Arquiteturas VLIW

Uma Abordagem Alternativa para Exploração de ILP

Flávia de Oliveira Santos (RA: 100594)
Instituto de Computação - Unicamp
Campinas – SP, Brasil
flavia.santos@students.ic.unicamp.br

RESUMO
Este artigo apresenta os principais conceitos relativos a arquiteturas VLIW para exploração de paralelismo em nível de instrução. São introduzidas também algumas técnicas de escalonamento estático de instruções necessárias para se obter código otimizado para arquiteturas VLIW. As técnicas abordadas são: *trace scheduling*, *software pipeline*, *loop unrolling* e *superblock schedulling*. O artigo aponta as vantagens e desvantagens na utilização da arquitetura. Finalmente é apresentada uma conclusão em termos de perspectivas futuras das arquiteturas VLIW.

Termos Gerais
Projeto de computadores.

Palavras-chave
VLIW, Escalonamento estático, Paralelismo em nível de instrução.

1. INTRODUÇÃO
A demanda por processadores que apresentem elevado desempenho constitui um fator importante para os investimentos em projetos para melhoria e pesquisa de novas arquiteturas. Um aspecto chave para o aumento do desempenho dos processadores é a habilidade de explorar o paralelismo em nível de instruções (ILP) com despacho múltiplo de instruções. Processadores superescalares podem despachar números variados de instruções por ciclo de *clock* e obter resultados consideráveis [1].

Uma abordagem alternativa é a VLIW (*Very Long Instruction Word*). Processadores VLIW despacham um número fixo de instruções formatadas como uma instrução com o paralelismo entre instruções indicado explicitamente. Visto que o *hardware* VLIW não é responsável por descobrir as oportunidades de executar múltiplas instruções concorrentemente, sua complexidade diminui bastante. Por esse motivo, arquiteturas VLIW oferecem um maior desempenho a um custo menor em comparação a processadores superescalares escalonados dinamicamente.

Por possuírem uma estrutura de controle simplificada, ao invés de explorar recursos em hardware, como o algoritmo de Tomasulo [4], arquiteturas VLIW utilizam técnicas de escalonamento estático pelo compilador, para escalonar múltiplas operações independentes em uma instrução longa, conseguindo assim o despacho simultâneo de diversas operações.

As palavras de instrução longas consistem de várias operações aritméticas, lógicas e de controle cada uma das quais poderia ser uma operação individual em um processador RISC simples. É responsabilidade do compilador escalonar as operações de modo a utilizar o melhor possível as unidades funcionais disponíveis no processador.

Mesmo antes do advento das primeiras máquinas VLIW, havia diversos processadores e dispositivos computacionais que utilizavam uma instrução longa para controlar o funcionamento de diversas unidades funcionais em paralelo. Mas foi o trabalho de Joseph Fisher sobre uma técnica global de compactação de micro código, chamada de *trace scheduling* [2] que impulsionou o desenvolvimento deste novo conceito de arquitetura.

Em 1984, Bob R. Rau e outros fundaram a Cydrome, Inc. para iniciar a construção de supercomputadores utilizando a filosofia VLIW e com suporte a técnica de *software pipeline* [4]. Nesse mesmo ano, Joseph Fisher e outros companheiros da Universidade de Yale fundaram a Multiflow Computer, Inc. com o mesmo objetivo. Ambas as empresas chegaram a lançar seus respectivos produtos finais, porém por volta do final dos anos 80, a Multiflow Computer Inc. e a Cydrome Inc. fecharam suas portas devido a dificuldades financeiras fazendo com que a arquitetura VLIW fosse praticamente esquecida pelo mercado.

Com o fechamento dessas empresas, o desenvolvimento da tecnologia VLIW procedeu lentamente, até que em meados da década de 90, descobriu-se que arquiteturas VLIW eram ideais para o processamento de algoritmos complexos e repetitivos. Processadores como o Itanium da Intel [5] e o Crusoe da Transmeta [6] são dois exemplos dessa nova geração de processadores VLIW.

