Optimização do Desempenho:
Técnicas Dependentes da Máquina

Arquitectura de Computadores

Lic. em Engenharia Informática

Luís Paulo Santos

Optimização: Dependência do Processador

	10 – Optimização do Desempenho
Conteúdos	10.3 – Super-escalaridade e Execução fora-de-ordem
	10.4 – *Loop Unrolling/Splitting*
Resultados de Aprendizagem	R10.1 – Descrever, aplicar e avaliar técnicas de optimização de desempenho
	R10.2 – Analisar e justificar o impacto de múltiplas unidades funcionais no desempenho da máquina

Optimização de `combine()`

```
void combine4(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int sum = 0;
  for (i = 0; i < length; i++)
    sum += data[i];
  *dest = sum;
}
```

- Função
  - Calcula a soma de todos os elementos de um vector
  - Alcançou um CPE de 2.00
    - Ciclos por elemento

Forma Geral de `combine()`

```
void abstract_combine4(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *data = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP data[i];
  *dest = t;
}
```

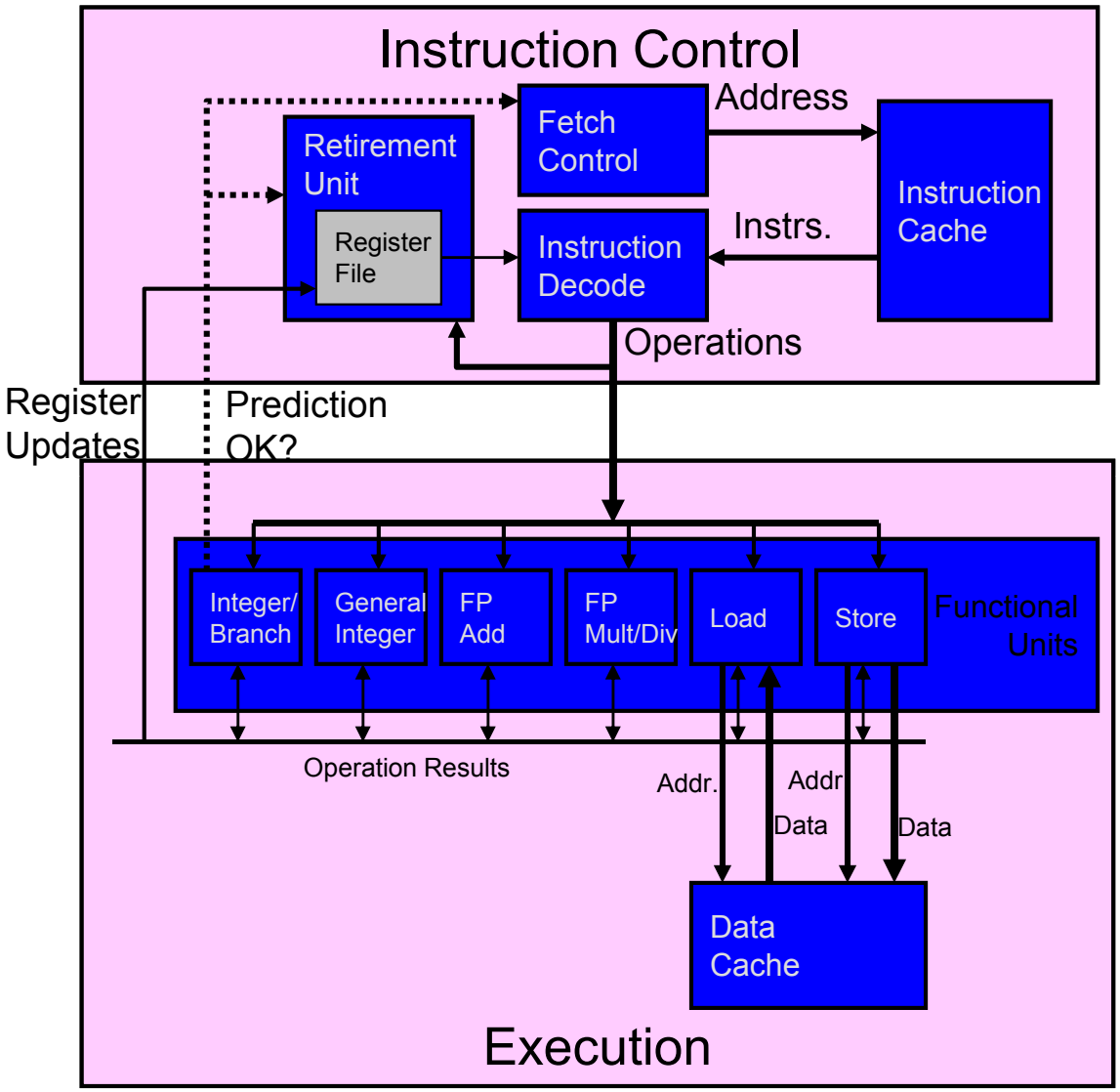
- Tipos de Dados
  - Usar diferentes declarações para `data_t`
    - `int`
    - `float`
- Operações
  - Diferentes definições para `OP` e `IDENT`
  - `+` / `0`
  - `*` / `1`

Optimizações Independentes da Máquina

Método	Integer		Floating Point	
	+	*	+	*
-g	42.06	41.86	41.44	160.00
-O2	31.25	33.25	31.25	143.00
Mover vec_length	20.66	21.25	21.15	135.00
acesso dados	6.00	9.00	8.00	117.00
Acum. em temp	2.00	4.00	3.00	5.00

- Anomalia Desempenho
  - Grande optimização quando o produto FP é acumulado em temp:
    - A Memória usa formato de 64-bits, os registos usam 80 bits
    - A operação causou *overflow* com 64 bits, mas não com 80

Processadores Actuais

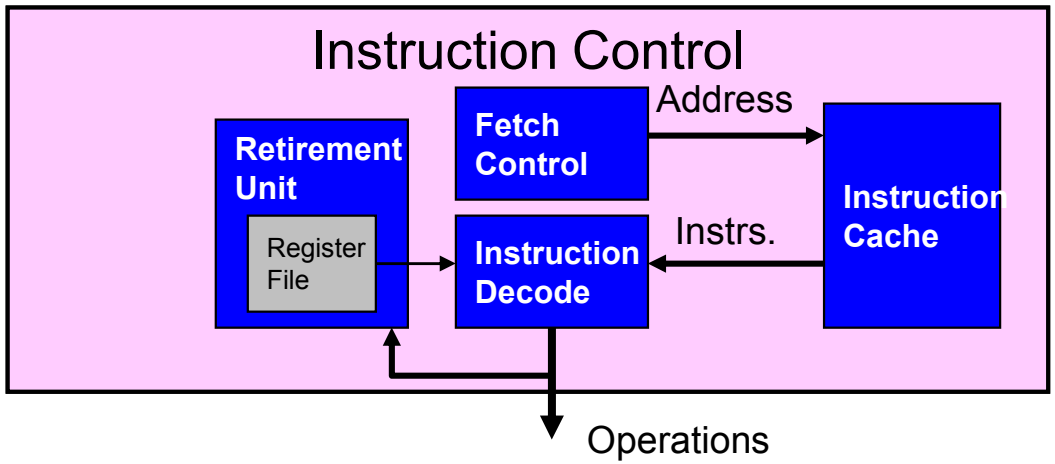


Unidades Funcionais: Desempenho (Pentium III)

