Paralelismo em Nível de Instrução: Abordagem por Software

Grupo:

André Lauar Sampaio Meirelles RA: 008072
Carlos Eduardo Fernandes RA: 008275
Gustavo Sverzut Barbieri RA: 008849
Ivens Prates Telles Alves RA: 008908

Sumário

1 Introdução

Paralelismo em nível de instruções é uma técnica que vem sendo desenvolvida já desde a década de 40. Basicamente existem duas maneiras de se aumentar a performance de um processador: uma tecnologia de semicondutores mais rápida ou um processamento paralelo eficiente. Neste trabalho será descrita uma forma de processamento paralelo em um único processador. Os processadores em questão são os da arquitetura VLIW(*Very Long Instruction Word*), e as técnicas de paralelismo estudadas serão duas, ambas tendo uma abordagem por *software*: *software pipelining* e *trace scheduling*.

2 Histórico Evolutivo da Arquitetura

A tecnologia VLIW é nasceu em meados dos anos 70, como um desenvolvimento natural do microcódigo horizontal. Esse microcódigo horizontal era usado para gerar os sinais de controle nas primeiras CPU[1]s, que possuiam instruções complexas de serem executadas. Eles eram guardados em uma memória *read-only* de alta velocidade.

Em contraste a essa complexidade de microcódigo horizontal surgiu o conceito de microcódigo vertical, que usa pequenas instruções ou séries de microinstruções para gerar os sinais de controle.

Em 1979, Joseph Fisher, um dos pioneiros do VLIW, estava trabalhando em microcódigo horizontal e se viu com sérias dificuldades para manter e usar essa tecnologia para uma arquitetura de microcódigo de 64 bits e resolveu procurar meios de gerar um microcódigo longo a partir de vários pequenos. Isso deu origem ao *trace scheduling*. Fisher percebeu que esse método podia ser usado por um compilador para gerar código para os processadores VLIW a partir de um código de programa seqüencial. Essa descoberta levou ao desenvolvimento do processador ELI-512 e do compilador Bulldog.

No início as máquinas equipadas com processadores VLIW eram destinadas ao meio científico e da engenharia, os quais possuiam aplicações com padrões bem conhecidos.

Em 1984 foram fundadas duas companhias que iriam produzir os processadores VLIW, a Multiflow, de Fisher e associados e a Cydrome, de Bob Rau, outro pioneiro da tecnologia. Em 1987 a Cydrome lançou seu primeiro produto, o Cidra 5, processador de 256 bits, que possuía suporte a *software pipelining*. A Multiflow lançou no mesmo ano o Trace-200, seguido pelo Trace-300 no ano seguinte e o Trace-500 em 1990. Infelizmente ambas empresas foram comercialmente infelizes. Apesar disso a contribuição tecnológica de ambas ainda está presente hoje. O Intel[2] Itanium[3], moderno processador de 64 bits usa muitos conceitos do Cidra 5. Outros notáveis processadores do presente que implementam essas tecnologias são Transmeta[4] Efficeon[5], Philips[6] TriMedia[7] e Texas Instruments[8] TMS320C62x[9].

3 Princípio Geral de Funcionamento

Na arquitetura do VLIW, o compilador é o responsável em passar para o hardware a informação sobre as instruções que podem executar em paralelo. Na verdade, o processador precisa de 3 informações, e passar essas 3 informações são as funções que devem ser cumpridas:

- As dependências entre as operações devem ser determinadas.
- As operações que são independentes de outras operações que ainda não foram executadas devem ser determinadas.

[1] CPU: *Central Processing Unit* ou Unidade Central de Processamento
[2] `http://www.intel.com`
[3] `http://www.itel.com/itanium/`
[4] `http://www.transmeta.com`
[5] `http://www.transmeta.com/efficeon/`
[6] `http://www.philips.com`
[7] `http://www.semiconductors.philips.com/products/nexperia/media_processing/`
[8] `http://www.texasinstruments.com`
[9] `http://dspvillage.ti.com/docs/catalog/generation/overview.jhtml?templateId=5154&path=templatedata/cm/dspovw/data/c62_ovw&familyId=326`

- Essas operações independentes devem ser agendadas para execução em algum momento em particular, em alguma unidade funcional em específico e deve ser definido um registrador no qual o resultado será colocado.

No caso dos processadores VLIW, existem dois conceitos importantes que devem ser entendidos bem, que são o de operação e instrução. Uma operação é uma unidade de processamento, como uma soma, um *load* ou um *store*. Uma instrução é o conjunto dessas operações que devem ser executadas simultaneamente. O compilador deve escolher quais são as operações que formam cada instrução. Esse processo é o escalonamento das operações. A ordem das operações dentro de cada instrução vai definir cada unidade funcional na qual cada operação vai executar.

O paralelismo no VLIW então advém de dois fatos: cada instrução que é passada para o processador e que ele executa, na verdade são formadas de um número definido máximo de operações, que para os processadores seqüenciais são idênticas às instruções que esses executam. Além das instruções multi-operacionais (Multi-Ops) os processadores VLIW ainda implementam *pipelining*, geralmente em 4 estágios (busca, decodificação, execução e escrita).

O escalonamento das operações é feito em um processo chamado de especulativo. O compilador determina as operações de cada instrução muitas vezes sem saber se elas realmente deveriam ser executadas,como no caso de um desvio. O processador não tem como identificar se a operação deveria ser executada, mas ele provê meios para que os efeitos da mesma sejam desfeitos caso o programa chegue em um estado que indique que a operação deva ser anulada. Basicamente são necessários alguns registradores extras, ler algumas instruções a mais e reprimir a geração de algumas exceções de erro que porventura tenham ocorrido.

Independente do tipo de arquitetura envolvida, algumas operações em comum devem ser feitas, seja em tempo de execução (super-escalar) ou em tempo de compilação (VLIW). O programa deve ser analisado para que as dependências sejam identificadas, e o ponto no tempo em que as operações são independentes das outras que nesse momento ainda não foram executadas. Devem ser feitos o escalonamento das operações e a alocação de registradores. Para que seja feito o escalonamento de maneira especulativa é necessário que se faça uma previsão de desvio, que vai indicar qual o possível caminho que será percorrido entre as opções existentes.

Aumentar o grau de paralelismo de um processador envolve muitas variáveis, e geralmente é complicado achar um equilíbrio entre elas de modo a gerar uma performance melhor. Um meio de baixo custo é acrescentando mais portas para *pipelining* nas unidades funcionais. Além de possuir um baixo custo essa solução pode duplicar ou triplicar o grau de paralelismo possível, mas em contrapartida existem pontos negativos sérios a serem considerados. Em geral o *clock* máximo sofrerá perdas e o gerenciamento da memória se torna mais complicado. Outro fator é que aumentar o grau de *pipelining* acrescenta atrasos(*delays*) na execução de operações individuais. Outras abordagens para o aumento de performance e paralelismo existem e junto com elas existem prós e contras que devem ser considerados.

4 Escalonamento de Instruções

Nesta seção estão descritos alguns algoritmos de escalonamento de operações, que irão formar as instruções VLIW a serem executadas pelo processador. O algoritmo predominante nesse campo é o *tace-schedule* de Fisher, mas ao longo dos anos surgiram dezenas de outros algoritmos. A grande maioria deles trata do desenrolar de laços ou *"loop unrolling"*.

Os laços recebem atenção especial em um programa pois são eles, em geral, que determinam o tempo de execução do mesmo. Em geral o estudo de paralelismo para outros trechos de código produz pouco efeito. O desenrolar de laços é uma técnica que tenta executar iterações diferentes do laço ao mesmo tempo.

4.1 Trace Scheduling - O Compilador Multiflow

O compilador *Multiflow* usa o *trace schedule* para achar e explorar o paralelismo através de blocos básicos de código. Esse compilador gera código para processadores que iniciam até 28 operações por ciclo e conseguem manter mais de 50 operações em execução ao mesmo tempo.

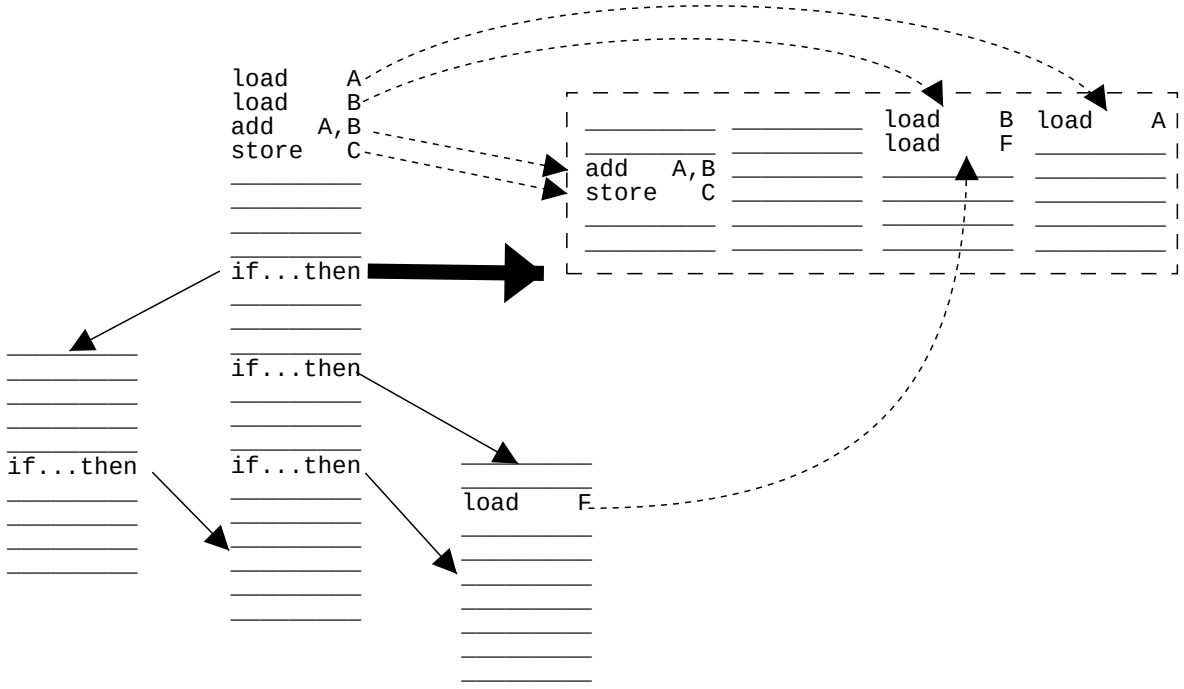


Figura 1: Escalonamento de código entre blocos básicos

O algoritmo recebe o grafo de fluxo do programa, que é produzido na etapa de otimização e tradução da linguagem de programação para uma linguagem intermediária. É feita uma estimativa, a partir do grafo, de quantas vezes uma operação será executada. Basicamente o algoritmo de escalonamento funciona da seguinte maneira:

1. Selecionar uma sequência de operações a serem agendadas juntas. Essa sequência é chamada de *trace*. Esses *traces* tem um limite de comprimento que é definido por vários fatores, entre os mais importantes estão os limites modulares (entrada/retorno), limites de *loop* e código já agendado. (Figuras 1 e 2)
2. Remover esse *trace* do grafo de fluxo, e repassá-lo para o agendador de instruções.(Figura 2)
3. Quando a instrução tiver sido formada pelo agendador de instruções, passar essa instrução para o grafo de fluxo, substituindo as operações que estavam originalmente no *trace* como um bloco único, indivisível (Figura 3). Possivelmente será necessário fazer cópias das operações para recolocá-las, afim de não ultrapassar os limites definidos no escalonamento. Código de compensação também pode ser necessário. (Figura 4)
4. Re-iterar até que todas as operações tenham sido incluídas em algum *trace* e todos os *traces* tenham sido substituídos por um agendamento.(Figura 5)
5. Selecionar a melhor ordem linear para o código e emití-lo.

O agendador de instruções recebe um *trace* em cada chamada. Ele constrói um grafo de precedência de dados (GPD) para representar restrições de precedência de dados na ordem de execução.

A seguir uma explicação mais detalhada de cada fase.

4.1.1 Expectativas

As expectativas se referem às estimativas de execução a serem calculadas a partir do grafo de fluxo. Elas são calculadas a partir do número de iterações de um laço e probabilidades de desvios condicionais serem tomados. São usadas as seguintes regras:

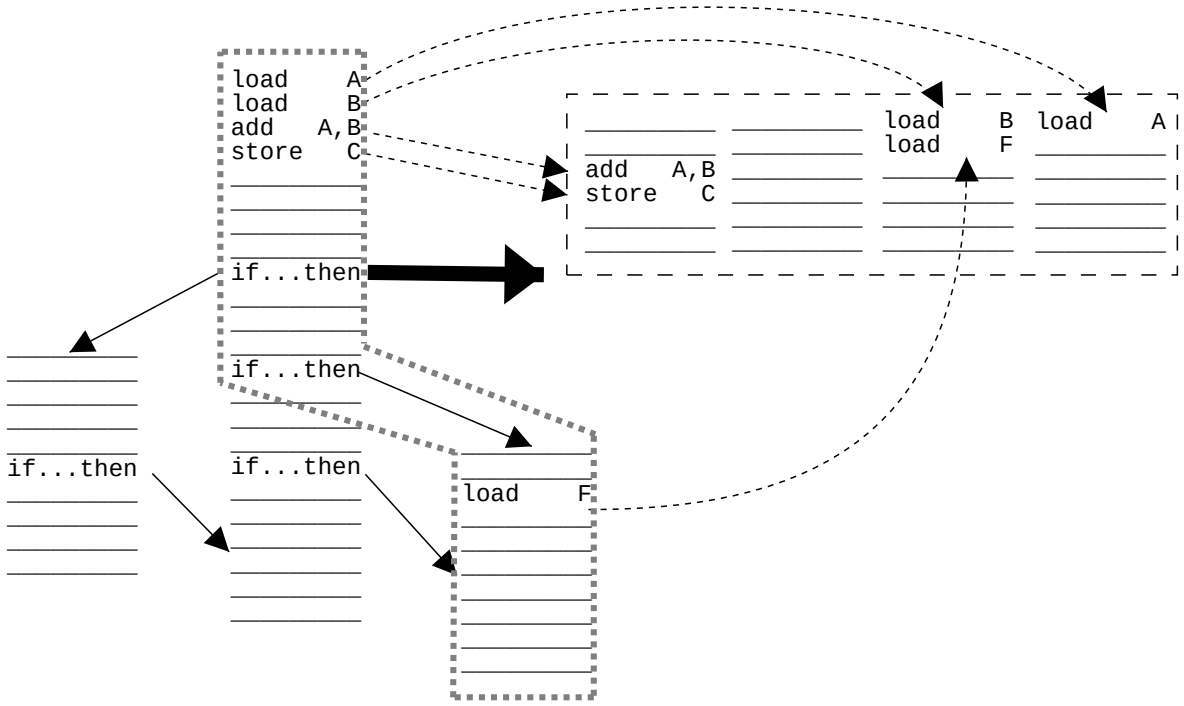


Figura 2: Selecionando um *trace* e uma organização no mesmo

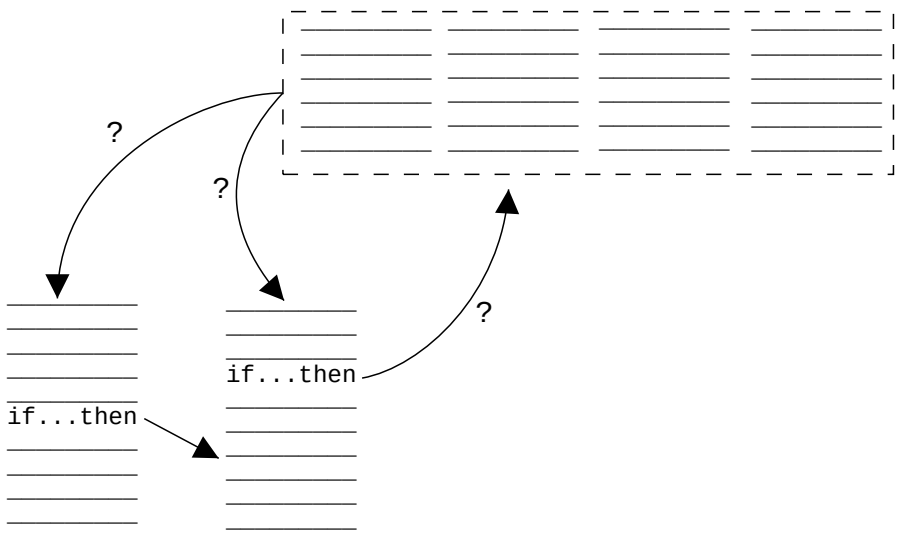


Figura 3: Substituindo o *trace* pelo agendamento e analisando casos de *splits* e *joins*

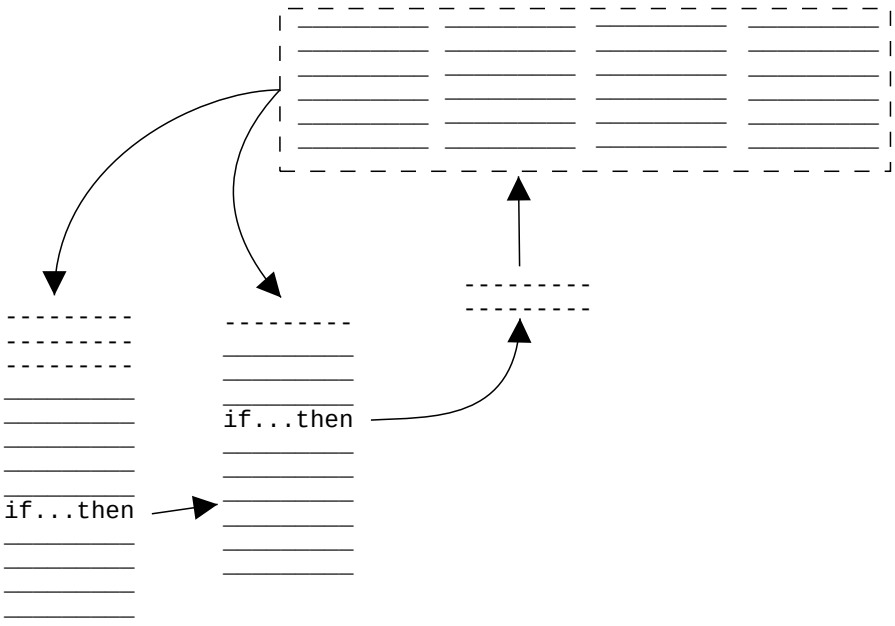


Figura 4: Gerando código de compensação para resolver diferenças de *splits* e *joins*

