Centro de Informática (CIn) / Universidade Federal de Pernambuco (UFPE)

Relatório Implementação Risc-V

Infraestrutura de Hardware

Alunos:

- Davi Brilhante (dsb6)
- Davi Dubeux (dld2)
- Eduardo Mabesoone (emm4)
- Gabriel Mayerhofer (gvm)

09 de Agosto de 2024

Sumário

Operações Feitas na ALU

Para realizar as operações na ALU, foi necessário, primeiramente, modificar o código no arquivo **ALU_CONTROLLER**. Esse arquivo contém um módulo em Verilog que decide o output para o registrador de operação, indicando qual operação deve ser feita na ALU. Os inputs desse módulo são:

- O opcode da operação.
- A **func7** e a **func3**, que são, respectivamente, os bits **25 a 31** e **12 a 14** da instrução.

Depois, é utilizado um switch-case para decidir o valor de cada bit do registrador de operação, que será usado na ALU. Inicialmente, o código do arquivo **ALU_CONTROLLER** apenas dava suporte para as instruções de LW, SW, BEQ, ADD e AND. Portanto, foi necessário adicionar valores para as outras operações nos switches, para que no código finalizado seja possível executar todas as operações da ALU.

No código da ALU, um switch-case é utilizado para determinar a operação a ser realizada. A ALU recebe os números a serem usados e o registrador de operação atualizado na **ALU_CONTROLLER** como inputs. O output é um registrador com o resultado desejado.

Classes de Operações

1. **Classe das Aritméticas**
    - **4'b0010: Adição (ADD, ADDI)**
        - **Código:** `ALUResult = $signed(SrcA) + $signed(SrcB);`
        - **Descrição:** Realiza a adição dos valores dos dois registradores de entrada (**SrcA** e **SrcB**). O resultado é armazenado em **ALUResult**.
    - **4'b0011: Subtração (SUB)**
        - **Código:** `ALUResult = $signed(SrcA) - $signed(SrcB);`
        - **Descrição:** Realiza a subtração do valor do registrador SrcB do valor de SrcA. O resultado é armazenado em **ALUResult**.
    - **4'b0111: Shift Arithmetic (SRA)**
        - **Código:** `ALUResult = $signed(SrcA) >>> SrcB[4:0];`
        - **Descrição:** Realiza um shift aritmético à direita no valor de SrcA pelo número de posições especificado por SrcB[4:0]. Este tipo de shift preserva o sinal do número e é utilizado para operações de

divisão por potência de dois e para ajustar o tamanho do número mantendo seu sinal.

2. **Classe das Comparações**
    - **4'b1000: Comparação Igual (BEQ)**
        - **Código:** `ALUResult = (SrcA == SrcB) ? 1 : 0;`
        - **Descrição:** Compara se SrcA é igual a SrcB. Se for verdade, ALUResult é definido como 1; caso contrário, é definido como 0.
    - **4'b1001: Comparação Diferente (BNE)**
        - **Código:** `ALUResult = (SrcA != SrcB) ? 1 : 0;`
        - **Descrição:** Compara se SrcA é diferente de SrcB. Se for verdade, ALUResult é definido como 1; caso contrário, é definido como 0.
    - **4'b1010: Comparação Menor (BLT)**
        - **Código:** `ALUResult = (SrcA < SrcB) ? 1 : 0;`
        - **Descrição:** Compara se SrcA é menor que SrcB. Se for verdade, ALUResult é definido como 1; caso contrário, é definido como 0.
    - **4'b1011: Comparação Maior ou Igual (BGE)**
        - **Código:** `ALUResult = (SrcA >= SrcB) ? 1 : 0;`
        - **Descrição:** Compara se SrcA é maior ou igual a SrcB. Se for verdade, ALUResult é definido como 1; caso contrário, é definido como 0.
    - **4'b1100: Comparação Menor (SLTI)**
        - **Código:** `ALUResult = (SrcA < SrcB) ? 1 : 0;`
        - **Descrição:** Compara se SrcA é menor que SrcB. Se for verdade, ALUResult é definido como 1; caso contrário, é definido como 0.

