

Sistemas de entrada/saída de dados

João Canas Ferreira

Maio 2016



1 Aspetos gerais

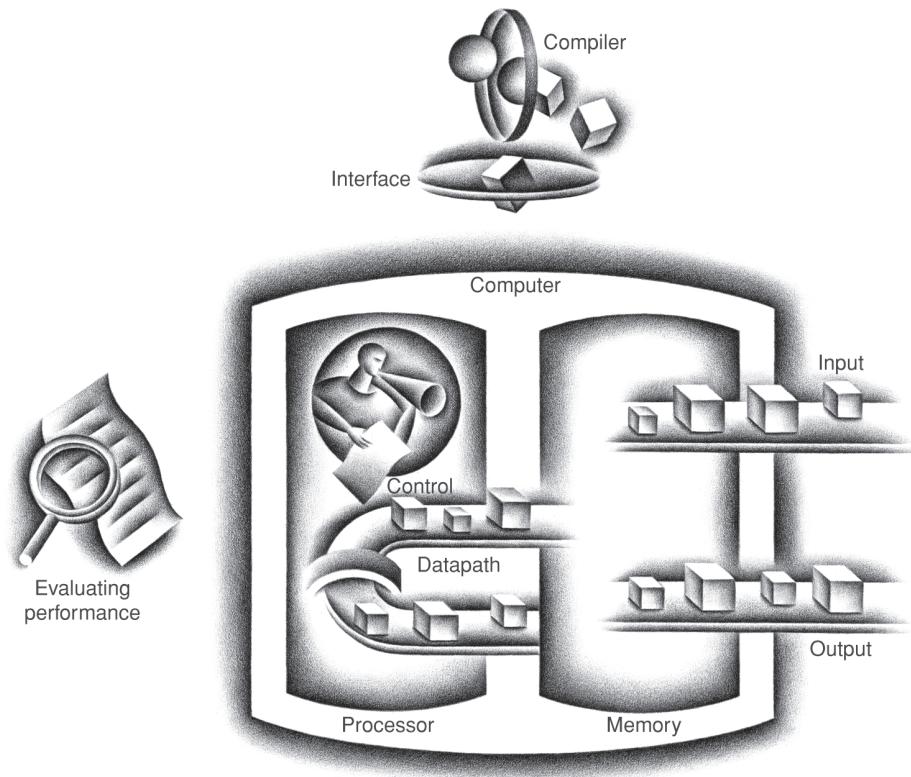
2 Armazenamento de informação

3 Comunicação com periféricos

4 Gestão de periféricos

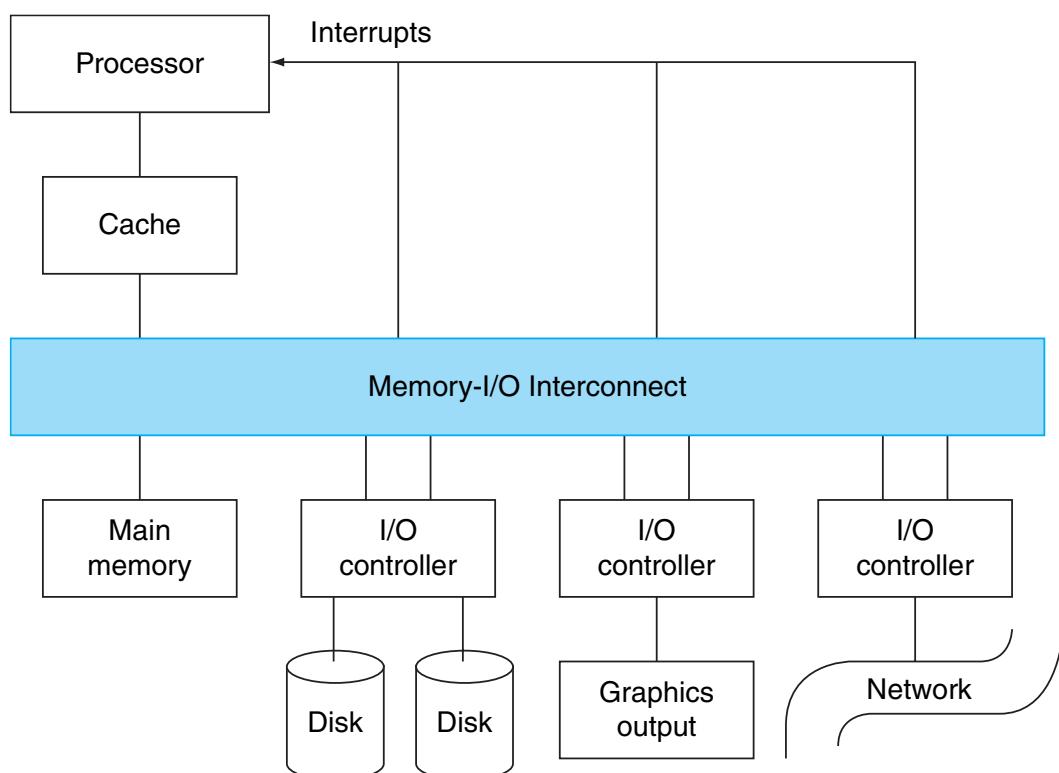
5 Sistemas RAID

Organização de um computador



Fonte: [COD4]

Dispositivos de entrada/saída



Fonte: [COD4]

Diversidade de periféricos

Device	Behavior	Partner	Data rate (Mbit/sec)
Keyboard	Input	Human	0.0001
Mouse	Input	Human	0.0038
Voice input	Input	Human	0.2640
Sound input	Input	Machine	3.0000
Scanner	Input	Human	3.2000
Voice output	Output	Human	0.2640
Sound output	Output	Human	8.0000
Laser printer	Output	Human	3.2000
Graphics display	Output	Human	800.0000–8000.0000
Cable modem	Input or output	Machine	0.1280–6.0000
Network/LAN	Input or output	Machine	100.0000–10000.0000
Network/wireless LAN	Input or output	Machine	11.0000–54.0000
Optical disk	Storage	Machine	80.0000–220.0000
Magnetic tape	Storage	Machine	5.0000–120.0000
Flash memory	Storage	Machine	32.0000–200.0000
Magnetic disk	Storage	Machine	800.0000–3000.0000

Fonte: [COD4]

Características de um sistema E/S

- *Confiabilidade* é importante
 - Em especial, para armazenagem de informação (discos magnéticos)
- Medidas de desempenho
 - Latência (tempo de resposta)
 - Débito (“largura de banda”)
