



Arquitetura de Alto desempenho

Paralelismo em nível de instrução (complementos)

António Rui Borges

Resumo - 1

- *Eficiência do pipeline* •

Dependências e perigos dos dados

- *Dependências de dados*
- *Dependências de nomes*
- *Perigos de dados*

– *Controlar dependência*

- *Técnicas básicas de compilador para execução ILF* •

Previsão avançada de ramificação

– *Correlacionando preditores de ramificação*

– *Preditores de torneios* •

Agendamento dinâmico

– *Placar*

– *Algoritmo de Tomasulo*

Resumo - 2

- Especulação baseada em hardware
- Exploração de ILP usando múltiplos problemas
- Processador superescalar estaticamente programado
 - BRAÇO Cortex-A8
- Processador superescalar agendado dinamicamente
 - Intel Core i7
- Leitura sugerida

Eficiência do pipeline - 1

A simplicidade de um conjunto de instruções é uma propriedade fundamental na construção de um pipeline para sua implementação. Conjuntos de instruções simples oferecem ainda outra vantagem, pois são fundamentais para tornar o agendamento de instruções individuais mais simples. Essas vantagens parecem ser tão significativas que quase todas as implementações recentes de conjuntos de instruções complexos em pipeline realmente traduzem primeiro as instruções em operações simples do topo RISC e só então processam para seu pipeline e escalonamento.

Como forma de expressar o grau de sucesso obtido ... a execução de um determinado programa em um processador em pipeline, a seguinte equação pode ser usada

$$\text{IPC}_{\text{programa}} = \text{CPIideal} + \text{paralisações estruturais} + \text{paralisações de dados} + \text{paralisações de controle} ,$$

onde o CPIideal, ciclos de clock por instrução na situação ideal, é uma medida do desempenho máximo atingível pela implementação em pipeline e os diferentes tipos de paralisações são assumidos como valores médios por instrução.

Eficiência de pipeline - 2

A quantidade de paralelismo disponível no *bloco básico* de um programa – um segmento de código sem ramificações, exceto no ponto de entrada, e sem ramificações, exceto no ponto de saída – é bastante pequena. Para um programa MIPS típico, por exemplo, a frequência média de ramificação dinâmica costuma estar entre 15% e 25%, o que significa que apenas três a seis instruções são executadas entre um par de ramificações. Além disso, como é provável que essas instruções dependam umas das outras, a quantidade de sobreposição de instruções que pode ocorrer dentro de um bloco básico provavelmente será menor que o tamanho médio do bloco. Portanto, para obter melhorias substanciais de desempenho, o ILP deve ser explorado em vários blocos básicos.

Existem duas abordagens amplamente distintas para atingir esse objetivo

- uma abordagem que depende de tecnologia de software para obter paralelismo estaticamente em tempo de compilação
- uma abordagem que depende de hardware para ajudar a descobrir e explorar dinamicamente o paralelismo inerente durante a execução.

Dependências e perigos de dados

Determinar como uma instrução depende de outra é fundamental para determinar quanto paralelismo existe em um programa e como esse paralelismo pode ser explorado. Em particular, para explorar o ILP de forma eficiente é preciso primeiro determinar quais instruções podem ser executadas em paralelo. Se duas instruções são *independentes* uma da outra, elas podem ser executadas sem impedimentos em um pipeline de profundidade arbitrária (nenhuma parada precisa ser inserida), assumindo que haja recursos suficientes (ou seja, não existam riscos estruturais). Se uma instrução *depender* de outra, haverá restrições de tempo que obrigarão sua execução, elas deverão prosseguir em ordem e muitas vezes poderão ser apenas parcialmente sobrepostas.

Três tipos de dependências são relevantes

- dependências *de dados*, também chamadas de dependências *de dados verdadeiras*
- *nomear* dependências
- *controlar* dependências.

Dependências de dados - 1

Uma instrução j é dita *dependente de dados* de uma instrução i se e somente se qualquer uma das seguintes condições é válida

- a instrução i produz um valor que é usado pela instrução j
- a instrução j é dependente de dados de uma instrução k e a instrução k é dependente de dados da instrução i
- as instruções j e i estão conectadas por uma cadeia de dependências do segundo tipo.

Observe que uma dependência dentro de uma instrução, DADD R1,R1,R1, por exemplo, é não é considerada uma dependência de dados.

Dependências de dados - 2

Considere a seguinte sequência de código

LAÇO: LD	F0,0(R1)	
	ADICIONAR.D F4,F0,F2	↓
	SD F4,0(R1)	↓
	DADDUIR1,R1,-8	↓
	BNE R1,R2,LACO	

Para simplificar, os efeitos das redefinições atrasadas são ignorados. As dependências de dados entre instruções são representadas por setas verticais.

A existência de dependências implica que haverá uma chance de um ou mais perigos de dados entre instruções sucessivas. A execução das instruções em um processador em pipeline com uma unidade de intertravamento e uma profundidade de pipeline maior que a distância entre as instruções medida em ciclos de clock faz com que o processador detecte um perigo e pare, se não puder ser resolvido por encaminhamento. Assim, reduzindo a sobreposição. Por outro lado, em um processador em pipeline sem unidade de intertravamento, é responsabilidade do compilador escalarizar instruções dependentes de maneira que elas não se sobreponham completamente, caso contrário o programa não será executado corretamente.

Dependências de dados - 3

A presença de uma dependência de dados em uma sequência de instruções reflete uma dependência de dados no código fonte a partir do qual a sequência de instruções foi gerada.

O efeito da dependência dos dados originais deve ser preservado. As dependências são, portanto, uma propriedade dos *programas*. Por outro lado, se uma determinada dependência resulta na detecção de um perigo real e se esse perigo realmente causa uma paralisação, são propriedades da *organização do pipeline*. Esta diferença é crucial para compreender como o PVI pode ser explorada.

Uma dependência de dados transmite três ideias

- a possibilidade de um perigo
- a especificação da ordem em que os resultados devem ser computados
- um limite superior para a quantidade de paralelismo que pode ser explorada.

Uma dependência pode ser superada de duas maneiras diferentes

- manter uma dependência, mas evitando um perigo
- eliminar uma dependência através da transformação do código.

Dependências de dados - 4

O escalonamento do código é o principal método para evitar um perigo sem alterar uma dependência e esse escalonamento pode ser feito tanto pelo compilador quanto pelo hardware.

Um valor de dados é comunicado entre instruções através de um registro do banco de registros ou de um local de memória. Quando o fluxo de informações ocorre através de um registrador, detectar a dependência é simples, pois os nomes dos registradores são fixos para cada instrução; entretanto, isso se torna mais complicado se as ramificações intervierem, uma vez que as preocupações com a correção forçam o compilador ou o hardware a serem conservadores. Quando o fluxo de informações ocorre através de locais de memória, a detecção é mais difícil, pois dois endereços podem referir-se ao mesmo local de memória, mas parecerem diferentes. Além disso, o endereço efetivo de uma instrução de carregamento ou armazenamento pode mudar de uma execução da instrução para outra, complicando ainda mais a detecção de uma dependência.

Dependências de nomes - 1

Uma *dependência de nome* está presente quando duas instruções utilizam o mesmo registrador ou local de memória, denominado *nome*, mas sem que ocorra qualquer fluxo de informação entre elas. Existem dois tipos de *dependências de nomes*

- uma *antidependência* entre duas instruções i e j ocorre quando a instrução j escreve em um registrador ou local de memória que a instrução i lê – a ordem original da instrução deve ser preservada para garantir que o valor correto seja lido
- uma *dependência de saída* entre uma instrução i e uma instrução j ocorre quando ambas as instruções i e j escrevem um valor no mesmo registrador ou local de memória – a ordem original da instrução deve ser preservada para garantir que o valor finalmente escrito seja o correto.

Dependências de nomes - 2

Como uma *dependência de nome* não é uma verdadeira dependência de dados, pois não há valor transmitido entre as instruções envolvidas. As próprias instruções podem ser executadas simultaneamente, ou ser renomeadas, desde que o nome (registro ou localização de memória) usado nas instruções seja modificado em ambos. Eles para remover o conflito.

Essa renomeação pode ser feita mais facilmente para operandos de registradores, onde é chamada de *renomeação de registradores*. A renomeação de registradores pode ser realizada estaticamente pelo compilador ou dinamicamente pelo hardware.

Perigos de dados - 1

Existe um *perigo de dados* sempre que existe uma dependência de dados ou nomes entre instruções e elas estão próximas o suficiente para gerar, por sua sobreposição no pipeline durante a execução, uma mudança na ordem de acesso ao operando envolvido na dependência.

Por causa da dependência, a *ordem de execução do programa* deve ser preservada, ou seja, a ordem de execução das instruções se elas forem tomadas uma de cada vez e executadas sequencialmente conforme determinado pelo código-fonte original. O objetivo das técnicas de software e hardware é explorar o paralelismo preservando a ordem do programa *apenas quando isso afeta o resultado do programa*, e não em todas as circunstâncias.

Detectar e evitar perigos garante que a ordem necessária do programa seja preservada.

Perigos de dados - 2

Os *perigos aos dados* podem ser classificados em três categorias diferentes dependendo da combinação de acessos aos operandos presentes nas instruções. É usada uma convenção de nomes que retrata a ordem das instruções que deve ser preservada pelo pipeline.

Considere duas instruções i e j , com i precedendo j no programa, então o possíveis perigos são

- RAW (*leitura após gravação*) – j tenta ler o operando antes de escrever um valor nele, então j obtém um valor errado; é a forma mais comum de perigo de dados e corresponde a uma verdadeira dependência de dados
- WAW (*write after write*) – j tenta escrever um valor em um operando antes de i escrever seu valor nele, então o valor final está errado; surge em pipelines que permitem a escrita em mais de um estágio de pipe, ou permitem a conclusão de instruções fora de ordem, e corresponde a uma dependência de saída

Perigos de dados - 3

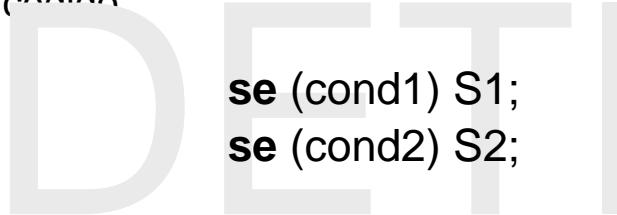
- WAR (*write after read*) – já tenta escrever um valor em um operando antes de ler-lo, então obtém um valor errado; não pode ocorrer na maioria dos pipelines de problemas estáticos, mesmo em pipelines mais profundos ou com operações de ponto flutuante, porque geralmente todas as leituras são antecipadas e todas as gravações atrasadas; surge quando há algumas instruções que gravam os resultados no início do pipeline e outras instruções que leem os operandos mais tarde, ou quando a execução fora de ordem é permitida.

Observe que o RAR (*leitura após leitura*) não representa um risco para os dados. A combinação de operações é idempotente.

Dependências de controle - 1

Uma *dependência de controle* determina a ordem de uma instrução *i* em relação a um desvio, de modo que a instrução *i* seja executada na ordem correta do programa e somente quando deveria ser. Cada instrução, exceto aquelas no primeiro bloco básico do programa, é dependente de controle de algum conjunto de ramificações e, em geral, essas dependências de controle devem ser preservadas para manter a ordem do programa.

Considere o segmento de código



se (cond1) S1;
se (cond2) S2;

S1 é o controle dependente da condição cond1 e S2 é o controle dependente da condição cond2, mas não da condição cond1.

Duas restrições são geralmente impostas pelas dependências de controle

- uma instrução cujo controle depende de uma ramificação, não pode ser movida *antes* da ramificação, de modo que sua execução *não é mais controlada* pela ramificação
- uma instrução que não depende de controle de um desvio, não pode ser movida *após* o desvio, de modo que sua execução seja *controlada* pelo desvio.

Dependências de controle - 2

Quando os processadores preservam a ordem estrita do programa, eles garantem que as dependências de controle também sejam preservadas. Deve-se salientar, entretanto, que é permitido, embora incorrectamente, executar instruções que não deveriam ter sido executadas, violando assim as dependências de controle. Se isso puder ser feito sem afetar a correção do programa. O que significa que a dependência do controle não é a propriedade crítica que deve ser preservada. As duas propriedades críticas para a correção do programa – e normalmente preservadas pela manutenção das dependências de dados e de controle – são o *comportamento de exceção* e o *fluxo de dados*.

Dependências de controle - 3

Preservar o comportamento da exceção significa que quaisquer alterações na ordem de execução das instruções não devem alterar a forma como as exceções são geradas no programa.

Freqüentemente, isso é atenuado para significar que a reordenação da execução da instrução não deve causar novas exceções no programa.

Considere o segmento de código (novamente, os efeitos das ramificações atrasadas são ignorados)

```
DADD R2, R3, R1  
BEC :    R2, .1  
LD      R1, (R2)  
L1: ...
```

É claro que se a dependência de dados envolvendo o registrador R2 não for mantida, o resultado do programa poderá ser alterado. Menos óbvio é o fato de que se alguém ignorar a dependência do controle e mover a instrução de carga para um local imediatamente antes do desvio, esta instrução poderá causar uma violação da proteção de memória.

Para poder reordenar as instruções e ainda preservar a dependência dos dados, seria necessário ignorar a exceção quando o desvio for executado. Posteriormente, será estudada uma técnica de hardware, denominada *especulação*, que, como se verá, permitirá a superação deste problema.

Dependências de controle - 4

O *fluxo de dados* é o fluxo real de valores de dados entre as instruções que os produzem e as instruções que os consomem. As ramificações tornam o fluxo de dados dinâmico, pois permitem que os dados de uma determinada instrução venham de diferentes locais de origem, portanto, uma instrução pode ser dependente de dados de várias instruções predecessoras. A ordem do programa é o que de fato determina qual antecessor entregará o valor em cada momento e a ordem do programa é garantida pela manutenção das dependências de controle.

Considere o segmento de código (novamente, os efeitos das ramificações atrasadas são ignorados)

PAI R1, R2, R3
BEQZ R4,L
DSUBU R1, R5, R6

EU:
...
OU R7,R1,R8

Como pode ser visto, o valor do registrador R1 utilizado pela instrução *ou* depende se o desvio é realizado ou não. A dependência dos dados por si só não é suficiente para preservar a correção; a dependência do controle também é necessária. Quanto ao problema da exceção, a *especulação* também ajudará a diminuir o impacto da dependência do controlo, mantendo ao mesmo tempo o fluxo de dados.

