MO601 2023S1 - Relatório do Projeto 2 - Simulador de RISC-V

> Felipe Martins Romeiro - 215720
> Vinícius Waki Teles - 257390

Introdução

Nesse projeto implementamos um simulador de RISC-V 32IM em Python 3.10. Ele lê os binários extraídos dos ELFs resultantes da compilação de código C para RISC-V e gera arquivos de log listando os valores dos registradores e o disassembly para cada instrução.

Execução

O simulador foi programado e testado com Python 3.10.10. O script de preparação dos testes depende do comando `riscv64-linux-gnu-objcopy`.

Para executar o projeto, primeiro transformamos os arquivos ELF compilados para arquivos binários que o simulador consegue interpretar. Para isso, executamos o comando a seguir, substituindo `[dir]` pela pasta onde os arquivos ELF (de extensão .riscv) estão localizados.

```
./build.sh [dir]
```

Esse comando vai colocar os binários extraídos dos ELFs no caminho `test/build/bin`.

Depois de rodar o script, podemos executar o simulador com o comando a seguir

```
python src/main.py
```

O simulador irá buscar pelos binários na pasta `test/build/bin` e irá colocar os arquivos de saída com a extensão `.log` na pasta `test/`.

Algoritmo

Ao compilar um arquivo C usando a toolchain do RISC-V, obtemos um arquivo ELF com várias informações e cabeçalhos além das seções de código. Para facilitar o trabalho de execução do código do simulador, optamos por extrair do ELF as seções que são transferidas para a memória usando o comando `objcopy -O binary [...]`. Para preparar a execução do código C, o compilamos indicando o runtime fornecido no repositório do ACStone.

Inicialmente, pensamos em implementar o simulador nos baseando em um pipeline escalar de 5 estágios - Instruction Fetch, Instruction Decode, Execution, Memory, Writeback. Porém, nós tivemos problemas na hora de dividir o trabalho e implementar a comunicação entre as etapas do pipeline. Optamos então por dividir as instruções pelo tipo, o que facilita a decodificação da instrução, e fornecer as interfaces necessárias para que elas possam se comunicar com o banco de registradores e com a memória.

No nosso algoritmo criamos abstrações para a memória, para o banco de registradores e para o cache de instruções.

- A memória é internamente representada como um dicionário que mapeia endereços a bytes e expõe interfaces para ler e escrever o valor de bytes. Os bytes são armazenados como strings de '0's e '1's, pois isso facilita a decodificação de instruções e a reordenação dos bytes para mudança de endianness e outras manipulações. A memória poderia ser mais eficientemente representada como um vetor esparso de sequência de bytes, já que a memória tende a ser dividida em seções (código, heap, stack), mas para os programas do ACStone não encontramos problemas de performance, então optamos por uma implementação mais simples.
- O banco de registradores é representado como um dicionário que mapeia o índice do registrador a um valor inteiro.
- A cache de instruções é uma abstração por cima da memória, carregando as instruções do arquivo binário na memória no início da simulação. As instruções são colocadas na memória a partir de um dado offset, pois no próprio ELF elas têm esse offset que é considerado pelas instruções de LOAD, STORE e JUMP. Ela também realiza a leitura de instruções a partir do endereço, fazendo a transformação de little endian para big endian, o que facilita a decodificação de cada parte da instrução.

Além disso, nosso código possui outras três estruturas, a Instruction, o Decoder e o Simulator.

- A Instruction é uma classe abstrata que interpreta a instrução indicada, calcula os valores comuns as instruções, como os registradores de entrada e o opcode, monta o seu log e armazena as abstraçõe de memória e banco de registradores para que sejam utilizados na execução da instrução. Ela ainda expõe os métodos abstratos para obter o nome, o disassembly, a execução da instrução e se é uma instrução de fim da simulação (indicado pelo `ebreak`). Esses métodos são implementados pelas classes filhas de Instruction, que representam os diferentes tipos de instrução, e portanto podem implementar as especificidades de decodificação de instrução de cada classe, como a representação do imediato e a presença de funct3 e funct7.
- O Decoder recebe uma instrução lida pelo cache de instruções, identifica o tipo da instrução, instancia a classe correta e a retorna.
- O Simulator lê o valor de PC do banco de registradores, lê a instrução correspondente do cache de instruções, usa o Decoder para obter a instância de Instruction, a executa e escreve seu log no arquivo de saída. Ele realiza esse processo até encontrar o `ebreak`.

A main itera pelos binários na pasta de teste e, para cada um, cria novas instâncias de memória, banco de registradores e cache de instruções e invoca o Simulator para que execute o teste até o fim.

Testes

Para testar executamos os códigos do ACStone pelo simulador e observamos que todos eles terminavam a execução. Também selecionamos exemplos específicos e comparamos o resultado do simulador com os comentários do ACStone.

Conclusão

Neste projeto estudamos as ferramentas `objdump` e `objcopy`, relevantes para investigar e manipular arquivos executáveis, para entender o formato dos arquivos ELF e para transformá-los em um formato mais fácil de interpretar. Também estudamos a documentação do RISC-V para descobrir quais instruções seriam implementadas, como elas são codificadas, quais seus comportamentos específicos e qual o seu modelo de memória.

Além disso também conseguimos ter noções gerais de questões envolvidas em compiladores, como representação de memória e conversão entre tipos.