Relatório - Simulador RV32I em C++

Nome: Luiz Carlos Schonarth Junior

Matrícula: 19/0055171

Universidade de Brasília - UnB

Objetivos

Esse trabalho tem como objetivo o estudo da arquitetura do processador RISC-V e da implementação do seu *instruction set* da versão de 32 bits.

Metodologia

Para implementação do simulador RV32I, foi utilizada uma linguagem de programação de alto nível, mais especificamente linguagem C++, com operações que podem interagir com os dados de bit a bit.

O software deve ler arquivos binários gerados pelo simulador RARS de um conjunto de instruções e dados criados após a montagem do código Assembly. Esse dado binário é inserido em conjuntos de 32 bits dentro de um array criado em software.

Foi necessário configurar a memória no simulador RARS para utilizar o modo compacto, endereçando o segmento de texto (código) em 0x0000 e o segmento de dados em 0x2000 em valores hexadecimais. Com essa informações, basta ler os arquivos binários gerados dentro do programa em C++ usando uma função ***loadmem( )*** e armazenar as informações de acordo com endereçamento descrito anteriormente.

Com os valores da memória carregados, agora é necessário capturar a palavra de 32 bits da memória endereçada pelo valor do registrador *PC (Program*

Counter), que tem valor inicial 0. A função que lê a palavra da memória endereçada por *PC* é a função ***fetch( )***.

Após isso, deve-se decodificar a palavra bits carregada da memória. Essa é a responsabilidade da função ***decode( )***. Nessa função, utilizando o mascaramento de bits, são lidos os 7 primeiros bits da palavra, o que corresponde ao código da operação (OPCODE), capaz de identificar qual o tipo de instrução que está representada na palavra, restringindo o número de instruções que a palavra pode representar.

Com a informação de tipo de instrução, é possível extrair informações como: registradores (*rd*) de destino e fonte (*rs1, rs2*), funções (*funct3, funct7*) e imediatos (*imm*), como representa a tabela a seguir:

CORE INSTRUCTION FORMATS

	31 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	Rd	opcode
I	imm[11:0]		rs1	funct3	Rd	opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
B	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				Rd	opcode
J	imm[20|10:1|11|19:12]				Rd	opcode

Armazenando essas informações a respeito da instrução em variáveis globais e identificando unicamente qual instrução a palavra diz respeito usando essas informações, agora é possível executar a instrução.

A função ***execute( )*** tem a responsabilidade de executar a instrução decodificada da palavra. Agora que as informações sobre a instrução já estão armazenadas, basta checar nas variáveis globais qual delas deve ser executada e executá-la, usando sua implementação em C++.

Após a execução da instrução contida na palavra, deve-se incrementar o *PC* de forma que este valor "aponte" para a próxima instrução. Feito isso, repete-se o ciclo *fetch( ), decode( ), execute( ), incrementa PC* até o final do programa, indicado pela instrução *ecall* com um valor específico armazenado no registrador *a7*. A seguir é possível ver as funcionalidades (*Syscalls*) implementadas para instrução *ecall* nesse simulador RV32I:

Função	Valor de a7	Saída
Imprime Inteiro	1	Valor de a0 em inteiro
Imprime String	4	String endereçada em a0
Encerra Programa	10	N/A

A implementação de instruções de baixo nível em linguagens de programação de alto nível não acontece sem inconveniências. Uma delas é a importância de tipagem de dados. Em linguagem de máquina, dados são um interpretados como um conjunto de bits; em C++, a implementação de operações pode depender da tipagem dos operandos, provocando resultados inesperados, especialmente quando se trata de tipo *signed* ou *unsigned*.

Um exemplo desse comportamento são as operações de *shift* aritmético e lógico de bits para a direita. Ambos em C++ são implementados usando o mesmo operador (>>), porém, qual operação é usada em qual situação depende da implementação que o compilador usar.

Usando o compilador *g++*, o *shift* aritmético para a direita é usado por padrão para tipos de dados como *int32_t* (inteiro de 32 bits). Porém, usando a tipagem *uint32_t* (inteiro sem sinal de 32 bits), o compilador executa um *shift* lógico para a direita de bits. Dessa forma, como os dados no interior da memória e registradores do simulador são do tipo *int32_t*, as instruções *srl* e *srli* do RV32I (instruções de shift lógico para a direita) precisam que os operandos passem por um *cast* de *int32_t* para *uint32_t*.

Conclusão

Diante do que foi apresentado, pode-se observar que o uso da linguagem de programação C++ para implementação de um simulador RV32I é mais convidativo que outras linguagens, já que C++ conta com operações que operam bit a bit e com tipagem robusta, permitindo uma melhor compreensão do passo a passo da execução de instruções em um processador como o RISC-V.

Referências bibliográficas

1 - David A. Patterson, John L. Hennessy, Computer Organization and Design RISC-V Edition: The Hardware Software Interface, Second Edition

2 - David Patterson, Andrew Waterman, Guia Prático RISC-V, Atlas de uma Arquitetura Aberta