- Múltiplas Instruções podem ser executadas em Paralelo
  - 1 load (leitura da memória)
  - 1 store (escrita na memória)
  - 2 operações inteiras (apenas uma pode ser um salto)
  - 1 Adição FP
  - 1 Multiplicação ou Divisão FP
- Algumas Instruções necessitam de > 1 Ciclo, mas podem ser em encadeadas (*pipeline*)

Instrução	**Latência**	**Ciclos entre Operações**
Load / Store	3	1
Multiplicação Inteira	4	1
Divisão Inteira	36	36
Multiplicação FP	5	2
Adição FP	3	1
Divisão FP	38	38

Controlo das Instruções



- Lê Instruções da Memória
  - Baseada no PC + alvos previstos para os saltos
  - Hardware prevê dinamicamentes se os saltos são tomados ou não
- Traduz Instruções em Micro-*Operações*
  - Micro-Operação: passos elementares para a execução de uma instrução
  - Instrução típica requer 1–3 micro-operações
- Referências aos registos são substituídas por etiquetas
  - Etiqueta: Identificador abstracto liga o destino de uma operação aos operandos das próximas

Tradução em micro-operações: Exemplo

- `Combine4()`: dados inteiros, multiplicação

```
.L24:                                  # Loop:
  imull (%eax,%edx,4),%ecx             # t *= data[i]
  incl %edx                            # i++
  cmpl %esi,%edx                       # i:length
  jl .L24                              # if < goto Loop
```

- Tradução de uma iteração:

```
.L24:
  imull (%eax,%edx,4),%ecx

  incl %edx
  cmpl %esi,%edx
  jl .L24
```

```
load (%eax,%edx.0,4)  ➔ t.1
imull t.1, %ecx.0     ➔ %ecx.1
incl %edx.0           ➔ %edx.1
cmpl %esi, %edx.1     ➔ cc.1
jl-taken cc.1
```

Tradução em micro-operações: `imull`

```
imull (%eax,%edx,4),%ecx
```

```
load (%eax,%edx.0,4) ➔ t.1
imull t.1, %ecx.0    ➔ %ecx.1
```

- Dividir em 2 operações
  - `load` lê da memória e produz o valor `t.1`
  - `imull` opera apenas sobre registos
- Operandos
  - `%eax` não muda no ciclo. Lido do banco de registos durante o decode
  - `%ecx` muda em cada iteração. Identificar diferentes versões com `%ecx.0`, `%ecx.1`, `%ecx.2`, …
    - **Register *renaming***
    - Valores passam directamente do produtor aos consumidores

Tradução em micro-operações: `incl, cmpl`

```
incl %edx
```

```
incl %edx.0          ➔ %edx.1
```

- `%edx` muda em cada iteração. Renomear `%edx.0`, `%edx.1`, `%edx.2`, …

```
cmpl %esi,%edx
```

```
cmpl %esi, %edx.1   ➔ cc.1
```

- Códigos de Condição são tratados como registos

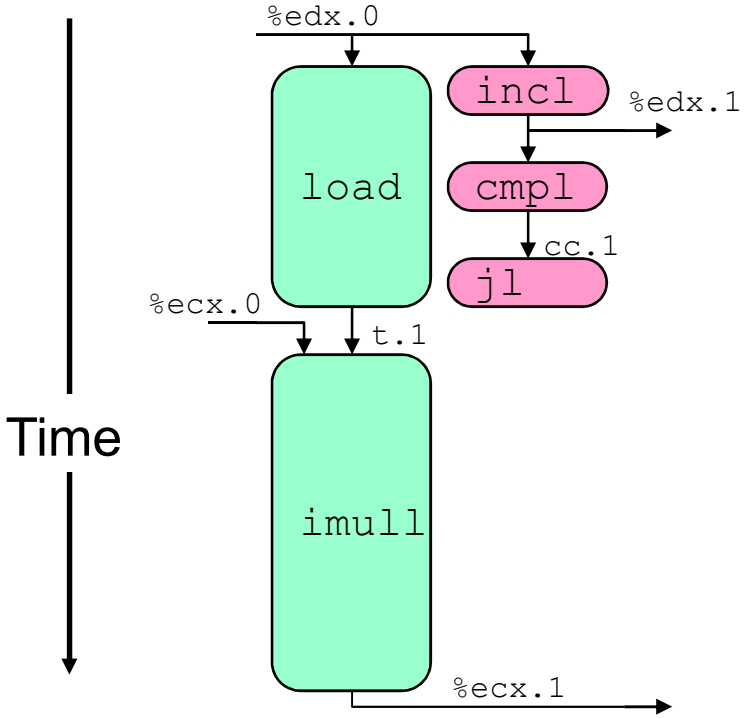
Tradução em micro-operações: `jl`

```
jl .L24
```

```
jl-taken cc.1
```

- *Instruction control* determina destino do salto
- Prevê se é tomado ou não
- Instruções são lidas do destino previsto
- *Execution unit* verifica se previsão foi correcta
- Se não, sinaliza *instruction control*
  - *Instruction control* invalida operações e valores gerados por instruções lidas inadvertidamente
  - Começa então a ler instruções a partir do endereço correcto

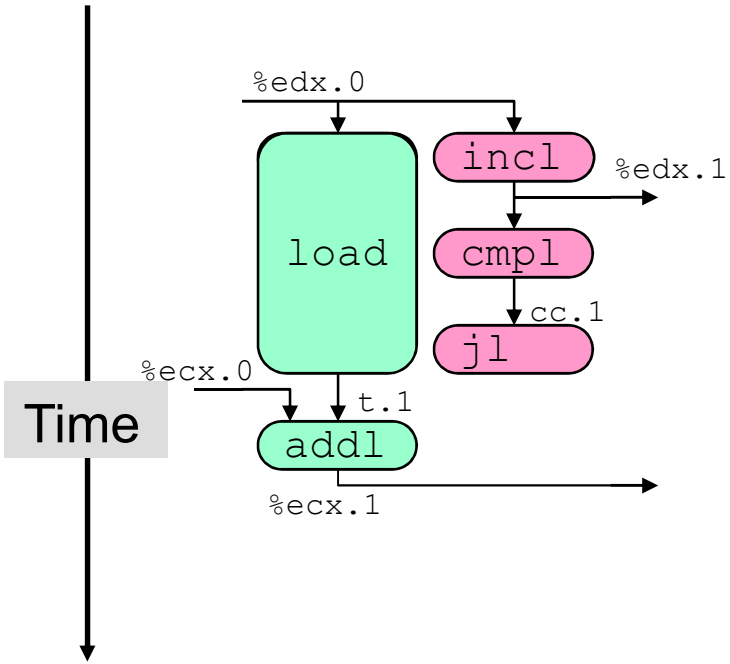
Visualização da execução das µoperações - `imull`




```
load (%eax,%edx,4) ➔ t.1
imull t.1, %ecx.0  ➔ %ecx.1
incl %edx.0        ➔ %edx.1
cmpl %esi, %edx.1  ➔ cc.1
jl-taken cc.1
```