Dependências de controle - 5

Às vezes pode ser determinado que a violação da dependência de controle não afetará o comportamento da exceção nem o fluxo de dados.

Considere o segmento de código (novamente, os efeitos das ramificações atrasadas são ignorados)

```
PAI R1, R2, R3  
BEQZ      R12, Pular  
DSUB R1, R5, R6  
PAI R5, R4, R9  
.  
Pular: OU      R7, R8, R9
```

Suponha que se saiba que os registradores R4 e R5, destinos das instruções de *subtração* e *adição*, respectivamente, não são usados após SKIP – a propriedade de saber se um valor será usado por uma instrução futura é chamada *de vivacidade*. Então, alterar os valores de R4 e R5 logo antes da ramificação não afetaria o fluxo de dados, pois R4 e R5 estariam mortos na região de código após SKIP.

Esse tipo de escalonamento de código também é uma forma de especulação, muitas vezes chamada de *especulação de software*. O compilador está apostando no resultado do ramo, a aposta sendo o ramo geralmente não é aceita.

Técnicas básicas de compilador para expor ILP - 1

Para manter um pipeline cheio, o paralelismo entre as instruções deve ser explorado, encontrando sequências de instruções não relacionadas que podem ser sobrepostas no pipeline.

Para evitar um bloqueio no pipeline, a execução de uma instrução dependente deve ser separada da instrução fonte por uma distância em ciclos de clock pelo menos igual à latência do pipeline dessa instrução fonte.

A capacidade do compilador de realizar esse escalonamento depende tanto da quantidade de ILP disponível no programa quanto das latências das unidades funcionais do pipeline.

Como exemplo, considere como o compilador pode aumentar a quantidade de recursos disponíveis ILP transformando loops como o abaixo

```
para (eu = 999; eu >= 0; eu--)  
    x[eu] += s;
```

Como se vê imediatamente, o loop é altamente paralelo: cada iteração é totalmente independente de qualquer outro.

Técnicas básicas de compilador para expor ILP - 2

Código MIPS direto do loop (ramificações atrasadas são ignoradas)

LAÇO:	LD	F0,0(R1)
	ADD	4,F0,F2
	SD	F4,-(R1)
	DAI	R1,-
	BNE	R1,R2,LAÇO

Supõe-se que o registrador R1 contém inicialmente o endereço do elemento x[999] do array e que o conteúdo do registrador R2+8 é o endereço do elemento x[0].

Técnicas básicas de compilador para expor ILP - 3

Latências de operações FP

Fonte: Arquitetura de Computadores: Uma Abordagem Quantitativa

Instrução produzindo o resultado	D	TF	Instrução produzindo o resultado	T	I	Latência em ciclos de clock
Operação FP ALU			outra operação FP ALU			3
Carga			loja dupla			2
operacional FP ALU dupla			Operação FP ALU			1
carregar duplo			loja dupla			0

Técnicas básicas de compilador para expor ILP - 4

**Código MIPS do loop, com eliminação de dependências de dados
(ramificações atrasadas são ignoradas)**

		<i>ciclo do relógio</i>
LAÇO: LD	F0,0(R1)	1
<i>parar</i>		2
ADI D F4,, 0,F	<i>parada</i>	3
<i>parada</i>		4
SD F4,0(R1)		5
DADDUI R1,R1,-8	<i>tenda</i>	6
<i>tenda</i>		7
BNE R1,R2,LAÇO		8
		9

O processamento de cada elemento leva 9 ciclos de clock (assume que não há encaminhamento no estágio ID).

Técnicas básicas de compilador para expor ILP - 5

**Código MIPS do loop, agendado para o pipeline
(ramificações atrasadas são ignoradas)**

		<i>ciclo do relógio</i>
LAÇO:	LD	1
		2
	DADDUIR1,R1,-8	3
	ADI D F4,F ,F <i>parada</i>	4
	<i>parar</i>	5
	SD F4,8(R1)	6
	BNE R1,R2,LAÇO	7

O processamento de cada elemento leva 7 ciclos de clock.

Técnicas básicas de compilador para expor ILP - 6

Código MIPS simples do loop, com o loop desenrolado 4 vezes
(ramificações atrasadas são ignoradas)

```
LAÇO: LD      F0,0(R1)
       ADICIONAR.D F4,F0,F2
       DP      F4,0(R1)
       LD F6,-8(R1)
       ADICIONAR.D F8,F6,F2
       SD      F8,-8(R1)
       LD F10,-16(R1)
       ADICIONAR.D F12,F10,F2
       SD      F12,-16(R1)
       LD F14,-24(R1)
       ADICIONAR.D F16,F14,F2
       SD      F16,-24(R1)
       DADDUI R1,R1,-32
       BNE     R1,R2,LAÇO
```

Em programas reais, o limite superior do loop geralmente não é conhecido. Supondo que seja n e desenrolando o loop k vezes, dois loops consecutivos são gerados: o primeiro itera $n \bmod k$ vezes e tem corpo igual ao loop original; o segundo itera n / k vezes e tem o corpo desenrolado.

Técnicas básicas de compilador para expor ILP - 7

**Código MIPS do loop, com loop desenrolado 4 vezes e programado para o pipeline
(ramificações atrasadas são ignoradas)**

LAÇO: LD	F0,0(R1)
LD	F6,-8(R1)
LD	F10,-16(R1)
LD	F14,-24(R1)
ADICIONAR.D	F4,F0,F2
ADICIONAR.D	F8,F6,F2
ADICIONAR.D	F12,F10,F2
ADICIONAR.D	F16,F14,F2
DP	F4,0(R1)
SD	F8,-8(R1)
DADDUIR1,R1,-32	
SD	F12,16(R1)
SD	F16,8(R1)
BNE	R1,R2,LAÇO

O processamento de cada elemento leva 3,5 ciclos de clock.

Técnicas básicas de compilador para expor ILP - 8

As decisões e transformações necessárias para desenrolar um loop podem ser resumidas nos seguintes pontos

- determinar se o desenrolar do loop é realmente útil, encontrando o grau de independência entre suas iterações
- uso de registros diferentes para evitar restrições desnecessárias que surgiriam pelo uso dos mesmos registros para alguns alterantes
- eliminação das instruções excessivas, otimização e renificação e ajuste da terminação do loop e do código de iteração
- determinação se as cargas e armazenamentos no loop desenrolado podem ser intercambiados – esta transformação requer a análise dos endereços de memória
- agendar o código preservando quaisquer dependências necessárias para produzir o mesmo resultado que o código original.

O principal requisito subjacente a todas estas transformações é a compreensão de como uma instrução depende de outra e como as diferentes instruções podem ser alteradas e/ou reordenadas, dadas as dependências.

Técnicas básicas de compilador para expor ILP - 9

Três efeitos diferentes limitam os ganhos do desenrolar do loop

- uma diminuição na quantidade de despesas gerais economizadas com cada desenrolamento – conforme previsto pela Lei de Amdhal
- limitações de tamanho de código – para loops grandes, o crescimento do tamanho do código pode levar a um aumento na taxa de falha do cache de instruções; além disso, é preciso também nos preocupar com a potencial escassez de registros, chamada pressão de registros, que é criada pela aplicação de estratégias agressivas de desenrolamento e escalonamento
- limitações do compilador – o uso de transformações sofisticadas de alto nível, cujas melhorias potenciais são difíceis de medir antes da geração detalhada do código, levou a aumentos significativos na complexidade dos compiladores modernos.

Previsão avançada de ramificação

As ramificações prejudicarão o desempenho do pipeline porque a aplicação de dependências de controle gera riscos que exigem a paralisação do progresso do pipeline em muitas situações. O desenrolamento do loop é um método para reduzir o número de riscos de controle durante a execução. A perda de desempenho das liais, entretanto, é tratada de forma mais adequada se seu comportamento puder ser previsto.

Preditores de ramificação ~~que~~ que dependem de informações de tempo de compilação ou do comportamento dinâmico observado de uma ramificação isoladamente já foram estudados. À medida que o número de instruções em voo, isto é, cuja execução está ocorrendo ou sendo considerada, aumentou significativamente nas últimas duas décadas, a necessidade de previsão precisa de desvios tornou-se mais premente.

Correlacionando preditores de ramificação - 1

O esquema de *predição de 2 bits* leva em consideração o comportamento recente de uma única ramificação para prever seu comportamento futuro. A precisão da previsão é melhorada se considerarmos também o comportamento recente de *outras* ramificações do programa.

Para ver por que essa ideia é relevante, observe o segmento de código a seguir, onde os efeitos das ramificações atrasadas são ignorados.

```

DAUDIU R3,R1    ; ramo b1: R1 ≠ 2?
BNZ   R3, L1      ; R1 = 0
PAI   R1,R0,R0
L1:  DSUBU R3,R2    ; ramo b2: R2 ≠ 2?
      BNEZ R3, L2      ; R2 = 0
      PAI  R2,R0,R0
L2:  BEQZ R3,L3      ; ramo b3: R1 = R2?

```

Observe que o comportamento do ramo *b3* está correlacionado ao comportamento dos ramos *b1* e *b2*: se ambos os ramos não forem *retirados*, então o ramo *b3* será *tomado*. Um preditor que usa apenas o comportamento de uma única ramificação para prever o que acontece a seguir nunca poderá capturar tal comportamento.

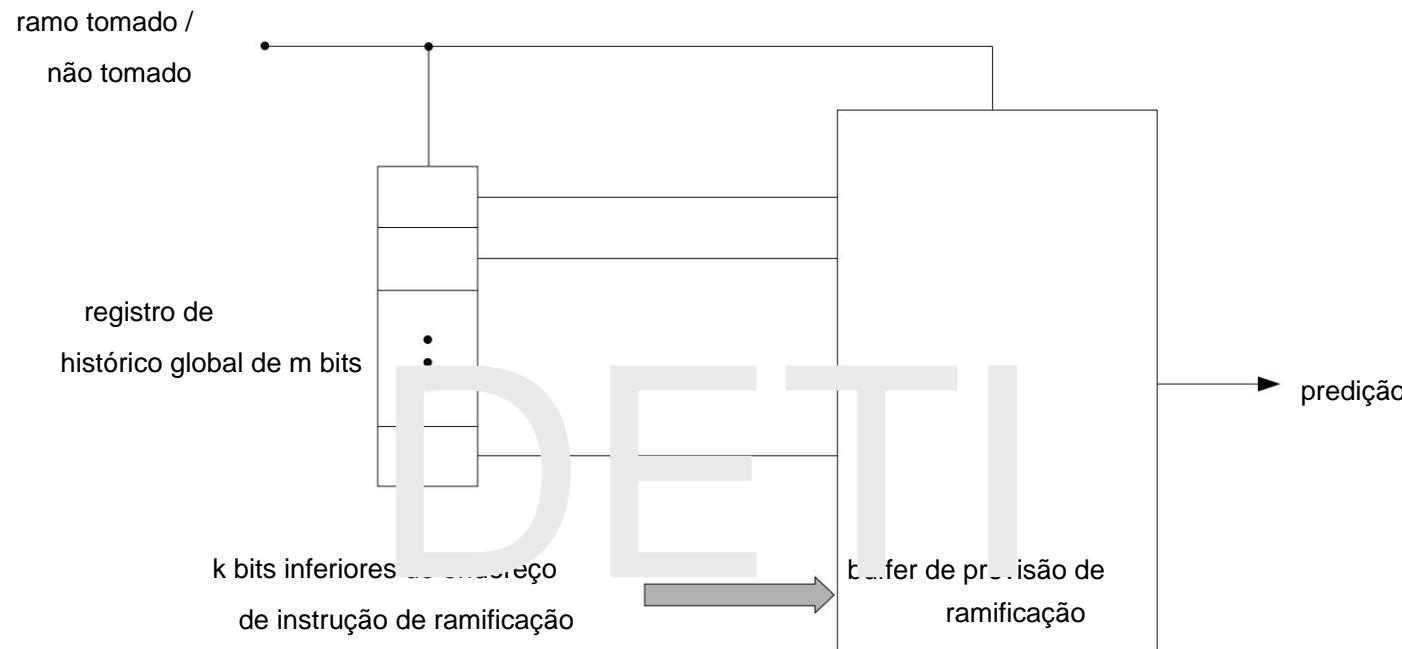
Correlacionando preditores de ramificação - 2

Preditores de ramificação que adicionam o comportamento de execução de outras ramificações, além dos seus próprios, para fazer uma previsão, são chamados de *preditores correlacionados*.

Um *preditor de ramificação correlacionada* (1,2) , por exemplo, considera o comportamento de execução da ramificação anterior para escolher entre um par de preditores de ramificação de 2 bits na previsão de uma ramificação específica. Em geral, um *preditor de ramificação* ($m.n$) adiciona o comportamento de execução de m ramificações anteriores para escolher entre m preditores de ramificação de n bits na previsão de uma ramificação específica.

A popularidade deste tipo de preditores de ramificação é que ele pode produzir uma taxa de predição mais alta do que o esquema de predição de 2 bits, ao mesmo tempo que requer uma quantidade trivial de hardware adicional: o histórico global das m ramificações mais recentes pode ser registrado em um m - registrador de deslocamento de estágio, denominado *registrator de histórico global*, onde o conteúdo do bit em cada estágio especifica se o ramo da ordem correspondente foi *obtido*, 1, ou *não obtido*, 0; o *buffer de previsão de desvio* agora é indexado pela concatenação do histórico global de m bits com os k bits de endereço de ordem inferior da instrução de desvio.

Correlacionando preditores de ramificação - 3



Um preditor de 2 bits sem histórico global é simplesmente um preditor de ramificação (0,2) correlacionado e, ao comparar preditores de ramificação, o tamanho do buffer do preditor de ramificação deve ser mantido invariante, ou seja,

$$2^{eu} \cdot 2^k = \text{constante.}$$

Correlacionando preditores de ramificação - 4

Um (2,2) preditor de ramificação correlacionado com 1K entradas é comparado com (0,2) simples preditores com entradas 4K e um número ilimitado de entradas.

