Trabalho Prático da Disciplina de Organização de Computadores I

Alex Eduardo Alves dos Santos
Matrícula: 2021032145

Kayque Meira Siqueira
Matrícula: 2022043850

Mateus Augusto Gomes
Matrícula: 2022043760

Departamento de Ciência da Computação (DCC) -
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil
alexeduardoaeas@ufmg.br
mateusg2022@ufmg.br

INTRODUÇÃO

O objetivo central do projeto foi implementar as funções **andi, mul, div** e **beq** no ambiente Risc-V pré-configurado, envolvendo alterações nos arquivos de tradução de instruções para representação hexadecimal, com inclusão de **OPCODE, Funct3, Funct 7** e especificações do tipo para mul e div. Ademais, foram necessárias adaptações na unidade de controle (**control.v**) para incorporar o tratamento do OPCode das novas instruções, além de ajustes na ALU e no controle da ALU **(alu_control)**, para integrar o tratamento de **Funct3** e **Funct7**, delineando como seriam realizadas as operações na ALU **(alu.v)**.

Essas modificações não apenas introduziram novas instruções ao processador Risc-V, mas também exigiram uma reconfiguração profunda nos elementos fundamentais do sistema, desde os **arquivos de tradução** até a **unidade de controle e a ALU**.

DESENVOLVIMENTO

- **instrução: ANDI**

O processo de implementação da função ANDI foi iniciado com uma análise do funcionamento do pipeline e do esquema/desenho para facilitar a compreensão. Observou-se que o ANDI compartilha características operacionais com o AND, sendo ambas operações lógicas do mesmo tipo e possuindo o mesmo funct3. As alterações necessárias foram realizadas na unidade de controle e na ALU. Em particular, o valor 2'd3 foi atribuído ao Aluop para direcionar o sinal de AND para a ALU, seguido pela execução da operação AND com o imediato.

```
always @(*) begin
    case(funct[2:0])
      3'd0:  _functi = 4'd2;  /* add */
      3'd6:  _functi = 4'd1;  /* or */
      3'd4:  _functi = 4'd13; /* xor */
      3'd2:  _functi = 4'd7;  /* slt */
      3'd1: _functi = 4'd5;  /* sll */
      3'd5: _functi = 4'd8;  /* srl */

      3'd7: _functi = 4'd0; /* and */
```

Figura 1: ALU Control do ANDI.

```
7'b0010011: begin /* andi */
  aluop <= 2'd3;
  alusrc <= 1'b1;
  regwrite <= 1'b1;
  ImmGen <= {{20{inst[31]}},inst[31:20]};
end
```

Figura 2: OPCODE ANDI - Unit Control

MUL e DIV

Primeiramente, foram definidos os códigos opcode, **Funct3** e **Funct7** para ambas as instruções. Devido ao comportamento semelhante dessas instruções - sendo operações aritméticas inversas - a implementação prosseguiu de forma paralela. Ao analisarmos o funcionamento da instrução **'add'**, adicionamos uma linha no código Verilog dentro da ALU, especificamente **'assign mul_ab = a * b'**. Uma dificuldade encontrada foi perceber que **'mul_ab'** deveria ser declarado como **'wire [31:0]'**, caso contrário, o valor seria calculado, mas não seria propagado corretamente. O mesmo ajuste foi aplicado à variável **'div_ab'**. Após essa correção, as instruções passaram a funcionar adequadamente.

```
wire [31:0] mul_ab;
wire [31:0] div_ab;
assign mul_ab = a * b;
assign div_ab = a / b;
wire    oflow_add;
```

*Definição dos fios e atribuição dos valores da multiplicação e da divisão.

```
    'mul': 'R',
    'div': 'R'
}
```

*Definição do Tipo das operações

```
    'mul': '0110011', #Opcode
    'div': '0110011'
}
```

*Definição do OPCODE das operações

```
    'mul': '0100000', #Funct7
    'div': '0100000',
}
```

*Definição do Funct7 das operações

```
'mul': '001', #Funct3
'div': '010'
```

*Definição do Funct3 das operações

```
wire [31:0] sub_ab;
wire [31:0] add_ab;
wire [31:0] mul_ab;
wire [31:0] div_ab;

wire    oflow_add;
wire    oflow_sub;
wire    oflow_mul;
wire    oflow_div;
```

*Declaração dos fios das operações

```
assign sub_ab = a - b;
assign add_ab = a + b;
assign mul_ab = a * b;
assign div_ab = a / b;
```

