

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

JOANA D'ARC OLIVEIRA DO NASCIMENTO (JDON)
PAULO VITOR BARBOSA SANTANA (PVBS)

RELATÓRIO DE PROJETO

RECIFE
2024

Índice

Introdução	3
Descrição das Implementações das Instruções por Classe de Instruções	4
R-type	4
I-type	4
B-type	5
I-type (load)	6
S-type	6
J-type	7
U-type	7
Halt	7
Descrição dos Sinais de Controle para cada Classe de Instruções	9
RegWrite	9
MemRead	9
MemWrite	9
MemtoReg	9
ALUSrc	9
ALUOp	10
Branch	10
JarlSel	10
HaltSel	10
WRMux	10
Simulações Realizadas	11
Operações da ALU	11
JAL e JALR	12
Halt	12
Branch	13
BGE	13
BNE	13
BLT	14
Store e Load	14
LUI	15
Conclusão	16

Introdução

O presente relatório tem como objetivo fornecer informações acerca da implementação de instruções do conjunto RV32I na ISA do processador RISC-V com pipeline utilizando SystemVerilog.

O processador RISC-V é uma arquitetura de conjunto de instruções (ISA) baseada em um design simplificado, o que facilita a implementação de suas instruções. As instruções RISC-V são categorizadas em diferentes classes, cada uma com um propósito e formato específico. Neste relatório, será apresentada uma visão geral de como foi realizada a implementação das instruções dentro de cada uma dessas classes, fornecendo uma visão clara sobre suas funcionalidades.

Além de entender as instruções que compõem a arquitetura RISC-V, é crucial compreender o papel dos sinais de controle no pipeline do processador. Esses sinais desempenham uma função fundamental na coordenação e na execução eficiente das instruções dentro do pipeline, garantindo que cada estágio do processamento de uma instrução ocorra na ordem correta e com a operação apropriada. Dessa forma, também iremos abordar a funcionalidade dos sinais de controle.

Para garantir que o processador RISC-V funcione corretamente e que as instruções sejam implementadas conforme o especificado, realizaremos uma análise dos resultados obtidos por meio da simulação. Utilizaremos a ferramenta ModelSim para este propósito, uma vez que é amplamente reconhecida por sua capacidade de simular e verificar circuitos digitais descritos em HDL (Hardware Description Language).

Descrição das Implementações das Instruções por Classe de Instruções

R-type

As instruções R-type realizam operações, lógicas e aritméticas, entre registradores e o resultado é escrito em um outro registrador. Todas as instruções R-type utilizam o mesmo ALUOp, portanto as instruções são diferenciadas pelos campos funct3 e funct7 informados na instrução.

Como o módulo Controller já tinha definido quais sinais de controle seriam utilizados pelas instruções R-type, foi necessário apenas modificar os módulos ALUController e ALU para implementar todas as instruções R-type.

O módulo ALUController recebe como entradas dados que identificam a instrução como o ALUOp, funct7 e funct3. A saída é um vetor de 4 bits chamado Operation, que é enviado para a ALU para que esta possa executar a instrução de acordo com o seu valor. No módulo ALUController definimos o valor de cada bit do vetor Operation através de condicionais utilizando os dados de entrada, dessa forma foi definido o valor Operation para cada operação da ALU, por exemplo, as operações BLT, SLT e SLTI ficaram com o valor 1010, pois executam a mesma comparação (<) na ALU.

O módulo ALU recebe o vetor Operation e os dois operandos (SrcA e SrcB) como entradas e tem o resultado da operação como saída. Nesse módulo foi preciso implementar as operações de acordo com o valor de Operation, por exemplo, se o Operation fosse 1010, ALUResult seria 1 se $\text{SrcA} < \text{SrcB}$ e caso contrário seria 0.

I-type

As instruções I-type realizam operações com operandos imediatos. Todas as instruções I-type (lógicas e aritméticas) utilizam o mesmo ALUOp que das instruções R-type, dessa forma as operações são diferenciadas pelos campos funct3 e funct7 da instrução. O sinal de controle ALUSrc define se o segundo operando da ALU será um valor imediato ou será um valor que está guardado em um registrador.

