Arquiteturas Superescalares

Ricardo Dutra da Silva (RA 089088)
Instituto de Computação
UNICAMP
Campinas, Brasil
rdutra@ic.unicamp.br

RESUMO

Processadores superescalares exploram paralelismo em nível de instruções de maneira a capacitar a execução de mais de uma instrução por ciclo de clock. Este trabalho discute características de projetos de arquiteturas superescalares para aumentar o paralelismo em programas sequenciais. As principais fases descritas relacionam-se com a busca e processamento de desvios, determinação de dependências, despacho e emissão de instruções para execução paralela, comunicação de dados e commit de estados na ordem correta, evitando interrupções imprecisas. Exemplos de processadores superescalares são usados para ilustrar algumas considerações feitas para essas fases de projeto.

1. INTRODUÇÃO

Em meados da década de 1980, processadores superescalares começaram a aparecer [2, 5, 6, 7, 9, 15]. Os projetistas buscavam romper a barreira de uma única instrução executada por ciclo de clock [13].

Processadores superescalares decodificam múltiplas instruções de uma vez e o resultado de instruções de desvio condicional são geralmente preditas antecipadamente, durante a fase de busca, para assegurar um fluxo ininterrupto. Instruções são analisadas para identificar dependências de dados e, posteriormente, distribuídas para execução em unidades funcionais. Conforme a disponibilidade de operandos e unidades funcionais, as instruções executam em paralelo, possivelmente fora da ordem sequêncial do programa (*dynamic instruction scheduling*). Após as instruções terminarem, os resultados são rearranjados na sequência original do programa, permitindo que o estado do processo seja atualizado na ordem correta do programa.

Pode-se dizer que processadores superescalares procuram remover sequenciamentos de instruções desnecessários, mantendo, aparentemente, o modelo sequêncial de execução [3, 10]. Os elementos essenciais de processadores superescalares são: (1) busca de várias instruções simultaneamente, possivelmente predizendo *branchs*; (2) determinação de dependências verdadeiras envolvendo registradores; (3) despacho de múltipla instruções; (4) execução paralela, incluindo múltiplas unidades funcionais com pipeline e hierarquias de memória capazes de atender múltiplas referências de memória; (5) comunicação de dados através da memória usando loads e stores e interfaces de memória que permitam o comportamento de desempenho dinâmico ou não previsto de hierarquias de memória; (6) métodos para *commit* do estado do processo, em ordem.

Um programa pode ser visto como um conjunto de blocos básicos, com um único ponto de entrada e um único ponto de saída, formados por instruções contíguas [8]. Instruções em um bloco básico serão eventualmente executadas e podem ser iniciadas simultaneamente em uma *janela de execução*. A janela de execução representa um conjunto de instruções que pode ser considerado para execução em paralelo, conforme a dependência de dados. Instruções na janela de execução estão sujeitas a dependências de dados (RAW, WAR, WAW).

Para obter paralelismo maior do que aquele que um bloco básico está sujeito, pode-se usar predição de *branchs* e despacho especulativo. Desta maneira, são superadas dependências de controle. Se a predição é correta, o efeito das instruções sobre o estado do programa sai de especulativo e torna-se efetivo. Se a predição é incorreta, devem ser realizadas ações de recuperação para não alterar incorretamente o estado do processo. Resolvidas dependências de controle e dependências de nome, as instruções são despachadas e iniciam a execução em paralelo.

O hardware rearranja as instruções para execução paralela. Isso é feito considerando-se restrições de dependências verdadeiras e de hardware (unidades funcionais e data paths). Instruções possivelmente completam fora de ordem e, também, podem ser erroneamente especuladas. Para evitar que o estado do processo seja alterado erroneamente, uma estado temporário é mantido para os resultados de instruções. Esse estado temporário é mantido até a identificação de que uma instrução realmente seria executada em um modelo sequencial de processamento. Só então o estado do processo é atualizado através de *committing*.

Processadores superescalares mantém projetos lógicos com-

plexos. Mais de 100 instruções são comumente encontradas em fase de execução, interagindo entre elas e gerando exceções. A combinação de estados que podem ser gerados é enorme. Por esse motivo algumas arquiteturas escolhem mecanismos com técnicas em ordem, mais simples, para diminuir a complexidade e o tempo de lançamento no mercado [12]. Em [11] são avaliadas algumas limitações de processadores superescalares e apontadas características críticas de desempenho. Técnicas para verificação de modelos de processadores superescalares em relação ao seu conjunto de instruções são descritas em [4]

Neste trabalho são apresentada microarquitetura de processadores superescalares. Na seção 2 são discutidos o modelo e as etapas de um pipeline para processadores superescalares. Na seção 3, são descritas algumas considerações para projetos e, na seção 4, são discutidos alguns processadores superescalares. Na seção 5, as considerações finais são apresentadas.

2. ARQUITETURA SUPERESCALAR

A Figura 1 mostra um modelo de execução superescalar. O processo de busca de instruções, com predição de desvio, é usado para formar um conjunto dinâmico de instruções. Dependências são então verificadas e as instruções são despachadas (*dispatch*) para a janela de execução. Nesse ponto, as instruções não são mais sequenciais, mas possuem certa ordenação causada por dependências verdadeiras. As instruções são então emitidas (*issue*) da janela conforme a ordem das dependências verdadeiras e a disponibilidade de recursos de hardware. Após a execução, as instruções são novamente colocadas em ordem sequencial e atualizam o estado do processo.