3. **Classe das Operações Bitwise**
    - **4'b0000: AND**
        - **Código:** `ALUResult = SrcA & SrcB;`
        - **Descrição:** Realiza a operação lógica **AND** bit a bit entre os valores dos registradores **SrcA** e **SrcB**. O resultado é armazenado em **ALUResult**.
    - **4'b0001: OR**
        - **Código:** `ALUResult = SrcA | SrcB;`

        - **Descrição:** Realiza a operação lógica **OR** bit a bit entre os valores dos registradores **SrcA** e **SrcB**. O resultado é armazenado em **ALUResult**.
    - **4'b0100: Shift Left Logical (SLL)**
        - **Código:** `ALUResult = SrcA << SrcB;`
        - **Descrição:** Realiza um deslocamento lógico à esquerda no valor de SrcA pelo número de posições especificado por SrcB. O resultado é armazenado em ALUResult. Esta operação é usada para multiplicar um número por uma potência de dois e para ajustar o valor de bits.
    - **4'b0105: Shift Right Logical (SRL)**
        - **Código:** `ALUResult = SrcA >> SrcB;`
        - **Descrição:** Realiza um deslocamento lógico à direita no valor de SrcA pelo número de posições especificado por SrcB. O resultado é armazenado em ALUResult. Esta operação é usada para dividir um número por uma potência de dois e para ajustar o valor de bits.
    - **4'b0110: XOR**
        - **Código:** `ALUResult = SrcA ^ SrcB;`
        - **Descrição:** Realiza a operação lógica **XOR** bit a bit entre os valores dos registradores **SrcA** e **SrcB**. O resultado é armazenado em ALUResult.

4. **Classe de Operações Imediatas**
    - **4'b1111: Load Upper Immediate (LUI)**
        - **Código:** `ALUResult = SrcB;`
        - **Descrição:** Carrega um valor imediato no registrador ALUResult, que é simplesmente o valor de SrcB. Esta operação é usada para carregar um valor constante superior em um registrador.

Classe de Memória

Para realizar as instruções de **load** e **store**, foi necessário modificar o arquivo **datamemory**. O módulo foi pensado da seguinte forma:

Inputs

- **MemRead** Sinal vindo da unidade de controle para sinalizar a leitura (Loads) da memória
- **MemWrite** - Sinal vindo da unidade de controle para sinalizar a escrita (Stores) na memória
- **a** - Endereço de Leitura / Escrita
- **wd** - Write Data, o valor a ser carregado na memória
- **Funct3** - opcode para sinalizar a instrução

Output

- **rd** - Read Data, valor que foi carregado da memória

O gerenciamento da memória é feito pelo módulo **Memoria32Data**, que é responsável por ler e escrever em 4 blocos de memória.

Na implementação dos **Loads**, um switch case usa a **Funct3** para identificar a instrução de leitura a ser executada e carrega os bits necessários do **Dataout**, passados pelo **Memoria32Data**, no output **rd**.

Na implementação dos **Stores**, outro switch case usa a **Funct3** para identificar a instrução de escrita a ser executada, atualiza a variável **Wr**, que habilita a escrita nos blocos de memória correspondentes à instrução, e carrega o input **wd** nos bits necessários do **Datain**, para o **Memoria32Data** carregá-lo na memória.

Testes e Simulações

Teste Instruções ALU (1)

Instruções testadas: `SLT, SLTI, XOR, SRAI, SRLI, SLLI, ADDI, OR`

Código executado:

```
addi x1,x0,1
addi x2,x0,2
addi x12,x0,8

# SLT (Set Less Than)
slt x3,x1,x2   # x3 = (x1 < x2) ? 1 : 0

# SLTI (Set Less Than Immediate)
slti x4,x1,10 # x4 = (x1 < 10) ? 1 : 0

# XOR (Exclusive OR)
xor x5,x1,x2  # x5 = x1 ^ x2

# SRAI (Shift Right Arithmetic Immediate)
srai x6,x12,3  # x6 = x12 >> 3 (arithmetic)

# SRLI (Shift Right Logical Immediate)
srli x7,x12,3  # x7 = x12 >> 3 (logical)

# SLLI (Shift Left Logical Immediate)
slli x8,x1,3  # x8 = x1 << 3

# ADDI (Add Immediate)
addi x9,x1,10 # x9 = x1 + 10

# OR (Logical OR)
or x10,x1,x2  # x10 = x1 | x2
```

