Arquiteturas Vetoriais

We call these algorithms *data parallel algorithms* because their parallelism comes from simultaneous operations across large data sets, rather than from multiple threads of control

W Daniel Hillis & Guy Steele, *Data Parallel Algorithms*, CACM 1986

Se você fosse arar um campo, o que usaria:
dois bois fortes ou 1024 galinhas?

Seymour Cray, pai do supercomputador

The most efficient way to execute a vectorizable application is a *vector processor*

Jim Smith, ISCA 1994

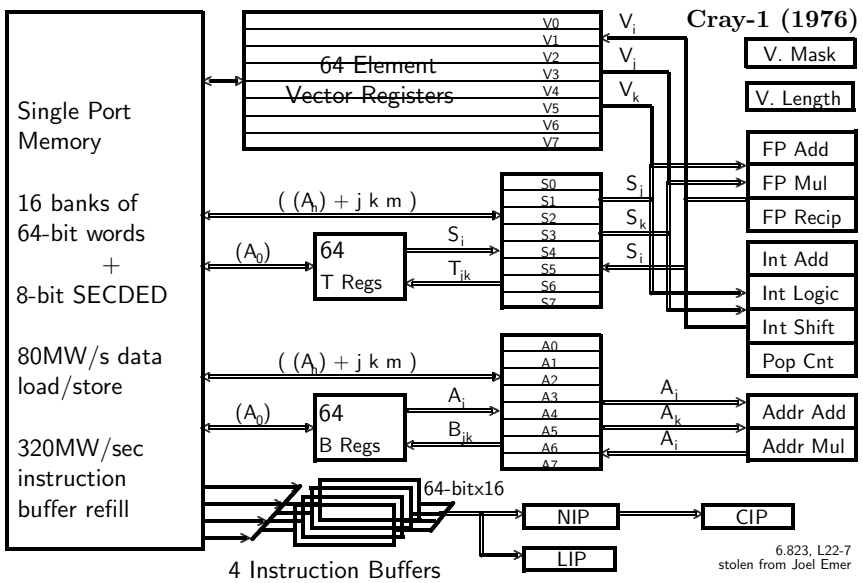
Modelo de Máquina – Cray-1

– 8 registradores vetoriais (64 elementos de 64 bits, `double`)
– 8 registradores escalares + 64 registradores de "rascunho"
– unidades funcionais segmentadas (FP add, mul, recíproca (1/x))
– unidades de inteiros para lógica e aritmética
– 8 registradores de endereço + 64 registradores de "rascunho"
– 2 unidades para cálculo de endereços
– 4 *buffers* de instrução (*loop cache* primitiva)

– memória organizada em 16 bancos de palavras de 64 bits
– toda em SRAM, sem memória virtual
– com detecção e correção de erros

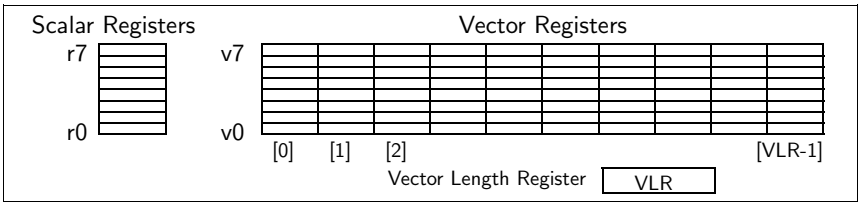
– relógio 80MHz, mais rápido dentre *mainframes* durante 10 anos
– microprocessadores alcançaram os 80MHz após 25 anos

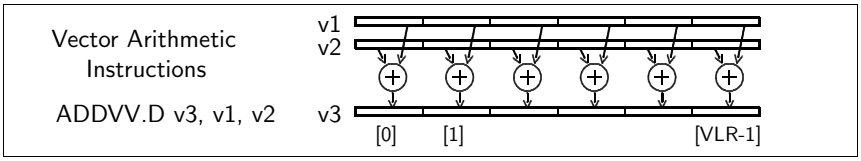
– seu processador escalar é muito eficiente

