Yuri Alessandro Martins
Yuri Reinaldo da Silva

RELATÓRIO SOBRE O PROCESSO DE CRIAÇÃO DE UM SIMULADOR DE PIPELINE EM C++

Natal - Rio Grande do Norte
15 de junho de 2016

Yuri Alessandro Martins
Yuri Reinaldo da Silva

RELATÓRIO SOBRE O PROCESSO DE CRIAÇÃO DE UM SIMULADOR DE PIPELINE EM C++

Relatório apresentado à disciplina de Introdução a Organização e Arquitetura de Computadores, ministrada pela professora Dra. Monica Magalhães Pereira, do Departamento de Informática e Matemática Aplicada, da Universidade Federal do Rio Grande do Norte, para fins avaliativos.

Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital
Bacharelado em Tecnologia da Informação

Natal - Rio Grande do Norte
15 de junho de 2016

Sumário

1 Introdução

Com este relatório, pretendemos analisar o processo de produção de um simulador de pipeline, mostrando o processo de pensamento e soluções de implementações encontradas para os problemas que encontramos.

O desafio de implementar um simulador de pipeline veio como proposta de uma atividade para fins avaliativos da disciplina de Introdução à Organização e Arquitetura de Computadores, ministrada pela Profª Dra Monica Magalhães Pereira da Universidade Federal do Rio Grande do Norte.

De acordo com a proposta, deveríamos implementar um simulador de um pipeline adaptado para instruções em Assembly do MIPS de 5 estágios, que utiliza de paradas como solução para problemas de conflitos. Para tal nós optamos por fazer um programa em C++ por ser a linguagem que nos é mais familiar.

2 O Pipeline

Nos projetos de arquitetura de computadores o pipeline surgiu como uma solução para o problema de que, em sistemas de monociclo e multiciclo, temos dispositivos constantemente ociosos, devido à natureza de apenas uma instrução ser processada em um dado momento.

A resposta para tal veio com a proposta de serem executadas várias instruções ao mesmo tempo. Isso é possível separando a instrução em partes que utilizam de componentes diferentes e fazendo com que cada instrução siga uma sequência fixa de uso desses componentes, sendo assim, no momento em que um destes é liberado a próxima instrução poderá usá-lo caso necessite. Uma abstração para maior compreensão do comportamento do pipeline pode ser vista na Figura 1.

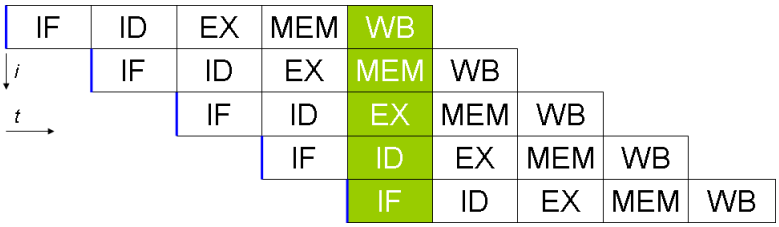

Figura 1 – Representação visual do comportamento de um pipeline de 5 estágios.

Fonte: https://upload.wikimedia.org/wikipedia/commons/2/21/Fivestagespipeline.png. Acessado às 16:21 de 15/06/2016

Com esse melhor uso dos componentes, várias formas mais avançadas de pipeline passaram a ser implementadas como, por exemplo, o Superescalar. Nesse trabalho estaremos trabalhando uma forma mais simples, porém com complexidade suficiente para compreendermos a lógica por trás da técnica.

3 Implementação

Para darmos início a análise tivemos que criar uma função que lê um arquivo externo - contendo as instruções a serem executadas pelo nosso simulador - corretamente, um método que identifica essas instruções, seu comportamento, seus operandos e seu registrador de destino. Em seguida partimos para a análise de conflitos, a parte principal do programa.

3.1 Leitura de arquivo

A função `readFile` (definida em `include/functions.h` e `include/functions.inl`), de assinatura:

```
1 bool readFile( std::string & _filename,  std::vector<Instruction> & inst_vec)
```

Em que `_filename` é o nome do arquivo recebido como argumento do `main` e `inst_vec` é um vetor de objetos do tipo `Instruction` que será preenchido. A função analisa se o arquivo passado como argumento na chamada do programa é válido e então inicia a leitura linha por linha, passando a string da linha para o construtor de um objeto `Instruction` e inserindo-o no vetor. Caso haja uma instrução com formato inválido, uma mensagem é exibida na tela e a função é terminada. O valor de retorno é `true` se a leitura foi bem sucedida e `false` caso contrário.