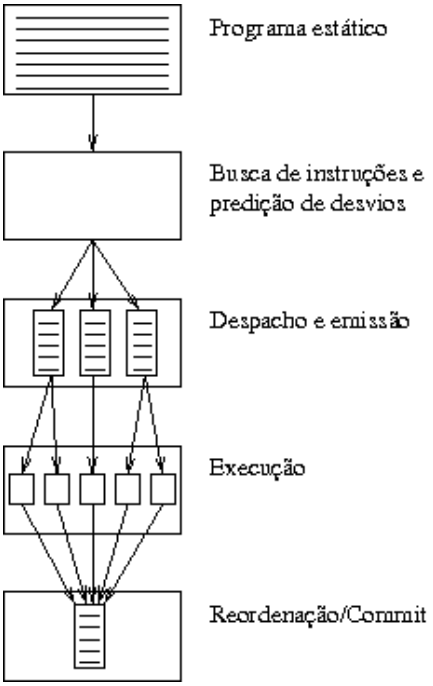

Figure 1: Execução em processadores superescalares.

2.1 Busca de instruções e predição de desvios

Para diminuir a latência de buscas são, em geral, usadas *caches de instruções*. As caches de instruções mantém blocos contendo instruções consecutivas. A arquitetura superescalar deve ser capaz de buscar, a partir da cache, múltiplas instruções em um único ciclo. A separação de caches de dados e de instruções é um ponto quase essencial para habilitar essa característica. Mas há exceções como o PowerPC [12, 13].

O despacho de um número máximo de instruções pode ser impedido por situações como *misses* na cache. Comumente, um buffer de instruções é usado para armazenar instruções buscadas e diminuir o impacto quando a busca está parada ou existem restrições de despacho [13].

O primeiro passo para realizar desvios rápidos envolve o reconhecimento de instruções de desvio. Uma maneira de tornar o processo rápido é armazenar, na cache de instruções, informações de decodificação. Essas informações são obtidas a partir de pré-decodificação das instruções durante a busca. Desvios condicionais costumam depender de dados não disponíveis no momento de decodificação. Os dados podem ainda ser provenientes de instruções anteriores. Nesses casos, podem ser usados métodos de predição.

O reconhecimento de desvios, modificação do PC (*Program counter*) e busca de instruções, a partir do endereço alvo, podem gerar atrasos e resultar em paradas do processador. A solução mais comum para esse tipo de problema envolve o uso do buffer de instruções para mascarar o atraso. Em buffers mais complexos, tanto instruções para o caso de o desvio ser tomado quanto para o caso de não ser tomado são armazenadas. Algumas arquiteturas usavam desvios atrasados (*delayed branchs*).

2.2 Decodificação, renomeação e despacho de instruções

Esta fase envolve a remoção de instruções do buffer de instruções, tratamento de dependências (verdadeiras e de nome) e despacho[1] de instruções para buffers de unidades funcionais. Durante a decodificação são preenchidos campos de: (i) operação; (ii) elementos de armazenamento relacionados com os locais onde operandos residem ou residirão; (iii) locais de armazenamento de resultado.

Os elementos de armazenamento são gerenciados por lógicas de renomeamento que, em geral, podem ser de dois tipos. Na primeira, uma tabela faz o mapeamento de registradores lógicos para registradores físicos. O número de registradores físicos é maior do que o número de registradores lógicos. Os mapeamento lógico-físico é realizado através de uma lista de registradores livres, como mostrado na Figura 2. Caso não haja registradores livres, o despacho é parado momentaneamente até algum registrador seja liberado. A renomeação é feita na ordem do programa [13].

[1]Despacho foi usado como tradução para a palavra *dispatch*, que envolve o envio de instruções para filas de unidades funcionais e estações de reserva. A palavra emissão foi usada para *issue*, que envolve o envio de instruções para unidades funcionais.

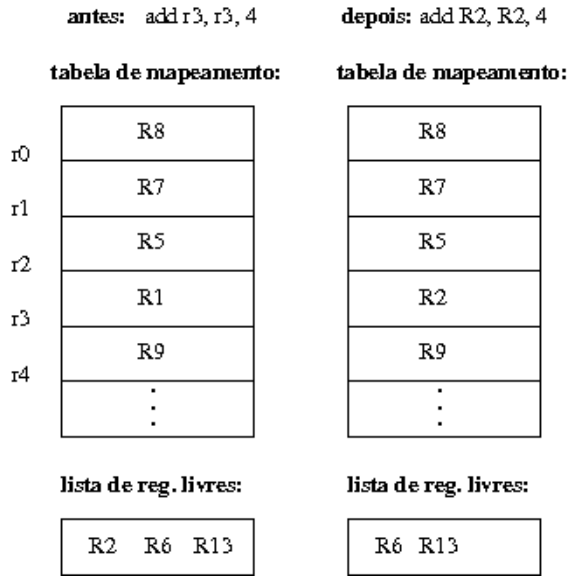

Figure 2: Renomeção de registradores. Registradores lógicos em letras maiúsculas e registradores físicos em letras minúsculas.

Um registrador físico deve ser liberado após a última referência feita a ele. Uma possível solução é incrementar um contador toda vez que uma renomeação de registrador fonte é feita e decrementar o contador sempre que uma instrução for emitida e ler o valor do registrador. Quando o contador chega a zero o registrador é liberado. Outro método, mais simples, espera até que uma instrução que renomeie o registrador receba commit.

