Pipelining

Lucas

20 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.

Escalonamento com pipeline dinâmico: é um modelo que escolhe quais serão as próximas instruções a serem executadas, possivelmente reordenando elas para evitar *stalls*. Em tais processadores que usam esse modelo, o pipeline é dividido em 3 unidades majoritárias: um buscador de pacotes de instrução (*mix*, *issue packet*), múltiplas unidades funcionais[k], e uma unidade de *commit*, que contém um buffer (*enfileirador reordenador*), é usado para dar suporte aos operandos (registradores) da mesma maneira que a lógica de forwarding em escalonamento com pipeline estático. Quando um resultado é cometido para o banco de registradores, o resultado pode ser buscado diretamente de lá, assim como em um pipeline normal.

pg 400 A implementação que cuida dos reservatórios e das unidades funcionais deve dentre outras coisas, enfileirar instruções que chegam para as unidades funcionais, as instruções terão prioridade de execução intrínseca muitas vezes, mas nada impede do modelo de escalonamento dinâmico reordenar instruções dentro do processo, trabalhando em cima das unidades funcionais, e quando tiver que executar algumas instruções com prioridade ele executa as prioridades, esperando que algumas unidades funcionais liberem seus resultados para que outras unidades funcionais que dependem

[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)

[k] Cada unidade funcional tem buffers, chamado de estações reservatório, o que segura os "operandos" e "operadores", em 2008 eram 12 unidades para se ter noção

desses resultados, possam processar as suas instruções, no final a unidade de entrega, vai reordenar as instruções para que siga a lógica que o programador fez, o produto final é o mesmo. Mas o modelo dinâmico possivelmente trocou a ordem de instruções para obter maiores taxas de emissões múltiplas (*out-of-order execution*).

Escalonamento dinâmico também permite uma opção chamada de *in-order*, ondem a ordem que o compilador gerou, vai ser a mesma ordem que entra e saí do pipeline com escalonador dinâmico, hoje em dia é o que mais se usa. Escalonadores dinâmicos em-ordem usam também especulação para BRANCHES e endereços de LOAD/STORES, visto que a execução é em ordem, após a unidade de entrega se pode fazer certas especulações sobre o resultados dessas classes de instruções.

pg 401 Mas por que usar escalonamento dinâmico? **(1)** Stalls são um vilão para os processadores, por causa de faltas na caches, stalls podem ser imprevistos, e o pipeline dinâmico pode utilizar os ciclos de stall para substituí-los inserindo outras instruções para serem processadas, não degradando *instruction level parallelism*, **(2)** incorporar especulação dinâmica funciona melhor com um hardware que escalona dinamicamente as instruções, pois o compilador não consegue especular BRANCHES, LOADS e STORES em tempo de compilação, **(3)** escalonamento dinâmico comporta melhor compilações de máquinas diferentes, e código legado também recebe um benefício rodando em máquinas que usam escalonamento dinâmico[l]

pg 402 Power Efficiency and Advanced Pipelining The downside to the increasing exploitation of instruction-level parallelism via dynamic multiple issue and speculation is power efficiency.

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/	Speculation Cores/Chip	Power
Intel 486	1989	25 MHz	5	1	No	1	5 W
Intel Pentium	1993	66 MHz	5	2	No	1	10 W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29 W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75 W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103 W
Intel Core	2006	2930 MHz	14	4	Yes	2	75 W
UltraSPARC IV+	2005	2100 MHz	14	4	No	1	90 W
Sun UltraSPARC T1 (Niagara)	2005	1200 MHz	6	1	No	8	70 W

Tabela 5: Tabela de processadores Intel e Microprocessadores Sun em termos de complexidade de pipeline, número de cores, e custo energético. Os estágios do pipeline Pentium 4 não incluem unidade de *commit*. Se incluirmos, o então pipeline do Pentium 4 será mais profundo.

Acessos à memória se beneficiam de caches que não bloqueiam. Execução fora-de-ordem precisa de modelos de cache que permitem instruções serem executadas durante um miss.

pg 404 Opteron X4 usa um esquema para resolver anti-dependências e especulação incorreta que usa reordenamento do buffer junto com renomeação de registradores. Renomeia registradores arquiteturais[m]

[l] Pipelining e emissão múltipla aumentam a vazão de instruções e ILP. Todavia, hazards de dados e controle colocam uma cota superior para a sustentabilidade da performance, por causa que processadores precisam esperar que a dependência seja resolvida, modelagens centradas em software exigem que o compilador seja eficaz em remover ao máximo as dependências de dados e controle para que o emissor múltiplo com pipeline possa ser eficaz em obter grandes rajadas de instruções. E modelagens centradas em hardware encontram saídas em extensões do pipeline e dispositivos de emissão. Especulação, realizada pelo compilador ou pelo hardware, pode aumentar o montante de ILP que pode ser explorado, no entanto atente que especulação incorreta reduz performance.

