

A photograph of a server room with rows of black server racks. A semi-transparent blue rectangle is overlaid on the right side of the image, containing white text. The room has a tiled floor and fluorescent lighting on the ceiling.

Visão sobre diferentes ISAs

Cap. 2: “Instructions: Language of the Computer”



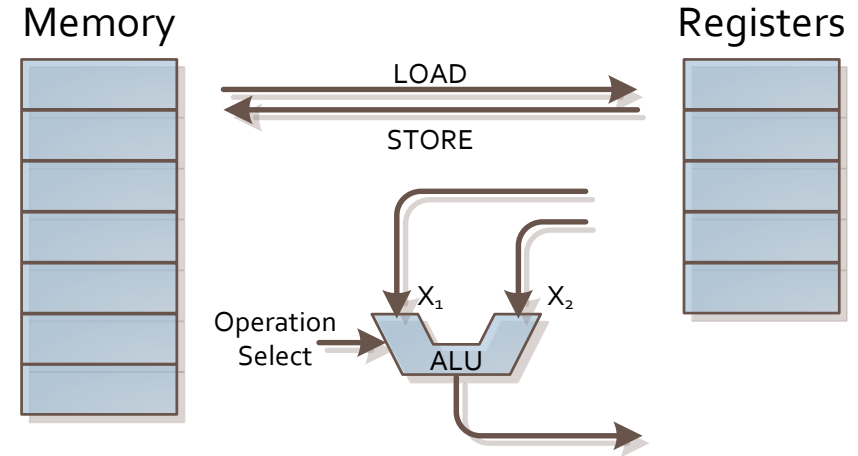
Tipos de instruções/máquinas

Perspetiva do código

Tipos de máquinas/instruções

Load/store or register-register machines

- As arquiteturas do tipo load/store tipicamente têm três operados.
 - Ex: `ADD X2, X1, X0`
`SUBI X3, X4, #4`
- Numa instrução de processamento de dados todos os operandos são registos (ou imediatos)
- Acesso a dados na memória é realizado apenas através de instruções de load/store.



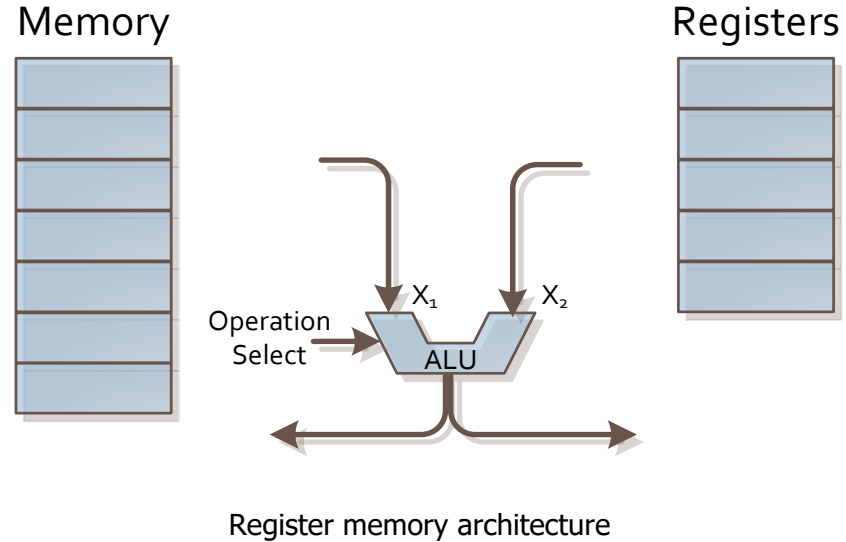
Register-register / Load-Store architecture

Este é o caso do RISC-V estudado até agora nas aulas teóricas

Tipos de máquinas/instruções

Register-memory machine

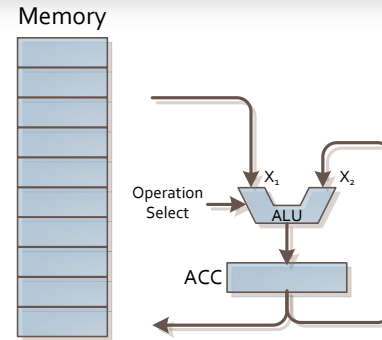
- Nas arquitecturas registo-memória as instruções geralmente têm dois operandos
 - Ex: `ADD X5, X6`
`SUB X7, M[X2+4]`
- Tipicamente permitem múltiplos modos de endereçamento:
 - Imediato
 - Registo
 - Directo
 - Registo Indirecto
 - Indexado
 - Baseado
 - ...