Comparando preditores de ramificação de 2 bits de diferentes tipos em benchmarks SPEC89

Fonte: Arquitetura de Computadores: Uma Abordagem Quantitativa



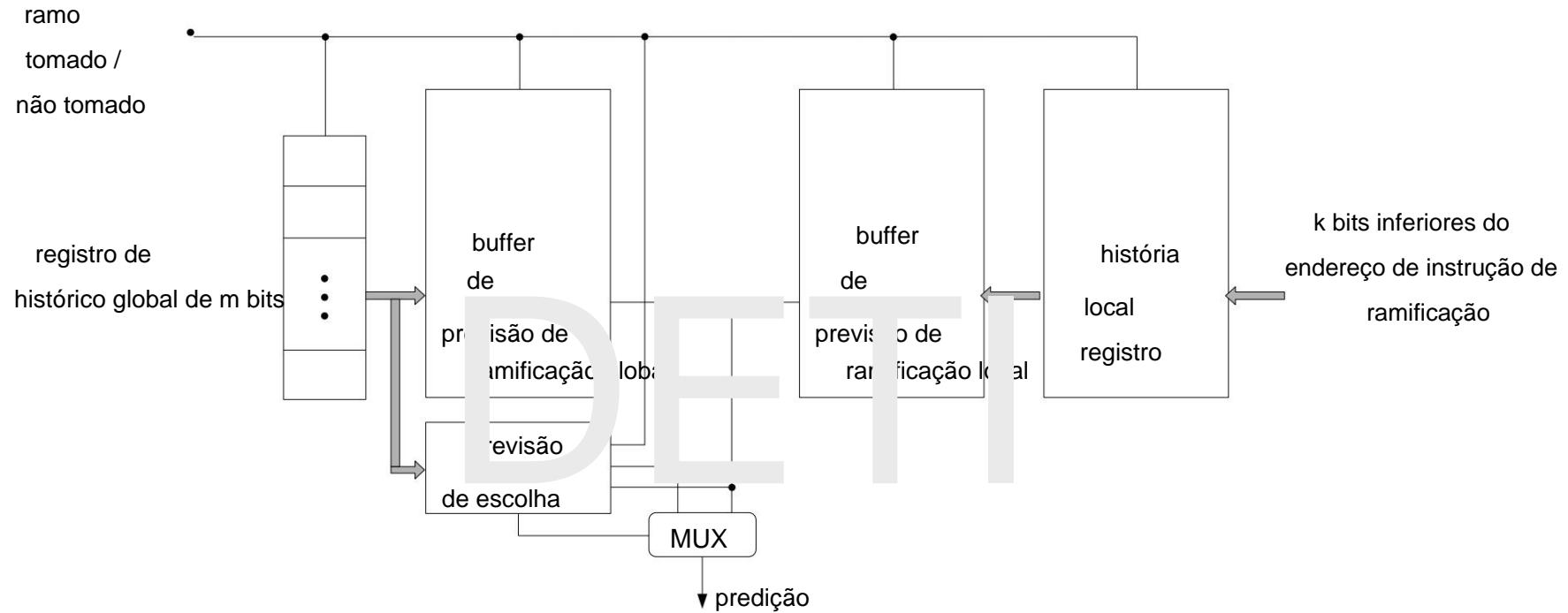
Preditores de torneio - 1

A principal motivação para correlacionar preditores de ramificação veio da observação de que o preditor padrão de 2 bits usando apenas informações locais falhou em algumas ramificações importantes e que, adicionando informações globais, o desempenho poderia ser melhorado. Os *preditores de torneio* levam esse insight para o próximo nível usando vários preditores, geralmente um baseado em informações globais e outro em informações locais, e combinando-os com um seletor. Os preditores de torneio podem obter melhor precisão em tamanhos médios do buffer do preditor de ramificação e também fazer uso eficaz de um grande número de bits de predição.

Os preditores de torneios existentes usam um contador de histerese de 2 bits por ramificação para escolher entre dois preditores diferentes com base em qual preditor (local, global ou uma mistura dos dois) foi o mais eficaz nas previsões recentes. Um *contador de histerese* de 2 bits requer duas previsões erradas sucessivas para mudar seu estado.

A vantagem dos preditores de torneio é a sua capacidade de selecionar o preditor certo para um ramo específico, o que é especialmente crucial para programas inteiros. Normalmente, eles selecionam o preditor global quase 40% das vezes para benchmarks de número inteiro e menos de 15% das vezes para benchmarks de ponto flutuante.

Preditores de torneio - 2



Tanto o *buffer de previsão de ramificação global* quanto o *buffer de previsão de escolha* consistem em preditores de $2^m n$ bits e o *buffer do preditor de ramificação local* de preditores de $2^u v$ bits. O *registro de histórico local*, por outro lado, consiste em registros de histórico local de $2^k u$ bits .

Preditores de torneio - 3

Comparando a taxa de previsão incorreta para três preditores diferentes nos benchmarks SPEC89

Fonte: Arquitetura de Computadores: Uma Abordagem Quantitativa



Embora esses dados sejam de uma versão mais antiga do SPEC, os benchmarks recentes do SPEC mostram um comportamento semelhante, talvez convergindo para os limites assintóticos em tamanhos de preditores ligeiramente maiores.

Agendamento dinâmico - 1

Uma grande limitação das técnicas simples de pipeline é que elas usam emissão e execução de instruções *em ordem*. Se ocorrer uma dependência de dados entre uma instrução no pipeline e uma instrução recebida, que não pode ser resolvida pela *unidade de encaminhamento*, a *unidade de intertravamento* paralisa o pipeline, começando na instrução que utiliza o resultado. Nenhuma nova instrução é obtida ou emitida até que a dependência seja eliminada.

O escalonamento dinâmico é outra maneira de resolver o problema: o hardware reorganiza a execução das instruções, enquanto mantém o fluxo de dados e o comportamento de exceção, de modo que as paralisações sejam minimizadas. Isto implica, no entanto, um aumento significativo na complexidade.

São diversas as vantagens resultantes da utilização desta técnica:

- permite que o código que foi compilado com um pipeline em mente seja executado com eficiência em um pipeline diferente, eliminando a necessidade de vários binários
- permite lidar com alguns casos onde as dependências são desconhecidas em tempo de compilação, envolvendo, por exemplo, referências de memória e ramificações dependentes de dados ou o uso de bibliotecas vinculadas dinâmicas
- permite que o processador lide com atrasos imprevisíveis, como falhas de cache, executando outro código enquanto espera a resolução da falha.

Agendamento dinâmico - 2

Considere o segmento de código.

```
DIV.D F0,F2,F4  
ADICIONAR.D F10,F0,F8  
SUB.D F12,F8,F14
```

A instrução SUB.D não pode ser executada porque a dependência de ADD.D em DIV.D faz com que o pipeline pare; ainda assim, SUB.D não depende dos dados das duas instruções anteriores. A limitação de desempenho criada por este perigo poderia ser eliminada pela não exigência de instruções para execução na ordem do programa.

Para cumprir este objetivo, o processo de emissão poderia ser decomposto em duas partes: verificação de perigos estruturais e espera pela ausência de perigo nos dados. Assim, *em ordem*. O problema de instrução ainda é usado, mas uma instrução pode iniciar a execução assim que seus operandos de dados estiverem disponíveis. Um pipeline com esses recursos apresenta desempenho *fora de ordem* execução, o que também implica conclusão *fora de ordem*.

Agendamento dinâmico - 3

A execução *fora de ordem* introduz a possibilidade de perigos WAR e WAW, que não existiam no pipeline clássico de 5 estágios, o primeiro, e apenas com operações de ponto flutuante multiciclo que dão origem à conclusão *fora de ordem*, esta última.

Considere o segmento de código:

```
DIV.D F0,F2,F4  
ADIC.DNAR F0,F4,F0,F8  
SUB.D F0,F10,F4  
MUL.D F6,F10,F8
```

Existe uma *antidependência* entre as instruções ADD.D e SUB.D e uma *dependência de saída* entre as instruções ADD.D e MUL.D. Ambos os perigos poderiam ser removidos se a *renomeação de registro* fosse usada.

Agendamento dinâmico - 4

A conclusão *fora de ordem* também cria grandes complicações no tratamento de exceções. O escalonamento dinâmico deve preservar o comportamento de exceção no sentido de que *exatamente* aquelas exceções que surgiriam se o programa fosse executado em uma ordem estrita de programa *realmente* surgem. Os processadores escalonados dinamicamente preservam o comportamento da exceção atrasando a notificação de uma exceção associada até que o processador saiba que a instrução envolvida é a próxima a ser concluída.

Embora o comportamento de exceção deva ser preservado, processadores agendados dinamicamente podem gerar exceções *imprecisas*. Exceções *imprecisas* podem ocorrer devido aos dois fatos abaixo:

- o pipeline pode *já ter concluído* algumas instruções que obtiveram sucesso na ordem do programa, a instrução que causou a exceção e cujo resultado não pode ser revertido
- o pipeline pode *ainda não ter concluído* algumas instruções que precedem na ordem do programa a instrução que causa a exceção.

A forma de resolver este problema e obter exceções precisas será discutida mais adiante no contexto dos processadores *especulativos*.

Agendamento dinâmico - 5

Para permitir a execução *fora de ordem*, o estágio de ID do pipeline clássico de 5 estágios é dividido em duas etapas

- *emissão* – decodificação de instruções e verificação de riscos estruturais
- *leitura de operandos* – aguardando até que todos os riscos de dados sejam eliminados antes de ler os operandos.

Um estágio *de busca de instruções* precede o estágio *decodificação*; a instrução que foi buscada é colocada em um *registrator de instruções* ou em uma *fila de instruções pendentes*; as instruções são então emitidas do registro ou da fila, se as condições permitirem. A etapa *de execução* segue a etapa *de leitura dos operandos*. Dependendo da operação, a etapa *de execução* pode levar um número variável de ciclos.

Nesse sentido, o momento em que uma instrução *inicia* a execução deve ser diferenciado do momento em que a instrução *completa* a execução; a instrução está *em execução* entre esses dois tempos. Ter múltiplas instruções *em execução* ao mesmo tempo requer múltiplas unidades funcionais, unidades funcionais em pipeline ou ambas.

Dado que estas duas capacidades são essencialmente equivalentes no que diz respeito ao controlo de pipeline, será assumido que o processador possui múltiplas unidades funcionais.

Agendamento dinâmico - 6

Em um pipeline agendado dinamicamente, todas as instruções passam pelo estágio *de emissão em ordem*. Eles podem, no entanto, ser paralisados e ultrapassados por outros na fase *de leitura dos operandos* e entrar em execução *fora de ordem*.

Existem duas técnicas básicas que permitem que instruções sejam executadas *fora de ordem* quando há recursos suficientes e não há dependências de dados entre eles: o *scoreboard* foi a primeira técnica a ser introduzida, surgindo no projeto do supercomputador CDC 6600 em meados da década de 1960. O *algoritmo de Tomasulo* foi o segundo, desenvolvido por Robert Tomasulo em 1967 e aplicado à unidade de ponto flutuante do IBM 360/91.

A principal diferença entre eles é que o *algoritmo de Tomasulo* lida com antidependências e dependências de saída renomeando de forma eficaz e dinâmica os registradores. Além disso, também pode ser estendido para lidar com *especulação*, uma técnica que visa reduzir o efeito das dependências de controle, prevendo o resultado de uma ramificação através da execução de instruções no endereço alvo previsto e tomando ações corretivas quando a previsão estiver errada.

Placar - 1

Scoreboarding - 1

O objetivo de um *placar* é tentar manter uma taxa de execução de uma instrução por ciclo de clock, desde que não haja riscos estruturais. Assim, as instruções são executadas o mais cedo possível. Quando uma instrução emitida é paralisada, outras instruções da tabela de processamento, que não dependem de nenhuma instrução ativa ou paralisada, são consultadas e se alguma for encontrada, ela será executada. O *placar* assume o controle total da emissão e execução das instruções, incluindo todas as detecções de perigos.

Em um processador com arquitetura MIPS, os placares fazem sentido principalmente na unidade de ponto flutuante, já que a latência das demais unidades funcionais é muito pequena. Será assumido que existem dois multiplicadores, um somador, um divisor e uma única unidade inteira para todas as referências de memória, ramificações e operações inteiras.

Placar - 2

Organização básica de um processador MIPS com placar

Fonte: Arquitetura de Computadores: Uma Abordagem Quantitativa

DETI

Placar - 3

Cada instrução passa por quatro etapas de processamento enquanto está sob controle do placar. Na verdade, isso é uma simplificação da situação real, uma vez que o acesso à memória, necessário para operações de carregamento e armazenamento, está sendo desconsiderado. As quatro etapas, que substituem ID, EX e WB no pipeline padrão, são as seguintes

1. *Emitir* – se uma unidade funcional FP do tipo requerido estiver livre e nenhuma outra instrução ativa tiver o mesmo registo com destino, o placar emite a instrução para a unidade e atualiza sua estrutura de dados interna – esta etapa substitui a primeira metade do Estágio de ID do pipeline MIPS ao assegurar que nenhuma outra unidade funcional activa escreve o seu resultado no mesmo registo de destino, é garantido que não estão presentes perigos WAW; se existir um perigo estrutural ou WAW, a emissão de instruções será interrompida e nenhuma instrução adicional será emitida até que esses perigos sejam eliminados; quando o estágio de emissão para, faz com que o buffer entre a busca de instrução e a emissão seja preenchido e a busca de instrução pare imediatamente, se o buffer for um registo único, ou quando o buffer estiver cheio, se for uma fila.

Placar - 4

2. *Ler operandos* – o placar monitora a disponibilidade dos registradores fonte: um registrador fonte está disponível se nenhuma instrução anterior, ainda em execução, for escrevendo-o – esta etapa substitui a segunda metade do estágio de identificação do pipeline MIPS; quando os operandos fonte estão disponíveis, o placar permite que a unidade funcional proceda à leitura dos operandos dos registradores e inicie a execução; Os perigos RAW são resolvidos dinamicamente nesta fase e as instruções podem ser enviadas para a execução fora de ordem.
3. *Execução* – a unidade funcional inicia a execução ao receber seus operandos; quando o resultado estiver pronto, ele notifica o placar de que concluiu a operação – esta etapa substitui o estágio EX do pipeline MIPS e leva um número variável de ciclos de clock no pipeline MIPS FP.
4. *Escreva o resultado* – assim que o placar estiver ciente de que a unidade funcional terminou Após sua operação, ele verifica os riscos de WAR e, se necessário, impede a conclusão da instrução – esta etapa substitui o estágio WB do pipeline MIPS.

