# Pipelining

Lucas

19 de outubro de 2015

pg 391 Pipeline explora paralelismo em potencial entre instruções[a]. Técnica (1) aumentar a profundidade do pipeline para transpor mais instruções para pegar a velocidade máxima é preciso rebalancear os outros estágios para acomodar as instruções preenchendo todo o datapath (sempre que possível) isto é, ter 5 instruções ocupando o datapath (nem sempre isso é possível), e uma técnica (2) é replicar componentes digitais do computador (datapath) para processar múltiplas instruções[b] ao mesmo tempo de relógio, por exemplo: um microprocessador de 4 GHz 4-way emissor-múltiplo pode executar uma taxa pico de 16 bi instruções por segundo e ter o melhor CPI de 0.25 ciclos por instrução, ou o contrário um IPC igual 4 instruções por ciclo, porém emissão múltipla não é sempre viável para o microprocessador, e existe a problemática dos hazards de dados e controle que degradam o CPI.

| Instruções | Ciclo 1 | Ciclo 2 | Ciclo 3 | Ciclo 4 | Ciclo 5 | Ciclo 6 | Ciclo ... |
|---|---|---|---|---|---|---|---|
| I0 | IF | ID | EX | MEM | WB | | |
| I1 | IF | ID | EX | MEM | WB | | |
| I2 | | IF | ID | EX | MEM | WB | |
| I3 | | IF | ID | EX | MEM | WB | |
| I4 | | | ... | ... | ... | ... | ... |
| I5 | | | ... | ... | ... | ... | ... |

Tabela 1: Exemplo de emissão múltipla

[a] ***ILP***.: significa paralelismo entre instruções.

[b] **Emissão múltipla**.: é um esquema onde múltiplas instruções são lançadas juntas em um ciclo de clock.

pg 392 Existem dois esquemas para implementar emissão múltipla: (1) estática[c], e (2) dinâmica[d]. Preocupações principais:

(a) Empacotar instruções em lacunas de emissão: significa que o processador tem que decidir quantas instruções ele vai poder empacotar para serem emitidas multiplamente, e também, quais instruções ele poderá combinar em cada ciclo para emitir elas em um pacote só.

| | | | | |
|---|---|---|---|---|
| Possibilidade 1 | instrução 1 | instrução 2 | instrução 3 | instrução 4 |
| Possibilidade 2 | instrução 1 | instrução 2 | instrução 3 | |
| Possibilidade 3 | instrução 1 | instrução 2 | | |
| Possibilidade 4 | instrução 1 | | | |

Tabela 2: Exemplo de um emissor 4-way, tentando decidir se ele pode emitir 4 instruções junta, senão 3, senão 2, senão 1 (pelo menos uma instrução ele emite) e no melhor caso emite 4 instruções de uma só vez.

(b) Lidando com hazards de controle e dados: em emissão estática as consequências geradas por esses tipos de hazards (dependências) é cuidada estaticamente pelo compilador. Em contraste emissão dinâmica também cuida das consequências geradas por essas classes de hazards (dados e controle) em tempo de compilação, mas Emi. din. tenta aliviar o impacto de hazards na emissão múltipla; em tempo de execução, adicionando técnicas de hardware (+custo).

pg 393 Uma abordagem clássica para aumentar *ILP* é a especulação[e] que é uma abordagem que permite o computador "adivinhar" sobre as propriedades de uma instrução, para habilitar a possibilidade de outras instruções serem executadas, ou seja, o processador começa buscar instruções "especulando"que o resultado delas vai ser $x$ e continua puxando instruções, se o resultado não for $x$, mas $y$, então acontece um descarregamento das instruções, na verdade ele simplesmente não grava o resultado no banco de registradores, ou na memória assim as instruções que foram anuladas por causa que deu $y$.

Exemplo 1 Poder-se-ai especular que um branch (à diante) vai dar certo, então o processador pode ir executando as instruções no bloco básico pertencente ao escopo do branch e quando chegar no branch as instruções já foram processadas.

Exemplo 2 Poder-se-ai especular que um store precedente a um load, e também, ambos não referenciarão o mesmo local da memória, o que permitiria ao processador executar o load anteriormente ao store.