Output esperado nos registradores:

Register	Decimal
x0 (zero)	0
x1 (ra)	1
x2 (sp)	2
x3 (gp)	1
x4 (tp)	1
x5 (t0)	3
x6 (t1)	1
x7 (t2)	1
x8 (s0/fp)	8
x9 (s1)	11
x10 (a0)	3
x11 (a1)	0
x12 (a2)	8

Obtido por meio da execução do código na plataforma
https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/#memory-table

Output obtido nos registradores:

```
 55: Register [ 1] written with value: [00000001] | [          1]
 65: Register [ 2] written with value: [00000002] | [          2]
 75: Register [12] written with value: [00000008] | [          8]
 85: Register [ 3] written with value: [00000001] | [          1]
 95: Register [ 4] written with value: [00000001] | [          1]
105: Register [ 5] written with value: [00000003] | [          3]
115: Register [ 6] written with value: [00000001] | [          1]
125: Register [ 7] written with value: [00000001] | [          1]
135: Register [ 8] written with value: [00000008] | [          8]
145: Register [ 9] written with value: [0000000b] | [         11]
155: Register [10] written with value: [00000003] | [          3]
```

Obtido por meio da geração de instruction.mif usando assembler.py e executado na plataforma ModelSim

Conclusão sobre teste:

Com o teste e comparação feita, conclui-se que todas as instruções abordadas estão funcionando corretamente.

Teste Instruções ALU (2)

Instruções testadas: `SUB, LUI`
Código executado:

```
addi x1,x0,8
sub x6,x6,x1
and x7,x6,x1
lui x6,3
```

Output esperado nos registradores:

Register	Decimal
x0 (zero)	0
x1 (ra)	8
x2 (sp)	0
x3 (gp)	0
x4 (tp)	0
x5 (t0)	0
x6 (t1)	12288
x7 (t2)	8

Obtido por meio da execução do código na plataforma
https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/#memory-table

Output obtido nos registradores:

```
55: Register [ 1] written with value: [00000008] | [          8]
65: Register [ 6] written with value: [fffffff8] | [         -8]
75: Register [ 7] written with value: [00000008] | [          8]
85: Register [ 6] written with value: [00003000] | [      12288]
```

Obtido por meio da geração de instruction.mif usando assembler.py e executado na plataforma ModelSim

Conclusão sobre teste:

Com o teste e comparação feita, conclui-se que todas as instruções abordadas estão funcionando corretamente.

Teste Instruções LOAD (1)

Instruções testadas: `LB, LH`
Código executado:

```
addi x7,x0,1
addi x2,x0,4
or x4,x2,x0
lb x6,0(x7)
add x6,x4,x2
lb x7,0(x6)
lh x8,0(x6)
lw x9,0(x6)
```

Output esperado nos registradores:

```
55: Register [ 7] written with value: [00000001] | [          1]
65: Register [ 2] written with value: [00000004] | [          4]
75: Register [ 4] written with value: [00000004] | [          4]
75: Memory [  1] read with value: [ffffff8f] | [       -113]
85: Register [ 6] written with value: [ffffff8f] | [       -113]
95: Register [ 6] written with value: [00000008] | [          8]
95: Memory [  8] read with value: [fffffffb] | [         -5]
105: Register [ 7] written with value: [fffffffb] | [         -5]
105: Memory [  8] read with value: [ffffaafb] | [     -21765]
115: Register [ 8] written with value: [ffffaafb] | [     -21765]
115: Memory [  8] read with value: [0001aafb] | [     109307]
125: Register [ 9] written with value: [0001aafb] | [     109307]
```