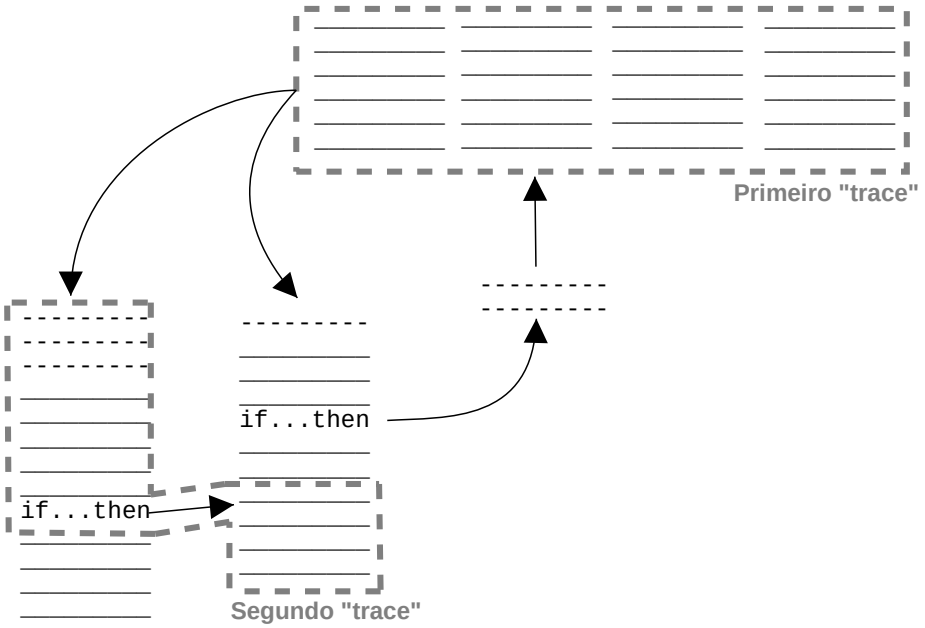


Figura 5: Iterar mais uma vez, seguindo prioridades sucessivas

Se a operação O for uma chamada a rotina **então:**
expect[O] = $\frac{1.0}{numero\ de\ chamadas}$
Se a operação O não for o cabeçalho de um loop **então:**
expect[O] = $\sum_i^P (prob[i] \times expect[i])$ *// P é o número de predecessores da operação, // desde que não sejam cabeçalho de um laço*

No qual *expect[i]* é a expectativa do predecessor i; *prob[i]* é a probabilidade de atravessar a distância desde predecessor i até a operação O.

Se a operação O for o cabeçalho de um loop **então:**
expect[O] = $iter\_count \times \sum_i^{LE} (prob[i] \times expect[i])$ *// LE é o número de entradas no laço*

No qual *iter_count* é o número de iterações esperado para o laço; *prob[i]* é a probabilidade de ir desde a entrada do laço i até a operação O e *expect[i]* é a expectativa da entrada i no laço.

Uma entrada no laço é definida como uma operação que não está no laço mas tem um sucessor que está. Quando é calculada a expectativa, os laços irredutíveis são tratados como redutíveis. Nas fórmulas acima, se uma operação não é um cabeçalho de laço, são ignoradas todas as entradas de laços que são predecessoras da operação, e se a operação é um cabeçalho de laço todas as entradas para aquele laço são tratadas como predecessores.

As probabilidades dos desvios são obtidas ou de uma base de dados coletada em uma execução prévia do programa, ou de diretrizes do usuário ou então simplesmente por heurística, partindo do princípio que tomar um desvio tem probabilidade de 50% desde que não seja um laço ou um *exit*. A probabilidade de sair de um laço é definida como $\frac{1}{iter\_count}$, onde *iter_count* tem o mesmo significado das fórmulas acima.

4.1.2 Formando um *Trace*

A partir daqui os *traces* serão chamados de blocos de operações. A formação de um bloco se inicia selecionando entre as operações ainda não agendadas, aquela que tem a maior expectativa. Esse passo é que desencadeia a criação de um bloco, e a partir dele são adotadas outras medidas heurísticas para que sejam escolhidas as outras operações que irão compor o bloco. O crescimento pode se dar avançando ou retrocedendo sobre o grafo de fluxo, escolhendo um sucessor ou predecessor respectivamente. Se nenhum sucessor ou predecessor satisfaz as condições heurísticas o bloco é fechado. O bloco também é encerrado se a busca por novas operações levar a outras que já foram agendadas ou a alguma que já esteja no bloco. Os blocos também respeitam o limite dos laços, não passando por dentro dos mesmos. Por fim, existe o limite de tamanho máximo dos blocos (*max_trace_length*).

As heurísticas usadas na escolha de operações são definidas em termos de bordas[10] entre operações no grafo de fluxos. Os mesmo critérios são usados para determinar se uma borda pode ser acrescentada ao bloco, independente da direção de crescimento do mesmo. A heurística é aplicada ao *pred* (predecessor) e ao *suc* (sucessor). Se o crescimento se dá avançando, *pred* já faz parte do bloco, se o avanço é retrocedendo, *suc* já faz parte.

Existem muitas heurísticas, duas serão brevemente descritas a seguir:

- ***Mutualy-most-likely* (Mutuamente-mais-provável):**

  A borda de *pred* para *suc* tem a maior probabilidade de todas as saídas de *pred* (i.e, se a posição atual é *pred*, é mais provável que se siga para *suc*).

  A borda de *pred* para *suc* contribui com a maior expectativa para *suc* de todos os predecessores de *suc* (i.e, se a posição atual é *suc* é muito provável que a posição anterior era *pred*).

- ***No compensation* (Sem compensação):**

[10] As bordas são a divisa entre um grupo de operações que estão dentro do bloco e as que não estão, e dependem de como o grafo está sendo percorrido, se para frente ou para trás

Essa abordagem é usada para evitar o uso de código de compensação, inserido pelo compilador para manter a lógica e a corretude do programa após as modificações que ele faz. Ela restringe os blocos a opções mais básicas.

Nessa heurística não é desejado que seja necessário o acréscimo de código de compensação após o agendamento de instrução. Se nem *pred* nem *suc* são um *rejoin* ou um *split* então a borda está correta. *Rejoins* e *splits* necessitam de atenção especial:

> Se *suc* é um *rejoin* (i.e, tem múltiplos predecessores) então encerre o bloco; compensação seria necessária se *suc* está posicionado antes de *pred* no agendamento.
>
> Se *pred* for um *rejoin* então a borda está OK.
>
> Se *suc* for um split (i.e, se ele tiver múltiplos sucessores) então encerre o bloco; código de compensação seria necessário se *suc* está posicionado antes de *pred* no agendamento.
>
> Se *pred* for um *split* então a borda está OK.

4.1.3 Código de Compensação

Após que um bloco de operações é criado, o agendador de blocos o passa ao agendador de instruções. O agendador de instruções retorna um agendamento pronto para ser executado pelo processador, mas antes disso é necessário que algumas verificações sejam feitas.

As operações que foram movidas devem ser analisadas para que sejam feitas eventuais cópias. Os *splits* e *joins* que foram produzidos devem ser tratados. *Splits* são desvios que direcionam a execução para fora de um bloco. Os *joins* são desvios que caem dentro do bloco. São esses *splits* e *joins* que determinam os limites básicos dos blocos no grafo de fluxo.

Um *split* possui mais de um sucessor. Um exemplo em que ocorre um *split* e é necessário a cópia de operação é o seguinte: Supondo uma sequência de operações que chegue a um desvio condicional. No grafo irão aparecer dois possíveis caminhos a partir dessa operação. Supondo que o teste condicional seja a operação B e que A seja precedente de B. Se B tem como sucessores as operações C e X (qual será o caminho correto depende do resultado da comparação), então o agendamento resultante provavelmente se inicia em B, e a partir dele existem dois caminhos possíveis, um que passa por C e um que passa por X, mas A é precedente de B, então deve ocorrer uma cópia de A para o caminho que passa por X (imaginando que X seja uma saída).

Um cenário semelhante pode ser formulado para um *join* que necessite da criação de uma cópia, só lembrando que nesse caso, a operação *join* possui múltiplos precedentes e é resultado de um desvio direcionado a ela.

4.1.4 Instruction Scheduler (Agendador de Instruções)

Nessa seção serão abordados tópicos gerais do funcionamento dessa etapa do processo do *trace schedule*. Aqui muitos algoritmos são implementados e o exemplo usado é o algoritmo BUG(*Bottom Up Greedy*).

O agendador de instruções transforma um bloco gerado pelo escalonador de blocos (seção anterior) em um grupo de longas instruções. Nessa etapa são definidos e alocados os registradores para os operandos, uma unidade funcional para as operações e ocorre a inserção das operações de um bloco em uma instrução que será passada ao processador.

Três etapas são cumpridas:

1. Construir um grafo de precedência de dados[11] (GPD) a partir do bloco recebido.
2. Atravessar o GPD e designar operações a unidades funcionais e valores a bancos de registradores.
3. Criar o agendamento alocando registradores. Agrupar referências à memória para minimizar conflitos

Ao designar unidades funcionais e bancos de registradores, é interessante agrupar as designações de maneira que vizinhos no GPD continuem vizinhos dentro do processador. Entretanto apenas com o propagação por toda a máquina e usando todas as unidades funcionais a performance ótima pode ser alcançada.

[11] *data precedence graph* (DPG)

Pode sair muito caro ter que esparramar todos esses dados pelo processador, já que cada unidade funcional tem o seu próprio banco de registradores. Se um operando não estiver no banco de registros local, é necessária uma transferência de registrador. Isso adiciona latência ao cálculo e consome uma unidade funcional, impedindo que outra operação seja executada. Além disso, escrever em um registrador remoto consome barramento, que deveria estar sendo usado para transferências de dados da memória.

O algoritmo BUG apresentada uma possível solução para esses problemas de agrupamento. Ele é um algoritmo complexo e se propõe a minimizar a latência na transferência de dados. Não serão expostos detalhes de seu funcionamento, mas ele foi inicialmente implementado no compilador Bulldog, desenvolvido por Fisher.

