Pequena Apresentação Sobre Paralelismo em nível de instrução

- Primórdios do Paralelismo:
  - Pipeline
  - Múltiplas unidades funcionais (sem *pipeline*)
- Paralelismo em Nível de Instrução
  - processadores *superescalares*
  - processadores *VLIW*

Breve Histórico

- Anos 40 e 50
  - O microcódigo horizontal inicia as primeiras experiências em paralelismo
- Anos 60
  - CDC6600: Computador da Control Data Corporation, continha 10 unidades funcionais que podiam executar uma operação independentemente das outras unidades.
  - IBM 360/91: possuia menos unidades funcionais que o CDC mas usava técnicas de re-organização das instruções muito mais agressivas que o anterior.
- Anos 80 e 90
  - O silício em fartura e as nanotecnologias permitem um grande salto de desempenho aos processadores.

Funcionamento Básico

- Paralelismo do VLIW
  - o VLIW além de oferecer *pipelining*, oferece paralelismo em nível de instrução. As instruções do VLIW são formadas de várias operações, cada uma semelhante às instruções de um processador superescalar.
  - O compilador deve fornecer essas instruções ao processador, ou seja o compilador faz todo o trabalho de organização e escalonamento das operações.
- Papel do Compilador
  - As dependências entre as operações devem ser determinadas.
  - As operações que são que são independetes de outras operações que ainda não foram executadas devem ser determinadas.
  - Essas operações independentes devem ser agendadas para execução em algum momento em particular e em uma unidade funcional em particular. Devem também ser alocados registradores.

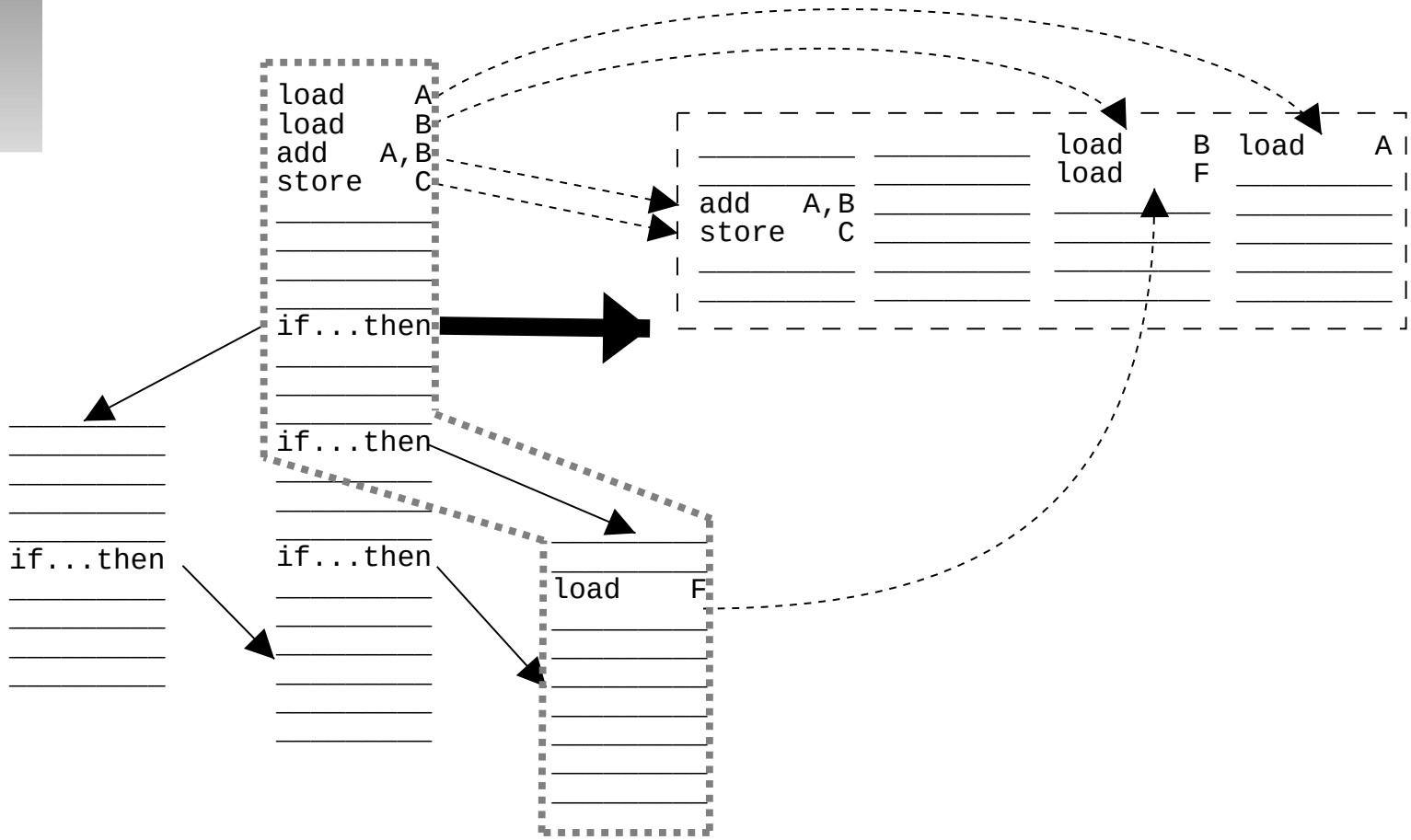
O processo de escalonamento de Instruções