Obtido no README.md do diretório de simulações de LOAD do projeto

Output obtido nos registradores:

```
 55: Register [ 7] written with value: [00000001] | [          1]
 65: Register [ 2] written with value: [00000004] | [          4]
 75: Register [ 4] written with value: [00000004] | [          4]
 75: Memory [  1] read with value: [ffffff8f] | [       -113]
 85: Register [ 6] written with value: [ffffff8f] | [       -113]
 95: Register [ 6] written with value: [00000008] | [          8]
 95: Memory [  8] read with value: [fffffffb] | [         -5]
105: Register [ 7] written with value: [fffffffb] | [         -5]
105: Memory [  8] read with value: [ffffaafb] | [     -21765]
115: Register [ 8] written with value: [ffffaafb] | [     -21765]
115: Memory [  8] read with value: [0001aafb] | [     109307]
125: Register [ 9] written with value: [0001aafb] | [     109307]
```

Obtido por meio da geração de instruction.mif usando assembler.py juntamente com o data.mif na pasta de simulação de load do projeto e executado na plataforma ModelSim

Conclusão sobre teste:

Com o teste e comparação feita, conclui-se que todas as instruções abordadas estão funcionando corretamente.

Teste Instruções LOAD (2)

Instruções testadas: `LBU`

Código executado:

```
addi x7,x0,1
addi x2,x0,4
or x4,x2,x0
lb x6,0(x7)
add x6,x4,x2
lbu x7,0(x6)
```

Output esperado nos registradores:

```
55: Register [ 7] written with value: [00000001] | [          1]
65: Register [ 2] written with value: [00000004] | [          4]
75: Register [ 4] written with value: [00000004] | [          4]
75: Memory [  1] read with value: [ffffff8f] | [       -113]
85: Register [ 6] written with value: [ffffff8f] | [       -113]
95: Register [ 6] written with value: [00000008] | [          8]
95: Memory [  8] read with value: [000000fb] | [        251]
105: Register [ 7] written with value: [000000fb] | [        251]
```

Obtido no README.md do diretório de simulações de LOAD do projeto

Output obtido nos registradores:

```
  55: Register [ 7] written with value: [00000001] | [          1]
  65: Register [ 2] written with value: [00000004] | [          4]
  75: Register [ 4] written with value: [00000004] | [          4]
  75: Memory [  1] read with value: [ffffff8f] | [       -113]
  85: Register [ 6] written with value: [ffffff8f] | [       -113]
  95: Register [ 6] written with value: [00000008] | [          8]
  95: Memory [  8] read with value: [000000fb] | [        251]
 105: Register [ 7] written with value: [000000fb] | [        251]
```

Obtido por meio da geração de instruction.mif usando assembler.py juntamente com o data.mif na pasta de simulação de load do projeto e executado na plataforma ModelSim

Conclusão sobre teste:

Com o teste e comparação feita, conclui-se que todas as instruções abordadas estão funcionando corretamente.

Teste Instruções BRANCHES

Instruções testadas: `BNE, BLT, BGE`
Código executado:

```
addi x5, x0, 9
addi x6, x0, 8

bne x5, x6, 40
beq x5, x6, 56
addi x6, x6, 1
beq x5, x6, 28
bne x5, x6, 52
bge x6, x5, 20
blt x6, x5, 52
addi x6, x6, -1
blt x6, x5, 8
beq x0, x0, 48

addi x28, x0, 1
addi x29, x0, 1
addi x30, x0, 1
addi x31, x0, 1
beq x0, x0, 36

addi x28, x0, 1
beq x0,x0, 28

addi x29, x0, 1
beq x0,x0, 20

addi x30, x0, 1
beq x0,x0, 12

addi x31, x0, 1
beq x0,x0, 4

addi x5, x0, 1
addi x6, x0, 1
```

Output esperado nos registradores:

