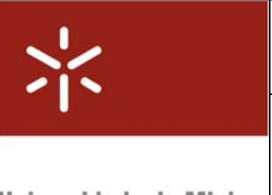


Universidade do Minho

Arquitectura de Computadores II
L.E.S.I. – 3º ano

Módulo nº 6

Avaliação e Optimização do Desempenho
Técnicas dependentes da máquina

1. Introdução

No final deste módulo os alunos deverão ser capazes de:

- interpretar medições do desempenho baseadas em número de ciclos por elemento – CPE;
- descrever e aplicar a técnica dos k-melhores resultados para obtenção de métricas fiáveis;
- avaliar o desempenho de um programa;
- decidir quais as técnicas de optimização dependentes da máquina a aplicar a um algoritmo genérico;

2. Material de Apoio

A bibliografia relevante para este módulo são as secções 5.7 a 5.11, 9.3 e 9.4 do livro "Computer Systems: a Programmer's Perspective", de Randal E. Bryant e David O'Hallaron.

Para realizar os exercícios propostos abaixo deve descarregar o código disponibilizado na página da disciplina em http://gec.di.uminho.pt/lesi/ac2/práticas/docs/ac2Mod6.tgz

3. O Funcionamento dos Processadores Modernos

Os processadores contemporâneos são peças de equipamento sofisticadas dedicadas em grande medida à optimização do desempenho dos programas. Uma das características comuns é que o seu funcionamento é bastante diferente daquilo que se poderia subentender analisando programas escritos em *assembly*. A este nível as instruções parecem ser executadas sequencialmente, uma de cada vez e o fluxo do programa é perfeitamente determinável conhecendo os dados. Na realidade, os processadores modernos executam várias instruções simultaneamente, fazem execução especulativa (execução de sequências de instruções que resultam de saltos condicionais que ainda não foram avaliadas) e execução fora-de-ordem.

➢ Modelo para um processador

Ao longo deste módulo apresenta-se um modelo para um processador baseado na micro-arquitectura Intel P6, que constituiu a base para o Intel Pentium Pro, Pentium II e Pentium III.

Trata-se de uma micro-arquitectura com dois blocos principais: a unidade de controlo de instruções e a unidade de execução. A primeira é responsável por ler instruções máquina da memória, convertê-las em micro-instruções enviando-as para a unidade de execução e por decidir se os resultados da execução devem ser escritos nos registos e memória. A segunda é responsável pelo execução das micro-instruções.

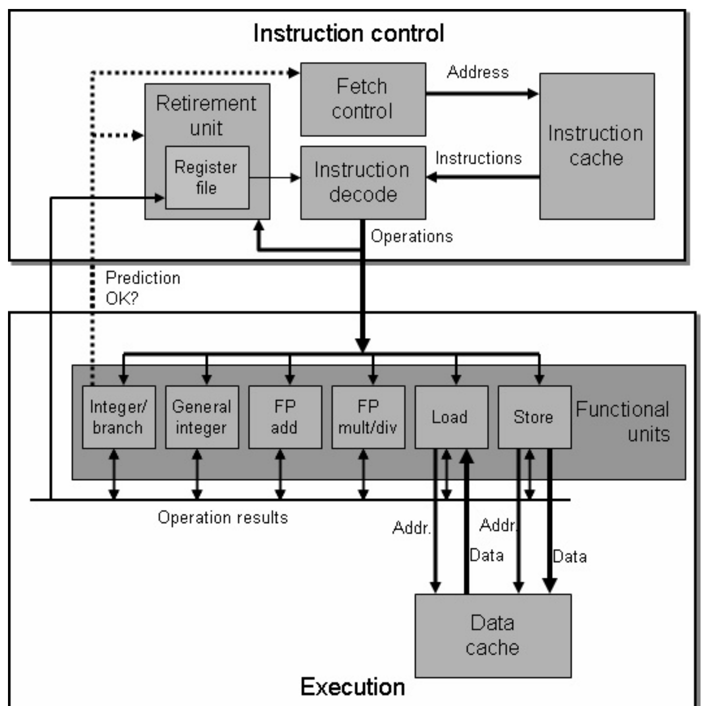
O bloco *Instruction Decode* converte as instruções máquina em micro-instruções. Estas micro-instruções apenas realizam operações elementares tais como somar dois valores, ler ou escrever na memória, etc. Assim uma instrução do tipo

```
add $4, %eax
```

é convertida numa única micro-instrução, enquanto a instrução

```
add $4, 10(%ebx)
```

é convertida em 3 micro-instruções: leitura da memória, adição e escrita em memória.



