Universidade Federal de Pernambuco

Centro de Informática

Relatório de Projeto

Dyego Ferreira da Silva (dfs10)

Erick Melo de Souza de Mendonça da Silva (emsms)

Pedro Guerra Pinto Silva (pgps)

Pedro Siqueira de Barros (psb2)

Vinícius Guedes de Macêdo (vgm)

18/02/2025

Índice

Introdução

Este relatório explora em profundidade a implementação de instruções na ISA (Instruction Set Architecture) de um processador RISC-V, examinando os métodos utilizados para integrá-las na arquitetura do processador. Ao longo do documento, são discutidos os principais conceitos envolvidos e as abordagens adotadas para garantir a eficiência e funcionalidade das instruções.

Inicialmente, é apresentada uma visão geral das instruções abordadas, explicando suas funções e o papel que desempenham na operação do processador RISC-V. Esta seção também analisa as diferentes classes de operações, investigando como elas gerenciam o fluxo e o processamento de dados nos registradores, além de avaliar o impacto dessas operações no desempenho geral do processador.

Em seguida, o relatório explora detalhadamente o papel dos sinais de controle, elementos essenciais para coordenar a execução das instruções e manter a estabilidade no fluxo de dados. Cada sinal é examinado em relação à sua função específica na arquitetura, mostrando como contribui para o funcionamento harmonioso do processador e para a sincronização entre os diferentes estágios de execução.

A etapa seguinte descreve os testes realizados com as instruções implementadas, bem como os resultados obtidos por meio de simulações na ferramenta ModelSim. O foco é demonstrar o comportamento prático das instruções no ambiente de simulação e avaliar seu desempenho conforme os parâmetros estabelecidos para a arquitetura RISC-V.

Por fim, o relatório oferece uma conclusão com um resumo dos principais resultados e uma análise crítica sobre a eficácia das instruções implementadas. São apresentadas ainda considerações finais sobre os desafios enfrentados e sugestões para aprimoramentos futuros no desenvolvimento da ISA para o processador RISC-V.

Implementação das Classes de Operações e Módulos Extras

No processador RISC-V, as instruções são organizadas em categorias conforme seus objetivos e formatos, compondo em conjunto a ISA (Instruction Set Architecture). No decorrer do projeto, foram desenvolvidas instruções voltadas para operações aritméticas, controle de fluxo – incluindo desvios condicionais e saltos incondicionais –, além de instruções de load-store e a pseudo-instrução halt. Nesta seção, será detalhado o processo de implementação de cada uma dessas categorias na arquitetura do processador.

Operações de branch condicional

As instruções do formato B-type no RISC-V são responsáveis por realizar desvios condicionais durante a execução. A definição da operação específica a ser executada depende do campo funct3, que determina o tipo de comparação a ser feita. Para implementar essas comparações, é utilizada a ALU, que retorna um valor booleano indicando o resultado da condição avaliada, o que influencia diretamente o comportamento do branch na execução das instruções.

A execução dos desvios requer interação com a unidade de branch, responsável pela atualização do PC (Program Counter), que aponta para a próxima instrução a ser executada. Essa atualização é baseada no incremento do PC_four, que avança o contador em passos de 4 bytes para garantir a sequência correta de execução.

Além disso, a BranchUnit calcula, em paralelo, os endereços para possíveis desvios condicionais enquanto o PC é atualizado. Para determinar o novo endereço do PC, são analisadas variáveis relacionadas ao branch, como o resultado da ALU e a variável branch, que indica a necessidade de desvio. Outros elementos, como Imm e PC_Imm, também são considerados para o cálculo do novo endereço, que é armazenado em BrPC. Por fim, o valor de BrPC é encaminhado, junto com o

PC_four, a um multiplexador que seleciona o próximo valor do PC, controlando assim o fluxo de execução do programa.