Register	Decimal
x0 (zero)	0
x1 (ra)	0
x2 (sp)	0
x3 (gp)	0
x4 (tp)	0
x5 (t0)	1
x6 (t1)	1
x7 (t2)	0
x8 (s0/fp)	0
x9 (s1)	0
x10 (a0)	0
x11 (a1)	0
x12 (a2)	0
x13 (a3)	0
x14 (a4)	0
x15 (a5)	0
x16 (a6)	0
x17 (a7)	0
x18 (s2)	0
x19 (s3)	0
x20 (s4)	0
x21 (s5)	0
x22 (s6)	0
x23 (s7)	0
x24 (s8)	0
x25 (s9)	0
x26 (s10)	0
x27 (s11)	0
x28 (t3)	1
x29 (t4)	1
x30 (t5)	1
x31 (t6)	1

Obtido por meio da execução do código na plataforma
https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/#memory-table

Output obtido nos registradores:

```
 55: Register [ 5] written with value: [00000009] | [          9]
 65: Register [ 6] written with value: [00000008] | [          8]
105: Register [28] written with value: [00000001] | [          1]
115: Register [29] written with value: [00000001] | [          1]
125: Register [30] written with value: [00000001] | [          1]
135: Register [31] written with value: [00000001] | [          1]
175: Register [ 5] written with value: [00000001] | [          1]
185: Register [ 6] written with value: [00000001] | [          1]
```

Obtido por meio da geração de instruction.mif usando assembler.py e executado na plataforma ModelSim

Conclusão sobre teste:

Com o teste e comparação feita, conclui-se que todas as instruções abordadas estão funcionando corretamente.

Teste Instruções JAL

Instruções testadas: `JAL`
Código executado:

```
addi x7,x0,1
addi x2,x0,4
jal x10,8
or x4,x2,x0
add x6,x4,x2
addi x7,x0,1
addi x8,x0,2
beq x7,x7,-8
or x4,x2,x0
```

Output esperado nos registradores:

```
 45: Register [ 7] written with value: [00000001] | [         1]
 55: Register [ 2] written with value: [00000004] | [         4]
 65: Register [10] written with value: [0000000c] | [        12]
 95: Register [ 6] written with value: [00000004] | [         4]
105: Register [ 7] written with value: [00000001] | [         1]
115: Register [ 8] written with value: [00000002] | [         2]
155: Register [ 7] written with value: [00000001] | [         1]
165: Register [ 8] written with value: [00000002] | [         2]
205: Register [ 7] written with value: [00000001] | [         1]
215: Register [ 8] written with value: [00000002] | [         2]
255: Register [ 7] written with value: [00000001] | [         1]
265: Register [ 8] written with value: [00000002] | [         2]
305: Register [ 7] written with value: [00000001] | [         1]
315: Register [ 8] written with value: [00000002] | [         2]
355: Register [ 7] written with value: [00000001] | [         1]
365: Register [ 8] written with value: [00000002] | [         2]
405: Register [ 7] written with value: [00000001] | [         1]
415: Register [ 8] written with value: [00000002] | [         2]
455: Register [ 7] written with value: [00000001] | [         1]
465: Register [ 8] written with value: [00000002] | [         2]
```

Obtido no README.md do diretório de simulações de JAL, BEQ do projeto

Output obtido nos registradores:

```
 55: Register [ 7] written with value: [00000001] | [          1]
 65: Register [ 2] written with value: [00000004] | [          4]
 75: Register [10] written with value: [0000000c] | [         12]
105: Register [ 6] written with value: [00000004] | [          4]
115: Register [ 7] written with value: [00000001] | [          1]
125: Register [ 8] written with value: [00000002] | [          2]
165: Register [ 7] written with value: [00000001] | [          1]
175: Register [ 8] written with value: [00000002] | [          2]
215: Register [ 7] written with value: [00000001] | [          1]
225: Register [ 8] written with value: [00000002] | [          2]
265: Register [ 7] written with value: [00000001] | [          1]
275: Register [ 8] written with value: [00000002] | [          2]
315: Register [ 7] written with value: [00000001] | [          1]
325: Register [ 8] written with value: [00000002] | [          2]
365: Register [ 7] written with value: [00000001] | [          1]
375: Register [ 8] written with value: [00000002] | [          2]
415: Register [ 7] written with value: [00000001] | [          1]
425: Register [ 8] written with value: [00000002] | [          2]
465: Register [ 7] written with value: [00000001] | [          1]
475: Register [ 8] written with value: [00000002] | [          2]
```

Obtido por meio da geração de instruction.mif usando assembler.py e executado na plataforma ModelSim

Conclusão sobre teste:

Com o teste e comparação feita, conclui-se que a instrução abordada está funcionando corretamente.