A unidade de execução contem 6 unidades funcionais sendo portanto designada de super-escalar:

Integer/branch – realiza as operações inteiras e lógicas mais simples e processa saltos condicionais;

General integer – realiza todas as operações inteiras, incluindo divisões e multiplicações;

FP add – realiza operações simples em vírgula flutuante;

FP mult/div – realiza multiplicações e divisões em vírgula flutuante;

Load – faz leituras de memória – inclui um somador para cálculo de endereços;

Store – faz escritas em memória – inclui um somador para cálculo de endereços;

Uma das receitas para o aumento do desempenho do processador consiste em escalonar as micro-instruções nas várias unidades funcionais com o objectivo de as manter com a maior taxa de ocupação possível com trabalho útil. Daí o recurso a execução especulativa (previsão dos saltos condicionais) e execução fora de ordem. Estas duas funcionalidades exigem que a máquina identifique claramente as dependências entre as várias instruções que podem estar a ser executadas simultaneamente para garantir que uma instrução apenas inicia quando os seus *inputs* estão disponíveis e que o resultado de uma instrução só é escrito em memória ou num registo quando for garantido que a instrução deve ser executada. Consulte a secção 5.7 do livro para pormenores sobre como etiquetar os resultados e a renomeação dos registos.

A tabela seguinte apresenta o desempenho das várias unidades funcionais em ciclos do relógio conforme descrito na literatura da Intel.

Operação	Latência	*Issue*
Adição inteira	1	1
Multiplicação inteira	4	1
Divisão inteira	36	36
Adição FP	3	1
Multiplicação FP	5	2
Divisão FP	38	38
Load (Cache hit)	3	1
Store (Cache hit)	3	1

A 2ª coluna indica quantos ciclos são necessários para completar uma operação. A 3ª coluna indica de quantos em quantos ciclos a unidade funcional pode começar uma nova operação, estando relacionada com o nível de encadeamento (*pipelining*) da mesma. Assim uma adição inteira demora apenas um ciclo. Uma multiplicação inteira demora 4 ciclos, mas a unidade funcional pode iniciar uma nova multiplicação a cada ciclo, isto é, tem 4 níveis de encadeamento. Já a divisão inteira demora 36 ciclos e a unidade apenas processa uma operação em cada instante, isto é, não é encadeada.

➢ Análise detalhada do funcionamento do processador

Consideremos a sequência de código da última versão do programa usado no módulo 5, para multiplicação de inteiros:

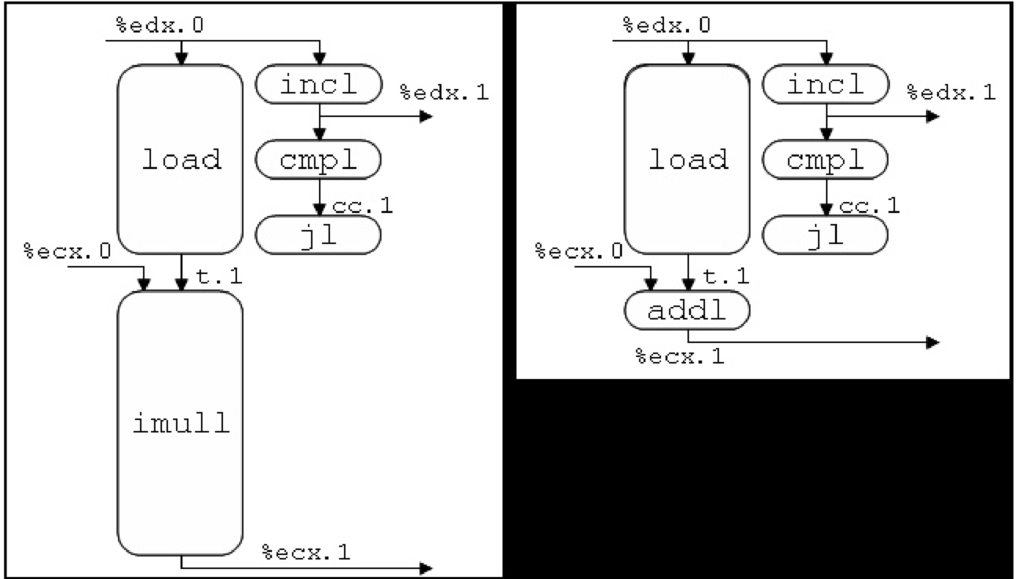
```
.L24:
      imull (%eax, %edx, 4), %ecx      # multiplicação de x por v[i]
      incl %edx                        # i++
      cmpl %esi, %edx                  # compara i com len
      jl .L24                          # se <  repete
```