Operações de jump

As instruções do tipo J-type no RISC-V são responsáveis por realizar saltos para diferentes partes do código. A execução dessas instruções é definida pelo campo funct3, que especifica o tipo de operação. Dentro dessa categoria, existem duas variantes principais: jal e jalr. Ambas utilizam o sinal RegWrite para controlar a gravação em registradores, mas diferem na forma como manipulam os endereços. A jalr, por lidar com valores imediatos, também faz uso do sinal ALUsrc para calcular o destino do salto.

As operações de jump utilizam a BranchUnit para determinar o endereço do salto. No entanto, diferentemente das instruções de branch, que verificam a condição de desvio por meio da variável branch, as instruções J-type utilizam uma variável específica que controla desvios incondicionais, garantindo a continuidade correta da execução do código.

Além disso, há uma distinção importante entre as duas instruções: enquanto a jal não requer o uso da ALU, pois o salto é sempre incondicional e o endereço é diretamente calculado, a jalr utiliza a ALU para determinar o endereço de destino com base em um valor armazenado em registrador, o que permite maior flexibilidade no cálculo do salto.

Operações aritméticas

As instruções aritméticas no RISC-V pertencem ao formato R-type e são usadas para realizar operações entre registradores, com o resultado sendo armazenado em um registrador de destino. A definição da operação específica é determinada pelos campos funct7 e funct3 presentes na instrução. Nesta seção, será descrito o processo de implementação dessas operações.

No projeto, as instruções R-type já estavam parcialmente configuradas no módulo de controle, exigindo apenas ajustes para especificar como seriam processadas na ALU. Para isso, no Controlador da ALU, foi configurado um vetor de 4 bits chamado Operation, que define a operação a ser realizada para cada instrução (por exemplo, para a instrução SUB, o valor atribuído é 0011). Condicionais foram utilizadas para mapear o valor de Operation com os campos funct7 e funct3 da instrução, determinando qual operação lógica ou aritmética seria executada na ALU.

O vetor Operation é então passado como entrada para a ALU, que utiliza esse valor para selecionar a operação correta. A ALU recebe como entradas os valores SrcA e SrcB, provenientes dos registradores especificados nos campos rs1 e rs2 da instrução. O resultado da operação é gravado no registrador de destino indicado pelo campo rd.

Essa abordagem foi aplicada a todas as instruções aritméticas. Por exemplo, para a instrução SUB, o valor 0011 em Operation é utilizado. As condições necessárias necessárias para identificar a operação de SUB são verificadas nos campos funct7 e funct3. Na ALU, o caso correspondente a 0011 realiza a subtração de SrcA menos SrcB, onde esses valores são recuperados dos registradores definidos por rs1 e rs2. As demais operações aritméticas foram implementadas de forma semelhante, adaptando o valor de Operation conforme o tipo de cálculo requerido.

Operações com imediato

As instruções do formato I-type no RISC-V são utilizadas para realizar operações envolvendo operandos imediatos, em vez de operar exclusivamente com registradores. A implementação dessas instruções segue uma abordagem semelhante à das operações aritméticas, utilizando os campos funct7 e funct3 para determinar a operação específica a ser executada. Assim como nas instruções R-type, é empregado o vetor de bits Operation para indicar à ALU qual operação lógica ou aritmética deve ser realizada.

Embora o gerenciamento de Operation seja semelhante ao das operações aritméticas, a principal diferença está na forma como os operandos são manipulados. Enquanto as operações R-type utilizam dois registradores como entrada, as instruções I-type combinam um registrador com um valor imediato. Essa distinção é controlada pelo sinal AluSrc, que define a origem do segundo operando na ALU.

Quando o sinal AluSrc é ativado, ele indica que a operação deve ser realizada entre um registrador e um valor imediato, permitindo o processamento direto de constantes embutidas na instrução. Por outro lado, quando AluSrc está desativado, a ALU utiliza dois registradores como entrada, funcionando de maneira idêntica às operações aritméticas do formato R-type. Dessa forma, o AluSrc desempenha um papel essencial na diferenciação entre as operações com imediato e as que utilizam apenas registradores, garantindo o comportamento correto na execução das instruções I-type.