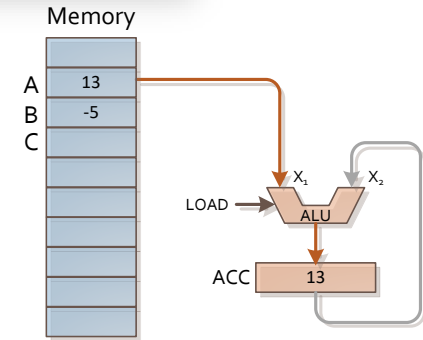
Tipos de máquinas/instruções

Accumulator machine

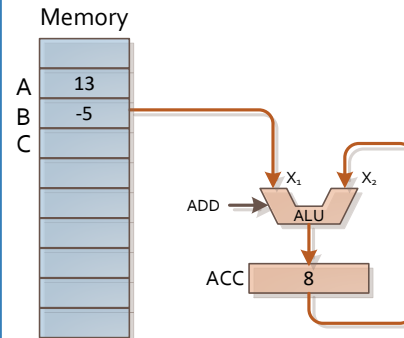
- As arquiteturas do tipo acumulador têm um registo (ou mais) que funcionam como acumulador (ACC).
 - Ex: `LoadA M[100h]`
`AddA M[104h]`
`StoreA M[200h]`
- Um dos operandos e o destino é um dos registos acumuladores; o outro operando é um endereço de memória.
 - Quando existem mais do que um registo acumulador, podem haver operações de transferência entre registos acumuladores.



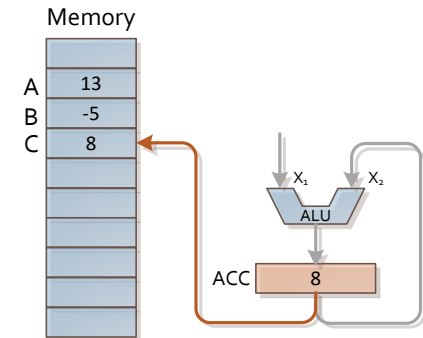
(1) – Accumulator-based architecture



(2) – After loading a value to the accumulator
LOAD A



(3) – After adding a value with the accumulator
ADD B



(4) – Storing the result back into memory
STORE C

Tipos de máquinas/instruções

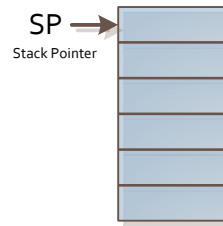
Stack machine

- As arquiteturas do tipo stack normalmente não têm operandos explícitos: o uso da pilha está implícito
- Existem duas operações básicas:
 - PUSH → colocar um elemento na pilha
 - POP → retirar um elemento da pilha

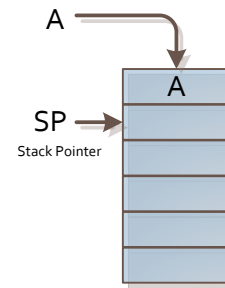
Exemplo 2: $z \times y + x + u$

```
push z
push y
Multiply ; PUSH  $z * y$ 
push x
add      ; PUSH  $(z * y) + x$ 
push u
add      ; PUSH  $(z * y) + x + u$ 
```

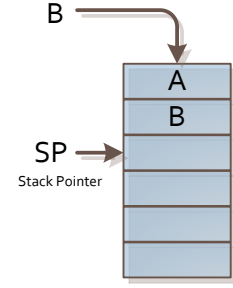
Exemplo 1: $A + B$



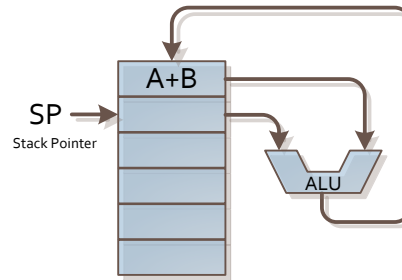
(1) – Empty Stack



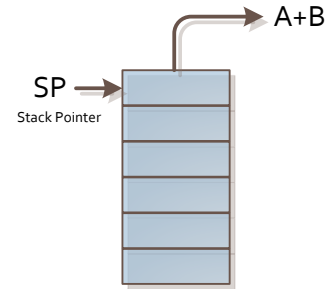
(2) – After pushing a value into the stack
PUSH



