Relatório de implementação simulador ISA RISC-V

Otho Teixeira Komatsu
Matrícula: 170020142
Professor: Ricardo Pezzuol Jacobi
Disciplina: Organização e Arquitetura de computadores
Turma: C

May 5, 2019

1 Problema

1.1 Objetivo

O projeto visa à simulação da arquitetura **RISC V RV321** desde a implementação das instruções de acesso à memória até a ISA do RISC V RV321 e sua execução(ciclo de fetch) por meio da linguagem C.

1.2 Descrição

A arquitetura **RISC V RV321** tem como sua característica de memória o seu armazenamento de trechos de `32 bits` , ou `4 bytes`; denotados como `words`, em cada endereço de acesso, também expresso em `32 bits`. *I.e.*, o acesso de cada endereço se dá de 4 em 4 bytes, e dentro, delas é possível obter subtrechos de memória: o `half words`, `16 bits`, ou `2 bytes`; e os `bytes`, trechos de `8 bits`, ou `1 byte`.

Para a implementação das funções tanto de obter o valor armazenado em cada trecho de memória quanto armazenar o valor nela; é importante considerar esses intervalos de acesso. Ao adotar esse padrão de arquitetura, não seria permitido, por exemplo, em uma memória iniciada em `0x00000000`, acessar uma word no endereço `0x00000001`, já que cada word é armazenado de 4 em quatro bytes iniciada em `0x00000000`. O mesmo valeria para as `half word`, que seguindo essa lógica; não poderiam serem acessadas no endereços `0x00000001` ou `0x00000003`, uma vez que seu acesso é de 2 em 2 bytes.

Um importante fator a ser considerado também é a forma como esses dados serão interpretados de acordo com suas finalidades. Ou seja, um mesmo número hexadecimal armazenado na memória pode tanto ser interpretado como um inteiro com sinal como um inteiro sem sinal. Os seus intervalos de valores também podem ser limitados de acordo com seu tipo de dado(`byte` , `halfword` ou `word`).

Todos esses dados de memória são manipulados por meio de registradores. No RISC V RV321, há um banco de 32 registradores, cada uma com uma finalidade e propiedade

voltada ao programa e ao seu sistema de execução. Além dessas 32, há o PC e o RI, voltada para indicar o endereço da próxima instrução e o código da instrução a ser executada, respectivamente.

As instruções básicas e o modo como elas são interpretadas nessa arquitetura é regida pela sua ISA. Na sua ISA, é notável que as instruções são agrupadas em tipos. Cada tipo possui os seus campos de descrição (imediatos, opcode, funct3, funct7, shamt, rs1, rs2 e rd) e tamanhos de bits diferentes(variando de 5 até 21 bits).

Para a execução dessas instruções, é aplicado o ciclo fetch. *I.e.*, `fetch`, que é obter a instrução (em hexadecimal) a ser executada e incrementar o `PC`; `decode`, obter os campos de dados associados à operação a que elas correspondem e aos dados envolvidos; e `execute`, que a partir dos dados obtidos pelos campos da instrução é efetuado uma operação especificada pelo `opcode`, `funct3`, e `funct7`, atráves dos registradores mencionados

1.3 A simulação

Seguindo a arquitetura visada, haveriam de ser implementadas:

- A memória, no caso foi utilizado um array de 4096 elementos do tipo *int*32_*t* , correspondente ao `word`; as instruções de carregar os dados armazenados na memória, o *load*, de acordo como o padrão solicitado(`word`, `halfword`, ou `byte`; *signed* ou *unsigned*); e as instruções de armazenar dados nelas, *store*, também seguindo um padrão (`word`, `halfword`, ou `byte`).

- Leitura de arquivo binário, programa responsável por ler os arquivos binários de dados do tipo data(dados) e text(código) e carregá-las na memória, nas suas respectivas áreas(text, 0 a `0x2000`; data, `0x2000` a `0x2FFC`).