O objetivo deste trabalho é abordar os principais conceitos envolvidos com processadores e técnicas que utilizam o conceito de palavras longas de instrução. Na seção 2, são abordados os aspectos arquiteturais gerais dos processadores VLIW. A seção 3 é dedicada a abordar algumas técnicas de escalonamento estático utilizadas para melhorar o desempenho desses processadores. Na seção 4 apresentamos as principais vantagens e desvantagens dos processadores VLIW. Uma conclusão é finalmente apresentada na seção 5.

2. ARQUITETURA VLIW
A arquitetura VLIW pode ser generalizada a partir de dois conceitos bem fundamentados: micro código horizontal e processamento superescalar [1]. Um processador VLIW típico

tem palavras com centenas de bits e utiliza múltiplas unidades funcionais independentes. Todas as unidades funcionais compartilham um grande banco de registradores comum com múltiplas portas de leitura (Figura 1). As operações a serem executadas paralelamente são empacotadas em uma instrução VLIW.

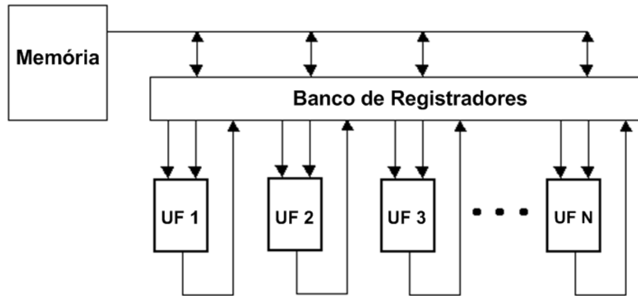

Figure 1. Arquitetura VLIW genérica.

Como ocorre no micro código horizontal, diferentes campos da instrução VLIW possuem *opcodes* que especificam as operações a serem executadas (Figura 2). A arquitetura VLIW busca uma palavra VLIW em um único endereço de memória e cada unidade funcional (UF) executa uma das diferentes operações contidas na palavra.

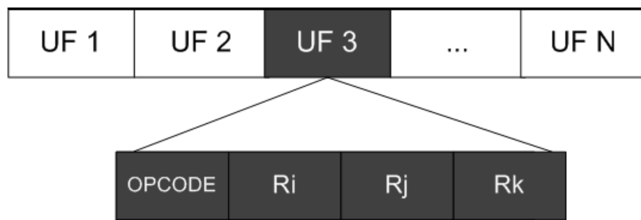

Figure 2. Instrução VLIW.

As instruções longas são montadas com a utilização de técnicas de escalonamento por *software* aplicadas em tempo de compilação ou através da compactação do código gerado por um compilador convencional.

Para tirar proveito dos recursos da máquina é necessário um compilador eficiente para escalonar estaticamente as instruções independentes de forma a não gerar conflitos estruturais e manter a semântica do programa durante a execução, em detrimento das dependências de dados existentes [1].

Para obter essa máxima utilização, um processador VLIW pode ter suas unidades funcionais totalmente em *pipeline*, implementar diferentes tipos de codificação da palavra de instrução, ter hardware de suporte ao tratamento de exceções e implementar mecanismos de bypassing.

Algumas dessas melhorias não fazem parte da idéia original da arquitetura VLIW, aumentando a complexidade do hardware, mas sua incorporação pode trazer muitos benefícios e solucionar eventuais problemas relacionados a esse tipo de arquitetura. Portanto, vamos abordar cada um desses mecanismos rapidamente nas próximas seções.

2.1 Pipeline

Na Figura 3 é apresentado um modelo de pipeline de 4 estágios com capacidade de execução de até 3 operações ao mesmo tempo.

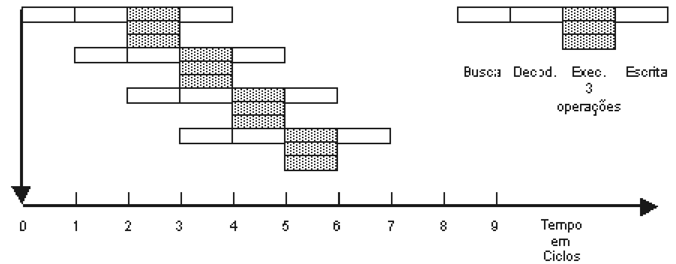

Figure 3. Pipeline em uma máquina VLIW. Extraído de [1].

