Processamento paralelo de dados

Instruções SIMD da arquitetura AArch64

João Canas Ferreira

Maio 2019

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

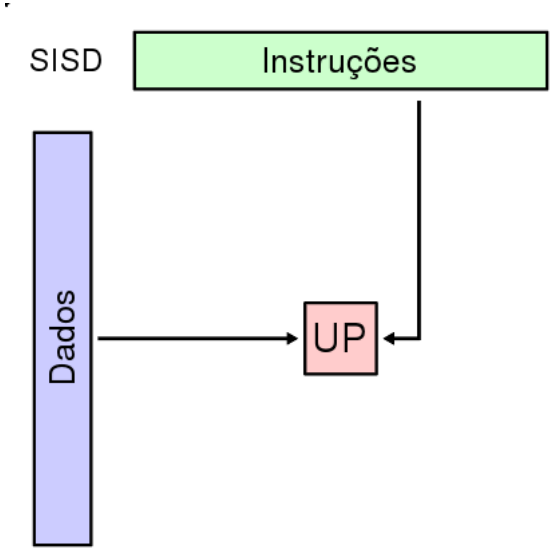
Assuntos

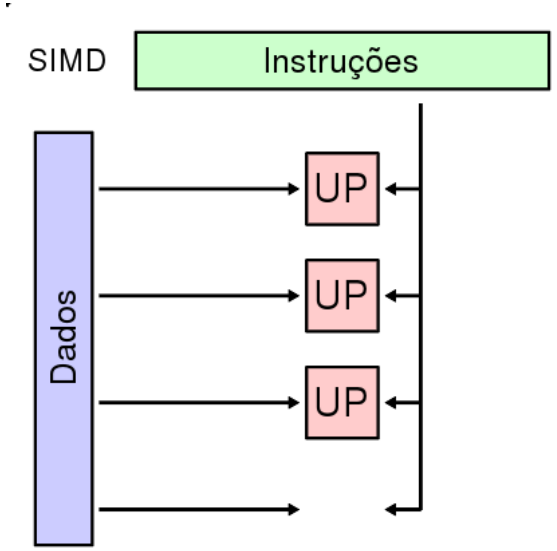
Modelos de execução de instruções

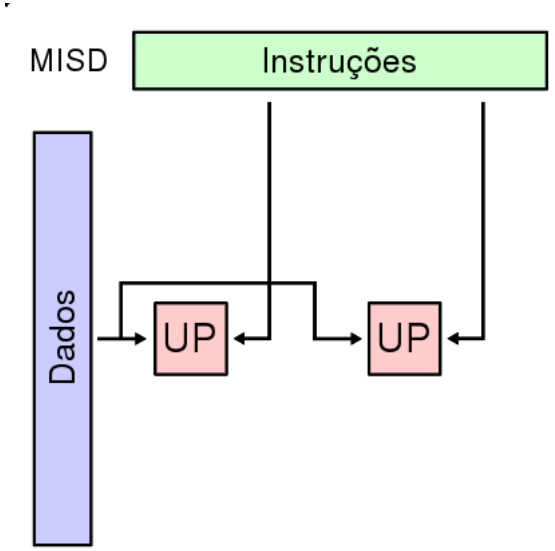
➠ As unidades de processamento podem ser classificadas segundo o número de instruções e de conjuntos de dados tratados em simultâneo:

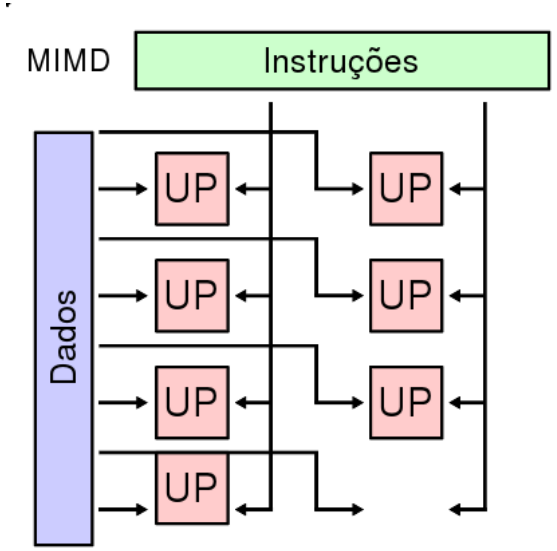
- **SISD:** *Single instruction stream, single data stream*
  processador convencional: 1 instrução processa 1 conjunto de dados
- **SIMD:** *Single instruction stream, multiple data streams*
  1 instrução processa vários conjuntos de dados (processamento vetorial, também usado em GPUs)
- **MISD:** *Multiple instruction streams, single data stream*
  processamento redundante (pouco usado)
- **MIMD:** *Multiple instruction streams, multiple data streams*
  múltiplas instruções (diferentes) processam múltiplos conjuntos de dados (por exemplo, um processador multi-núcleo)

Representação dos modelos de execução



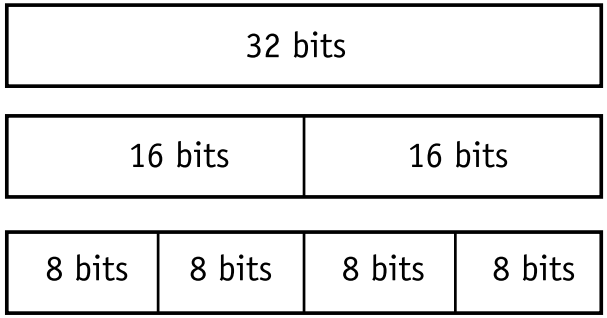


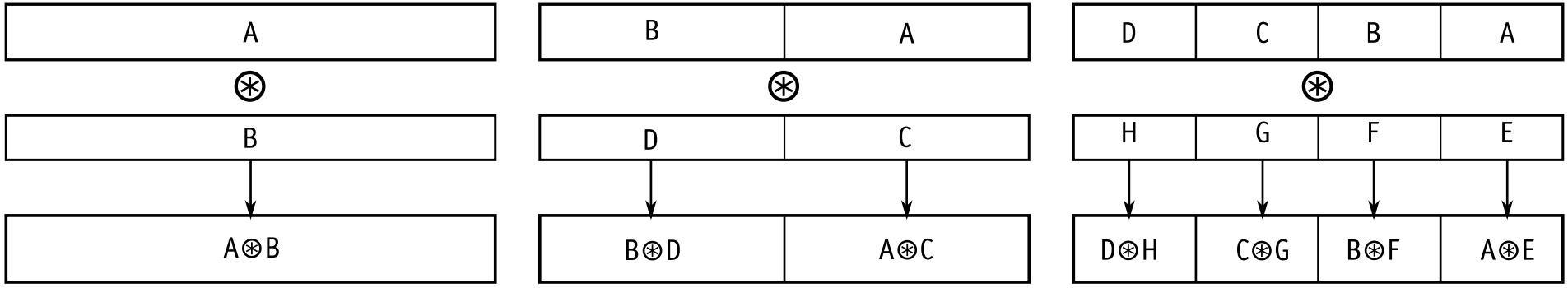




Instruções SIMD: modelo abstrato

➠ Um registo pode ser interpretado como uma unidade ou um vetor com um número fixo (p. ex., 2 ou 4) de registos **independentes**.





➠ Cada elemento do vetor pode ser combinado com o elemento correspondente de outro vetor com uma *única* instrução.