Operações de load

As instruções do tipo load no RISC-V também pertencem ao formato I-type, mas diferem das demais por serem destinadas ao carregamento de dados da memória principal para os registradores. A execução dessas instruções é definida pelo valor do sinal ALUop, que indica que uma operação de load deve ser realizada.

Para determinar o tipo específico de operação de load, é necessário consultar o módulo datamemory, responsável por diferenciar as variantes dessas instruções. Ao acessar essa unidade, o valor do sinal MemRead é verificado. Quando MemRead está ativado, o campo funct3 é analisado para identificar qual tipo de load será executado, como load byte, load halfword ou load word.

Além do MemRead, outras variáveis de controle também são afetadas durante as operações de load. O sinal ALUsrc é utilizado para permitir a entrada de valores imediatos, conforme especificado no arquivo de instruções. O sinal MemtoReg gerencia a transferência de dados da memória para o registrador de destino, enquanto RegWrite controla a gravação desses valores no registrador especificado

pela instrução. Esses sinais trabalham em conjunto para garantir que o dado seja corretamente carregado da memória e armazenado no registrador correspondente.

Operações de store

As instruções do formato S-Type no RISC-V são responsáveis por armazenar dados dos registradores na memória principal, funcionando de maneira inversa às operações de load. A implementação dessas instruções segue um processo semelhante ao das instruções de load, porém com algumas diferenças nos sinais de controle utilizados.

Assim como nas operações de load, o sinal ALUsrc é ativado para permitir o uso de valores imediatos na entrada da ALU. No entanto, o diferencial das instruções S-type é o uso do sinal MemWrite, que indica que uma operação de escrita na memória deve ser realizada, permitindo o armazenamento do valor do registrador no endereço especificado pela instrução.

No módulo datamemory, o MemWrite é verificado para determinar se uma operação de store é necessária. Caso positivo, o campo funct3 é analisado para definir qual tipo específico de store será executado, como store byte, store halfword ou store word. Dessa forma, o controle preciso sobre a escrita na memória é garantido, assegurando que o dado seja armazenado corretamente no local desejado.

Halt

A pseudo-instrução Halt no RISC-V tem como função interromper a execução do código no momento em que é chamada. Como não estava originalmente presente no arquivo assembler.py, foi necessário estender a função translate_instructions para que essa instrução pudesse ser interpretada a partir do arquivo instructions.txt. Além disso, foi criado um opcode específico para o Halt nesse mesmo módulo em Python.

No módulo Controller, foi realizado o mapeamento desse novo opcode para um sinal de controle exclusivo, também denominado Halt, permitindo que a unidade de

controle reconheça e responda adequadamente à sua execução. Esse sinal garante que, ao ser identificado, a execução do programa seja interrompida imediatamente.

Por último, a lógica de controle para a instrução Halt foi integrada ao Datapath do processador, assegurando que o fluxo de execução fosse interrompido conforme esperado. Com isso, o comportamento de interrupção foi implementado de maneira consistente com o restante da arquitetura.

Sinais de Controle para as Classes de Instruções

Para executar operações, processar instruções e gerenciar dados e memória, é necessário utilizar mecanismos que informem aos componentes do processador como proceder com base no tipo de instrução em execução. Esses mecanismos, conhecidos como sinais de controle, regulam o fluxo de informações e garantem que o tipo de instrução seja corretamente identificado e tratado durante a execução. Alguns dos sinais principais incluem ALUsrc, MemtoReg, RegWrite, MemRead e MemWrite. No entanto, com a necessidade de implementar instruções mais complexas e específicas, foram criados sinais adicionais para lidar com essas operações de forma mais eficiente. A seguir, discutiremos em detalhes esses sinais e suas funções.

