Arquiteturas Superescalares

Mirian Ellen de Freitas – ra 029043
Fundação CPqD Centro de Pesquisa e Desenvolvimento em Telecomunicações
Rodovia Campinas - Mogi-Mirim, km 118,5
CEP 13086-902 - Campinas, SP - Brasil.
+55 19 3705-4493
mfreitas@cpqd.com.br

ABSTRACT
Este trabalho apresenta o conceito de arquiteturas superescalares. São descritas as principais limitações de para a execução paralela de instruções na arquitetura, bem como as características dos principais processadores de mercado com arquitetura superescalar.

Categories and Subject Descriptors
C.1.2 [**Multiple Data Stream Architectures**]: *Pipeline processors*.

General Terms
Performance, Design.

Keywords
Superscalar, pipelining technique,

1.INTRODUCTION
A busca por processadores cada vez mais rápidos segue três principais tendências: 1. o Superpipeline (pipelines mais longos com balanceamento de seus estágios); 2. a técnica Superescalar (replicação dos componentes internos do processador de modo que se possa colocar várias instruções em cada estágio do pipeline); 3. o escalonamento dinâmico do pipeline ou pipeline dinâmico (execução por hardware para evitar conflitos no pipeline).

As arquiteturas superescalares, que são alvo deste trabalho, são obtidas através do uso de processadores de múltiplo issue com o objetivo de permitir que múltiplas instruções possam ser processadas em um único ciclo de *clock*.

2.O DESENVOLVIMENTO DE PROCESSADORES DE MÚLTIPLO ISSUE
Processadores de múltiplo issue são basicamente de dois tipos: Processadores superescalares e processadores VLIW (very long instruction word).

O conceito de projetos de múltiplos issues surgiu de um trabalho publicado por volta dos anos 70 e focava abordagens de programação estatística 5.

Conforme Patterson e Hennessy 5, a IBM foi pioneira no trabalho de múltiplo issue. Em 1990, anunciou o primeiro processador superescalar de arquitetura RISC, o IBM Power-1 da linha RS/6000. Mais tarde, o projeto do Power-2 já descrevia as vantagens de programação dinâmica em comparação à programação estática.

Os anos 1994–95 foram marcados pelo anúncio de uma variedade de processadores superescalares (3 ou mais instruções/ciclo de *clock*) por cada um dos maiores produtores: Intel Pentium Pro e Pentium II, AMD K5, K6, e Althon, Sun UltraSPARC, Alpha 21164 e 21264, MIPS R10000 e R12000, PowerPC 603, 604, 620 e HP 8000.

Os últimos anos da década de 90 trouxeram uma segunda geração de muitos desses processadores (Pentium III, AMD Athlon, Alpha 21264, entre outros), todos contanto com programação dinâmica (dynamic scheduling) e quase universalmente suportando especulação.

3.ARQUITETURAS SUPERESCALARES
3.1Organização geral
Arquiteturas de processadores superescalares exploram o paralelismo de instrução - Instrucion Level Parallelism (ILP).

As múltiplas unidades funcionais independentes permitem despachar simultaneamente mais de uma instrução por ciclo. As figuras Figura 1 e Figura 2 mostram a organização geral de uma arquitetura superescalar.

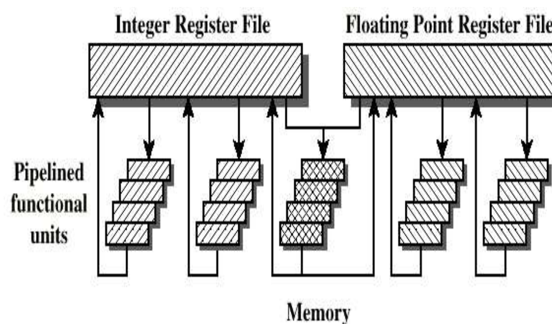

Figura 1 - Organização geral da arquitetura superescalar 5.

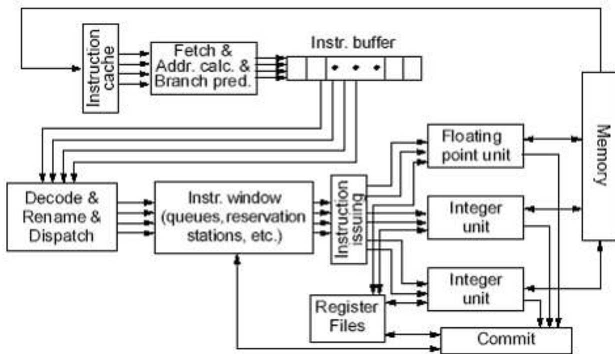

Figura 2 – Datapath de arquitetura superescalar 5.

3.2Execução Superescalar

Processadores superescalares variam o número de instrução por ciclo de *clock* escalando-as de modo estático ou dinâmico. As máquinas superescalares tentam paralelizar a execução de instruções independentes, em cada estágio do *pipeline*. Caso as instruções possuam dependências, um menor número de instruções será executado por ciclo.

