Trabalho Prático 2 - Verilog
Organização de Computadores 1

Luisa Vasconcelos de Castro Toledo - 2020006795
Maria Luiza Leão Silva - 2020100953
Raissa Miranda Maciel - 2020006965

Problema 1: ANDI - Bitwise or immediate

A instrução “andi” compara o valor de um registrador (rs1) e um imediato (imm) bit a bit. Caso ambos sejam 1, o bit correspondente no registrador de resultado (rd) é 1, caso contrário, ele é 0. O formato para o RISC-V é mostrado a seguir:

imm[11:0]	rs1	funct3	rd	opcode
Valor imediato	Registrador 1	111	Registrador destino	0010011

A ALU implementada já possuía a instrução andi com seu respectivo funct3 na célula responsável pela ALU e controle da ALU. Foi necessário apenas identificá-la como mostrado abaixo:

```
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 */
      4'd6:  _funct = 4'd1;   /* or */
      4'd5:  _funct = 4'd8;   /* srli */
      4'd4:  _funct = 4'd8;   /* bgt */
      4'd9:  _funct = 4'd13;  /* xor */
      4'd7:  _funct = 4'd0;   /* andi */
      4'd10: _funct = 4'd7;   /* slt */

      default: _funct = 4'd0;

    endcase
  end
```

Para testar a funcionalidade desta instrução, foram criados 4 testes. Inicialmente, é armazenado um valor nos registradores x7, x8, x9 e x10 e, em seguida, o resultado da instrução ANDI com um imediato é alocado nos registradores destino x1, x2, x3 e x4. Cada teste possui comentado os valores em binário e o resultado esperado após a operação bit a bit. É possível verificar os valores dos registradores após a execução dos testes em hexadecimal na imagem ao lado, onde cada linha corresponde a um registrador em ordem crescente a partir do x0.

```
[ ]  %%writefile simple.s

     # TESTE 1
     addi x7, x0, 19        # x7  = 10011
     andi x1, x7, 14        # Imm = 01110
                            # x1  = 00010

     # TESTE 2
     addi x8, x0, 27        # x8  = 11011
     andi x2, x8, 10        # Imm = 01010
                            # x2  = 01010

     # TESTE 3
     addi x9, x0, 17        # x9  = 10001
     andi x3, x9, 13        # Imm = 01101
                            # x3  = 00001

     # TESTE 4
     addi x10, x0, 89       # x10 = 1011001
     andi x4, x10, 23       # Imm = 0010111
                            # x4  = 0010001
```

```
[90] # Final Register file
     !cat reg.data

     // 0x00000000
     00000000
     00000002
     0000000a
     00000001
     00000011
     00000005
     00000006
     00000013
     0000001b
     00000011
     00000059
     0000000b
     0000000c
     0000000d
     0000000e
     0000000f
```

Problema 2: SRLI - Shift Right Logical Immediate

A instrução SRLI realiza uma operação de deslocamento para a direita do valor presente no rs1, preenchendo os bits da esquerda com 0's. O resultado após o deslocamento é armazenado no registrador destino (rd). Realizar um deslocamento é equivalente a dividir o valor do registrador por 2^{imm}, uma vez que os valores são representados na base binária. O formato dessa instrução para o RISC-V é mostrado a seguir:

funct5	funct2	rs2	rs1	funct3	rd	opcode
00000	00	shamt[4:0]	Registrador 1	101	Registrador destino	0010011

A ALU implementada não possuía a operação de deslocamento. Dessa forma, foi adicionada no módulo ALU a implementação da instrução SRLI. Em seguida, assim como no problema anterior, foi necessário identificá-la no módulo de controle da ALU utilizando o valor 101 para o funct3.

```
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;               /* bgt */
    4'd8: out <= a >> b;              /* srli */

    default: out <= 0;
  endcase
```

```
always @(*) begin
  case(funct[3:0])
    4'd0:  _funct = 4'd2;   /* add */
    4'd8:  _funct = 4'd6;   /* sub */
    4'd6:  _funct = 4'd1;   /* or */
    4'd5:  _funct = 4'd8;   /* srli */
    4'd4:  _funct = 4'd8;   /* bgt */
    4'd9:  _funct = 4'd13;  /* xor */
    4'd7:  _funct = 4'd0;   /* andi */
    4'd10: _funct = 4'd7;   /* slt */

    default: _funct = 4'd0;
```

