Trabalho Prático 2

Organização de Computadores I

Arthur Nader (2019022294)
Isabel Elise (2020006639)
Mathias Oliveira (2020006884)

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

1. Introdução

Junto do enunciado do trabalho prático 02 foi anexada uma implementação em Verilog do caminho de dados de um processador com 5 estágios de pipeline, cuja arquitetura reconhecida é a RISC-V.

Dessa forma, a proposta do trabalho prático 02 consiste em estudar o caminho de dados fornecido e alterá-lo para que esse suporte mais instruções. Mais precisamente, as instruções Bitwise or immediate (ORI), Shift left logical immediate (SLLI) e Branch on less than (BLT) deveriam ser adicionadas ao caminho de dados do processador apresentado.

2. Instruções Tipo-I

O caminho de dados do processador apresentado reconhecia apenas uma instrução do Tipo I, a instrução addition with immediate (ADDI). Para adicionar outras instruções Tipo-I, foi necessário alterar o comportamento dos módulos responsáveis por decodificar os sinais de controle que serão utilizados para realizar o processamento das instruções do Tipo-I. Os módulos alterados foram: o controle (control), o controle da ALU (alu_control) e a ALU (alu).

Embora as instruções do Tipo-R e Tipo-I possuam estruturas muito parecidas, existem algumas diferenças na representação binária dessas instruções que dificultam tratá-las de modo homogêneo no caminho de dados.

Por exemplo, a instrução add (Tipo-R) possui uma instrução correspondente que recebe como operandos um registrador e um imediato de 12 bits, a instrução addi (Tipo-I). Entretanto, a instrução sub (Tipo-R) não possui uma instrução do Tipo-I correspondente. Ademais, o 2º bit mais significativo do campo funct7 das instruções Tipo-R é o que diferencia as instruções add e sub, esse bit pertence ao campo correspondente ao imediato da instrução addi. Esse comportamento está ilustrado na Figura 1.

funct7	rs2	rs1	funct3	rd	opcode	R-type
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:0]		rs1	000	rd	0010011	ADDI

Figura 1 - Padrão de instruções Tipo-R e Tipo-I

Isso impossibilita tratar as instruções add e addi de forma uniforme, pois caso o bit que diferencia as instruções add e sub seja 1, a decodificação de uma instrução addi, identificaria que a operação que deve ser executada pela ALU é a operação de subtração e não a de adição, ou seja, a decodificação da instrução seria feita de forma incorreta. Portanto, para contornar esse problema de forma prática e escalável, optou-se por tratar de forma separada as decodificações das instruções do Tipo-R e das instruções do Tipo-I.

Para diferenciar instruções Tipo-R de instruções Tipo-I, o código 11 do sinal aluop foi destinado especificamente para instruções do tipo I. Vale ressaltar que, essa alteração ocorreu dentro do módulo control. Anteriormente, este código era utilizado de forma redundante para representar a operação de adição. Dessa forma, para evitar uma profunda reformulação no caminho de dados apresentado e implementar as novas instruções de forma escalável, foi feita a seguinte alteração no caminho de dados apresentado:

```
case (opcode)
 7'b0000011: begin // lw == 3
      alusrc   <= 1'b1;
      aluop[1:0]  <= 2'b00;
      memtoreg <= 1'b1;
      regwrite <= 1'b1;
      memread  <= 1'b1;
      ImmGen   <= {{20{inst[31]}},inst[31:20]};
    end
  7'b0010011: begin /* Instruções Tipo-I */
    //regdst   <= 1'b0; // rt or rd (only mips)
    /* aluop = 11 passa a ser o código correspondente às instruções Tipo-I */
    aluop[1:0] <= 2'b11;
    alusrc   <= 1'b1;
    ImmGen   <= {{20{inst[31]}},inst[31:20]};
  end
  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;
  end
  7'b0100011: begin /* sw */
    memwrite <= 1'b1;
    aluop[1] <= 1'b0;
    alusrc   <= 1'b1;
    regwrite <= 1'b0;
    ImmGen   <= {{20{inst[31]}},inst[31:25],inst[11:7]};
  end
  7'b0110011: begin /* add */
  end
  6'b000010: begin  /* j jump */
    jump <= 1'b1;
  end
endcase
```