(3) – After pushing another value into the stack
PUSH



(4) – After an addition operation
ADD



(5) – After taking a value from the stack
POP



Classes de ISAs

Perspetiva de complexidade

RISC:

REDUCED INSTRUCTION SET COMPUTER

- Baseado numa máquina do tipo load/store
- Palavra de instrução com tamanho fixo (caso típico)
- Suporta um reduzido número de instruções e modos de endereçamento → instruções mais simples
 - Requer mais registos
 - A compilação de código dá origem a mais instruções → programas maiores
 - Maior número de instruções por ciclo de relógio (IPC)

CISC:

COMPLEX INSTRUCTION SET COMPUTER

- Baseado numa máquina registo-memória
- Podem ter instruções com tamanho variável (ex: Intel)
- Suportam um grande número de instruções e de vários tipos e com múltiplos modos de endereçamento diferentes
 - Necessita de menos registos
 - Resultam em programas com um menor número de instruções
 - Menor número de instruções por ciclo de relógio (IPC)
 - A stack tende a ser mais usada

Classes de ISAs


Perspetiva de complexidade

RISC:

REDUCED INSTRUCTION SET COMPUTER

CISC:

COMPLEX INSTRUCTION SET COMPUTER



A extração de paralelismo ao nível da instrução (ILP – Instruction Level Parallelism) é deixada para o processador

versus

VLIW:

VERY LONG INSTRUCTION WORD

A extração de paralelismo ao nível da instrução (ILP – Instruction Level Parallelism) é realizada pelo compilador (menos hardware, maior eficiência energética, mas nem todas as dependências podem ser extraídas pelo compilador)



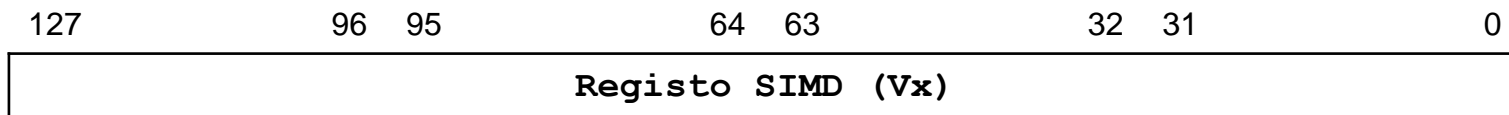
Data-Level Parallelism (DLP)

Exploração do paralelismo ao nível dos dados

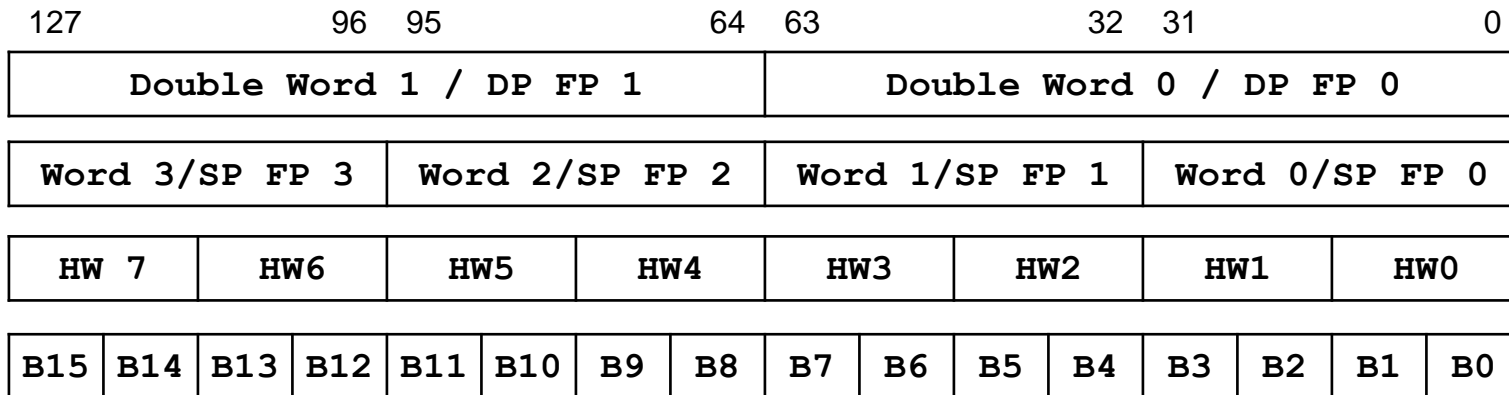
Single Instruction, Multiple Data (SIMD)