- Banco de registradores, um array global de 32 elementos tipo `int32_t` e algumas variáveis globais tipo `int32_t`, como o `PC` e `RI`. Além delas, é necessário algumas variáveis associadas aos campos de instrução.

- Ciclo Fetch, separada nas funções `fetch`, `decode`, `execute`. Todas elas são executadas dentro de uma `step`, que por sua vez é executada a cada `run`, que além de fazer o padrão de execução mencionado, verifica se o `PC` alcançou seu valor máximo(`0x2000`).

2 Implementação

Todas as funções implementadas abaixo tratam casos de exceção, nos erros de endereçamento e offset's equivocados.

2.1 load word(lw)

Função que simula a instrução de carregar um `word` da memória. Recebe os parâmetros input, localiza o endereço(múltiplo de 4) no array de memória, cujo índice correspondente é a divisão do endereço inserido por 4 (cada elemento do array é um `word`); e retorna o valor armazenado no endereço.

2.2 load half word(lh)

Função que simula a instrução de carregar um `half word` da memória. Recebe os parâmetros input, localiza o endereço como na função anterior, desloca para a direita a quantidade de bytes offset(nesse caso, múltiplos de 2), e lê o valor indicado pelos `16 bits` menos significativos, assim sendo retornado. Por ser retornado como um número de `32 bits` com sinal, é propagado o 15º bit para os mais significativos, preservado seu sinal(operação prevista pela conversão de tipos na linguagem C).

2.3 load half word(lhu)

Função que simula a instrução de carregar um `half word` da memória. Recebe os parâmetros input, localiza o endereço como nas funções anteriores, desloca para a direita a quantidade de bytes offset(múltiplos de 2 também), e lê o valor indicado pelos `16 bits` menos significativos, assim sendo retornado. Por ser retornado como um número de `32 bits` sem sinal, os bits após o 15º são zerados.

2.4 load byte(lb)

Função que simula a instrução de carregar um `byte` da memória. Recebe os parâmetros input, localiza o endereço como nas funções anteriores, desloca para a direita a quantidade de bytes offset(de 0 e, no máximo, a 3), e lê o valor indicado pelos `8 bits` menos significativos, assim sendo retornado. Por ser retornado como um número de `32 bits` com sinal, é propagado o 7º bit para os mais significativos, preservado seu sinal(operação prevista pela conversão de tipos na linguagem C).

2.5 load byte unsigned(lbu)

Função que simula a instrução de carregar um `byte` da memória. Recebe os parâmetros input, localiza o endereço como nas funções anteriores, desloca para a direita a quantidade de bytes offset(de 0 e, no máximo, a 3), e lê o valor indicado pelos `8 bits` menos significativos, assim sendo retornado. Por ser retornado como um número de `32 bits` sem sinal, os bits após o 7º são zerados.

2.6 store word(sw)

Função que simula a instrução de armazenar um dado do tipo `word` na memória. É inserido o input, é localizado o endereço como nos procedimentos anteriores, e o dado inserido é sobre-escrito em cima do valor armazenado na memória.

2.7 store half word(sh)

Função que simula a instrução de armazenar um dado do tipo `half word` na memória. É inserido o input, é localizado o endereço como nos procedimentos anteriores, e é feito uma máscara com o `0xFFFF` para apagar os dados relativos ao offset inserido no input(múltiplos de 2). Logo após, o dado inserido é deslocado a quantidade de bytes offset para a esquerda e inserida com a operação ou bit a bit (|) do C no local de memória.

2.8 store byte(sb)

Função que simula a instrução de armazenar um dado do tipo `byte` na memória. É inserido o input, é localizado o endereço como nos procedimentos anteriores, e é feito uma

máscara `0xFF` para apagar os dados relativos ao offset inserido no input(de 0 e, no máximo, a 3). Logo após, o dado inserido é deslocado a quantidade de bytes offset para a esquerda e inserida com a operação "ou" bit a bit ,(|) do C, no local de memória correspondente ao offset.

