Universidade Federal de Pernambuco
Centro de Informática

Relatório de Projeto

Artur Vinicius Pereira Fernandes (avpf)
Caio Cesar Nascimento Vilas Boas (ccnvb)
Fábio Pereira de Miranda (fpm3)
Felipe Mateus Falcão Barreto (fmfb)
João Victor da Silva Nascimento (jvsn2)

29/02/2023

Índice

Introdução

Este relatório tem como objetivo fornecer uma análise sobre a implementação de instruções na *ISA* (*Instruction Set Architecture*) de um processador RISC-V. Ao longo deste documento, são explorados os métodos que foram utilizados para a implementação de tais instruções.

Primeiramente, apresentaremos uma visão geral das instruções que serão aplicadas, destacando suas funcionalidades e impactos dentro do processador RISC-V. Também será feita uma análise do funcionamento das classes de operações, cuja finalidade é de gerenciar tanto a transição quanto o tratamento de dados que serão utilizados pelo registrador.

Após isso, iremos abordar sobre a funcionalidade de cada um dos sinais de controle. Esses elementos são mecanismos cruciais para a fluidez de instruções da *ISA*, gerando um bom funcionamento da máquina no tratamento de dados, tendo cada sinal sua própria utilidade no código.

Posteriormente, serão examinados os testes que foram realizados utilizando essas instruções e os resultados obtidos através de simulações realizadas usando a ferramenta *ModelSim*, com o intuito de demonstrar o funcionamento da *ISA* do processador.

Por fim, iremos concluir com uma síntese dos resultados obtidos com a utilização das instruções implementadas e apresentar nossas considerações finais sobre o projeto.

Implementação das Classes de Operações e Módulos Extras

No RISC-V, as instruções pertencem a classes baseadas em objetivos e formatos específicos, formando, juntas, a *ISA* do processador. Durante a realização do projeto, foram implementadas instruções aritméticas, de controle de desvio, sendo esses *branches* condicionais ou *jumps* incondicionais, de *load-store* e a pseudo-instrução *halt*. Dessa forma, iremos desenvolver nesta seção como foi realizada a implementação de cada uma dessas classes presentes na arquitetura.

Operações de branch condicional

Sendo do formato *B-type*, essas instruções buscam realizar desvios condicionais na execução das instruções. A sua execução depende do seu conjunto de bits *funct3*, que determinam qual operação será realizada. Por realizarem operações comparativas, foi necessário utilizar a *ALU*, retornando um booleano que determina como o branch se comporta durante a execução de acordo com a resposta comparativa da execução.

Para realizar as operações de desvio, é necessário uma comunicação com a unidade de *branch*, que administra a atualização do *PC*, contador que determina a partir de uma pilha qual instrução será executada. Essa atualização se deve do *PC_four*, que é atualizado em intervalos de 4 para gerar progressão na execução de instruções.

Entretanto, na *BranchUnit* também são calculados os endereços de desvios condicionais paralelamente ao cálculo de endereço de instruções de *PC*. Após análise comparativa entre variáveis de *branch*, como o resultado da *ALU* anteriormente citado e a variável *branch*, que determina existência ou não do desvio, além de *Imm* e *PC_Imm*, é realizado o cálculo do possível novo endereço apontado por *PC*, armazenado em *BrPC*. Após isso, esse endereço é enviado junto do *PC_four* a um mux que, a partir de lá, é definido qual o novo valor de *PC*.

Operações de jump

Operações *J-type* realizam jump para outros pontos do código. A execução desse tipo de operação é determinada pelo conjunto de bits *funct3* da entrada. Por se dividir entre *jal* e *jalr*, o caminho de execução de ambas acaba por se diferenciar um pouco, tendo ambas usando o valor do sinal *RegWrite*, porém a *jalr*, por manusear valores imediatos, usa também o sinal *ALUsrc*.

Outro detalhe importante é o uso da unidade *BranchUnit*, que, diferentemente das operações de *branch*, que usa da variável *branch* para determinar a necessidade de um desvio condicional no progresso de execução de instruções, as operações de *jump* utilizam a variável de mesmo nome, que determina se é preciso haver um desvio não condicional.

Por fim, vale ressaltar que a instrução de *jal* não possui acesso à *ALU*, uma vez que, por não haver operação condicional para execução do desvio, não necessita realizar operações nessa unidade. Entretanto, a *jalr* precisa, uma vez que ela usa a unidade para operar o *jump* a partir do registrador.