O módulo `imm_Gen` já tinha implementado o formato do imediato para as instruções de load (I-type), mas não para as outras instruções I-type (lógicas e aritméticas). Estas possuem o mesmo formato do imediato, porém utiliza um opcode diferente, dessa forma incluímos mais um valor no case e repetimos o formato do imediato.

Também foi necessário modificar o módulo `Controller` para definir quais sinais de controle as instruções I-type utilizam, que seriam a `ALUSrc`, o `RegWrite` e o `ALUOp[1]`.

Foi preciso modificar os módulos `ALUController` e `ALU`, de forma semelhante ao que foi feito nas instruções R-type, para adicionar algumas operações, como a operação `SLLI` que ficou com o valor `Operation 0101`. Por outro lado algumas instruções I-type utilizam a mesma operação que algumas R-type, como, por exemplo, `ADD` e `ADDI`, dessa forma não foi preciso adicionar mais uma operação de adição.

B-type

As instruções B-type realizam desvios condicionais durante a execução do código. As diferentes instruções B-type são diferenciadas pelo campo `funct3` informado na instrução.

As operações de branch são gerenciadas pela `BranchUnit`, que determina se o PC será atualizado com `PC+4` para executar a próxima instrução ou se acontecerá algum desvio, nesse caso a `BranchUnit` também define o endereço de desvio, esse endereço é armazenado em `BrPC`, e o valor de `PC+4` em `PC_Four`. O sinal `PcSel` define se o PC é atualizado com o endereço de `PC_Four` ou com o endereço de `BrPC`, através de um mux.

Não foi necessário modificar o módulo `imm_Gen` para as instruções B-type porque o formato do imediato já estava implementado.

Também não foi necessário modificar o módulo `Controller`, porque já estavam definidos os sinais de controle utilizados pelas instruções B-type.

A comparação das instruções de branch são realizadas pela `ALU`, dessa forma modificamos o módulo `ALUController` para definir o valor da `Operation` e colocamos no módulo `ALU` as operações de comparação, como exemplo temos a instrução `BGE` com `Operation 1011`.

I-type (load)

As instruções de load também são I-type, porém executam a operação para carregar dados da memória de dados para os registradores. Além disso, possuem formato do imediato diferente das instruções I-type (lógicas e aritméticas), sendo diferenciadas pelo opcode. O ALUSrc é utilizado para indicar que o segundo operando da ALU será um imediato (para cálculo do endereço da memória que vai ser acessada), RegWrite é utilizado para indicar que o dado lido da memória vai ser escrito na memória e MemtoReg para indicar que o dado vai ser lido da memória e escrito no registrador.

Não foi necessário modificar o módulo imm_Gen para as instruções I-type (load) porque o formato do imediato já estava implementado.

Também não foi necessário modificar o módulo Controller, porque já estavam definidos os sinais de controle utilizados pelas instruções I-type (load).

Foi necessário modificar o módulo datamemory para adicionar os outros tipos de load (inicialmente só LW estava implementado). Dessa forma os tipos de load (LW, LB, LBU e LH) seriam diferenciados pelo campo funct3, e as implementações diferem quanto à quantidade de bits e o sinal estendido.

S-type

As instruções S-type são responsáveis por realizar carregar dados dos registradores para a memória de dados. O ALUSrc é utilizado pelo mesmo motivo das operações de load e MemWrite indica que o dado irá ser escrito na memória.

Também não foi necessário modificar o módulo Controller, porque já estavam definidos os sinais de controle utilizados pelas instruções S-type (load).

Foi necessário modificar o módulo datamemory para adicionar os outros tipos de store (inicialmente só SW estava implementado). Dessa forma os tipos de load (SW, SB e SH) seriam diferenciados pelo campo funct3, e as implementações diferem quanto à quantidade de bits dos dados de escrita seriam armazenados na memória.