4.2 Outras Abordagens de Paralelismo

Além do *trace-schedule* existem outras abordagens para expor o paralelismo de um código. A grande maioria se preocupa em tratar a execução de laços, "desenrolá-los", já que são eles que consomem a grande maioria do tempo de execução. O *software pipelining* provê inúmeras abordagens para esse tratamento de laços e dezenas de algoritmos estão disponíveis nos meios acadêmicos e de pesquisa privada. Outras possibilidades incluem o *block-schedule*.

Essas diferentes abordagens serão apenas citadas, e referências para as mesmas são encontradas na bibliografia.

Pequena descrição sobre *trace-schedule* vide [RF92] e [LMK$^+$92] e Arquitetura VLIW e outras abordagens: [Com], [Mat97] e [Gro].

5 Mecanismos de Previsão de Desvios

5.1 Motivação

Estruturas de desvio como *if...then...else* possibilitam ao programador de alto nível criar estruturas de controle muito poderosas, contudo o uso de desvios pode reduzir substancialmente as vantagens estabelecidas pela técnica de *pipeline*.

Para que o desempenho do processador *pipelined* possua um bom desempenho é essencial que o primeiro estágio receba um fluxo contínuo de instruções, entretanto desvios provocam constantes interrupções nesse fornecimento. Isto é agravado pelo fato de que quando ocorre um desvio, somente após 3 ciclos de *clock* é que o estágio de busca percebe que trata-se de uma operação de desvio. Enquanto essa constatação não ocorre, duas novas operações são inseridas no *pipe* (Supondo que este é formado por 5 estágios: Busca instrução, Identifica Operação, Executa Operação, Acesso a Memória e Escrita no Banco de Registradores).

Quando detecta-se que o caminho tomado no desvio é incorreto, é necessário retirar do *pipe* as operações que entraram porém não deveriam ser executadas e passar a executar as operações corretas. Este é um processo caro, e diminui em muito a eficiência do hardware.

Diante deste quadro, técnicas são adotadas visando diminuir o custo das instruções de desvio. A arquitetura VLIW utiliza uma abordagem por software, isto é, em tempo de compilação. A seguir, descrevemos como processadores VLIW abordam a questão de desvios.

5.2 Delayed Branch

Para evitar os custos da operação de desvio, esta técnica tenta reorganizar o código original, mantendo a mesma semântica.

Esta técnica baseia-se no fato de que uma operação de desvio condicional possui um ciclo de *clock* de retardo, fazendo com que durante a fase de busca da operação de ramificação a operação adjacente a operação de destino seja executada. Veja o exemplo na Figura 6.

Com a terceira operação (`if ( X == W ) goto 7`) possuindo um atraso de 1 ciclo de *clock*, a quarta operação (`W--`) também seria executada, então teríamos a seqüência de execução: 1, 2, 3, 4, 7. O que não está correto pois `Z` conteria o valor de `W` decrementado.

```
1.   X = Y;
2.   Y--;
3.   if ( X == W )
        goto 7;
4.   W--;
5.   printf( "Teste" );
6.   printf( "Outro Teste" );
7.   Z = W;
```

Figura 6: Código genérico para ilustrar o *delayed branch*

```
1.   X = Y;
2.   if ( X == W )
        goto 7;
3.   Y--;          // Note que esta operação estava antes do 'branch'
4.   W--;
5.   printf( "Teste" );
6.   printf( "Outro Teste" );
7.   Z = W;
```

Figura 7: Código da Figura 6 modificado pelo *delayed branch*.

A técnica *Delayed Branch* irá pegar o código original e irá modificar a ordem de execução, colocando depois do *branch* uma operação independente do desvio condicional, ou seja, uma operação que está acima dele, no caso `Y--`. Deste modo, o código ficaria como na Figura 7.

Então teríamos a sequência de execução: 1, 2, 3, 7. Apesar, de considerarmos o atraso e a terceira operação ser executada desnecessariamente (pois seu valor não é utilizado), temos a seqüência de execução correta.

Uma outra opção a esta ação, seria a utilização de *NOP*s (*No Operation*) porém poderíamos ter um código muito maior, e sem nenhuma otimização [Cam99].

5.3 Branch Folding

Branch Folding é uma técnica de predição que visa diminuir ou eliminar as perdas por desvios condicionais.

Ela consiste em incluir na operação atual o endereço de sua sucessora.

No caso de um comando de desvio incondicional, o endereço almejado fica armazenado na instrução que precede o comando de desvio.

No caso de desvios condicionais, é função do compilador (por isso abordagem por *software*) determinar qual a operação seguinte. Isto é feito através de análise estatística para determinar qual a operação com maior probabilidade de ser executada. Após escolhida, o endereço da operação é colocado antecedendo a operação de desvio. Além do endereço que ficou embutido na operação de desvio, o processador também armazena na memória cache o endereço da outra operação alternativa. Este endereço, denominado contador de programa alternativo será usado se a previsão feita pelo compilador estiver errada.

Quando uma operação de desvio entra no *pipeline* a operação escolhida pelo compilador entra no *pipeline* no estágio seguinte. Quando a operação de desvio chega ao estágio 3, determina-se se o caminho escolhido é o correto. Se estivermos no caminho certo, então as operações no *pipeline* continuam a ser executadas. Caso esteja incorreto, as instruções em progresso ao longo dos três estágios são descartadas, o contador alternativo torna-se o corrente, e as instruções da outra direção são introduzidas no *pipe*.

Na presença de múltiplos desvios, a movimentação das instruções que alteram os indicadores de condição, torna-se uma tarefa de difícil realização, o que limita o alcance desta técnica. [Cam99].

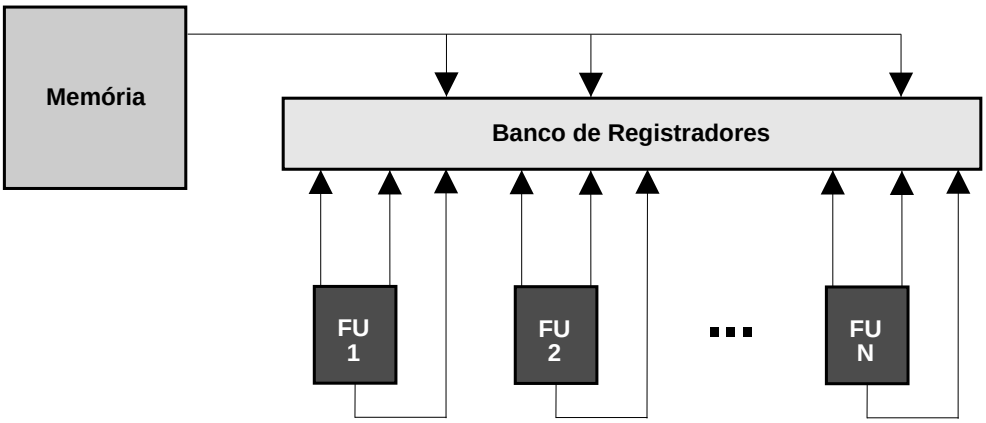


Figura 8: Esquema de processador VLIW.

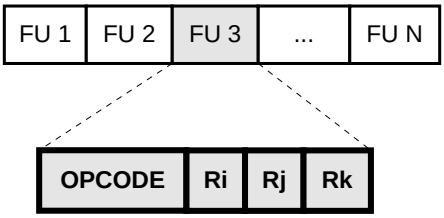


Figura 9: Esquema de instrução VLIW.

6 Emissão de instruções

6.1 Organização da arquitetura VLIW

A arquitetura compartilha um grande banco de registradores entre várias unidades funcionais, responsáveis pelo processamento das instruções.

As instruções, localizadas em longas palavras de centenas de bits na memória de instruções, contém diversas operações que serão executadas paralelamente em uma unidade funcional específica, por este motivo a arquitetura VLIW recebe a denominação de SIMOMD (*Single Instruction, Multiple Operation, Multiple Data*) ou simplificadamente MultiOp.

Esta arquitetura apesar de apresentar um hardware relativamente simples, exige um compilador extremamente eficiente para escalonar as operações nas unidades funcionais corretas, mantendo a semântica correta, e evitando conflitos estruturais, conflitos de dados ou de controle.

6.2 Múltiplas Instruções

O uso de longas palavras permite a arquitetura VLIW utilizar a emissão de múltiplas instruções. Isto permite a realização do *pipeline*.

Além de executar diversas instruções através do uso de grandes palavras de instruções, a arquitetura VLIW também possui emissão múltipla de operações através do uso das diversas unidades funcionais. Deste modo, o paralelismo da arquitetura VLIW se dá não somente através do uso de *pipeline*, mas o paralelismo se dá através do trabalho simultâneo de várias unidades funcionais e do *pipeline*.

Utilizando um *pipeline* de 4 estágios (busca, decodificação, execução e escrita) temos que a utilização das unidades funcionais se dá apenas no estágio de execução, como mostra a Figura 10.

Estágios do *pipeline*:

1. **Busca:** A busca de instruções na arquitetura VLIW é simples.

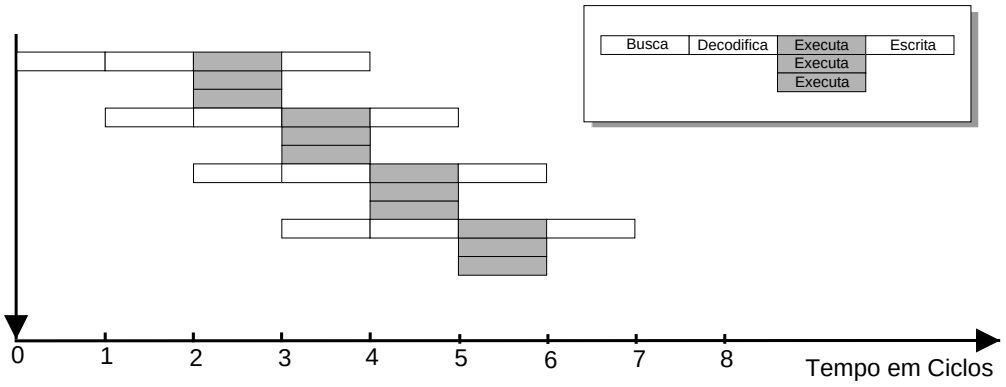


Figura 10: Gráfico de funcionamento do *pipeline* em processadores VLIW.