2.9 dump memory

Função que imprime na tela o array de memóri na sua forma indexada, e o valor armazenado no local de memória correspondente e indicada pelo índice do array(mem[0] corresponde à 1ª `word`; mem[1] , à 2ª `word`; etc.)

2.10 dump register

Função que imprime na tela o array de registros na sua forma inxedada, e o valor armazenado no registrador correspondente e indicada pelo índice do array(breg[x0] corresponde ao valor no registro `zero`; breg[x1], ao valor no `ra`, etc.)

2.11 read bin

Função responsável por ler os arquivos *text.bin* e *data.bin* e transferir os dados lidos pra memória, representado pelo array `mem` na simulação.

2.12 fetch

Função responsável por carregar do array `MEM` as instruções e carregá-las ao `RI`, e incrementar em 4 o `PC`; processo responsável por avançar para a próxima instrução.

2.13 decode

Função responsável por obter os campos que compõe a instrução. Nesse tratamento, é obtido todos os campos possíveis que podem compor uma instrução, satisfazendo cada formato possível. Além disso, os bits dos imediatos já são tratados com a extensão de sinal e rearranjados, dessa forma disponíveis para uso na execução de instruções.

2.14 execute

Uma vez obtido os campos das instruções, em específico o `opcode`, `funct3` e `funct7`; é possível assimilar a instrução solicitada. Nesse sentido, a função detecta a instrução requerida e a executa de acordo com os padrões requisitados e dados inseridos e decodificados da instrução.

2.15 step

Função que representa a passagem de um ciclo do procedimento `fetch`. Nela, é realizado o `fetch`, `decode`, e `execute`.

2.16 run

Função responsável por executar os ciclos por meio do `step`, e dessa forma, operar um código `assembly` por inteiro. Ela executa os ciclos até o programa executar uma chamada de sistema de `exit` do programa ou quando o `PC` atinge seu valor máximo(`0x2000`)

3 Testes e resultados

O desenvolvimento do programa foi conjunta com os testes, ou seja, orientada a testes. Por causa disso, cada

bloco de testes é associada a uma função específica, seguindo uma ordem cronológica de criação e desenvolvimento. *I.e.*, os primeiros testes são associados às primeiras funções desenvolvidas, e utiliza somente suas anteriores; as últimas, as mais recentes.

3.1 Read bin

Nesse teste foi chamado a função de leitura do arquivos binários e seus carregamentos na memória. Para sua verificação, foi executadas o mesmo programa no RARS, comparando-se os dados relativos ao `data` e `text` com os prints realizados do array de memória(`MEM`). Nela verifica-se o sucesso não só da leitura como também do respeito dos intervalos de memória dedicadas a cada tipo de dado(índices `0` a `2048`, data; `2049` a `4098`, text).

3.2 Fetch

Foi executado um `Fetch` e verificou-se os registros `RI` e `PC`. Como esperado, a cada `Fetch` realizado, o `PC` incrementa em 4 e o `RI` assume o valor hexadecimal da instrução a ser decodidificada.

3.3 Decode

Chamado o decode em cima dos códigos armazenados e provenientes do `read bin`, os campos associados apresentam os valores esperados e consistentes com seus padrões de acordo com o valor da instrução , como é printado na tela. No teste de execução do run, no programa `assembly_code`, é mais observável a regularidade do seu funcionamento.

3.4 execute

Nesse banco de testes, foi verificado a execução de cada tipo de instrução. Para cada uma, foi verificado seus sucessos e diferentes situações. De modo geral, apresentaram um funcionamento regular e esperado.

3.4.1 LUI

Nesse teste, foi realizado a operação com o valor `0xFFFFCAFE`, e o valor esperado `0xFCAFE000` é exibido(valor de 5 `bytes` shiftado 3 em `bytes` à esquerda).

3.4.2 AUIPC

Nesse teste, foi realizado a operação com o valor `0x0001234B` e `PC` 4, e o valor esperado `0x1234B000` é exibido(valor shiftado 3 `bytes` à esquerda somada com o `PC`)

3.4.3 Loads

São realizados o diferentes loads dos endereço `0x2000`, `0x2004` e `0x2003` proveniente arquivo `data.bin` arbitrariamente selecionado. O valor verificado corresponde com o esperado. Basta abrir o RARS e compará-lo com o valor no endereço correspondente.