Os estágios de *pipeline* desse exemplo são: busca, decodificação, execução e escrita. O estágio de execução demonstra a principal diferença dos processadores VLIW. A busca é feita em um único endereço, porém, cada instrução VLIW especifica múltiplas operações. A instrução contém os *opcodes* de cada uma das operações a serem executadas sobre os dados. Como a instrução VLIW tem um tamanho fixo, sua decodificação é mais simples se comparada à decodificação de processadores *superescalares*. No estágio de execução, as operações são despachadas para as unidades funcionais correspondentes, não havendo quaisquer mecanismos em *hardware* para detecção e tratamento de dependências. Esta tarefa deve ter sido realizada de forma estática por um compilador que tenha todo conhecimento dos detalhes da arquitetura, como número de estágios do *pipeline*, número de unidades funcionais e a latência de cada uma delas.

Entretanto, a eficiência deste tipo de *pipeline* é limitada a capacidade do compilador em encapsular as operações em instruções longas. Uma baixa densidade de operações por instrução irá resultar em unidades funcionais ociosas e conseqüentemente num baixo número de operações completadas por ciclo.

2.2 Mecanismo de Busca das Instruções

Arquiteturas VLIW requerem mecanismos de busca de instruções que suportem uma elevada largura de banda, para permitir que as instruções longas possam ser conduzidas ao estágio de decodificação do *pipeline*. A importância de ter mecanismos de busca que atendam essas exigências é evidente, pois o limite da largura de banda das *caches* de instrução pode passar a ser um gargalo nesse tipo de arquitetura [1].

Em termos de codificação da instrução, pode-se classificar as arquiteturas VLIW em duas categorias [1]:

- *Uncompressed encoding* - formato de instrução com número fixo de bits, codificando explicitamente NOPs (operações que não fazem nada) quando uma operação não pode ser escalonada na instrução. Requer um *hardware* mais simples na busca de instruções.
- *Compressed encoding* - formato de instrução com número variável de bits, não escalonando NOPs na instrução. Requer um hardware mais complexo para a busca de

instruções, mas obtém um melhor uso dos recursos da memória.

2.3 Bypassing

A técnica de *bypassing*, ou *forwarding*, é muito utilizada para resolver possíveis dependências de dados. Nessa técnica, durante a execução de uma instrução do tipo *Load/Store*, no estágio execução calcula-se o endereço de memória referenciado e o resultado é alimentado de volta às unidades de funcionais. Caso seja detectado que a operação anterior utiliza o valor contido no endereço de memória referenciado, a lógica de controle seleciona o resultado recebido como entrada no estágio de execução ao invés do valor lido.

O custo para se implementar o *bypassing* se concentra principalmente nos comparadores e na conexão entre os registradores do *pipeline* e as entradas das unidades funcionais.

2.4 Tratamento de Interrupções

Devido à sua filosofia, as operações em uma arquitetura VLIW podem inerentemente completar fora da ordem seqüencial, pois essa foi previamente modificada pelo compilador. Portanto, em tempo de execução, o processador tem conhecimento apenas da ordem das múltiplas operações do programa escalonado.

Quando uma interrupção ocorre, o estado de processador com capacidade de despacho fora de ordem, incluindo VLIW, dificilmente está em um estado consistente com o estado seqüencial. A grande dificuldade encontrada em processadores com capacidade de despacho fora de ordem é o fato de que instruções que sucedem a instrução interrompida podem já ter completado, enquanto instruções que a precedem podem nem ter sido despachadas. Para lidar com esse tipo de situação, técnicas como *Reorder Buffer* e *Future File* podem ser estendidas para o uso em arquiteturas VLIW, como discutido em [1].

3. ESCALONAMENTO ESTÁTICO

Para manter as unidades funcionais ocupadas, é preciso haver paralelismo suficiente em uma seqüência de código para preencher as unidades funcionais disponíveis. Esse paralelismo é descoberto desdobrando-se loops e escalonando-se o código dentro do corpo do loop. Se o desdobramento gerar código em sem fluxo de controle interno, as técnicas de escalonamento local, que operam sobre um único bloco básico, podem ser utilizadas. Se a localização e a exploração do paralelismo exigir a mudança de instruções entre os desvios, um algoritmo de escalonamento global substancialmente mais complexo terá que ser utilizado. Os algoritmos de escalonamento global não são apenas mais complexos em estrutura, mas também precisam lidar com escolhas significativamente mais complicadas em otimização, pois a movimentação de código entre os desvios é dispendiosa [4].