- Operações
  - Eixo Vertical determina tempo
    - Uma operação não pode começar antes de os operandos estarem disponíveis
  - Altura representa latência
- Operandos
  - Representados por arcos

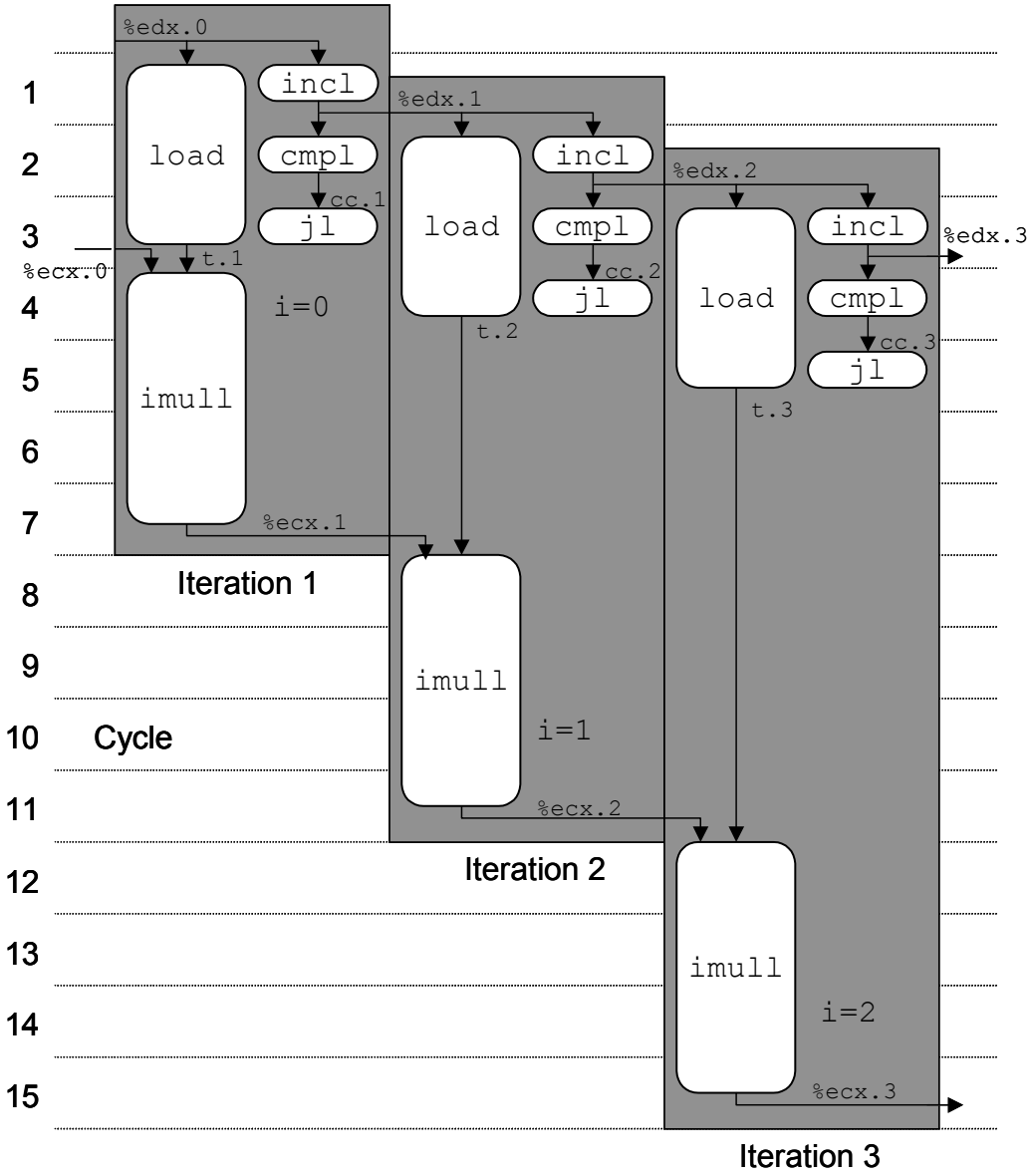
Visualização da execução das µoperações -`addl`




```
load (%eax,%edx,4) ➔ t.1
iaddl t.1, %ecx.0  ➔ %ecx.1
incl %edx.0        ➔ %edx.1
cmpl %esi, %edx.1  ➔ cc.1
jl-taken cc.1
```

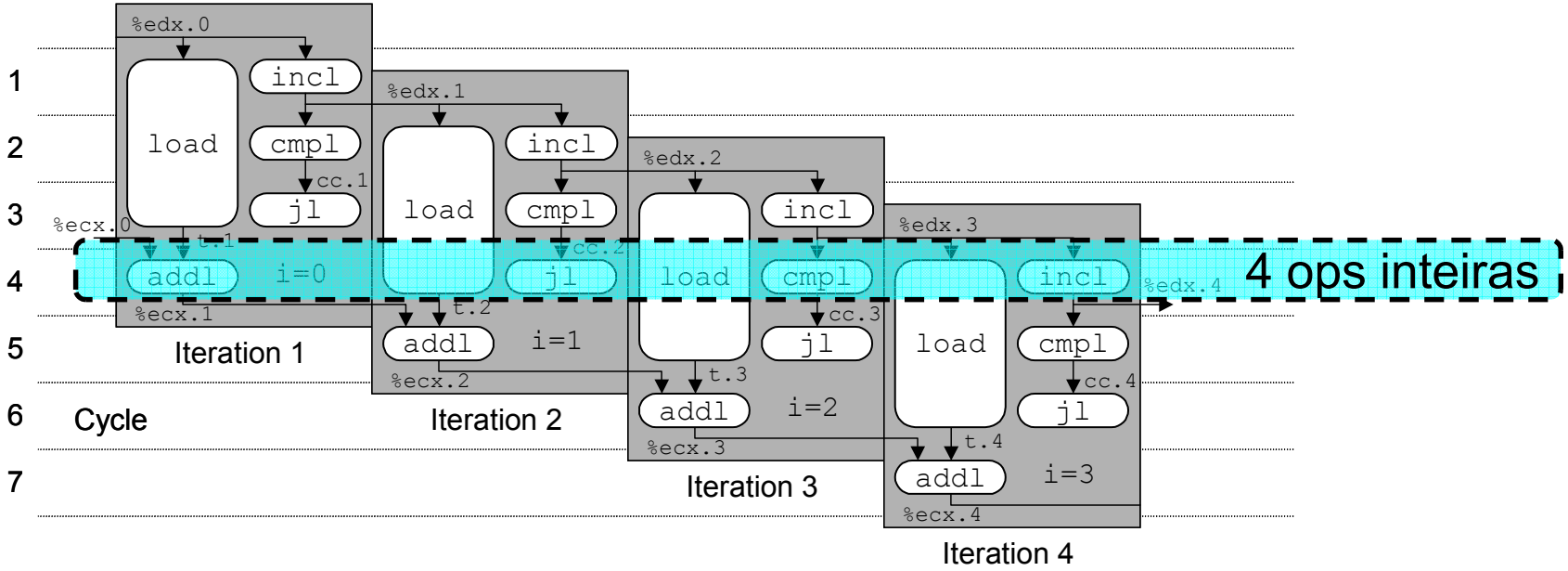
- Operações
  - Idêntico ao anterior, mas adição tem latência=1