2. **Decodificação:** A decodificação das operações também é simples pois as operações dentro da instrução possuem posições pré-definidas. Sendo assim, lê se uma operação, identifica-se seu opcode e envia-o a unidade funcional específica.

3. **Execução:** Neste estágio, temos a concretização das diversas operações contidas em cada instrução. Nesta etapa ocorre o grande diferencial da arquitetura VLIW, pois ao invés de ter apenas uma operação sendo executada no estágio, teremos várias, dependendo do número de unidades funcionais disponíveis e da compatibilidade das operações.

4. **Escrita:** A escrita na arquitetura também é simples, os resultados vindos do estágio de execução são encaminhados ao banco de registradores ou a memória é acessada dependendo da operação realizada. Estas operações podem ser consideradas simples pois as operações executadas pelas unidades funcionais são independentes uma das outras, isto quer dizer, não existe qualquer tipo de dependência (controle ou dados). Como as unidades funcionais compartilham o banco de registradores (Figura 8), então a escrita no banco de registradores é feita de maneira independente.

Uma exigência da arquitetura VLIW é que exista um elevado no número de portas de acesso à cache, ou seja, uma elevada largura de banda de memória para suprir as diversas unidades funcionais. Este requisito é essencial para a arquitetura pois ele será responsável pela concretização das operações de *load*/*store* na fase de escrita.

O sucesso da arquitetura VLIW está atrelado a um bom compilador que consiga gerenciar todo processo estaticamente, ou seja, durante o processo de compilação. Isto porque na arquitetura VLIW é função do compilador organizar todo o processo de execução de instruções desde a busca por instruções, passando pela decodificação, o pipeline, o funcionamento das diversas unidades funcionais e a escrita de dados.

Esta responsabilidade sobre o compilador, é a causa da simplicidade do hardware de uma arquitetura VLIW, representada na Figura 8.

7 Recursos nos Compiladores para expor e explorar ILP

Para um bom desempenho utilizando paralelismo em nível de instrução é necessário, ou desejável, que as unidades funcionais permaneçam ocupadas a maior parte do tempo. Buscando esse uso mais intenso do hardware, várias técnicas têm sido desenvolvidas tanto em hardware (*bypassing*) como em *software* (Escalonamento estático).

A arquitetura VLIW utiliza escalonamento estático realizado pelo compilador, ou seja, utiliza técnicas de *software* para melhorar o seu desempenho . A seguir apresentamos duas técnicas utilizadas pela arquitetura VLIW para melhor utilizar seus recursos de hardware.

```
for ( i = 0; i < 7; i++ )
{
   a[ i ] = 2.0 * b[ i ];
}
```

Figura 11: Exemplo de um laço a ser otimizado pelo *software pipelining*

```
load   r101,    b( i )
fmul   r101,    2.0,    r101
decr   r200
nop
store  a( i )+, r101
```

Figura 12: Representação em código de máquina do laço na Figura 11.

7.1 Trace Scheduling

Ao se utilizar paralelismo a nível de instrução é necessário preocupar-se com dependências de dados e de controle afim de que os resultados apresentem os resultados desejados. As dependências dentro de um programa é o principal entrave para o paralelismo a nível de instrução. Pensando nisso, Joseph Fisher desenvolveu um método chamado *Trace Scheduling* que tenta solucionar o problema das dependências e viabilizar o paralelismo.

Segundo Fisher, um programa original pode ser organizado em trechos de instruções independentes que serão escalonadas juntas. Para determinar como serão criados e escalonados os trechos, devemos primeiro realizar um primeiro teste com o código não otimizado, afim de colher estatísticas sobre a execução. Com isso calculamos a propabilidade de cada *branch* condicional.

Das informações colhidas com o bloco inicial, criamos um grafo de precedência de dados (DAG ou *Directed Acyclic Graph*), selecionamos dele uma seqüência linear de trechos de dados (*traces*) livres de laços (*"loop free"*).

Com o *trace* pronto, escolhemos o caminho de maior probabilidade de ser executado e então, utilizando de técnicas de realocação de código tais que preservem o resultado final do código como desejado, otimizamos o algoritmo de modo a proporcionar o máximo de paralelismo.

A partir de então, considera-se este caminho otimizado um bloco básico, sendo assim indivisível, e prossegue-se otimizando o próximo caminho mais utilizado.

7.2 Software Pipelining

É uma técnica de escalonamento para processadores com múltiplas unidades funcionais, que busca otimizar o código de um laço explorando o paralelismo entre iterações distintas deste, que não precisam ser adjacentes, pelo contrário podem ser iterações *distantes* da atual.

As dependências de uma dada iteração são satisfeitas pelas instruções distribuídas nas outras iterações do *software pipelined loop*.

Vamos a um exemplo, considere o seguinte código em linguagem C na Figura 11 e seu equivalente em linguagem de máquina na Figura 12

Neste código supõe-se que a unidade de multiplicação de ponto flutuante (*pipelined*) leva 3 ciclos para apresentar o resultado. A Tabela 1 relaciona o ciclo do *clock* e as iterações.

No ciclo 5 as seguintes operações são realizadas em paralelo: `store` (iteração 1), `nop` (iter. 2), `decr` (iter. 3), `fmul` (iter. 4), `load` (iter. 5).

Os ciclos de 5 a 7 tem um padrão repetitivo que pode ser re-escrito como mostra a Figura 13.

Depois deste escalonamento, o que se obtém é a execução *software-pipelined* do laço, que consiste basicamente de 3 partes: prólogo, novo corpo do laço e epílogo.

No exemplo acima, cada iteração do novo laço pode ser executada em apenas um ciclo. Determinar o

Ciclo	Número da Iteração						
	1	2	3	4	5	6	7
1	load						
2	fmul	load					
3	decr	fmul	load				
4	nop	decr	fmul	load			
5	store	nop	decr	fmul	load		
6		store	nop	decr	fmul	load	
7			store	nop	decr	fmul	load
8				store	nop	decr	fmul
9					store	nop	decr
10						store	nop
11							store

Tabela 1: Relação entre iterações e ciclos de *clock*.

```
loop:
   store ( i );
   decr  ( i + 2 );
   fmul  ( i + 3 );
   load  ( i + 4 );
   bc    loop;
   /* Este loop deve ser executado para i=1 a 3 */
```

Figura 13: Código em linguagem de máquina equivalente ao laço das Figuras 11 e 12, porém restrito aos ciclos de 5 a 7.

melhor padrão repetitivo para compor o novo corpo do laço e que seja executado no menor número possível de ciclos é o grande problema na técnica de *software pipelining*.

Essa técnica de escalonamento busca reduzir o intervalo de iniciação entre duas iterações e não a duração de cada iteração, criando um *pipeline* em *software* entre as iterações do laço.

Uma possível dificuldade na implementação do *software pipelining* é que é necessário conhecer previamente o número de iterações a ser realizado. Em nosso exemplo, seriam realizadas 7 iterações. Como conhecíamos esse número, foi possível executar a técnica do *software pipelining* e reduzir as iterações para apenas 3 [Sil].

8 Exemplo de Arquitetura VLIW: Transmeta Efficeon

O processador Transmeta Efficeon (`http://www.transmeta.com/efficeon/`) é o novo processador VLIW compatível com x86 da Transmeta baseado em outro processador da mesma empresa, o Crusoé. Esse processador ainda não foi lançado no mercado, o que torna difícil encontrar dados atuais e confiáveis sobre ele, principalmente relativos a performance e instruções.

As vantagens do Efficeon em relação ao Crusoé se baseiam em um melhor equilíbrio entre performance, energia e custo. Ele adiciona ainda novos recursos, como *HyperTransport*, uso de AGP, DDR de barramento 400MHz, SSE e SSE2, e até 1Mb de Cache L2. Em compensação, o consumo de energia do Efficeon chega a 7W, enquanto o do Crusoé é de apenas 2W. A Transmeta pretende com o Efficeon ter uma linha de processadores mais poderosa, que chegue ao nível de seus grandes concorrentes (Intel e AMD).

O processo de fabricação desse processador é de 130nm, mas a Transmeta pretende para sua próxima geração utilizar um processo de 90nm, visando um aumento de performance e de um menor gasto de energia.

O mercado visado por esse processador é o de portáteis, como *notebooks* e *tablets PCS*, e ainda computadores pessoais silenciosos, sistemas integrados e até servidores.

Os processadores da Transmeta utilizam-se da tecnologia VLIW visando benefícios como baixo custo de energia, tamanho dos processadores e compatibilidade. Como o mercado visado pela Transmeta é o mercado

móvel, um baixo custo de energia se torna essencial para atingir essa tarefa, assim como o tamanho dos processadores. O tamanho do processador também é um fator vital para permitir uma integração completa do processador com outros controladores, como controladores gráficos e de memória. O Efficeon integra todo o *northbridge*, ou seja, basicamente os controladores de CPU, memória e gráficos, e ainda um controlador de *HyperTransport* de 400MHz. Um desenho esquemático está representado na Figura 14.