O *pipeline* mais simples: (Uma instrução / 3 passos):

Figura 3 – Execução de instrução em *pipeline* simples 5.

Nas arquiteturas superescalares o *pipeline* de hardware permite que todas as unidades possam executar concorrentemente, porém ele pode ser *interrompido (stall)*por, por exemplo, instruções de *branchs*.

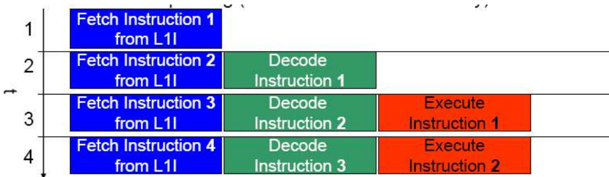

Figura 4 - Execução de instrução em hardware pipeline 5.

Aplicando analogia de correspondência de cada componente do computador a um componente de uma lavanderia, numa lavanderia superescalar seriam encontradas várias máquinas de lavar e várias de secar. Portanto, poder-se-ia lavar e secar o dobro de roupa com um jogo de duas máquinas de lavar e duas de secar, por exemplo, embora fosse necessário um maior esforço de manejo devido ao aumento do volume de roupas no mesmo intervalo de tempo. A desvantagem seria representada por este trabalho extra que se tem para manter todas as unidades ocupadas, além de se transferir um maior volume de informações para o próximo estágio.

Quando cada unidade pode ser colocada no *pipeline* (Multiply Pipeline) e cada estágio de *pipeline* leva menos ciclos para completar a operação, na arquitetura superescalar, se todos os estágios estão ativos em paralelo no *pipeline*, obtêm-se a finalização de mais de uma instrução por ciclo como ilustrado na Figura 5. Múltiplas unidades possibilitam a utilização de Instrucion Level Parallelism (ILP).

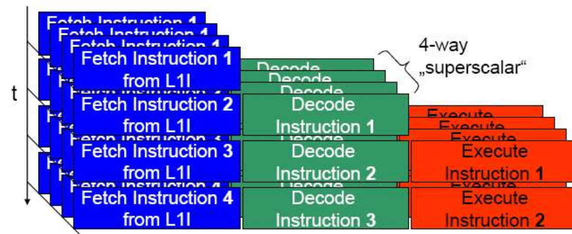

Figura 5 - - Execução de instrução em *pipeline* superescalar 5.

Dessa forma esses processadores podem sustentar uma taxa de execução maior que uma instrução por ciclo; o que resulta em um ganho de desempenho significativo em relação às arquiteturas de *pipeline* escalares, capazes de executar apenas uma instrução por ciclo.

A execução de várias instruções por estágio permite que se exceda a taxa do *clock*, isto é, permite a CPI menor que um. Por exemplo, um microprocessador de 1000 MHz superescalar com quatro instruções simultâneas, pode operar com uma taxa máxima de quatro bilhões de instruções por segundo, obtendo-se assim CPI de 0,25.

Para garantir o ganho potencial das arquiteturas superescalares em relação às outras, é necessário manter as unidades funcionais sempre ocupadas, ou seja, despachar o maior número de instruções possível a cada ciclo.

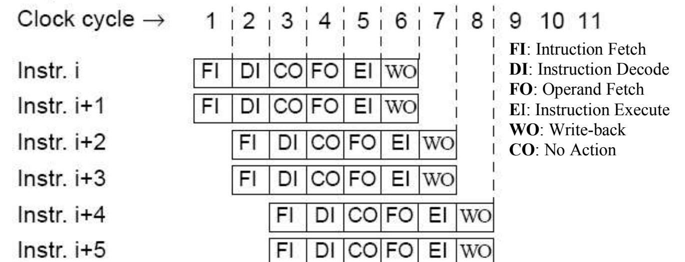

Figura 6 – Seqüência de instruções em execução Superescalar 5

3.3Melhorias em arquiteturas superescalares

Métodos de previsão de desvios, execução especulativa e escalonamento de instruções são objetos de interesse na melhoria das arquiteturas superescalares.

Para efeito de estudo de melhorias, três categorias de limitações devem ser consideradas 5 para otimização do uso do pipeline superescalar:

1. Conflitos por recursos que ocorrem quando duas ou mais instruções competem pelo mesmo recurso (registrador, memória, unidade funcional) ao mesmo tempo; a introdução de unidades de *pipeline* paralelas tem por objetivo reduzir possíveis conflitos por uso de recursos.

2. Dependência de controle (procedural), como por exemplo:

    a. A presença de *branches*, considerada o maior problema para obtenção de um paralelismo ótimo.

b. Instruções de tamanho variável, que não podem ser buscadas e despachadas em paralelo, pois uma instrução deve ser decodificada para se identificar a próxima. Assim arquiteturas RISC conseguem uma aplicação eficiente de técnicas superescalares, pois possuem instrução com tamanho e formato fixos.