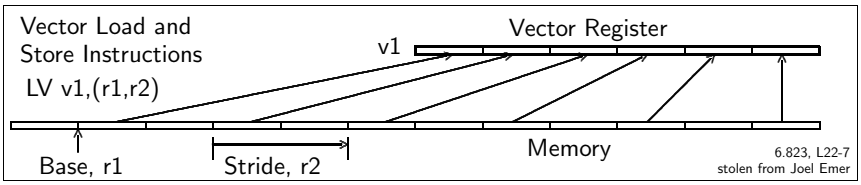


memory bank cycle: 50ns processor cycle: 12.5ns (80MHz)

Modelo de Programação



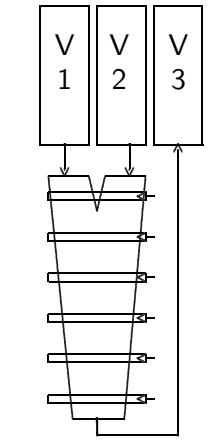




Execução de Aritmética Vetorial

– Unidades funcionais segmentadas
– iniciam uma nova operação a cada ciclo
– operandos são dois vetores: sufixo `VV`: `addVV`
– ou vetor e escalar: sufixo `VS`: `addVS`

– controle mais simples que OoO:
  ∃m riscos estruturais e riscos de dados
  MAS
  ∄ dependência entre os elmtos de um vetor
  ⇝ compilador/programador detectam e resolvem dependências



v3 ← v1 ∗ v2

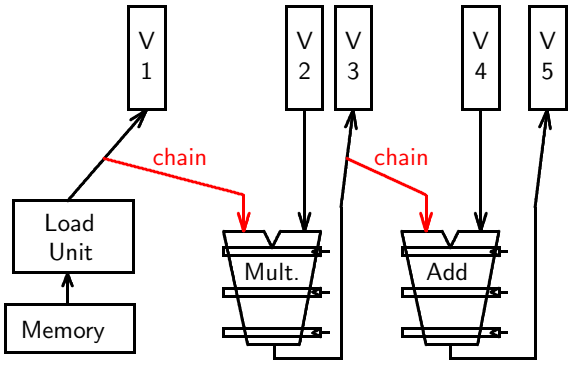
6.823, L22-7
stolen from Joel Emer

Encadeamento (*Chaining*)

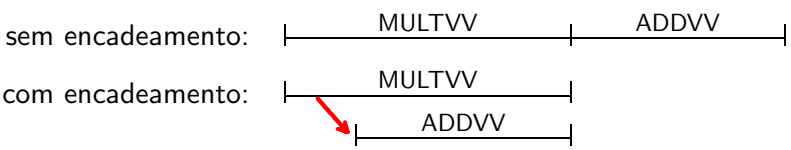
LV v1

MULVV v3,v1,v2

ADDVV v5, v3, v4

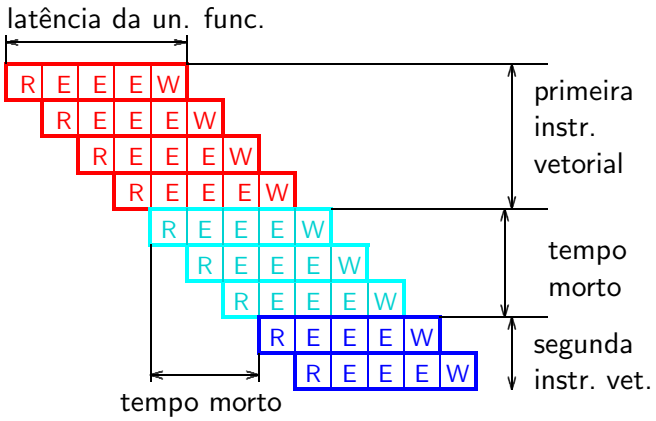


6.823, L22-7
stolen from Joel Emer



Inicialização

Execução de instrução vetorial incorre em duas penalidades:
– latência da unidade funcional (núm. de estágios)
– tempo morto ou de recuperação (intervalo entre 2 instruções)