Operações aritméticas

As operações aritméticas são do formato *R-type* e tem como objetivo realizar operações entre registradores, onde o resultado dessa operação é escrito em outro registrador. Como instruções no formato *R-type*, seu campo de bits *funct7* e *funct3* definirão qual operação será realizada. A implementação dessas operações será descrita a seguir.

As operações do formato *R-type* já estavam previamente implementadas no módulo de controlador, portanto, só foi preciso adicionar como as operações seriam executadas na *ALU*. No Controlador da *ALU* selecionamos como seria o vetor de 4 bits *Operation* que define a operação da *ALU* para cada instrução (por exemplo, definimos que para a instrução *SUB* seria correspondente a 0011 no vetor), e então, usamos condicionais para definir como cada bit do *Operation* corresponderia à operação recebida do controlador sobre o que seria a operação realizada na *ALU*, o *funct7* e *funct3* da instrução. Esse vetor de bits *Operation* é recebido como input na

ALU, que a partir de qual valor está contido nesse vetor, realiza a operação lógica/aritmética, recebendo como *input* os valores *SrcA* e *SrcB* contidos nos registradores, indicados pelos endereços contidos nos campos *rs1* e *rs2* da instrução. Após a realização da operação, o resultado é então escrito no registrador destino indicado pelo espaço *rd* dos bits da instrução.

Implementamos todas as instruções aritméticas dessa forma, por exemplo, a instrução *SUB* utiliza o valor ‘3’(0011) no vetor “*Operation*” , em cada bit deste vetor selecionamos as condições que indiquem que a instrução será de *SUB*, neste caso os bits que indicam um tipo de operação na *ALU funct7* e *funct3* de *SUB*. Na *ALU* indicamos que o caso 3 realizará uma operação de subtração tirando de *srcA* - *srcB* onde *srcA* e *srcB* são os valores contidos nos registradores indicados por *rs1* e *rs2* na instrução. As outras operações aritméticas são implementadas de maneira análoga à instrução *SUB*.

Operações com imediato

Com uma implementação similar ao das operações aritméticas e pertencente ao formato *I-type*, essas instruções buscam realizar operações com operadores imediatos ao invés de apenas registradores.

A implementação das instruções com operadores imediatos possui uma abordagem similar ao dos operadores aritméticos, uma vez que utiliza também os conjuntos de bits *funct7* e *funct3* para gerenciar a operação que será realizada, como também o gerenciamento do vetor de bits *Operation*, que, pela diferença entre as operações com imediato e as aritméticas ser a entrada de valores e não o uso delas, o valor repassado pelo vetor à ULA acaba por ser o mesmo.

Entretanto, o seu diferencial fica pelo uso do *AluSrc*. A principal funcionalidade deste sinal é de controlar a operação que será feita. O *AluSrc*, quando é ativado a partir do sinal do *Opcode*, realiza uma operação entre um registrador e um imediato, porém, quando desativado, realiza entre dois registradores, uma vez que é necessário informar ao processador qual o formato do valor que está sendo utilizado como entrada.

Operações de load

Embora pertençam também ao formato *I-type*, essas instruções possuem propósitos distintos ao anterior, tendo como objetivo realizar o carregamento de dados da memória principal para os registradores.

Para determinar a execução de instruções do tipo *load*, é necessário apenas conferir o valor armazenado em *ALUop*. Entretanto, para definir qual tipo de load será executado, é necessário acessar o *datamemory*, uma vez que ele faz a diferenciação das instruções deste tipo. Ao acessar essa unidade, é conferido o valor em *MemRead* que, ao ativado, verifica o valor em *funct3* para, assim, determinar o tipo de *load* que será executado.

Além do *MemRead*, as operações de *load* alteram outros sinais, entre eles *ALUsrc* (para conferir valores imediatos na entrada de valores em instructions.txt), *MemtoReg* (que gerencia a passagem de valores da memória para o registrador) e *RegWrite* (que salva os valores utilizados nos registradores da instrução).

Operações de store

Sendo do formato *S-type*, o objetivo dessas operações é realizar o caminho oposto ao das operações de *load*, carregando dados dos registradores à memória principal.

As operações de *store* são implementadas de uma forma bastante semelhante aos de *load*, tendo a principal diferença nos sinais que são atualizados, sendo eles *ALUsrc* (mesmo motivo das instruções de *load*) e *MemWrite*, que serve para indicar a necessidade de escrita de informações na memória.