3. Conflitos de Dados produzidos por dependências entre instruções do programa, que devem ser consideradas para validar o despacho e completude das operações em execuções fora de ordem.

A dependência de dados pode ser classificada em:

- Dependência real de dados - quando a saída de uma instrução é requerida como entrada da instrução subseqüente; não pode ser eliminada por compiladores ou técnicas de hardware.
- Dependência de saída – quando duas ou mais instruções estão escrevendo em um mesmo local.
- Antidependência - quando uma instrução usa um local e um operando enquanto a seguinte está escrevendo no mesmo local

As conseqüências dessas situações em arquiteturas superescalares são mais severas que em outros tipos de *pipeline*, pois a oportunidade potencial de grande paralelismo pode ser perdida.

Uma ferramenta importante nas arquiteturas superescalares mais modernas é a programação dinâmica das instruções, que as despacha por execução dinâmica, em paralelo e fora de ordem, quando são independentes e podem ser despachadas apenas com base na disponibilidade de recursos. Porém deve-se garantir que os resultados sejam idênticos aos produzidos por execução estrita.

Dependências de dados devem ser consideradas com cuidado, pois apenas parte das instruções são potencialmente adequadas para execução em paralelo. Para encontra-las o processador necessita selecionar de um conjunto suficientemente grande de seqüência de instruções, isto é, uma grande **janela de execução** é necessária.

Dependências de dados e de controle entre instruções impõem uma ordem de precedência na execução, de forma que instruções interdependentes não possam ser executadas em paralelo ou fora da ordem em que se encontram no código do programa. Tais dependências restringem o número de instruções que podem ser despachadas e, conseqüentemente, limitam o desempenho das arquiteturas superescalares, portanto o estudo de formas para minimizar o efeito de dependências entre instruções, principalmente em relação ao controle (desvios condicionais), são importantes para a obtenção do ganho de desempenho projetado.

A execução de instruções em paralelo requer aplicação de técnicas como o desdobramento de *loop*, *software pipeline,* escalonamento estático ou dinâmico de instruções (veja Figura 7), que permitam implementar políticas de execução como: despacho em ordem com finalização em ordem, despacho em ordem com finalização fora de ordem e despacho e finalização fora de ordem.

Dependências de saída e antidependências podem ser eliminadas automaticamente, com alocação de registradores extras. Esta técnica é chamada *register renaming*.

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	dynamic	hardware	static	in-order execution	Sun UltraSPARC II/ III
Superscalar (dynamic)	dynamic	hardware	dynamic	some out-of-order execution	HP PA 8500, IBM RS64 III
Superscalar (speculative)	dynamic	hardware	dynamic with speculation	out-of-order execution with speculation	Pentium III/4, MIPS R10K, Alpha 21264

Figura 7 – Características de uso de escalomaneto de Processadores com arquiteturas superescalares **5**.

3.4Escalonamento Superescalar Estático

No escalonamento superescalar estático as instruções são levadas para execução em ordem e todo conflito é checado em tempo de escalação pelo controle lógico do *pipeline*, que verifica conflitos entre as instruções que executarão em um dado ciclo de *clock*, e também entre todas as instruções que ainda estão em execução. Se algum fluxo de instrução é dependente (causa conflito de dados) ou não possui coerência com o critério de execução (pode causar um conflito estrutural), apenas instruções da seqüência que precede a instrução do conflito serão despachadas para executar no ciclo em questão.

No exemplo a seguir, de cálculo da norma de um vetor, as duas unidades de ponto flutuante de mult/add (MADD) não podem estar ocupadas ao mesmo tempo devido à dependência na soma armazenada na variável *t* (veja Figura 8).

t=0

do i=1,n

t=t+a(i)*a(i)

end do

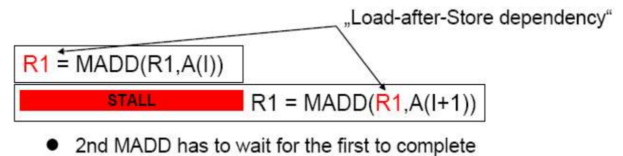

Figura 8 – Exemplo de dependência de dados 5

Numa versão otimizada duas *threads* independentes podem ser processadas por duas unidades separadas de ponto flutuante Mult/Add. Compiladores modernos fazem muitas dessas otimizações automaticamente de forma a aproveitar a estrutura da arquitetura superescalar (veja Figura 9).

t1=0

t2=0

do I=1,N,2

t1=t1+a(i)*a(i)

t2=t2+a(i+1)*a(i+1)

end do

t=t1+t2

R1 = MADD(R1,A(I))	R2 = MADD(R2,A(I+1))
R1 = MADD(R1,A(I+2))	R2 = MADD(R2,A(I+3))
...	

Figura 9 – Exemplo de otimização para eliminação de dependências 5 com adição de registrador extra.