VMIPS

– 8 registradores vetoriais (64 elementos de 64 bits, `double`)
– unidades funcionais segmentadas (FP add, mul, div)
– unidade de memória para leitura e escrita de vetores
– unidades de inteiros para lógica e aritmética como no MIPS escalar
Latências [ciclos]: FPadd 6, FPmul 7, FPdiv 20, Vload 12

```
XXX = {add,sub,mul,div}; YYY = {sub,div}

XXXvv.d   v1,v2,v3      // double vetor-vetor
XXXvs.d   v1,v2,f0      // double vetor-escalar
YYYsv.d   v1,f0,v2      // double escalar-vetor

lv        v1,r1         // load vector v1, ender fonte em r1
sv        v1,r1         // store vector v1, ender destino em r1
```


Exemplo de Execução – DAXPY (escalar)

Double precision a*X plus Y
no MIPS escalar

```
for (i=0; i<64; i++)
  Y[i] = a*X[i] + Y[i];
```

```
       l.d     f0,a         ; escalar a
       daddiu  r4,rX,512    ; limite=64*8
lasso: l.d     f2,0(rX)     ; f2 ← X[i]
       mul.d   f2,f2,f0     ; a*X[i]      depend em f2
       l.d     f4,0(rY)     ; f4 ← Y[i]
       add.d   f4,f4,f2     ; a*X[i] + Y[i]
       s.d     f4,0(rY)     ; Y[i] ← a*X+Y    dep em f4
       daddiu  rX,rX,8      ; X++
       daddiu  rY,rY,8      ; Y++
       dsubu   r20,r4,rX    ; chegou ao limite?
       bnez    r20,lasso
```

Exemplo de Execução – DAXPY (vetorial)

Double precision a∗X plus Y
no MIPS vetorial

```
for (i=0; i<64; i++)
  Y[i] = a*X[i] + Y[i];
```

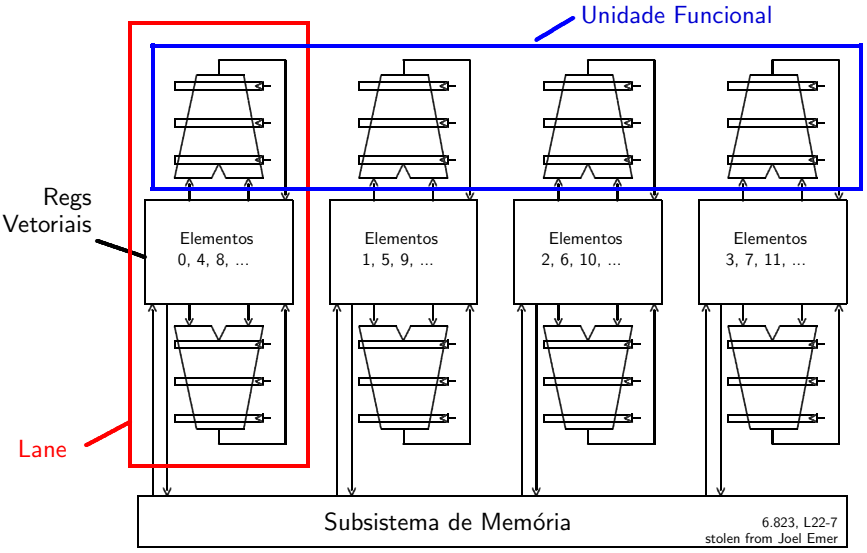
```
l.d      f0,a       ; escalar a
lv       v1,rX      ; v1 ← X
mulvs.d  v2,v1,f0   ; v2 ← a*X
lv       v3,rY      ; v3 ← Y
addvv.d  v4,v3,v2   ; a*X[i] + Y[i]
sv       v4,rY      ; Y ← ...
```

