Universidade Federal de Lavras

Departamento de Ciência da Computação
Bacharelado em Ciência da Computação
Disciplina: Arquitetura de Computadores II – GCC 123 – 2023/1
Professor: Luiz Henrique A. Correia

TRABALHO PRÁTICO 1

Alunos :

Bryan De Lima Naneti Barbosa
Gustavo Soares Silva
Rafael Furtado Moraes
Ranulfo Mascari Neto

Lavras - MG
2023

Sumário

1. Introdução

Este trabalho prático tem como objetivo simular o funcionamento de um processador RISC (Reduced Instruction Set Computer). Através dessa simulação, buscamos compreender os princípios e conceitos fundamentais por trás dessa arquitetura, assim como explorar a lógica e o funcionamento interno de um processador RISC.

A arquitetura RISC é conhecida por sua abordagem de conjunto de instruções reduzido, que prioriza instruções simples e básicas. Essa abordagem permite que o processador execute operações de forma mais rápida e eficiente, ao mesmo tempo que simplifica o projeto e a implementação do hardware.

Para a simulação do processador RISC, optamos por utilizar a linguagem de programação C++, aproveitando a sua flexibilidade e recursos adequados para a implementação de um projeto complexo como esse. Além disso, nossa familiaridade com a linguagem nos permitirá explorar de forma mais eficaz os detalhes e desafios envolvidos na construção de um simulador de processador RISC.

2. Resumo da Máquina

O simulador do processador é formado pelas seguintes etapas:

- Interpretador De Instruções: Nessa etapa é lido um arquivo txt e então é traduzido para o formato de instruções do RISC.

- Instruction Fetch: Nessa etapa a instrução é lida e é incrementado o valor de PC após cada leitura.

- Instruction Decoder: Nessa etapa a instrução é decodificada para ver qual operação será feita e os operandos envolvidos. Também são definidos quais registradores serão usados e as flags são atribuídas de acordo com cada instrução.

- Execução/Memória : Nessa etapa as operações são realizadas, e se envolver acesso a memória (leitura/escrita) também é feito nessa etapa.

- Write Back : Nessa etapa o resultado da instrução é escrito no registrador especificado pela própria instrução.

3. Implementação

A primeira parte da implementação foi o desenvolvimento do Interpretador de Instruções. Para isso, utilizamos a estrutura de Mapa, da biblioteca "Map", para identificar os Opcodes de cada instrução e como cada uma é classificada. Também foi necessário usar a biblioteca Bitset para armazenar as instruções decodificadas em 32 bits. Cada instrução é decodificada individualmente, por linha, e armazenada no vetor de 64K na classe CacheL1Instrucoes.

Em seguida foi criado o arquivo do processador, em que a chamada de todas as funções são feitas nele. Mas, durante o desenvolvimento desse arquivo, viu-se necessário criar um cabeçalho para armazenar o Banco de Registradores, a Memória (CacheL1Dados e CacheL1Instrucoes) e outras funções que seriam usadas por cada uma das etapas do processador. Nessas classes foram criados métodos “getRegistrador”, para retornar um valor armazenado em um endereço, e “setRegistrador”, para armazenar um valor em um endereço.

No início ficamos em dúvida sobre a separação de cada etapa do processamento, pois teríamos que retornar vários valores de uma vez. Dessa forma, ao invés de usar funções, optamos por usar classes e structs. A entrada dos valores foi feita pelo método “executar” de cada classe, para evitar a criação de objetos, e a saída foi feita pelos atributos, que são utilizados como parâmetros pela próxima etapa do processamento. Além disso, para as operações de soma, subtração, multiplicação, entre outras, foram feitas sobrecargas dos operadores “+”, “-”, “*”, “/” e “<” para a classe bitset. E também um conversor de bitset para inteiro.

Também foram feitos métodos de “recorte”, que retorna um pedaço de um bitset, “signalExtension”, para estender um bitset de 16b para 32b, e “bitsetToInt”, que converte o valor de um bitset para inteiro. E, para contornar um problema que encontramos durante o desenvolvimento, tivemos que modificar uma parte do caminho de dados do monociclo. Para executar todas as operações na execução, foi necessário usar 5 bits para o ALUOp. E o cálculo do desvio não é feito no ID, e sim na Execução.

4. Instruções Projetadas

- addi (soma com imediato):

  A instrução addi é utilizada para realizar uma soma entre um valor de um registrador e um valor imediato (constante). Ela adiciona o valor imediato ao valor do registrador especificado e armazena o resultado em um registrador de destino.