Por esses motivos, realizar o escalonamento estático tem sido um desafio para os projetistas durante vários anos, do qual originaram uma série de técnicas tanto em software como em hardware.

No caso da arquitetura VLIW, é necessário um compilador eficiente para escalonar estaticamente as instruções independentes de forma a não gerar conflitos estruturais e manter a semântica do programa durante a execução, em detrimento das dependências de dados existentes. Esse fato motiva a apresentação nas próximas subseções de algumas das técnicas de software que podem ser utilizadas para prover uma melhor utilização dos recursos de hardware. São elas: *loop unrolling*, *trace scheduling*, *software pipeline* e *superblock scheduling*.

3.1 Loop Unrolling

O *loop unrolling* [1][4] é uma técnica que consiste em diminuir o número de iterações de um *loop* estendendo o código contido em seu corpo. Dessa forma, o overhead de controle do *loop* é diminuído e instruções que originalmente pertenciam a iterações distintas podem ser escalonadas em conjunto.

No *loop* da Figura 5, desdobramos três iterações do *loop*. Foram considerados dados do tipo *double* e latências arbitrárias para as operações, resultando em algumas paradas no *pipeline* representadas pelos *stalls*. Após a aplicação do *loop unrolling* (Figura 5(b)), as paradas ocasionadas pelo código original (Figura 5(a)) foram eliminadas e as instruções de controle foram divididas por 3 (fator de *unrolling*).

```
Loop: LD F0, 0(R1)
      stall
      ADDD F4, F0, F2
      stall
      stall
      SD 0(F1), F4
      SUBI R1, R1, #8
      stall
      BNEZ R1, Loop
      stall
```

(a) *Loop* original

```
Loop: LD F0, 0(R1)
      LD F6, -8(R1)
      LD F10, -16(R1)
      ADDD F4, F0, F2
      ADDD F8, F6, F2
      ADDD F12, F10, F2
      SD 0(F1), F4
      SUBI R1, R1, #24
      SD -8(F1), F8
      BNEZ R1, Loop
      SD 8(R1), F12
```

(b) Após aplicado *loop unrolling.*

Figura 5 – Exemplo da técnica de *loop unrolling*. Adaptado de [1].

A restrição dessa técnica está no conhecimento do número de iterações do *loop* em tempo de compilação, na disponibilidade de registradores para armazenar os resultados intermediários da iteração e no crescimento no tamanho do código [4].

3.2 Trace Scheduling

O uso de *trace scheduling* [2][3][4] é útil para processadores com um grande número de despachos por ciclo, onde a técnica de *loop unrolling* não é suficiente por si só para alcançar ILP necessária para manter as unidades funcionais trabalhando.

O escalonamento de trace é um modo de organizar o processo de movimento de código global, de modo a simplificar o escalonamento de código contraindo os custos do possível movimento do código nos caminhos menos freqüentes [4].

Um *trace* é um segmento de código que representa um provável "caminho" durante a execução e que não está restrito aos limites de bloco básico [1]. No lado esquerdo da Figura 6 há um exemplo de dois traces no código original. Já no lado direito, temos o *trace* após a adição de código de compensação efetuada durante o processo de *trace scheduling*.

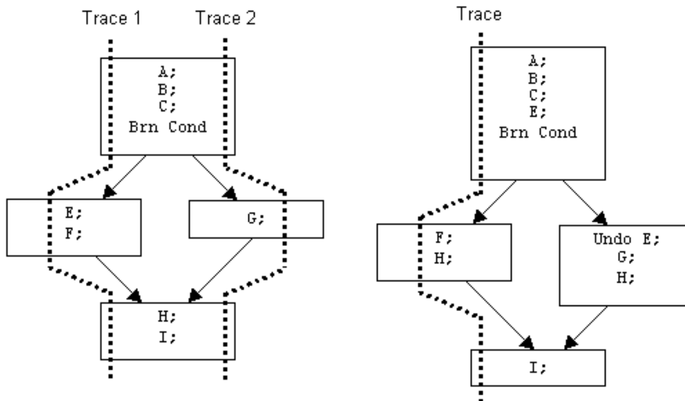

Figura 6 - *Trace scheduling*. Extraído de [1].