Execução `imull`: Recursos Ilimitados



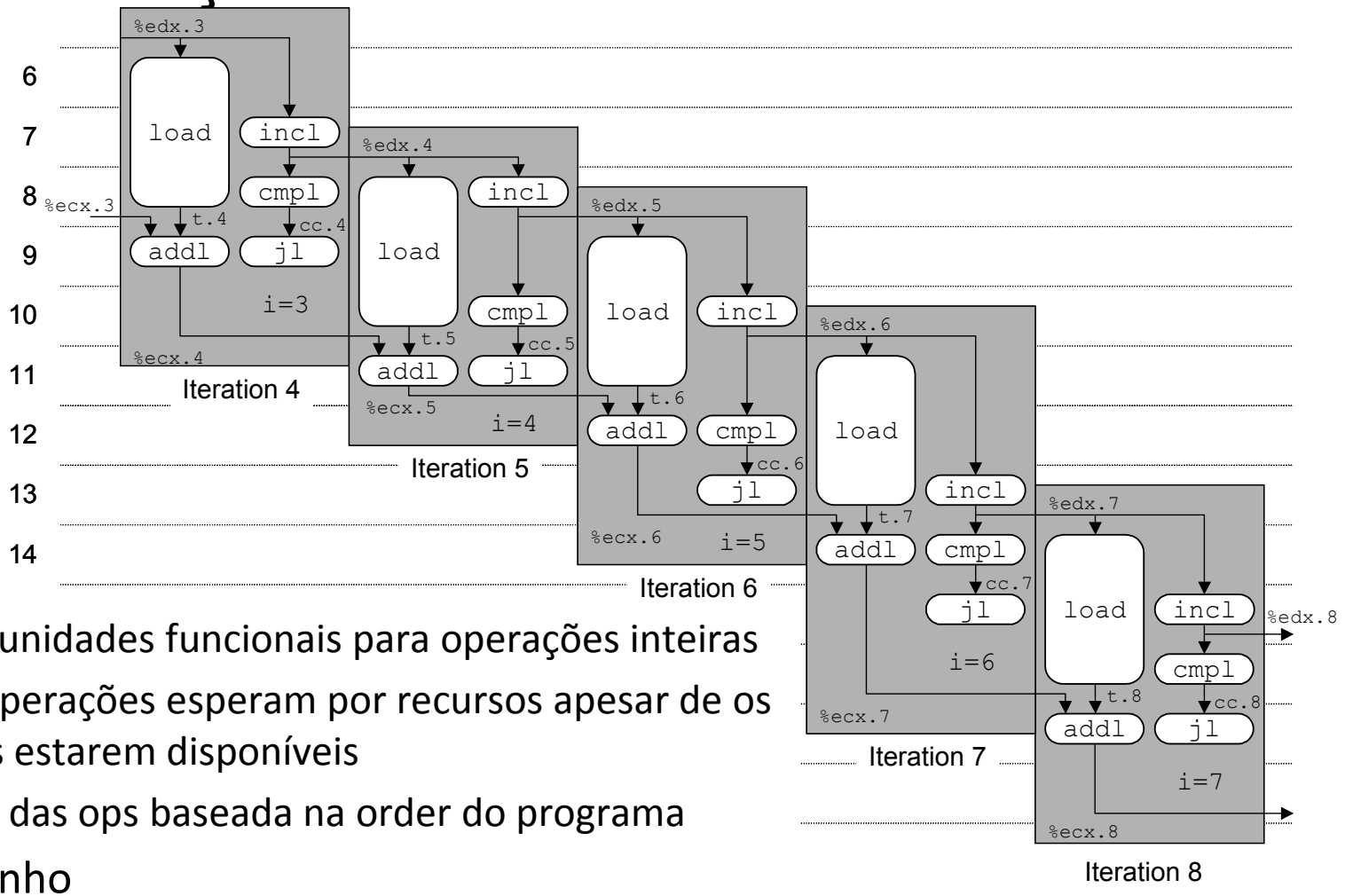
- 3 iterações
  - Assume que as operações podem começar logo que os operandos estão disponíveis
  - Operações de diferentes iterações sobrepõem-se no tempo
- Desempenho
  - Limitado pela latência da unidade de multiplicação
  - CPE = 4.0

Execução `addl`: Recursos Ilimitados



- Desempenho
  - Começa uma nova iteração a cada ciclo
  - CPE = 1.0
  - Requer a execução de 4 operações inteiras em paralelo

Execução `addl`: Recursos Limitados



- Apenas 2 unidades funcionais para operações inteiras
- Algumas operações esperam por recursos apesar de os operandos estarem disponíveis
- Prioridade das ops baseada na order do programa

- Desempenho
  - CPE = 2.0

Escalonamento de µoperações

- 1 µoperação pode ser escalonada para execução quando:
  - Os seus operandos estão disponíveis
  - Existem recursos disponíveis, isto é, uma unidade funcional livre que a possa executar

Optimização: *Loop unrolling*

As operações de controlo do ciclo representam um custo:
- actualização da variável de controlo
- teste da condição
- salto

Desenrolar do ciclo:
- combinar múltiplas operações numa única iteração
- reduz custo do controlo do ciclo