J-type

Jal e Jalr são instruções J-type, são utilizadas para saltos incondicionais na execução do código, ou seja, para modificar o fluxo de execução do programa para uma outra posição. A instrução Jal realiza o salto incondicional para o endereço especificado pelo deslocamento (offset) e armazena o endereço da instrução seguinte no registrador, esse endereço é calculado com base no endereço atual. Enquanto a instrução Jalr realiza o salto incondicional para um endereço especificado pela soma do valor de um registrador com um valor imediato e guarda o endereço da instrução seguinte em um registrador.

Ambas instruções utilizam o sinal RegWrite por escrever um endereço em um registrador e a Jalr utiliza o sinal ALUSrc por utilizar valor imediato no cálculo do endereço de salto.

Foi necessário modificar o módulo BranchUnit para adicionar um sinal chamado JalrSel, para indicar a ocorrência de uma instrução Jalr, pois utilizam a ALU para calcular o endereço de desvio com base no valor de registrador e um valor imediato, enquanto que os sinais Branch e AluResult[0] indicam a ocorrência de uma instrução de branch ou Jal. Esses sinais controlam o PcSel que indica se o mux vai ter como saída o BrPC (endereço de desvio) ou Pc_Four (endereço da próxima instrução), atualizando o PC.

U-type

A instrução LUI é U-type, usada para carregar um valor imediato de 20 bits nos bits mais significativos de um registrador, sendo útil para carregar constantes grandes e endereços de memória.

Para implementar a instrução LUI foi necessário definir quais sinais de controle seriam utilizados no módulo Controller e adicionar a instrução nos módulos ALUController e ALU, tendo o mesmo valor de Operation que o JAL.

Halt

O Halt é uma pseudo-instrução utilizada para interromper a execução do código. Para implementar o Halt foi necessário adicionar um sinal de controle chamado HaltSel no módulo Controller, ativado se o opcode da instrução fosse

1111111. Foi necessário adicionar o sinal HaltSel no módulo Datapath. Como a instrução Halt não estava definida no arquivo assembler.py, foi necessário modificá-la para que a instrução fosse traduzida de forma correta.

Descrição dos Sinais de Controle para cada Classe de Instruções

RegWrite

O sinal RegWrite determina se a escrita no registrador deve ser realizada ou não. Se o sinal RegWrite estiver ativado, o processador vai escrever o resultado da operação no registrador de destino. Sendo utilizado por instruções R-type, I-type (lógicas e aritméticas), I-type (load), S-type e J-type.

MemRead

O sinal MemRead determina se uma leitura da memória principal deve ser realizada ou não. Se o sinal MemRead estiver ativado, o processador vai acessar a memória de dados para ler um valor. Sendo utilizado por instruções I-type (load).

MemWrite

O sinal MemWrite determina se uma escrita na memória deve ser realizada ou não. Se MemWrite estiver ativo, vai ser gravado um valor na memória de dados. Sendo utilizado por instruções S-type.

MemtoReg

O sinal MemtoReg indica a origem do dado a ser escrito no registrador destino, o sinal é utilizado pelo resmux, se o valor for 0 o dado vem da ALU, se o valor for 1 o dado vem da memória.

ALUSrc

O sinal ALUSrc determina se a instrução utilizará imediatos ou não, se o sinal estiver desativado o operando SrcB da ALU vem da instrução (R-type), caso contrário o operando é um imediato. Vários tipos de instrução utilizam valores imediatos, dessa forma o módulo imm_Gen fica responsável de selecionar o formato correto de acordo com o opcode da instrução, podendo ser I-type (load), I-type (lógica e aritmética), Jalr, S-type e B-type.

ALUOp

O sinal ALUOp é um vetor de 2 bits, que define qual tipo de instrução a ALU irá executar, este sinal é utilizado pela ALUController para formar o vetor Operation que define a operação a ser realizada na ALU. O sinal ALUOp possui o valor 00 para instruções de acesso à memória (load e store), 01 para instruções de branch, 10 para instruções Rtype/Itype e 11 para instruções Jtype/Utype.

Branch

O sinal Branch é utilizado para indicar a ocorrência de uma instrução de branch, sendo utilizado pela BranchUnit para alterar a posição que o PC aponta de acordo com o offset informado na instrução. Na ALU as diferentes instruções de branch são diferenciadas pelo campo funct3 da instrução.