O escalonamento estático usa a execução em ordem, como nos primeiros processadores superescalares, mas é o escalonamento dinâmico, com execução fora de ordem, o mais utilizado na maioria dos computadores *desk-top* e servidores atuais.

3.5Processadores Superescalares Escalonado Estaticamente

Num processador superescalar típico de oito estágios, podem ser executadas de zero (em *stall*) até oito instruções em um ciclo de *clock*.

Supondo-se que se tem um processador superescalar estático de quatro issues; durante a busca de instruções, o *pipeline* receberá uma das instruções de uma unidade de busca, que pode nem sempre ser capaz de entregar as quatro instruções, que constituem o conjunto/pacote de quatro instruções que possivelmente poderão ser despachadas.

A unidade de busca examina cada instrução, dentro do pacote escalado na ordem de programação. Se uma instrução pode causar um conflito estrutural ou um conflito de dados, com uma instrução ainda em execução ou com uma instrução do mesmo pacote, a instrução não é despachada.

Pode-se ter então de zero a quatro instruções de um pacote sendo executadas em um dado ciclo de *clock*. Embora a decodificação e a escalação da instrução logicamente sigam a ordem seqüencial, na prática, a unidade de despacho examina todas as instruções do pacote a ser escalado e checa os conflitos, decidindo se a instrução pode ser despachada.

Essas checagens de escalação são complexas, de modo que executa-las em um ciclo pode significar a determinação do comprimento mínimo do ciclo. Em muitos escaladores estáticos e em todos os superescalares dinâmicos, o estágio de escalação é quebrado e *pipelined*, de forma que possa escalar instruções em todo ciclo de *clock*. Isso não é totalmente simples porque o processador pode também detectar conflitos entre dois pacotes de instruções enquanto eles ainda estão na escalação do *pipeline*.

Uma abordagem é usar o primeiro estágio do *pipeline* para decidir quantas instruções do pacote poderão ser despachadas simultaneamente, ignorando instruções já despachadas, e usar o segundo estágio para examinar conflitos entre as instruções selecionadas e aquelas que já foram despachadas.

Dividindo-se o despacho em dois estágios e colocando-os em *pipeline*, o custo de desempenho do despacho da instrução superescalar tende a ter maior penalidade de *branch*, aumentando a importância de *branch prediciton*.

Com o aumento da taxa de escalação do processador, o pipelining de estágios de despacho pode tornar-se necessário. Embora quebrar o estágio de escalação em dois seja razoavelmente simples, é menos óbvia a maneira de colocar em *pipeline*. Assim, o despacho de instruções pode ser uma limitação na taxa de *clock* de processadores superescalares.

3.6Processador Superescalares MIPS Programados Estaticamente

Assumindo duas instruções possam ser despachadas por ciclo de *clock* e que, uma dessas instruções pode ser *load*, *store*, *branch* ou operações de inteiro na ULA e a outra possa ser uma operação de ponto flutuante (*loads* e *stores* consideradas operações com inteiros incluindo operações com registradores de ponto flutuante).

Despachar uma operação de inteiro com uma de ponto flutuante é muito mais simples e demanda menos arbitrariedade, simplificando o controle da unidade de despacho.

Buscar duas instruções é mais complexo que buscar apenas uma, pois um par de instruções pode aparecer em qualquer lugar do bloco de cachê. Muitos processadores apenas buscam uma instrução se a primeira instrução do par for última palavra do bloco de cachê.

Esta configuração é muito parecida com aquela utilizada no processador HP 7100. Embora computadores *desktop* tenham quatro ou mais despachos por ciclo de *clock*, o despacho dual em pipelines superescalares (de controle mais complexo) é comum no mercado de processadores embarcados.

Escalar duas instruções por ciclo requer busca e decodificação de 64 bits de instrução. Os primeiros superescalares limitavam a colocação dos tipos de instrução, por exemplo, a instrução de inteiro deveria vir primeiro, mas essa restrição desapareceu em processadores mais modernos.

Assumindo que a colocação da instrução não é limitada, existem três passos envolvidos na busca e escala: buscar duas instruções da cachê, determinar quantas podem ser escaladas (zero a duas) e despacha-las corretamente na unidade funcional.

Computadores superescalares geralmente se valem de uma unidade independente de pré-busca de instrução.

Para esse exemplo simples de superescalar, fazer a checagem de conflitos é relativamente direto, pois a restrição de um inteiro e uma instrução de ponto flutuante elimina a maioria das possibilidades de conflito na escalação de um pacote, permitindo em muitos casos a simples avaliação dos *opcodes* das instruções. A única dificuldade é quando a instrução de inteiro é um *load*, *store* ou *move* de ponto flutuante.

Essa possibilidade cria contenção para as portas dos registradores de ponto flutuante e pode criar também um conflito RAW, quando a segunda instrução do par depende da primeira (ex. a primeira é um PF *load* e a segunda uma operação de ponto flutuante, ou a primeira é uma operação de ponto flutuante e a segunda é um *store* de ponto flutuante).