- O escalonamento de instruções pode ser feito de muitas maneiras diferentes:
  - O principal mecanismo para processadores VLIW é o ***trace schedule***.
  - Além dele outras abordagens tratam de blocos de código específi cos. Existem inúmeros algoritmos para **desenrolar laços**.

O Algoritmo de trace schedule

- 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.
- Remover esse trace do grafo de fluxo, e repassá-lo para o agendador de instruções.
- Quando o trace estiver pronto em um agendamento, passar esse agendamento para o grafo de fluxo, substituindo as operações que estavam originalmente no trace. 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.
- Re-iterar até que todas as operações tenham sido incluídas em algum trace e todos os traces tenham sido substituidos por um agendamento.
- Selecionar a melhor ordem linear para o código e emití-lo.

O Algoritmo de trace schedule



Código de compensação

- São gerados basicamente por dois fenômenos:
  - ***split:*** Acontece quando se chega a uma operação que leva a dois ou mais possíveis caminhos, possui mais que 1 sucessor.
  - ***join:*** Contrário de *split*, é uma operação que pode ser alcançada por mais de um caminho, possui mais que um precedente.
- Os *splits* e *joins* geralmente trazem a necessidade de se fazer cópias de instruções para que o Grafo de Precedência de Dados fi que correto.

Agendador de Instruções

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

Mecanismo de previsão de desvio

- Os desvios podem diminuir substancialmente o ganho do *pipeline*.
- Técnicas tentam minimizar estas perdas.

Considere o código:

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

Delayed Branch

- Seqüência 1: 1, 2, 3, 4, 7; Pois **existe um atraso** de um ciclo na 3^{a} instrução (`if ( X == W ) goto 7`).
- Seqüência **incorreta**, pois executa a instrução 4 (`W--`) que **não é independente**

Soluções:

1. inserir um NOP como 4^{a} instrução e ter uma operação nula
   - **Problema:** Porém poderíamos ter muitos NOPs em um código que fosse maior. *Perda de recursos!*
2. alterar ordem de operações independentes. Princípio do **Delayed Branch**