A problemática da especulação é que ela pode errar, e isso joga fora ciclos de processamento. Precisa de mecanismo de hardware para, em tempo de execução, especular sobre os resultados das instruções futuras e também um mecanismo para poder voltar atrás e não executar as instruções, como eu disse é um componente digital que verifica se o acerto deu correto ao não, isto é, se o resultado atual é igual ao resultado "chutado" se for igual, prossegue, se for diferente o datapath tem que acionar o enfileirador de instruções para eliminar instruções[f]. Especulação pode ser feita tanto

[c] **Emissão múltipla estática**.: É uma abordagem onde muitas decisões são feitas pelo compilador, antes da execução das instruções

[d] **Emissão múltipla dinâmica**.: É uma abordagem onde muitas decisões são feitas, sobre as instruções, em tempo de execução.

[e] **Speculation**.: é uma abordagem onde o compilador ou processador tenta adivinhar o que será de uma instrução para remover ela como uma dependência, quando executando outras instruções.

[f] Essas instruções que o especulador buscou "acreditando" que elas iriam ser executadas se o "chute" tivesse dado certo

pelo compilador, quanto pelo hardware. O compilador pode usar especulação para reordenar instruções, movendo uma instrução através do branch ou um load através do store.

Especulação trás outro problema: Especular certos tipos de instruções podem introduzir exceções! Imagine uma instrução LOAD é movida de maneira especulativa, mas o ENDEREÇO que ela usou não é LEGAL se a especulação estiver incorreta! Uma exceção que não deveria acontecer, mas acontecerá.

pg 394 EMISSÃO ESTÁTICA, basicamente essa técnica "mistura" instruções para serem emitidas em um mesmo ciclo, isso aumenta performance, mas requer que o datapath suporte esses caminhos múltiplos para acomodar as instruções, o mix de instruções (pacote de instrução gerado pelo emissor múltiplo estático) precisa de componente para administrar hazards de controle e dados, esse mix de instruções em um mesmo ciclo é na verdade uma grande instrução com várias operações, a saber, independentes [g]. Se no mix de instruções, acontecer de apenas uma instrução poder ser emitida em um ciclo, é recomendado que as outras lacunas do pacote emitido sejam preenchidas com `nop`.

Em emissão estática, varia como é feita o tratamento de hazards de dados e controle. Em alguns modelos, o compilador leva total responsabilidade em remover todos os hazards, escalonando o código e inserindo no-ops para que o código execute sem a necessidade de detecção de hazards, ou suspensões hardware-geradas. Em outras palavras o hardware detecta hazards e gera as suspensões entre duas emissões múltiplas, enquanto requerer que o compilador se livre de todas as dependências entre pares de instruções. Mesmo assim, um hazard geralmente força um pacote inteiro contendo a instrução dependente a suspender seu processamento no datapath ("entrar em *stall*"). Onde o software precisa tratar todos os hazards ou apenas tentar reduzir uma fração de hazards entre pacotes de emissão, a aparência de ter uma "única instrução larga" com múltiplas operações é reforçada. Nós vamos assumir a segunda abordagem para esse exemplo.

[g] **Very Long Instruction Word**.: *VLIW*, que é muito parecida, porém processadores VLIW emitem uma grande instrução com várias operações que foram decididas como sendo independentes e tipicamente essa instrução vem com vários códigos operacionais.

pg 395 Para emitir uma operação com ALU e uma transferência de dado em paralelo ("processador emissor estático duas vias"), o que precisamos fazer é adicionar portas extras no banco de registradores (além de que precisamos do detector de hazard e componente para a logica de suspensões). Isso aumenta performance 2x mais. Porém requer que duas vezes mais instruções sejam transposta em execução. Por exemplo, loads têm 'latência de uso' igual a 1 ciclo de relógio, o que antecipa uma instrução em usar o resultado sem ser suspendida. Em um emissor duplo, o resultado de uma instrução load não pode ser usado no próximo ciclo. Isso significa que as próximas duas instruções não podem usar o resultado do load sem suspender.

pg 396 No código escalonado abaixo, foi feito reordenamento (pelo compilador) para escapar do máximo possível de suspensões no pipeline ("*stalls*"), possível. Estamos assumindo que branchs são preditos, assim hazards de controle são tratados pelo hardware.