6 instruções contra $2 + 64 \times 9$

escalar: $64 \times \{$`l.d` $\rightsquigarrow$ `mul.d` ; `l.d` $\rightsquigarrow$ `add.d` $\rightsquigarrow$ `s.d`$\}$

vetorial: $\leq$1 bloqueio por instrução

Otimização – várias pistas, execução vetorial paralela

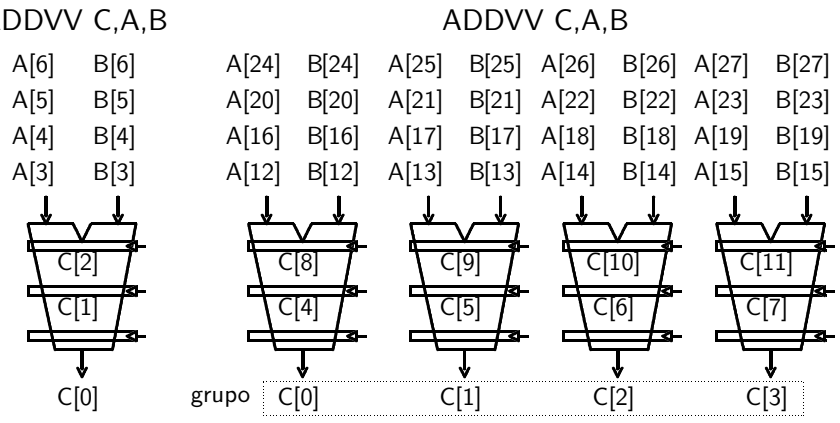


Execução de Aritmética Vetorial Paralela

Execução com uma unidade funcional segmentada

Execução em paralelo com quatro unidades funcionais segmentadas

6.823, L22-7
stolen from Joel Emer



Otimização – *stripmining*

Problema: registradores vetoriais são finitos
Solução: quebrar laços em pedaços do tamanho dos registradores

⇝ *Vector Lenght Register (VLR)* e *Maximum Vector Length* (MVL)

```
for (i=0; i<N; i++)
  Z[i] = a*X[i]+Y[i];
```

```
low = 0;
vl = (N % MVL);
for (j=0; j<(N/MVL); j++) {
  for (i=low; i<(low+vl); i++)
    Z[i] = a*X[i]+Y[i];
  low = low + vl;
  vl = MVL;
}
```

0 VL-1	VL 2VL-1	2VL 3VL-1	3VL 4VL-1	…
N%MVL	MVL	MVL	MVL	

Otimização – execução condicional

Registrador de máscara (*vector mask*) permite execução condicional:
⇝ se `vm[i]=1` então efetua operação *if-conversion*

```
for (i=0; i<64; i++)
  if (A[i] != 0.0)
    C[i] = B[i]/A[i];
```

```
lv v1,ra        ; v1 ← A
lv v2,rb        ; v2 ← B
lv v3,rc        ; v3 ← C
li f0,0.0       ; f0 ← 0.0
snevs v1,f0     ; vm[i] ← (v1[i]≠f0)
divvv v3,v2,v1  ; vm[i]? v3 ← v2/v1
sv v3,rc        ; vm[i]? C ← v3
```

sCCvs = set vectorMask if *v[i] CC scalar* CC ∈ {eq,ne,gt,lt,ge,le}
sCCvv = set vectorMask if *v1[i] CC v2[i]*

se VM ativado (vm[i]≠0, algum i) todas operações são mascaradas

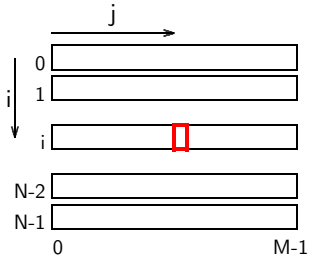
Otimização – acesso a matrizes (i)

```
/* multiplicação de matrizes A=B×C */
for (i=0; i<100; i++)
   for (j=0; i<100; j++) {
      A[i,j]=0.0;
      for (k=0; k<100; k++)
         A[i,j] = A[i,j] + B[i,k]*C[k,j];
   }
```