*Atribuição dos resultados nas variaveis

```
4'd9:  out <= mul_ab;
4'd10:  out <= div_ab;
```

Saida da ALU

```
4'd9: _funct = 4'd9;
4'd10: _funct = 4'd10;
```

Controle da ALU das operações MUL e DIV

BEQ

Para implementar a **instrução BEQ**, primeiramente, foi necessário incorporar o tratamento do **OPCODE** da operação, entender a captura do Imediato **(ImmGen)** por meio da documentação do Risc-V e decidir que a operação a ser executada pela **ALU** seria uma subtração. Durante a análise da implementação existente, identificou-se a presença do fio denominado **"branch_eq"**, crucial para o êxito da operação, resultando na ativação do sinal **"pc_src"**. Além disso, uma análise minuciosa dos pipelines revelou a dinâmica de transferência de informações entre os diferentes estados, envolvendo elementos como **baddr_s2_s3** (Branch Address do Estágio 2 para o Estágio 3), **branch_eq_s2** (condição de desvio igual no Estágio 2), **branch_eq_s3** (condição de desvio igual no Estágio 3) e branch_s2_s3 (registrador que armazena e fornece o valor de branch_eq_s2 como branch_eq_s3 no próximo ciclo de clock).

Detalhes importantes considerados:

- Incorporação do tratamento do OPCODE da operação.
- Compreensão da captura do Imediato (ImmGen) a partir da documentação do Risc-V.
- Escolha da operação de subtração na ALU.
- Análise dos elementos presentes nos pipelines, incluindo baddr_s2_s3, branch_eq_s2, branch_eq_s3 e branch_s2_s3.
- Identificação do fio "branch_eq" como essencial para o sucesso da operação e ativação do sinal "pc_src".

```
    7'b1100011: begin /* BEQ */
                ImmGen   <= {{20{inst[31]}},inst[31:25],inst[11:7]};
                aluop <= 2'd1;
                branch_eq <= 1'b1;
      end
```

Testes:

Teste 1 - ANDI

obs: addi está sendo usado para inicializar os registradores

```
nop
addi x2, x0, 5
addi x3, x0, 7

andi x4, x2, 31
andi x5, x3, 40
end: nop
```

Teste feito para verificar o retorno quando o registrador é impar e o imediato é par ou impar.
x4 deve ser positivo
x5 deve ser negativo

Teste 2 - MUL

```
nop
addi x5, x0, 3
addi x4, x0, 1
addi x2, x0, 7
addi x3, x0, 9
mul x5, x4, x5
mul x6, x2, x3
end: nop
```

Testes realizando a operação mul, após atribuir valores aos registradores, o valor verificado dos registradores x5 e x6 são, 3 e 63, respectivamente.

Teste 3 - DIV

```
nop
addi x5, x0, 3
addi x4, x0, 9
addi x2, x0, 90
addi x3, x0, 9
div x5, x4, x5
div x6, x2, x3
end: nop
```

Testes realizando a operação div, após atribuir valores aos registradores, o valor verificado dos registradores x5 e x6 são, 3 e 10, respectivamente.

Teste 4 - BEQ

```
nop
addi x3, x3, 10
addi x2, x2, 10
beq x3, x2, end_program // termina se os valores forem iguais
addi x11, x11, 9
addi x12, x12, 8
addi x13, x13, 7
end_program: nop
```

verifica se os valores são iguais e finaliza o programa.

```
nop
addi x3, x3, 10
addi x2, x2, 9
beq x3, x2, end_program // termina se os valores forem iguais
addi x11, x11, 9
addi x12, x12, 8
addi x13, x13, 7
end_program: nop
```

Outro teste, mas com o branch falhando

CONCLUSÃO

Em conclusão, a implementação bem-sucedida das instruções ANDI, MUL, DIV e BEQ no ambiente Risc-V exigiu uma abordagem meticulosa e colaborativa. Ao enfrentar desafios desde a compreensão do pipeline até a modificação das unidades de controle e ALU, colaboraram para o aprendizado em OC1 aplicado na prática. A análise cuidadosa dos pipelines e a integração eficaz dos novos nos levou a pesquisar sobre o funcionamento da linguagem verilog para implementação das novas instruções, implicando na continuidade do aprendizado de ISL.

Este projeto não apenas aprimorou nossas habilidades práticas na manipulação de circuitos e instruções, mas também ressaltou a importância da linguagem verilog e ded uma boa estruturação do pipeline.