Placar - 5

À primeira vista, pode parecer que o painel de avaliação tem dificuldade em distinguir entre os perigos RAW e WAR. Como os operandos de uma instrução são somente lidos quando o conteúdo de ambos os registradores de origem tem o valor atualizado, o placar não aproveita o encaminhamento. Esta não é uma grande penalidade, uma vez que as instruções escrevem o seu resultado no registo de destino assim que completam a sua execução (desde que não haja perigos de WAR). A consequência é uma latência reduzida da pipeline e os benefícios do encaminhamento são alcançados de forma indireta. Há no entanto, um ciclo de clock extra adicionado à latência porque as operações de gravação do resultado e leitura do operando não podem se sobrepor.

Com base na sua própria estrutura de dados interna, o painel de avaliação controla a progressão das instruções de um passo para o seguinte através da comunicação com as unidades funcionais. Ainda há uma pequena complicação. Dado que o número de autocarros que ligam o banco de registos e as unidades funcionais é limitado, conduzindo a riscos potencialmente estruturais, o painel de avaliação deve garantir que o número de unidades funcionais autorizadas a avançar para as etapas 2 e 4 não excede o número de autocarros disponíveis.

Placar - 6

A estrutura de dados interna do placar consiste em três elementos

- *status da instrução* – é uma tabela com tantas entradas quanto o número de instruções em processamento; para cada instrução, especifica em qual dos quatro estados a instrução está
- *estado da unidade funcional* – é uma tabela com tantas entradas quanto o número de unidades funcionais; cada entrada tem nove campos
 - \ddot{y} ocupado – indica se a unidade está ocupada ou livre
 - \ddot{y} op – indica a operação a ser realizada na unidade
 - \ddot{y} F – número do registrador destino
 - \ddot{y} Fj , F – números dos registradores fonte
 - \ddot{y} Qj , Qk – identificação das unidades funcionais cujo resultado será armazenado nos registradores fonte Fj e Fk
 - \ddot{y} Rj , Rk – flags sinalizando quando os registradores fonte estão prontos para serem lidos e ainda não foram lidos
- *situação do resultado do cadastro* – é uma tabela de entrada única com tantos campos quantos os registros do banco de cadastros; indica qual unidade funcional escreverá o registrador se uma instrução ativa tiver o registrador como destino.

Placar - 7

Considere o segmento de código

```
LD      F6,34(R2)
LD      F2,45(R3)
MUL.D F0,F2,F4
SUB.D F8,F6,F2
DIV.D F10,F0,F6
ADICIONAR.D F6,F8,F2
```

Existem verdadeiras dependências de dados entre a primeira instrução LD e a SUB.D e instruções DIV.D , entre a segunda instrução LD e as instruções MUL.D, SUB.D e instruções ADD.D , entre as instruções MUL.D e DIV.D e entre as instruções SUB.D e ADD.D , potencialmente levando a perigos de RAW. Há também uma antidependência entre as instruções SUB.D e DIV.D e as instruções ADD.D instrução e uma dependência de saída entre a primeira instrução LD e a instrução ADD.D instrução, potencialmente levando a perigos WAR e WAW, respectivamente.

Suponha as seguintes latências: carga – 1 ciclo de clock, adição – 2 ciclos de clock, multiplicação – 6 ciclos de clock e divisão – 12 ciclos de clock.

Placar - 8

Componentes da estrutura de dados interna do placar quando a segunda instrução de carregamento está prestes a gravar o resultado no registrador de destino

<i>Status da instrução</i>									
<i>Instrução</i>	<i>Emitir</i>	<i>Ler operandos</i>	<i>Execução encerrada</i>			<i>Escrever resultado</i>			
LD F6,34(R2)	sim	sim				sim			sim
LD F2,45(R3)	sim	sim				sim			
MUL.D F0,F2,F4	sim								
SUB.D F8,F6,F2	sim								
DIV.D F10,F0,F6	sim								
ADICIONAR.D F6,F8,F2	sim								

<i>Status da unidade funcional</i>									
<i>Nome</i>	<i>ocupado</i>	<i>operação</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rio</i>	<i>Rk</i>
inteiro	sim	carregar	F2	R3					não
mult1	sim	mult	F0	F2	F4	inteiro		não	sim
mult2	não								
adicionar	sim	sub	F8	F6	F2		inteiro	sim	não
dividir	sim	divisão	F10	F0	F6	mult1		não	sim

<i>Registrar status do resultado</i>									
<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	<i>...</i>	<i>F30</i>	
mult1	inteiro			adicionar	dividir				

Placar - 9

Componentes da estrutura de dados interna do placar quando a instrução de multiplicação está prestes a escrever o resultado no registrador de destino

<i>Status da instrução</i>									
<i>Instrução</i>	<i>Emitir</i>	<i>Ler operandos</i>	<i>Execução encerrada</i>			<i>Escrever resultado</i>			
LD F6,34(R2)	sim	sim				sim			sim
LD F2,45(R3)	sim	sim				sim			sim
MUL.D F0,F2,F4	sim	sim				sim			
SUB.D F8,F6,F2	sim	sim				sim			sim
DIV.D F10,F0,F6	sim								
ADICIONAR.D F6,F8,F2	sim	sim				sim			

<i>Status da unidade funcional</i>									
<i>Nome</i>	<i>ocupado</i>	<i>operação</i>	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rio</i>	<i>Rk</i>
inteiro	não								
mult1	sim	mult	F0	F2	F4			não	não
mult2	não								
adicionar	sim	adicionar	F6	F8	F2			não	não
dividir	sim	divisão	F10	F0	F6	mult1		não	sim

<i>Registrar status do resultado</i>									
<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	<i>...</i>	<i>F30</i>	
mult1				adicionar		dividir			

Placar - 10

Componentes da estrutura de dados interna do placar quando a instrução de divisão está prestes a gravar o resultado no registrador de destino

Instrução	Emitir	Ler operandos	Execução encerrada	Escrever resultado
LD F6,34(R2)	sim	sim	sim	sim
LD F2,45(R3)	sim	sim	sim	sim
MUL.D F0,F2,F4	sim	sim	sim	sim
SUB.D F8,F6,F2	sim	sim	sim	sim
DIV.D F10,F0,F6	sim	sim	sim	sim
ADICIONAR.D F6,F8,F2	sim	sim	sim	sim

Status da unidade funcional									
Nome	ocupado	operação	<i>Fi</i>	<i>Fj</i>	<i>Fk</i>	<i>Qj</i>	<i>Qk</i>	<i>Rio</i>	<i>Rk</i>
inteiro	não								
mult1	não								
mult2	não								
adicionar	não								
dividir	sim	divisão	F10	F0	F6	mult1		não	não

Registrar status do resultado									
<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>	
					dividir				

Placar - 11

Verificações e ações contábeis necessárias para cada etapa na execução de instruções quando um placar é usado

Status da instrução	Espera até	Escrituração contábil
Emitir	(ocupado[FU] == não) && (regResult[DR] == vazio)	ocupado[FU] = sim; op[FU] = op; Fi[FU] = DR; Fj[FU] = SR1; Fk[FU] = SR2; Qj[FU] = regResult[SR1]; Qk[FU] = regResult[SR2]; Rj[FU] = (Qj[FU] == vazio) ? sim não; Rk[FU] = (Qk[FU] == vazio) ? sim não; regResult[DR] = FU;
Ler operandos	(Rj[FU] == sim) && (Rk[FU] == sim)	Qj[FU] = vazio; Qk[FU] = vazio; Rj[FU] = não; Rk[FU] = não;
Execução	Operação FU encerrada	
Escrever resultado	ÿf se (Qj[f] == FU) Rj[f] == sim; ÿf se (Qk[f] == FU) Rk[f] == sim; regResult[Fi[FU]] = vazio; ocupado[FU] = não;	

onde FU é a unidade funcional associada à instrução, SR1 e SR2
 seus registradores de origem e DR seu registrador de destino. O teste na fase de gravação
 do resultado é realizado para evitar riscos de WAR.

Placar - 12

Um placar usa o ILP disponível para minimizar o número de paralisações decorrentes das verdadeiras dependências dos dados do programa. Ao eliminar as paralisações, um placar é limitado por vários fatores

- *a quantidade de paralelismo inherente ao programa* – este fator determina se instruções independentes podem ser encontradas; se cada instrução depende de sua antecessora, nenhum esquema de escalonamento dinâmico pode reduzir o número de paralisações
- *o número de entradas no placar* – esse fator determina o quanto longe o pipeline pode procurar por instruções independentes
- *o número e os tipos de unidades funcionais* – este fator determina quanto relevantes são os riscos estruturais, que podem aumentar quando a programação dinâmica é usada
- *a presença de antidependências e dependências de resultados* – o que leva a perigos de GUERRA e MTA e pode gerar mais estagnações.

Algoritmo de Tomasulo - 1

No esquema desenvolvido por Robert Tomasulo, os perigos RAW são evitados executando uma instrução apenas quando seus operandos estão disponíveis, que é exatamente o que o placar mais simples fornece. Os perigos WAR e WAW, que surgem de dependências de nomes, são eliminados pela renomeação de registros. A *renomeação de registradores*, na verdade, elimina esses riscos, alterando o nome de todos os registradores de destino, incluindo aqueles com leitura ou gravação pendente de uma instrução anterior, de modo que a gravação fora de ordem não afete nenhuma instrução que dependa do mais antigo. valor do operando.

Algoritmo de Tomasulo - 2

Para entender melhor como funciona a renomeação de registradores, considere a seguinte sequência de código

```
DIV.D F0,F2,F4  
ADICIONAR.D F6,F0,F8  
SD F6, (R1)  
SUB D F8,F0,F4  
MUL D F6,F0,F4
```

Existem antidependências entre a instrução ADD.D e a SUB.D e entre a instrução SD e a instrução MUL.D , e uma dependência de saída entre a instrução ADD.D e a instrução MUL.D , potencialmente levando a perigos WAR e WAW, respectivamente. Existem também dependências de dados verdadeiras entre a instrução DIV.D e a instrução ADD.D , entre a instrução ADD.D e a instrução SD e entre a instrução SUB.D e a instrução MUL.D.

Algoritmo de Tomasulo - 3

As três dependências de nomes podem ser eliminadas pela renomeação de registradores. Para simplificar, assuma a existência de dois registradores temporários, S e T. A sequência de código pode ser reescrita sem qualquer dependência de nome como

```
DIV.D F0,F2,F4  
ADICIONA.S DS F0,F8  
SD S,0 '1'  
SUE DT,F10,F14  
MUL D DS, 10,T
```

Além disso, quaisquer usos subsequentes do registro F8 devem ser substituídos pelo registro T. Neste segmento de código, o processo de renomeação pode ser feito estaticamente pelo compilador. Encontrar quaisquer usos do registrador F8 que apareçam posteriormente no código requer uma análise sofisticada do compilador ou suporte de hardware, uma vez que pode haver ramificações intermediárias entre o segmento de código e um uso posterior de F8.

Algoritmo de Tomasulo - 4

No esquema de Tomasulo, a renomeação de registradores é fornecida por estações de reserva, que armazenam em buffer os operandos das instruções que aguardam execução. A ideia básica é que uma estação de reserva busque e armazene um operando em buffer assim que ele estiver disponível, eliminando a necessidade de obter o operando de um registrador do banco de registradores. As instruções pendentes designam o buffer de reserva que fornecerá sua entrada. Finalmente, quando escritas sucessivas no mesmo registrador se sobrepõem na execução, apenas a última na ordem do programa irá realmente atualizar o registrador. À medida que as instruções são emitidas, os especificadores de registro para operandos pendentes são renomeados com os nomes das estações de reserva que fornecem os valores atualizados.

Como pode haver mais estações de reserva do que registradores reais, esta abordagem pode até eliminar riscos decorrentes de dependências de nomes que não poderiam ser atendidas pelo compilador.

Algoritmo de Tomasulo - 5

A utilização de estações de reservas, em vez de um banco de registo centralizado, leva a duas propriedades importantes

- *a detecção e o controle de perigos são distribuídos* – as informações mantidas nas estações de reserva em cada unidade funcional determinam quando uma instrução pode começar a ser executada naquela unidade
- *os resultados são passados diretamente para as unidades funcionais a partir das estações de reserva onde são armazenados*. Isto é, em vez de passarem pelos registradores do banco de registradores – esse desvio é feito usando um barramento de resultados comum que permite que todas as unidades que aguardam um operando sejam carregadas simultaneamente (o barramento é chamado de *barramento de dados comum* no IBM 360/91); em pipelines com múltiplas unidades de execução e emitindo múltiplas instruções por ciclo de clock, é necessário mais de um barramento de resultados.

Algoritmo de Tomasulo - 6

Organização básica de uma unidade de ponto flutuante MIPS utilizando o algoritmo de Tomasulo

Fonte: Arquitetura de Computadores: Uma Abordagem Quantitativa

DETI

Algoritmo de Tomasulo - 7

Cada estação de reserva contém uma instrução que foi emitida e está aguardando execução na unidade funcional associada, e os valores dos operandos para essa instrução, se estiverem disponíveis, ou os nomes das estações de reserva que fornecerão os valores dos operandos uma vez. eles são computados.

Os buffers de carregamento e armazenamento armazenam endereços e dados transferidos de ou para a memória e se comportam quase exatamente como as estações de reserva, portanto, serão diferenciados apenas quando necessário. Em particular, os buffers de carregamento e armazenamento têm três funções

- eles contêm o endereço de origem ou destino completo após seu cálculo pelo *unidade de endereço*
- eles rastreiam cargas pendentes que aguardam a conclusão da transferência
- controlam os resultados de cargas completas que aguardam a disponibilidade do CDB, ou armazenam o valor a ser armazenado até que a *unidade de memória* esteja disponível.