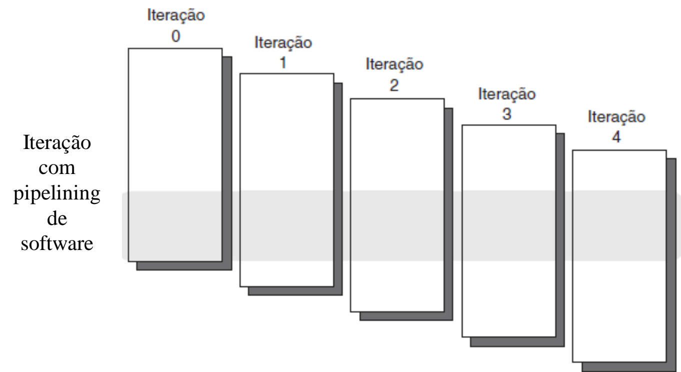

Figura 7 - *Software Pipeline*. Extraído de [4].

Existem duas etapas na técnica de *trace scheduling* [4]. A primeira etapa, chamada seleção de *trace*, tenta encontrar uma provável seqüência de blocos básicos cujas operações serão reunidas no *trace*. Quando o *trace* é selecionado, o segundo processo, chamado compactação de *trace*, tenta espremer o *trace* em um pequeno número de instruções longas. A compactação consiste basicamente em tentar escalonar (agrupando e movendo) as operações do *trace* no menor número de palavras longas possível.

Por poder gerar *overheads* significativos sobre o caminho pouco freqüente designado, a técnica de *trace scheduling* é melhor utilizada onde a informação de perfil indica diferenças significativas na freqüência entre diferentes caminhos e onde a informação de perfil é altamente indicativa do comportamento do programa, independente da entrada [4]. Isso limita sua aplicabilidade efetiva a certas classes de programas.

3.3 Software Pipeline

O *pipelining* de *software* [1][3][4] é uma técnica para reorganizar loops de modo que cada iteração no código preparado seja feita a partir de instruções escolhidas de diferentes iterações do *loop* original. O escalonador neste exemplo basicamente intercala instruções de diferentes iterações de *loop*, de modo a separar as instruções dependentes que ocorrem dentro de uma única iteração do *loop*. Essa técnica é muitas vezes denominada de "*Symbolic Loop Unrolling*", pois permite obter instruções independentes de diferentes iterações, sem efetivamente desenrolar o *loop*, mantendo o código mais compacto.

Um loop com pipelining de software intercala instruções de diferentes iterações sem desdobrar o *loop*, conforme pode ser visto na Figura 7. Essa técnica é o correspondente no *software* ao que o algoritmo de Tomasulo faz no *hardware*.

O *pipelining* de *software* e o desdobramento de *loop*, além de gerar um *loop* interno mais bem escalonado, reduzem um tipo diferente de *overhead* [4]. O desdobramento do *loop* reduz o *overhead* do *loop* – o código de desvio e atualização de contador. O *pipelining* de *software* reduz o tempo em que o *loop* não está executando na velocidade de pico para uma vez por *loop* no início e no final. Como essas técnicas atacam dois tipos diferentes de *overhead*, o melhor desempenho pode vir da realização de ambos.

3.4 Superblock Scheduling

Superblocos [4] são formados por um processo semelhante ao que é usado para traces, mas são uma forma de blocos básicos estendidos, que são restritos a um único ponto de entrada, mas permitem múltiplas saídas.

Para se formar os superblocos, primeiramente encontram-se os *traces* através de informações de *profiling*. Em seguida, eliminam-se os pontos de entrada que estão no meio dos *traces* através da utilização do método de *tail duplicatication* [4], que move os pontos de entrada para blocos básicos duplicados.

Como os superblocos têm apenas um único ponto de entrada, a compactação de um superbloco é mais fácil do que a compactação de um *trace*, pois somente o movimento do código passando por uma saída precisa ser considerado. Esses blocos podem, então, ser escalonados mais facilmente.