O segundo método de renomeação é o *Reorder Buffer* [8]. O número de registradores lógicos e físicos é igual. O *reorder buffer* mantém um entrada para cada instrução despachada mas que ainda não recebeu *commit*. O nome vem do fato que o buffer mantém a ordenação das instruções para interrupções precisas. Pode-se pensar no *reorder buffer* como uma fila circular (Figura 3).

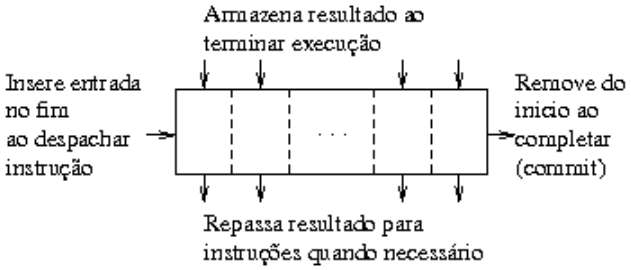

Figure 3: Modelo de *reorder buffer*. Entradas são inseridas e retiradas em ordem de fila. Resultados podem ser armazenados e lidos a qualquer momento.

A medida que instruções são despachadas na ordem sequencial do programa, elas são relacionadas a uma entrada no fim do *reorder buffer*. Os resultados de instruções que completam a execução são enviados para as suas entradas respectivas. Assim que uma instrução chega ao início da fila, seu resultado é escrito em um registrador e a instrução é retirada do *reorder buffer*. O despacho é interrompido se o reorder buffer está cheio. A Figura 4 mostra um exemplo de renomeação.

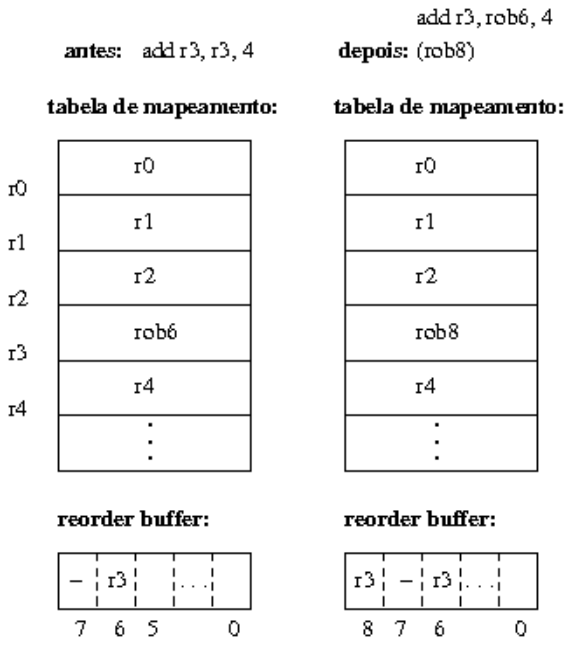

Figure 4: Exemplo de renomeação usando *reorder buffer*.

2.3 Emissão de instrução e execução paralela

Após a decodificação, é preciso identificar quais instruções podem ser executadas. Assim que os operandos estão disponíveis, a instrução está pronta para entrar em execução. No entanto, ainda á preciso verificar a disponibilidade de unidades funcionais, portas do arquivo de registradores ou do *reorder buffer*.

Três modos de organizar buffers para emissão de instruções são: *Método de Fila Única*, *Método de Filas Múltiplas* e *Estações de Reserva* [13]. No Método de Fila Única, apenas uma fila é usada, sem emissão fora de ordem. Não é preciso renomeamento de registradores. A disponibilidade de registradores é gerenciada através de bits de reserva para cada registrador. Um registrador é reservado quando uma instrução que o usa é emitida e liberado quando a instrução completa. Uma instrução pode ser emitida se os registradores de seus operandos não estão reservados.

No Método de Filas Múltiplas, instruções em uma mesma fila são emitidas em ordem, mas as diversas filas (organizadas conforme o tipo de instrução) podem emitir instruções fora de ordem.

Estações de reserva permitem que instruções sejam emitidas fora de ordem. Todas as estações de reserva monitoram simultaneamente a disponibilidade de seus operandos. Quando uma instrução é despachada para uma estação de reserva, os operandos disponíveis são armazenados nela. Caso os operandos não estejam disponíveis, a estação de reserva

espera o operando da operação que irá escrevê-lo. A instrução é emitida quando todos os operandos estão disponíveis. Também podem ser usados apenas apontadores para os locais onde encontram-se os operandos (registradores, reorder buffer) [8]. As estações de reserva podem ser divididas conforme o tipo de instrução ou colocadas juntas em um único bloco.

2.4 Operações de memória

Operandos de instruções de memória, ao contrário do que acontece com operações de ULA, não são conhecidos durante a decodificação. A determinação do local de acesso requer uma adição para formar o endereço, o que é feito na fase de execução. Após o cálculo do endereço, pode ainda ser necessária uma tradução (*translation*) para gerar o endereço físico. Esse processo é comumente acelerado por uma TLB. Uma vez obtido o endereço válido, a operação pode ser submetida para a memória. Existe a possibilidade de traduções e acessos serem realizados simultaneamente.

