Trabalho Prático 2

Gabriela Tavares Barreto

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

1 Introdução

O objetivo desse trabalho prático consistia em nos familiarizar com a linguagem de descrição de Hardware Verilog, por meio da implementação de um processador RISC-V de 5 estágios. A tarefa compreendia em adicionar quatro diferentes instruções faltantes na implementação já disponibilizada. Como isso foi feito será detalhado a seguir.

2 Implementação

As instruções a serem implementadas eram:

- mul,
- div,
- andi,
- beq.

2.1 Instruções mul e div

A instrução mul realiza a multiplicação de dois registradores, e armazena o resultado em um registrador de destino. Ela tem seguinte forma:

funct7	rs2	rs1	funct3	rd	opcode
0000001	registrador 2	registrador 1	000	registrador de destino	0110011

Analogamente, a instrução div realiza a operação de divisão entre dois registradores, arrendondando o resultado para baixo quando necessário. Ela segue o formato abaixo:

funct7	rs2	rs1	funct3	rd	opcode
0000001	registrador 2	registrador 1	100	registrador de destino	0010011

Podemos ver que ambas instruções diferem entre si apenas pelo funct3. Além disso, elas possuem o mesmo opcode de outras operações aritméticas já implementadas no processador(add, sub, entre outras). Elas se diferenciam dessas operações pelo algarismo menos significativo do funct7, que para ambas é 1.

Na última célula do Decode Unit, está declarado um fio que recebe 5 bits denominado **func_s3**. Este mesmo fio é usado como parâmetro quando o módulo alu_control é instanciado na célula Exec Code no Execution Stage.

```
72 // modified ** update ALU Control **
73   wire [4:0] func_s3;
```

```
alu_control alu_ctl1(.funct(func_s3), .aluop(aluop_s3), .aluctl(aluctl));
```

Inicialmente o func_s3 recebia de forma concatenada o quinto bit de funct[7](que é utilizado para diferenciar instruções aritméticas) e os três bits de funct3. Assim, para poder verificar se uma instrução era mul ou div, foi necessário atribuir também o bit menos significativo de funct7 a func_s3. Essa alteração foi feita na célula Pipeline Bar (Decode → Exec).

```
regr #(.N(5)) func7_3_s2(.clk(clk), .clear(stall_s1_s2 || flush_s2), .hold(1'b0),
          .in({func7[0],func7[5],func3}), .out(func_s3));
```

Feita essa alteração, func_s3 passou a receber funt7[0]. Depois disso, dentro da célula ALU Control and ALU, foi adicionada a verificação do funct[4] no módulo **alu_control** para poder diferenciar a soma da multiplicação(ambas intruções tem o mesmo opcode e funct3, mas se diferem no funct7[0]) e **div de xor**(que também tem o mesmo opcode e funct3, mas se diferem no funct7[0]).

```
always @(*) begin
  case(funct[3:0])
    4'd0:  begin
          _funct = 4'd2;  /* add */
          case(funct[4])
            1'd1: begin  /* mul */
              _funct = 4'd3;
            end
          endcase
    end
```

(a) add e mul tem funct3 = 000

```
4'd4:  begin
       _funct = 4'd13; /* xor */
       case(funct[4])  /* div */
         1'd1: begin
           _funct = 4'd4;
         end
       endcase
end
```

(b) xor e div tem funct3 = 100

Foram adicionados novos fios ao módulo **alu**, e atribuídos a eles as correspondentes operações.

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

(a) Fios para mul e div

```
assign mul_ab = a * b;
assign div_ab = a/b;
```

(b) Definindo operações

Por fim, adicionamos ao case(ctl) as operações de mul e div, para garantir o output correto.

```
4'd3: out <= mul_ab;
4'd4: out <= div_ab;
```

2.2 Instrução andi

A instrução **andi** faz a operação lógica **&** bit a bit entre um registrador e um imediato, e guarda o resultado em um registrador de destino. Ela tem o seguinte formato:

imm[11:0]	rs1	funct3	rd	opcode
0000001	registrador 1	111	registrador de destino	0010011

Por ela ter o mesmo opcode da operação addi, não foi necessário fazer mudanças na Control Unit, pois addi já estava implementada. Para que a operação pudesse ocorrer corretamente, foi necessário alterar apenas o módulo alu_control, adicionando o caso que reconhecesse o andi levando em conta que seu funct3 é 111.

```
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'd7:  _functi = 4'd0;  /* andi */
      3'd2:  _functi = 4'd7;  /* slt */
      3'd1: _functi = 4'd5;  /* sll */
      3'd5: _functi = 4'd8;  /* srl */
      default: _functi = 4'd0;
    endcase
  end
```

2.3 Instrução beq

A instrução beq(branch equal) verifica se dois registradores tem o mesmo conteúdo. Caso sim, ela faz um desvio condicional para o endereço indicado.