- subi (subtração com imediato):

  A instrução subi realiza uma subtração entre um valor de um registrador e um valor imediato. Ela subtrai o valor imediato do valor do registrador especificado e armazena o resultado em um registrador de destino.

- mul (multiplicação inteira):

  A instrução mul é utilizada para realizar uma multiplicação inteira. Ela multiplica o valor de dois registradores e armazena o resultado em um registrador de destino.

- div (divisão inteira):

  A instrução div é utilizada para realizar uma divisão inteira. Ela divide o valor de um registrador pelo valor de outro registrador e armazena o resultado da divisão no registrador de destino.

- storei (armazenar inteiro na memória):

  A instrução storei é utilizada para armazenar um valor inteiro em um endereço de memória especificado. Ela copia o valor para a posição de memória indicada.

- bgt (branch greater than):

  A instrução bgt é um desvio condicional. Ela compara dois valores de registradores e, se o primeiro valor for maior que o segundo, desvia o fluxo de execução para uma determinada instrução especificada.

- blt (branch less than):

  A instrução blt é um desvio condicional similar ao bgt, porém desvia o fluxo de execução se o primeiro valor for menor que o segundo.

- nor (or + not/passnota):

  A instrução nor é uma operação lógica que realiza uma operação OR bit a bit entre dois registradores e, em seguida, realiza uma negação lógica (NOT) no resultado. O resultado final é armazenado em um registrador de destino.

- nand (and + not/passnota):

  A instrução nand é uma operação lógica que realiza uma operação AND bit a bit entre dois registradores e, em seguida, realiza uma negação lógica (NOT) no resultado. O resultado final é armazenado em um registrador de destino.

5. Como usar o software

Para utilizar o software é necessário seguir alguns passos.

Primeiramente, escreva o código que será executado em um arquivo “.txt”. O código escrito deverá seguir o padrão descrito abaixo (utilize os exemplos para entender melhor).

- Não utilize tabs ou espaços antes de qualquer instrução. As instruções devem ser escritas “coladas” na parte esquerda, sem espaços ou tabs.

- Os registradores são referenciados com um “R” seguido do número do registrador, logo depois uma vírgula, e se tiver algo à frente coloque um espaço para separar (Ex: add R2, R2, R3).

-Não utilize nome para labels, apenas defina quantas linhas deverá pular ou voltar (Ex : beq R1, R2, 4)

- Não coloque linhas vazias entre instruções, mantenha uma instrução por linha.

Em seguida execute o arquivo “Principal.cpp” e digite no Terminal o nome do arquivo que contém as instruções para serem executadas pelo simulador do processador Risc.

6. Estruturas Utilizadas

Na classe Interpretador :

- Classe Interpretador: É uma classe que contém métodos para converter as instruções em linguagem assembly em instruções binárias.

- Mapas (opcodeMap e tipoInstrucaoMap): São estruturas de dados do tipo mapa (map) que mapeiam as instruções assembly para seus respectivos códigos binários e tipos de instrução.

- Métodos de conversão: A classe Interpretador contém uma série de métodos para converter cada tipo de instrução assembly em instrução binária. Esses métodos recebem a instrução assembly como entrada e retornam a instrução binária correspondente.

- Estruturas condicionais: Dentro do método "converter", existem estruturas condicionais (if-else) que verificam o tipo de instrução assembly e chamam o método correspondente de conversão.

- Uso de bibliotecas: O código inclui várias bibliotecas, como "fstream" para manipulação de arquivos, "string" para manipulação de strings, "map" para uso de mapas e "algorithm" para a função "remove".

Na classe Processador :

- Classe Processador: É uma classe que representa o processador e contém os seguintes membros de dados:

- rb (BancoRegistradores*): Um ponteiro para uma instância da classe BancoRegistradores, que representa o banco de registradores do processador.

- dados (CacheL1Dados*): Um ponteiro para uma instância da classe CacheL1Dados, que representa a cache de nível 1 de dados do processador.

- instrucoes (CacheL1Instrucoes*): Um ponteiro para uma instância da classe CacheL1Instrucoes, que representa a cache de nível 1 de instruções do processador.

- Construtor: O construtor da classe Processador recebe um nome de arquivo como argumento. Ele instancia os membros de dados, cria uma instância do Interpretador para converter as instruções do arquivo em instruções binárias e inicia um loop que simula a execução do pipeline de cinco estágios.

- Variável pc: É uma variável do tipo "unsigned short" que representa o contador de programa.

- Instanciação dos estágios : O código cria instâncias das classes IF, ID, EXMEM e WR, que correspondem aos estágios do pipeline (Fetch, Decode, Execute/Memory, Write Back).