Registos SIMD (vetoriais)

- Registos de dimensão fixa (ex: 128B):



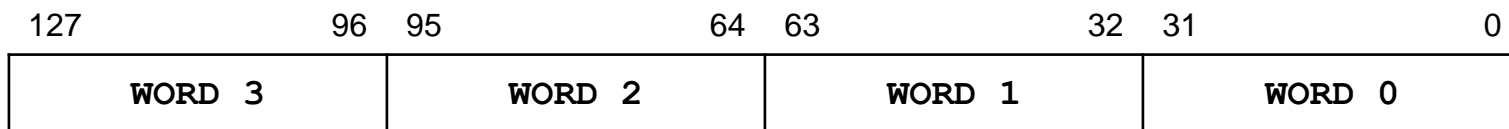
- Dependendo do tamanho dos dados, opera sobre diferentes números de elementos



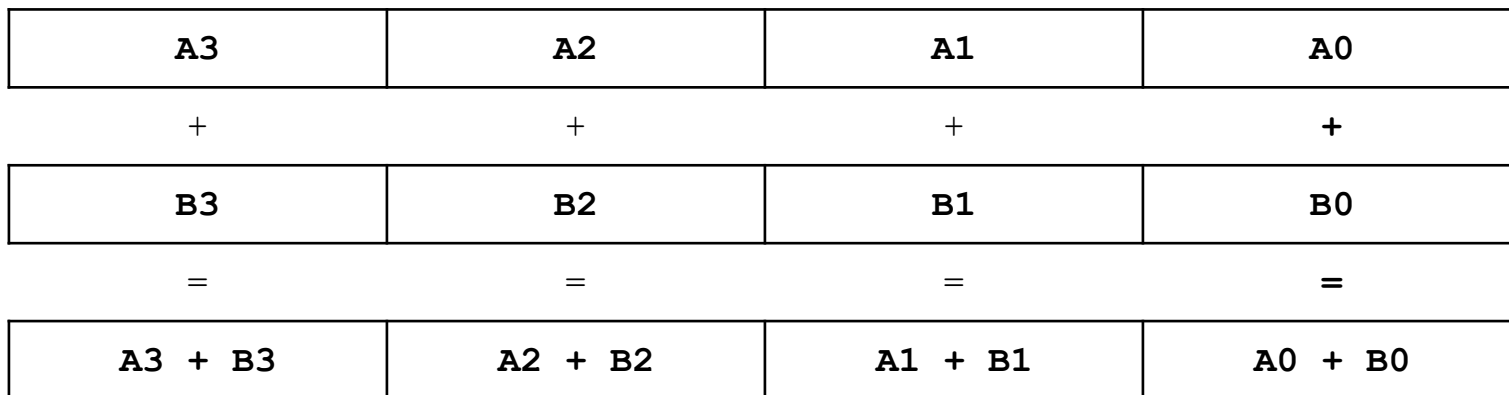
Single Instruction, Multiple Data (SIMD)

Ideia base

- Ler múltiplos elementos de um vetor para registo:



- Operar sobre todos os elementos em paralelo



Single Instruction, Multiple Data (SIMD)

Programação com registos vetoriais / instruções SIMD

1. Desenrolar o código (*loop unrolling*)
2. Empacotar grupos de N instruções em instruções assembly vetoriais

Single Instruction, Multiple Data (SIMD)

Programação com registos vetoriais / instruções SIMD

1. **Desenrolar o código (*loop unrolling*)**
2. Empacotar grupos de N instruções em instruções assembly vetoriais

P. ex.:

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



```
for (i=0; i<(N>>2)<<2; i+=4) {  
    A[i]  =B[i]  +C[i];  
    A[i+1]=B[i+1]+C[i+1];  
    A[i+2]=B[i+2]+C[i+2];  
    A[i+3]=B[i+3]+C[i+3];  
}
```

Arredondamento ao múltiplo
de 4 mais próximo (não excedendo N)