| **LABEL** | **ALU/BRANCH** | **LOAD/STORE** | **CICLO** |
|---|---|---|---|
| LOOP: | addi $s1, $s1, -16 | lw $t0, 0($s1) | 1 |
| | | lw $t1, 12($s1) | 2 |
| | add $t0, $t0, $s2 | lw $t2, 8($s1) | 3 |
| | add $t1, $t1, $s2 | lw $t3, 4($s1) | 4 |
| | add $t2, $t2, $s2 | sw $t0, 16($s1) | 5 |
| | add $t3, $t3, $s2 | sw $t1, 12($s1) | 6 |
| | | sw $t2, 8($s1) | 7 |
| | bne $s1, $zero, LOOP | sw $t3, 4($s1) | 8 |

Tabela 3: Exemplo de emissão múltipla, considerando latência de uso do LOAD, esse código foi feito desenrolamento de laço 4 vezes. Durante o desenrolar, o compilador introduz registradores adicionais, criando espaço para que não haja dependência entre instruções, permitindo acomodar elas nos ciclos, e criando uma anti-dependência entre elas. **Anti-dependência**, ou dependência de nome, é o ordenamento forçado pelo reuso de nomes, tipicamente um registrador, mais do que a dependência de dados mesmo. Tratar desse problema é reordenando, renomeando registradores, porém as vezes o compilador não consegue ir muito longe para cuidar disso.

Ou seja, a latência de uso de uma LOAD gera um 1 ciclo extra, e é apenas quando existe a dependência de dados gerado no LOAD e consumido em uma próxima instrução, nesse caso o hardware deve acomodar as instruções para que a instrução que consumir o dado gerado no LOAD fique a 1 ciclo de distância do LOAD, para se desviar do hazard de dados (em inglês o termo é "use latency").

| **LABEL** | **ALU/BRANCH** | **LOAD/STORE** | **CICLO** |
|---|---|---|---|
| LOOP: | | lw $t0, 0($s1) | 1 |
| | addi $s1, $s1, -4 | | 2 |
| | addu $t0, $t0, $s2 | | 3 |
| | bne $s1, $zero, LOOP | sw $t0, 4($s1) | 4 |

Tabela 4: O mesmo código (sem desenrolamento quádruplo) sendo processado por um emissor múltiplo duas vias. Nesse ensaio temos que o "use latency" do primeiro **load** está em ação, é necessário um ciclo de distância para evitar o hazard de dados.

pg 397 Desenrolamento de laços é uma técnica importante, onde múltiplas cópias do laço repetitivo são feitas. Essa é claramente uma técnica para aumentar a taxa de instruções por ciclo, em um emissor múltiplo essa técnica é bem aproveitada. Busca aumentar a vazão de instruções pelo datapath. Essa técnica exige implementação extra—renomeador de registradores[h].

pg 398 Processadores com emissores dinâmicos também são conhecidos como super escalares (*superscalar*). No mais simples, as instruções são emitidas em ordem[i] E o processador decide quando zero, uma, ou mais instruções poderão ser emitidas em um pacote único em um mesmo ciclo de relógio.

Existe uma grande diferença entre um simples super escalar e um VLIW: o código, quando escalonado (*scheduled*) ou não, é garantido pelo hardware que irá executar corretamente. Além do mais, código compilado vai sempre rodar corretamente independente da taxa de emissão, ou estrutura do pipeline do processador. Em alguns VLIW, isso não tem sido o caso, por vezes quando código compilado para uma máquina emissora estática era rodado em outra máquina tinha perda de performance e tinha de ser recompilado.

pg 399 Uma técnica boa é escalonamento em pipeline dinâmico, onde escolhe quais instruções irão executar em um dado ciclo de relógio, enquanto tentando desviar de hazards e suspensões (*stalls*).

```
lw   $t0, 20($s2)
addu $t1, $t0, $t2
sub  $s4, $s4, $t3
slti $t5, $s4, 20
```

Mesmo que o SUB já esteja apto a ser processado, ele terá que esperar pelo seus companheiros LW e ADDU, o que ira levar muitos ciclos de relógio se a memória for lenta [j]. Escalonamento em pipeline dinâmico permite tais hazards serem "anulados" completamente ou parcialmente.

pg 400

pg 401

pg 402

pg 403

pg 404

pg 405

pg 406

**4.29.4**

[h] busca renomear quantos necessário, visto que no desenrolar irão aparecer mais instruções executando a mesma tarefa de maneira adiantada, renomear registradores remove anti-dependências.

[i] No emissor estático o compilador pode porventura reordenar as instruções, para eliminar hazards.

[j] Faltas na caches torna o acesso a memória custoso em termos de ciclos de relógio (tempo)

**4.29.5**

**4.29.6**