- Loop de execução do pipeline: O loop while simula a execução contínua do pipeline até que não haja mais instruções a serem buscadas na memória de instruções. Dentro do loop, os seguintes passos são realizados:

    - O estágio IF executa o estágio de busca de instrução, passando a instrução buscada para o estágio ID.
    - O estágio ID executa o estágio de decodificação da instrução, calculando os valores dos registradores de origem (value_Ra e value_Rb) e outras informações relevantes para o processamento da instrução.
    - O estágio EXMEM executa o estágio de execução e acesso à memória, recebendo os valores dos registradores de origem (value_Ra e value_Rb), o endereço de memória (endereco), as informações de controle (alu_op, alu_src, branch, jump, mem_read, mem_write) e o contador de programa (pc).
    - O estágio WR executa o estágio de escrita nos registradores, recebendo as informações de controle (reg_write, Write_Adrr) e o resultado da operação executada no estágio EXMEM.

Na classe Instruction Fetch (IF) :

- Classe IF: É uma classe que representa o estágio de busca de instrução . Ela possui os seguintes membros de dados:

- instrucao (bitset<32>): Uma variável do tipo bitset de 32 bits que armazena a instrução buscada no estágio de busca.

- Função executar: É uma função pública que realiza a execução do estágio de busca de instrução. Ela recebe dois parâmetros: instrucoes, um ponteiro para uma instância da classe CacheL1Instrucoes, que representa a cache de nível 1 de instruções do processador, e pc, uma

referência a uma variável unsigned short que representa o contador de programa.

- Funcionalidade da função executar: A função executar busca uma instrução da cache de instruções (instrucoes) no endereço indicado pelo valor atual do contador de programa (pc). Em seguida, incrementa o valor de pc em 1 para apontar para a próxima instrução.

Na classe Instruction Decoder (ID) :

- Struct ControlUnit: Essa estrutura representa a unidade de controle do processador. Ela possui diversos sinais de controle, como jump, branch, mem_read, mem_write, alu_src, reg_write e reg_dst, que são utilizados para controlar as operações do processador durante a execução de uma instrução. Além disso, a estrutura possui membros de dados como alu_op (bitset<5>) e endereco (bitset<16>).

- Função verificaTipoR: É uma função que verifica se o opcode da instrução pertence ao tipo R (registro-registro). Ela recebe o opcode (bitset<8>) como parâmetro e retorna true se o opcode corresponder a uma instrução do tipo R e false caso contrário.

- Função verificaTipoB: É uma função que verifica se o opcode da instrução pertence ao tipo B (branch). Ela recebe o opcode (bitset<8>) como parâmetro e retorna true se o opcode corresponder a uma instrução do tipo B e false caso contrário.

- Função setControlUnit: É uma função que configura os sinais de controle da unidade de controle com base no opcode da instrução e na instrução completa. Ela recebe o opcode (bitset<8>) e a instrução completa (bitset<32>) como parâmetros. Dentro da função, são realizadas várias verificações para determinar os sinais de controle corretos com base no tipo de instrução. Os sinais de controle são configurados de acordo com as regras do conjunto de instruções específico.

- Classe ID: Essa classe herda a estrutura ControlUnit e representa a etapa de decodificação de instrução (Instruction Decode) do pipeline. Ela possui os seguintes membros de dados:

    - value_Ra (bitset<32>): Armazena o valor do registrador Ra.
    - value_Rb (bitset<32>): Armazena o valor do registrador Rb.
    - Write_Adrr (bitset<8>): Armazena o endereço de escrita para a instrução atual.

- Função executar: É uma função que executa a etapa de decodificação de instrução. Ela recebe a instrução (bitset<32>) e uma referência para um objeto BancoRegistradores como parâmetros. Dentro da função, são realizados recortes na instrução para obter os campos relevantes, como o opcode e os registradores. Em seguida, são chamadas as funções da estrutura ControlUnit para configurar os sinais de controle corretos. Além disso, os valores dos registradores Ra e Rb são obtidos do objeto BancoRegistradores e o endereço de escrita é determinado com base no controle de destino de registro (reg_dst).

Na classe Execução/Memória (EXMEM) :

- Atributos:
    - ovf: Indica se ocorreu overflow durante a execução.
    - Zero: Indica se o resultado da operação é zero.
    - carry: Indica se ocorreu carry durante a execução.
    - neg: Indica se o resultado da operação é negativo.
    - result: Armazena o resultado da operação em formato de bitset de 32 bits.

- Método executar:Responsável por executar as etapas de execução, memória e exibição de dados.
  Recebe como parâmetros os valores dadoRa e dadoRb, endereco, ALUOp, ALUSrc, branch, jump, pc, mem_read, mem_write e um ponteiro para CacheL1Dados.
  Chama os métodos resetar, execucao, memoria e exibirDados.