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

Restantes elementos

Single Instruction, Multiple Data (SIMD)

Programação com registos vetoriais / instruções SIMD

1. **Desenrolar o código (*loop unrolling*)**
2. Empacotar grupos de N instruções em instruções assembly vetoriais

Código alternativo

```
for (i=N>>2; i>0; i--, A+=4, B+=4, C+=4) {  
    *A      = *B      + *C;  
    *(A+1) = *(B+1) + *(C+1);  
    *(A+2) = *(B+2) + *(C+2);  
    *(A+3) = *(B+3) + *(C+3);  
}  
  
for (i=N-(N>>2)<<2; i>0; i--, A+=1, B+=1, C+=1){  
    *A = *B + *C;  
}
```

Single Instruction, Multiple Data (SIMD)

Programação com registos vetoriais / instruções SIMD

1. Desenrolar o código (*loop unrolling*)
2. **Empacotar grupos de N instruções em instruções assembly vetoriais**

Código alternativo

```
vector  vect0, vect1;

for (i=N>>2; i>0; i--, A+=4, B+=4, C+=4) {
    vect0 = load_vector(B);
    vect1 = load_vector(C);
    vect0 = add_vector(vect0,vect1);
    store_vector(A,vect0)
}
for (i=N-(N>>2)<<2; i>0; i--, A+=1, B+=1, C+=1){
    *A = *B + *C;
}
```


Single Instruction, Multiple Data (SIMD)

Programação com registos vetoriais / instruções SIMD

1. Desenrolar o código (*loop unrolling*)
2. Empacotar grupos de N instruções em instruções assembly vetoriais

```
      # Assumindo X10=A, X11=B, X12=C, X13=N, X14=i
      sra      x14,x13,2          # i=N>>2
      ble      x14,x0,endfor
vectorfor: vlw      v0,0(x10)      # lê 4 elementos de A
          vlw      v1,0(x11)      # lê 4 elementos de B
          vadd.w    v0,v0,v1      # soma o grupo de elementos
          vsw       v0,0(x12)      # guarda 4 elementos em C
          addi      X13,X13,-4
          addi      x10,x10,-16    # atualiza o valor dos ponteiros
          addi      x11,x11,-16    # atualiza o valor dos ponteiros
          addi      x12,x12,-16    # atualiza o valor dos ponteiros
          bgt       volta
Sai:    # fazer as restantes iterações
```

Single Instruction, Multiple Data (SIMD)

Programação com registos vetoriais / instruções SIMD

1. Desenrolar o código (*loop unrolling*)
2. Empacotar grupos de N instruções em instruções assembly vetoriais

Solução vetorial:

```
vectorfor:  sra      x14,x13,2
            ble      x14,x0,endifor
            vlw      v0,0(x10)
            vlw      v1,0(x11)
            vadd.w   v0,v0,v1
            vsw      v0,0(x12)
            addi     x13,x13,-4
            addi     x10,x10,-16
            addi     x11,x11,-16
            addi     x12,x12,-16
            bgt      volta
```

#instruções: $9(N \gg 2) + 2$ ➡

A solução vetorial necessita de cerca de 1/L instruções, onde $L = \#$ palavras/registo vetorial (neste caso $L=4$)

Solução escalar:

```
vectorfor:  ble      x13,x0,endifor
            lw       x5,0(x10)
            lw       x6,0(x11)
            add      x5,x5,x6
            sw       x6,0(x12)
            addi     x13,x13,-4
            addi     x10,x10,-16
            addi     x11,x11,-16
            addi     x12,x12,-16
            bgt      volta
```

⬅ #instruções: $9N + 1$

Single Instruction, Multiple Data (SIMD)

Instruções predicativas

1. Como resolver problemas do género

```
for (i=0; i<N; i++) {  
    if (Z[i]!=0)  
        A[i]=(B[i]+C[i])/Z[i];  
    else  
        A[i]=0;  
}
```

Single Instruction, Multiple Data (SIMD)

Instruções predicativas

1. Como resolver problemas do género

```
for (i=0; i<N; i++) {  
    if (Z[i]!=0)  
        A[i]=(B[i]+C[i])/Z[i];  
    else  
        A[i]=0;  
}
```

2. Solução: usar instruções predicativas

```
add        x0,x1,x2    # x0 ← x1+x2  
add.cond   x0,x1,x2    # if (cond=true) x0 ← x1+x2; else NOP (no  
operation)
```