imm[12‖10 : 5]	rs2	rs1	funct3	imm[4:1‖11]	opcode
imediato	registrador 1	registrador 2	000	imediato	1100011

Para poder implementá-la, tivemos que adicionar a Control Unit um caso que identificasse a instrução pelo seu opcode.

```
7'b1100011: begin /* beq */
  aluop <= 2'd1;
  branch_eq <= (f3 == 3'b000) ? 1'b1 : 1'b0;
  ImmGen   <= {{19{inst[31]}},inst[31],inst[7],inst[30:25],inst[11:8],{1'b0}};
end
```

Os bits do caminho de dados foram alterados da seguinte maneira: ao aluop foi atribuído o valor 1, visto que o beq faz a subtração entre os dois valores para verificar sua igualdade. O bit branch_eq, que por default é 0, torna-se 1 ao ser verificado pelo funct3 que a instrução é beq. Por fim, ImmGen foi atualizado para receber o endereço da instrução para onde o programa deve fazer o desvio condicional caso beq seja verdadeiro.

3 Testes

Para validar com eficiência o funcionamento das instruções nos testes, foram inseridas instruções que dependiam das novas instruções implementadas. Além disso, verificamos o passo a passo do caminho de dados por meio da visualização das instruções em cada estágio no processador.

3.1 Testando o mul

Para verificar a corretude da instrução implementada, foi executado o código de teste a seguir.

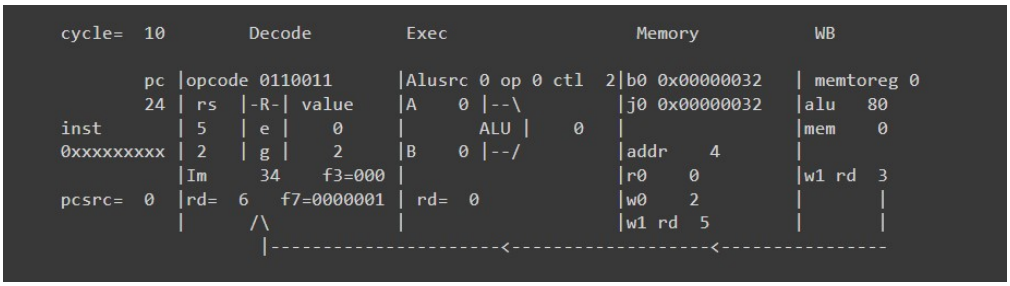
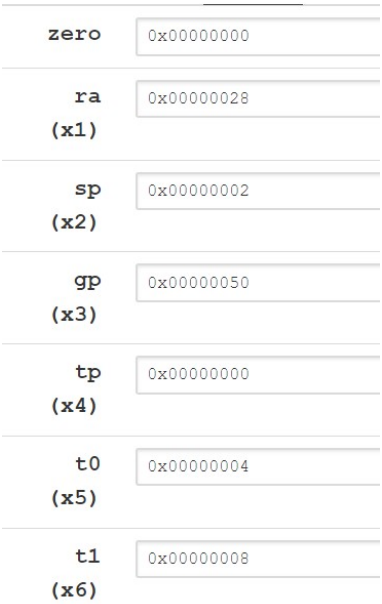


Figura 3: Visualizando o caminho de dados

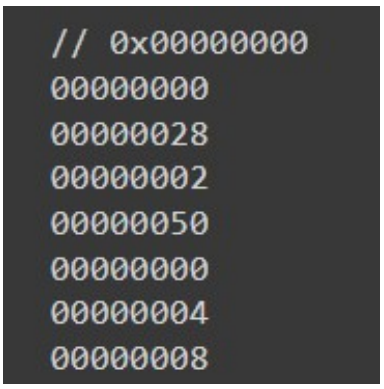
```
1 nop
2 addi x1, x0, 40
3 addi x2, x0, 2
4 mul x3, x1, x2
5 mul x5, x2, x2
6 mul x6, x5, x2
```

Figura 4: Instruções de teste

Executamos as intruções no Venus e no Notebook, e comparamos o resultado usando o reg.data para confirmar se os resultados estavam de acordo.



