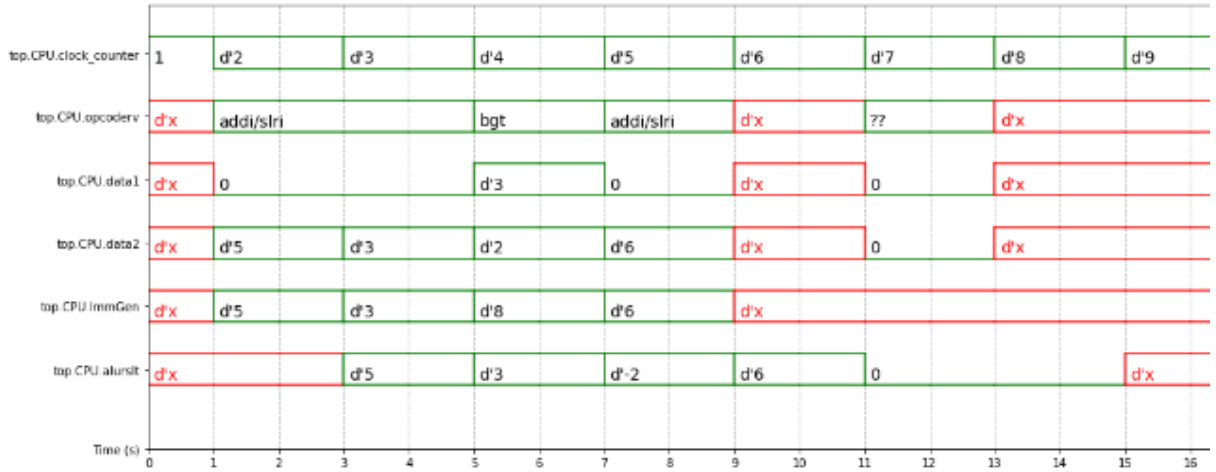
5. PROBLEMA 4: BGT-Branch or Greater Than.

Para implementar o *BGT*, foram necessárias as seguintes modificações:

- Control: por padrão, o código sempre iria tomar o *bgt* como verdadeiro. Para corrigir isso, foi necessário inicializar a *flag* que controla o *bgt* em *0*.

```
/* defaults */
aluop[1:0]  <= 2'b10;
alusrc    <= 1'b0;
branch_eq <= 1'b0;
branch_ne <= 1'b0;
// MODIFIED: set branch_lt input to 0 as default
branch_lt <= 1'b0;
memread   <= 1'b0;
memtoreg  <= 1'b0;
memwrite  <= 1'b0;
regdst    <= 1'b1;
regwrite  <= 1'b1;
jump    <= 1'b0;
```

Através das formas de onda, é possível verificar que o resultado do bgt sempre será negativo quando houver desvio e sempre será positivo caso contrário.



6. 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.