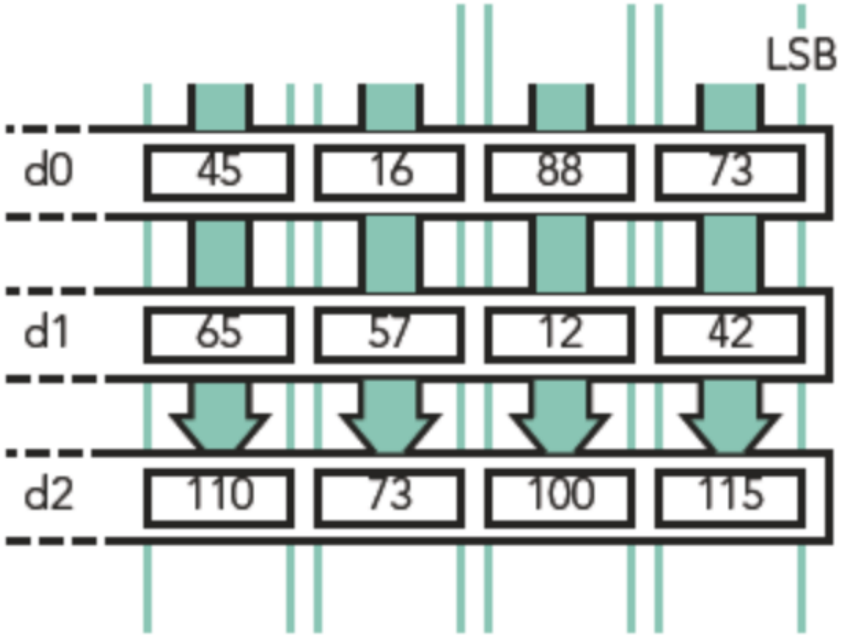
Instruções SIMD: vantagens e desvantagens

- Vantagens
  - Aumento de desempenho
  - Aproveitamento da capacidade de integração (capacidade de integrar número elevado de ALUs e outras unidades de processamento)
  - Paralelismo explícito é mais fácil de aproveitar (operações são naturalmente independentes)
- Desvantagens
  - Requer algoritmos com processamento de dados "uniforme" (todos os elementos do vetor são processados da mesma forma)
  - Necessidade de adaptar a codificação do algoritmo
  - Alguns compiladores têm dificuldade em aproveitar bem este tipo de instruções: recurso a linguagem *assembly*.

Assuntos

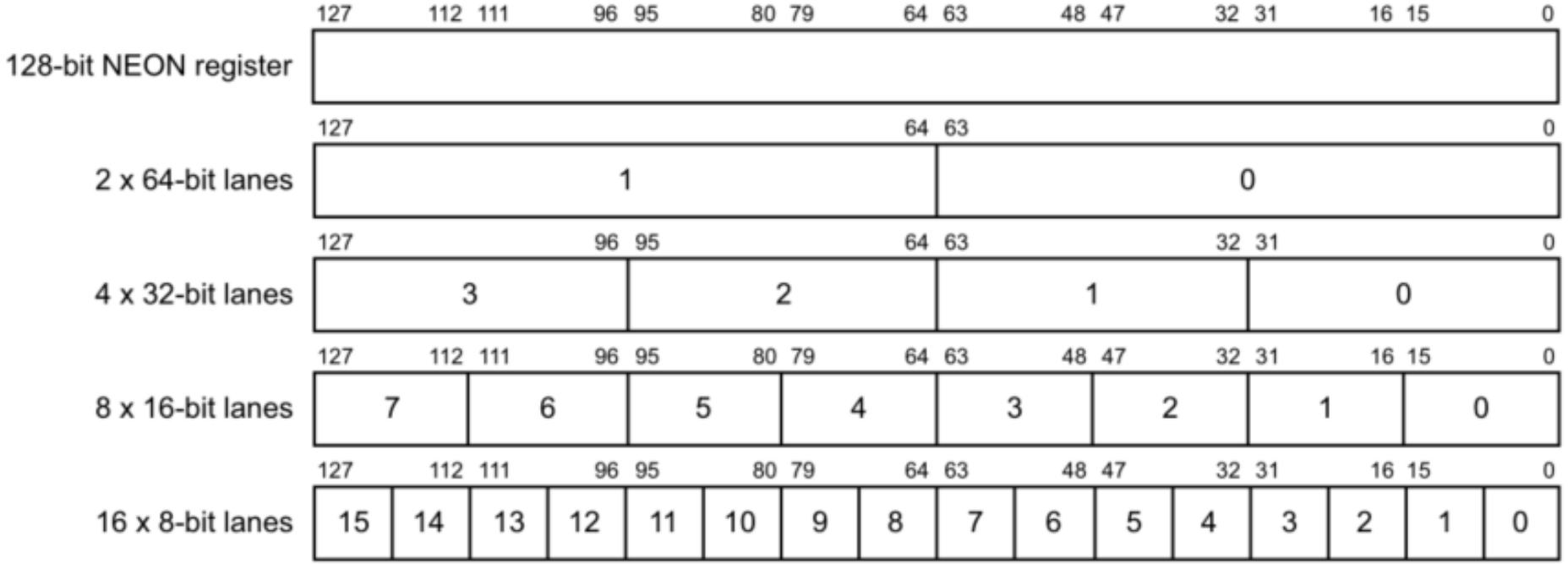
Princípios básicos

- A arquitetura “ARM Advanced SIMD”, as suas implementações e software de apoio são designadas or *tecnologia NEON*.
- Foco: NEON para AArch64
- Suporte para os seguintes tipos de dados:
  - U inteiros sem sinal (*unsigned*) de 8, 16, 32 e 64 bits;
  - S inteiros *com* sinal (cpl 2) 8, 16, 32 e 64 bits;
  - F números em vírgula flutuante de 32 e 64 bits.
- Na terminologia ARM, os cálculos SIMD são realizador por faixa (*lane*)