JarlSel

O sinal JarlSel indica a ocorrência da instrução Jalr. Assim como o sinal Branch, o JarlSel também é utilizado pela BranchUnit para alterar o valor de PC de acordo com o endereço calculado pelo registrador base mais o imediato.

HaltSel

O sinal HaltSel indica a ocorrência da pseudo-instrução Halt, controlando a interrupção de execução do código, zerando o valor de PC, impedindo a execução de novas instruções. O BrPC é zerado e valor de PcSel é 1, então o PC é atualizado com o valor de BrPC que é 0.

WRMux

O sinal WRMux é um vetor de 2 bits que indica para o mux do estágio WB do pipeline qual dado será escrito no registrador, PC+4, PC+Imm, Imm (instrução LUI) ou um dado da memória.

Simulações Realizadas

Operações da ALU

```
addi x10,x0,12
addi x11,x0,35
addi x12,x0,-4
add x13,x10,x11
add x14,x10,x12
sub x15,x11,x10
sub x16,x10,x11
sub x17,x10,x12
and x18,x10,x11
or x19,x10,x11
xor x20,x10,x11
slli x21,x10,2
srli x22,x11,1
srai x23,x11,2
slt x24,x10,x11
slt x25,x10,x12
slti x26,x12,-3
```

```
45: Register [10] written with value: [0000000c] | [      12]
55: Register [11] written with value: [00000023] | [      35]
65: Register [12] written with value: [fffffffc] | [4294967292]
75: Register [13] written with value: [0000002f] | [      47]
85: Register [14] written with value: [00000008] | [       8]
95: Register [15] written with value: [00000017] | [      23]
105: Register [16] written with value: [ffffffe9] | [4294967273]
115: Register [17] written with value: [00000010] | [      16]
125: Register [18] written with value: [00000000] | [       0]
135: Register [19] written with value: [0000002f] | [      47]
145: Register [20] written with value: [0000002f] | [      47]
155: Register [21] written with value: [00000030] | [      48]
165: Register [22] written with value: [00000011] | [      17]
175: Register [23] written with value: [00000008] | [       8]
185: Register [24] written with value: [00000001] | [       1]
195: Register [25] written with value: [00000000] | [       0]
205: Register [26] written with value: [00000001] | [       1]
```

x10 recebe 12 (addi)

x11 recebe 35 (addi)

x12 recebe -4 (addi)

x13 recebe $x10 + x11 = 12 + 35 = 47$ (add)

x14 recebe $x10 + x12 = 12 + (-4) = 8$ (add)

x15 recebe $x11 - x10 = 35 - 12 = 23$ (sub)

x16 recebe $x10 - x11 = 12 - 35 = -23$ (sub)

x17 recebe $x10 - x12 = 12 - (-4) = 16$ (sub)

x18 recebe $x10 \& x11 = 0$ (and)

x19 recebe $x10 | x11 = 47$ (or)

x20 recebe $x10 \wedge x11 = 47$ (xor)

x21 recebe $x10 * 2^2 = 48$ (slli)

x22 recebe $x11 / 2 = 17$ (divisão inteira - srli)

x23 recebe $x11 / 2^2 = 8$ (divisão inteira - srai)

x24 recebe 1, pois $x10 < x11$ ($12 < 35$) (slt)

x25 recebe 0, pois $x10 > x12$ ($12 > -4$) (slt)

x26 recebe 1, pois $x12 < -3$ ($-4 < -3$) (slti)

JAL e JALR

```
addi x7,x0,1
addi x2,x0,4
jal x10,16
or x4,x2,x0
add x6,x4,x2
addi x7,x0,1
addi x8,x0,2
or x4,x2,x0
```

```
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 [ 8] written with value: [00000002] | [ 2]
105: Register [ 4] written with value: [00000004] | [ 4]
```