Foi testado também casos em que o `offset` é negativo. Nessas situações, o bit sinal é indicado pelo 12^{o} `bit`, como o esperado para o padrão de imediatos em `loads`.

3.4.4 Store

São realizados diferentes `stores` em cima dos endereços selecionados. Como verifica-se pelo padrão dos endereços e `offsets`, os dados são inseridos exatamente nos endereços corre-

spondentes(resultado da soma do endereço com o `offset`).

Verificou-se também as situações em que os `offsets` eram negativos. Conforme os resultados esperados, o `store` ocorreu sem nenhuma irregularidade, armazenando nos endereços resultantes do `offset`. Vale notar que o 12º bit sinal é propagado, como o esperado.

3.4.5 Tipo B

Para cada instrução dessa categoria, é listada os valores nos registros e o comportamento da instrução conforme às condições requisitadas por ela em relação a esses registros. Por exemplo, no `BEQ`, a instrução é executada quando ambos registros inseridos na instrução possuem valores iguais. Para seu teste de sucesso, os valores dos registros foram colocadas como iguais, e espera-se que ela promova uma `branch` para o endereço resultantes do `offset` inserido.

Em todas as situações e instruções, o programa respondeu conforme às condições impostas pela instrução. Foi testado também as `branches` de valor negativo. O seu resultado condiz com o esperado.

Vale notar que o 13º bit sinal é propagado, como o esperado, e seus valores são sempre pares.

3.4.6 Tipo JAL

No JAL, foi realizada tanto saltos de `offsets` positivos quanto negativos. O `PC` resultante condiz com o esperado(verificável por meio de uma simples soma).

Vale notar que o 21º bit sinal é propagado, como o esperado, e seus valores são sempre pares.

3.4.7 Tipo JALR

No JALR, também foi realizada saltos de `offsets` positivos e negativos. O `PC` resultante condiz com o esperado(verificável por meio de uma soma do `PC` com `RS1`).

Vale notar que o 12º bit sinal é propagado, como o esperado, e seus valores são sempre pares.

3.4.8 Tipo ILATypes

Nas intruções do `ILATypes`, foram testados diferentes combinações de valores do registrador com o imediato. Todos os valores resultantes comportam-se conforme o resultado esperado, previamente e facilmente calculado.

Vale notar que o 12º bit sinal é propagado, como o esperado. Consequentemente, os resultados são influenciados com a propagação desse `bit` sinal. Muito observável quando é realizado, por exemplo, na operação `ORI` do imediato `0xFFF` com o valor 0, resultando-se em `0xFFFFFFFF`; em contraste com a operação `ORI` do imediato `0x7FF` com o valor 0, que resulta em `0x000007FF`. Tais resultados, de fato, são os esperados.

3.4.9 Tipo RegType

Nesse conjunto de instruções, os registros foram manipulados pelas suas diversas operações. E foi explorado a variação dos sinais dos operandos. No final, os resultados são corretamente computados com a consideração desses sinais, sucesso observado ao serem comparadas com o valor esperado. Uma situação básica mas notável é a operação `ADD` de `0xFFFFFFFF` com `0x00000001`, que resulta corretamente

em 0. Portanto, as instruções apresentam um bom funcionamento.

3.4.10 Tipo ECALL

Como último banco de testes, realizou-se a requisição das chamadas de sistema mais comuns(`PrintString`, `PrintChar`, `PrintInt`, e `EXIT`). Um conjunto de dados foi armazenado na memória, e foi executada cada uma dos tipos de operações `SYSCALL`. Todas as chamadas funcionaram regularmente, inclusive o `EXIT`. Basta notar que quando chamada, o `printf` que vem logo abaixo dela não é executada, de fato, encerrando de vez o programa.