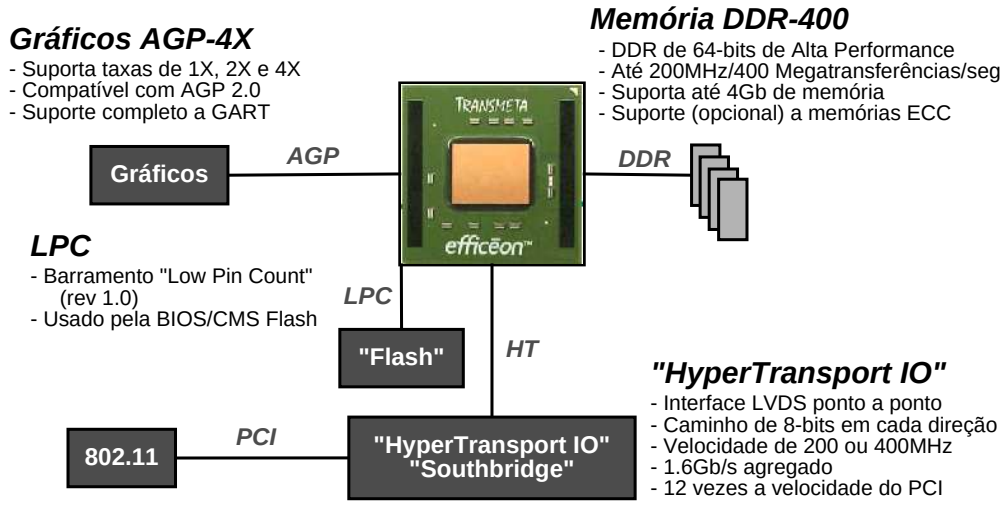


Figura 14: O processador Efficeon tem os controladores AGP, DDR e LPC e HT (*HyperTransport*) integrados.

8.1 A Arquitetura do Processador

O processador Efficeon utiliza uma arquitetura VLIW de oito vias de 32 bits, ou seja, ele é um VLIW de 256 bits. Assim o processador consegue executar até 8 instruções em paralelo, sendo que cada uma delas pode utilizar uma das 11 unidades lógicas existentes. A Figura **??** ilustra uma instrução interna do Efficeon.

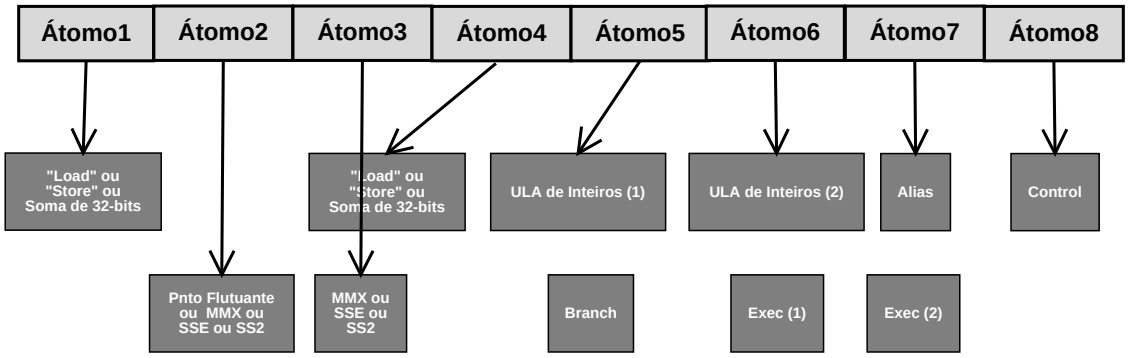


Figura 15: cada via de 32bits é chamada de átomo. Cada átomo pode executar em paralelo uma das 11 unidades lógicas existentes no processador.

Portanto o processador pode, por ciclo, realizar até duas operações *load*/*store*, duas operações em sua ULA, uma operação de *branch*, uma de controle, e até duas operações SSE, por exemplo. Isso apresenta uma altíssimo ganho de performance do processador, aproveitando ao máximo cada ciclo do processador.

8.2 Registradores e Cache

O Efficeon conta com os seguintes registradores:

- 64 registradores inteiros de 32 bits
- 64 registradores de ponto flutuante de 80 bits, com suporte a MMX, SSE e SSE2
- 4 registradores de predicado

O cache do Efficeon é dividido em:

- Cache L2 de 1Mb ECC
- Cache de instrução L1 de 128Kb
- Cache de dados L1 de 64Kb

A Figura 16 ilustra a divisão do cache e a Figura 17 um mapa geral do Efficeon.

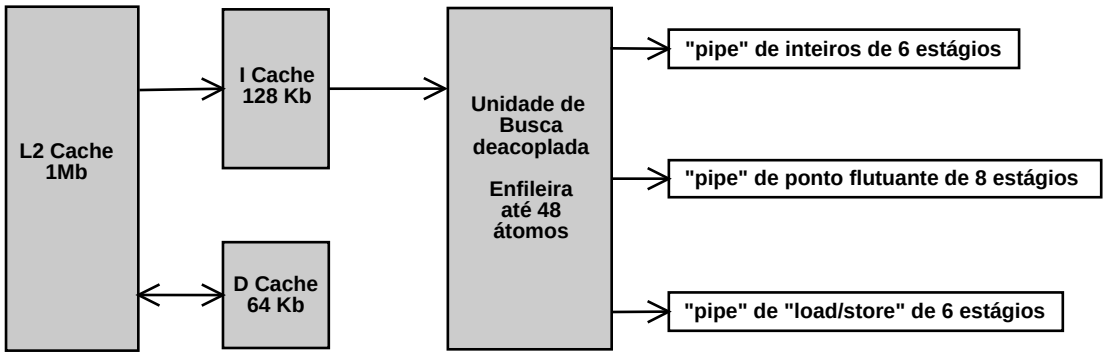


Figura 16: A divisão do cache do Efficeon.

8.3 Advanced Code Morphing

Um dos problemas de se usar a tecnologia VLIW é o fato dela não ser compatível com os binários x86. Para contornar esse problema, a Transmeta fez, via *software* uma camada que faz a "metamorfose" entre o código x86 e o código VLIW. O Efficeon utiliza-se dessa tecnologia, chamada de *Code Morphing*, assim como seu predecessor, o Crusoé. O *Code Morphing* é uma camada de software que dinamicamente traduz instruções x86 para instruções VLIW, dando a impressão ao software de que ele está rodando em um processador x86 nativo. Ou seja, converte as instruções do x86 em MultiOps para o VLIW.

Além disso o processador determina quais instruções irá executar e quando nessa mesma camada se *software*, eliminando transistores lógicos do *hardware* e assim conseguindo menos dissipação de calor. Essa tecnologia armazena o código sequência x86 compilado e recompila-o para o código paralelo VLIW nativo em tempo de execução, sem perda de performance aparente para o usuário final. Ele recompila pequenas partes do código e a guarda no seu cache, e portanto em um laço, as instruções compactadas (paralelizadas) permanecem, se couberem, no cache, não sendo necessário recompactá-las, aumentando a performance.

Podemos considerar então que no caso do Efficeon há um desvio entre a idéia original do VLIW, pois não é possível compilar código nativo VLIW pensando no paralelismo das instruções. Com certeza há uma grande perda de performance do processador nessa conversão, e portanto fazer sistemas dedicados no Efficeon não é uma boa opção. No entanto o fato de existir uma camada de software entre o binário e o núcleo permite uma compatibilidade enorme do processador. Por exemplo, a Transmeta poderia fazer o *code morphing* reconhecer o código de PowerPC, ou até de Sparc.

O *Advanced Code Morphing* do Efficeon difere do do Crusoé no uso de um VLIW de 256 bits, sendo mais eficiente que seu predecessor, que utilizava 128 bits.

8.4 *Enhanced Long Run* - Uma tecnologia de gerenciamento dinâmico de Energia e Temperatura

Essa tecnologia permite que o Efficeon ajuste continuamente a freqüência operacional de seu processador e sua voltagem, de acordo com os requisitos dos aplicativos que estão rodando naquele instante. Essa tecnologia consegue fazer esses ajustes centenas de vezes por segundo, o que permite aumentar o tempo útil de bateria do computador. O *Enhanced Long Run* é mais eficiente que os *"cycle clock throttling power management schemes"* presentes no mercado. Infelizmente a Transmeta ainda não anunciou as vantagens do *Enhanced Long Run* quanto ao *Long Run*, utilizado no Crusoé.

