MO601 - Projeto 1

Casio Pacheco Krebs - ra264953

[1]Instituto de Computação - IC
Universidade Estadual de Campinas - UNICAMP
Campinas, Brasil

1. Introdução

Este projeto tem como finalidade a criação de um simulador de processador utilizando como base o conjunto de instruções RISC-V RV32IM. Esse conjunto de instruções é composto pelas instruções básicas de 32 bits, representadas pelo RV32I, e pela extensão M, que possibilita a execução de instruções de multiplicação e divisão. O objetivo desse simulador é permitir a realização de Debug dos kernels. Com isso, é possível visualizar as instruções simuladas e os valores dos registradores associados a elas.[1].

A descrição desse relatório está separado em Execução (Seção 2), onde são apresentados as dependências e o seu procedimento de instalação e execução, Implementação (Seção 3), onde são detalhados os métodos e funções utilizadas, Testes (Seção 4), ande são apresentadas as metodologias dos testes executados, e por fim, Conclusão (Seção 5), onde são apresentados os aprendizados tirados por este projeto.

2. Execução

Nessa seção estão descritos as etapas da execução do projeto, Para a sua implementação foi utilizando a linguagem de programação *Python* em sua versão 3.9.

2.1. Dependências

Para a execução do projeto deve-se ter instalado as seguintes bibliotecas:

- git: utilizado para realizar o download dos arquivos a partir do GitHub;
- python3.9: para realizar a execução do script desenvolvido;
- os.path: Utilizado para a manipulação de diretórios;

2.2. Como executar

O código fonte do simulador desenvolvido neste projeto está disponível no repositório gerado pelo GitHub, acessível através do link: https://github.com/CPKrebs/MO601-P2.git . Todos os arquivos necessários para a execução do simulador estão armazenados neste repositório, incluindo o código-fonte do simulador e os códigos-base das aplicações que serão testadas..

Para a correta execução do simulador desenvolvido, é necessário ter configurado o compilador **gcc-riscv64-linux-gnu**. Além disso, os códigos-base das aplicações escritos em C devem estar dispostos no diretório $'/MO601 - P2/test/'$. O simulador foi projetado para realizar o debug de todos os códigos existentes nesse diretório, de forma contínua.

Para gerar os executáveis dos kernels a partir dos códigos C armazenados no diretório mencionado acima, basta executar o comando **make** a partir do arquivo Makefile que se encontra dentro da pasta "/*MO*601 – *P*2/*test*/". Esse comando utiliza o compilador **riscv64-linux-gnu-gcc** para gerar os executáveis, que serão armazenados na mesma pasta. Em seguida, é utilizado o **riscv64-linux-gnu-Objdump** para extrair as instruções em representação hexadecimal e combiná-las com o endereço correspondente. Os arquivos gerados são armazenados na pasta "/*MO*601 – *P*2/*assemble*/".

A saída do programa é gerada na forma de um arquivo de debug, cujo nome é formado pela combinação do nome da aplicação com ".log"no final. Os arquivos gerados são salvos na pasta "/*MO*601 – *P*2/*test*/". Além disso, o programa exibe no terminal o tempo gasto para a simulação de cada aplicação executada.

Para facilitar a execução do simulador, foi desenvolvido um script denominado **compilar-e-executar.sh**, que realiza todo o processo de compilação das aplicações, extração dos hexadecimais, execução do simulador e limpeza dos arquivos temporários. Esse script está localizado na raiz do repositório e pode ser executado de forma simples. Com isso, é possível realizar os testes de forma automatizada e eficiente.

3. Execução

Nessa seção serão descritos as etapas do fluxo de execução do simulador, separadas em preparação dos códigos de entrada e execução do scpity desenvolvido.

3.1. Implementação

O script do simulador de processador RISC-V desenvolvido, se inicia identificando as representações em hexadecimal dos kernel existentes na pasta '/*MO*601 – *P*2/*assemble*/' através do da função os.listdir(), e na sequência, o código abre o arquivo de código assembly a ser executado e abre um arquivo de log para armazenar informações sobre a execução do simulador.

São desenvolvidos um array, denominado de **RG**, que representa o banco de registradores, e um dicionário, denominado de **Mem**, que representa a memória mapeável pelo processador. Em seguida é implementado um loop responsável por carregar as instruções do arquivo de código assembly na memória, identificando seu endereço e armazenando seu conteúdo. Nessa etapa também é identificado o endereço da instrução de início do código assembly.

Foi implementado um loop que realiza o processo de busca, decodificação e execução de cada instrução. A busca é feita incrementando o valor do contador de programa (PC) em 4, que representa o tamanho de uma instrução na arquitetura RISC-V. A decodificação é feita lendo a instrução em hexadecimal da memória, convertendo-a para binário, em seguida são identificando os campos da instrução, como opcode, funct3, funct7, registradores de destino e origem. Finalmente, o valor dos registradores de origem são carregados conforme os registradores indicados na instrução e armazenados em variáveis auxiliares, para registar o valor do registrador antes da execução da instrução.

As instruções são identificadas por meio dos campos opcode, funct3 e funct7 (se aplicável) e, em seguida, a execução apropriada é realizada, incluindo cálculo e atribuição de valores para registradores e/ou mudança no valor do PC. Cada instrução do conjunto RV32IM é tratada separadamente em um bloco if-elif. Para todas as instruções, a saída é registrada em um arquivo de log para fins de depuração.