3.2 Classe Instruction

Utilizamos de classes de objetos do C++ para criarmos uma classe `Instruction` (declarada no arquivo `include/instruction.h` e implementada em `include/instruction.inl`), da qual teremos uma série de métodos que darão suporte à análise das instruções, sendo estes:

`getName()`

Este método retorna uma string com o nome da instrução. Toda instrução obrigatoriamente deve possuir um nome logo no início de sua chamada, este é utilizado para analisar o que a instrução deve fazer e qual o seu formato.

`getDest()`

Retorna a string com o registrador designado como o destino do resultado da instrução. É utilizado com frequência no processo de análise de conflitos que será explicado logo após nesse relatório.

`getOp1() e getOp2()`

São dois métodos similares. Retornam o primeiro e o segundo operador da instrução, respectivamente. Também são de vital importância para o funcionamento da análise de conflitos.

`void print()`

Exibe no terminal a instrução com o seguinte o formato:

```
[nome destino operando1 operando2]
```

Algumas instruções não têm alguns operandos (como `addi`), outras não tem destino (como `beq`) e algumas não tem nenhum operando (como `mfhi`). Tanto as funções `getDest()`, quanto `getOp1()` e `getOp2()`, têm seus valores pré-definidos como uma string contendo "`devoid`" para o caso de não haver algum desses parâmetros.

3.3 Análise de conflitos

A função de análise (`analysis`) está definida no arquivos `include/functions.h` e implementada em `include/functions.inl`. É ela aonde se mantém a maior parte do funcionamento do programa. A sua assinatura é na seguinte forma:

```
void analysis( std::vector<Instruction> inst_vec );
```

O vetor `inst_vec` é o mesmo vetor de objetos `Instruction` que foi preenchido em `readFile`. É a partir dele que iremos "processar" cada instrução checando se há algum conflito com as instruções que ainda estão sendo executadas. Para tal criamos duas variáveis para auxiliar no processo:

`unsigned int t`

É uma variável que contém o tempo total de execução, é incrementada a cada passo do "processamento" das instruções.

`Queue<std::string> operators`

Uma fila implementada por nós (para outras disciplinas) que recebe as strings dos registradores de destino das instruções que estão sendo executadas. Foi uma estrutura de dado escolhida por conta do seu comportamento FIFO (First In First Out) que simula a o término e a entrada da execução das intruções na fila de "processamento". Optamos por utilizá-la pois facilitaria a busca de um elemento interno. Na Figura 2 podemos ver o comportamento da fila e o porquê de seu uso.

Para simular o processamento criamos um laço que será percorrido enquanto não passarmos por todas as instruções do vetor. Durante cada iteração três casos são possíveis:

- A instrução atual é a primeira a ser processada. Nesse caso nós introduzimos o destino dela na fila, incrementamos o tempo e incrementamos a posição da instrução ser analisada.
- Os operandos da instrução atual ainda estão sendo processados (estão na fila de destinos). Para essa situação o tempo é incrementado em um e é inserido uma string para ocupar o espaço que a instrução vazia seria inserida no pipeline normal.
- Se não há conflito, o destino da instrução é inserido na fila, o tempo é incrementado e a posição da instrução passa a ser a próxima.

A partir do momento em que chegamos no tempo 5, em cada iteração estaremos retirando o elemento da fila mais antigo, representando o fim do processamento de uma instrução.

Depois que terminamos de percorrer o vetor de instruções, removemos os valores restantes e incrementamos o tempo para cada restante na fila de destinos, como se estivéssemos encerrando todas as funções que faltam.