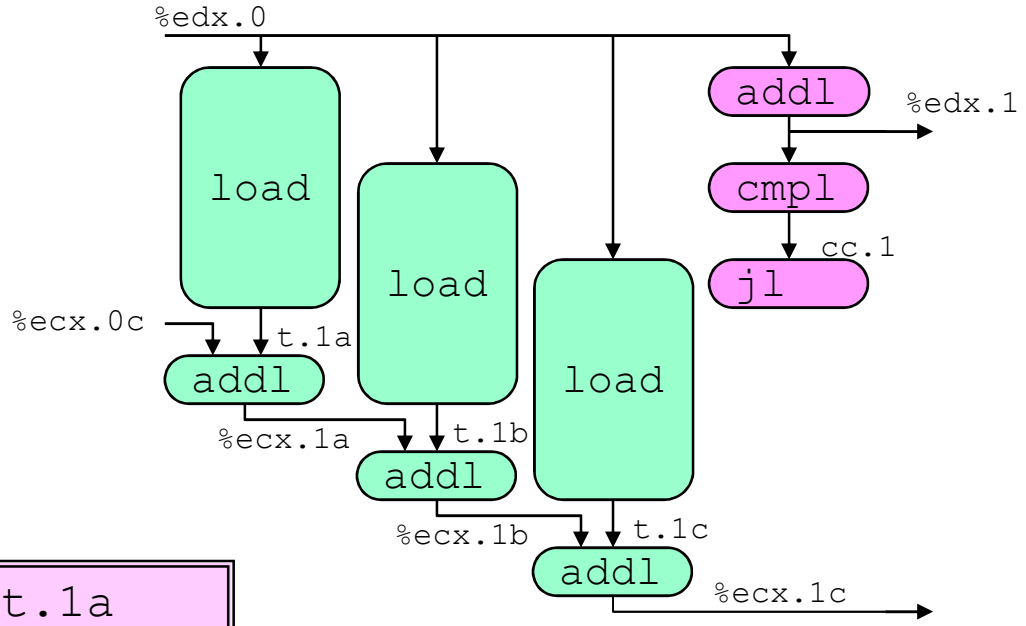
CPE = 1.33

Esta optimização pode ser feita autonomamente pelo compilador

```
void combine5(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-2;
  int *data = get_vec_start(v);
  int sum = 0;
  int i;
  /* Combine 3 elements at a time */
  for (i = 0; i < limit; i+=3) {
    sum += data[i] + data[i+2]
           + data[i+1];
  }
  /* Finish any remaining elements */
  for (; i < length; i++) {
    sum += data[i];
  }
  *dest = sum;
}
```

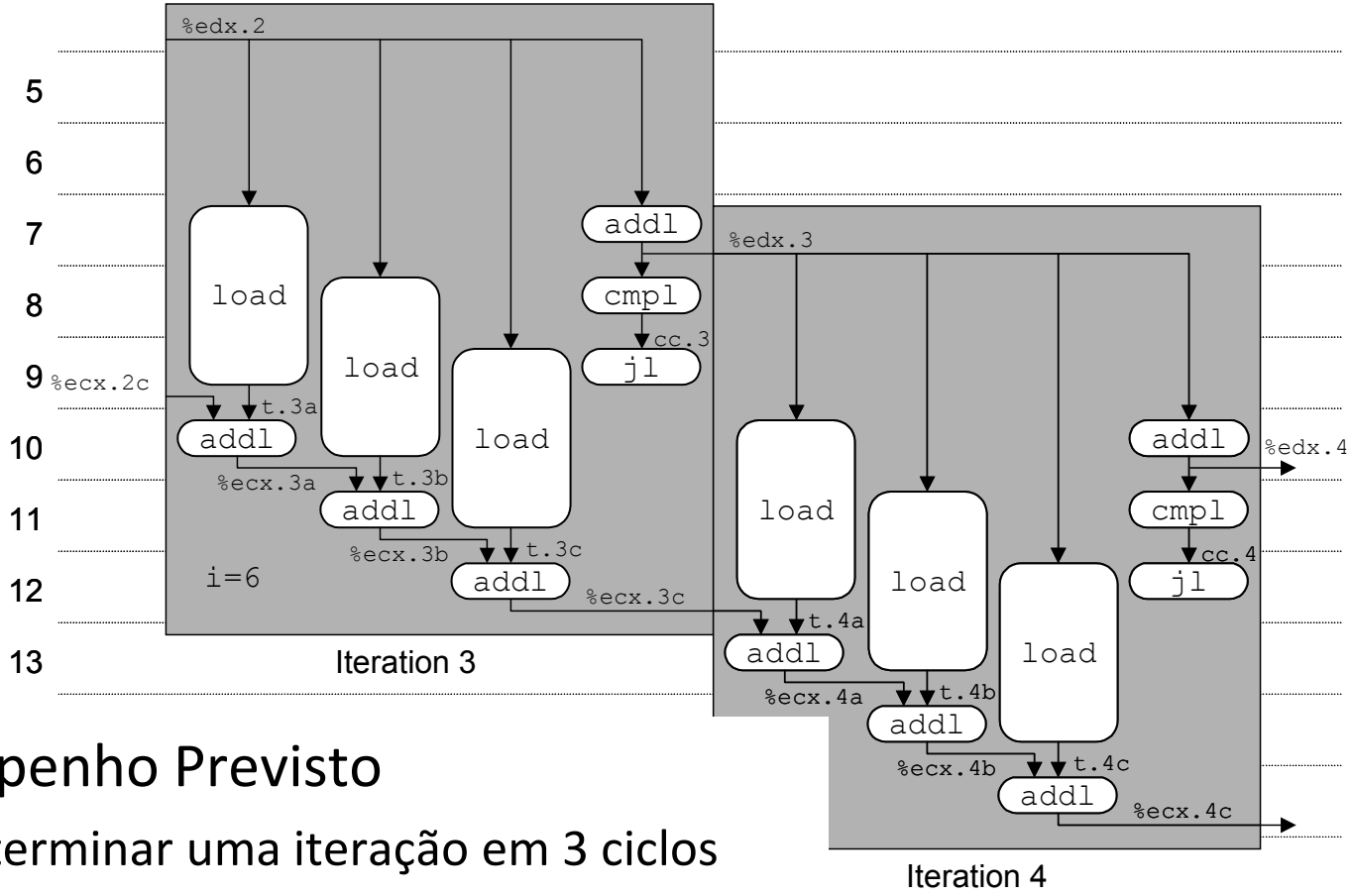
Visualização: *Loop unrolling*

- *Loads* encadeados (não têm dependências)
- Apenas 1 conjunto de operações de controlo




```
load (%eax,%edx.0,4)    ➔ t.1a
iaddl t.1a, %ecx.0c     ➔ %ecx.1a
load 4(%eax,%edx.0,4)   ➔ t.1b
iaddl t.1b, %ecx.1a     ➔ %ecx.1b
load 8(%eax,%edx.0,4)   ➔ t.1c
iaddl t.1c, %ecx.1b     ➔ %ecx.1c
iaddl $3,%edx.0         ➔ %edx.1
cmpl %esi, %edx.1       ➔ cc.1
jl-taken cc.1
```