- Método resetar: Reinicia os atributos da classe, setando-os para seus valores iniciais.

- Método execucao:Realiza a etapa de execução da instrução.
  Recebe como parâmetros referências para dadoRa, dadoRb, endereco, ALUOp e pc.
  Verifica o valor de ALUOp para determinar a operação a ser executada.
  Atualiza o valor de result com o resultado da operação.
  Verifica se a instrução é de salto (jump) e atualiza o valor de pc de acordo com a operação.
  Verifica se a instrução é de desvio condicional (branch) e atualiza o valor de pc de acordo com a operação e o resultado da comparação.

- Método ALU:Realiza a operação da Unidade Lógica Aritmética (ALU) com base no valor de ALUOp.
  Recebe como parâmetros referências para dadoRa, dadoRb e ALUOp.
  Atualiza o valor de result com o resultado da operação.

:

- Método exibirDados:Exibe os valores dos registradores dadoRa, dadoRb e result em formato binário e decimal.

- Método memoria:Realiza operações de leitura e escrita na memória (cache) de acordo com os valores de mem_read e mem_write.
  Recebe como parâmetros mem_read, mem_write, dadoRa, dadoRb e um ponteiro para CacheL1Dados.
  Caso mem_read seja verdadeiro, lê o valor da memória com base no endereço contido em dadoRa e armazena em result.
  Caso mem_write seja verdadeiro, escreve o valor de dadoRa na memória com base no endereço contido em dadoRb.

Na classe Write Back (WR) :

- Funçao executar: Uma função que recebe vários parâmetros (reg_write, reg_dst, rb, result) e executa a operação de escrita em um banco de registradores (BancoRegistradores). Essa função verifica se a flag reg_write está ativada e, em caso afirmativo, atualiza o registrador especificado por reg_dst com o valor contido em result. Em seguida, exibe no console a informação sobre o registrador atualizado.

7. Testes

Esta seção apresenta os testes realizados durante o desenvolvimento. O objetivo dos testes foi garantir o correto funcionamento do processador, validando suas funcionalidades principais e identificando possíveis erros ou comportamentos inesperados.

Para uma devida validação do simulador, foi necessário o desenvolvimento de uma biblioteca para a realização dos testes, chamada “Teste”, onde de forma intuitiva é possível identificar falhas e sucessos, semelhante a utilizada em outras linguagens de programação como Java na biblioteca JUnit.

A biblioteca “Teste”, possui métodos de validação de resultado real e resultado esperado, onde é possível afirmar (“assert”) o bom funcionamento do simulador utilizando os métodos “assertEqual”, “assertTrue” e “assertFalse”, onde é passado o valor resultante de uma instrução ou registrador e o valor esperado, assim garantindo a confiabilidade dos resultados de um programa.

1. Execução de Instruções

Foram realizados testes para garantir a execução correta de algumas das instruções básicas, como operações aritméticas, lógicas, deslocamento, controle de fluxo e movimentação de dados. Cada instrução foi testada individualmente e em conjunto (teste de programas), fornecendo diferentes valores de entrada e verificando se o resultado correspondia ao esperado.

Por exemplo:

- add:

```C/C++
exmem.executar(bitset<32>(20), bitset<32>(30), bitset<16>(2),
bitset<5>("00001"), false, false, false, n, false, false, NULL);
test.assertEqual(exmem.result, bitset<32>(50));
// output: Test Completed!
```

- mult:

```C/C++
exmem.executar(bitset<32>(7), bitset<32>(5), bitset<16>(2),
bitset<5>("11001"), false, false, false, n, false, false, NULL);
test.assertEqual(exmem.result, bitset<32>(35));
// output: Test Completed!
```

2. Execução de Programas

Ao decorrer do desenvolvimento, foi realizada uma bateria de testes para comprovar a validação do simulador, utilizando testes massivos e programas reais, como: Cálculo de Fatorial, Média e etc.

Internamente ao repositório é possível identificar a pasta “output” contendo alguns dos programas utilizados nos testes em conjunto com a biblioteca de “Teste” desenvolvida pelo grupo e testados no arquivo externo “testes.cpp”.

8. Conclusão

Neste trabalho podemos ter um entendimento melhor sobre os conceitos empregados na operação de um processador monociclo, através do simulador RISC desenvolvido, deixando mais clara a leitura e entendimento de datapath's, a divisão dos estágios em um monociclo, bem como os demais conceitos associados à disciplina, contribuindo de forma fundamental para o aprofundamento do aprendizado.