Para cada iteração do ciclo estas instruções são convertidas em micro-instruções. Se os dados de entrada de uma instrução dependerem de cálculos de uma iteração anterior estes são etiquetados com um número correspondente à iteração em que são gerados. Na tabela seguinte apresentam-se as micro-instruções para a primeira iteração:

Instrução *assembly*	Micro Instrução
`imull (%eax, %edx, 4), %ecx`	`load (%eax, %edx.0, 4)  ->  t.1`
	`imull t.1, %ecx.0  ->  %ecx.1`
`incl %edx`	`incl %edx.0  ->  %edx.1`
`cmpl %esi, %edx`	`cmpl %esi, %edx.1  ->  cc.1`
`jl .L24`	`jl cc.1`

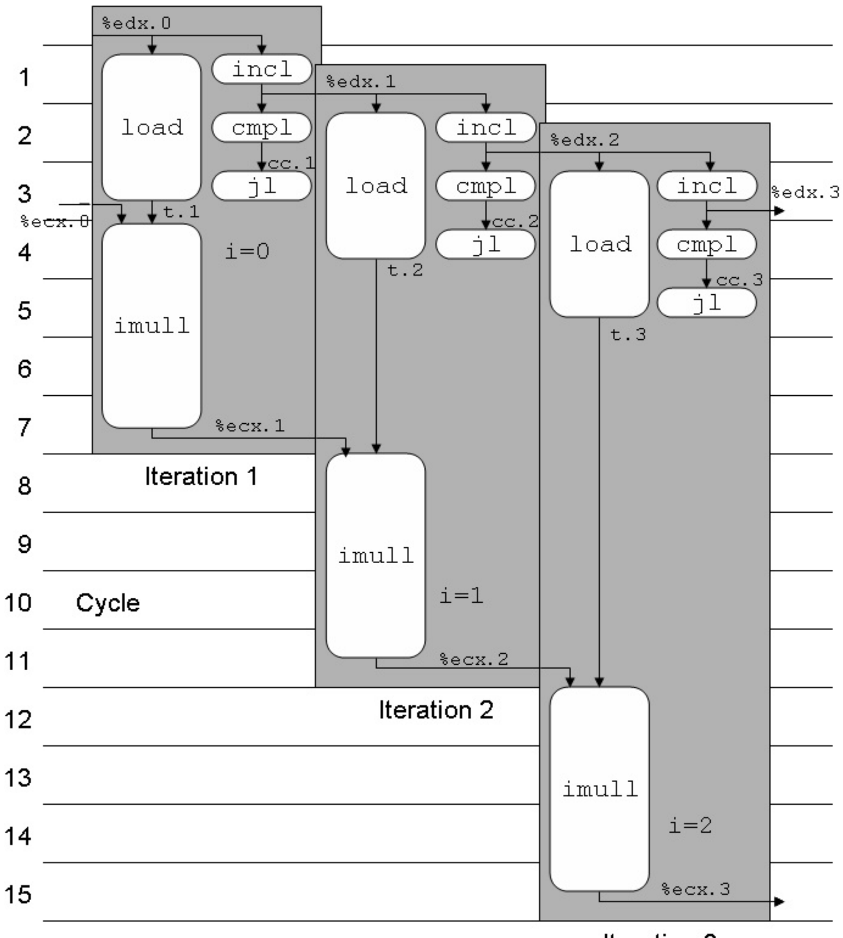
A próxima figura ilustra a execução destas operações elementares como um grafo de computação. O tempo (ciclos de relógio) apresenta-se na vertical; operações que possam ocorrer simultaneamente aparecem na mesma linha. A colocação das operações depende da