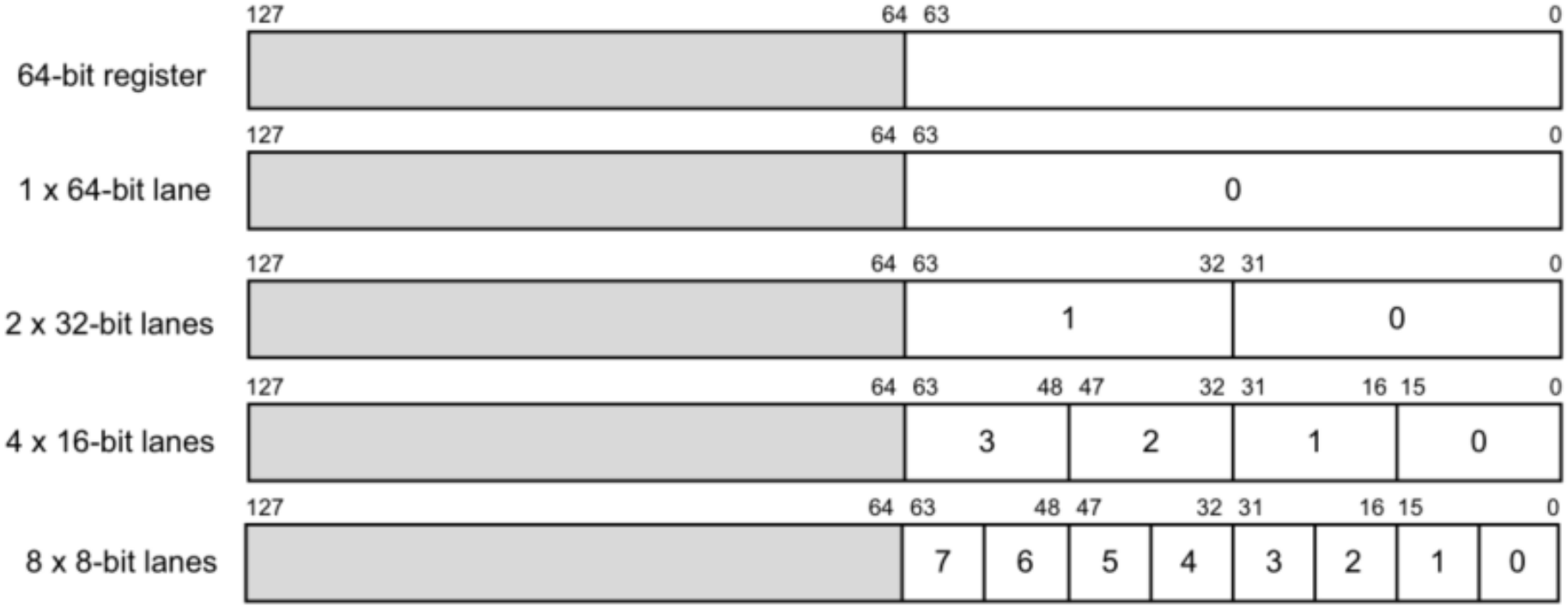
Banco de registo NEON

- O banco de registos tem 32 registos de 128 bits (`quadword`): `V0-V31`.
- É *o mesmo banco de registos* que é usado para os operandos de vírgula flutuante.



Banco de registo NEON (2)

➠ O banco de registos também pode ser considerado como contendo 32 registos de 64 bits (`doubleword`): `D0-D31`.



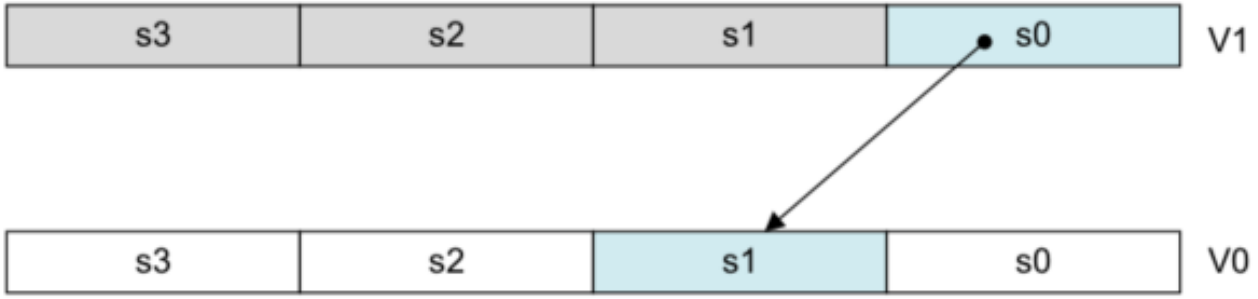
Valores escalares e vetores

- Os registos V<n> podem ser considerados como pequenos vetores (1, 2, 3, 4, 8 ou 16 componentes).
- Geralmente, as instruções SIMD operam sobre vetores, mas algumas instruções usam um componente apenas.

```
<Instrução> Vd.TS[i], Vn.Ts[j]
```

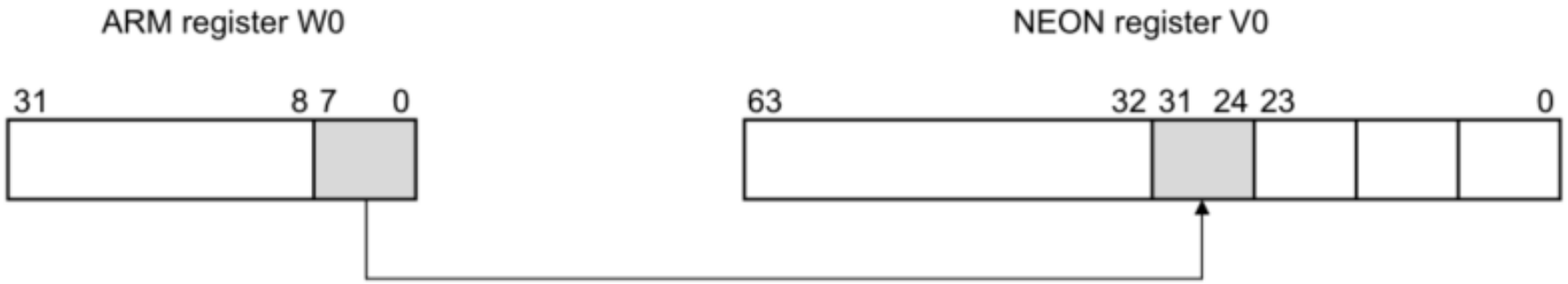
Vd	registo de destino
Vn	registo de origem
Ts	especificador de tipo
i, j	índices dos elementos

- Exemplo: `INS V0.S[1], V1.S[0]`



Valores escalares

- Valores escalares podem ter 8, 16, 32 ou 64 bits
- Instruções que usam valores escalares pode aceder a qualquer elemento do banco de registos
- Exemplo: `MOV V0.B[3], W0`



- Exceção: *instruções de multiplicação* só usam escalares de 16 ou 32 bits e só podem aceder aos primeiros 128 escalares do banco de registos
  - escalares de 16 bits: Vn.H[x], com $0 \leq n \leq 15$, $0 \leq x \leq 7$
  - escalares de 32 bits: Vn.S[x], com $0 \leq n \leq 15$, $0 \leq x \leq 3$

Assuntos

Mnemónicas têm utilização expandida

➠ A mesma mnemónica pode representar várias instruções (i.e., resultam em codificações diferentes).

➠ Exemplo:

- `ADD W0,W1,W2` instrução básica (A64)
- `ADD V0.4H,V1.4H,V2.4H` instrução vetorial (NEON)

➠ Adicionar à mnemónica um dos prefixos S, U ou F para indicar o tipo. (Se fizer sentido.)

- `FADD D0, D1, D2`

➠ A organização do vetor (tamanho do elemento e número de "faixas") é definido pelo qualificador do registo: 8B, 16B, 4H, 8H, 2S, 4S, 2D.

Exemplo: realizar duas adições simultâneas de valores de 64 bits

- `ADD V0.2D,V1.2D,V2.2D` inteiros com sinal
- `FADD V0.2D,V1.2D,V2.2D` VF, precisão dupla

Variantes de instruções NEON

➠ Algumas instruções NEON estão disponíveis nas variantes *Normal*, *Long*, *Wide*, *Narrow* ou *Saturating*.

As variantes *Long*, *Wide* e *Narrow* são indicadas por um sufixo no nome: **L**, **W** e **N**, respetivamente.

A variante *Saturating* usa um dos prefixos **SQ** ou **UQ** consoante se trate de operandos com ou sem sinal, respetivamente.