Quais as equações que definem os endereços das referências aos elementos de A,B,C?

$$e(b,i,j,N,M,s) = b + s(iM+j)$$

b=base = &(A[0,0])
s=`sizeof(elem)`



Otimização – acesso a matrizes (ii)

```
for (i=0; i<100; i++)
   for (j=0; i<100; j++) {
      A[i,j]=0.0;
      for (k=0; k<100; k++)
         A[i,j] = A[i,j] + B[i,k]*C[k,j];
   }
```

Qual a distância entre
$e(C, k, j, N, M, 8)$ e $e(C, k+1, j, N, M, 8)$?

A largura da passada (*stride*) nos elementos de A e B é 1×8;
nos elementos de C é 100×8...

Ocorre conflito no acesso a um banco de memória se

$$\frac{\text{MínMúltComum}(stride, \#bancos)}{stride} < T_{mem}$$

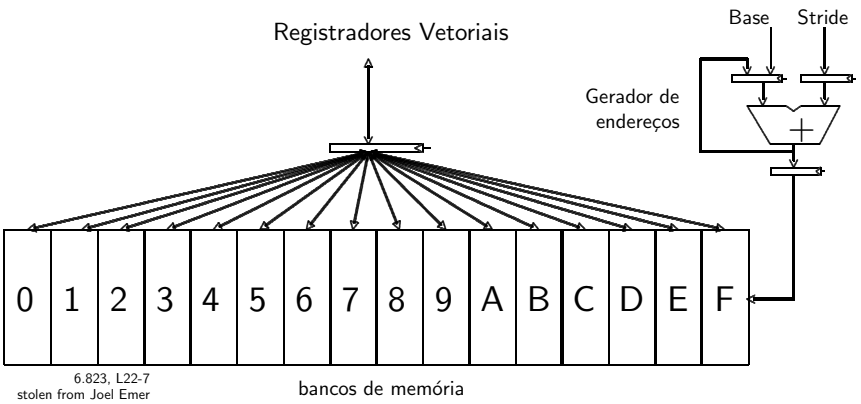
Otimização – acesso a matrizes (iii)

```
for (k=0; k<100; k++) { A[i,j] = A[i,j] + B[i,k]*C[k,j]; }
```

`lvws v1,(r1,r2)` = *load vec with stride* `v[i] ← M[r1 + i*r2]`
`svws (r1,r2),v1` = *store vec with stride* `M[r1 + i*r2] ← v[i]`

```
# este código ignora Max-
VecLenght
lv v1,rb          ; v1 ← B
li rs,800         ; rs ← stride=800
la rc,C           ; rc ← C
lvws v2,(rc,rs)   ; v2 ← M[rc + i * rs]
mulvv v3,v1,v2    ; v3 ← B * C
addvv v4,v4,v3
sv v4,ra          ; A ← A + B * C
```


Circuito para Acessos à Memória



Otimização – *gather–scatter*

Gather carrega um registrador vetorial através de um vetor-índice
`lvi v1,(r1,v2)` = *load vec indexed* `v1[i]` ← `M[r1 + v2[i]]`

Scatter armazena vetor através de vetor-índice
`svi (r1,v2),v1` = *store vec indexed* `M[r1 + v2[i]]` ← `v1[i]`