A técnica de superbloco reduz a complexidade de manutenção e escalonamento, mas pode aumentar o tamanho do código em comparação com a técnica mais geral de geração de *trace*.

4. VANTAGENS E DESVANTAGENS

As arquiteturas VLIW possuem prós e contras. Algumas das principais vantagens são [1]:

- Arquitetura regular e exposta ao compilador permitindo um escalonamento de instrução com elevado grau de liberdade;
- O compilador tem conhecimento prévio de todos os efeitos das operações sobre a arquitetura, ou seja, está apto a resolver conflitos estruturais e de dados em tempo de compilação;
- Capacidade de despacho de múltiplas operações;
- *Hardware* de controle simples, permitindo teoricamente um ciclo de *clock* menor.

Alguns dos pontos negativos são [1]:

- Previsão estática do caminho tomado em desvios condicionais pode afetar seu desempenho, pois informações importantes disponíveis em tempo de execução são completamente negligenciadas;
- Problemas de compatibilidade de código com máquinas não paralelas e também com máquinas de diferentes famílias VLIW;

- Pior densidade de código quando muitos NOPs são codificados na instrução longa, levando a uma má utilização da memória;
- Requer um elevado número de portas de acesso à cache de dados para suprir as múltiplas unidades funcionais.

Em geral, a escolha do modelo de arquitetura VLIW justifica-se pela necessidade de atingir um desempenho diferenciado com um hardware composto por módulos independentes que, em conjunto, podem ser facilmente controlados e coordenados para a execução paralela de instruções [7].

5. PERSPECTIVAS FUTURAS

Devido a sua natureza não convencional, até recentemente, arquiteturas VLIW têm sido a exceção e não a regra. Entretanto, como processadores escalares estão atingindo o limite de seu potencial de desempenho [3], os projetistas estão começando a focar mais em arquiteturas VLIW como o próximo estágio na evolução da arquitetura de computadores.

Nos últimos anos, diversos centros de pesquisa vêm realizando estudos visando o desenvolvimento de novas técnicas para arquiteturas VLIW. São pesquisas na área de compiladores (desenvolvimento de novas técnicas de otimizacao e escalonamento que aumentem a capacidade de exploração de ILP), na arquitetura (extensões, mecanismos de busca e decodificação de instruções) e finalmente pesquisas que visam investigar a solução dos problemas inerentes à utilização de palavras longas de instrução e hardware extremamente simples (e.g. incompatibilidade de código objeto) [1].

6. CONCLUSÕES

Este trabalho apresentou diversos conceitos referentes à organização do *hardware* e suporte de compilação para tornar o uso da arquitetura VLIW viável. Foi abordado o fato de que a utilização de alguns conceitos da arquitetura VLIW, para manter certo grau de complexidade, aliado a mecanismos mais poderosos tanto de *hardware* quanto de *software*, permitem a resolução de diversos problemas atrelados a palavras longas de instrução.

Apesar dessa arquitetura não fazer parte das principais máquinas comercializadas atualmente, pelos diversos problemas e restrições apresentadas no uso de palavras longas de instrução, centros de pesquisa têm realizado esforços para superar esses problemas e alavancar um maior interesse nessa arquitetura.

7. REFERÊNCIAS

[1] S. A. Ito. Uma alternativa para a exploração de paralelismo a nível de instrução. Relatório técnico, Universidade Federal do Rio Grande do Sul, 2000.

[2] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. IEEE Trans. Computers, 30(7):478{490, 1981.

[3] J.P. Grossman, Compiler and Architectural Techniques for Improving the Effectiveness of VLIW Compilation, International Conference on Computer Design, 2000.

[4] J. L. Hennessy and D. A. Patterson. Arquitetura de Computadores: Uma Abordagem Quantitativa. Rio de Janeiro: Elsevier, 4 ed., 2008.

[5] G. Doshi, Understanding The IA-64 Architecture. Relatório técnico. Agosto, 1999.

[6] A. Klaiber, The Technology Behind Crusoe Processors. Relatório técnico. Janeiro, 2000.

[7] E. D. Moreno; F. D. Pereira. Arquitetura e Desempenho de um Criptoprocessador VLIW. XI Iberchip Workshop: Anais do Iberchip, 2005.