Para executar operações de memória mais rapidamente, podem ser usadas técnicas como: redução da latência, execução de múltiplas instruções ao mesmo tempo, execução sobreposta de operações de memória e operações sem acesso a memória e, possivelmente, execução de operações de memória fora de ordem [14]. Múltiplas requisições à memória exigem hierarquia com múltiplas portas, possivelmente apenas no primeiro nível, uma vez que acessos não costumam subir na hierarquia de memória [13]. Modos de implementação incluem o uso de células de memória com múltiplas portas, o uso de múltiplos bancos de memória ou a capacidade de realizar múltiplas requisições seriais em um mesmo ciclo.

A sobreposição de operações de memória com outras operações, sejam de memória ou não, exige uma hierarquia de memória não bloqueante. Dessa maneira, se houver *miss* em uma operação, outras operações prosseguem. Além disso, para permitir que instruções de memória sejam sobrepostas ou procedam fora de ordem, deve-se assegurar que *hazards* são tratados e que a semântica sequencial é preservada. *Store address buffers*, que mantém endereços de todas as operações de store pendentes, são usados para assegurar a submissão de operações à hierarquia de memória sem violações de hazard. Antes de uma instrução load/store ser emitida para a memória, o *store buffer* é verificado em busca de instruções store pendentes para o mesmo endereço.

2.5 Commit

A fase de *commit* permite manter os efeitos das instruções como se a execução fosse sequencial. Duas técnicas são comumente usadas para recuperar estados precisos. Ambas mantêm um estado enquanto a operação executa e outro estado para recuperação.

A primeira técnica usa *checkpoints*. O estado da máquina é salvo em determinados pontos enquanto instruções executam e, também, quando uma estado preciso é necessário. Estados precisos são recuperados de um *history buffer*. Na fase de *commit* são eliminados estados do *history buffer* que não são mais necessários.

A segunda técnica divide em dois o estado da máquina: estado físico e estado lógico. O estado físico é atualizado assim que as instruções completam. O estado lógico é atualizado na ordem sequencial do programa, assim que os estados especulativos são conhecidos. O estado especulativo é mantido em um *reorder buffer* que, após o *commit* de uma instrução, é movido para os registradores ou para a memória. A técnica com *reorder buffer* é mais popular pois, além de proporcionas estados precisos, ajuda a implementar a renomeação de registradores.

2.6 Software

Compiladores podem aumentar a possibilidade de paralelismo em um grupo de instruções, permitindo que sejam emitidas simultaneamente. Outra maneira é manter espaçamento de instruções dependentes para evitar paradas no pipeline. Isso pode ser feito com *scheduling estático* [8].

3. CONSIDERAÇÕES DE PROJETOS SUPERESCALARES

Em [12] é apresentada uma comparação entre duas vertentes de projetos: processadores superescalares desenvolvidos para obter menores tempos de clock e processadores superescalares projetados para maiores taxas de despacho.

Processadores que priorizam projetos de velocidade possuem pipelines profundos. A compensação para maiores frequências de clock é, em geral, obtida com menores taxas de despacho e maiores latências para loads e identificação de predições errôneas (DEC Alpha 21064).

Processadores projetados com taxas de despacho maiores procuram proporcionar um maior número de tarefas feitas por ciclo de clock. Características desses projetos são baixas penalidades para loads e mecanismos que permitem execução de instruções dependentes (IBM POWER).

Seis categorias de sofisticação são apresentadas entre os processadores. (1) Co-processadores de ponto-flutuante que não despacham várias instruções inteiras em um ciclo, nem mesmo uma instrução inteira e uma de desvio. Ao invés disso, são feitas emissões de uma instrução de ponto-flutuante e uma de inteiro. O desempenho é obtido em códigos de ponto-flutuante, permitindo que unidades de inteiros executem loads e stores de ponto-flutuante necessários (MIPS R5000). (2) Processadores que permitem despacho conjunto de instruções de inteiros e desvios, melhorando a performance em códigos de inteiros (HyperSPARC). (3) Múltiplas emissões de inteiros e instruções de memórias são permitidas com a inclusão de múltiplas unidades de inteiros (Intel Pentium). (4) Processadores que usam ULAs de três entradas para permitir despacho múltiplo de instruções inteiras dependentes (Motorola 68060). (5) Processadores que enfatizam exceções precisas. São usados mecanismos de recuperação e múltiplas unidades funcionais, com pouca ou nenhuma restrição de despacho (Motorola 88110). (6) Processadores que permitem despacho fora de ordem para todas as instruções (Pentium Pro).

4. MICROPROCESSADORES SUPERESCALARES

Nesta seção serão descritos alguns processadores superescalares, enfatizando aspectos de projeto característicos de cada processador.

4.1 Motorola 88110

O Motorola 88110 era um processador de emissão dupla com extensões para gráficos [6]. Uma estação de reserva era usada para branchs e três estações para stores. Desta maneira, o processador emitia instruções em ordem, com exceção de branchs e stores. Não era usada renomeação e permitia-se a emissão de instruções com dependências WAR. Stores podiam ser emitidos juntamente com as instruções que calculavam seu resultado. A unidade de store/load continha um buffer de quatro entradas.

