Relatório de implementação simulador ISA RISC-V

Otho Teixeira Komatsu
Matrícula: 170020142
Turma: C

April 25, 2019

1 Problema

1.1 Objetivo

O projeto visa à simulação da arquitetura **RISC V RV321** , especificamente, da implementação das instruções de acesso à memória 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`).

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 $int32\_t$, 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`).

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 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.)

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 store word

Primeiramente é testado a função *store word*, em duas áreas de memória consecutivas(índice 0 e 1 do array de memória). Nelas foram escrito, respectivamente, o valor `0x100000FF` e `0x0000CAFE`.

Quando foram printadas as áreas de memória correspondente no array, foram exibidas exatamente esses valores esperados.

3.2 load word

Nessa bateria de testes, foram considerados os casos comuns de leitura, em que foram lidas as duas `words`, anteriormente gravadas, consecutivamente; em seguida, os valores correspondentes são atribuídas a variáveis; e logo depois, essas variáveis são printadas no programa de teste.

No resultado, tanto os valores foram exibidos na ordem em que foram escritas como também apresentam seu valor de retorno correto, demonstrado pela sua conversão em decimal logo ao lado(`0x100000FF` = 268435711; e `0x0000CAFE` = 51966).

Houve teste de casos em que as entradas são inseridas erradas, como no caso de endereços inválidos(não múltiplos de 4), e offsets inválidos(valores diferentes de 0). Nesse caso, os valores retornados são 0, e é reportado uma mensagem de erro correspondente.

3.3 store half word

Foram testados o simples armazenamento de `half words` na área de memória e processos de sobre-escrita.

Ao se verificar o print das áreas de memória testadas, evidencia-se os locais corretos dos valores de inserção(offset 0: `0X00FE` e offset 2: `0XAAFF`) e sobre-escrita(offset 2: `0XCAFE`) dos `half words`, como pode se verificar na execução do teste.

Nos casos de exceção, ou seja, acesso a endereços e offsets inválidos(não múltiplos de 4 e não múltiplos de 2, respectivamente), verifica-se que o valor do endereço não se altera, e é reportado uma mensagem de erro.

3.4 store byte

Nesse teste, foram feitos 4 stores consecutivos em um mesmo endereço(offset 0: `0XFE`, offset 1: `0XCA`, offset 2: `0XBB`, offset 3: `0XAA`), seguida de sobre-escritas nelas(offset 0: `0XAA`, offset 1: `0XBB`, offset 2: `0XCC`, offset 3: `0XDD`).

Como esperado, houve a escrita e sobre-escritas desses valores nos locais corretos(primeiro gera-se `0xAABBCAFE`, depois `0xDDCCBBAA`)

Em seguida, foram feitas mais 4 stores em um endereço consecutivo à anterior(offset 0: `0X04`, offset 1: `0X03`, offset 2: `0X02`, offset 3: `0X01`), gerando o resultado esperado `0x01020304`. Isso mostra que não há erros quando há a transição de endereço.

3.5 Load half word(signed)

Nesse caso, foram realizados duas leituras em half word de um endereço(`word: 0X1111CAFE`, offset 0: `0XCAFE`, offset 1: `0X1111`), no formato com sinal. Como esperado, os valores mostrados são as correspondentes às do offset, e no formato com sinal correto(-13570 e 4369, respectivamente).

Nos casos de exceção, como anteriormente no load word, o valor retornado é 0; seguida da reportação de um erro.

3.6 Load half word(unsigned)

Realizada duas leituras half word do mesmo endereço anterior na mesma forma, espera-se os mesmos valores porém no formato sem sinal. Seguindo a conveção esperada, os valores hexadecimais convertidas em decimal apresenta-se nos seus valores correspondentes (`0XCAFE` = 51966 e `0X1111` = 4369).

Os casos de erro, seguem o mesmo padrão mencionado no teste anterior.

3.7 Load byte(signed)

Foram realizadas 4 leituras em byte da área de memória(`word: 0x04030201`, offset 0: `0X01`, offset 1: `0X02`, offset 2: `0X03`, offset 3: `0X04`), exibindo-se corretamente seus valores correspondentes e em formato com sinal 1, 2, 3, 4; respectivamente.

Em seguida, leu-se mais 4 bytes de memória em um endereço consecutivo, (`word: 0X7711CAFE`, offset 0: `0XFE`, offset 1: `0XCA`, offset 2: `0X11`, offset 3: `0XFE`), exibindo-se os valores corretos em formato com sinal: -2, -54, 17, 119.

3.8 Load byte(unsigned)

O mesmo procedimento foi adotado com a última área de memória referida anteriormente(`word: 0X7711CAFE`), exibindo-se os valores e seus formatos unsigned correspondentes corretamente: 254, 202, 17,119.

3.9 Dump mem

Nesse último teste, foi chamado a função dump mem, verificando não somente seu bom funcionamento como também a corretude dos valores armazenados na memória. Verificando-se as operações listadas acima, de escrita e sobreescrita; pode-se confirmar que todas elas se localizam em áreas correspondentes de memória, ou seja, os valores gravados estão sequencialmente dispostas na memória de acordo com a ordem dos testes listados e que operaram sobre essa

memória(índices 0 e 1, store word; 2, store half words; 3 e 4, store byte; 5, load half; 6 e 7, load byte).