RegWrite

Pela arquitetura do RISC-V ter sido elaborada para ser intuitiva e de fácil implementação, este sinal acaba por ser um dos mais utilizados, uma vez que o RegWrite tem como objetivo administrar a escrita de dados em registradores, tendo seu valor determinado pela ocorrência de instruções de load, jump, aritméticos ou imediatos.

MemRead

O sinal MemRead é responsável por ativar a leitura da memória principal durante a execução de certas instruções. Embora seja utilizado exclusivamente por instruções de tipo load, ele controla todo o processo de leitura no data memory, determinando, junto com o campo funct3, qual operação de leitura será realizada. Após selecionar o tipo de leitura adequado, o data memory atribui o valor lido ao registrador especificado na instrução, indicado pelo campo rd.

MemWrite

Anteriormente apresentado na seção de operações de store, o MemWrite é responsável por controlar a escrita na memória realizada pelas funções S-type. Ao afirmar o sinal de MemWrite, cabe à unidade de data memory utilizar os valores de funct3 para selecionar qual tamanho do dado que será escrito na memória.

Memtoreg

O sinal Memtoreg tem como função selecionar a origem do resultado que será escrito no register file. Para isso, o Memtoreg é encaminhado para um mux que, dependendo do valor do sinal, seleciona se o dado a ser utilizado será da ALU (quando não afirmado), ou da data memory (quando afirmado).

ALUsrc

O sinal ALUsrc tem como função gerenciar operações que envolvem valores imediatos, ou seja, números diretamente inseridos na instrução, ao invés de utilizar valores provenientes dos registradores. Ele controla, através de sinais do Opcode, se a operação será realizada com um imediato ou não. Quando ativado, o ALUsrc permite que operações com imediatos sejam processadas, enquanto, quando desativado, a ALU realiza operações entre registradores.

Quando o ALUsrc é ativado, o módulo imm_gen é acionado para determinar qual operação com imediato deve ser realizada, usando um mux para selecionar entre diferentes tipos de instruções, como I-type, jalr, S-type e B-type. O mux ajusta o valor do imediato conforme necessário e o envia para o srcB, permitindo que o cálculo seja efetuado corretamente.

ALUop

A ALUop, representada por um conjunto de 2 bits, é um dos sinais mais importantes utilizados no projeto, com a função de indicar ao vetor Operation da ALUController qual instrução deve ser executada em determinado momento. Quando uma operação precisa ser realizada na ALU, a ALUop sinaliza qual operação deve ser escolhida, levando em consideração o tipo de instrução.

A atribuição ao vetor Operation é feita com base no opcode de cada tipo de instrução, e essa configuração armazena 2’b00 para instruções dos formatos load, store, jal e halt. Para instruções nos formatos R-Type e I-Type, o valor é 2’b01, enquanto que para instruções de tipo branch, o valor se torna 2’b10, e para instruções do formato jal, o valor é 2’b11. Cada combinação de bits define de maneira clara qual operação deve ser realizada pela ALU.

Branch

Definido apenas pelas instruções de formato de mesmo nome, este sinal é um dos principais responsáveis pelo controle do fluxo do código visto que o mesmo é afirmado ao detectar a ocorrência de uma instrução de branch e envia essa informação para a BranchUnit que altera a posição do PC de acordo com o offset informado na instrução. É na ALU onde fica a diferenciação das instruções branch, definindo a operação que será realizada.

Jump

Esse sinal é ativado em ocorrências de desvios não condicionais. Ele administra a necessidade de pulos quando necessário e salta para a posição em que é definida na instrução. Junto do Branch ele define a necessidade de saltos na execução do código.