Teste Instruções JALR

Instruções testadas: `JALR`

Código executado:

```
addi x7,x0,-1
sw x7,0(x0)
lw x9,0(x0)
or x4,x2,x0
add x6,x4,x2
jalr x12,x0,12
```

Output esperado nos registradores:

```
55: Register [ 7] written with value: [ffffffff] | [         -1]
55: Memory [  0] written with value: [ffffffff] | [         -1]
65: Memory [  0] read with value: [ffffffff] | [         -1]
75: Register [ 9] written with value: [ffffffff] | [         -1]
85: Register [ 4] written with value: [00000000] | [          0]
95: Register [ 6] written with value: [00000000] | [          0]
105: Register [12] written with value: [00000018] | [         24]
135: Register [ 4] written with value: [00000000] | [          0]
145: Register [ 6] written with value: [00000000] | [          0]
155: Register [12] written with value: [00000018] | [         24]
185: Register [ 4] written with value: [00000000] | [          0]
195: Register [ 6] written with value: [00000000] | [          0]
205: Register [12] written with value: [00000018] | [         24]
235: Register [ 4] written with value: [00000000] | [          0]
245: Register [ 6] written with value: [00000000] | [          0]
255: Register [12] written with value: [00000018] | [         24]
285: Register [ 4] written with value: [00000000] | [          0]
295: Register [ 6] written with value: [00000000] | [          0]
305: Register [12] written with value: [00000018] | [         24]
335: Register [ 4] written with value: [00000000] | [          0]
345: Register [ 6] written with value: [00000000] | [          0]
355: Register [12] written with value: [00000018] | [         24]
385: Register [ 4] written with value: [00000000] | [          0]
395: Register [ 6] written with value: [00000000] | [          0]
405: Register [12] written with value: [00000018] | [         24]
435: Register [ 4] written with value: [00000000] | [          0]
445: Register [ 6] written with value: [00000000] | [          0]
455: Register [12] written with value: [00000018] | [         24]
485: Register [ 4] written with value: [00000000] | [          0]
495: Register [ 6] written with value: [00000000] | [          0]
505: Register [12] written with value: [00000018] | [         24]
```

Obtido no README.md do diretório de simulações de JAL, BEQ do projeto

Output obtido nos registradores:

```
 55: Register [ 7] written with value: [ffffffff] | [        -1]
 55: Memory [  0] written with value: [ffffffff] | [        -1]
 65: Memory [  0] read with value: [ffffffff] | [        -1]
 75: Register [ 9] written with value: [ffffffff] | [        -1]
 85: Register [ 4] written with value: [00000000] | [         0]
 95: Register [ 6] written with value: [00000000] | [         0]
105: Register [12] written with value: [00000018] | [        24]
135: Register [ 4] written with value: [00000000] | [         0]
145: Register [ 6] written with value: [00000000] | [         0]
155: Register [12] written with value: [00000018] | [        24]
185: Register [ 4] written with value: [00000000] | [         0]
195: Register [ 6] written with value: [00000000] | [         0]
205: Register [12] written with value: [00000018] | [        24]
235: Register [ 4] written with value: [00000000] | [         0]
245: Register [ 6] written with value: [00000000] | [         0]
255: Register [12] written with value: [00000018] | [        24]
285: Register [ 4] written with value: [00000000] | [         0]
295: Register [ 6] written with value: [00000000] | [         0]
305: Register [12] written with value: [00000018] | [        24]
335: Register [ 4] written with value: [00000000] | [         0]
345: Register [ 6] written with value: [00000000] | [         0]
355: Register [12] written with value: [00000018] | [        24]
385: Register [ 4] written with value: [00000000] | [         0]
395: Register [ 6] written with value: [00000000] | [         0]

405: Register [12] written with value: [00000018] | [        24]
435: Register [ 4] written with value: [00000000] | [         0]
445: Register [ 6] written with value: [00000000] | [         0]
455: Register [12] written with value: [00000018] | [        24]
485: Register [ 4] written with value: [00000000] | [         0]
495: Register [ 6] written with value: [00000000] | [         0]
505: Register [12] written with value: [00000018] | [        24]
```