Esse uso de restrição de escalonamento para reduzir a complexidade da estrutura do piepline e da detecção de conflitos, que representa um conflito estrutural, é comum nos processadores (também há possibilidade de conflitos WAR e WAW entre os limites dos pacotes).

Finalmente, as instruções escolhidas para a execução são despachadas para as unidades funcionais apropriadas.

A figura Figura 10 mostra como as instruções aparecem em pares no *pipeline*. Por simplicidade a instrução de inteiros é sempre a primeira, embora possa ser a segunda instrução no pacote escalado.

Instruction type	Pipe stages							
Integer instruction	IF	ID	EX	MEM	WB			
FP instruction	IF	ID	EX	EX	EX	MEM	WB	
Integer instruction		IF	ID	EX	MEM	WB		
FP instruction		IF	ID	EX	EX	EX	MEM	WB
Integer instruction			IF	ID	EX	MEM	WB	
FP instruction			IF	ID	EX	EX	EX	MEM
Integer instruction				IF	ID	EX	MEM	WB
FP instruction				IF	ID	EX	EX	EX

Figura 10 - *Pipeline* Superescalar em operação. 5

As instruções de inteiro e de ponto flutuante são despachadas ao mesmo tempo e são executas em diferentes unidades do *pipeline*. (assume-se que todas as instruções de ponto flutuante são adds e levam 3 ciclos para executar). O esquema apresentado irá melhorar apenas o desempenho de programas com grande quantidade de instruções de ponto flutuante.

Com esse *pipeline*, a taxa em que se pode escalar instruções de ponto flutuante é substancialmente impulsionada.

Será preciso também *pipelined* unidades de ponto flutuante, ou múltiplas unidades independentes, ou a *datapath* de ponto flutuante irá rapidamente se tornar um gargalo, reduzindo as vantagens conseguidas com a escalação.

Escalando operações de inteiro e ponto flutuante em paralelo, a necessidade por hardware adicional, além da de aperfeiçoamento de detecção lógica de conflitos é minimizada devido ao uso de diferentes conjuntos de registradores e unidades funcionais por esses tipos de operação em arquiteturas *load-store*.

Permitir escalação de *load* e *stores* de ponto flutuante junto a operações de ponto flutuante cria a necessidade de portas adicionais de *read/write* no arquivo de registro de ponto flutuante. Devido à existência de duas ou mais instruções no *pipeline*, um conjunto maior de trajetos de desvios serão necessários.

Uma complicação final é a manutenção de um modelo de exceção preciso. Para exemplificar quão impreciso o modelo de exceção pode ser, considere que:

- Uma instrução de ponto flutuante pode finalizar depois de uma instrução de inteiro que era posterior a ela na ordem do programa (ex. quando uma instrução de ponto flutuante é a primeira instrução num pacote escalado e ambas instruções são escaladas).
- A exceção de uma instrução de ponto flutuante pode ser detectada depois que a instrução de inteiro tenha sido concluída.

Esta situação, sem intervenção pode resultar numa exceção imprecisa porque a instrução de inteiro, que na ordem de programação segue a de ponto flutuante que gerou a exceção será completada.

Várias soluções são possíveis como a detecção antecipada de exceções de ponto flutuante, uso de mecanismos de software para restaurar o estado preciso de exceção reiniciando a execução e retardando a finalização da instrução até que a ocorrência de uma exceção seja impossível.

Manter o pico de *throughput* para esse *pipeline* dual é muito mais difícil que para o *pipeline* de escala de uma única instrução.

No *pipeline* clássico de cinco estágios, a operação de *load* tem uma latência de um ciclo de *clock*, o que previne o uso do resultado sem *stall*.

No *pipeline* superescalar o resultado de um *load* não pode ser usado no mesmo ciclo de *clock* ou no próximo ciclo, uma vez que no próximo ciclo, três instruções não podem usar o resultado do *load* sem *stalling*. A espera da decisão para tomar o *branch* torna-se equivalente a duas ou três instruções, dependendo se o *branch* é a primeira ou segunda instrução de um par.

Para explorar efetivamente o paralelismo disponível no processador superescalar são necessárias técnicas mais ambiciosas de compilação ou escalação de *hardware*, como, pois sem elas, o processador superescalar fornece um desempenho adicional muito pequeno.

3.7Software Pipeline

O uso de *pipelining* (IPL) requer compiladores inteligentes com reagendamento/reordenação das instruções para ocultar latências e auxiliar o processamento de instruções *fora de ordem* em todos os estágios do fluxo de dados. Isso é chamado ***software pipelining***. Há remoção de interdependências que bloqueiam a execução paralela de instruções.