No *datamemory*, processo bastante semelhante também às instruções de load, que usa o *MemWrite* para ver se, como dito antes, é necessário realizar uma instrução de armazenamento de valores na memória. Caso necessário, utiliza-se também o conjunto de bits *funct3* para determinar qual instrução de store será executada.

Halt

O *Halt* como objetivo interromper a execução do código no momento em que é executada. Por se tratar de uma pseudo-instrução, não veio implementado no arquivo *assembler.py*. Dessa forma, completamos a função *translate_instructions* para que possibilitasse a leitura por meio do *instructions.txt*. Também criamos o opcode dela nesse mesmo arquivo *python*.

Além disso, na unidade *Controller*, atribuímos o controle de chamada dessa instrução por meio do *opcode* criado, sendo atribuído para o sinal criado por nós de mesmo nome.

Por fim, implementamos a atualização de execução de instruções no *Datapath* do projeto.

Sinais de Controle para as Classes de Instruções

Para execução de operações, instruções, e gerenciamento de dados e memória, é preciso utilizar mecanismos que informam aos componentes da *ISA* do processador o que precisa ser feito de acordo com cada tipo de instrução. Assim, são utilizados os sinais, mecanismos que controlam o fluxo de informações de instruções para que, dessa forma, possa ser definido de forma adequada que tipo de instrução está sendo tratada no momento. Os principais sinais são *ALUsrc*, *MemtoReg*, *RegWrite*, *MemRead* e *MemWrite*, porém, devido a necessidade de implementar instruções de natureza mais variada, criamos outras que auxiliam na execução de instruções mais específicas. Iremos abordar mais sobre a seguir.

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

Como indicado no nome, o sinal *MemRead* possui o objetivo de habilitar a execução de instruções de leitura da memória principal. Mesmo sendo gerenciado apenas pelas instruções de tipo *load*, esse sinal administra no *data memory* todo o sistema de leitura, realizando, juntamente do conjunto de bits *funct3*, a escolha de qual tipo de leitura será efetuado. Ao selecionar o tipo de leitura, o *data memory* atribui o dado lido ao registrador utilizado na instrução, representado como *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

Como previamente dito na parte que apresenta sobre operações com imediato, o *ALUsrc* possui como objetivo controlar a execução de instruções que envolvem (ou não) elementos imediatos, ou seja, ao invés do uso de registradores, utilizamos diretamente os próprios números.

Ele administra a execução de operações imediatas por meio de sinais do *Opcode*, que, quando ativa o *ALUsrc*, faz ele realizar essas operações, aplicando-se também o mesmo para operações aritméticas quando o *ALUsrc* está desativado.

Após a ativação do *ALUsrc*, é utilizado o *imm_gen* para definir qual tipo de operação imediata será realizada por meio de um *mux*. Esse *mux* seleciona entre instruções de *I-type load part*, *I-type*, *jalr*, *S-type* e *B-type* qual é o tipo de imediato que está sendo utilizado, retornando para o *srcB* o valor corretamente ajustado para a situação, realizando assim o cálculo.

ALUop

Sendo representada por um conjunto de 2 bits e um dos principais sinais utilizados no projeto, a *ALUop* possui como finalidade sinalizar ao vetor *Operation* da *ALUController* qual instrução deve ser executada no momento, realizando na *ALU* a operação, caso essa seja necessária. Nesse caso, a atribuição do valor de

Operation se deve os *opcode* de cada tipo de instrução, armazenando 2'b00 para instruções dos formatos *load*, *store*, *jal* e *halt*, 2'b01 para instruções dos formatos *R-Type*, *I-Type*, 2'b10 para instruções do formato *branch* e 2'b11 para instruções do formato *jal*.

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 controla a interrupção de execução do código de instruções. Essa administração de interrupções é realizada no *controller*, com o *opcode* indicando sua ativação, carregando assim no *DataPath* o seu valor binário para *Halt_selector* dentro da *Branch Unit*. Caso o *Halt* esteja ativado, o valor de PC será zerado, interrompendo o programa e impedindo a execução de novas instruções.