Single Instruction, Multiple Data (SIMD)

Instruções predicativas

1. Como resolver problemas do género

```
for (i=0; i<N; i++) {  
    if (Z[i]!=0)  
        A[i]=(B[i]+C[i])/Z[i];  
    else  
        A[i]=0;  
}
```

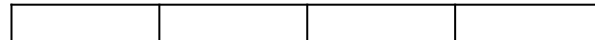
Em **instruções escalares** *cond* é uma flag

Em **instruções vectoriais** *cond* é um registo de predicado com número de elementos igual ao tamanho do vector

2. Solução: usar instruções predicativas

```
vadd          v0,v1,v2      #  $x_0 \leftarrow x_1+x_2$   
vadd.cond     v0,py,v1,v2   # if (cond=true)  $v_0 \leftarrow v_1+v_2$ ; else NOP (no operation)
```

Vector Vx



Px é um registo de predicados que contem 1 bit por cada elemento do vector

Em alguns casos pode-se indicar o que fazer com os elementos inativos (ex, "py/z" coloca zero; "py/m" não altera o valor)

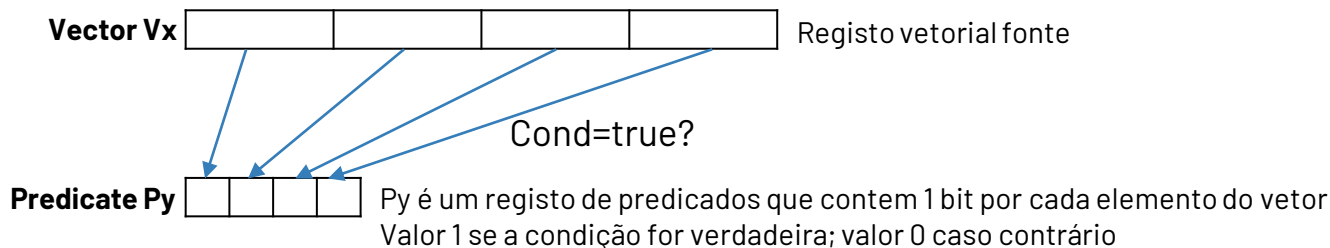
Single Instruction, Multiple Data (SIMD)

Instruções predicativas

- Registo de predicado

Em **instruções vetoriais** *cond* é um registo de predicado com número de elementos igual ao tamanho do vector

`pset cond py,vx` (*predicate set if cond*)



Single Instruction, Multiple Data (SIMD)

Instruções predicativas

Solução vetorial

Assumindo X10=A, X11=B, X12=C, X13=N, X14=Z

```
sra      x15,x13,2
blez     x15,sai
for:     lv      v0,0(x14)
         psetne  p1,v0,x0
         vlw     v1,p1/m,0(x11)
         vlw     v2,p1/m,0(x12)
         vadd.w  v1,p1/m,v1,v2
         vdiv.w  v1,p1/z,v1,v2
         sv      x16,0(x10)
         addi    x10,x10,16
         addi    x11,x11,16
         addi    x12,x12,16
         addi    x14,x14,16
         addi    X15,X15,-1
         bgt     x15,x0,for
```

load Z[i]

o bit i do predicado p1 é 1 se elemento i for v0[i]!=0

load B[i] se Z[i]!=0, se p1[i]=false, não escreve no elemento v1[i]

load C[i] se Z[i]!=0, se p1[i]=false, não escreve no elemento v1[i]

v1=B[i]+C[i] se Z[i]!=0

v1=v1/z se Z[i]!=0, caso contrário (predicado falso) escreve 0

guarda X16 em A[i]

incrementa os endereços dos vectores A, B, C e Z

controlo de loop

```
for (i=0; i<N; i++) {
    if (Z[i]!=0)
        A[i]=(B[i]+C[i])/Z[i];
    else
        A[i]=0;
}
```

sai:

Single Instruction, Multiple Thread (SIMT)

Modelo de programação das GPUs

- O modelo de programação de GPUs é semelhante.
- As grandes diferenças (do ponto de vista da programação) são:
 - Nas GPUs cada elemento do vetor corresponde a uma *thread* diferente
 - O número de elementos do vetor é constante
 - O tamanho do “registro” depende da dimensão de cada palavra de dados