Desvios eram preditos usando uma cache de *target instruction*, que matinha um registro dos últimos 32 desvios. Essa cache era indexada por endereços virtuais e precisava ser esvaziada em trocas de contexto. Instruções emitidas especulativamente eram rotuladas e anuladas em caso de predições erradas. Qualquer registrador escrito por instruções preditas erroneamente era restaurado a partir do *history buffer*. Stores condicionais, no entanto, não atualizavam a cache de dados, eram mantidos nas estações de reserva até que o desvio fosse resolvido.

As unidades funcionais eram dez: duas ULAs, somador, multiplicador, divisor, *bit-field*, *instruction/branch*, *data-cache* e duas *graphics*. Para manter exceções precisas e se recuperar de desvios previstos erroneamente, o processador usava *histoy buffers* de instruções. As 10 unidades de funções compartilhavam dois barramentos de 80 bits. Esses barramentos eram disputados pelas instruções por causa da latências diferentes.

4.2 MIPS R8000

O MIPS R8000 tinha o objetivo de ser um processador para computações de ponto-flutuante. Para evitar *misses* em caches, foram separadas referências feitas às instruções de inteiros e de memória das referências às instruções de ponto flutuante. A busca de instruções era feita a partir de uma cache de memória de 16KB, diretamente mapeada e com entradas de 32 bytes. Quatro instruções eram buscadas por ciclo. A cache de instruções, preenchida por uma cache externa em 11 ciclos, usava indexadores e tags virtuais. Um único bit era responsável pela predição de desvios para cada bloco de quatro instruções na cache.

O processador emitia até quatro instruções por ciclo, usando oito unidades funcionais. Havia quatro unidades de inteiros, duas de ponto flutuante e duas para loads e stores. Instruções de ponto flutuante eram armazenadas em uma fila até que pudessem ser despachadas. Desta forma, pipelines de inteiros podiam continuar mesmo quando um load de ponto-flutuante, que demorava cinco ciclos, era emitido com instruções de ponto-flutuante dependentes.

As referências de dados inteiros usavam uma cache interna de 16 KBytes, enquanto instruções de ponto-flutuante usavam uma cache externa de 16 MBytes. Por causa das decisões de projeto, poderiam haver problemas de coerência caso dados de ponto-flutuante e dados inteiros fossem reunidos em uma mesma estrutura de dados, como *union* [9]. A cache interna evitava esse tipo de problema mantendo um bit de validade para cada palavra.

4.3 MIPS 10000

O MIPS 10000 [15] busca até quatro instruções, as quais são pré-decodificadas (quatro bits são usados para identificar o tipo da instrução) antes da inserção na cache de 512 linhas. A cache de instruções, *two-way set associative*, contém uma tag de endereços e um campo de dados. Uma pequena TLB de oito entradas mantém um subconjunto das traduções da TLB principal. Logo após a busca, são calculados os endereços de jumps e branches, que são, então, preditos. A tabela de predição, com 512 entradas de 2 bits, está localizada no mecanismo de busca de instruções. A janela do processador considera até 32 instruções em busca de paralelismo.

Ao tomar um desvio, um ciclo é gasto no redirecionamento da busca de instruções. Durante o ciclo, instruções para um caminho não-tomado do desvio são buscadas e postas em uma *resume cache*, para o caso de uma predição incorreta. A *resume cache* tem espaço para 4 blocos de instruções, o que permite que até 4 desvios sejam considerados em qualquer momento.

Quando um branch é decodificado, o processador salva seu estado numa pilha de branch com 4 entradas. O processador para de decodificar se um branch chega e a pilha está cheia. Se um branch é determinado incorreto, o processador aborta todas as instruções do caminho errado e restabelece o estado a partir da pilha de branch.

Após a busca, as instruções são decodificadas e seus operandos são renomeados. O despacho para a fila apropriada (memória, inteiros ou ponto-flutuante) é feito com base nos bits da pré-decodificação. O despacho é parado se as filas estiverem cheias. No despacho, um *busy-bit* para cada registrador físico de resultado é estabelecido como ocupado. O bit volta ao estado não ocupado quando uma unidade de execução escreve no registrador. Todos registradores lógicos de 32 bits são renomeados para registradores físicos de 64 bits usando *free lists* (Figura 2). As *free lists* de inteiros e ponto-flutuante são quatro listas circulares, paralelas, de profundidade oito. Isso permite até 32 instruções ativas.

Cada instrução, nas filas, monitora os *busy-bits* relacionados com seus operandos até que os registradores não estejam mais ocupados. Filas de inteiros e ponto-flutuante não seguem uma regra de FIFO, funcionam de forma similar a estações de reserva. A fila de endereço é uma FIFO circular que mantém a ordem do programa.

Existem cinco unidades funcionais: um somador de endereços, duas ULAs, uma unidade de ponto flutuante (multiplicação, divisão e raiz quadrada) e um somador de ponto flutuante. Os pipelines de inteiros ocupam um estágio, os de load ocupam dois e os de ponto flutuante ocupam três. O resultado é escrito nos registradores no estágio seguinte. Estados precisos são mantidos no momento de exceções com

um *reorder buffer*. Até quatro instruções por ciclo recebem *commit* na ordem original do programa.

A hierarquia de memória é implementada de modo não bloqueante com dois níveis de cache *set-associative*. Todas as caches usam algoritmo de realocação LRU. Endereços de memória virtual são calculados como a soma de dois registradores de 64 bits ou a soma de um registrador e um campo imediato de 16 bits. A TLB traduz esses endereços virtuais em endereços físicos.

