UNIVERSIDADE DE SÃO PAULO

BACHARELADO EM CIÊNCIA DE COMPUTAÇÃO

João Ricardo de Almeida Lustosa - 15463697
Jhonatan Barboza da Silva - 15645049
Beatriz Alves dos Santos - 15588630

SIMULADOR PIPELINE

Trabalho Prático II - Organização e Arquitetura de Computadores - SSC0902

São Carlos
2025

1. Introdução:

A história da computação anda de mãos dadas com uma busca incessante por maior poder de processamento. Juntamente da evolução da potência dos processadores, as inovações na arquitetura trouxeram grandes avanços na performance dos computadores. Nesse contexto, a técnica de pipelining se destaca como um dos pilares dos processadores modernos, permitindo um grande aumento na vazão de instruções executadas.

Este projeto propõe o desenvolvimento de um simulador de *pipeline*, ferramenta essencial para a análise de processadores modernos. A técnica de *pipelining* consiste em fracionar a execução de uma instrução em múltiplos estágios independentes. Assim que uma instrução conclui um estágio, ela avança para o próximo, liberando o anterior para que a instrução seguinte comece a ser processada. Isso permite que diversas instruções sejam executadas de forma sobreposta e simultânea, cada uma em um estágio diferente, resultando em um aumento significativo na performance do processador.

Diante disso, esse projeto objetiva fornecer uma ferramenta didática visual e interativa, que torne mais simples a compreensão do funcionamento do pipeline, solidificando o conhecimento teórico.

2. Desenvolvimento do Trabalho

O simulador é uma ferramenta educacional completa que demonstra visualmente como funciona um pipeline de processador, incluindo os desafios e soluções para hazards.

2.1. Idealização do projeto

A ideia do simulador começou nas aulas de Organização e Arquitetura de Computadores, em que eram feitas simulações de

pipeline na lousa, e para simples instruções tomavam um espaço muito grande de informações. A partir disso, tendo em vista o público alvo mais jovem, veio a ideia de uma simulação mais interativa, que fosse mais atrativa e simples para o público, mas mantendo a didática das simulações.

2.2. Ferramentas utilizadas

Para a implementação do trabalho foram utilizadas as linguagens de programação: html, css e javascript, que são a base do desenvolvimento web moderno e permite que o simulador possa ser executado em qualquer computador sem a instalação de softwares adicionais.

2.3. Estrutura do Pipeline:

O simulador trata da implementação de um pipeline RISC-V de 5 estágios, sendo eles: IF, ID, EX, MEM, WB.

- **IF (Instruction Fetch / Busca da Instrução)**
  - Responsável por buscar uma nova instrução na memória de instruções utilizando o contador PC (Program Counter).
  - **Implementação:** No código, o método instructionFetch() é responsável por esta etapa. Ele acessa o array this.program, que contém as instruções carregadas, na posição indicada por this.pc. A instrução recuperada é então armazenada no registrador do estágio IF, e o pc é incrementado para apontar para a próxima instrução. Se o pc ultrapassar o tamanho do programa, o estágio IF é marcado como inativo.
- **ID (Instruction Decode / Decodificação da Instrução)**
  - Nesse estágio a instrução é decodificada, identificando os operandos e o tipo de instrução.

    - **Implementação:** O método instructionDecode() gerencia a passagem da instrução para o próximo estágio. A lógica de decodificação está intrinsicamente ligada à detecção de conflitos, realizada no método detectHazards(). Neste ponto, a instrução em ID é analisada: suas fontes de registradores são comparadas com os registradores de destino das instruções nos estágios subsequentes (EX, MEM, WB). Caso um conflito seja detectado e o adiantamento (*forwarding*) não seja possível ou esteja desabilitado, o método define a flag this.stallPipeline, que congela os estágios IF e ID no ciclo seguinte.
- **EX (Execute / Execução)**
    - Parte em que as operações lógicas e aritméticas são calculadas, assim como cálculo de endereços.
    - **Implementação:** O método execute() utiliza uma estrutura switch para direcionar a lógica com base no mnemônico da instrução (ex: 'add', 'sub', 'lw'). Na implementação atual, este método atua como um *placeholder*, preparando a estrutura para a lógica completa da ULA (Unidade Lógica e Aritmética). Para desvios como beq, é neste estágio que o conflito de controle é identificado e sinalizado.
- **MEM (Memory Access / Acesso à Memória)**
    - Tem por função ler ou escrever dados na memória. Em casos de instruções que não acessam a memória, esse estágio é inativo.
    - **Implementação:** O método memoryAccess() verifica se a instrução é do tipo lw (load word) ou sw (store word). Para lw, ele simula uma leitura da estrutura this.memory, e para sw, simula uma escrita. Para todas as outras instruções que não interagem com a memória, este estágio atua de forma passiva, apenas propagando os dados para o estágio seguinte.