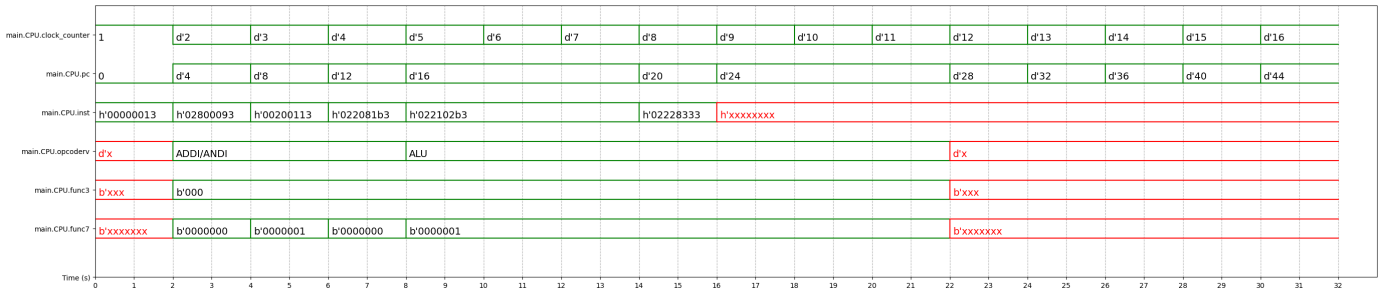
(a) Resultado no Venus



(b) Resultado no notebook

Como esperado, o código funcionou sem erros. Também é possível checar a consistência da operação na forma de onda abaixo.

A waveform demonstra que ao executar uma instrução mul, o valor de funct7 de funct3 estão de acordo com o esperado.



3.2 Testando div

A seguir, verifica-se o código de teste utilizado.

```
1 nop
2 addi x1, x0, 40
3 addi x2, x0, 2
4 div x3, x1, x2
5 div x5, x3, x2
6 addi x1, x0, 11
7 div x6, x1, x2
```

Figura 6: Instruções de teste

Novamente, foram executadas as intruções no Venus e no Notebook, e comparados os resultados usando o reg.data para checar se estavam de acordo.

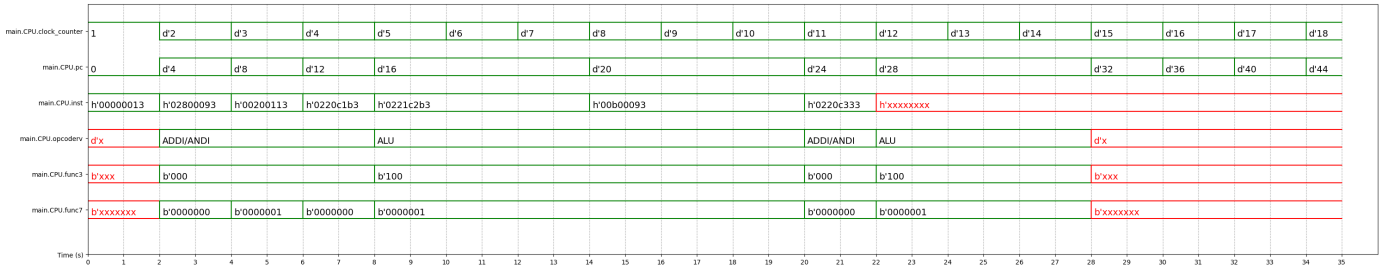
zero	0x00000000
ra (x1)	0x0000000b
sp (x2)	0x00000002
gp (x3)	0x00000014
tp (x4)	0x00000000
t0 (x5)	0x0000000a
t1 (x6)	0x00000005

(a) Resultado no Venus

```
00000000
0000000b
00000002
00000014
00000000
0000000a
00000005
```

(b) Resultado no notebook

Também foi gerada a waveform do código de teste, com os campos de funct3 e funct7. Seus resultados infatizam a conclusão que a instrução foi corretamente implementada.



3.3 Testando andi

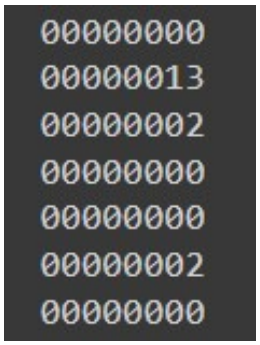
A seguir, verifica-se o código de teste utilizado.

```
1 addi x1, x0, 19
2 andi x2, x1, 14
3 andi x5, x2, 38
4 andi x6, x5, 45
```

Figura 8: Instruções de teste

Novamente, foram executadas as intruções no Venus e no Notebook, e comparados os resultados usando o reg.data para veficar se estavam de acordo.

(a) Resultado no Venus

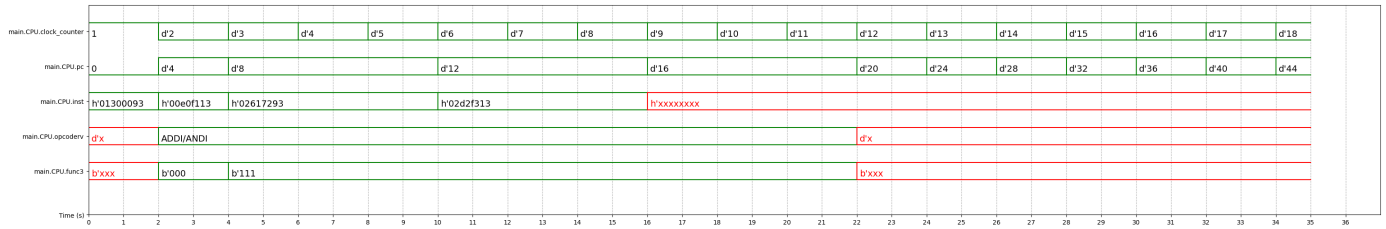


(b) Resultado no notebook

Foi gerado o waveform do código de teste com o campo de funct3 adicionado, para exibir a diferença entre o addi e o andi. O waveform saiu como esperado, garantindo que a instrução foi corretamente implementada.