Todos os resultados das unidades funcionais e de memória são colocados no barramento de dados comum, que conecta tudo, exceto os buffers de carga. Todas as estações de reserva e buffers de carga e armazenamento possuem campos de tags empregados pelo controle do pipeline.

Algoritmo de Tomasulo - 8

Cada instrução passa por três etapas de processamento, embora cada uma possa levar um número arbitrário de ciclos de clock. As três etapas, que substituem ID, EX e WB no pipeline padrão, são as seguintes

1. *Emissão* – a próxima instrução é obtida do chefe da fila de instruções, que é mantida na ordem FIFO para garantir o fluxo de dados correto. Se uma estação ou buffer de reserva correspondente estiver livre, a instrução é emitida para a estação ou buffer junto com os valores do operando. Os ciclos escondidos atualmente são salvos nos registradores do banco de registradores. Caso contrário, é realizado o rastreamento das unidades funcionais que produzirão os valores dos operandos. É aqui que os registos são renomeados, eliminando, consequentemente, os perigos de WAR e WAW.
Este estágio às vezes é chamado *de despacho*, no contexto de processadores agendados dinamicamente.

Algoritmo de Tomasulo - 9

2. Executar – Se um ou ambos os operandos estiverem indisponíveis, o barramento de dados comum é monitorado aguardando que ele ou eles sejam computados. Assim que um operando fica disponível, ele é armazenado no banco de registradores e em qualquer estação ou buffer de armazenamento que o requeira. Quando todos os operandos estiverem disponíveis, a operação poderá ser executada na unidade funcional ou de memória correspondente. Ao atrasar a execução da instrução até que todos os operandos estejam disponíveis, os riscos RAW são evitados.

Várias instruções podem ficar prontas no mesmo ciclo de clock. Embora unidades funcionais independentes possam iniciar a execução no mesmo ciclo de clock, se mais de uma instrução estiver pronta para ser executada na mesma unidade funcional, uma seleção deverá ser feita entre elas. As estações de reserva de ponto flutuante podem ser escolhidas arbitrariamente.

As instruções de carregamento e armazenamento, entretanto, são mantidas na ordem do programa através do uso de uma fila de carregamento/armazenamento que contém a identificação do buffer que está sendo usado. A sua execução requer um processo de duas etapas: na etapa 1, o endereço efetivo é calculado; no passo 2 é realizado o acesso à memória para transferência de dados.

Algoritmo de Tomasulo - 10

Para preservar o comportamento de exceção, nenhuma instrução pode iniciar a execução até que todas as ramificações que a precedem na ordem do programa sejam concluídas. Esta restrição garante que uma instrução que cause uma exceção durante a execução terá sido realmente executada. Em um processador que usa previsão de desvio, como fazer todos os processadores escalonados dinamicamente? Isto significa que deve-se saber que a previsão de desvio está correta, antes de permitir que uma instrução após o desvio na ordem do programa inicie a execução. Se o processador registrar a ocorrência de uma exceção, mas na verdade não a acionar, uma instrução pode iniciar a execução, mas não parar até atingir a etapa de gravação do resultado.

3. *Resultado da escrita* – Quando o resultado da operação estiver disponível, o CDB é acessado para salvá-lo em um cadastro do banco de registradores e nas estações de reserva, inclusive buffers de armazenamento, que o aguardam.

Algoritmo de Tomasulo - 11

As estruturas de dados que detectam e eliminam perigos estão anexadas às estações de reserva, ao banco de registros e aos buffers de carga e armazenamento, com informações ligeiramente diferentes anexadas a cada tipo. Essas tags são essencialmente nomes para o conjunto estendido de registros virtuais usados no processo de renomeação.

Na ilustração mostrada, o campo tag é uma quantidade de 4 bits que denota uma das cinco estações de reserva (três somadores FP para adição e subtração e dois multiplicadores FP para multiplicação e divisão) ou um dos cinco buffers de carga. Isto produz o equivalente a dez registradores que podem ser designados como *registros de resultado*.

O campo tag descreve qual estação de reserva, ou buffer de carga, contém a instrução que produzirá um resultado necessário como operando de origem. Uma vez emitida uma instrução e aguardando um operando fonte, ela denota o operando através do número da estação reserva, ou número do buffer de carga, onde foi atribuída a instrução que irá escrever o registrador. Valores não utilizados, como zero, indicam que o operando já está disponível em um registro do banco de registros. O fato de haver mais estações de reserva, ou buffers de carga, do que números de registro reais, torna trivial a eliminação dos perigos de WAR e WAW.

Algoritmo de Tomasulo - 12

No esquema de Tomasulo, os resultados são transmitidos em um barramento que é monitorado pelas estações de reserva e pelos buffers de armazenamento para recuperação de dados. Esse tipo de arranjo implementa os mecanismos de encaminhamento e desvio usados em um pipeline programado estaticamente. Ao fazer isso, entretanto, um esquema escalonado dinamicamente adiciona um ciclo de clock de latência entre a origem e o resultado, uma vez que a correspondência de um resultado e seu uso não pode ser feitos até o estágio de gravação do resultado. Assim, em um pipeline programado dinamicamente, a latência efetiva entre uma instrução de produção e uma instrução de consumo é pelo menos um ciclo de clock maior que a latência da unidade funcional que produz o resultado.

É importante lembrar que as tags do algoritmo Tomasulo referem-se à unidade ou buffer que produz o resultado: os nomes dos registradores são descartados assim que uma instrução é emitida. Esta é, de facto, uma diferença fundamental entre o esquema de Tomasulo e o placar. No scoreboard, os operandos permanecem nos registradores do banco de registradores e só são lidos após a conclusão das instruções produtoras e a instrução consumidora estar pronta para execução.

Algoritmo de Tomasulo - 13

Cada estação de reserva ou buffer possui sete campos

- ocupado – indica se a estação reserva ou buffer está ocupado ou livre
- op – indica a operação a ser realizada na unidade
- Qj , Qk – identificação da estação reserva, ou buffer, que produzirá o operando fonte correspondente; um valor zero indica que o operando fonte já está disponível nos registradores Vj e Vk , ou é desnecessário
- Vj , Vk – valor dos operandos. Note que observe que apenas um dos campos, Q ou V, é válido para cada operando; para buffers de carga, o campo Vk é usado para manter o campo de deslocamento
- A – usado para armazenar o endereço de memória efetivo para uma instrução de carregamento ou armazenamento.

Cada registro do banco de registros possui um

- campo
- Qi – identificação da estação de reserva, ou buffer, que produzirá o resultado a ser armazenado no registro; um valor zero indica que nenhuma instrução ativa está computando um valor a ser armazenado no registrador.

Algoritmo de Tomasulo - 14

Considere a sequência de código

```
LD      F6,32(R2)
LD      F2,44(R3)
MUL   D F0,. .?,F4
SUB   D F8,F0,F6
DIV   D F10,F0,F0
ADD.CVR.D .,.,.,2
```

Qual é o estado de execução quando apenas a primeira instrução de carregamento foi concluída e escreveu seu resultado?

Suponha as seguintes latências: carregar/armazenar – 1 ciclo de clock, adição – 2 ciclos de clock ciclos, multiplicação – 6 ciclos de clock e divisão – 12 ciclos de clock.

Algoritmo de Tomasulo - 15

Estações/buffers de reserva e tags de registro quando apenas a primeira instrução de carregamento foi concluída e gravou seu resultado

Status da instrução (não faz parte do hardware)			
Instrução	Emitir	Executar	Escrever resultado
LD F6,32(R2)	sim	sim	sim
LD F2,44(R3)	sim	sim	
MUL.D F0,F2,F4	sim		
SUB.D F8,F2,F6	sim		
DIV.D F10,F0,F6	sim		
ADICIONAR.D F6,F8,F2	sim		

Estações/buffers de reserva							
Nome	ocupado	operação	Vj	Vc	Qj	Qk	A
carregar1	não						
carregar2	sim	carregar					44+reg[R3]
adicionar1	sim	sub		mem[32+reg[R2]]	carregar2		
adicionar2	sim	adicionar			adicionar1	carregar2	
adicionar3	não						
mult1	sim	mult		reg[F4]	carregar2		
mult2	sim	divisão		mem[32+reg[R2]]	mult1		

Status do registro									
Campo	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	mult1	carregar2		adicionar2	adicionar1	mult2			

Algoritmo de Tomasulo - 16

Estações/buffers de reserva e tags de registro quando a instrução de multiplicação estiver pronta para escrever seu resultado

Status da instrução (não faz parte do hardware)			
Instrução	Emitir	Executar	Escrever resultado
LD F6,32(R2)	sim	sim	sim
LD F2,44(R3)	sim	sim	sim
MUL.D F0,F2,F4	sim	sim	
SUB.D F8,F2,F6	sim	sim	sim
DIV.D F10,F0,F6	sim		
ADICIONAR.D F6,F8,F2	sim	sim	sim

Estações/buffers de reserva							
Nome	ocupado	operação	V _j	V _c	Q _j	Q _k	A
carregar1	não						
carregar2	não						
adicionar1	não						
adicionar2	não						
adicionar3	não						
mult1	sim	mult	mem[44+reg[R3]]	registro[F4]			
mult2	sim	divisão		memória[32+reg[R2]]	mult1		

Status do registro									
Campo	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	mult1					mult2			

Algoritmo de Tomasulo - 17

Verificações e ações contábeis necessárias para cada etapa da execução da instrução quando o algoritmo de Tomasulo é usado

Status da instrução	Espere até	Escrituração contábil
Emitir Operação PF	(RS[r].ocupado == não)	<pre> if (regStat[rs].Qi != 0) RS[r].Qj = regStat[rs].Qi; senão { RS[r].Vj = reg[rs]; RS[r].Qj = 0; } if (regStat[rt].Qi != = 0) RS[r].Qk = regStat[rt].Qi; senão { RS[r].Vk = reg[rt]; RS[r].Qk = 0; } regStat[rd].Qi = r; RS[r].ocupado = sim; </pre>
Carregamento de problemas	(RS[r].ocupado == não)	<pre> if (regStat[rs].Qi != 0) RS[r].Qj = regStat[rs].Qi; senão { RS[r].Vj = reg[rs]; RS[r].Qj = 0; } RS[r].A = im; regStat[rt].Qi = r; insira r na fila de armazenamento de carga; RS[r].ocupado = sim; </pre>
Emitir loja	(RS[r].ocupado == não)	<pre> if (regStat[rs].Qi != 0) RS[r].Qj = regStat[rs].Qi; senão { RS[r].Vj = reg[rs]; RS[r].Qj = 0; } RS[r].A = im; if (regStat[rt].Qi != 0) RS[r].Qk = regStat[rt].Qi; senão { RS[r].Vk = reg[rt]; RS[r].Qk = 0; } insira r na fila de armazenamento de carga; RS[r].ocupado = sim; </pre>

Algoritmo de Tomasulo - 18

Verificações e ações contábeis necessárias para cada etapa da execução da instrução quando o algoritmo de Tomasulo é usado (continuação)

Execução Operação PF	$(RS[r].Qj == 0) \&\& (RS[r].Qk == 0)$	resultado do cálculo – os operandos estão em Vj e Vk
Carga de execução/armazenamento etapa 1	$(RS[r].Qj == 0) \&\& r \text{ é o chefe da fila de armazenamento do cargo}$	$RS[r].A = RS[r].Vj + RS[r].A;$
Carga de execução etapa 2	atualiza o bloco de carregamento com o resultado	ler de memória $S[r].A$
Escrever resultado Operação ou carga FP	r concluiu a execução && CDB está disponível	$\begin{aligned} &yx (\text{if } (regStat[x].Qi == r) \{ reg[x] = \\ &r \text{ resultado; regStat}[x].Qi = 0; \}) \end{aligned}$ $\begin{aligned} &yx (\text{if } (RS[x].Qj == \\ &(RS[x].Qk == r) \{ RS[x].Vj = \text{resultado; RS}[x].Qj = 0; \}) \end{aligned}$ $\begin{aligned} &RS[x].Vk = \text{resultado; RS}[x].Qk = 0; \}) \\ &RS[r].ocupado = \text{não; } \end{aligned}$
Escrever resultado loja	r concluiu a execução && $(RS[r].Qk == 0)$	$mem[RS[r].A] = RS[r].Vk;$ $RS[r].ocupado = \text{não; }$

onde $RS[r]$ é a estação/buffer de reserva associada à instrução e $regStat$ é o status do registrador.

Algoritmo de Tomasulo - 19

Um loop é considerado a seguir para ilustrar todo o poder de eliminar WAW e Riscos de WAR através da renomeação de registros (os efeitos de ramificações atrasadas são ignorados)

Loop: LD MUL.D F0,0(R1)
 F4,F0,F2
 SD F4,0(R1)
 DADDIU R1,R1,-8
 BNE ,1,R2 Ciclo

Supõe-se que o registrador R1 contém inicialmente o endereço do último elemento do array e que o conteúdo do registrador R2+8 é o endereço do primeiro elemento do array.

Se a ramificação for prevista, múltiplas execuções do loop podem prosseguir em paralelo através do uso de múltiplas estações de reserva e buffers de carga e armazenamento.

Este recurso é obtido sem qualquer alteração de código, uma vez que o loop é desenrolado dinamicamente pelo hardware. As estações de reserva e os buffers de carga e armazenamento desempenham o papel de registros adicionais.

Suponha as mesmas latências de instrução do último exemplo.