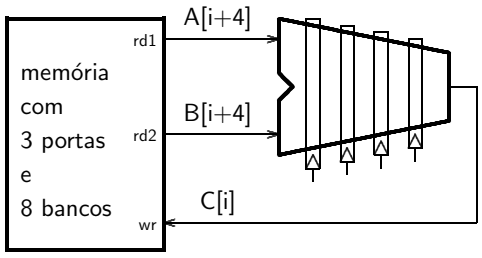
```
for (i=0; i<64; i++)          lv vp,rp          ; vp ← M[rp]
  A[P[i]] =                   lvi va,(ra+vp)    ; va ← M[P[ ]]
    A[P[i]] + C[Q[i]];        lv vq,rq          ; vq ← M[rq]
                              lvi vc,(rc+vq)    ; vc ← M[Q[ ]]
                              addv va,va,vc     ; va ← va+vb
                              svi (ra+vp),va    ; M[P[ ]] ← va
```

Execução com *gather-scatter* é mais lenta que com 'vetores normais'

Memória para Vetores (i)

```
for (i=0; i<N; i++)      ⇒   lv v1,ra
   C[i] = A[i] + B[i];       lv v2,rb
                             addvv v3,v1,v2
                             sv v2,rc
```

Memória deve suportar 2 leituras e 1 escrita na mesma taxa em que somador consome operandos e produz resultados



. somador segmentado com 4 estágios

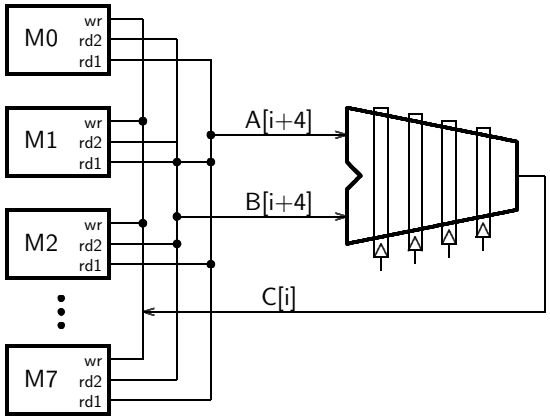
Memória para Vetores (ii)

$$C \leftarrow A + B \;\equiv\; c_i \leftarrow a_i + b_i\,,\;\; 0 \le i < N$$

Supondo que
1 acesso à memória toma 2 ciclos da CPU, a banda da CPU é 6x a banda da memória

memória intercalada em $\geq$ 4 bancos

$\#$ bancos $= 2^n$



Memória para Vetores (iii)

```
for (i=0; i<N; i++)      ⇒  lv v1,ra
   C[i] = A[i] + B[i];      lv v2,rb
                            addvv v3,v1,v2
                            sv v2,rc
```

Alocação dos vetores nos bancos:

M7	a_7	b_5	c_3	-
6	a_6	b_4	c_2	-
5	a_5	b_3	c_1	-
4	a_4	b_2	c_0	-
3	a_3	b_1	-	c_7
2	a_2	b_0	-	c_6
1	a_1	-	b_7	c_5
0	a_0	-	b_6	c_4

Por que a_i é deslocado com relação a b_i?

Memória para Vetores (iv)

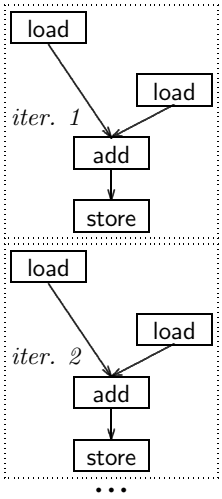
```
for (i=0; i<N; i++)      ⇒  lv v1,ra              ai e bi lidos
   C[i] = A[i] + B[i];      lv v2,rb              ci escrito
                            addvv v3,v1,v2
                            sv v2,rc
```