8.5 Conclusão Sobre a Arquitetura

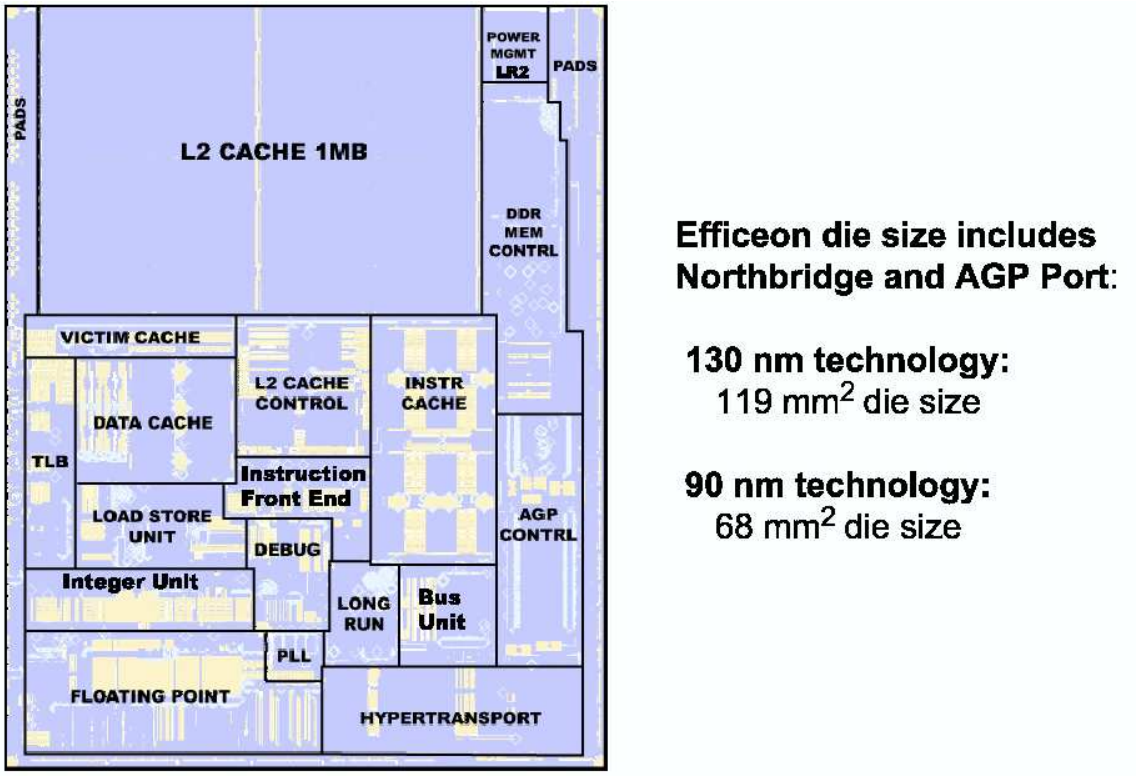


Figura 17: O mapa do processador Efficeon.*(Imagem obtida de* `http://www.transmeta.com`*)*

Podemos concluir que a maior vantagem aparente na especificação da arquitetura VLIW do Efficeon é seu tamanho reduzido. Isso possibilitou a integração não só de toda a *northbridge* no processador, mas também do *HyperTransport*, do *Enhanced Long Run* entre outros, assim como um menor consumo de energia e conseqüentemente menos calor. Outra grande vantagem aparente é o *Advanced Code Morphing*, que possibilita o uso desse processador para binários x86, e quem sabe um dia para outras plataformas, sendo uma incrível inovação da Transmeta.

8.6 Análise de desempenho

Para análise de desempenho, utilizaremos o Processador Transmeta TM8000, que foi o divulgado como exemplo pela própria Transmeta. Como o processador ainda não está no mercado, não existe outra fonte disponível de análise de desempenho desse processador. Portanto note que utilizaremos dados de desempenho

providos pela própria Transmeta, o que nos leva a pensar que esse estudo infelizmente pode ser muito mais parcial que o aceitável.

8.6.1 Análise de Desempenho: Números Inteiros

O tipo mais básico de *benchmarks* é o de cálculo de inteiros. A Transmeta utilizou três tipos de testes de criptografia para analisar o Efficeon em relação ao Pentium 4 e ao Centrino (o *Pentium 4-Mobile*), dividindo o resultado pelo *clock* da máquina. Podemos perceber que o Efficeon é mais eficiente que seus concorrentes na utilização de cada ciclo, provavelmente por que da utilização de suas 8 vias graças ao VLIW. Infelizmente para o usuário final isso não é tão importante na prática. O gráfico comparativo de desempenho para números inteiros pode ser visto na Figura 18.

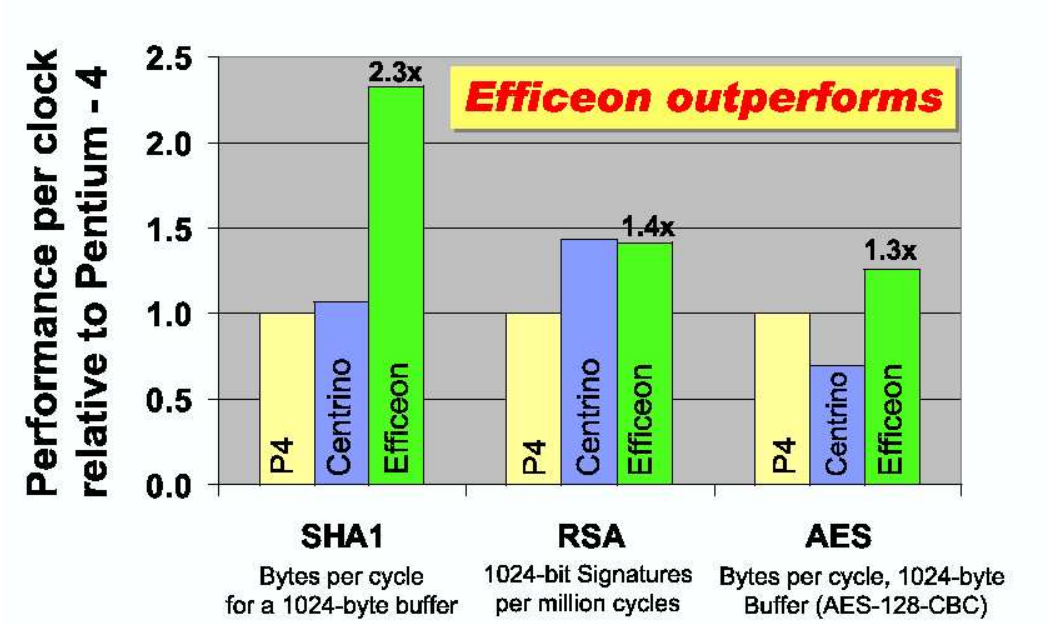


Figura 18: Comparação de desempenho para números inteiros, por *clock*.(*Gráfico obtido de* `http://www.transmeta.com`)

Nos próximos *benchmarks* a Transmeta impôs um limite de 7W ao processador, ou seja, o limite suposto que um *laptop* pode rodar sem um *cooler*. A Transmeta fez isso para poder comparar o processador baseado no consumo de energia, e não somente no *clock*. Ou seja, seria o mesmo que pegar a performance e dividir pela energia dissipada, para assim poder comparar com justiça os processadores. Portanto só teremos comparações entre o Efficeon e o Centrino.

8.6.2 Análise de Desempenho: Ponto Flutuante

Como pode ser visto na Figura 19, a 7W, um Efficeon de 1100MHz consegue ultrapassar em aproximadamente 30% um centrino de 900MHz. Fazendo uma estimativa aproximada temos que o *clock* do Efficeon utilizado é 22% maior que o do Centrino. Portanto mesmo se o Centrino supostamente tivesse 1100MHz, o Efficeon seria 8% mais rápido.

8.6.3 Análise de Desempenho: Performance Geral do Sistema

Os *benchmarks* que melhor representam o comportamento real do processador, são os que analisam o sistema como um todo, também conhecidos como *benchmark suites*. No gráfico da Figura 20 a Transmeta utilizou

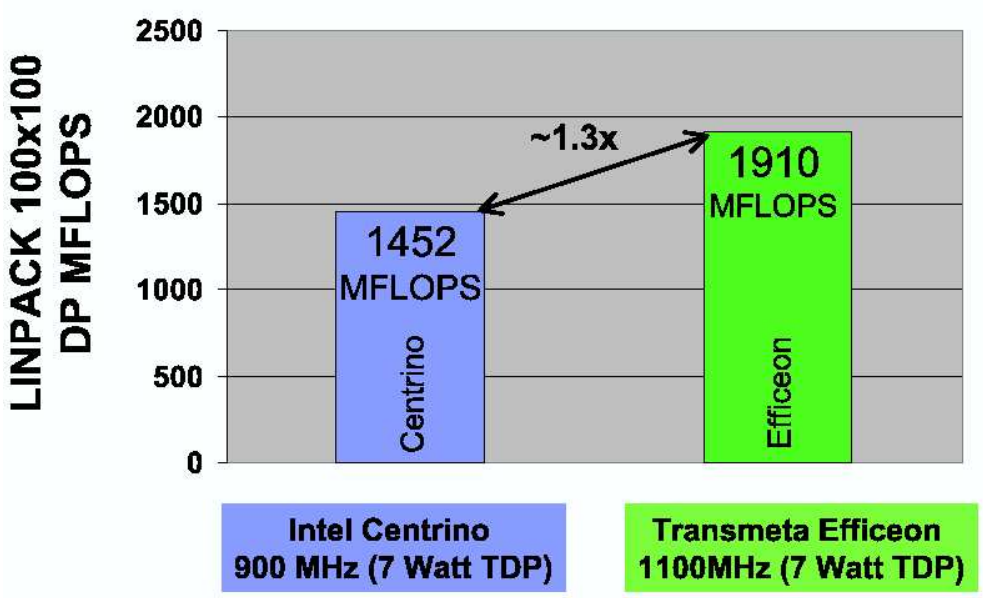


Figura 19: Cálculo de Ponto Flutuante pelo Linpack, restrição de 7W.*(Gráfico obtido de* `http://www.transmeta.com`*)*