- **WB (Write Back / Escrita de Volta)**
    - Atualiza o banco de registradores com os resultados da instrução.
    - **Implementação:** O método writeBack() conclui o ciclo de vida da instrução. Ele verifica se a instrução produz um resultado a ser escrito (como add, addi ou lw) e, em caso afirmativo, atualiza o valor correspondente no array this.registers no índice do registrador de destino (rd).

2.4. Detecção e Solução dos Hazards:

A detecção e o tratamento de conflitos são centralizados no método detectHazards(), que é invocado a cada ciclo de clock.

- **Conflitos de Dados (RAW - Read After Write):**

  A implementação atual foca no conflito entre os estágios ID e EX. O código verifica se uma instrução em EX escreve em um registrador que é lido pela instrução em ID.

    - **Solução:** A ação depende da flag forwardingEnabled. Se ativada, o conflito é registrado e a simulação continua, assumindo que o dado será "adiantado" (a implementação completa do adiantamento atualizaria os operandos da ULA). Se desativada, a flag stallPipeline é acionada, inserindo uma bolha no pipeline ao paralisar os estágios IF e ID por um ciclo.

- **Conflitos de Controle:** A implementação identifica a instrução beq no estágio EX e, caso a detecção de *hazards* esteja ativa, insere um *stall* para simular o tempo de resolução do desvio. A lógica de *flush* (limpeza do pipeline em caso de desvio tomado) é o próximo passo natural desta implementação.

2.5. Funcionamento

A transição de estados do pipeline é orquestrada pela interação entre a lógica do simulador e a interface do usuário.

O Ciclo de Clock: A função step() na classe PipelineSimulator representa um único ciclo de clock. Cada chamada a esta função executa a lógica de cada um dos cinco estágios e avança as instruções entre eles.

Propagação das Instruções: Dentro do step(), as instruções progridem de um estágio para o outro. Ao final da execução da lógica de um estágio (ex: execute()), o resultado e a instrução são passados para o registrador do próximo estágio (ex: this.mem = { instruction: this.ex.instruction, ... }).

Atualização Visual: Após cada chamada de step(), a função updateUI() do código da interface é executada. Ela lê o estado atual do simulador (o conteúdo de cada estágio, os valores dos registradores e da memória) e redesenha a visualização do pipeline em HTML. A função updatePipelineVisualization cria um diagrama que exibe o histórico de cada ciclo, usando classes CSS para indicar visualmente se uma instrução está ativa, parada (stalled) ou envolvida em um conflito. Para a execução contínua, um setInterval chama step() e updateUI() repetidamente, criando a animação do pipeline em funcionamento.

2.6. Interface Gráfica

A tela é organizada em painéis funcionais: uma área para edição do código Assembly, um painel de controles (Executar, Passo a Passo, Reiniciar), e monitores para o estado dos registradores e da memória.

O componente central é a **visualização do pipeline**, que exibe dinamicamente o fluxo de instruções através dos cinco estágios (IF, ID, EX, MEM, WB) a cada ciclo de clock. É usado também um sistema de cores intuitivo para fornecer feedback imediato sobre o estado de cada instrução: **verde** para ativa, **amarelo** para *stall* (parada) e **vermelho** para *hazard* (conflito).Ademais, há painéis adicionais que apresentam estatísticas de desempenho em tempo real, como o CPI (Ciclos por Instrução) e o número de *stalls*, permitindo que o usuário observe de forma clara o impacto dos conflitos e das otimizações no funcionamento do processador.

3. Conclusões

Motivados pelas simulações em sala de aula, esse trabalho surge na tentativa de tornar mais simples e divertido o aprendizado sobre o *pipelining,* a partir de uma ferramenta didática, visual e interativa, que realiza as simulações.

Com isso, é visto, portanto, que o projeto satisfaz os objetivos traçados na sua idealização, combinando uma lógica de pipeline com uma interface gráfica interativa, o simulador atende tudo aquilo que pensamos ao ver as simulações em sala de aula, como: A demonstração do paralelismo, o aumento do throughput (vazão das instruções), a visualização clara dos hazards e do fowarding.

Por fim, para além de um simulador, esse trabalho é um facilitador do aprendizado, que adiciona um teor descontraído a um conteúdo que, muitas vezes, é considerado denso e complicado, unindo didática e diversão em uma ferramenta só.

4. Bibliografia

Explicação sobre o pipeline: link

Slides utilizados em aula: Arquitetura Risc-V Pipeline