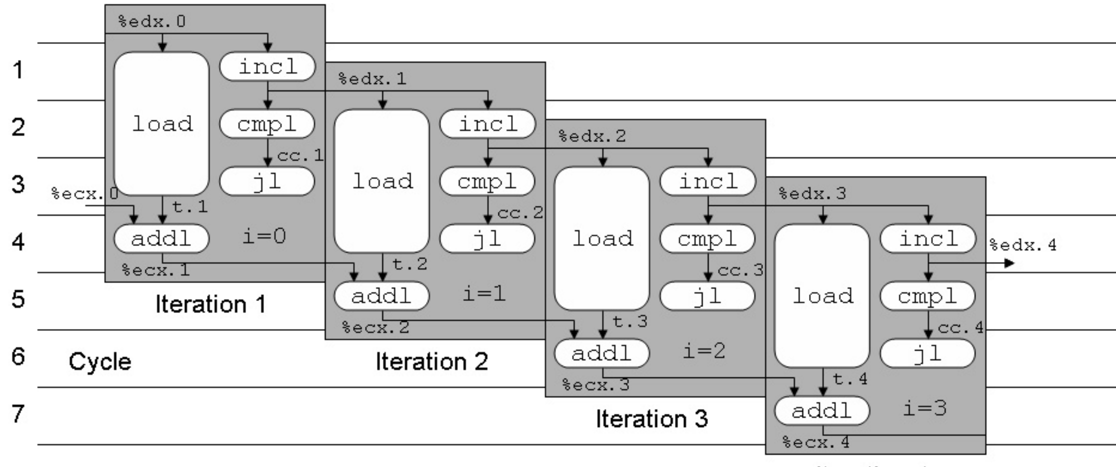
disponibilidade dos dados de entrada e da disponibilidade de unidades funcionais para as



executarem. Os grafos correspondem à multiplicação e adição.

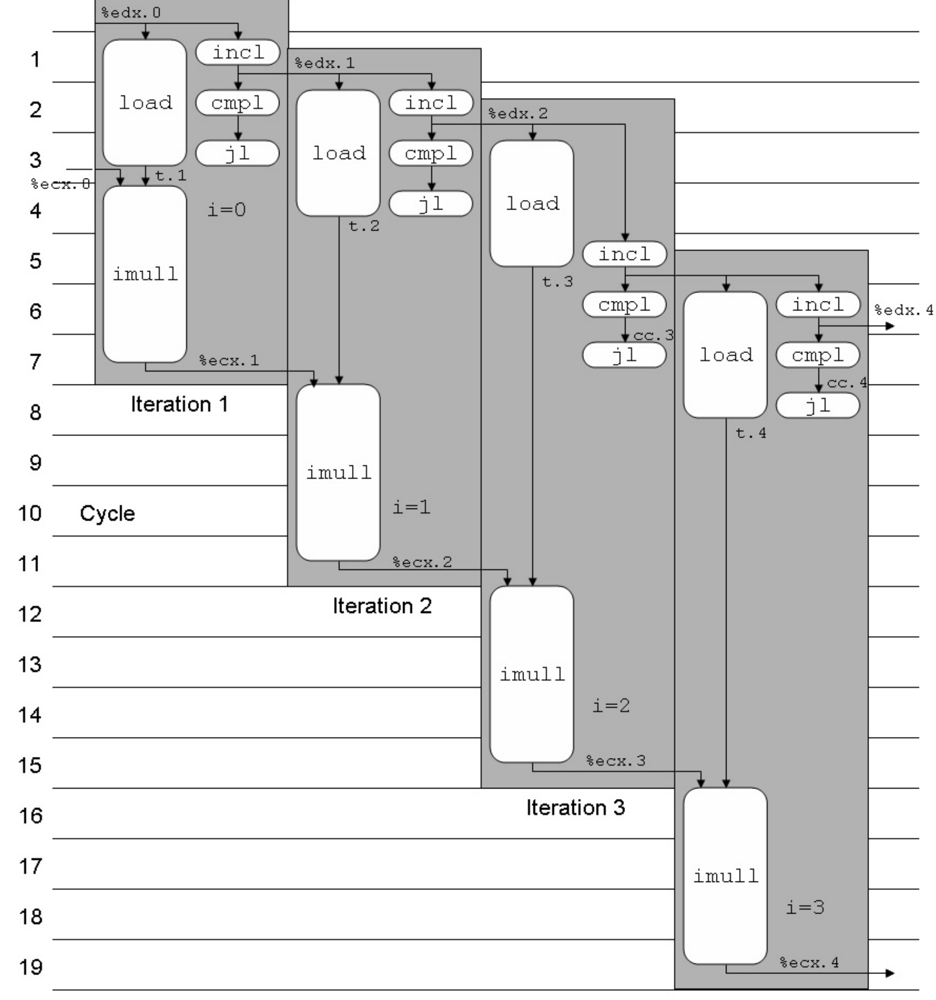
Se os recursos fossem ilimitados, isto é, se existisse um número ilimitado de unidades funcionais sempre disponíveis, então o nosso processador ficaria apenas restringido pelas dependências de dados, resultando em escalonamentos como são apresentados nas figuras seguintes. Note que a multiplicação de inteiros atinge um CPE teórico de 4.0 e a adição de 1.0.