4.4 Alpha 21164

O Alpha 21164 abdica das vantagens de *dynamic scheduling* e favorece taxas de clock altas. Quatro instruções são buscadas de uma vez em uma cache de 8 Kbytes. Desvios são preditos usando uma tabela associada com a cache de instruções. Existe uma entrada de *branch history* com um contador de dois bits para cada instrução na cache. Apenas um branch predito e não resolvido é permitido a cada momento. Se houver um outro desvio sem que um prévio seja resolvido, a emissão é parada.

Instruções são despachadas para dois buffers de instruções, cada um com capacidade para 4 instruções. As instruções são emitidas, a partir dos buffers, na ordem do programa. Um buffer precisa estar totalmente vazio antes do outro começar a emitir instruções, o que restringe a taxa de emissão mas simplifica muito a lógica de controle.

São quatro as unidades funcionais: duas ULAs inteiras, um somador de ponto-flutuante e um multiplicador de ponto-flutuante. As ULAs não são idênticas, apenas uma faz deslocamentos e multiplicações inteiras, a outra é a única que avalia desvios. Resultados que ainda não receberam commit são usados através de *bypassing*. Instruções de ponto-flutuante podem atualizar os registradores fora de ordem, o que permite que aconteçam exceções imprecisas.

Dois níveis de cache são usados no chip. Existe um par de caches primárias de 8 Kbytes, uma para instruções e outra para dados. A cache secundária, *3-way set associative*, de 96 Kbytes, é compartilhada por instruções e dados. A cache primária, diretamente mapeada, permite acesso com taxa de clock muito alto. Existe um *miss address file* (MAF), com seis entradas, que contém o endereço e registrador alvo para loads que têm *miss*. O MAF pode armazenar até 6 misses na cache.

4.5 AMD K5

O AMD K5 usa um conjunto complexo de instruções com tamanho variável, Intel x86 [5]. Por esse motivo, as instruções são determinadas sequencialmente. O processo é feito na pré-decodificação, antes de entrar na cache de instruções. Cinco bits são usados após a pré-decodificação para informar se o byte é o começo ou fim de uma instrução. Também são identificados bytes relacionados com operações e operandos. A taxa de busca de instruções na cache é de 16 bytes por ciclo, que são armazenados em uma fila de 16 elementos para o despacho.

A lógica de predição é integrada com a cache de instruções, com uma entrada por linha da cache. Apenas um bit é usado para indicar a última predição de desvio. A entrada de predição contém um apontador para a instrução alvo, indicando onde esta pode ser encontrada na cache de instruções. Com isso reduz-se o atraso para busca de uma instrução alvo predita.

Dois ciclos são gastos para a decodificação. O primeiro estágio lê os bytes da fila, convertendo-os em instruções simples (ROPs – *RISC-like Operations*). Até quatro ROPs são formadas por vez. Frequentemente, a conversão requer um conjunto pequeno de ROPs por instrução x86 e, neste caso, a conversão é feita em um único ciclo. A conversão de instruções mais complexas é feita através de buscas em uma ROM. Até quatro ROPs são geradas por ciclo através da ROM. Depois da conversão, as instruções são geralmente executadas como operações individuais, ignorando a relação original com as instruções x86.

Após o ciclo de decodificação, as instruções lêem os operandos disponíveis, em registradores ou no reorder buffer, e são despachadas para estações de reserva a uma taxa de até quatro ROPs por ciclo. Se os operandos não estão prontos, a instrução espera na estação de reserva.

Existem 6 unidades funcionais: duas ULAs, uma unidade de ponto-flutuante, duas unidades de load/store e uma unidade de desvio. Uma das ULAs faz deslocamentos e a outra pode fazer divisão de inteiros. As estações de reserva são divididas para cada unidade funcional. Exceto pela unidade de ponto-flutuante, que tem uma estação de reserva, as outras têm duas. Conforme a disponibilidade dos operandos, ROPs são emitidas para a unidade funcional associada. Existem portas de registradores e paths de dados suficientes para que até quatro ROPs sejam emitidas por ciclo.

Existe uma cache de 8 Kbytes com quatro bancos. Store e loads duais são permitidos, desde que sejam para bancos diferentes, com exceção do caso em que a referência é para a mesma linha e duas requisições podem ser servidas.

Para manter estados precisos em casos de interrupção, é usado um *reorder buffer* de 16 entradas. O resultado de uma instrução é mantido no reorder buffer até que possa ser colocado no arquivo de registradores. O reorder buffer possui *bypass* e também é usado para recuperar predições de desvio incorretas.

4.6 AMD Athlon

Usa algumas das abordagens dos processadores K5 e K6, diferenciando-se no uso de um pipeline mais profundo, MacroOps e manipulação especial para instruções de ponto-flutuante e instruções multimídia. A parte inicial do pipeline, relacionada com o despacho contém 6 estágios. A predição de branchs para essa primeira parte do pipeline utiliza uma BHT (*branch history table*) de 2048 entradas, uma BTAC (*Branch Target Address Cache*) de 2028 entradas e um pilha de retorno de 12 entradas [1].

A decodificação é feita por três decodificadores *DirectPath* que podem produzir uma MacroOps cada, ou, para instruções complexas, por um decodificador *VectorPath* que sequencia três MacroOps por ciclo. Bits de pré-decodificação

auxiliam a decodificação.