Visualização: *Loop unrolling*



Desempenho Previsto

- Pode terminar uma iteração em 3 ciclos
- CPE deveria ser 1.0

Desempenho Medido

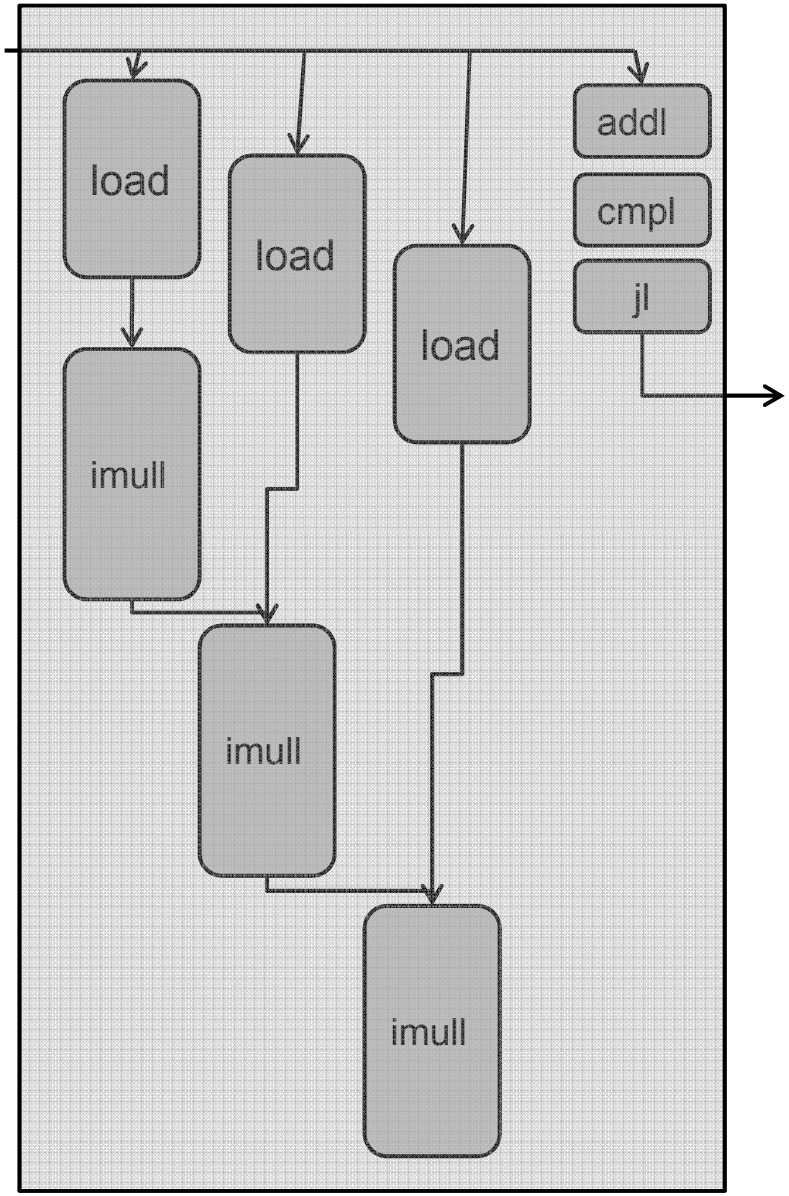
- CPE = 1.33
- Uma iteração cada 4 ciclos

combine5(): resultado de desenrolar o ciclo

Grau de *unrolling*		1	2	3	4	8	16
Integer	Sum	2.00	1.50	1.33	1.50	1.25	1.06
Integer	Product	4.00					
FP	Sum	3.00					
FP	Product	5.00					

- Neste exemplo só benefecia a soma de inteiros
  - Outros casos limitados pela latência das unidades funcionais
- Optimização é não linear com o grau de *unrolling*
  - Existem muitos efeitos subtis que condicionam o escalonamento das operações

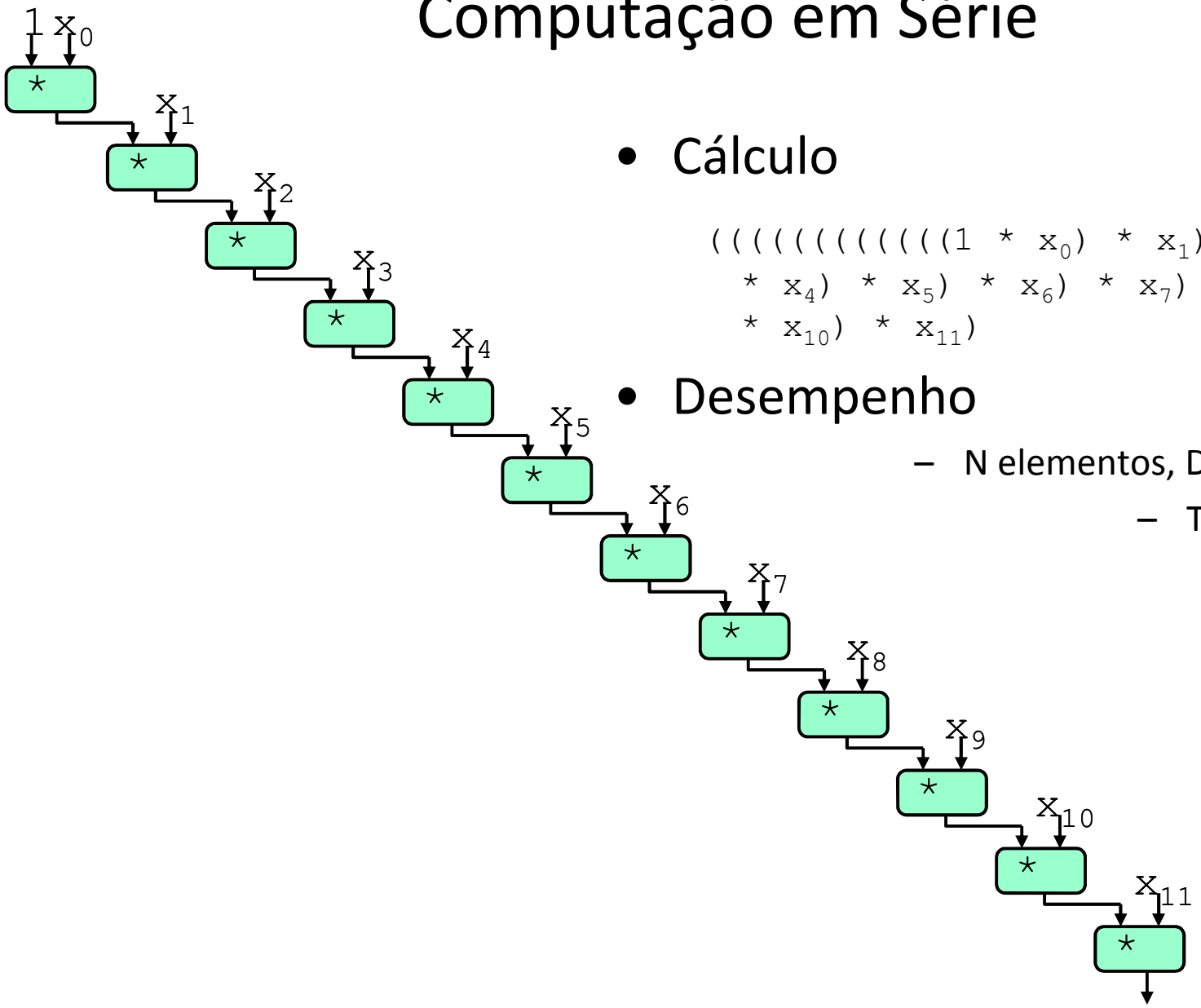
Multiplicação: *unrolling*



Não há vantagem com o desenrolar do ciclo:

- não tiramos partido do encadeamento da unidade de multiplicação devido às dependências de operandos

Computação em Série



- Cálculo

$$(((((((((((1 * x_0) * x_1) * x_2) * x_3) * x_4) * x_5) * x_6) * x_7) * x_8) * x_9) * x_{10}) * x_{11})$$