Figura 2 - Alterações implementadas no módulo control

Ademais, foi necessário alterar o módulo alu_control com o propósito de propagar pelo caminho de dados as mudanças realizadas no módulo control. No módulo alu_control a

decodificação de instruções Tipo-R e Tipo-I foi feita de modo independente. Para implementar essa decisão de projeto, o antigo sinal _funct foi separado em dois outros sinais _funct_r e _funct_i que selecionam, respectivamente, as operações correspondentes às instruções do Tipo-R e do Tipo-I.

Junto a isto, para evitar o problema de decodificação das instruções do Tipo-I discutido previamente, apenas o campo funct3 foi considerado durante o processamento das instruções do Tipo-I realizado pelo módulo alu_control. As alterações estão capturadas na imagem abaixo.

```
// Map do funct3 para operações da ALU
// Esse funct é um funct7[5],funct3[2:0]
reg [3:0] _funct_r;
reg [3:0] _funct_i;

// Trata as operações do Tipo R
always @(*) begin
  case(funct[3:0])
    4'd0:  _funct_r = 4'd2; /* add */
    4'd8:  _funct_r = 4'd6; /* sub */
    4'd5:  _funct_r = 4'd1; /* or */
    4'd6:  _funct_r = 4'd13;  /* xor */
    4'd7:  _funct_r = 4'd12;  /* nor */
    4'd10: _funct_r = 4'd7; /* slt */
    default: _funct_r = 4'd0;
  endcase
end

// Trata as operações do Tipo-I
always @(*) begin
  case(funct[2:0])
    // Aqui vamos considerar apenas o funct 3 para fins de simplicidade
    3'b001: _funct_i = 4'd3; // slli
    3'b000: _funct_i = 4'd2; // addi
    3'b110: _funct_i = 4'd1; // ori
    default: _funct_i = 4'd2;
  endcase
end

always @(*) begin
  case(aluop)
    2'd0: aluctl = 4'd2;  /* add LOAD/STORE */
    2'd1: aluctl = 4'd6;  /* sub BEQ */
    2'd2: aluctl = _funct_r; /* Instrução Tipo-R */
    2'd3: aluctl = _funct_i;  /* TIPO-I */
    default: aluctl = 0;
  endcase
```

Figura 3 - Alterações implementadas no módulo alu_control

Dessa forma, com as alterações implementadas, foi possível adicionar novas instruções do Tipo-I ao caminho de dados apresentado de forma funcional e escalável.

2.1. ORI - Bitwise or immediate

É possível observar que a ALU implementada no caminho de dados disponibilizado já suportava a operação de Bitwise or. Uma vez que o caminho de dados já conduzia

corretamente os operandos até a ALU, foi necessário apenas atribuir o código da operação de Bitwise or ao sinal que define a operação que será executada pela ALU (_funct_i). Essa atribuição está ilustrada na Figura 3. Dessa forma a operação ORI foi adicionada ao caminho de dados de forma funcional.

2.2. SLLI - Shift Left Logical Immediate

Dado que, a ALU do caminho de dados disponibilizados não possuía suporte à operação de Shift left logical (SLL), foi necessário adicionar essa operação à ALU. Para tal, foi necessário definir a correspondência entre o código de controle 0011 da ALU e a operação SLL, essa modificação se encontra na Figura 4. Ademais, foi necessário alterar o módulo alu_controle para que esse selecionasse a operação correta para a ALU executar ao processar uma instrução de SLLI. Para implementar esse comportamento foi necessário adicionar uma nova possível atribuição de valor do sinal _funct_i, essa alteração está ilustrada na Figura 3.

```
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'b0011: out <= (a << b[4:0]); /* sll */
    default: out <= 0;
  endcase
end
```