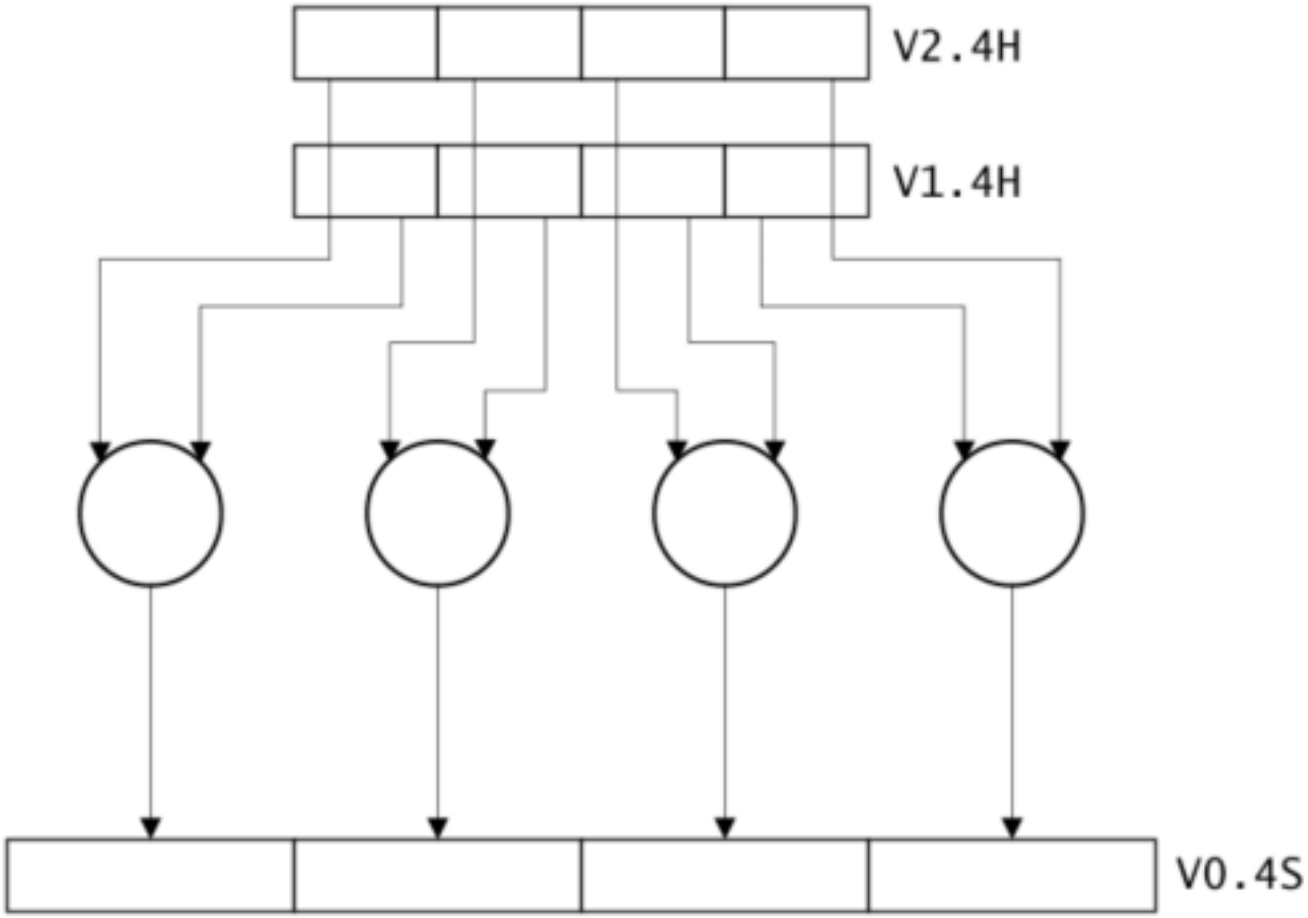
➠ A variante *Normal* opera sobre qualquer tipo de vetor e produz vetores da mesma dimensão (número de componentes) e (geralmente) do mesmo tipo.

➠ Para além das variantes, existem também instruções que operam sobre pares de registos adjacentes (operandos `doubleword` ou `quadword`).

Estas instruções usam o sufixo **P**.

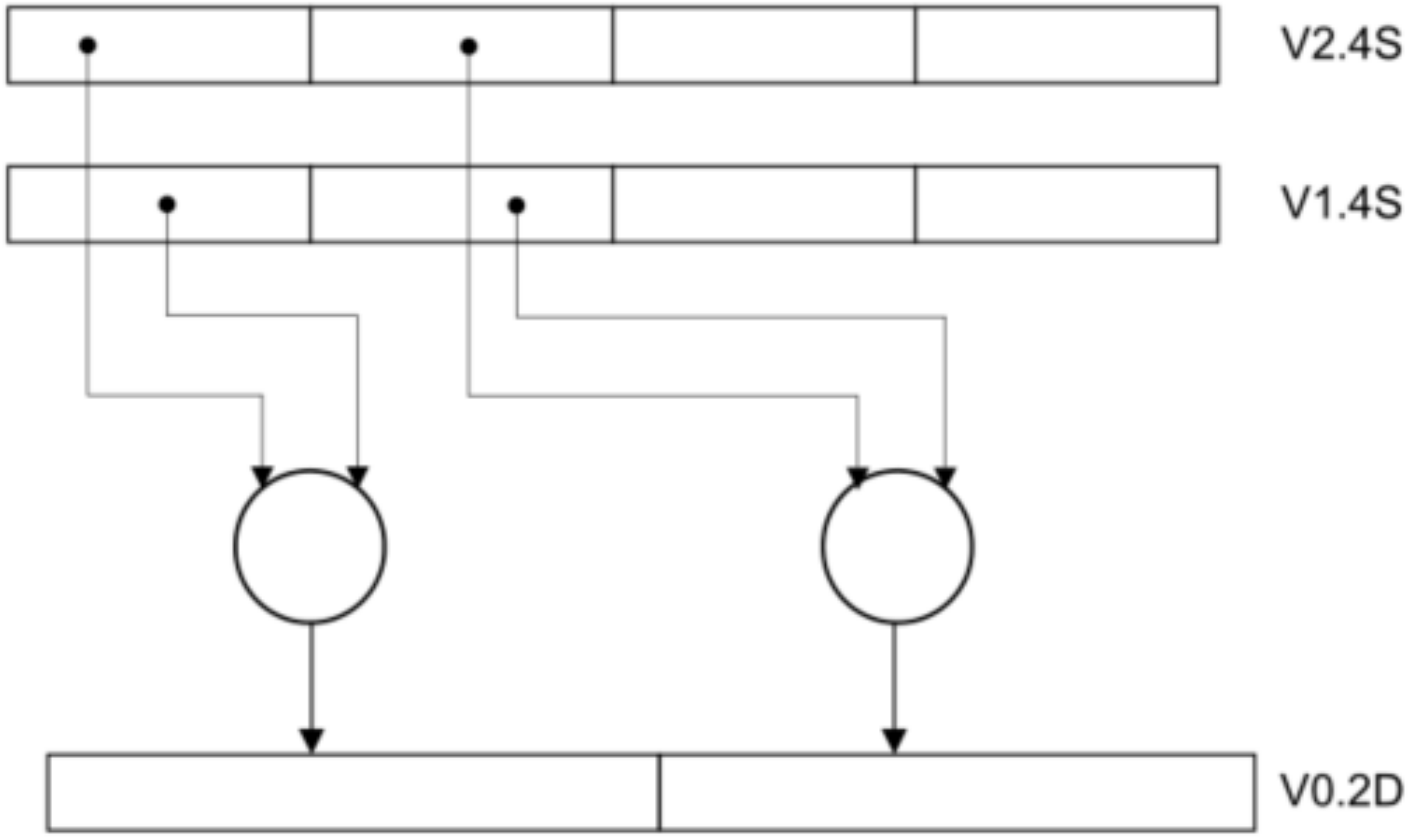
Variante “long”