- Desempenho
  - N elementos, D ciclos/operação
    - Total = N*D ciclos

Desenrolar do ciclo paralelo: *Loop Spliting*

```
void combine6(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-1;
  int *data = get_vec_start(v);
  int x0 = 1;
  int x1 = 1;
  int i;
  /* Combine 2 elements at a time */
  for (i = 0; i < limit; i+=2) {
    x0 *= data[i];
    x1 *= data[i+1];
  }
  /* Finish any remaining elements */
  for (; i < length; i++) {
    x0 *= data[i];
  }
  *dest = x0 * x1;
}
```

– Produto de Inteiros

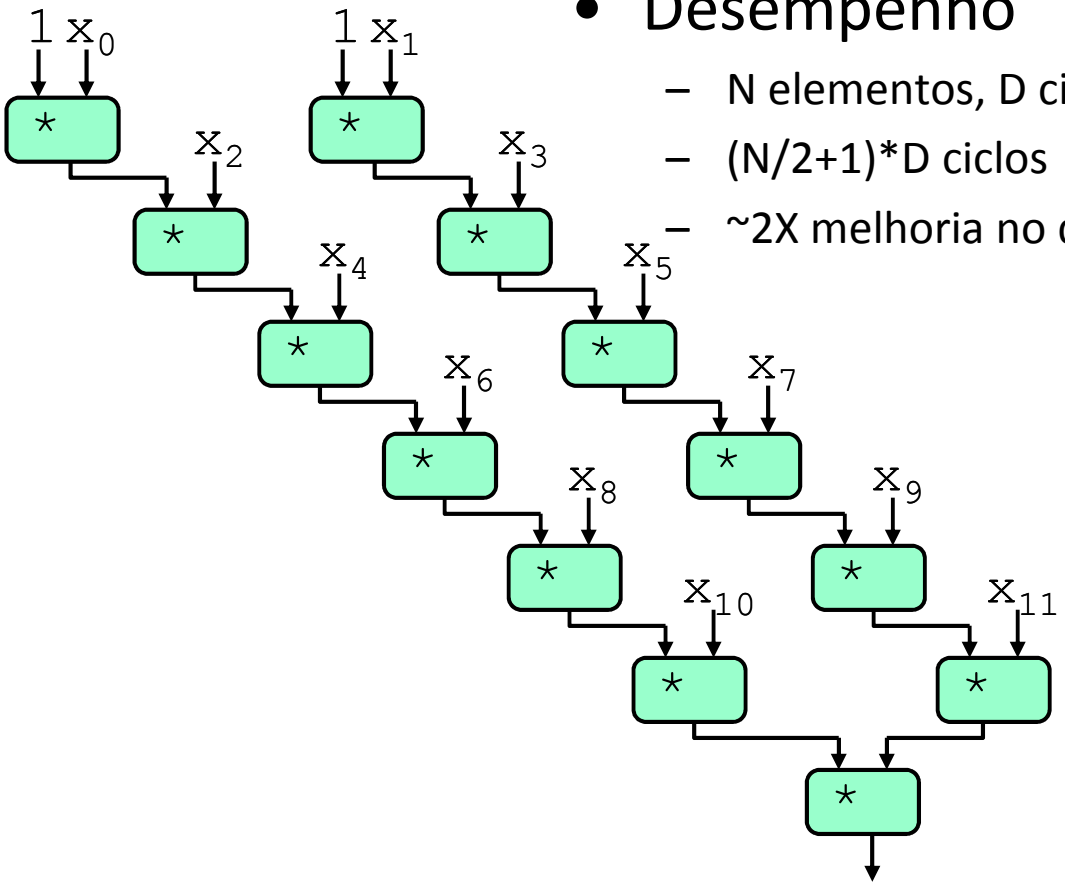
- Optimização
  - Acumular em 2 produtos diferentes
    - Realizados simultaneamente
  - Combinar no fim
- Desempenho
  - CPE = 2.0 (era 4.0)

Loop Spliting grau 2

- Computação

$$((((((1 * x_0) * x_2) * x_4) * x_6) * x_8) * x_{10}) *$$
$$((((((1 * x_1) * x_3) * x_5) * x_7) * x_9) * x_{11})$$

- Desempenho
  - N elementos, D ciclos/operação
  - (N/2+1)*D ciclos
  - ~2X melhoria no desempenho



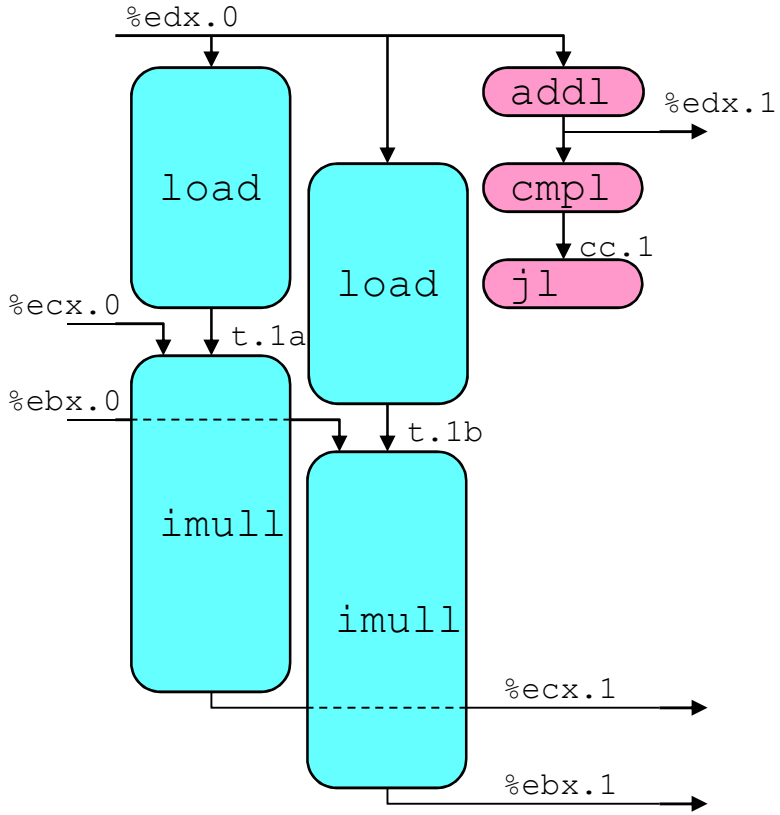
Requisitos para *Loop Spliting*

- Matemáticos
  - A operação tem que ser associativa e comutativa
    - Multiplicação de inteiros: OK
    - Não é necessariamente verdade para operações em vírgula flutuante