ciclo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Segm 4	-	-	-	-	-	0	1	2	3	4	5	6	7	-	-
3	-	-	-	-	0	1	2	3	4	5	6	7	-	-	-
2	-	-	-	0	1	2	3	4	5	6	7	-	-	-	-
1	-	-	0	1	2	3	4	5	6	7	-	-	-	-	-
M7	.	.	.	.	.	b_5	b_5	a_7	a_7	c_3	c_3	.	.	.	.
M6	.	.	.	.	b_4	b_4	a_6	a_6	c_2	c_2	.	.	.	.	.
M5	.	.	.	b_3	b_3	a_5	a_5	c_1	c_1	.	.	.	.	.	.
M4	.	.	b_2	b_2	a_4	a_4	c_0	c_0	.	.	.	.	.	.	.
M3	.	b_1	b_1	a_3	a_3	.	.	.	.	.	.	.	.	c_7	c_7
M2	b_0	b_0	a_2	a_2	.	.	.	.	.	.	.	.	c_6	c_6	.
M1	.	a_1	a_1	.	.	.	.	b_7	b_7	.	.	c_5	c_5	.	.
M0	a_0	a_0	.	.	.	.	b_6	b_6	.	.	c_4	c_4	.	.	.

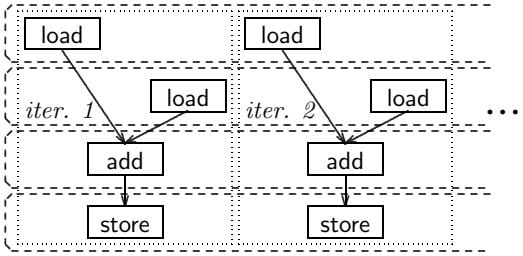
Vetorização automática

```
for (i=0; i<N; i++) { C[i] = A[i] + B[i]; }
```

cód. escalar seq.



código vetorizado



Vetorização implica em enorme reordenação da ordem de execução pelo compilador

necessita análise de dependência nos laços

Loop carried dependences (i)

Loop carried dependences :
utilização de valores produzidos em iterações anteriores
$\rightsquigarrow$ a computação deve ser serializada por causa da cadeia de dependências

Quais as dependências existem neste trecho de código?

```
for (i=999; i>=0; i=i-1)
   x[i] = x[i] + s;
```


Loop carried dependences (ii)

Quais as dependências existem neste trecho de código?

```
for (i=999; i>=0; i=i-1)
   x[i] = x[i] + s;
```

1. em x – leitura e atualização do mesmo endereço x[i] na mesma iteração $\rightsquigarrow$
2. em i – há dependência entre valores de iterações distintas
   MAS i é uma variável de indução e pode ser facilmente eliminada;
   em máq. vetoriais, a variável de indução é implicitamente removida

Loop carried dependences (iii)

Quais as dependências existem neste trecho de código?

```
for (i=0; i<100; i=i+1) {
   A[i+1] = A[i] + C[i];
   B[i+1] = B[i] + A[i+1];
}
```

Loop carried dependences (iv)

Quais as dependências existem neste trecho de código?

```
for (i=0; i<100; i=i+1) {
   c1:  A[i+1] = A[i] + C[i];
   c2:  B[i+1] = B[i] + A[i+1];
}
```

1. em `c1:` – dependência em valor produzido na iteração anterior A[i]→A[i+1] é *loop carried*;
2. em `c2:` – dependência em valor produzido na iteração anterior B[i]→B[i+1] é *loop carried*;
3. a dependência de `c2:` em `c1:` por causa de A[i+1] não é *loop carried* porque mesmo valor é escrito e lido na mesma iteração.

Eliminação de Computações Dependentes (i)

Quais as dependências existem neste trecho de código?

```
for (i=9999; i>=0; i=i-1)
   sum = sum + x[i] * y[i];
```


Eliminação de Computações Dependentes (ii)

```
for (i=9999; i>=0; i=i-1)
   sum = sum + x[i] * y[i];
```

1) expansão do escalar `sum`: *scalar expansion*

```
for (i=9999; i>=0; i=i-1)
   sum[i] = x[i] * y[i];
```

2) redução: *reduction*

```
for (i=9999; i>=0; i=i-1)
   final = final + sum[i];
```

Geralmente, máq. vetoriais e SIMD possuem suporte em HW para reduções eficientes