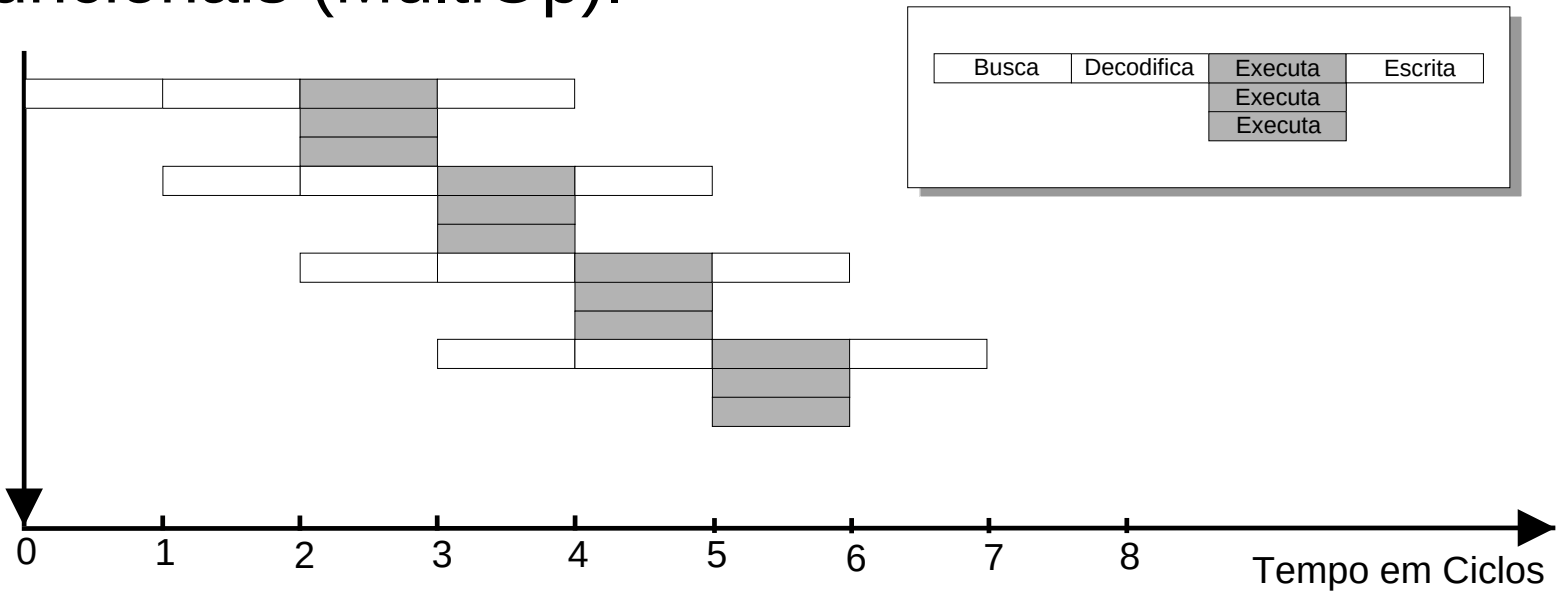
```
1.   X = Y;
2.   If ( X == W )
        goto 7;
3.   Y--;
4.   W--;
5.   printf("Teste");
6.   printf("Outro Teste");
7.   Z = W;
```

Branch Folding

- Utiliza-se uma análise estatística para avaliar qual o melhor caminho a ser seguido no *branch*
- Escolhido melhor caminho, coloca-se o endereço da instrução seguinte precedendo a instrução de *branch* e deixa o outro caminho como alternativo.
- Caso o caminho escolhido seja o correto, continua a execução.
- Caso não, retira-se as operações incorretas que estão no *pipeline* e coloca o endereço alternativo no PC para que seja executado.

Emissão de Instruções

- Emissão de múltiplas instruções possível pelo uso de *pipeline*.
- Utilização de múltiplas operações com unidades funcionais (MultiOp).



- Dependência de efi ciência do compilador, para bom escalonamento.

Recursos nos Compiladores para expor e explorar ILP

- Objetivo: Manter unidades funcionais ocupadas maior parte do tempo
- Técnicas em software:
  - *Trace Scheduling*
  - *Software Pipelining*

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

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

```
loop:
  store ( i );
  decr  ( i + 2 );
  fmul  ( i + 3 );
  load  ( i + 4 );
  bc    loop;
```

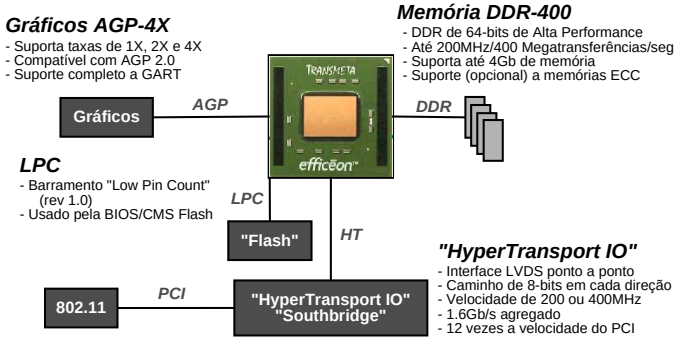
/* Este loop deve ser executado para i=1 a 3 */