Ao executar a instrução de jal, x10 guarda o endereço da próxima instrução (12) e faz o jump para o endereço atual + 16 = 24.

```
addi x7,x0,1
addi x2,x0,4
jalr x11,x2,16
or x4,x2,x0
add x6,x4,x2
addi x7,x0,1
addi x8,x0,2
or x4,x2,x0
```

```
45: Register [ 7] written with value: [00000001] | [ 1]
55: Register [ 2] written with value: [00000004] | [ 4]
65: Register [11] written with value: [0000000c] | [12]
95: Register [ 7] written with value: [00000001] | [ 1]
105: Register [ 8] written with value: [00000002] | [ 2]
115: Register [ 4] written with value: [00000004] | [ 4]
```

Ao executar a instrução de jalr, x11 guarda o endereço da próxima instrução (12) e faz o jump para o endereço $x2 + 16 = 4 + 16 = 20$.

Halt

```
addi x7,x0,1
addi x2,x0,4
halt
jalr x11,x2,16
or x4,x2,x0
add x6,x4,x2
addi x7,x0,1
addi x8,x0,2
or x4,x2,x0
```

```
45: Register [ 7] written with value: [00000001] | [ 1]
55: Register [ 2] written with value: [00000004] | [ 4]
```

Ao executar a instrução halt, a execução do código é interrompida.

Branch

BGE

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

```
45: Register [ 7] written with value: [00000001] | [ 1]
55: Register [ 2] written with value: [00000004] | [ 4]
65: Register [ 4] written with value: [00000004] | [ 4]
105: Register [ 2] written with value: [00000004] | [ 4]
115: Register [ 4] written with value: [00000004] | [ 4]
155: Register [ 2] written with value: [00000004] | [ 4]
165: Register [ 4] written with value: [00000004] | [ 4]
205: Register [ 2] written with value: [00000004] | [ 4]
215: Register [ 4] written with value: [00000004] | [ 4]
255: Register [ 2] written with value: [00000004] | [ 4]
265: Register [ 4] written with value: [00000004] | [ 4]
305: Register [ 2] written with value: [00000004] | [ 4]
315: Register [ 4] written with value: [00000004] | [ 4]
355: Register [ 2] written with value: [00000004] | [ 4]
365: Register [ 4] written with value: [00000004] | [ 4]
405: Register [ 2] written with value: [00000004] | [ 4]
415: Register [ 4] written with value: [00000004] | [ 4]
455: Register [ 2] written with value: [00000004] | [ 4]
465: Register [ 4] written with value: [00000004] | [ 4]
505: Register [ 2] written with value: [00000004] | [ 4]
```

Ao executar a instrução bge, como x0 é igual a x0 sempre, o branch é tomado para o endereço atual - 8 = 12 - 8 = 4, ficando em loop infinito.

BNE

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

```
45: Register [ 7] written with value: [00000001] | [ 1]
55: Register [ 2] written with value: [00000004] | [ 4]
65: Register [ 4] written with value: [00000004] | [ 4]
85: Register [ 6] written with value: [00000008] | [ 8]
95: Register [ 7] written with value: [00000001] | [ 1]
105: Register [ 8] written with value: [00000002] | [ 2]
115: Register [ 4] written with value: [00000004] | [ 4]
```

Ao executar a instrução bne, como x0 é igual a x0, o branch não é tomado e a execução do código continua normalmente.

BLT

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

```
45: Register [ 7] written with value: [00000001] | [      1]
55: Register [ 2] written with value: [00000004] | [      4]
65: Register [ 4] written with value: [00000004] | [      4]
105: Register [ 2] written with value: [00000004] | [      4]
115: Register [ 4] written with value: [00000004] | [      4]
155: Register [ 2] written with value: [00000004] | [      4]
165: Register [ 4] written with value: [00000004] | [      4]
205: Register [ 2] written with value: [00000004] | [      4]
215: Register [ 4] written with value: [00000004] | [      4]
255: Register [ 2] written with value: [00000004] | [      4]
265: Register [ 4] written with value: [00000004] | [      4]
305: Register [ 2] written with value: [00000004] | [      4]
315: Register [ 4] written with value: [00000004] | [      4]
355: Register [ 2] written with value: [00000004] | [      4]
365: Register [ 4] written with value: [00000004] | [      4]
405: Register [ 2] written with value: [00000004] | [      4]
415: Register [ 4] written with value: [00000004] | [      4]
455: Register [ 2] written with value: [00000004] | [      4]
465: Register [ 4] written with value: [00000004] | [      4]
505: Register [ 2] written with value: [00000004] | [      4]
```

Ao executar a instrução blt, como x0 é menor que x7 sempre, o branch é tomado para o endereço atual - 8 = 12 - 8 = 4, ficando em loop infinito.

