Relatório do Segundo Trabalho de OAC

Simulador do MIPS em C++

Gabriel Araujo - 12/0050943
19/04/2018

Descrição do Problema

MIPS é uma *ISA*(*Instruction Set Architecture*) RISC(*Reduced Instruction Set Computer*) criada por John L. Hennessy. O objetivo de implementar uma simulação de um processador capaz de executar a *ISA* MIPS é entender os estágios e as abstrações necessárias para um *hardware* executar um *software* de maneira correta.

Descrição da Implementação

Classe MIPS

A Classe MIPS definida em *simulator.h* e implementada em *simulator.cpp* é a classe base para a simulação, em software, de um processador MIPS. Nela está implementada as 3 principais funções que simulam um processador de *ISA MIPS*, que são *fetch*(), *decode*() e *execute*(). Essas funções simulam os três estados principais do processador. Também está implementado na Classe MIPS os registradores *pc*, *ri*, *hi* e *lo* como variáveis privadas da classe. Tanto a memória quanto o banco de registradores foi simulado como um array em C de inteiros de 32 *bits* com 4096 e 32 elementos respectivamente. Os campos *opcode*, *rs*, *rt*, *rd*, *shamt*, *funct*, *kte16* e *kte26* também foram implementados como variáveis de classe e apenas *kte26* foi declarado como sem sinal.

Load Memory

Foi implementada uma função para carregar tanto a seção de código(seção *.text* do *assembly*) quanto a seção de dados(*.data* do *assembly*) na memória simulada. Esta é a função *loadMemory*(), esse método recebe 3 parâmetros, os dois primeiro são os caminhos para os arquivos binários de código e de dados, respectivamente, enquanto que o terceiro parâmetro é um booleano e define se haverá impressão das instruções simuladas.

Fetch

A função *fetch*() carrega no registrador *ri* a instrução apontada pelo registrador *pc* a partir da memória de instruções e incrementa o *pc* para a próxima instrução.

Decode

A função *decode*() tem o papel de extrair os campos *opcode*, *rs*, *rt*, *rd*, *shamt*, *funct*, *kte16* e *kte26* da instrução carregada em *ri*. Essa extração é feita por meio de máscaras e todos os campos são extraídos, independente do formato(R, I ou J).

Execute

O método *execute*() descobre qual a instrução deverá ser executada, por meio de um *switch-case*, e realiza a ação necessária para simular a instrução usando os operadores do C/C++.

Print Execute

Foi implementado também o método *printExecute*() que é capaz de imprimir na tela as ações realizadas pelo simulador. Mostrando os valores, registradores, condicionais e acessos efetuados pelo simulador. O formato utilizado para a impressão do código é inspirada no arquivo “MIPS reference data”. Essa impressão é opcional, para desativar a *flag print_instructions* precisa ser falsa.

Run

A função *run*() roda em *loop* a sequência *fetch*(), *decode*() e *execute*() até o registrador *pc* alcançar o limite da memória de código ou até a flag de finalização da simulação for verdadeira. Antes de executar o *loop*, é checado se a memória de instruções está carregada. Se a memória de instruções não tiver sido preenchida a simulação é encerrada.

Step

O método *step*() roda apenas um passo de simulação, isto é, a sequência *fetch*(), *decode*() e *execute*() apenas uma vez. Da mesma forma que em *run*(), o valor do pc, a flag de encerramento da simulação e a memória de instruções são checadas antes de se realizar o passo de simulação.

Dump Mem

A função *dump_mem*() imprime no console em formato sequencial os valores guardados na memória simulada e como pedido recebe 3 parâmetros, o primeiro referente ao endereço inicial de impressão, o segundo referente ao endereço final e o terceiro referente ao formato de impressão dos valores da memória que pode ser decimal ou hexadecimal.

Dump Reg

Assim como o método *dump_mem*(), *dump_reg*() imprime no console o banco de registradores e em sequência os registradores especiais *pc*, *hi* e *lo*. Também é suportado os formatos decimal e hexadecimal de impressão.

Notas

- Os endereços base da seção de código e de dados são 0x0000 e 0x2000, respectivamente, assim como no *MARS* na configuração compacta com .text começando em 0x0;
- Os registradores *global pointer* e *stack pointer* são inicializados com os valores 0x1800 e 0x3FFC, também baseado nos valores de configuração do *MARS*;
- A instrução *addiu* foi implementada com o *opcode* 0x09, essa instrução não é pedida na especificação do trabalho;
- O GDB foi utilizado como ferramenta para verificar a escrita e a leitura na memória simulada.

Testes e Resultados

Para esse trabalho, 3 testes foram implementados. O primeiro é descrito na especificação do trabalho, referente à impressão dos oito primeiros números primos. O segundo teste é uma programa de teste oferecido no site do MARS, esse código em *assembly* calcula os doze primeiros números da sequência de *Fibonacci*, guarda esses números na memória e então os imprime no *console*. O terceiro e último teste foi oferecido pelo professor e tem como objetivo testar todas as instruções implementadas. Todos os testes estão guardados na pasta *test/*, os códigos em

assembly tem extensão *asm* enquanto que os arquivos binários contendo o segmento de código e o segmento de dados tem extensão *bin*. Os arquivos de teste são passados para o simulador por meio de argumentos do programa *simulator*. O caminho até os programas de teste devem ser passados na seguinte ordem: caminho até o arquivo do segmento de código e então o caminho até o arquivo de segmento de dados.

- Exemplo: $ ./bin/simulator test/primos_text.bin test/primos_data.bin

Teste Primos

Este teste tem como objetivo imprimir uma string e um vetor de inteiros a partir da memória. É possível testar as instruções de acesso à memória, saltos condicionais, saltos e operações aritméticas além dos syscalls de impressão de inteiros, impressão de string e saída do programa.

Teste Fibonacci

Neste teste a sequência de *Fibonacci* é calculada até o décimo segundo elemento. Assim como no teste anterior, instruções de manipulação da memória, saltos condicionais e operações aritméticas são utilizadas.

Teste de Instruções

A papel desse teste é executar todas as instruções implementadas e verificar se a execução foi bem sucedida. Se alguma instrução não for executada corretamente será impresso uma mensagem de erro e o número do teste que falhou.

Resultado dos testes

Todos os testes devem passar sem quaisquer erros. O primeiro teste deve imprimir de maneira correta uma *string* e o vetor de inteiros primos. O segundo deve imprimir uma *string*, calcular corretamente a sequência e imprimi-la na tela. Já o terceiro além de realizar todas as instruções sem erro deve imprimir uma *string* afirmando quais teste foram bem sucedidos. Todos os resultados esperados e o estado final do banco de registradores estão armazenados em forma de arquivo texto na pasta *test/expected*.

Notas

- A opção *test_simulator* do Makefile fornecido roda automaticamente os três testes acima e compara a saída dos testes com a saída esperada de cada um;