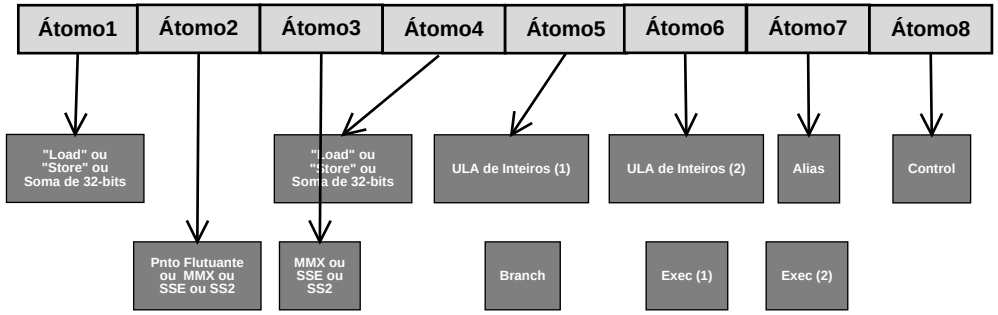
Software Pipelining

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

Caso de Estudo: O processador Efficeon

- O processador Efficeon é o novo processador VLIW de 256 bits compatível com x86 da Transmeta.
- Baseado no Crusoé, ele possui recursos como AGP, DDR de barramento 400MHz e HyperTransport integrados, e ainda conjunto de instruções MMX, SSE e SSE2, e até 1Mb de Cache L2.
- Foi criado para ter o maior desempenho possível em um processador que roda sem cooler, dissipando apenas 7W, visando o mercado de portáteis (notebooks e tablets pcs).





Caso de Estudo: A arquitetura do Efficeon

- O processador Efficeon utiliza uma arquitetura VLIW de oito vias de 32 bits, totalizando 256 bits, chamadas de Átomos. Uma seqüência de átomos compõe uma Molécula.
- Cada átomo pode executar uma operação e portanto uma molécula pode executar até 8 operações em paralelo. Todos os átomos têm acesso à todas as 11 unidades lógicas existentes.
- Ele conta com 64 registradores inteiros de 32 bits, 64 registradores de ponto flutuante de 80 bits, com suporte a MMX, SSE e SSE2 e 4 registradores de predicado.
- O cache do Efficeon é separado em um cache L2 de 1Mb ECC, um cache de instrução L1 de 128Kb e um cache de dados L1 de 64Kb.

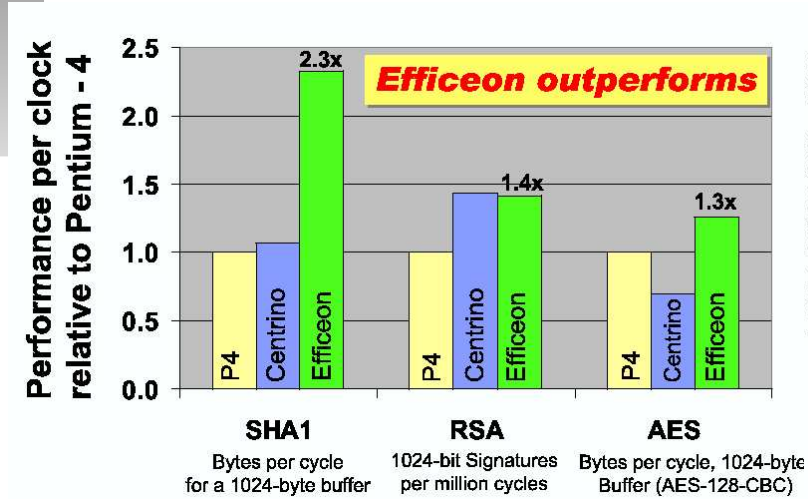
Caso de Estudo: A arquitetura do Efficeon

Inovação: *Advanced Code Morphing*

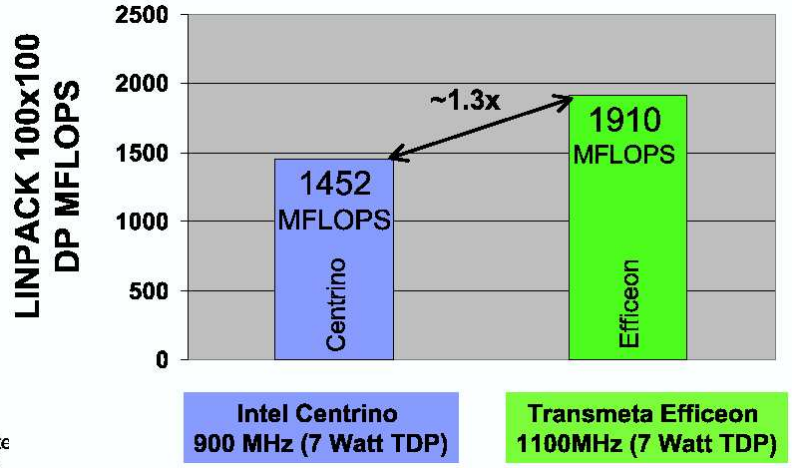
- **Problema:** Um binário VLIW não é compatível com a arquitetura x86.
- **Solução:** Criar uma camada de software entre o núcleo do processador e o executável. Esta deve traduzir e compactar (paralelizar) dinamicamente instruções x86 para VLIW.
- **Efeito Colateral:** não podemos compilar um código nativo VLIW para otimizar o paralelismo.
- É possível mudar o conjundo de instruções aceitas pelo processador mudando apenas esta camada. Por exemplo, pode-se interpretar código Sparc ou mesmo Java.