Além disso, como a operação ANDI e SRLI possuem o mesmo opcode, como mostrado nas tabelas de formato anteriormente, também foi necessário alterar o módulo da Unidade de Controle. A diferenciação das duas instruções é feita pelo valor do funct3 e foi implementada da seguinte forma:

```
7'b0010011: begin       /* addi */
if(f3 == 3'b101) begin /* srli */
  ImmGen   <= {{27{inst[24]}},inst[24:20]};
end
else begin              /* andi */
   ImmGen   <= {{20{inst[31]}},inst[31:20]};
end
  //regdst   <= 1'b0; // rt or rd (only mips)
  alusrc   <= 1'b1;
```

Para testar a funcionalidade desta instrução, foram criados 3 testes. Inicialmente, é armazenado um valor nos registradores x1, x3 e x5. Em seguida, a instrução SRLI realiza o deslocamento de n bits, sendo n o imediato especificado na instrução. O resultado após o deslocamento é armazenado nos registradores x2, x4 e x6, respectivamente. Cada teste possui comentado o valor inicial em binário e o resultado esperado após a realização da operação de deslocamento. É possível verificar os valores dos registradores após a execução dos testes em hexadecimal na imagem ao lado, onde cada linha corresponde a um registrador em ordem crescente a partir do x0.

```
[91] %%writefile simple.s

     # TESTE 1
     addi x1, x0, 14     # x1 = 1110
     slri x2, x1, 2      # x2 = 0011

     # TESTE 2
     addi x3, x0, 213    # x3 = 11010101
     slri x4, x3, 4      # x4 = 00001101

     # TESTE 3
     addi x5, x0, 693    # x5 = 1010110101
     slri x6, x5, 8      # x6 = 0000000010
```

```
[94] # Final Register file
     !cat reg.data

     // 0x00000000
     00000000
     0000000e
     00000003
     000000d5
     0000000d
     000002b5
     00000002
     00000007
     00000008
     00000009
     0000000a
     0000000b
     0000000c
     0000000d
     0000000e
     0000000f
```

Problema 3: J - Jump

A instrução JUMP realiza um salto incondicional para um novo endereço, somando o valor do imediato passado na instrução ao endereço do PC. O formato é mostrado a seguir:

Imm[31:12]	rd	opcode
Valor imediato	Registrador destino	1101111

Para isso, foi alterado o módulo da Unidade de Controle, utilizando o valor correto do opcode e adicionando o valor imediato multiplicado por 2 de acordo com a implementação a seguir:

```
7'b1101111: begin /* j jump */
  jump <= 1'b1;
  ImmGen   <= {{11{inst[31]}},inst[31],inst[20],inst[30:21],inst[19:12],1'b0};
end
```

Também foi preciso modificar o valor do PC para PC + Imediato no Código de Decodificação da seguinte forma:

```
assign imm      = inst_s2[15:0];
assign shamt    = inst_s2[10:6];
assign jaddr_s2 = {pc[31:28], inst_s2[31:12],1'b0};
assign seimm  = {{16{inst_s2[15]}}, inst_s2[15:0]};
```

Para testar a funcionalidade desta instrução, foram criados 2 testes. O primeiro teste, inicialmente, armazena dois valores nos registradores x1 e x2 e, em seguida, realiza um jump para convenção SUM, responsável por somar esses dois valores. Caso não fosse feito o desvio, os registradores x1 e x2 teriam sido incrementados de 1. É possível verificar que o desvio foi realizado ao analisar o valor em hexadecimal presente nos registradores após a realização desse teste, onde cada linha corresponde a um registrador em ordem crescente a partir do x0. A seguir é mostrado o código e o resultado após a execução e comprova que os registradores x1 e x2 seguiram o caminho do desvio e não foram incrementados.

```
[160] %%writefile simple.s

      # TESTE 1
      addi x1, x0, 15      # x1 = 15
      addi x2, x0, 12      # x2 = 12
      j SUM
      addi x1, x1, 1
      addi x2, x2, 1
      SUM:
        add x3, x1, x2     # x3 = x1 + x2
```

```
# Final Register file
!cat reg.data
```

```
// 0x00000000
00000000
0000000f
0000000c
0000001b
00000002
00000007
0000000c
00000007
00000008
00000009
0000000a
0000000b
0000000c
0000000d
0000000e
0000000f
```