Obtido por meio da geração de instruction.mif usando assembler.py juntamente com o data.mif na pasta de simulação de load do projeto e executado na plataforma ModelSim

Conclusão sobre teste:

Com o teste e comparação feita, conclui-se que a instrução abordada está funcionando corretamente.

Teste Instruções STORE

Instruções testadas: `SB, SH`

Código executado:

```
addi x7,x0,0
sb x7,2(x0)
lw x9,0(x0)
sh x7,2(x0)
lw x8,0(x0)
```

Output esperado nos registradores:

```
55: Register [ 7] written with value: [00000000] | [          0]
55: Memory [  2] written with value: [00000000] | [          0]
65: Memory [  0] read with value: [ff00aa80] | [  -16733568]
75: Register [ 9] written with value: [ff00aa80] | [  -16733568]
75: Memory [  2] written with value: [00000000] | [          0]
85: Memory [  0] read with value: [0000aa80] | [      43648]
95: Register [ 8] written with value: [0000aa80] | [      43648]
```

Obtido no README.md do diretório de simulações de STORE do projeto

Output obtido nos registradores:

```
55: Register [ 7] written with value: [00000000] | [          0]
55: Memory [  2] written with value: [00000000] | [          0]
65: Memory [  0] read with value: [ff00aa80] | [  -16733568]
75: Register [ 9] written with value: [ff00aa80] | [  -16733568]
75: Memory [  2] written with value: [00000000] | [          0]
85: Memory [  0] read with value: [0000aa80] | [      43648]
95: Register [ 8] written with value: [0000aa80] | [      43648]
```

Obtido por meio da geração de instruction.mif usando assembler.py juntamente com o data.mif na pasta de simulação de STORE do projeto e executado na plataforma ModelSim

Conclusão sobre teste:

Com o teste e comparação feita, conclui-se que as instruções abordadas estão funcionando corretamente.

Observações:

Para garantir o funcionamento correto das instruções de troca de **PC**, além das instruções especificadas acima, foram necessárias alterações nos módulos de **Branch Unit**, **Controller**, **Datapath**, **RISC_V**, dentre outros. Inicialmente, o pipeline funciona de maneira que, a cada ciclo, o **PC** é incrementado em 4, ou com o offset do branch, mas outras instruções de troca de pc não são consideradas. Para resolver isso, novos sinais de controle foram adicionados ao código, indicando para a **Branch Unit** quando mudar o **PC** incluindo assim as instruções de jump, jalr e halt. O halt, entretanto, acabou não funcionando como o esperado.

Visão Geral - Testes e Simulações

#	Instrução	Implementada	Testada	Funcionando
1	JAL	✅	✅	✅
2	JALR	✅	✅	✅
3	BNE	✅	✅	✅
4	BLT	✅	✅	✅
5	BGE	✅	✅	✅
6	LB	✅	✅	✅
7	LH	✅	✅	✅
8	LBU	✅	✅	✅
9	SB	✅	✅	✅
10	SH	✅	✅	✅
11	SLTI	✅	✅	✅
12	ADDI	✅	✅	✅
13	SLLI	✅	✅	✅
14	SRLI	✅	✅	✅
15	SRAI	✅	✅	✅
16	SUB	✅	✅	✅
17	SLT	✅	✅	✅
18	XOR	✅	✅	✅
19	OR	✅	✅	✅
20	LUI	✅	✅	✅
21	HALT	✅	✅	❌