Figura 4 - Alterações no módulo alu

3. BLT - Branch on Less Than

Instruções do tipo Branch comparam dois registradores e dependendo do resultado da comparação alteram o ponto de execução do programa. A instrução BLT toma o branch se o valor do registrador rs1 é menor que o valor do registrador rs2. No código em verilog que implementa o pipeline, a comparação entre os registradores é feita por meio de uma operação de subtração no terceiro estágio. Se o sinal do resultado for negativo, significa que rs1 - rs2 < 0 e, portanto, rs1 < rs2, o que indica que o branch deve ser tomado. Portanto, se o bit de sinal da variável que guarda o resultado da ALU (alurslt) for 1, então o sinal que indica o branch on less than (branch_lt) é passado para 1. Depois disso, o valor do PC é incrementado de acordo com o imediato passada no código da instrução.

O primeiro problema encontrado na implementação da instrução Branch on less than se refere ao valor do imediato quando se usa números inteiros para indicar o desvio. O assembler utilizado não suporta esses valores. Isso pôde ser percebido dado que, ao mudar o valor do imediato, nenhuma alteração ocorria no código de máquina gerado, tal como pode ser observado na Figura 5 a seguir:

```
[250] %%writefile simple.s
      blt x0, x0, 8
      blt x0, x0, 16
      blt x0, x0, 128
      blt x0, x0, 1024

      Overwriting simple.s

[251] !rm -f -r simple/*
      assemble()

      ------Writing to Text file------
      Output file: simple.txt
      Number of instructions: 4

[252] !cat simple/txt/simple.txt

      01111100000000000100101111100011
      01111100000000000100101111100011
      01111100000000000100101111100011
      01111100000000000100101111100011
```

Figura 5 - Códigos de máquina de instruções blt com imediato inteiro gerados pelo assembler disponibilizado

A solução encontrada para esse erro foi a utilização de “labels” ao invés de inteiros, como a label “fim” utilizada no código abaixo:

```
blt x0, x0, fim
addi x1, x0, 4
addi x2, x0, 8
addi x3, x0, -1
add x4, x2, x3
sub x5, x2, x1
fim:
```

Figura 6 - Instrução blt com desvio indicado por uma label

Com essa alteração, os branches se comportam como o esperado, direcionando o pc para as instruções corretas quando o desvio é tomado. Entretanto, mesmo após essa correção, labels que se referem a posições anteriores da memória em relação à atual geram o mesmo erro de código de máquina que ocorre quando se usa inteiros.

Outro problema encontrado foi no fio branch_lt_s2, que, após uma instrução blt, não muda de nível lógico alto (1) para nível lógico baixo (0) nas próximas instruções a serem executadas. Isso leva a erros no program counter (pc) e no conteúdo dos registradores.

Para melhor explicação, analisa-se o seguinte código assembly:

```
blt x0,x0,fim
addi x1, x0, 4
addi x2, x0, 8
addi x3, x0, -1
add x4, x2, x3
sub x5, x2, x1
fim:
```

Código 1 - Exemplo de erro de execução

Percebe-se que o desvio não deve ser tomado, e, espera-se que ao final da execução os registradores tenham os seguintes valores:

```
x1 =  4    (0x00000004)
x2 =  8    (0x00000008)
x3 = -1    (0xffffffff)
x4 =  7    (0x00000007)
x5 =  4    (0x00000004)
```

Os seguintes sinais de onda são gerados durante a execução:

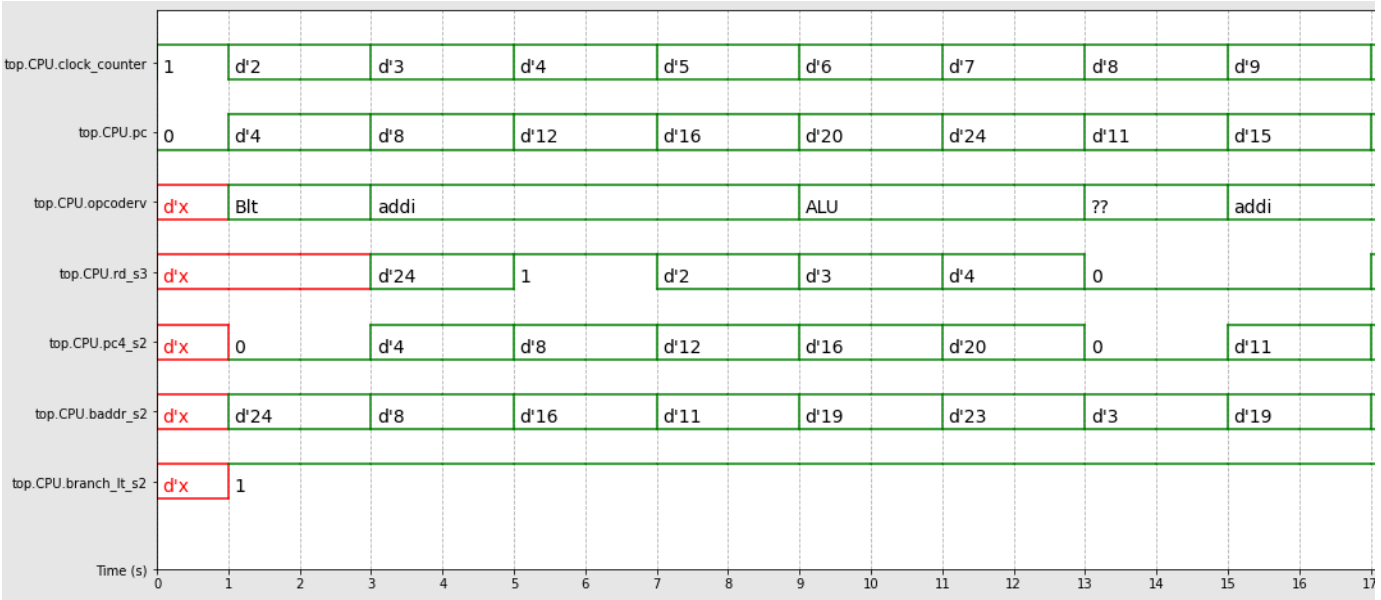

Figura 7 - Sinais de onda da execução do código

Nota-se que, como dito anteriormente, o branch_lt_s2 continua em nível alto nas instruções seguintes ao blt e, em dado instante, o program counter (pc) pula de 24 para 11. Dessa forma, percebe-se que o processador tratou uma das instruções aritméticas como se fosse um desvio, o que acabou levando a resultados errôneos.

Os valores dos registradores ao final da execução são diferentes do esperado, tal qual mostra a Figura 8 a seguir:

```
[79] !cat reg.data

// 0x00000000
00000000
00000004
00000008
ffffffff
00000004
00000005
00000006
00000007
00000008
00000009
0000000a
0000000b
0000000c
0000000d
0000000e
0000000f
```

Figura 8 - Conteúdo dos registradores depois da execução. Percebe-se que os registradores x4 e x5 possuem valores diferentes do esperado.

Para consertar esse erro, foi necessário atribuir o valor da variável branch_lt para 0 na Unidade de Controle. Dessa forma, antes de cada instrução ser decodificada, o branch_lt será 0. Assim, esse valor só passará a ser 1 se a instrução realmente for um Branch less on than. A Figura 9 mostra o comando de atribuição acrescentado.

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

Figura 9 - Alteração que desativa o sinal antes da decodificação de um nova instrução

Desse modo, após a adição dessa linha, o mesmo código assembly gera os seguintes sinais de onda:

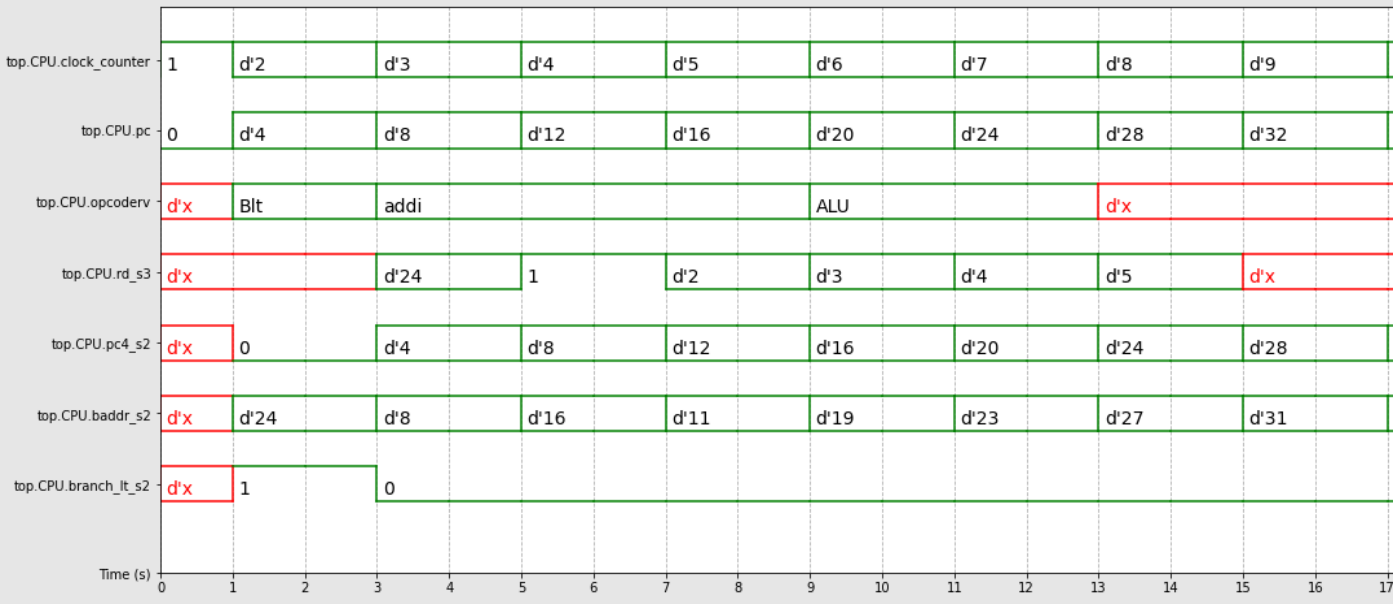

Figura 10 - Sinais de onda da execução do código após alteração

Notamos que o sinal branch_lt_s2, que recebe o sinal de branch_lt na mudança de estágio, agora volta ao valor 0 após o término da identificação da instrução blt. Assim, o program counter (pc) se comporta adequadamente e os valores dos registradores ao final da execução são exatamente os esperados, como pode ser visto na Figura 11 a seguir:

```
[85] !cat reg.data

// 0x00000000
00000000
00000004
00000008
ffffffff
00000007
00000004
00000006
00000007
00000008
00000009
0000000a
0000000b
0000000c
0000000d
0000000e
0000000f
```

Figura 11 - Valores dos registradores após alteração do código

Com isso, conclui-se que a alteração aplicada proporcionou o funcionamento esperado da instrução Branch on less than, não tomando o branch quando o valor de rs1 ≥ o valor de rs2.

4. Conclusão

Para o desenvolvimento do trabalho proposto, foi necessário exercitar os conceitos sobre processamento em pipeline vistos em aula. Com isso, a maior dificuldade se encontrou na interpretação do código em Verilog: identificar os diferentes estágios do pipeline, traçar os caminhos percorridos por cada tipo de instrução e elaborar testes para a depuração.

Após um estudo aprofundado do código, foi possível entender melhor a implementação do processador e a organização das unidades que compõem o caminho de dados. Dessa forma, foram desenvolvidas soluções para as instruções solicitadas. Essas envolveram tanto alterações simples como a atualização do sinal branch_lt, no caso da instrução de BLT, quanto alterações mais complexas que mudam o modo como a unidade de controle processa as instruções do Tipo-I, como no caso do ORI e SLLI.

Vale ressaltar que, os testes que ilustram o funcionamento das instruções implementadas estão no notebook entregue junto ao relatório. Por fim, todas as instruções solicitadas puderam ser implementadas de maneira funcional, de acordo com a documentação do RISC-V, e assim, cumprindo o objetivo proposto.

5. Referências

- The RISC-V Instruction Set Manual. Acessado em: 21/01/2022