O segundo teste, inicialmente, armazena valores nos registradores x4 e x5 e, em seguida, realiza o desvio para a convenção SOMA1, onde x4 é somado ao imediato 10. Caso o desvio não fosse corretamente executado, o registrador x6 receberia valores diferentes do especificado pelo SOMA1. Ao analisar os valores em hexadecimal presentes nos registradores após toda a execução do teste, é possível perceber que x6 recebeu os valores corretos.

```
# TESTE 2
addi x4, x0, 2      # x4 = 2
addi x5, x0, 7      # x5 = 7
j SOMA1
add x6, x4, x5      # x6 = x4 + x5
SOMA1:
  addi x6, x4, 10   # x6 = x4 + 10
  j END             # vai para o final
SOMA2:
  addi x6, x6, 100
END:
```

```
# Final Register file
!cat reg.data
```

```
// 0x00000000
00000000
0000000f
0000000c
0000001b
00000002
00000007
0000000c
00000007
00000008
00000009
0000000a
0000000b
0000000c
0000000d
0000000e
0000000f
```

Problema 4: BGT - Branch on Greater Than

A instrução BGT realiza um desvio condicional. São comparados dois valores presentes nos registradores rs1 e rs2 e, caso rs1 > rs2, o desvio é realizado. O formato dessa instrução é mostrado a seguir:

Imm[12|10:5]	rs2	rs1	funct3	Imm[4:1|11]	opcode
Valor Imediato	Registrador 2	Registrador 1	100	Valor Imediato	1100011

A ALU implementada não possuía a operação de BGT, assim, a instrução foi adicionada no módulo da ALU além de também ser identificada no módulo do controle da ALU, utilizando o valor do funct3. As modificações são mostradas a seguir:

```
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;               /* bgt */
    4'd8: out <= a >> b;              /* srli */

    default: out <= 0;
  endcase
end
```

```
always @(*) begin
  case(funct[3:0])
    4'd0:  _funct = 4'd2;   /* add */
    4'd8:  _funct = 4'd6;   /* sub */
    4'd6:  _funct = 4'd1;   /* or */
    4'd5:  _funct = 4'd8;   /* srli */
    4'd4:  _funct = 4'd8;   /* bgt */
    4'd9:  _funct = 4'd13;  /* xor */
    4'd7:  _funct = 4'd0;   /* andi */
    4'd10: _funct = 4'd7;   /* slt */

    default: _funct = 4'd0;

  endcase
end
```

Para testar a funcionalidade desta instrução, foram criados 2 testes. O primeiro armazena, inicialmente, valores nos registradores x1 e x2. Em seguida, é realizada a instrução BGT que verifica qual deles possui o maior valor, e, dependendo do resultado, o desvio é realizado para endereços diferentes. Nesse exemplo, como x2 > x1, o CASO1 é executado e o registrador x3 receberá o valor de x2. Caso contrário, x3 receberia o valor de x1. Ao analisar os valores em hexadecimal presentes nos registradores após toda a execução do teste, é possível perceber que x3 recebeu o valor de x2 e a instrução executou corretamente. A seguir é mostrado o teste e os valores nos registradores em ordem crescente.

```
[90] %%writefile simple.s

    # TESTE 1
    addi x1, x0, 10     # x1 = 10
    addi x2, x0, 20     # x2 = 20

    bgt x2, x1, CASO1  # se (x2>x1) vai para CASO1
    bgt x1, x2, CASO2  # se (x1>x2) vai para CASO2
    j END              # vai para o final

    CASO1:
      add x3, x2, x0   # x3 <- x2
      j END

    CASO2:
      add x3, x1, x0  # x3 <- x1
      j END

    END:
```

```
# Final Register file
!cat reg.data

// 0x00000000
00000000
0000000a
00000014
00000014
00000004
00000005
00000006
00000007
00000008
00000009
0000000a
0000000b
0000000c
0000000d
0000000e
0000000f
```

O segundo teste, armazena inicialmente, valores nos registradores x4 e x5 e verifica se x5 > x4 através da instrução BGT. Caso seja maior, é realizada a instrução AND entre esses dois valores e o resultado é armazenado em x6. Os comentários ao lado do código

apresentam os valores nos registradores em binário e o valor esperado para x6. Caso contrário, o programa deve terminar sem alterar o valor de x6. Após a execução de todo o teste, é possível verificar os valores em hexadecimal nos registradores, sendo um por linha em ordem crescente a partir do x0. Dessa forma, como no exemplo x5 > x4, o valor de x6 está devidamente alterado.

```
[91] %%writefile simple.s

     # TESTE 2
     addi x4, x0, 12      # x4 = 12  (01100)
     addi x5, x0, 23      # x5 = 23  (10111)
     bgt x5, x4, OP       # se (x5>x4) vai para OP
     j END
     OP:
       and x6, x4, x5     # x6 = 00100
     END:
```

```
# Final Register file
!cat reg.data

// 0x00000000
00000000
00000001
00000002
00000003
0000000c
00000017
00000004
00000007
00000008
00000009
0000000a
0000000b
0000000c
0000000d
0000000e
0000000f
```