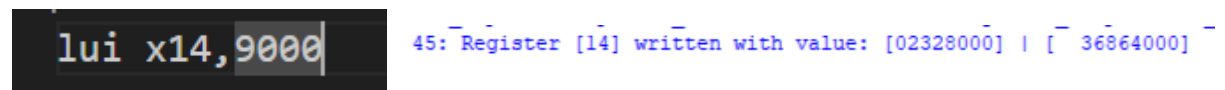
Store e Load

```
addi x7,x0,252
sb x7,0(x0)
lb x8,0(x0)
lbu x9,0(x0)
sh x7,0(x0)
lh x10,0(x0)
sw x7,0(x0)
lw x11,0(x0)
```

```
45: Memory [  0] written with value: [000000fc] | [    252]
45: Register [ 7] written with value: [000000fc] | [    252]
55: Memory [  0] read with value: [xxxxxxxx] | [      x]
55: Memory [  0] read with value: [00000000] | [      0]
60: Memory [  0] read with value: [fffffffc] | [4294967292]
65: Register [ 8] written with value: [fffffffc] | [4294967292]
65: Memory [  0] read with value: [000000fc] | [    252]
75: Memory [  0] written with value: [000000fc] | [    252]
75: Register [ 9] written with value: [000000fc] | [    252]
85: Memory [  0] read with value: [000000fc] | [    252]
95: Memory [  0] written with value: [000000fc] | [    252]
95: Register [10] written with value: [000000fc] | [    252]
105: Memory [  0] read with value: [000000fc] | [    252]
115: Register [11] written with value: [000000fc] | [    252]
```

O byte com valor 252 é carregado para a memória utilizando o sb, o lb escreve -4 no registrador x8 por ter sinal estendido (o bit mais significativo é 1), o lbu escreve 252 no registrador x9. O valor 252 também é carregado para a memória utilizando sh e sw, e são escritos em registradores utilizando lh e lw.

LUI



The image shows a screenshot of assembly code and its effect on a register. On the left, a dark box contains the assembly instruction `lui x14, 9000` in white text. On the right, a light blue box contains the text `45: Register [14] written with value: [02328000] | [36864000]` in blue text.

A instrução LUI é usada para carregar um valor imediato de 20 bits nos 5 bytes mais significativos de um registrador, nesse caso a constante 9000 foi escrita nos bytes mais significativos do registrador x14.

Conclusão

Durante o desenvolvimento do projeto, foi discutida a implementação de instruções do conjunto RV32I do processador RISC-V através das classes de instruções, que incluem R-type (operações lógicas e aritméticas entre registradores), I-type (operações lógicas e aritméticas entre registrador e um valor imediato ou operações de load), S-type (operações de store), B-type (operações de branch), J-type (operações de jump and link, como jal e jalr), U-type (operação lui) e a instrução halt.

Ainda foram descritos as funções dos sinais de controle que comunicam os diferentes estágios do pipeline, sendo essenciais para a coordenar e dirigir o fluxo de dados e a execução das instruções. No contexto da implementação das instruções do conjunto RV32I, esses sinais são essenciais para garantir que cada componente do pipeline funcione corretamente e que as instruções sejam executadas de acordo com a especificação.

As simulações realizadas com a ferramenta ModelSim foram essenciais para verificar se cada instrução implementada estava funcionando corretamente conforme o especificado, além de permitir a validação da integração das instruções no pipeline do RISC-V, verificando o fluxo de dados e controle. Através das simulações foi possível detectar os erros, facilitando a depuração do código.