Algoritmo de Tomasulo - 20

Estações/buffers de reserva e tags de registro quando duas iterações de loop sucessivas foram emitidas, mas nenhuma instrução foi concluída ainda

Status da instrução (não faz parte do hardware)				
Instrução	Iteração de Loop	Emitir	Executar	Escrever resultado
LDF0,0(R1)	eu	sim	sim	
MUL.D F4,F0,F2	eu	sim		
DP F4,0(R1)	eu	sim		
LDF0,0(R1)	eu+1	sim	sim	
MUL.D F4,F0,F2	eu+1	sim		
DP F4,0(R1)	eu+1	sim		

Estações de reserva							
Nome	ocupado	operação	Vj	Vc	Qj	Qk	A
carregar1	sim	carregar					registro[R1]+0
carregar2	sim	carregar					registro[R1]-8
mult1	sim	mult		registro[F2]	carregar1		
mult2	sim	mult		registro[F2]	carregar2		
loja1	sim	loja	registro[R1]+0			mult1	
loja2	sim	loja	registro[R1]-8			mult2	

Status do registro									
Campo	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	carregar2		mult2						

Algoritmo de Tomasulo - 21

Quando estendida para questões de múltiplas instruções, a abordagem de Tomasulo pode sustentar mais de uma instrução por ciclo de clock. Cargas e armazenamentos podem ser executados fora de ordem com segurança, desde que acessem diferentes endereços de memória. Quando se referem ao mesmo endereço, então

- a carga está *antes* do armazenamento na ordem do programa e a troca de sua execução resulta em risco de WAR, ou
- a carga está *após* o armazenamento na ordem do programa e a troca de sua execução resulta em um risco de RAW.

Da mesma forma, trocando a execução de dois armazenamentos para o mesmo endereço de memória resulta em um perigo WAW.

Portanto, para determinar se uma instrução de carregamento pode ser executada, o processador deve verificar se existe alguma instrução de armazenamento incompleta precedendo o carregamento na ordem do programa e acessando o mesmo endereço. Da mesma forma, uma instrução de armazenamento deve esperar até que não haja cargas e armazenamentos inesperados que a precedam e acessem o mesmo endereço.

Algoritmo de Tomasulo - 22

Para evitar a ocorrência de riscos de dados, o processador deve ter calculado o endereço efetivo associado a qualquer operação de memória anterior ainda em andamento.

Isto pode ser feito de uma maneira simples, mas não necessariamente ideal, realizando todos os cálculos de endereços efetivos na ordem do programa. (Na verdade, basta manter a ordem relativa entre os armazenamentos e outras referências de memória, ou seja, as operações de carregamento podem ser reordenadas livremente).

Se uma operação de carregamento ocorrer primeiro, a existência de um conflito de endereço poderá ser facilmente verificada examinando o campo A de todos os buffers de armazenamento ativos. Sendo detectado um conflito, a instrução de carregamento não é emitida até que o armazenamento conflitante seja concluído.

No caso de instruções de armazenamento, o procedimento é semelhante, exceto que o processador deve verificar se há conflitos nos buffers de carga e de armazenamento, uma vez que os armazenamentos conflitantes não podem ser reordenados em relação a cargas e armazenamentos.

Algoritmo de Tomasulo – 23

O esquema de Tomasulo não foi usado por muitos anos após sua aplicação à unidade de ponto flutuante IBM 360/91, mas se tornou muito popular para processadores de múltiplos problemas, a partir da década de 1990, por vários motivos.

- embora o algoritmo tenha sido projetado antes da era do cache, a presença de caches com atrasos inherentemente ~~improváveis~~ provocou uma das principais motivações para o escalonamento dinâmico porque a execução *forwarding* permite que o processador continue executando instruções enquanto espera pela conclusão de uma falta de cache, ocultando assim toda a complexidade da ~~lógica~~ de
- à medida que os processadores se tornam mais agressivos em sua capacidade de emissão e os projetistas mais preocupados com o desempenho de código difícil de escalar, como é a maioria dos códigos não numéricos, a aplicação de técnicas como *renomeação de registradores, escalonamento dinâmico e especulação* torna-se mais relevante
- através de sua aplicação, pode-se obter alto desempenho sem exigir que o compilador direcione o código gerado para uma estrutura de pipeline específica.

Especulação baseada em hardware - 1

Manter dependências de controle torna-se um fardo crescente quando se tenta explorar ainda mais o paralelismo no nível de instrução. A previsão de ramificação reduz as paralisações diretas atribuíveis às ramificações, mas apenas prever as ramificações com precisão pode não ser suficiente para gerar a quantidade desejada de paralelismo no nível de instrução para um processador que executa múltiplas instruções por ciclo de clock. Um processador de grande porte pode precisar executar uma ramificação a cada ciclo de clock para manter o desempenho em um valor máximo. Portanto, explorar esse paralelismo exige que a limitação da dependência do controle seja superada.

A superação da dependência de controle é alcançada *especulando* sobre o resultado das ramificações e executando o código como se a estimativa estivesse correta. Essa ideia representa uma extensão sutil, mas crucial, da previsão de desvio com escalonamento dinâmico: com a especulação, as instruções são buscadas, emitidas e executadas como se as previsões de desvio estivessem sempre corretas; o agendamento dinâmico apenas busca e emite tais instruções. É claro que são necessários mecanismos para lidar com a situação em que a especulação se revela incorreta.

Especulação baseada em hardware - 2

A especulação baseada em hardware combina três ideias principais

- *previsão dinâmica de ramificação*, para selecionar quais instruções executar
- *especulação*, para permitir a execução de instruções antes que as dependências de controle sejam resolvidas (com o entendimento de que os efeitos produzidos por uma sequência especulada incorreta podem ser desfeitos)
- *agendamento dinâmico*, para lidar com a emissão de diferentes combinações de blocos básicos.

Em contraste, o escalonamento dinâmico [sem especulação] sobrepõe os blocos básicos apenas parcialmente, porque requer que um desvio seja resolvido antes de realmente executar qualquer instrução no bloco sucessor.

Conseqüentemente, a especulação baseada em hardware segue o fluxo previsto de valores de dados para determinar quando executar as instruções. Este método de execução é essencialmente uma *execução de fluxo de dados*: as operações são realizadas assim que seus operandos de origem ficam disponíveis.

Especulação baseada em hardware - 3

Para estender o algoritmo de Tomasulo para suportar especulação, o desvio de resultados entre instruções, que é necessário para executar instruções especulativamente, deve ser separado da conclusão real das instruções. Se tal mecanismo for realizado, uma instrução pode ser executada e ignorar seu resultado para outras pessoas, sem permitir que a instrução execute quaisquer atualizações que não possam ser revertidas até que seja estabelecido que a instrução não é mais especulativa.

Usar um valor ignorado é como realizar uma leitura especulativa do registrador, pois pode não se saber naquele momento se a instrução que fornece o valor para o registrador [fonte] está fornecendo um resultado *real*. Somente quando a instrução não for mais especulativa, o registrador específico do banco de registradores, ou a referida localização de memória, poderá ser atualizado – esta etapa adicional na execução da instrução é chamada de *commit de instrução*.

Portanto, as instruções podem ser executadas *fora de ordem*, mas são forçadas a cometer *em ordem* para que qualquer ação irreversível, como atualização de estado ou tomada de exceção, seja evitada. A separação entre a conclusão da instrução e a confirmação da instrução é essencial porque as instruções podem terminar a execução consideravelmente antes de estarem prontas para serem confirmadas.

Especulação baseada em hardware - 4

A introdução da fase de commit na execução de instruções requer um conjunto adicional de buffers de hardware que armazenam os resultados de instruções completadas que ainda não foram confirmadas. Esse banco de buffers, chamado *buffer de reordenação* (ROB), também é usado para passar resultados entre instruções.

O *buffer de reordenação* fornece registros adicionais da mesma maneira que as estações de reserva e os buffers de carga e armazenamento estendem o registro definido no algoritmo de Tomasulo. A principal diferença é que no algoritmo de Tomasulo, uma vez que uma instrução escreve seu resultado, quaisquer instruções emitidas posteriormente obterão o valor do banco de registros, enquanto com especulação, o banco de registros não é atualizado até que a instrução seja confirmada (ou seja, quando é estabelecido definitivamente que a instrução deve ser executada), o que significa que o ROB fornece operandos para outras instruções no intervalo entre a conclusão da instrução e o commit da instrução. O ROB é semelhante aos buffers de armazenamento no algoritmo de Tomasulo, portanto, a funcionalidade dos buffers de armazenamento é integrada ao ROB para simplificar.

Especulação baseada em hardware - 5

Cada entrada ROB contém cinco campos

- *ocupado* – indica se a entrada está ocupada ou livre
- *tipo de instrução* – sinaliza se a instrução é um desvio, sem resultado de destino, uma loja, com destino na memória, ou uma carga ou Operação ALU, com destino de banco de registradores
- *localização de destino* - fornece o endereço de memória, para armazenamentos, ou o número do registro, para cargas e operações da ALU, onde o resultado da instrução deve ser escrito
- *valor* – contém o valor do resultado da instrução entre instruções confirmação de conclusão e instrução
- *ready* - indica se a instrução completou a execução.

Especulação baseada em hardware - 6

**Organização básica de uma unidade de ponto flutuante MIPS utilizando o algoritmo de
Tomasulo estendido para lidar com especulação**

Fonte: Arquitetura de Computadores: Uma Abordagem Quantitativa

DETI

Especulação baseada em hardware - 7

As lojas ainda são executadas em duas etapas, mas a segunda etapa está incluída na instrução commit.

Embora a função de renomeação das estações de reserva seja substituída pelo ROB, ainda é necessário um local para armazenar operações e operandos entre o momento em que são emitidos e o momento em que terminam a execução. Esta função é realizada pelas estações de reserva e pelos buffers de carga. Por outro lado, como cada instrução tem uma entrada no ROB até ser confirmada, o resultado da operação é marcado usando a entrada ROB em vez da estação de reserva ou do número do buffer de carga. Essa marcação requer que a entrada ROB atribuída à instrução seja rastreada na estação de reserva ou no buffer de carga. Posteriormente, uma implementação alternativa, que utiliza registradores extras para renomeação e uma fila para substituir ROB, é descrita e mostrada como ela pode decidir quando as instruções podem ser confirmadas.

Especulação baseada em hardware - 8

As quatro etapas envolvidas na execução da instrução são as seguintes

1. *Emissão* – a próxima instrução é obtida do chefe da fila de instruções, que é mantida na ordem FIFO para garantir o fluxo de dados correto. Se uma estação ou buffer de reserva correspondente e um slot ROB estiverem livres, a instrução será emitida para a estação ou buffer junto com os valores do operando, se eles estiverem atualmente salvos nos registradores do banco de registadores ou nas entradas ROB.
Caso contrário, será realizada a remoção das entradas ROB que eventualmente conterão os valores do operando. O número da entrada ROB alocada para a instrução também é enviado para a estação de reserva ou buffer para que o número possa ser usado para marcar o resultado quando for colocado no CDB.
2. *Executar* – o CDB é monitorado enquanto aguarda o cálculo dos operandos para evitar riscos de RAW. Quando os operandos estiverem disponíveis, a operação é executada. A execução da instrução pode levar vários ciclos de clock. As cargas ainda requerem duas etapas. As lojas, por outro lado, calculam apenas o endereço efetivo.

Especulação baseada em hardware - 9

3. *Gravação do resultado* – quando a instrução é concluída, o resultado é colocado no CDB, junto com a tag de entrada do ROB, para ser escrito no ROB e em todas as estações de reserva e buffers que aguardam o valor. A estação ou buffer de reserva é marcado como livre e o campo *pronto* da entrada ROB é definido. Ações especiais são necessárias para instruções de armazenamento: se o valor a ser armazenado estiver disponível, ele será escrito no campo *de valor* da entrada ROB; se o valor não está disponível, o CDB é informado para a transmissão do valor, ele é salvo no buffer e então o campo *de valor* da entrada ROB é atualizado.

4. *Commit* – o conjunto de ações associado depende se a instrução de commit é uma ramificação com uma previsão errada, um armazenamento ou qualquer outro caso (*commit normal*). O *commit normal* ocorre quando a instrução atinge o início do ROB e o campo *pronto* é definido, o processador então atualiza o registro do banco de registros, se houver, e finaliza. A confirmação de um armazenamento é semelhante, exceto que um local de memória é atualizado em vez de um registro. Finalmente, para uma ramificação com uma previsão errada, a especulação é considerada incorreta, o ROB é liberado e a busca é reiniciada no endereço adequado definido pelo comportamento da ramificação.

Especulação baseada em hardware - 10

Considere a sequência de código (idêntica àquela usada no exemplo para o Algoritmo de Tomasulo e assume as mesmas latências de instrução)

LD	F6,32(R2)
LD	F2,44(R3)
MUL.D	F0,F2,F4
SD	F8,F2,F6
DIV.D	F10,F0,F6
ADICIONAR.D	F6,F8,F2

Qual é o estado de execução quando a instrução de multiplicação está pronta para ser confirmada, ou seja, a instrução MULT está no início do ROB?

Especulação baseada em hardware - 11

Reordene buffer, estações/buffers de reserva e tags de registro quando a instrução de multiplicação estiver pronta para confirmação

<i>Reordenar buffer</i>						
<i>entrada</i>	<i>estado</i>	<i>ocupado</i>	<i>instrução tipo</i>	<i>destino localização</i>	<i>valor</i>	<i>preparar</i>
1	comprometer-se	sem LD F6,32(R2)		F6	mem[reg[R2]+32]	sim
2	comprometer-se	sem LD F2,44(R3)		F2	mem[reg[R3]+44] #2 x	sim
3	escrever resultado	sim MUL.D F0,F2,F4 sim SUB.D		F0	reg[F4]	sim
4	escrever resultado	F8,F2,F6 sim DIV.D F10,F0,F6 sim		F8	#2 - #1	sim
5	executar	ADD.D F6,F8,F2		F10		não
6	escrever resultado			F6	#4 + #2	sim

<i>Estações/buffers de reserva</i>							
<i>Nome</i>	<i>ocupado</i>	<i>operação</i>	<i>Vj</i>	<i>Vc</i>	<i>Qj</i>	<i>Qk</i>	<i>destino A</i>
carregar1	não						
carregar2	não						
adicionar1	não						
adicionar2	não						
adicionar3	não						
mult1	não	mult	memória[reg[R3]+44]	reg[F4]			#3
mult2	sim	divisão		mem[reg[R2]+32]	#3		#5

<i>Status do registro</i>									
<i>Campo</i>	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
ocupado	sim	não	não	sim	sim	sim	não	...	não
Entrada ROB	3			6	4	5		...	

Especulação baseada em hardware - 12

Considere a seguinte sequência de código, idêntica àquela usada no exemplo para o algoritmo de Tomasulo (os efeitos das ramificações atrasadas são ignorados)

```
Loop: LD MUL.D      F0,0(R1)
          F4,F0,F2
          SD      F4,0(R1)
          DADDIU R1,R1,-8
          BNE    R1,R2, Ciclo
```

Supõe-se que o registrador R1 contém inicialmente o endereço do último elemento do array e que o conteúdo do registrador R2+8 é o endereço do primeiro elemento do array.