- Hardware
  - Múltiplas unidades funcionais ou unidades funcionais encadeadas

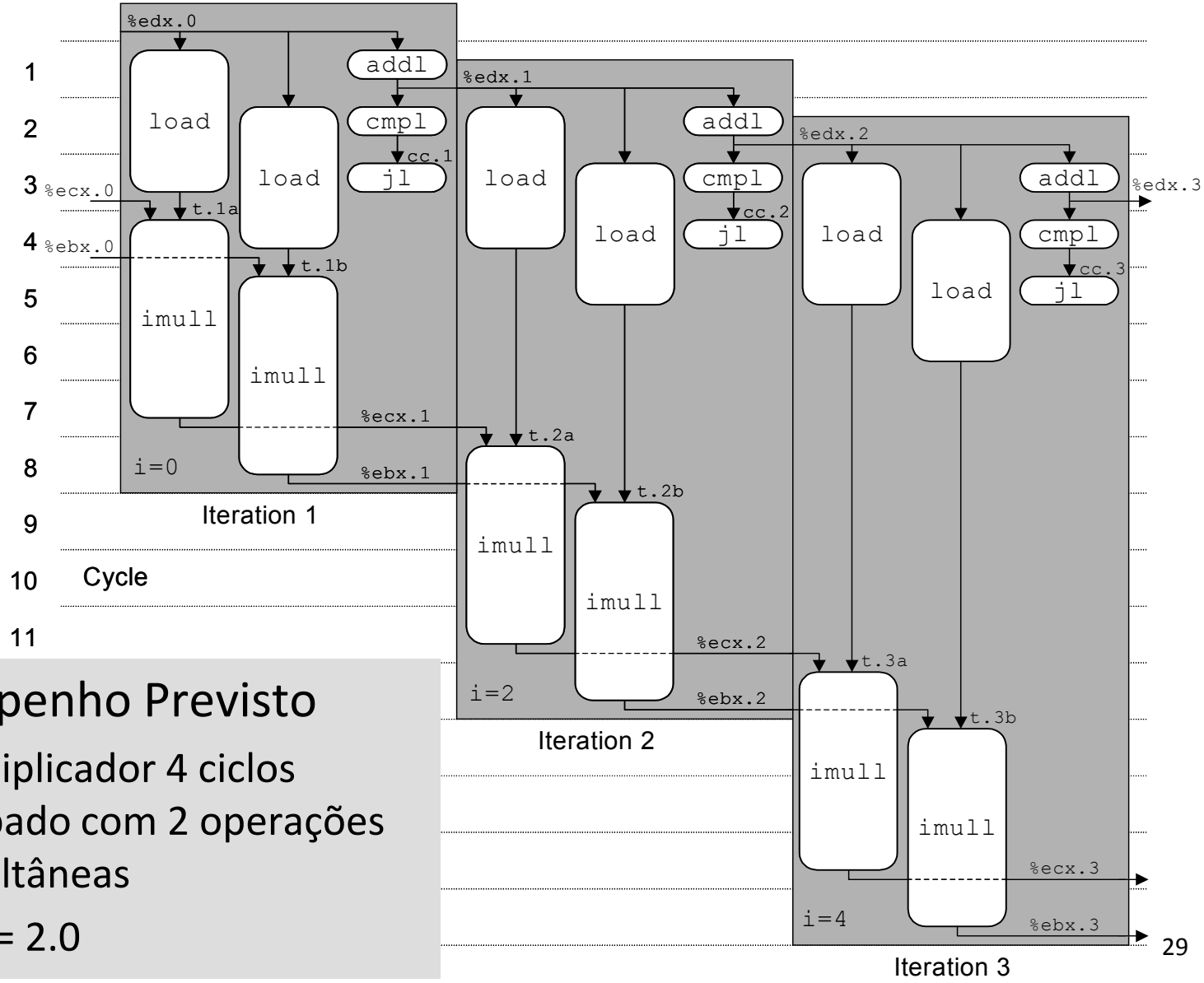
Loop Spliting: Visualização

- As multiplicações não dependem uma da outra
- Podem ser executadas em paralelo




```
load (%eax,%edx.0,4)    ➔ t.1a
imull t.1a, %ecx.0      ➔ %ecx.1
load 4(%eax,%edx.0,4)   ➔ t.1b
imull t.1b, %ebx.0      ➔ %ebx.1
iaddl $2,%edx.0         ➔ %edx.1
cmpl %esi, %edx.1       ➔ cc.1
jl-taken cc.1
```

Loop Spliting: Visualização



- Desempenho Previsto
  - Multiplicador 4 ciclos ocupado com 2 operações simultâneas
  - CPE = 2.0

Resultados para `combine()`

Método	Integer		Floating Point	
	+	*	+	*
Abstract -g	42.06	41.86	41.44	160.00
Abstract -O2	31.25	33.25	31.25	143.00
Mover vec_length	20.66	21.25	21.15	135.00
acesso dados	6.00	9.00	8.00	117.00
Acum. em temp	2.00	4.00	3.00	5.00
Unroll x4	1.50	4.00	3.00	5.00
Unroll x16	*1.06*	4.00	3.00	5.00
Split 2 X 2	1.50	2.00	2.00	2.50
Split 4 X 4	1.50	*2.00*	*1.50*	2.50
Split 8 X 4	1.25	*1.25*	1.50	*2.00*
Ponto Óptimo Teór.	1.00	1.00	1.00	2.00
Pior/Melhor	*39.7*	*33.5*	*27.6*	*80.0*

Limitações de *Spliting*

- Necessita de muitos registos
  - Para armazenar somas/produtos temporários
  - Apenas 6 registos inteiros (na arquitectura IA32)
    - Usados também para apontadores e variáveis auxiliares (ex. Ciclo)
  - 8 registos FP (na arquitectura IA32)
  - Se os registos não são suficientes: *register spilling* na *stack*
    - Elimina quais quer ganhos de desempenho

Saltos Condicionais

- Previsão de saltos condicionais para manter o *pipeline* ocupado
- Nos processadores modernos uma previsão errada implica custos elevados
  - Pentium III: ≈ 14 ciclos do relógio
- Os saltos condicionais podem, em algumas situações, ser evitados pelo compilador com ajuda do programador

Saltos Condicionais

Exemplo: Calcular o máximo de 2 valores

```
int max(int x, int y)
{
  if (x < y) return y;
  else return x;
}
```

```
  movl 12(%ebp),%edx  # Get y
  movl 8(%ebp),%eax   # ret val=x
  cmpl %edx,%eax      # rval-y
  jge L11             # skip when >=
  movl %edx,%eax      # ret val=y
L11:
```

Pentium III:

- 14 ciclos se previsão correcta
- 29 ciclos se previsão errada

`mov` condicional

- Adicionadas à microarquitectura P6 (PentiumPro)
- **`cmovXXl %edx, %eax`**
  - Se condição *XX* verdadeira, copia `%edx` para `%eax`
  - Não há saltos condicionais
  - Corresponde a uma única µoperação

```
int max(int x, int y)
{
 return(x < y) ? y:x;
}
```

```
  movl 12(%ebp),%edx  # Get y
  movl 8(%ebp),%eax   # ret val=x
  cmpl %edx, %eax     # ret val-y
   cmovll %edx,%eax   # If <, ret val=y
```

- Sem as opções correctas o compilador pode não usar esta optimização:
  - Compila para o 386
- Desempenho
  - 14 ciclos