Para operações envolvendo acessos á memória (Load e Store), primeiro é identificado o comprimento da operação por meio do campo funct3. No caso do LOAD WORD, que movimenta 4 bytes, a palavra a ser escrita é dividida por bytes, que são armazenados em endereço consecutivo com base no endereço calculado a partir do registrador informado e do imediato associado.

As instruções LUI e AUIPC são usadas para carregar valores imediatos de 32 bits nos registradores. A instrução LUI carrega os 20 bits mais significativos de uma constante imediata nos bits 31:12 de um registrador destino, enquanto os bits 11:0 são preenchidos com zeros. A instrução AUIPC é semelhante, mas soma o valor do PC atual aos 20 bits mais significativos da constante imediata e carrega o resultado no registrador destino.

As instruções JAL e JALR são usadas para controlar o fluxo de execução do programa. A instrução JAL salta para um destino calculado adicionando um deslocamento imediato ao endereço do PC atual. O endereço do PC atual é armazenado no registrador destino, e o PC é atualizado com o novo endereço de destino. A instrução JALR é semelhante, mas o endereço de destino é calculado adicionando o valor de um registrador fonte ao deslocamento imediato.

As instruções de desvio condicional são beq, bne, blt, bge, bltu e bgeu. Após a verificação do opcode, cada uma dessas operações é identificada pelo valor de funct3, em seguida é checado a condição especificada pela operação e, caso ela seja verdadeira, o valor de PC é atualizado para apontar para a instrução que será executada em seguida.

Para tratar as instruções do Tipo R, compostas pelas instruções aritméticas ADD, SUB, SLL, SLT, SRL, SLTU, SRA, MUL, MULH, MULHSU, MULHU, DIV, DIVU, REM e REMU, e pelas instruções lógicas XOR, OR e AND, precisam ser verificadas os campos opcode, funct7 e funct3, para verificar a operação a ser realizada. Por fim, a rotina correta é executada sobre os registradores especificados nos campos rs1 e rs2. O resultado é armazenado no registrador especificado no campo rd.

Em caso de operações envolvendo imediatos, o código extrai a constante da instrução e a converte de binário para inteiro e, em seguida, analisa se o bit mais significativo é 1, caso positivo isso indica que o número é negativo, então o código faz um ajuste para converter o número em complemento a dois.

A instrução EBREAK foi implementara para encerrar o a execução da simulação, para isso após a identificação do opcode, funct7 e funct3, o laço que realiza o processo de busca é encerrado e o controle é repassado para a função Main, que ira apresentar o tempo gasto na simulação da aplicação e, em seguida, ira carregar a próxima aplicação para simulação, caso contrário, a execução do

scripy se encerra.

4. Metodologias de testes e avaliação

Para a realização da corretude do simulador primeiro foi realizado um teste unitário para avaliar a implementação de cada instrução, nesse teste realizados em python, foram aplicadas quatro entradas. A primeira com os dois registradores de entrada com os 32 bits em nível lógico 0 (0x00000000). A segunda com os dois registradores em nível lógico 1 (0xFFFFFFFF). A terceira e a quarta, são referentes as combinações de apenas um registrador em nível lógico 0 , enquanto o outro está em nível lógico 1. Esse teste tinha como objetivo verificar o comportamento individual da instrução nas bordas de operação. O mesmo procedimento foi realizado para as instruções que envolvam imediatos, nesse caso, as bordas foram limitadas pela largura de bits do imediato.

Em seguida foi realizado um teste funcional, onde foi aplicado o conjunto de aplicações do benchmark ACStone [2], excluindo os que requisitam instruções atômicas ou de ponto flutuante. Para todos as entradas foi realizado a checagem manual das informações dispostas no arquivo log, permitindo constatar a correta execução das aplicações.

Também foram analisados os tempos de execução e o número de instruções executadas, onde foi possível constatar que a aplicação 000.main, é a mais rápida das analisadas, com apenas 12 instruções executadas, precisando de apenas 216 microssegundos para ser completamente simulada, enquanto a aplicação 142.array, se apresentou como a mais demorada com 87201 instruções executadas, precisando de 746 milissegundos para ser completamente simulada.

5. Conclusão

O projeto de simulador de processador RISC-V permitiu um estudo detalhado do comportamento de cada instrução do processador de forma isolada. O estudo incluiu a compreensão de como operações que apresentam overflow no resultado devem ser tratadas, especialmente em operações de MUL e REM. Também foi estudado como as variáveis char, short e long int são representadas em hardware de 32 bits e como as operações com sinal e sem sinal são tratadas.

Durante o desenvolvimento enfrentei várias dificuldades na validação das rotinas de cada instrução. Foi necessário realizar a validação de cada instrução isoladamente e com entradas que cobrissem todos os casos possíveis, uma vez que foram encontradas inconsistências em operações envolvendo variáveis char ou short, principalmente quando ocorria overflow. Neste caso, o uso do benchmark ACStone facilitou a visualização dos casos de exceção, permitindo calibrar as rotinas para o comportamento esperado.

6. Código

O DockerFile completo, será entregue juntamente com este relatório na plataforma GOOGLE Classroom.

Referências

[1] https://www.ic.unicamp.br/ rodolfo/mo601/projeto2/, acessado em 24/04/2023

[2] https://github.com/rjazevedo/ACStone, acessado em 24/04/2023