Se a ramificação for prevista, múltiplas execuções do loop poderão prosseguir em paralelo. Suponha que todas as instruções no loop foram emitidas duas vezes, as instruções de carga e multiplicação da primeira iteração foram confirmadas e todas as outras concluíram a execução.

Suponha também as mesmas latências de instrução do último exemplo.

Especulação baseada em hardware - 13

Reordene tags de buffer e registro quando duas iterações de loop sucessivas forem emitidas, as instruções de carregamento e multiplicação da primeira iteração forem confirmadas e todas as outras tiverem concluído a execução

<i>Reordenar buffer</i>						
<i>entrada</i>	<i>estado</i>	<i>ocupado</i>	<i>instrução tipo</i>	<i>destino localização</i>	<i>valor</i>	<i>preparar</i>
1	comprometer-se	sem LD	F0,0(R1)	F0	memória[reg[R1]+0]	sim
2	comprometer-se	sem MUL.D F4,F0,F2		F4	#1 x registro[F2]	sim
3	escrever resultado	sim SD sim	F4,0(R1)	registro[R1]+0	#2	sim
4	escrever resultado	DADDIU R1,R1,-8 R1,R1,Loop		R1	registro[R1] – 8	sim
5	escrever resultado	sim, BNE				sim
6	escrever resultado	sim LD sim	F0,0(R1)	F0	mem[#4]	sim
7	escrever resultado	MUL.D F4,F0,F2 sim SD F4,0(R1)		F4	#6 xreg[F2]	sim
8	escrever resultado	sim DADDIU R1,R1,-8 sim BNE		#4+0	#7	sim
9	escrever resultado	R1,R1,Loop		R1	#4 – 8	sim
10	escrever resultado					sim

<i>Status do registro</i>									
<i>Campo</i>	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
ocupado	sim	não	sim	não	não	não	não	...	não
Entrada ROB	6		7					...	

Especulação baseada em hardware - 14

Como nem os valores de registro nem os valores de memória são realmente escritos até que uma instrução seja confirmada, o processador pode facilmente desfazer suas ações especulativas quando um desvio for considerado mal previsto. As instruções anteriores à ramificação serão confirmadas sucessivamente quando cada uma chegar ao chefe do ROB. Quando chega a vez da ramificação, o ROB é liberado e o processador começa a buscar instruções no caminho correto.

Em processadores especulativos, o desempenho é muito sensível à previsão de ramificação. Assim, todos os aspectos do tratamento de ramificações, como a precisão da previsão, a latência da detecção de previsões erradas e o tempo de recuperação de previsões erradas, tornam-se muito importantes.

As exceções são tratadas não reconhecendo a exceção até que a instrução que a produziu esteja pronta para ser confirmada. Se uma instrução especulada gerar uma exceção, a exceção será registrada na entrada ROB associada. Quando surge uma previsão incorreta de desvio e a instrução não deveria ter sido executada, a exceção é liberada junto com o ROB. Se a instrução chegar ao topo do ROB, ela não será mais especulativa e a exceção será aberta.

Especulação baseada em hardware - 15

Verificações necessárias e ações contábeis para cada etapa na execução de instruções quando a especulação baseada em hardware é usada

Status da instrução	Espere até	Escrituração contábil
Emitir Operação PF	(RS[r].ocupado == não) && (ROB[b].ocupado == não)	<pre> if (regStat[rs].busy == sim) { x = regStat[rs].reorder; if (ROB[x].pronto == sim) { RS[r].Vj = ROB[x].valor; RS[r].Qj = 0; } senão RS[r].Qj = x; } else { RS[r].Vj = reg[rs]; RS[r].Qj = 0; } if (regStat[rt].busy == sim) { x = regStat[rt].reorder; if (ROB[x].pronto == sim) { RS[r].Vj = ROB[x].valor; RS[r].Qk = 0; } senão RS[r].Qk = x; } else { RS[r].Vj = reg[rt]; RS[r].Qk = 0; } regStat[rd].reordenar = b; regStat[rd].busy = sim; RS[r].dest = b; RS[r].ocupado = sim; ROB[b].inst = código de operação; ROB[b].dest = rd; ROB[b].pronto = não; ROB[b].ocupado = sim; </pre>
Carregamento de problemas	(RS[r].ocupado == não) && (ROB[b].ocupado == não)	<pre> if (regStat[rs].busy == sim) { x = regStat[rs].reorder; if (ROB[x].pronto == sim) { RS[r].Vj = ROB[x].valor; RS[r].Qj = 0; } senão RS[r].Qj = x; } else { RS[r].Vj = reg[rs]; RS[r].Qj = 0; } regStat[rt].reordenar = b; regStat[rt].busy = sim; RS[r].A = im; RS[r].dest = b; RS[r].ocupado = sim; ROB[b].inst = código de operação; ROB[b].dest = rt; ROB[b].pronto = não; ROB[b].ocupado = sim; </pre>

Especulação baseada em hardware - 16

Verificações e ações contábeis necessárias para cada etapa na execução da instrução quando a especulação baseada em hardware é usada (continuação)

Emitir loja	(RS[r].ocupado == não) && (ROB[b].ocupado == não)	<pre> if (regStat[rs].busy == sim) { x = regStat[rs].reorder; if (ROB[x].pronto == sim) { RS[r].Vj = ROB[x].valor; RS[r].Qj = 0; } senão RS[r].Qj = x; } else { RS[r].Vj = reg[rs]; RS[r].Qj = 0; } if (regStat[rt].busy == sim) { x = regStat[rt].reorder; if (ROB[x].pronto == sim) { RS[r].Vk = ROB[x].valor; RS[r].Qk = 0; } senão RS[r].Qk = x; } else { RS[r].Vk = reg[rt]; RS[r].Qk = 0; } RS[r].A = im; RS[r].ocupado = sim; ROB[b].inst = código de operação; ROB[b].pronto = não; ROB[b].ocupado = sim; </pre>
Execução Operação PF	(RS[r].Qj == 0) && (RS[r].Qk == 0)	resultado do cálculo – os operandos estão em Vj e Vk
Etapa de carregamento de execução 1	(RS[r].Qj == 0) && não há lojas anteriores na fila ROB	RS[r].A = RS[r].Vj + RS[r].A;
Carga de execução etapa 2	a etapa 1 de carregamento foi concluída && todas as lojas anteriores na fila ROB têm endereços efetivos diferentes	ler de mem[RS[r].A]
Execução loja	(RS[r].Qj == 0) && b é o chefe da fila ROB	ROB[b].dest = RS[r].Vj + RS[r].A;

Especulação baseada em hardware - 17

Verificações e ações contábeis necessárias para cada etapa na execução da instrução quando a especulação baseada em hardware é usada (continuação)

Escrever resultado Operação ou carga FP	r concluiu a execução && CDB está disponível	<pre> b = RS[r].dest; ýx (se (RS[x].Qj == b) { RS[x].Vj = resultado; RS[x].Qj = 0; }) ýx (if (RS[x].Qk == b) { RS[x].Vk = resultado; RS[x].Qk = 0; }) ROB[b].valor = resultado; ROB[b].pronto = sim; RS[r].ocupado = não; </pre>
Escrever resultado loja	r concluiu a execução && (RS[r].Qk == 0)	<pre> ROB[b].valor = RS[r].VK; RS[r].ocupado = não; if (ROB[b].inst == filial) { if (ramo previsto incorretamente) { liberar ROB; redefinir estações de reserva/buffers de carga; redefinir o status do registro; buscar no destino da filial; } } else if (ROB[b].inst == armazenar) mem[ROB[b].dest] = [ROB[b].value; senão { reg[ROB[b].dest] = ROB[b].valor; if (regStat[ROB[b].dest].reorder == b) regStat[ROB[b].busy = não; } ROB[b].ocupado = não; </pre>
Comprometer-se	b é o chefe da fila ROB && (ROB[b].ready == yes)	

onde ROB[b] é a entrada ROB e RS[r] a estação/buffer de reserva associada à instrução e regStat é o status do registrador.

Explorando a PLI usando vários problemas - 1

As técnicas que acabamos de descrever podem ser usadas para eliminar paralisações resultantes de dependências de dados e controle e aproximar-se de um CPI ideal de um ao executar um determinado programa.

$$\text{IPC}_{\text{programa}} = \text{IPCideal} + \text{barracas estruturais.}$$

Para melhorar ainda mais o desempenho, é necessário diminuir o CPI ideal para um valor menor que um, mas isso não pode ser feito se apenas uma instrução for emitida por ciclo de clock.

Processadores de múltiplos problemas são de três tipos básicos

- *processadores superescalares estaticamente programados* – um número variável de instruções são emitidas juntas em múltiplos pipelines, cada um programado estaticamente •

Processadores VLIW (palavra de instrução muito longa) – um número fixo de instruções formatadas como uma instrução grande ou um pacote de instruções fixas, são emitido em conjunto

- *processadores superescalares escalonados dinamicamente* – um número variável de instruções é emitido junto em múltiplos pipelines, cada um escalonado dinamicamente.

Explorando a PLI usando vários problemas - 2

Abordagens primárias em uso para processadores de múltiplos problemas

Fonte: Arquitetura de Computadores: Uma Abordagem Quantitativa

Comum nome	Emitir estrutura	Detecção de perigo	Agendamento	Característica distintiva	Exemplos
superescalar (estático)	dinâmico	hardware	estático	execução em ordem	principalmente em ambientes incorporados: MIPS e ARM (incluindo ARM Cortex-A8)
superescalar (dinâmico)	dinâmico	hardware	dinâmico	alguma execução fora de ordem, mas nenhuma especulação	nenhum no momento
superescalar (especulativo)	dinâmico	hardware	dinâmico com especulação	execução fora de ordem com especulação	Intel Core i3, i5, i7, AMD Phenom e IBM Power 7
VLIW / LIW	estático	principalmente software	estático	todos os perigos determinados e indicados pelo compilador (muitas vezes implicitamente)	principalmente no processamento de sinal: TI C6x
ÉPICO	principalmente estático	principalmente software	principalmente estático	todos os perigos determinados e indicados pelo compilador (explicitamente)	Itânia

Processador superescalar agendado estaticamente

Um *processador superescalar estaticamente programado* normalmente emite em ordem um número variável de instruções por ciclo de clock até um limite superior que corresponde ao número de pipelines paralelos que são implementados.

A razão pela qual o número de instruções emitidas por ciclo de clock é variável tem a ver principalmente com:

- as múltiplas tubulações são normalmente iguais, o que pode levar a riscos estruturais se qualquer combinação de instruções for considerada
- embora o encaminhamento seja usado de forma extensiva nos diferentes pipelines, não é possível, mesmo com a ajuda do compilador, evitar travamentos devido a perigos de dados entre instruções sucessivas.

Assim, há de fato vantagens decrescentes para uma arquitetura superescalar estaticamente programada à medida que a largura de emissão de instruções aumenta. É por isso que a largura do problema normalmente é de apenas dois.

BRAÇO Cortex-A8 - 1

O A8 é um processador superescalar estaticamente agendado de emissão dupla com dinâmica detecção de problemas, que permite emitir uma ou duas instruções por ciclo de clock.



Estrutura básica de pipeline de 13 estágios A8

Fonte: Arquitetura de Computadores: Uma Abordagem Quantitativa

BRAÇO Cortex-A8 - 2

O A8 usa um preditor de ramificação dinâmico com um buffer de destino de ramificação associativo de 512 entradas e um buffer de histórico global de entrada de 4K, que é indexado pelo histórico de ramificação e pelo PC atual. No caso de o buffer de destino de ramificação falhar, a previsão é obtida do buffer de histórico global, que é então usado para calcular o endereço da ramificação.

Além disso, uma pilha de retorno de 8 entradas é mantida para rastrear os endereços de retorno. Um incorreto a previsão resulta em uma penalidade de 13 ciclos conforme o pipeline é liberado.

BRAÇO Cortex-A8 - 3

Até duas instruções por ciclo de clock podem ser emitidas usando um mecanismo de emissão em ordem. Uma estrutura simples de placar é usada para rastrear quando uma instrução pode ser emitida. Um par de instruções dependentes pode ser processado por meio da lógica de emissão, mas serão serializadas no placar, a menos que os caminhos de encaminhamento possam resolver a dependência.



Decodificação de instrução A8 de 5 estágios

Fonte: Arquitetura de Computadores: Uma Abordagem Quantitativa

BRAÇO Cortex-A8 - 4

A instrução 1 ou 2 pode ir para o pipeline de carregamento/armazenamento. O desvio total é suportado entre os dutos para minimizar a paralisação no placar.



Pipeline de execução A8

Fonte: Arquitetura de Computadores: Uma Abordagem Quantitativa

BRAÇO Cortex-A8 - 5

O A8 tem um IPC ideal de 0,5 devido à sua estrutura de dupla emissão. As paralisações do oleoduto podem surgir de três fontes

- *riscos estruturais* – ocorrem quando duas instruções adjacentes selecionadas para emissão exigem simultaneamente o mesmo pipeline funcional; como A8 é escalonado estaticamente, é dever do compilador tentar evitar tais conflitos; quando não podem ser evitados, o A8 pode emitir no máximo uma instrução naquele ciclo de clock
- *perigos de dados* – são conflitos nos buffers no início do pipeline, no placar, e podem travar ambas as instruções (se a primeira instrução não puder ser emitida, a segunda estará sempre travada) ou apenas a segunda de um par; novamente, é dever do compilador tentar evitar tais travamentos sempre que possível
- *riscos de controle* – eles só surgem quando as ramificações são mal previstas.

Além de travamentos devido a perigos, falhas de cache L1 e L2 no pipeline de carga/armazenamento também produzirão travamentos.

BRAÇO Cortex-A8 - 6

**Composição estimada do IPC do ARM A8 ao executar o
Conjunto de benchmark Minnespec**

Fonte: Arquitetura de Computadores: Uma Abordagem Quantitativa

DETI

Processador superescalar agendado dinamicamente - 1

Para simplificar, será assumida uma taxa de emissão de duas instruções por ciclo de clock. Os conceitos-chave não são diferentes daqueles encontrados nos processadores modernos que emitem três ou mais instruções por ciclo de clock.