Assumindo-se que a latência do Load (LD) = 1 ciclo, 2 LD e 2 MADD (MA) por ciclo são disponíveis como mostra o exemplo da :

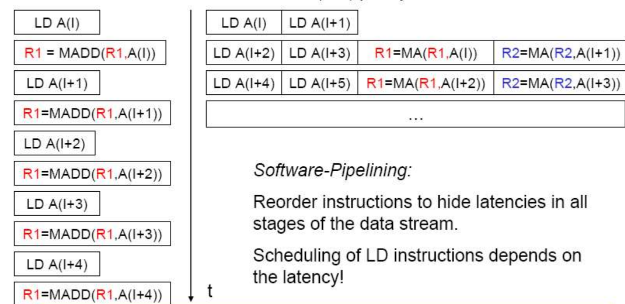

Figura 11 – Exemplo de *software pipelining* 5.

A técnica nem sempre pode ser aplicada, como no caso de *Pipelines* longos de ponto flutuante que são ineficientes para *loops* muito pequenos, pois o *Pipeline* deve ser preenchido e isso requer longos tempos iniciais.

3.8Escalonamento Dinâmico de múltiplas instruções

A combinação do escalonamento dinâmico com a predição dos desvios condicionais é conhecida como execução especulativa. Para total aproveitamento de sua possibilidade de paralelismo, as

máquinas dinâmicas precisam olhar vários segmentos para verificar qual instrução será executada e efetuar a execução especulativa das instruções com base na suas predições e dependências.

A execução escalonada dinamicamente permite esconder a latência da operação da memória, evitar paradas que o compilador não possa escalonar e executar instruções de maneira especulativa, enquanto aguarda a solução de conflitos.

O processador projetado em *pipeline* é dividido em três blocos principais: uma unidade de busca de instruções, várias unidades de execução e uma unidade de entrega. Cada unidade funcional tem seus *buffers*, chamados unidades de reserva, que armazenam os operandos e operações.

O escalonamento dinamico é um método para aumentar o desempenho em um processador com issue de múltiplas instruções, que aplicada em processadores superescalares traz benefícios de impulso do desempenho no conflito de dados, mas também permite que a potencialidade do processador supere as restrições de issue, pois embora o hardware não seja capaz de iniciar mais que uma operação de inteiro e uma de ponto flutuante num mesmo ciclo de *clock*, a programação dinâmica elimina essa restrição na escala da instrução ao menos quando o hardware roda fora de estações de reserva (stall).

O processo de escalonamento dinâmico busca instruções para serem executadas antes da parada (stall), enquanto se esperará que as dependências e conflitos que causaram a parada sejam resolvidos.

Não se deseja escalar instruções de reserva fora de ordem, pois pode levar a uma violação da semântica do programa. Para total ganho da vantagem do escalonamento dinâmico, será preciso remover a restrição de despachar uma operação de inteiro e uma de ponto flutuante num mesmo ciclo de *clock* o que complica o despacho da instrução. Uma alternativa seria usar um esquema simples: separa estruturas para registradores de dados de inteiros e de ponto flutuante, podendo utiliza-los simultaneamente em suas respectivas estações de reserva devido às instruções não acessarem o mesmo conjunto de registradores. Porém essa abordagem barra o despacho de duas instruções com uma dependência no mesmo ciclo de *clock*, como um *load* de ponto flutuante (instrução de inteiro) e uma adição de ponto flutuante, por exemplo.

Permitindo que o estágio de despacho possa manusear duas instruções arbitrárias por ciclo de *clock*, duas diferentes abordagens podem ser usadas para despachar múltiplas instruções por ciclo de *clock* em um processador escalado com programação dinâmica. Ambas fiam-se na observação de que sua chave é associada para a estação de reserva, atualizando as tabelas de controle do *pipeline*.

Uma abordagem é rodar esse passo em metade do ciclo de *clock*, assim duas instruções podem ser processadas em um ciclo inteiro. Uma segunda alternativa é construir a lógica necessária para manusear as duas instruções de uma vez, incluindo quaisquer possibilidades de dependência entre as instruções. Processadores superescalares modernos, que despacham quatro ou mais instruções por ciclo de *clock*, geralmente incluem ambas abordagens: fazem o *pipeline* e aumentam a lógica do despacho.

Existe um assunto final para ser discutido que é o uso da integração de *dynamic branch prediction* ao escalonamento dinâmico do *pipeline*.

O IBM 360/91 usa um esquema estático de *prediction*, permitindo que instruções sejam buscadas e despachadas, mas não executadas até que o *branch* tenha sido completado.

Assumindo que se possui uma implementação mais geral de um processador de dois *issues* dinâmicos, significando que se pose despachar qualquer par de instruções se estiverem disponíveis estações de reserva de direito, também se pode estender o esquema para lidar com registradores e unidades funcionais de ponto flutuante e inteiros.