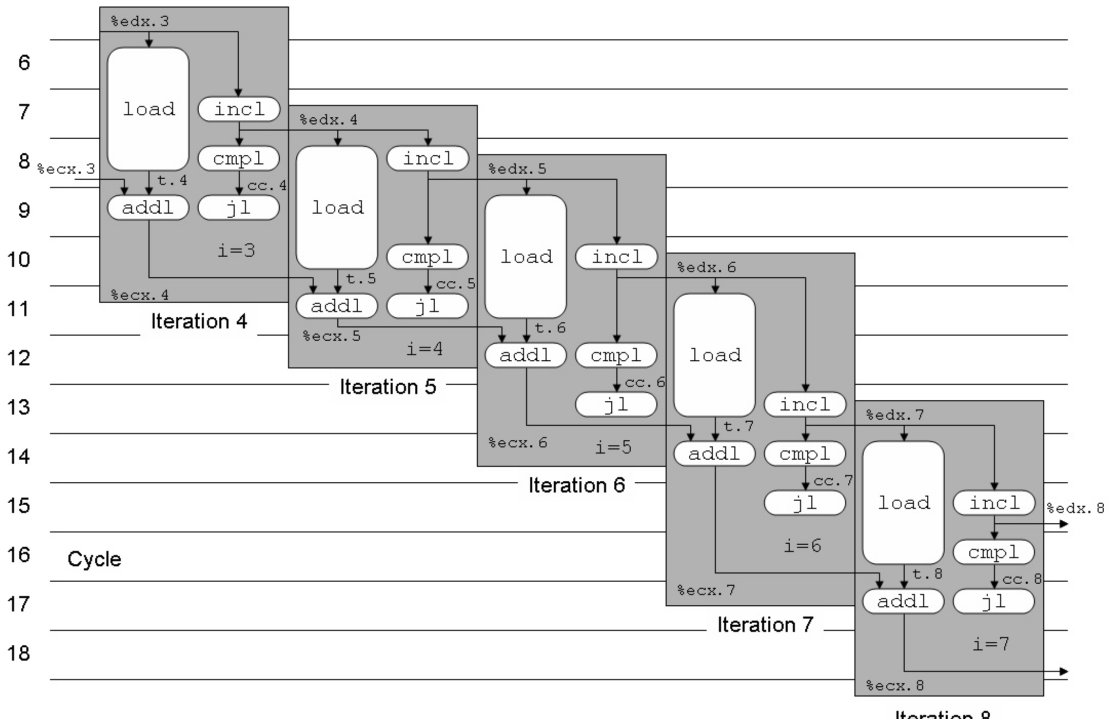




No entanto, o nosso processador apenas tem 2 unidades funcionais capazes de realizar operações com inteiros, sendo que apenas uma pode realizar multiplicações e a outra é também responsável pelos saltos condicionais. Assim a situação verificada no ciclo 4 da figura anterior não é possível, pois estão representadas simultaneamente 4 operações com inteiros.

A micro-arquitectura apresentada anteriormente condiciona o escalonamento das operações aos diagramas apresentados a seguir. Note que o CPE da multiplicação de inteiros continua a ser 4.0, limitado pela latência da operação de multiplicação e pela dependência de dados em %ecx, mas a adição apresenta agora um CPE teórico de 2.0.