O algoritmo de Tomasulo se estende para suportar pipeline superescalar especulativo de múltiplos problemas com unidade funcional separadas de número inteiro, carga/armazenamento e ponto flutuante, cada uma das quais pode iniciar uma operação a cada ciclo de clock. Para obter todas as vantagens do escalonamento dinâmico, o pipeline pode emitir qualquer combinação de duas instruções.

Como a interação de instruções inteiras e de ponto flutuante é determinante, o esquema de Tomasulo também é estendido para lidar com unidades funcionais e registradores inteiros e de ponto flutuante.

Processador superescalar agendado dinamicamente - 2

Organização básica de um processador de múltiplos problemas com especulação

Fonte: Arquitetura de Computadores: Uma Abordagem Quantitativa

DETI

Processador superescalar agendado dinamicamente - 3

Emitir múltiplas instruções no mesmo ciclo de clock em um processador escalonado dinamicamente, com ou sem especulação, é uma tarefa muito complexa, pois as instruções podem depender umas das outras. Devido a este fato, as tabelas de controle devem ser atualizadas em paralelo; caso contrário, os valores estarão incorretos ou a dependência poderá ser perdida.

Duas abordagens diferentes foram usadas. A primeira é executar a etapa em uma fração de todo o ciclo do clock para cada instrução. Por exemplo, quando a largura do problema é dois, isso significa que ele é executado na metade do ciclo do clock. Infelizmente, não pode ser este método devido de maneira direta para lidar com quatro instruções! A segunda é construir a lógica necessária para executar simultaneamente duas ou mais instruções, incluindo possíveis dependências entre elas.

Os processadores superescalares modernos que emitem quatro ou mais instruções por ciclo de clock podem incluir ambos: eles canalizam e ampliam a lógica do problema.

Esta etapa problemática é um dos gargalos mais fundamentais no desenvolvimento da dinâmica processadores superescalares normalmente escalonados.

Processador superescalar agendado dinamicamente - 4

Verificações e ações contábeis necessárias para a emissão de um pacote de duas instruções onde inst1 é uma carga FP e inst2 é uma operação FP

Status da instrução	Espere até	Escritação contábil
Carga de emissão (primeira instrução do pacote)	(RS[r1].ocupado == não) && (ROB[b1].ocupado == não)	<pre> if (regStat[rs1].busy == sim) { x = regStat[rs1].reorder; if (ROB[x].pronto == sim) { RS[r1].Vj = ROB[x].valor; RS[r1].Qj = 0; } senão RS[r1].Qj = x; } else { RS[r1].Vj = reg[rs1]; RS[r1].Qj = 0; } regStat[rt1].reordenar = b1; regStat[rt1].ocupado = sim; RS[r1].A = im; RS[r1].dest = b1; RS[r1].ocupado = sim; ROB[b1].inst = carregar; ROB[b1].dest = rt1; ROB[b1].pronto = não; ROB[b1].ocupado = sim; </pre>
Emitir Operação FP (segunda instrução do pacote) o primeiro operando vem da carga	(RS[r2].ocupado == não) && (ROB[b2].ocupado == não)	<pre> RS[r2].Qj = b1; if (regStat[rt2].ocupado == sim) { x = regStat[rt2].reordenar; if (ROB[x].pronto == sim) { RS[r2].Vj = ROB[x].valor; RS[r2].Qk = 0; } senão RS[r2].Qk = x; } else { RS[r2].Vj = reg[rs2]; RS[r2].Qk = 0; } regStat[rd2].reordenar = b2; regStat[rd2].ocupado = sim; RS[r2].dest = b2; RS[r2].ocupado = sim; ROB[b2].inst = operação FP; ROB[b2].dest = rd2; ROB[b2].pronto = não; ROB[b2].ocupado = sim; </pre>

onde ROB[b1] e ROB[b2] são entradas ROB e RS[r1] e RS[r2] as estações/buffers de reserva associados às duas instruções e regStat é o status do registrador.

Processador superescalar agendado dinamicamente - 5

Numa situação real, todas as combinações possíveis de instruções dependentes permitidas para serem emitidas no mesmo ciclo de relojueiro devem ser consideradas. Uma vez que o número de possibilidades aumenta exponencialmente com o quadrado do número de instruções emitidas num ciclo de relógio, isto acaba por ser uma reocupação permanente para uma grande largura de emissão (acima de quatro, por exemplo).

Processador superescalar agendado dinamicamente - 6

A estratégia básica para atualizar a lógica de problemas em um processador superescalar escalonado dinamicamente com até n problemas por ciclo de clock é a seguinte

1. Atribua um buffer de reordenação e uma estação/buffer de reserva para cada instrução que possa ser emitida no próximo pacote. Essa atribuição pode ser feita antes que os tipos de instruções sejam conhecidos pré-alcando as entradas do buffer de reordenação sequencialmente às instruções no pacote e garantindo que haja estações/buffers de reserva suficientes disponíveis dependendo do conteúdo do pacote. Caso não estejam disponíveis estações/buffers de reserva suficientes, o pacote tem de ser quebrado e apenas um subconjunto destas instruções, na ordem do programa original, é emitido. As instruções restantes serão incluídas no próximo pacote.

2. Encontre todas as dependências entre todas as instruções do pacote.

Processador superescalar agendado dinamicamente - 7

3. Se for encontrada uma dependência de uma instrução no pacote com uma anterior no pacote, o número de entrada do buffer de reordenação atribuído deve ser usado para atualizar a tabela de reservas para a instrução dependente; caso contrário, o buffer de reordenação existente e as informações da tabela de reservas deverão ser usados para atualizar a tabela de reservas para a instrução emissora.

No final do pipeline, também é necessário concluir a confirmar várias instruções por ciclo de clock. Os passos aqui são, no entanto, um pouco mais simples do que o problema do problema, uma vez que as instruções que podem realmente ser confirmadas no mesmo ciclo de clock já devem ter tratado e resolvido as dependências.

Processador superescalar agendado dinamicamente - 8

Considere a sequência de código abaixo e analise sua execução em um processador de 2 problemas sem e com especulação (os efeitos de ramificações atrasadas são ignorados)

Ciclo:	LD DADDIU	R2,0(R1)
	R2,R2,1	
	SD	R2,0(R1)
	DADDIU R1,R1,8	
	BNE	R1,R3, Ciclo

Supõe-se que o registrador R1 contém inicialmente o endereço do primeiro elemento do array e que o conteúdo do registrador R3-8 é o endereço do último elemento do array.

Supõe-se também que existam unidades funcionais separadas para cálculo de endereço efetivo, para operações de ALU e para avaliação de condições de ramificação. As três primeiras iterações devem ser analisadas.

Processador superescalar agendado dinamicamente - 9

Execução de código em um processador de 2 problemas sem especulação

<i>iteração número</i>	<i>instrução</i>	<i>problemas no ciclo do clock número</i>	<i>executa no ciclo do clock número</i>	<i>acesso à memória no ciclo do clock número</i>	<i>escrever CDB no ciclo do clock número</i>	<i>Comente</i>
1	LD R2,0(R1)	1	2	3	4	primeira edição
1	DADDIU R2,R2,1	1	5		6	espere pelo LD
1	SD R2,0(R1)	2	3	7		espere por DADDIU
1	DADDIU R1,R1,8	2	3		4	executar
1	BNE R1, R3, Ciclo	3	7			espere por DADDIU
2	LD R2,0(R1)	4	8	9	10	espere pelo BNE
2	DADDIU R2,R2,1	4	11		12	espere pelo LD
2	SD R2,0(R1)	5	9	13		espere por DADDIU
2	DADDIU R1,R1,8	5	8		9	executar
2	BNE R1, R3, Ciclo	6	13			espere por DADDIU
3	LD R2,0(R1)	7	14	15	16	espere pelo BNE
3	DADDIU R2,R2,1	7	17		18	espere pelo LD
3	SD R2,0(R1)	8	15	19		espere por DADDIU
3	DADDIU R1,R1,8	8	14		15	executar
3	BNE R1, R3, Ciclo	9	19			espere por DADDIU

Processador superescalar agendado dinamicamente - 10

Execução de código em um processador de 2 problemas com especulação

iteração número	instrução	problemas no ciclo do clock número	executa no ciclo do clock número	acesso à memória no ciclo do clock número	escrever CDB no ciclo do clock número	comprometer-se no ciclo do clock número	Comente
1	LD R2,0(R1)	1	2	3	4	5	primeira edição
1	DADDIU R2,R2,1	1	5		6	7	espere pelo LD
1	SD R2,0(R1)	2	3			7	espere por DADDIU
1	DADDIU R1,R1,8	2	3		4	8	comprometer em ordem
1	BNE R1, R3, Ciclo	3	7			8	espere por DADDIU
2	LD R2,0(R1)	4	5	6	7	9	sem atraso de execução
2	DADDIU R2,R2,1	4	8		9	10	espere pelo LD
2	SD R2,0(R1)	5	6			10	espere por DADDIU
2	DADDIU R1,R1,8	5	6		7	11	comprometer em ordem
2	BNE R1, R3, Ciclo	6	10			11	espere por DADDIU
3	LD R2,0(R1)	7	8	9	10	12	o mais cedo possível
3	DADDIU R2,R2,1	7	11		12	13	espere pelo LD
3	SD R2,0(R1)	8	9			13	espere por DADDIU
3	DADDIU R1,R1,8	8	9		10	14	executa mais cedo
3	BNE R1, R3, Ciclo	9	13			14	espere por DADDIU

Processador superescalar agendado dinamicamente - 11

O exemplo mostra como a especulação pode ser vantajosa quando existem ramificações dependentes de dados, o que é outra forma linda a de desempenho. Esta vantagem depende, no entanto, da previsão correta dos ramos. É importante notar que a especulação incorreta prejudica o desempenho e reduz drasticamente a eficiência energética!

Por que é tão?

Intel Core i7-1

O Intel Core i7 usa uma microarquitetura especulativa agressiva e fora de ordem com pipelines razoavelmente profundos tendo como objetivo atingir alto rendimento de instruções combinando múltiplos problemas e altas taxas de clock.

Estrutura de pipeline do Intel Core i7 com interface do sistema de memória

Fonte: Arquitetura de Computadores: Uma Abordagem Quantitativa



Intel Core i7-2

Alguns recursos do pipeline Intel Core i7 são apresentados a seguir

1. O processador usa um buffer alvo de ramificação multinível, localizado no estágio de busca de instruções, para atingir um equilíbrio entre velocidade e precisão de previsão. Há também uma pilha de endereços de retorno para acelerar o retorno da função. Previsões erradas causam uma penalidade de cerca de 15 ciclos de clock. Usando o endereço previsto, a unidade de busca busca 16 bytes do cache de instruções.
2. Esses 16 bytes são colocados no buffer de instruções de pré-decodificação, onde um procedimento chamado fusão macro-operatória é executado. A fusão *macro-op* pega combinações de instruções como uma comparação seguida por uma ramificação e gera uma única operação. O estágio de pré-decodificação também divide os 16 bytes em instruções x86 individuais. Instruções x86 individuais, incluindo algumas instruções fundidas, são colocadas na fila de instruções de 18 entradas.

Intel Core i7-3

3. Instruções x86 individuais são traduzidas em micro-operações, que são instruções simples do tipo MIPS executadas diretamente pelo pipeline. Esta abordagem foi introduzida no Pentium Pro e tem sido usada desde então. Três dos decodificadores lidam com instruções x86 que se traduzem diretamente em um micro-op. Para instruções x86 que possuem semântica mais complexa, existe um mecanismo de microcódigo que produz a sequência de micro-ops correspondentes. Pode gerar até quatro micro-ops por ciclo de clock e continua até que toda a sequência seja produzida. As micro-operações são colocadas no buffer micro-op de 28 entradas de acordo com a ordem das instruções x86.
4. O buffer micro-op realiza *detecção de fluxo de loop* e *microfusão*. Se houver uma pequena sequência de instruções, com menos de 28 ou 256 bytes de comprimento que compreende um loop, o detector de fluxo de loop encontrará o loop e emitirá diretamente mico-ops do buffer, eliminando a necessidade de busca de instrução e a instrução estágios de decodificação a serem ativados. Por outro lado, a microfusão combina pares de instruções como carga/operação ALU e operação/armazenamento ALU e os emite para uma única estação de reserva, onde ainda podem emitir de forma independente.

Intel Core i7-4

3. A instrução básica consiste em procurar a localização dos registros nas tabelas de registros, renomear os registros, alocar uma entrada de reordenação e buscar quaisquer resultados dos registradores ou do buffer de reordenação, antes de enviar os micro-ops para as estações de reserva.
6. O i7 utiliza uma estação de reservas centralizada com 36 entradas, compartilhada por seis unidades funcionais. As seis micro-operações podem ser enviadas para as unidades funcionais a cada ciclo de clock.
7. As micro-operações são executadas pelas unidades funcionais individuais e os resultados são enviados de volta para qualquer estação de reserva em espera, bem como para a unidade de retirada de registros, onde o estado do registro é atualizado assim que for afirmado que a instrução não é mais especulativa. A entrada correspondente à instrução no buffer de reordenação é marcada como concluída.
8. Quando uma ou mais instruções no topo do buffer de reordenação forem marcadas como concluídas, as escritas pendentes na unidade de retirada de registradores serão executadas e as instruções serão removidas do buffer de reordenação.

Intel Core i7-5

**Desempenho do Intel Core i7 CPI para o conjunto de benchmark
SPEC CPU2006**

Fonte: Arquitetura de Computadores: Uma Abordagem Quantitativa

DETI

Leitura sugerida

- *Arquitetura de Computadores: Uma Abordagem Quantitativa*, Hennessy JL, Patterson DA, 6^a Edição, Morgan Kaufmann, 2017
 - Capítulo 3: *Paralelismo em nível de instrução e sua exploração* (Seções 1 a 12)
- *Organização e arquitetura de computadores: prioridade para desempenho*, Stallings W., 10^a Edição, Pearson Education, 2016
 - Capítulo 16: *Paralelismo em nível de instrução e processadores superescalares*