EXAMPLE: Considerando a execução de um *loop* simples que soma um escalar em F2 para cada elemento de um vetor em memória. Usa-se um pipeline MIPS estendido com o algoritmo de Tomasulo 5 e com múltiplo issue:

```
Loop: L.D F0, 0(R1) ;        F0=array element
ADD.D F4, F0, F2 ;           add scalar in F2
S.D F4, 0(R1) ;              store result
DADDIU R1, R1, #-8 ;         decrement pointer; 8 bytes (per DW)
BNE R1, R2, LOOP ;           branch R1!=zero
```

Assume-se que ponto flutuante e inteiro possam ser despachados em todo ciclo de *clock*, mesmo sendo dependentes, e que uma unidade funcional de inteiros ALU é usada para operações e cálculos de endereço, além de uma unidade de *pipeline* de ponto flutuante separada para cada tipo de operação.

Assume-se que o issue e as escrita de resultados levam um ciclo cada e que há um hardware de *brach prediction* dinâmico e uma unidade funcional separada para avaliar as condições de *branch*.

Ma maioria dos processadores dinâmicos, a presença do estágio de escrita dos resultados significa uma latência efetiva de um ciclo extra, considerando-se um pipeline *em ordem* simples. Assim o número de ciclos de latência entre uma origem de instrução e o consumo da instrução que consome seu resultado é um ciclo para operações da ALU de inteiros, dois ciclos para *loads* e três ciclos para adição de ponto flutuante.

A Figura 12 mostra o despacho das instruções, o início delas e a escrita dos resultados para o CDB na primeira iteração do *loop*. Assume que existem dois CDBs e que branchs são contextualizados de forma simples (não há espera por *branchs*), mas a previsão de desvios (*branch prediction)* é perfeita. Também mostra o recurso usado pela unidade de inteiros, a unidade de ponto flutuante, o cachê de dados e os dois CDBs.

O *loop* será dinamicamente desenrolado e sempre que possível, as instruções serão despachadas em pares. O tempo de execução é mostrado na figura.

O *loop* irá continuar buscando e despachando uma nova iteração do *loop* a cada três ciclos de *clock* e sustentando uma iteração a cada três ciclos pode levar a um IPC de 5/3=1.67. A taxa de instruções, porém, é menor, olhando para o estágio de execução, pode-se verificar que a taxa sustentada de finalização de instrução é 15/16=0.94. Assumindo que *branchs* são perfeitamente

previsíveis, a unidade de issue irá eventualmente preencher toda estação de reserva com *stall*.

Iter. #	Instructions	Issues at	Executes	Memory access at	Write CDB at	Comment
1	L.D F0,0(R1)	1	2	3	4	First issue
1	ADD.D F4,F0,F2	1	5		8	Wait for L.D
1	S.D F4,0(R1)	2	3	9		Wait for ADD.D
1	DADDIU R1,R1,#-8	2	4		5	Wait for ALU
1	BNE R1,R2,Loop	3	6			Wait for DADDIU
2	L.D F0,0(R1)	4	7	8	9	Wait for BNE complete
2	ADD.D F4,F0,F2	4	10		13	Wait for L.D
2	S.D F4,0(R1)	5	8	14		Wait for ADD.D
2	DADDIU R1,R1,#-8	5	9		10	Wait for ALU
2	BNE R1,R2,Loop	6	11			Wait for DADDIU
3	L.D F0,0(R1)	7	12	13	14	Wait for BNE complete
3	ADD.D F4,F0,F2	7	15		18	Wait for L.D
3	S.D F4,0(R1)	8	13	19		Wait for ADD.D
3	DAADIU R1,R1,#-8	8	14		15	Wait for ALU
3	BNE R1,R2,Loop	9	16			Wait for DADDIU

Figura 12 – O ciclo de *clock* de despácho, execução e escrita de resulatdos para uma versão dual de *pipeline* 5.

O estágio de escrita dos resultados não se aplica para *stores* e *branchs*, uma vez que eles não escrevem em registradores. Assume-se que o resultado é escrito para o CDB que está disponível ao final do ciclo de *clock*. Para L.D. e S.D., a execução é o cálculo efetivo do endereço. Para *branchs*, o ciclo executado mostra quando a condição de *branch* está disponível. Qualquer instrução seguindo um *branch* não pode começar até que a condição de *branch* tenha sido avaliada e a *prediction* checada; assume-se que isso pode ocorrer um ciclo antes do ciclo de despacho, se os operandos estiverem disponíveis.

Assume-se uma unidade de memória, um pipeline de inteiros e um somador de ponto flutuante. Se duas instruções necessitam utilizar uma mesma unidade funcional num determinado tempo, a prioridade é dada para a instrução mais antiga.

Note que o load da próxima operação realiza um acesso de memória antes do store da iteração corrente.

Todos os desktops mais modernos e processadores de servidores são grandes e complexos chips com mais de 15 milhões de transistores por processador. Um simples superescalar de dois caminhos de issue que despacha instruções de ponto flutuante com instruções de inteiros, ou issue dual de instruções de inteiros (mas sem referência de memória) pode provavelmente ser construído com baixo impacto na taxa de *clock* e com um discreto tamanho em comparação com os processadores atuais. Tais processadores poderão desempenhar alta sustentação de taxa de pico que os processadores superescalares mais modernos e podem ter uma excelente relação custo-benefício.