Simulações Realizadas

Usando aritméticas, load e store:

```
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. x7 recebe o 0 armazenado em x0 + 0 e tem seu valor guardado no endereço indicado por 2(x0), depois carrega-se os valores 0 em x8 e x9.

Aritméticas, *branch* condicional e *halt*:

```
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
```

```
45: Register [ 7] written with value: [00000004] | [         4]
55: Register [ 6] written with value: [00000001] | [         1]
65: Register [ 7] written with value: [00000005] | [         5]
```

x7 recebe 0+4, x6 recebe 0+1, os valores em cada registrador são adicionados e guardados em x7 (4+1 = 5). No *branch greater or equal*, o valor em x7 é maior ou igual ao valor em x6, o que implica que ocorrerá *branch,* e portanto o código irá "pular" para 2 instruções (visto que a próxima instrução é indicada por PC + 4 e o *branch* indica um desvio de +8) até o *halt* onde o código para, e a instrução sub subsequente não será usada.

Branch not taken:

```
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 parará no *halt*.

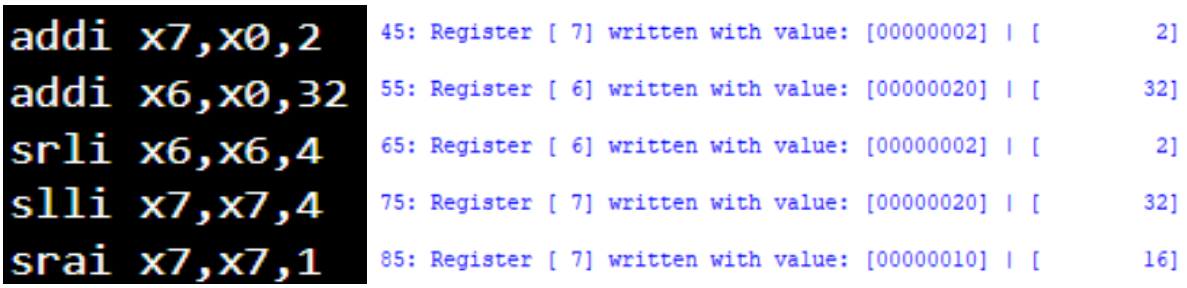
Aritméticas e *jump*:

```
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 recebe 3, o x6 recebe 1, o *jal* escreve no registrador x5 o endereço de retorno da instrução e então desvia o código para instrução *slt*, que armazena 1 no registrador x3 visto que o valor de x6 é menor que o valor em x7.

***Shift*:**



Operações de *shift* lógico, dividem ou multiplicam por 2 o valor armazenado no registrador, (para direita divide, e para esquerda multiplica). A quantidade de *shifts* que ocorrem é dada pelo número indicado na instrução. Em x6, que recebe o valor 32 ocorre um *shift right* desse valor de modo que o resultado é igual a 32 dividido por 2, 4 vezes, ou 32/2^4, cujo resultado é 2. No x7, ocorre um *shift left*, uma multiplicação análoga a divisão no *shift right*, onde o valor armazenado nesse registrador (2) é multiplicado por 2, 4 vezes (4 *shifts left*), onde 2*2^4 = 2^5 = 32. Além disso, o srai realiza a operação similar ao *shift-right* , mas utilizando imediatos para definir a quantidade de *shifts*.

Conclusão

Ao longo deste relatório, foi explorada a implementação de instruções do processador RISC-V, abordando as classes de operações presentes na sua *ISA*. Foi discutido o funcionamento das operações aritméticas, de controle de desvio, *load-store* e da pseudo-instrução *halt*, destacando como cada uma foi implementada ao longo do projeto.

Além disso, foram apresentados os sinais de controle presentes, cuja finalidade é de se comunicar com as classes de instruções presentes na arquitetura, explicando tanto a implementação quanto o funcionamento de cada sinal para a correta implementação no *hardware*.

As simulações realizadas utilizando o *ModelSim* contribuíram para uma melhor visão da execução de uma *ISA* de um processador, ajudando a entender a conexão entre cada componente e as várias formas de implementar as instruções necessárias em um computador no nível de máquina.

Dessa forma, concluímos que a implementação da *ISA* do processador RISC-V nesse projeto foi de suma importância para um entendimento mais aprofundado sobre os temas abordados em sala de aula, sendo crucial para um aprofundamento na visão sobre o funcionamento das máquinas atuais, tão como uma melhor compreensão sobre a necessidade do desenvolvimento de computadores mais adaptados ao mercado do mundo moderno, compreendendo como os componentes estão interligados para uma execução que atenda de forma eficaz os clientes do mundo atual.