Uma MacroOp é uma representação de uma instrução IA32 com tamanho fixo e que pode conte uma ou duas operações. Para o pipeline de inteiros as operações podem ser de seis tipos: load, store, load-store combinados, geração de endereços, ULA, e multiplicação. Desta maneira, instruções IA32 de registrador para memória e instruções de memória para registrador pode ser representadas por uma única MacroOp. No pipeline de ponto-flutuante, as operações podem ser: multiplicação, adição ou miscelânea. A vantagem de MacroOps é a redução do número de entradas em buffer necessárias.

Durante o estágio de despacho, MacroOps são alocadas em um *reorder buffer* de 72 entradas chamado *instruction control unit* (ICU). O buffer é organizado em 24 linhas de três entradas cada. O pipeline de inteiros é organizado simetricamente com uma unidade de geração de endereços e uma unidade de funções de inteiros conectados a cada entrada. A Multiplicação de inteiros é a única operação assimétrica, localizada na primeiro entrada. Instruções multimídia e de ponto-flutuante tem maiores restrições para as entradas.

Da ICU, as MacroOps são colocadas em um planejador (*scheduler*) de inteiros, de 18 entradas organizadas em seis linhas de 3 entradas cada, ou no planejador de ponto-flutuante e multimídia, de 36 entradas organizadas em 12 linhas de três entradas cada. Operações de load e store são enviadas para uma fila de 44 entradas para processamento. Inteiros ainda usam uma IFFRF (*future file and register file*) de 24 entradas. Operandos e tags são lidos dessa unidade durante o despacho e os resultados são escritos na unidade e na ICU quando as instruções completam.

Ao invés de ler operandos e tags durante o despacho, referências a registradores de ponto-flutuante e multimídia são renomeadas usando 88 registradores físicos. Como os operandos não são lidos durante o despacho, um estágio extra para leitura de registradores físicos é necessário. A execução dessas instruções não começa até o estágio 12 do Athlon.

O Athlon contém uma L1 integrada de 64 Kbytes e, inicialmente, continha um controlador para uma L2 externa de até 8 MBytes. Posteriormente foram usadas L2 internas. A L1 contém múltiplos bancos, o que permite dois loads ou stores por ciclo.

4.7 Intel P6

A microarquitetura faz a decomposição de instruções IA32 em micro-instruções [7]. Um pipeline de oito estágios para busca e tradução aloca micro-instruções em um *reorder buffer* de 40 entradas e em uma estação de reserva de 20 entradas.

Até três instruções IA32 podem ser decodificadas em paralelo. No entanto, por características de desempenho, as instruções devem ser rearranjadas de maneira que que apenas a primeira gere até quatro micro-instruções e, as outras duas, apenas uma micro-instrução. Instruções IA32 com operadores na memória requerem múltiplas instruções e limitam a taxa de decodificação para uma instrução por ciclo.

O preditor de branchs é adaptativo, de dois níveis, e quando o branch não é encontrado na tabela, um mecanismo é usado para fazer a predição baseando-se no sinal do deslocamento.

A estação de reserva é escaneada em modo de fila a cada ciclo, na tentativa de emitir até quatro micro-instruções para cinco portas. A primeira pota de emissão está ligada a seis unidades funcionais: inteiros, soma de ponto-flutuante, multiplicação de ponto-flutuante, divisão de inteiros, divisão de ponto-flutuante e deslocamento de inteiros. A segunda porta cuida de uma segunda unidade de inteiros e de uma unidade de branch. A terceira porta é dedicada para loads, enquanto as portas quatro e cinco cuidam de stores.

4.8 Pentium 4

O projeto é semelhante ao P6. Uma cache de instruções decodificadas, chamada *trace cache*, foi introduzida. A *trace cache* é organizada em 2048 linhas de seis micro-instruções cada. A largura de banda para busca na *trace cache* é de três micro-instruções por ciclo.

A profundidade do pipeline é maior do que no P6, 30 ou mais estágios. O pipeline para predição errada de desvios contém 20 estágios (o P6 continha 10). Dois preditores de branch são usados, uma para o início do pipeline e outro para a *trace cache*. A primeira BTB (*Branch target buffer*) contém 4096 entradas e usa um esquema híbrido [8].

O Pentium 4, ao contrário do P6, não armazena valores de fonte e resultado nas estações de reserva e no *reorder buffer*. Ao invés disso, são usados 128 registradores físicos para renomeação dos registradores inteiros e mas 128 registradores físicos para renomeação da pilha de pontos-flutuantes.

As micro-instruções são despachadas para duas filas. Uma é usada para operações de memória e a outra para as operações restantes. São usadas quatro portas de emissão, duas para load e store e duas para as outras operações. As duas últimas portas contém planejadores que podem emitir uma instrução por ciclo e outros planejadores que podem emitir duas micro-operações de ULA por ciclo. ULAs de inteiros usam pipelines que operam em três meio-ciclos, com dois meio-ciclos do pipeline de execução para cada atualização. *Offsets* de endereços do ponteiro de pilha são ajustados quando necessário para micro-operações de load e store que referenciam a pilha. Um *history buffer* grava as atualizações especulativas do ponteiro de pilha no caso de um branch predito errado ou no caso de uma exceção.