4. *Medição do desempenho de f()*

Em http://gec.di.uminho.pt/lesi/ac2/práticas/docs/ac2Mod6.tgz pode descarregar o código necessário para realizar este modulo. Pode descompactar este ficheiro escrevendo

```
tar xvzf ac2Mod6.tgz
```

Este código corresponde ao utilizado no Módulo 5 com pequenas alterações.

Comece por preencher a tabela apresentada no fim deste módulo para as 4 versões (inteiros e FP, adição e multiplicação), para o vector com 2K elementos e para:

- a versão inicial sem qualquer optimização introduzida pelo compilador ou pelo programador (ficheiro `function1.c`)
- a versão final do Módulo 5 correspondente à utilização de optimizações pelo compilador (`-O2`), remoção da invocação de procedimentos do ciclo e utilização de uma variável local para acumulação do resultado (ficheiro `function4.c`).

➢ Redução dos custos dos ciclos

O desempenho da adição de inteiros está limitada pelo facto de cada iteração do ciclo necessitar de realizar 4 operações inteiras mas apenas existirem 2 unidades funcionais capazes de o fazer. No entanto, duas destas operações têm a ver apenas com determinar se o ciclo deve continuar ou não. Se em cada iteração do ciclo processarmos 2 elementos do vector em vez de apenas um, conseguimos reduzir o custo associado à implementação do ciclo – a esta optimização chamamos *loop unrolling*.

Estude cuidadosamente a versão do código apresentada em `function5.c` e preencha a tabela para a versão Unroll x2.

Altere este ficheiro de forma a processar primeiro 3 elementos do vector em cada iteração e depois 4. Tenha especial cuidado com a definição de `limit`. Anote os resultados na tabela nas linhas etiquetadas com unroll 3x e unroll 4x. O que conclui sobre esta optimização?

Note que esta técnica pode resultar em piores desempenhos, dependendo da micro-arquitectura do processador e do próprio algoritmo a ser tratado. O ciclo final, devido ao facto do tamanho do vector não ser necessariamente um múltiplo do número de elementos processado em cada iteração, representa sempre um custo adicional.

Alguns compiladores fazem automaticamente o *loop unrolling*. O `gcc` necessita de ser invocado com a opção –`funroll-loops` para o fazer. Altere a `Makefile` para utilizar esta opção (variável de compilação `F_OPTIONS`) e construa uma nova versão do programa usando o ficheiro `function4.c` Preencha a linha apropriada da tabela.

O que conclui sobre esta opção?

➢ Aumento do paralelismo

As versões do programa que fazem multiplicações não apresentaram melhorias de desempenho com a técnica anterior. A latência da multiplicação impõem-se ao longo da execução. Apesar das unidades funcionais responsáveis pela multiplicação (int e fp) serem encadeadas, este programa não lhes permite tirar partido deste facto, pois estamos a acumular todas as multiplicações numa única variável local. Quer isto dizer que enquanto esta variável não tiver sido actualizada com o resultado da última multiplicação o processador não pode iniciar a próxima.

Loop splitting consiste em subdividir os elementos a processar em subconjuntos e calcular o resultado separadamente para cada subconjunto. O resultado de cada subconjunto é depois combinado numa operação final. Tomando cada subconjunto como independente dos outros é agora possível iniciar multiplicações sempre que a respectiva unidade funcional o possa fazer – a dependência de dados é desfeita. Note que o problema não se levanta com a adição de inteiros porque a sua latência é de 1 ciclo.

Estude cuidadosamente o código apresentado em function6.c. Compile-o e obtenha o CPE para os vários tipos de dados e operações.

Note também que esta optimização só é possível se a operação a realizar for associativa. No mundo digital dos computadores a adição e multiplicação de inteiros é associativa; no entanto, as operações em vírgula flutuante não são associativas devido a limitações da representação utilizada. Os resultados obtidos com o reordenamento das operações podem levar a resultados diferentes devido a arredondamentos, *overflows* e *underflows*.

Estudando as latências e *issue time* das unidades funcionais responsáveis pelas multiplicações parece-lhe que ainda seria possível diminuir o CPE aumentando o nível de *loop unrolling e splitting*?

Versão	Inteiros		Virgula Flutuante	
	+	*	+	*
Sem optimização				
function4.c				
function5.c Unroll 2x				
Unroll 3x				
Unroll 4x				
-funroll-loops				
function6.c - Unroll 2x ; 2 way parallelism				