JumpReg

Diferentemente do jump, esse sinal é ativado apenas em ocorrência de instruções do formato jalr. Por lidar com um formato de instrução que opera com saltos envolvendo valores imediatos, ele define para o PC se ele pode continuar a execução normal de instruções ou se deve alterar o seu valor de acordo com o valor imediato definido na execução, alterando o valor de PC de acordo com a instrução.

Halt

Esse sinal é responsável por interromper a execução das instruções no código. A gestão dessa interrupção ocorre no controlador, onde o opcode determina quando ela deve ser ativada, transmitindo seu valor binário para o Halt_selector na Branch

Unit dentro do DataPath. Quando a interrupção é acionada, o valor do contador de programa (PC) é zerado, interrompendo a execução do programa e impedindo a execução de instruções subsequentes.

Simulações Realizadas

Usando aritméticas, load e store

Com o código:

addi x7, x0, 0

sb x7, 2(x0)

lw x9, 0(x0)

sh x7, 2(x0)

lw x8, 0(x0)

```
45: Memory [  2] written with value: [00000000] | [          0]
45: Register [ 7] written with value: [00000000] | [          0]
55: Memory [  0] read with value: [xxxxxxxx] | [          x]
55: Memory [  0] read with value: [00000000] | [          0]
65: Memory [  2] written with value: [00000000] | [          0]
65: Register [ 9] written with value: [00000000] | [          0]
75: Memory [  0] read with value: [00000000] | [          0]
85: Register [ 8] written with value: [00000000] | [          0]
```

Não há leitura ou armazenamento de valores diferentes de 0. O x7 recebe o 0 armazenado em x0 com deslocamento 0, e tem seu valor guardado no endereço indicado por 2(x0), depois carrega-se os valores 0 em x8 e x9.

Usando aritméticas, load e store

Com o código:

addi x7, x0, 4

addi x6, x0, 1

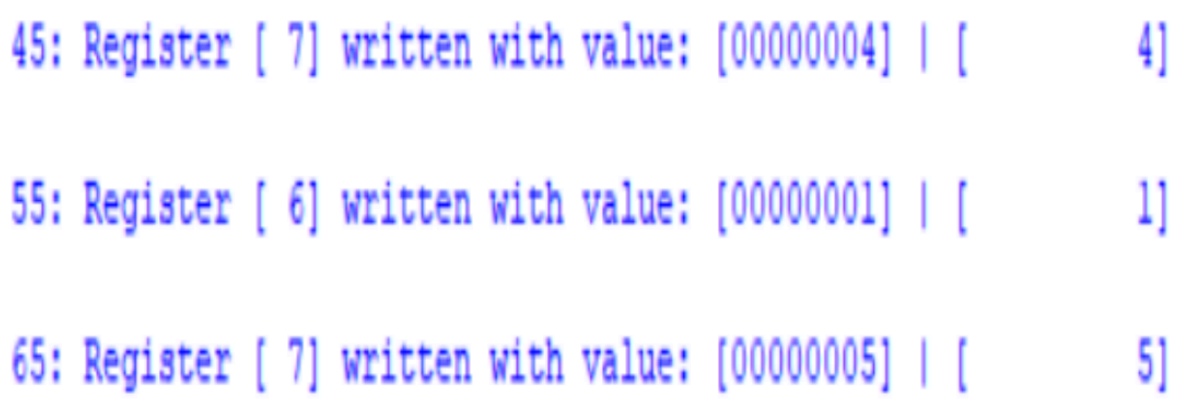
add x7, x7, x6

bge x7, x6, 8

sub x6, x6, x0

halt

sub x7, x7, x6



Os registradores x7 e x6 recebem os valores 0+4 e 0+1, respectivamente. Esses valores são somados, e o resultado (4+1) é armazenado em x7, que agora contém o valor 5. Quando a instrução de desvio condicional (branch greater or equal) é executada, a comparação entre x7 e x6 indica que o valor em x7 é maior ou igual ao de x6. Isso provoca o desvio para uma posição de memória que está 8 bytes à frente da instrução atual, resultando na interrupção da execução normal. Como resultado, o código é interrompido antes que a próxima instrução seja executada, levando ao término do processo com a instrução halt.