- Operam sobre vetores de 64 bits e produzem um vetor de 128 bits.
- Os elementos do resultados têm o dobro do tamanho dos operandos.
- Exemplo: `SADDL V0.4S,V1.4H,V2.4H`



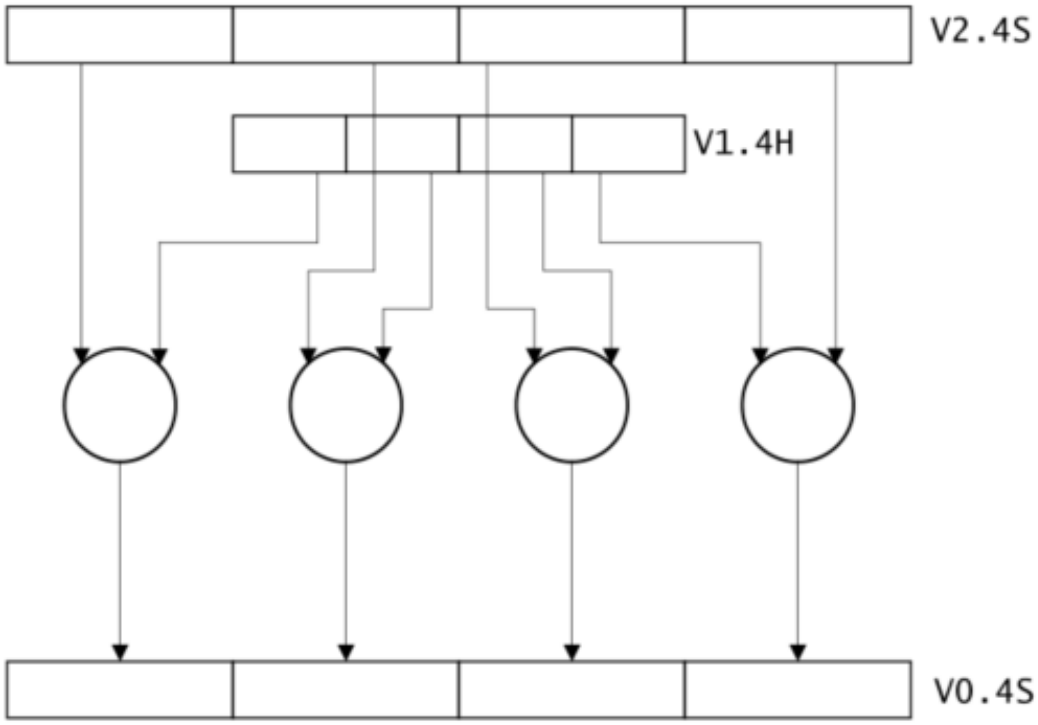
Variante "long" para elementos mais significativos

- Estas operações obtêm os seus dados das pistas superiores dos operandos.
- Os elementos do resultados têm o dobro do tamanho dos operandos.
- Exemplo: `SADDL2 V0.2D,V1.4S,V2.4S`



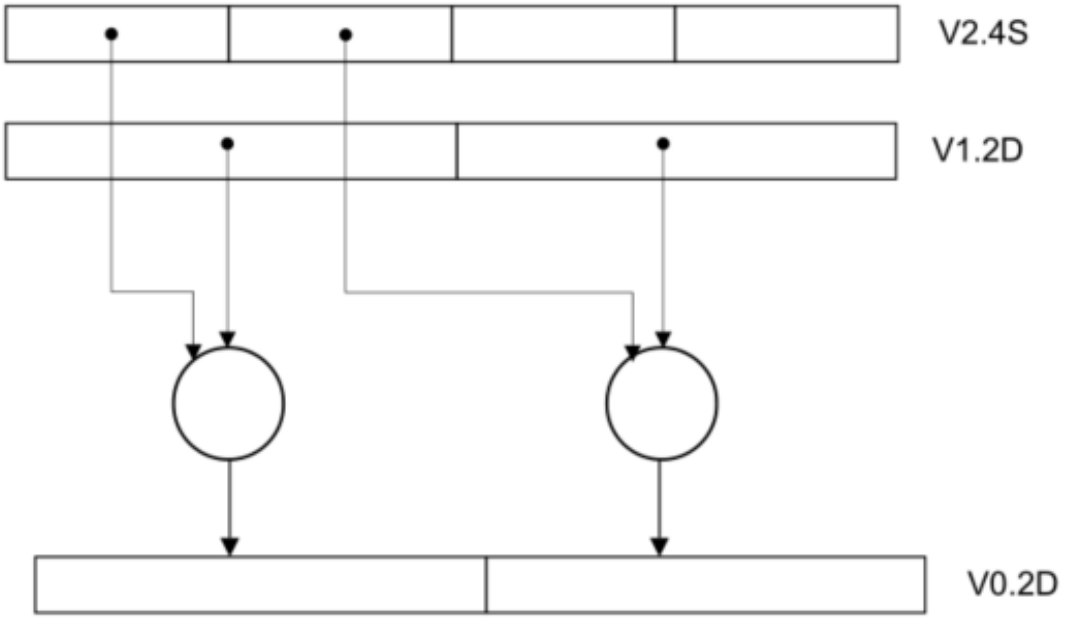
Variante "wide"

- Operam sobre um vetor de 64 bits e e um vetor de 128 bits, produzindo um vetor de 128 bits.
- Os elementos do resultado e do 1º operando têm o dobro do tamanho dos elementos do 2º operando.
- Exemplo: `SADDW V0.4S,V1.4H,V2.4S`