Caso de Estudo: Análise de Desempenho

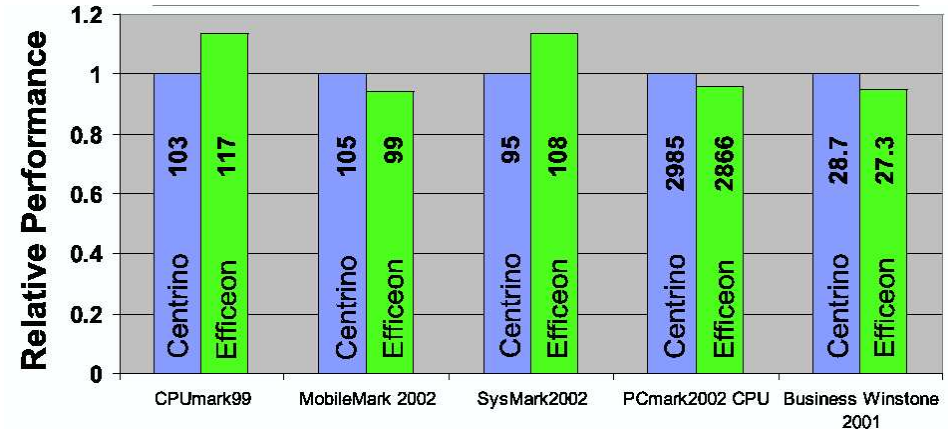
Números Inteiros:



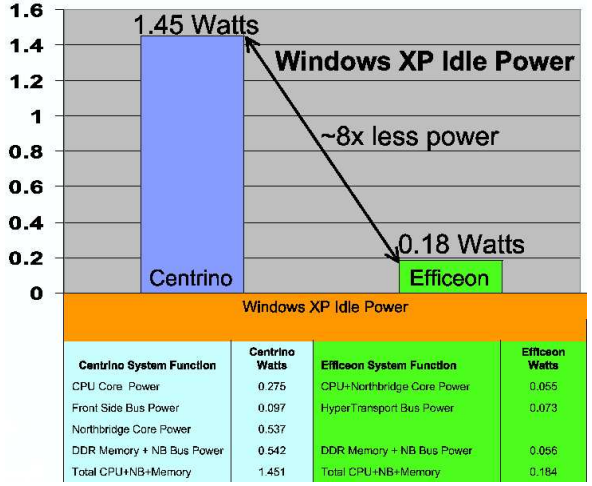
Ponto Flutuante:



Performande Geral do Sistema:



Gasto de Energia:



Caso de Estudo: Conclusão

- O ***Advanced Code Morphing*** extende as possibilidades do VLIW, mantendo compatibilidade com plataformas já existentes.
- Porém esta camada de software custa ao processador perda de desempenho e deixa de aproveitar integralmente as vantagens do VLIW.
- O processador atinge o mesmo desempenho de concorrentes de tecnologia escalar, porém com melhor relação desempenho/consumo.
- Nota: *Os gráficos de desempenho foram providos pela própria Transmeta, portanto podem ser tendenciosos.*

Vantagens e Desvantagens do VLIW

Vantagens:

- Acesso ao código fonte permite ao compilador analisar janelas maiores de código, aumentando o nível de otimização, sem aumentar complexidade do *hardware*.
- Acesso ao código fonte provê ao compilador informações que serão perdidas após a compilação, porém 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.
- **Os problemas de otimização são resolvidos somente uma vez**: em tempo de compilação.
- Complexidade no compilador = menos *hardware* = menos consumo de energia = menos calor gerado.

Desvantagens:

- Dificuldade em manter espaços de operações ocupados em uma molécula pode acarretar em perda de recursos (banda, cache, ...).
- Código fonte 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 existir compatibilidade.
- Impraticável a programação direta em *assembly*.