[m] O conjunto de instruções de registradores visíveis do processador; por exemplo, em MIPS, eles são os 32 registradores para inteiros e os 16 para ponto flutuantes.

4.29 Nesse execício, nós consideramos a execução do loop em escalonamento estático super-escalar. Para simplificar o exercício, assuma que qualquer combinação de tipos de instruções é possível no mesmo ciclo de relógio, e.g., em um 3-*issue* super-escalar, as 3 instruções pode ser 3 operações ALU, 3 branches, 3 load/store, ou combinações dessas mesmas. Notar que isso apenas remove hazards estruturais, mas permanece a possibilidade de hazards de controle e dados e esses dois últimos precisam ser tratados corretamente.

4.29.4b Desenrole o laço do código abaixo uma vez apenas e escalone ele para um emissor duplo estático super-escalar. Assumir que o laço sempre executa um número par de iterações. Você pode usar registradores R10 até R20 quando mudar o código para eliminar possíveis dependências.

antes do desenrolar.

a.

```
Loop: ADDI R1,R1,4
      LW R2,0(R1)
      LW R3,16(R1)
      ADD R2,R2,R1
      ADD R2,R2,R3
      BEQ R2,zero,Loop
```

Loop:	ADDI R1,R1,4	**nop**
	nop	LW R2,0(R1)
	nop	LW R3,16(R1)
	ADD R2,R2,R1	**nop**
	ADD R2,R2,R3	**nop**
	BEQ R2,zero,Loop	**nop**

pós desenrolar.

```
Loop: ADDI R1, R1, 4

      LW R2, -4(R1)
      LW R3, 12(R1)

      LW R10, 0(R1)
      LW R11, 16(R1)

      ADD R2, R2, R1
```

```
ADD R2, R2, R3
ADD R2, R10, R11

BEQ R2, zero, Loop
```

Loop:	ADDI R1,R1,4	**nop**
	nop	LW R2, -4(R1)
	nop	LW R3, 12(R1)
	ADD R2, R2, R1	LW R10, 0(R1)
	ADD R2, R2, R3	LW R11, 16(R1)
	ADD R2, R10, R11	**nop**
	BEQ R2, zero, Loop	**nop**

antes do desenrolar. **b.**

```
Loop: LW R1,0(R1)         F D E M W
                          X X X X X
      AND R1,R1,R2          X F D E M W
                            X X X X X X
      LW R2,0(R2)               F D E M W
                                X X X X X
      BEQ R1,zero,Loop            F D E M W
                                  X X X X X
```

Podemos mover LW R@,0(R2) para segunda linha.

Loop:	LW R1,0(R1)	**nop**
	AND R1,R1,R2	**nop**
	LW R2,0(R2)	**nop**
	BEQ R1,zero,Loop	**nop**

pós desenrolar.

```
Loop: LW R1,0(R1)        F D E M W
                         X X X X X
      LW R3,4(R1)          F D E M W
                           X X X X X
      AND R1,R1,R2           F D E M W
                             X X X X X
      AND R3,R3,R2             F D E M W
                               X X X X X
      AND R1,R1,R3               F D E M W
      LW R2,0(R2)                F D E M W
      LW R2,4(R2)                  F D E M W
      BEQ R1,zero,Loop             F D E M W
```

> "A static two-issue datapath. The additions needed for double issue are highlighted: another 32 bits from instruction memory, two more read ports and one more write port on the register file, and another ALU. Assume the bottom ALU handles address calculations for data transfers and the top ALU handles everything else."
>
> ——page 395, Patterson & Hennessy, Computer Org. and Design

No entanto, subir uma instrução AND R3,R3,R2 para combinar em um issue-packet causa dependência de dados com LW R3,4(R1) que exige *use latency* de 1 ciclo.

LABEL	EMISSÃO 1	EMISSÃO 2	Ciclo
Loop:	**nop**	LW R1,0(R1)	1
	nop	LW R3,4(R1)	2
	AND R1,R1,R2	**nop**	3
	AND R3,R3,R2	**nop**	4
	AND R1,R1,R3	LW R2,0(R2)	5
	nop	LW R2,4(R2)	6
	nop	BEQ R1,zero,Loop	7

Número de instruções $7 * 2 = 14$ instruções contando os nops junto e $14 * 4 = 56$ bytes ao total.

4.29.5

4.29.6