4.9 Pentium M

O Pentium M usa duas extensões para previsão de desvios. Um detector de loops captura e armazena contagens de loops em um conjunto de contadores em hardware. Isso produz predições precisas para loops *for*. Uma segunda extensão é um esquema adaptativo para desvios indiretos, projetado para desvios dependentes de dados. Para desvios indiretos preditos de forma errada, são alocadas entradas novas correspondentes ao conteúdo corrente de registradores de *global history*. Desta forma o *global histoy* pode ser usado para escolher entre várias instâncias de preditores para desvios indiretos dependentes de dados.

4.10 POWER4

Cada processador possui dois *cores* com oito unidades funcionais e uma cache L1. A cache L2 é compartilhada juntamente com o controlador da cache L3 (*off-ship*). Os cores emitem oito instruções por ciclo e a arquitetura considera maior desempenho através de um pipeline profundo, ao contrário do um pipeline menor com maiores taxas de emissão, usado anteriormente. Até 200 instruções permanecem simultaneamente no fluxo de execução.

A busca de instruções usa um preditor híbrido incomum de 1 bit. Um seletor escolhe entre 16K entradas de um preditor local e um preditor de 16K entradas global. O POWER4 usa grupos de cinco instruções com a quinta instrução sendo de desvio (nops são usados para completar *slots* não usados. Esses grupos são rastreados por uma tabela de 20 entradas, usada para rastrear quando instruções completam.

Apenas um grupo é despachado para as filas de emissão por ciclo. Um vez nas fila, as instruções podem ser emitidas fora de ordem. Onze filas de emissão são usadas, formando um total de 78 entradas. Bastantes registradores físicos são usados: 80 registradores físicos gerais, 72 registradores físicos para ponto-flutuante, 16 registradores físicos para *link* e *count*, 32 registradores físicos para campos de condições de registradores.

Nove estágios de pipeline são usados antes da emissão. Dois estágios são usados para busca das instruções e seis estágios são usados para quebrar instruções e formar grupos. Um estágio faz mapeamento de recursos e despacho. Instruções de inteiros requerem 5 estágios para execução incluindo emissão, leitura de operandos, execução, transferência, e escrita do resultado. Grupos completam em um último estágio. O total é um pipeline de 15 estágios para inteiros. Instruções de ponto-flutuante requerem mais cinco estágios.

5. CONCLUSÕES

Este trabalho apresentou os conceitos principais bem como a organização geral relacionada com processadores superescalares. Uma descrição histórica de alguns microprocessadores ainda foi apresentada, com o intuito de fornecer uma idéia de como as técnicas para processadores superescalares se apresentam em arquiteturas comercializadas.

Considerações de projeto envolvem cada uma das etapas em um pipeline superescalar. Durante a fase de busca usualmente são integradas caches de instruções e mecanismos de predição de desvios. O tamanho da cache e as lógicas de predição permitem ganhos de desempenho ao custo de mecanismos mais complexos para tradução de endereços e buscas antecipadas.

Os mecanismos de despacho apresentam desde restrições para despacho em ordem, que mantém estados mais precisos e simplificam circuitos, até restrições mais livres, que aumentam o paralelismo ao custo de projetos cuidadosos para manter a semântica do programa.

O conjunto de unidades funcionais varia bastante entre projetos diferentes. Pipelines profundos, em geral, são explorada para diminuir o ciclo de clock. Ciclos mais rápidos são comumente obtidos mantendo menores taxas de despacho, enquanto projetos com despachos mais agressivos mantém tempos de ciclos maiores para verificações de paralelismo entre instruções.

As limitações de desempenho das técnicas superescalares levaram à investigação de alternativas como VLIW (*Very Long Instruction Word*), EPIC (*Explicitly Parallel Instruction Computing*), SMT (*simultaneous multithreading*) e processadores multi-core.

6. REFERÊNCIAS

[1] AMD. AMD Athlon Processor – Technical Brief. Technical report, Advanced Micro Devices, Inc, 1999.

[2] P. Bannon and J. Keller. Internal architecture of alpha 21164 microprocessor. In *COMPCON '95: Proceedings of the 40th IEEE Computer Society International Conference*, page 79, Washington, DC, USA, 1995. IEEE Computer Society.

[3] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. *SIGPLAN Not.*, 26(6):241–255, 1991.

[4] J. R. Burch. Techniques for verifying superscalar microprocessors. In *DAC '96: Proceedings of the 33rd annual Design Automation Conference*, pages 552–557, New York, NY, USA, 1996. ACM.

[5] D. Christie. Developing the amd-k5 architecture. *IEEE Micro*, 16(2):16–26, 1996.

[6] K. Diefendorff and M. Allen. Organization of the motorola 88110 superscalar risc microprocessor. *IEEE Micro*, 12(2):40–63, 1992.

[7] L. Gwennap. Intel's p6 uses deeoupled supersealar design. *Microprocessor Report*, 9(2), 1995.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, May 2002.

[9] P. Y.-T. Hsu. Design of the R8000 microprocessor. Technical report, MIPS Technologies, Inc., 1994.

[10] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 272–282, New York, NY, USA, 1989. ACM.

[11] S. Palacharla, N. P. Jouppi, and J. E. Smith. Quantifying the complexity of superscalar processors, 1996.

[12] J. Shen and M. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Publishing Company, New Delhi, 2005.

[13] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609–1624, December 1995.

[14] G. S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. *SIGPLAN Not.*, 26(4):53–62, 1991.

[15] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, 1996.