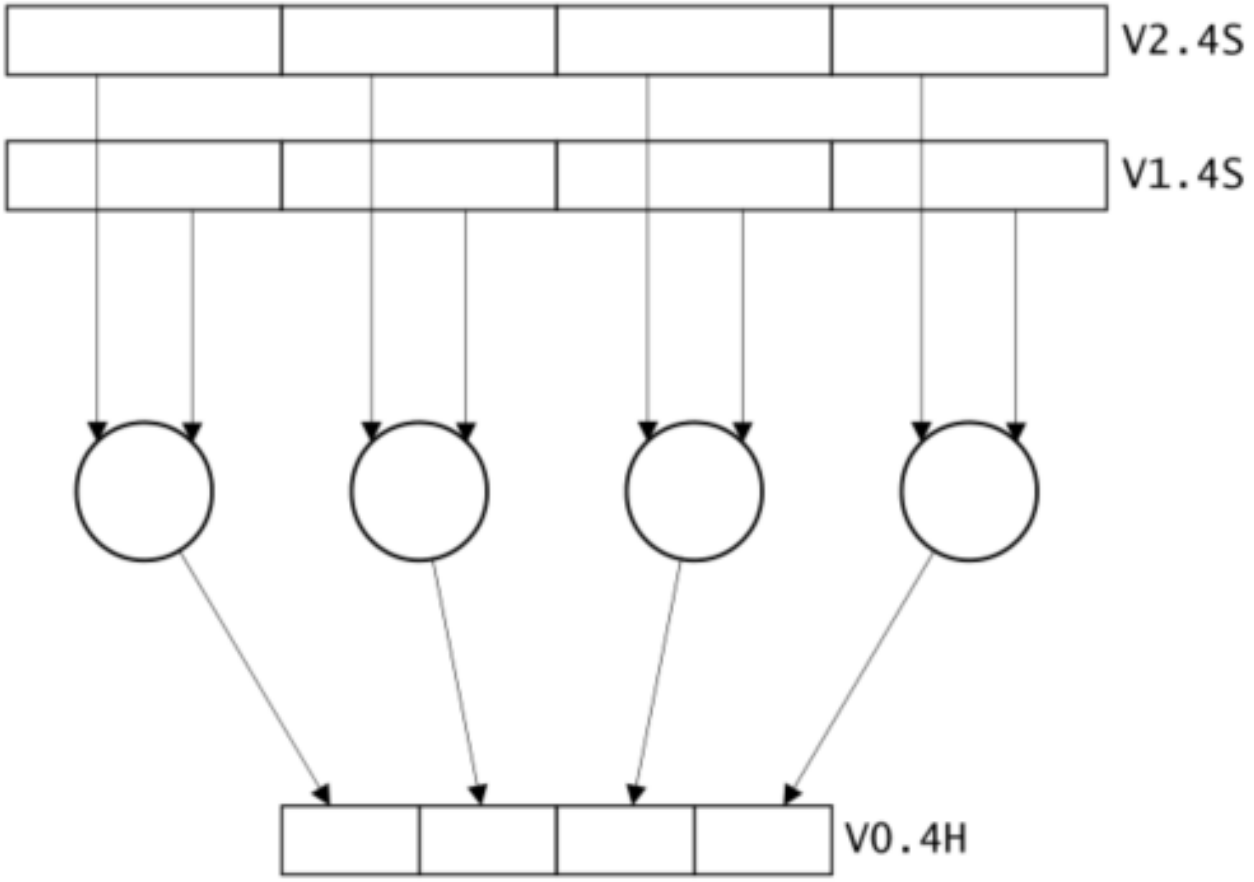
Variante "wide" para pistas superiores

- stas operações obtêm os seus dados das pistas superiores do 2º operando (mais pequenos) e guardam os resultados (expandidos) no operando de destino.
- Exemplo: `SADDW2 V0.2D,V1.2D,V2.4S`



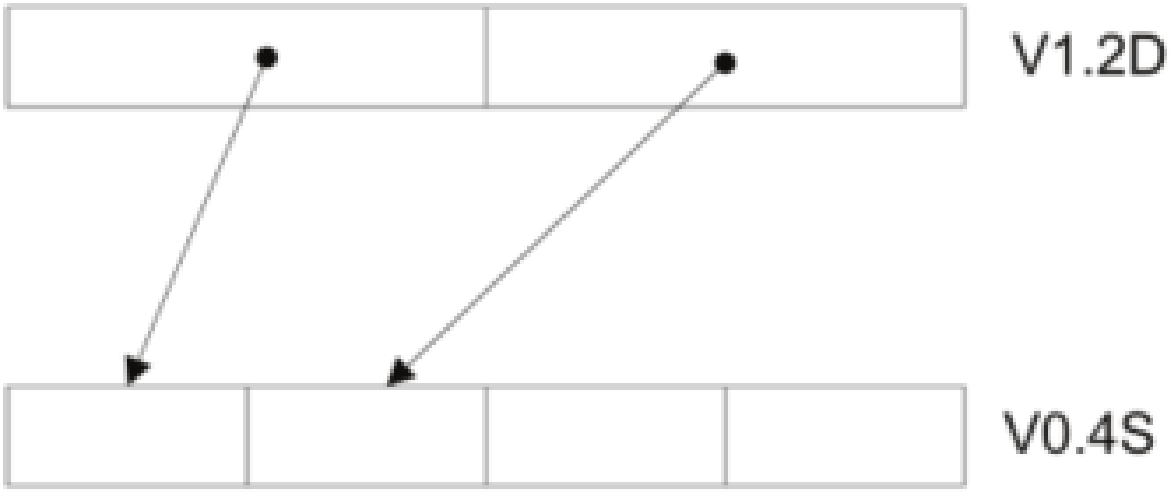
Variante "narrow"

- Operam sobre vetores de 128 bits e produzem um vetor de 64 bits.
- Os elementos do resultados têm metade do tamanho dos operandos.
- Exemplo: `SUBHN V0.4H,V1.4S,V2.4S`



Variante "narrow" para pistas superiores

- Operam sobre vetores de 128 bits e produzem um vetor de 64 bits.
- Os elementos do resultados têm metade do tamanho dos operandos e são colocados nas pistas superiores do registo de destino.
- Exemplo: `XTN2 V0.4S,V1.DS` (extract narrow)



Operações com saturação

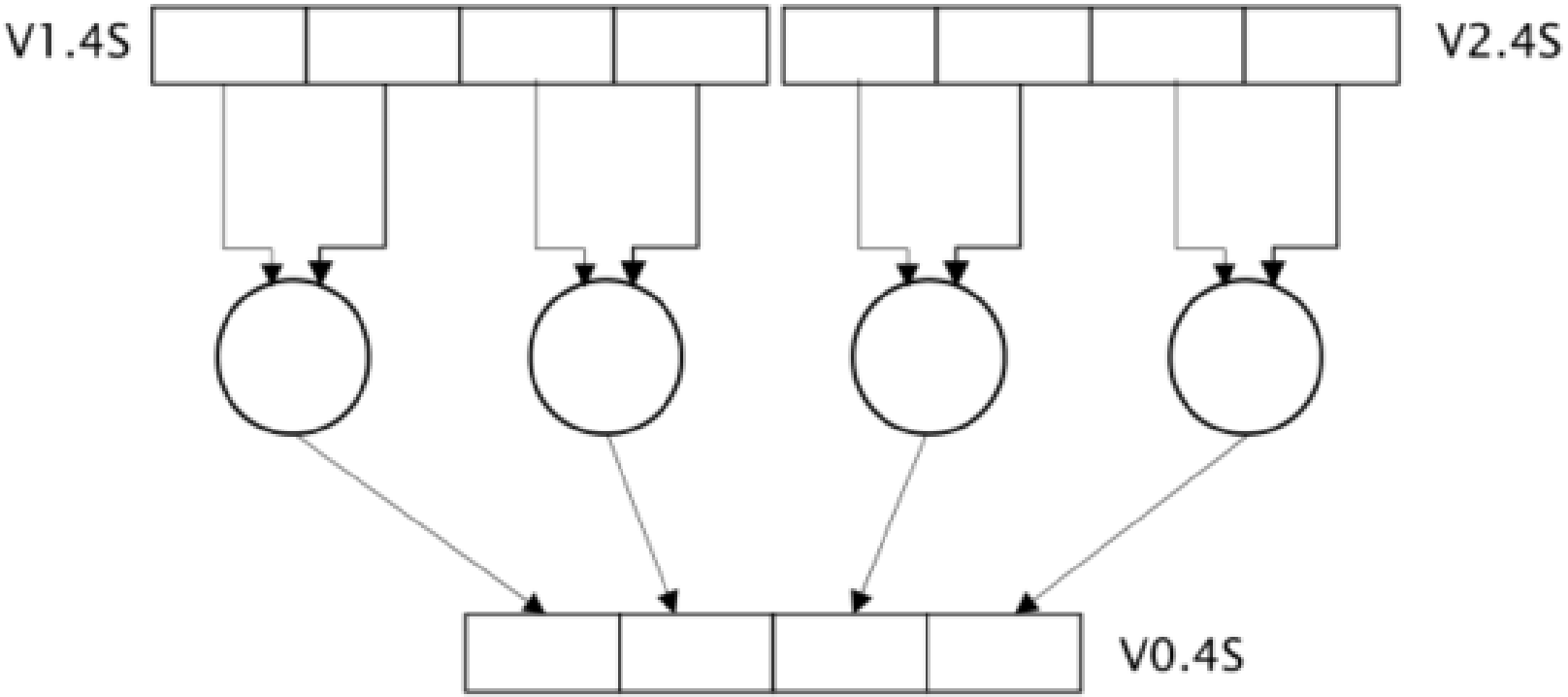
- Nas operações com saturação, o resultado nunca passa do valor máximo ou mínimo da representação usada.
- Existem versões “signed” (prefixo SQ) e “unsigned” (prefixo UQ).