4.Conclusões

A execução em *pipeline* superescalar aumenta o *throughput* de instruções, porém dependências de dados e de controle, aliadas à latência das instruções, significam um limite superior para o desempenho, uma vez que o processador necessita esperar a solução dessas dependências como ocorre, por exemplo, no caso de falha da predição do desvio condicional ou de solução de conflitos para o despacho das instruções. Portanto sem o uso de outras técnicas um maior número de unidades funcionais não melhora significativamente o desempenho.

Além de um melhor desempenho deve-se procurar a execução correta da seqüência de instruções. Portanto as otimizações realizadas em software pelos compiladores devem levar em conta a necessidade de uma compreensão precisa da estrutura do pipeline superescalar.

As arquiteturas com tamanho fixo da instrução permitem maior aproveitamento das características superescalares, favorecendo o aumento do desempenho.

Ao invés de adotar novas e dramáticas abordagens de microarquitetura, os últimos cinco anos forma focados no surgimento de taxas de *clock* de máquinas de múltiplos issues e estreitando o intervalo entre pico e desempenho sustentável. Os processadores de múltiplo issue com escalonamento dinâmico anunciados nos últimos dois anos (o Alpha 21264, o Pentium III, Pentium 4, e o AMD Athlon) possuem a mesma estrutura básica (três a quarto instruções por ciclo de *clock*) que o primeiro processador de múltiplo issue com escalonamento dinâmico anunciado em 1995, porém as taxas de *clock* são 4 a 8 vezes maiores, as *caches* são 2 a 4 vezes maiores, há 2 a 4 vezes mais registradores e o dobro de unidades de *load/store*. O resultado é uma performance 6 a 10 vezes maior 5.

A figura a seguir mostra os principais processadores dos últimos cinco anos e suas características 5.

Processor	System ship	Max. current CR (MHz)	Power (W)	Transistors (M)	Window size	Rename registers (int/FP)	Issue rate: Maximum/ Memory / Integer / FP / Branch	Branch Predict Buffer	Pipe-stages (int/ load)
MIPS R14000	2000	400	25	7	48	32/32	4/1/2/2/1	2K x 2	6
Ultra SPARC III	2001	900	65	29	N.A.	None	4/1/4/3/1	16K x 2	14/15
Pentium III	2000	1000	30	24	40	Total: 40	3/2/2/1/1	512 entries	12/14
Pentium 4	2001	1700	64	42	126	Total:128	3/2/3/2/1	4K x 2	22/24
HP PA 8600	2001	552	60	130	56	Total: 56	4/2/2/2/1	2K x 2	7/9
Alpha 21264B	2001	833	75	15	80	41/41	4/2/4/2/1	multilevel (see p. 258)	7/9
Power PC 7400 (G4)	2000	450	5	7	5	6/6	3/1/2/1/1	512 x 2	4/5
AMD Athlon	2001	1330	76	37	72	36/36	3/2/3/3/1	4K x 9	9/11
IBM Power 3-II	2000	450	36	23	32	16/24	4/2/2/2/2	2K x 2	7/8

Figura 13 – Principais processadores com arquitetura superescalar 5.

O Pentium III e o Althon escalam micro-operações e a janela é o número máximo de micro-operações em execução. O IBM, HP e o UltraSPARC suportam escalonamento dinâmico, mas não especulação.

A exploração de ILP para aumento de desempenho iniciou-se com os primeiros processadores de pipeline nos anos 60. Nas décadas de 80 e 90, o uso dessas técnicas permitiu um rápido aumento de desempenho, porém a questão atual é sobre como a exploração de ILP é crítica para a habilidade de melhoria de desempenho a longo prazo.

5.REFERENCES

[1] Patterson, David A., and Hennessy, John L. *Computer Organization and Design, The Hardware/Software Interface*. Morgan Kaufmann. Third Edition, 2005. Capítulos 6 e 7.

[2] Wellein, G., and Hager, G. *Programming Techniques for Supercomputers (PTfS*-m) - *Programming techniques for advanced microprocessors. HPC Services* - University Erlangen-Nürnberg. Notas de apresentação, 2004.

[3] Yang, Laurence T. *Instruction-Level Parallelism and Superscalar Processors*. Department of Computer Science - St. Francis Xavier University -Antigonish, NS, B2G 2W5, Canada. Disponível em http://juliet.stfx.ca/~lyang/csci-365/lectures/superscalar.ppt

[4] Yang, Laurence T. *Notas de aula e leitura complementar da disciplina csci-365.* Department of Computer Science - St. Francis Xavier University -Antigonish, NS, B2G 2W5, Canada. Disponível em http://juliet.stfx.ca/~lyang/csci-235/lectures/lecture5.pdf.