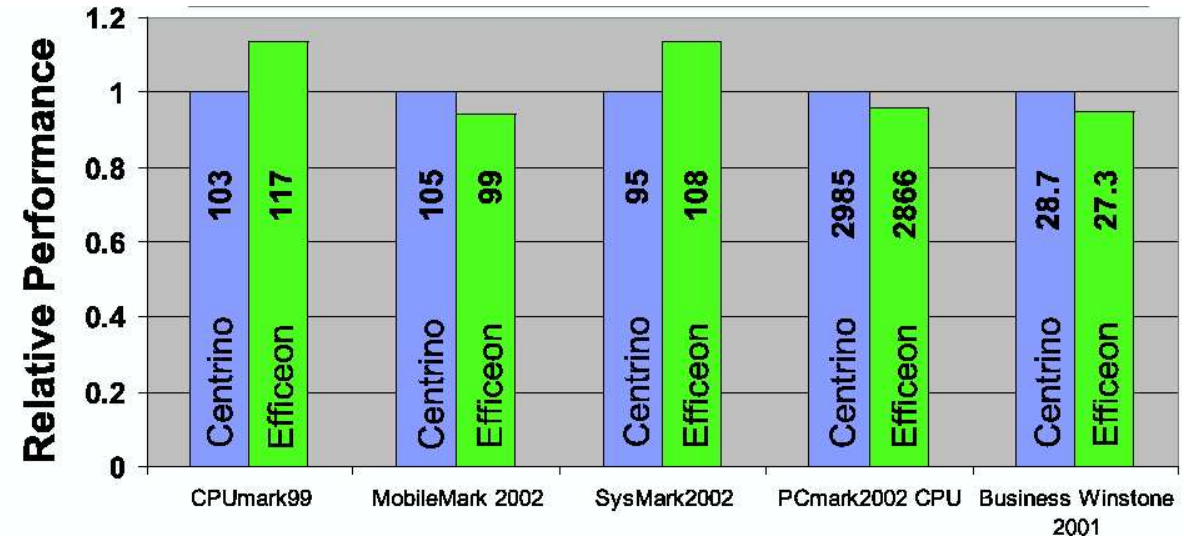
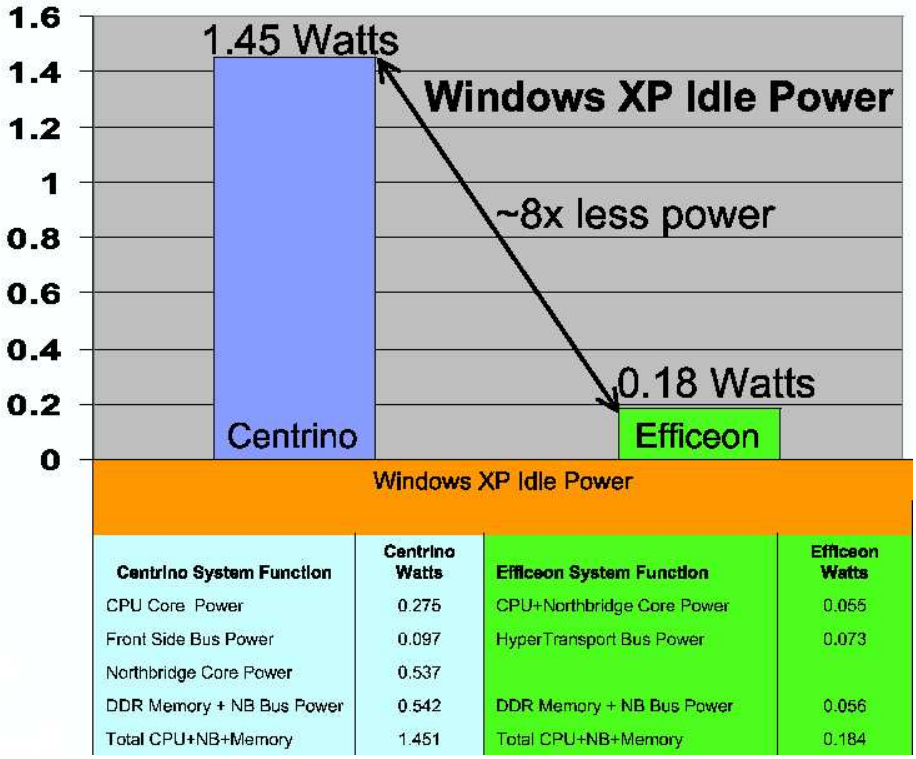


Figura 20: Comparativo de performance em vários *benchmarks* de avaliação do sistema como um todo.*(Gráfico obtido de* `http://www.transmeta.com`*)*

algumas das suites mais famosas: CPUMark99[12], MobileMark 2002[13], SysMark 2002[14], PCMark2002[15], e Business Winstone 2001[16]. Note que algumas dessas *suites* utilizadas estão desatualizadas: O Business Winstone estava na versão 2002 quando o Efficeon foi testado, e o PCMark2002 em uma versão Pro. E exatamente nesses testes o Efficeon perde para o Centrino. Pode-se concluir por isso, e pelo fato do Efficeon ter 200MHz a mais que o Centrino utilizado, que o Centrino ainda pode ser uma opção melhor para o uso geral de um computador.

8.6.4 Análise de Desempenho: Gasto de Energia



Centrino System Function	Centrino Watts	Efficeon System Function	Efficeon Watts
CPU Core Power	0.275	CPU+Northbridge Core Power	0.055
Front Side Bus Power	0.097	HyperTransport Bus Power	0.073
Northbridge Core Power	0.537		
DDR Memory + NB Bus Power	0.542	DDR Memory + NB Bus Power	0.056
Total CPU+NB+Memory	1.451	Total CPU+NB+Memory	0.184

Figura 21: Comparativo de gasto de energia, sem processos rodando (*"idle"*), no Windows XP.*(Gráfico obtido de* `http://www.transmeta.com`*)*

Infelizmente a transmeta não divulgou nenhum teste real de bateria, apenas dados de consumo do processador com o Windows XP ocioso. Nesse teste em específico, o Efficeon consome aproximadamente 8 vezes menos energia que o Centrino. Tal resultado é ilustrado na Figura 21.

Pela arquitetura do Efficeon com o *Enhanced Long Run*, esse é um resultado esperado, pois está de acordo com a comparação geral entre o Crusoé e os processadores da Intel.

8.6.5 Análise de Desempenho: Conclusão

Tendo em mente que os resultados apresentados aqui foram fornecidos pelo próprio fabricante do processador e não por uma terceira parte independente, percebemos que o Efficeon, um processador VLIW, está no mesmo nível de velocidade de processamento do processador da Intel, um processador escalar.

[12] `http://www.etestinglabs.com/bi/cont1999/1999print/scoring.asp`
[13] `http://www.futuremark.com/products/mobilemark2002/`
[14] `http://www.futuremark.com/products/sysmark2002/`
[15] `http://www.futuremark.com/products/pcmark2002/`
[16] `http://www.veritest.com/benchmarks/bwinstone/bwinstone.asp`

Segundo a Transmeta, o Efficeon tem por objetivo reduzir custos, reduzir consumo de energia, reduzir a dissipação de calor e continuar compatível com a plataforma x86. Utilizando a plataforma VLIW facilitou o alcance dos três primeiros objetivos e o último foi conseguido devido à inovação trazida pelo *Advanced Code Morphing*, que é uma solução às principais desvantagens do VLIW, como falta de compatibilidade, muita complexidade no compilador dentre outros, isto sem adicionar complexidade ao *hardware*. Porém isto vem ao custo de performance, que apesar de não ser muito grande, não aproveita integralmente o VLIW como no caso de outras plataformas que podem fazer o *trace schedule* em tempo de compilação.

9 Vantagens e Desvantagens do VLIW

9.1 Vantagens

- Por ter acesso ao código fonte, o compilador consegue analisar janelas maiores de código, aumentando o nível de otimização. Para um efeito semelhante em *hardware*, muita complexidade teria que ser adicionada, aumentando a área do *chip* e conseqüentemente encarecendo-o.
- Como o compilador tem acesso ao código fonte ele tem informações que são perdidas após a compilação e que seriam úteis para uma melhor otimização.
- Com um número suficiente de registradores é possível imitar o comportamento de reordenação de *buffer*, como nos processadores superescalares. Com isso o processador VLIW pode executar instruções antes de seu tempo e caso seja necessário, descartar tais valores (no caso de um *branch*, por exemplo). Isto é feito utilizando-se os registradores temporários para operações executadas antes da hora.
- Os problemas de otimização são resolvidos somente uma vez: em tempo de compilação.
- Como a complexidade é movida para o compilador, uma quantidade menor de *hardware* é necessária, barateando custos, diminuindo gastos com energia e dissipação de calor.

9.2 Desvantagens

- Por ser difícil manter sempre todos os espaços para operações ocupados em uma instrução, sendo assim, muitos recursos podem ser perdidos: banda do barramento, espaço do cache de instruções e espaço da memória de instruções.
- Código fonte do programa é necessário.
- Não tem a vantagem de rodar o mesmo código que outras plataformas, como, por exemplo, os superescalares que rodam código dos escalares;
- A dependência da plataforma é muito evidente, sendo difícil haver compatibilidade. O compilador precisa, muitas vezes, ser dedicado para tal plataforma afim de conseguir uma alta otimização, isto é, compiladores que tentam compilar para várias plataformas, ie o GCC[17], dificilmente serão um bom compilador VLIW.
- Impraticável a programação direta em *assembly*;

Referências

[Cam99] Tatiane Jesus de Campos. *Técnicas de Redução do Custo de Desvio por Software*. http://www.inf.ufrgs.br/procpar/disc/cmp134/trabs/T1/991/ReducaoCustos/desvios.html, 1999.

[Com] *Computer Architecture*. www.ic.unicamp.br/ra008848/mc427/computer_architecture.pdf.

[17] GCC: *CNU Compiler Collection*. `http://www.gnu.org/software/gcc`

[Gro] J. P. Grossman. Compiler and archtectural techniques for improving the efectiviness of vliw compilation.

[LMK+92] P. Geoffrey Lowney, Stefan M.Freudenberg, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. *The Multiflow Trace Scheduling Compiler*, 1992.

[Mat97] Binu K. Mathew. Vliw processors and trace scheduling. Technical report, Utah Unniversity, 1997.

[RF92] B. Ramakrishna Rau and Joseph Fisher. Instruction level parallel processing: History overview and perspective. Technical report, Hewlett Packard, October 1992.

[Sil] Gabriel P. Silva. *Escalonamento de Instruções - Microarquiteturas de Alto Desempenho*. Universidade Federal do Rio de Janeiro, http://equipe.nce.ufrj.br/gabriel/microarq/.