➠ Os limites são (números inteiros):

Data type	**Saturation range of *x***
Signed byte (S8)	$-2^7 <= x < 2^7$
Signed halfword (S16)	$-2^{15} <= x < 2^{15}$
Signed word (S32)	$-2^{31} <= x < 2^{31}$
Signed doubleword (S64)	$-2^{63} <= x < 2^{63}$
Unsigned byte (U8)	$0 <= x < 2^8$
Unsigned halfword (U16)	$0 <= x < 2^{16}$
Unsigned word (U32)	$0 <= x < 2^{32}$
Unsigned doubleword (U64)	$0 <= x < 2^{64}$

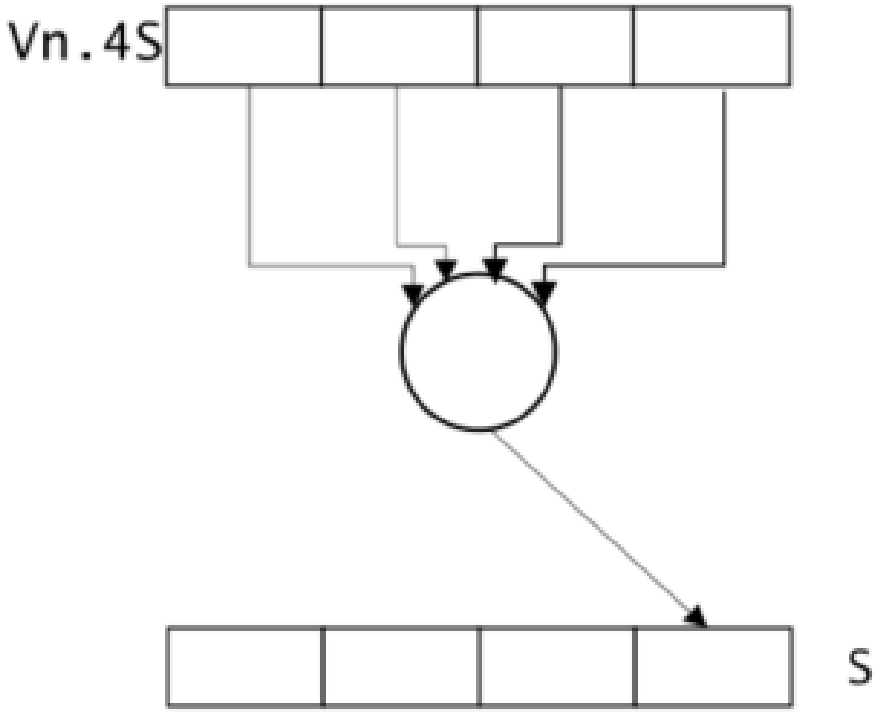
Operações para operações sobre pares de registos

- Operações “horizontais”
- Estas operações usam pares de registos `doubleword` ou `quadword` adjacentes .
- Exemplo: `ADDP V0.4S,V1.4S,V2.4S`



Operações sobre todos os elementos de um vetor

- Operações “horizontais” sobre todos os elementos de um vetor
- Os nomes destas operações usam o prefixo V.
- Exemplo: `VADD S0,V1.4S`



Assuntos

Regras gerais

- Cada registo *SIMD&VF* pode conter:
  1. um valor escalar VF ou inteiro
  2. Um vetor de 64 bits com um ou mais componentes
  3. Um vetor de 128 bits com 2 ou mais componentes

- Quando um registo *SIMD&VF* não é totalmente usado, os dados ocupam a parte menos significativa e qualquer escrita coloca os bits mais significativos a zero.

- Grande parte das instruções de vírgula flutuante têm uma variante que usa vetores.

- Existem instruções aritméticas, de conversão e operações lógicas bit-a-bit para:
  1. vírgula flutuante (precisão simples, dupla e meia-precisão)
  2. Inteiros com e sem sinal

Operações aritméticas sobre vetores inteiros (1)

- Formato geral: `<instr> V<d>.<T>, V<n>.<T>`
- <T> pode ser: 8B, 16B, 4H, 8H, 2S, 4S, 2D (varia com as instruções)
- Instruções: ABS, ADD, MUL, NEG, SMAX, SMIN, SUB, UMAX, UMIN

➠ Dados para exemplo

```
signed char vchar[16]={4, 12, -5, 90, 2, -2, 8, 9,
                       -1,0,16,-16,25,1,0,20},
            rchar[16];
short int vshort[8] = {1, 2, 3, 4, -1, -2, -3, 4},
          rshort[8];
int vinteger[4] = { 12, -21, 8, -4}, rinteger[4];
int vintegerB[4] = { 6, -31, 13, -2};
long int vlongA[2] = { 23, -10}, rlong[2];
long int vlongB[2] = {-3, 2 };
```

Operações aritméticas sobre vetores inteiros (2)

```
int main(void)
{
  func_abs16(vchar, rchar);  print16(rchar);
  func_abs8(vshort, rshort); print8(rshort);
  func_add8(vshort,rshort, rshort); print8(rshort);
  func_mul4(vinteger, vinteger, rinteger); print4(rinteger);
  func_mul4e(vinteger, vinteger, rinteger); print4(rinteger);
  func_smax4(vinteger, vintegerB, rinteger); print4(rinteger);
  func_sub2(vlongA, vlongB, rlong);
  print2(rlong);
  return EXIT_SUCCESS;
}
```

```
func_abs16:
  ldr   Q0, [X0]
  abs   V1.16B, V0.16B
  str   Q1, [X1]
  ret
  //4 12 5 90 2 2 8 9 1 0 16 16 25 1 0 20
```

```
func_abs8:
  ldr   Q3,[X0]
  abs   V4.8H, V3.8H
  str   Q4, [X1]
  ret       // 1 2 3 4 1 2 3 4
```

```
func_add8:
  ldr   Q3,[X0]
  ldr   Q4, [X1]
  add   V4.8H, V4.8H, V3.8H
  str   Q4, [X2]
  ret       // 2 4 6 8 0 0 0 8
```

Operações aritméticas sobre vetores inteiros (3)