Quando Branch not taken

Com o código:

addi x7, x0, 4

addi x6, x0, 1

add x7, x7, x6

beq x7, x6, 8

sub x7, x7, x6

halt

sub x7, x7, x6

```
45: Register [ 7] written with value: [00000004] | [         4]

55: Register [ 6] written with value: [00000001] | [         1]

65: Register [ 7] written with value: [00000005] | [         5]

85: Register [ 7] written with value: [00000004] | [         4]
```

Nesse caso, não ocorrerá desvio, visto que o valor de x7 é diferente do valor de x6.

O código vai parar no halt.

Aritméticas e Jump

addi x7, x0, 3

addi x6, x0, 1

jal x5, 8

sub x7, x7, x6

slt x3, x6, x7

```
45: Register [ 7] written with value: [00000003] | [          3]

55: Register [ 6] written with value: [00000001] | [          1]

65: Register [ 5] written with value: [0000000c] | [         12]

95: Register [ 3] written with value: [00000001] | [          1]
```

O registrador x7 é atribuído com o valor 3, e o x6 recebe 1. A instrução jal grava no registrador x5 o endereço de retorno da instrução e, em seguida, redireciona a execução para a instrução slt. Esta, por sua vez, compara os valores de x6 e x7, e como o valor de x6 é inferior ao de x7, armazena 1 no registrador x3.

Shift

addi x7, x0, 2

addi x6, x0, 32

srli x6, x6, 4

slli x7, x7, 4

srai x7, x7, 1

```
45: Register [ 7] written with value: [00000002] | [          2]

55: Register [ 6] written with value: [00000020] | [         32]

65: Register [ 6] written with value: [00000002] | [          2]

75: Register [ 7] written with value: [00000020] | [         32]

85: Register [ 7] written with value: [00000010] | [         16]
```

As operações de shift lógico alteram o valor armazenado em um registrador, dividindo ou multiplicando por 2. Quando o shift é para a direita, o valor é dividido

por 2, e quando é para a esquerda, o valor é multiplicado por 2. A quantidade de shifts é determinada pelo número fornecido na instrução. No caso de x6, que inicialmente contém o valor 32, é realizado um shift à direita, dividindo o valor por 2 quatro vezes, ou seja, 32/2^4, resultando em 2. Em x7, é executado um shift à esquerda, multiplicando o valor armazenado (2) por 2, quatro vezes, ou seja, 2 * 2^4, o que dá 32. Além disso, a instrução srai realiza um shift à direita aritmético, que não apenas move os bits à direita, mas também preenche o bit mais significativo com o mesmo valor do bit mais significativo original, ao invés de preenchê-lo com zero, como no shift lógico.

Conclusão

Neste relatório, foi explorada a implementação das instruções do processador RISC-V, com foco nas diferentes operações presentes em sua ISA. Foram analisadas as operações aritméticas, de controle de desvio, load-store e a pseudo-instrução halt, destacando as abordagens adotadas durante o desenvolvimento do projeto.

Além disso, foi discutido o papel dos sinais de controle, que comunicam as operações com as classes de instruções da arquitetura, detalhando sua implementação e funcionamento para garantir a operação correta no hardware.

As simulações no ModelSim proporcionaram uma compreensão mais clara sobre o funcionamento da ISA de um processador, facilitando a análise da interação entre os componentes e das formas de implementar as instruções no nível de máquina.

Em conclusão, a implementação da ISA do processador RISC-V foi essencial para um maior entendimento dos temas trabalhados, fornecendo uma visão mais ampla sobre como as máquinas funcionam atualmente e a importância de adaptar os computadores às necessidades do mercado moderno, compreendendo a interconexão dos componentes para uma execução eficiente que atenda às demandas dos usuários.