3.4 Testando beq

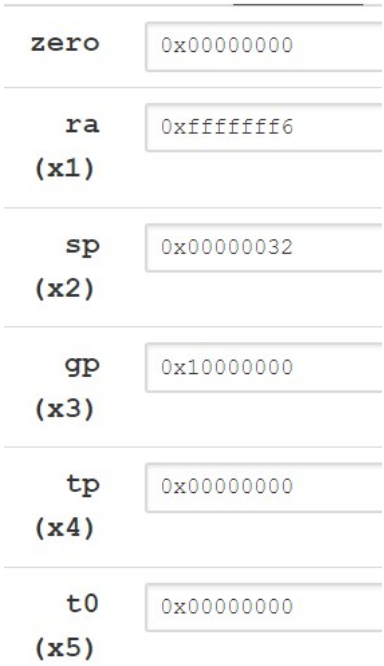
A seguir, verifica-se o código de teste utilizado.



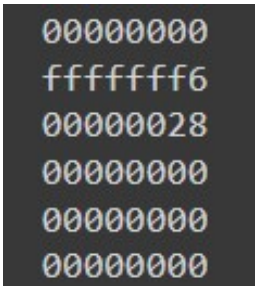

```
1  nop
2  addi x1, x0, 40
3  addi x2, x0, 40
4  beq x2, x1, teste1
5  addi x5, x0, 1  #se teste falhar, oq não deve ocorrer,
6                  # x5 no final será 1
7  teste1: addi x1, x0, 30
8  beq x1, x2, teste2
9  sub x1, x1, x2
10 teste2: addi x2, x2, 10
```

Figura 10: Instruções de teste

Foram executadas as intruções no Venus e no Notebook, e comparados os resultados usando o reg.data para veficar se estavam de acordo.

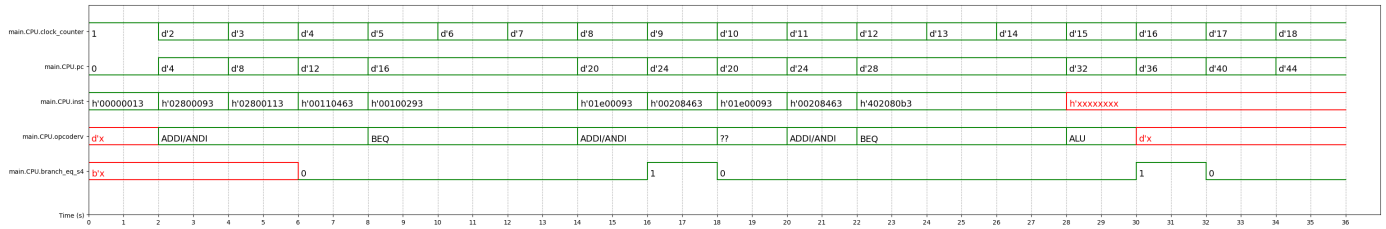


(a) Resultado no Venus



(b) Resultado no notebook

Por fim, foi gerado o waveform do código, com o sinal de branch_eq adicionado, para verificar se sua leitura estava sendo feita de forma correta. Novamente, os resultados gerados foram de acordo com o esperado, reiterando a corretude operacional da instrução implementada.



4 Conclusão

Em conclusão, este trabalho foi uma experiência enriquecedora para o meu aprendizado em Verilog e na implementação de um processador RISCV de 5 estágios. Ao longo do trabalho, pude aprofundar meus conhecimentos na linguagem e aplicá-los na prática.

A implementação das novas instruções exigiu um estudo cuidadoso da documentação do RISC-V e a compreensão dos conceitos subjacentes. Ao enfrentar os desafios de modificar o caminho de dados existente, aprendi a tomar decisões de projeto fundamentadas, considerando aspectos como a sincronização dos estágios, os sinais de controle e as operações de registradores.

Em suma, este trabalho proporcionou um desafio estimulante, permitindo a aplicação dos conhecimentos teóricos adquiridos em sala de aula em um contexto prático. Através da implementação das novas instruções e da análise dos resultados obtidos, pude consolidar meus conhecimentos em Verilog e aprimorar minhas habilidades em projetos de circuitos digitais.

Referências

Material disponibilizado em sala.