- Continuação dos exemplos

```
func_mul4:
    ldr     Q3,[X0]
    ldr     Q4, [X1]
    mul     V4.4S, V4.4S, V3.4S
    str     Q4, [X2]
    ret       // 144 441 64 16

func_mul4e:
    ldr   Q6,[X0]
    ldr   Q7, [X1]
    // por componente
    mul   V6.4S, V6.4S, V7.S[1]
    str   Q6, [X2]
    ret   // -252 441 -168 84
```

```
func_smax4:
    ldr   Q6,[X0]
    ldr   Q7, [X1]
    smax V6.4S, V6.4S, V7.4S
    str   Q6, [X2]
    ret   // 12 -21 13 -2

func_sub2:
    ldr     Q6,[X0]
    ldr     Q7, [X1]
    sub     V6.2D, V6.2D, V7.2D
    str     Q6, [X2]
    ret       // 26 -12
```

Instruções de transferência

- MOV pode ser usado para copiar vetores, mas também para fazer cópias de/para registos gerais (W<n> ou X<n>)
  `MOV V1.16B, V2.16B` ou `MOV V1.8B, V2.8B`
  `MOV V1.B[2],W10` `MOV X10,V2.D[1]` usar com B,H,S,D

- UMOV e SMOV copiam elemento de vetor para registo sem (ou com) extensão de sinal.
  `UMOV W10, V2.H[3]` (16 bits $\rightarrow$ 32 bits, extensão com 0)
  `SMOV X10, V2.S[3]` (32 bits $\rightarrow$ 64 bits, extensão de sinal)

- DUP copia valor de registo de uso geral para vetor, replicando o valor
  `DUP V1.16B,W2` byte menos significativo de W2 replicado 16 vezes

Exemplo

```
extern void func_replicate_byte(void *out, signed char val);
signed char vchar_b[16];

int main(void)
{
    func_replicate_byte(vchar_b, -12);
    print16(vchar_b);
}
```

```
// X0 endereço do vetor de entrada
// W1 (byte menos significativo) valor a replicar
func_replicate_byte:
      dup  V0.16B, W1
      str  Q0, [X0]
      ret
```

produz no monitor:

```
-12 -12 -12 -12 -12 -12 -12 -12 -12 -12 -12 -12 -12 -12 -12 -12
```

Instruções de conversão inteiro $\leftrightarrow$ VF

- UCVTF: converte cada elemento inteiro (sem sinal) de um vetor para o formato VF do mesmo tamanho (com arredondamento)

  `UCVTF H<d>, H<n>` meia precisão

  `UCVTF V3.4S, V2.4S` número de 32 bits para VF precisão simples

- Para números com sinal, usar a instrução SCVTF

- Para conversões no sentido inverso, usar FCVTNU e FCVTNS, respetivamente

Instruções de comparação para inteiros

- Formato geral: `CM<cmp> V<d>.<T>, V<n>.<T>, V<m>.<T>`
- Estas instruções colocam em V<d> o resultado da comparação dos elementos correspondentes de V<n> e V<m>.
  Comparação tem 2 resultados possíveis: tudo 1's (verdade) ou tudo 0's (falso) [número de bits dependente do tipo de dados]
- T pode ser: `8B,16B,4H,8H,2S,4S,2D`.
- Comparação não afeta NZCV.
- Instruções: `CMEQ` (igual),
  `CMGE` (maior ou igual), `CMGT` (maior) [com sinal]
  `CMHI` (maior que), `CMHS` (maior ou igual) [sem sinal]
- Existem variantes para comparação com #0:
  `CM<cmp> V<d>.<T>, V<n>.<T>, #0`
- Para comparação com zero, existem ainda (números com sinal): `CMLE` (menor ou igual) e `CMLT` (menor)
- Existem versões `FCM<cmp>` para dados em VF (com sinal!)

Exemplo

```
extern void anular_ident(void* in1, void *in2);
float vFloat[4]=  {1.0f, 3.51f, 2.67f, 2.764e9f};
float vFloat_b[4]={1.0f, -3.5f, -23.67f, 2.764e9f};

int main(void)
{
   anular_ident(vFloat_b, vFloat);
   print_single(vFloat_b);
}
```

```
// X0 endereço de vetor de 4 núumeros de precisão simples
// X1 endereço de vetor de 4 núumeros de precisão simples
anular_ident:
     ldr     Q0, [X0]
     ldr     Q1, [X1]
     fcmeq   V3.4S, V0.4S, V1.4S
     bic     V0.16B, V0.16B, V3.16B
     str     Q0,[X0]
     ret
```

produz no monitor:

```
   0 -3.5 -23.67 0
```

Operações lógicas

- Formato geral: `<instr> V<d>.<T>, V<n>.<T>, V<m>.<T>`
- Calculam a operação lógica bit-a-bit entre elementos correspondentes e guardam o resultado em V<d>.
- T pode ser: 8B (versão de 64 bits), 16B (versão de 128 bits).
- Instruções possíveis: AND, BIC ($a \cdot \overline{b}$), EOR, NOT, ORN ($a + \overline{b}$), ORR.
- BSL (Bit select): copia (para V<d>) bits de V<n> ou de V<m> consoante o respetivo bit de V<d> é 1 ou 0.
- SHL desloca os elementos de um vetor para a esquerda
  `SHL V0.4H, V1.4H, #10`
- SSHR desloca elementos de um vetor para a direita (extensão de sinal)
- USHR desloca elementos de um vetor para a direita (introduzindo 0's)
  `USHR V0.4H,v3.4S,10` `SSHR V0.4S,V0.4S,18`

Exemplo

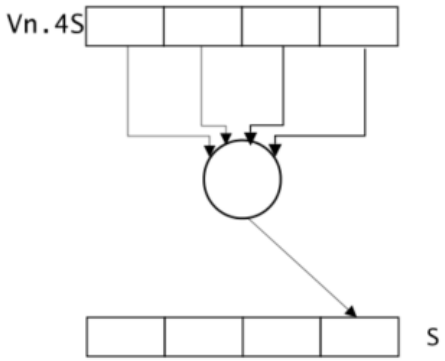
```
extern void manter_se_maiores(void *in, void *out, int val);
int vint_a[4] = {-23, 9, 11, 17 };
int vint_b[4];

int main(void)
{
    manter_se_maiores(vint_a, vint_b, 10);
    print4(vint_b);
}
```

```
// X0 endereço de 1º vetor (4 inteiros)
// X1 endereço de vetor para resultado
// W2 valor limite
manter_se_maiores:
    ldr   Q0, [X0]
    dup   V1.4S, W2
    cmge  V3.4S, V0.4S, V1.4S
    bsl   V3.16B, V0.16B, V1.16B
    str   Q3, [X1]
   ret
```

produz no monitor:

```
10 10 11 17
```

Operações de redução (horizontais)



- Estas instruções usam como operandos os elementos de um vetor produzindo um único resultado que fica na parte menos significativa do destino (B, H ou S)
- Tamanhos possíveis: 8B, 16B, 4H, 8H, 4S

  Exemplo: `ADDV B1, V2.16B`
- Outras instruções deste tipo: SMAXV, UMAXV, SMINV, UMINV

Exemplo

```
extern signed char func_addv16(void *);
signed char vchar[16]={4,12,-5,30,2,-2,8,9,-1,0,16,-16,25,1,0,20};

int main(void)
{
   signed char r;
   print16(vchar);
   r = func_addv16(vchar);
   printf("Soma=%hhd\n", r);
}
```

```
// X0 endereço de vetor de 16 bytes
// W0 resultado (byte menos significativo)
func_addv16:
    ldr     Q2, [X0]
    addv    B1, V2.16B
    smov    W0, V1.B[0]
    ret
```

produz no monitor:

```
 4 12 -5 30 2 -2 8 9 -1 0 16 -16 25 1 0 20
 Soma=103
```