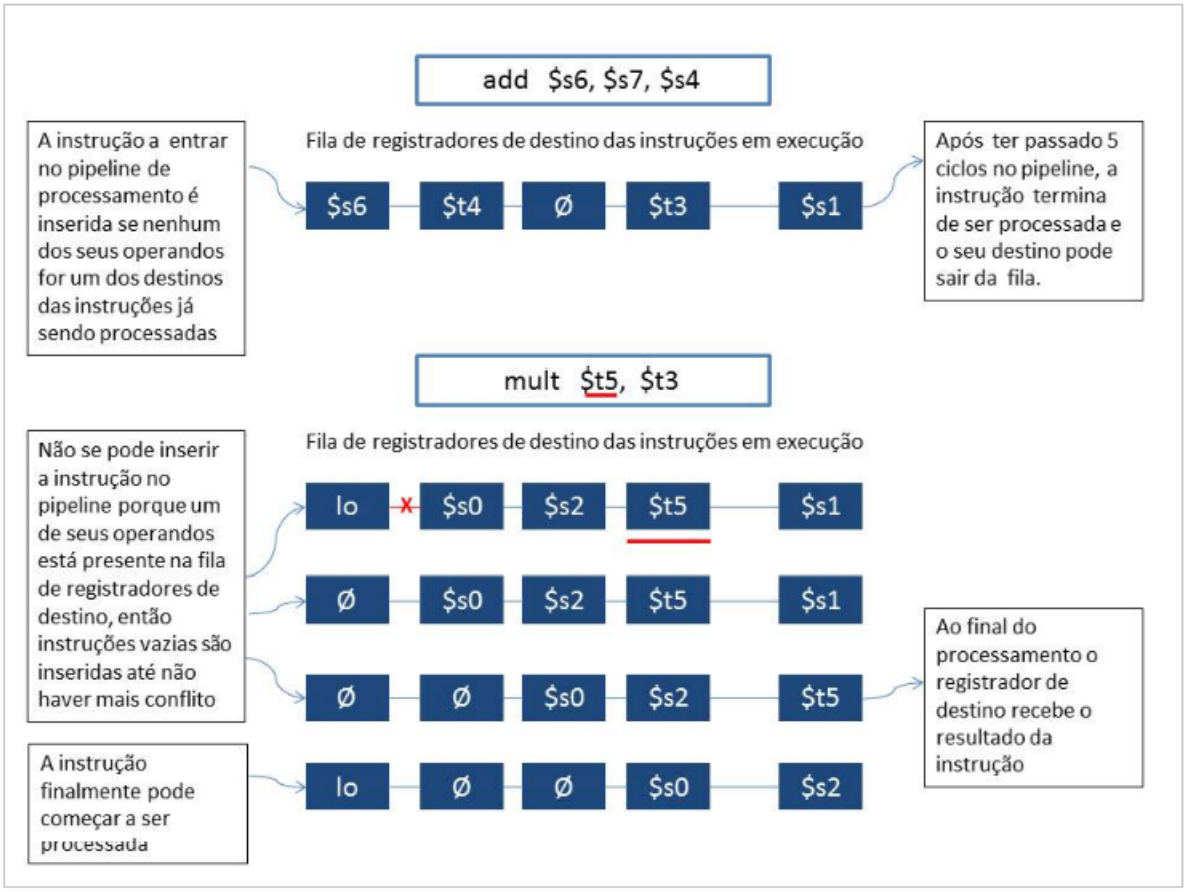

Figura 2 – Demonstração do comportamento da fila (`Queue`) operators e como ela funciona em conjunto com a análise de conflitos. No primeiro exemplo (da instrução `add`) temos o caso em que não há nenhum conflito. No segundo exemplo (`mult`) temos que o operando guardado em `$t5` está aguardando o resultado de uma instrução que ainda está em execução.

Por fim exibimos na tela o tempo que foi necessário para realizar a análise de conflito, logo após termos exibido o estado do pipeline a cada passo.

4 Compilação e Execução

Para compilar esse projeto, na plataforma **Linux**, basta abrir o diretório padrão no seu terminal e dar o comando:

```
$ make
```

Isso irá gerar um arquivo executável no diretório `bin`. Para executar o projeto, você deverá então dar o seguinte comando no terminal:

```
$ ./bin/drive_pipeline <nome_arquivo_entrada>
```

onde `nome_arquivo_entrada` deve ser um arquivo contendo as instruções em Assembly do MIPS. Um exemplo:

```
$ ./bin/drive_pipeline instructions.txt
```

Você talvez queira escrever os resultados em um arquivo externo, que inclusive irá lhe garantir uma melhor vizualização. Para isso, ainda no terminal, após compilar, basta dar o seguinte comando:

```
$ ./bin/drive_pipeline <nome_arquivo_entrada> > <nome_arquivo>
```

Onde `<nome_arquivo>` será o seu arquivo de saída contendo a execução da aplicação. Exemplo:

```
$ ./bin/drive_pipeline instructions.txt > saida.txt
```

5 Conclusão

Em resumo, o pipeline foi simulado usando a ideia de uma fila de instruções e de registradores, o que, de fato, acontece na aplicação real desse tipo de processador - mas na memória. Para isso, foi utilizado um laço que corresponde ao número de estágios realizados. Para cada cinco estágios (que corresponde ao número de estágios do MIPS, nosso estudo de caso) o registrador destino da instrução executada é retirado da nossa fila e isso significa que ele não terá mais qualquer conflito com próximas instruções que queiram usar o valor atribuído a esse registrador.

Com a implementação desse simulador, foi possível notar que o recurso de usar a "pseudo-ociosidade[1]" para melhorar o tempo de execução o torna bem mais eficiente do que outras implementações, como a de multiciclo, por exemplo, mesmo quando existe conflito de dados.

Também é importante notar que essa aplicação não utiliza quaisquer métodos de evitar os conflitos, como **Rederecionamento** ou **Renomeação** de instruções, o que poderia tornar o pipeline ainda mais eficiente em questão de tempo.

[1] Isso porquê o Pipeline na verdade não utiliza paralelismo de componenetes, mas sim o chamado paralelismo de instruções.