Quantidade de informação processada por unidade de tempo
- PC desktop e sistemas embarcados
 - tempo de resposta e diversidade
- Servidores
 - débito e expansibilidade

Confiabilidade

- *Confiabilidade*: qualidade do sistema que permite confiar justificadamente no serviço oferecido.
- Vários aspectos quantificáveis:
 - Fiabilidade (*reliability*)
Quantificação:
 - ➡ medida do tempo de funcionamento até falhar
 - ➡ probabilidade de não falhar durante o tempo de missão
- Disponibilidade (*availability*)
Quantificação:
 - ➡ tempo (ou %) em que o sistema está operacional
 - ➡ tempo médio de reparação
- Reparabilidade (*Maintainability*)
- Segurança contra acidentes (*safety*)
- Segurança contra acesso não autorizado (*security*)
Fatores:
 - ➡ integridade, confidencialidade, autenticidade

Medidas de confiabilidade

- Fiabilidade: tempo médio até falhar
 - ➡ MTTF: mean time to failure
- Interrupção de serviço: tempo médio de reparação
 - ➡ MTTR: mean time to repair
- Tempo médio entre falhas:
 - ➡ MTBF: mean time between failures
 - ➡ $MTBF = MTTF + MTTR$
- Disponibilidade: $MTTF / (MTTF + MTTR)$
- Para aumentar a disponibilidade:
 - Aumentar MTTF
 - ➡ tolerância a falhas (redundância), previsão de falhas, etc.
 - Reduzir MTTR
 - ➡ melhores ferramentas e processos de diagnóstico e reparação

Exemplo: Falhas de discos na empresa Google

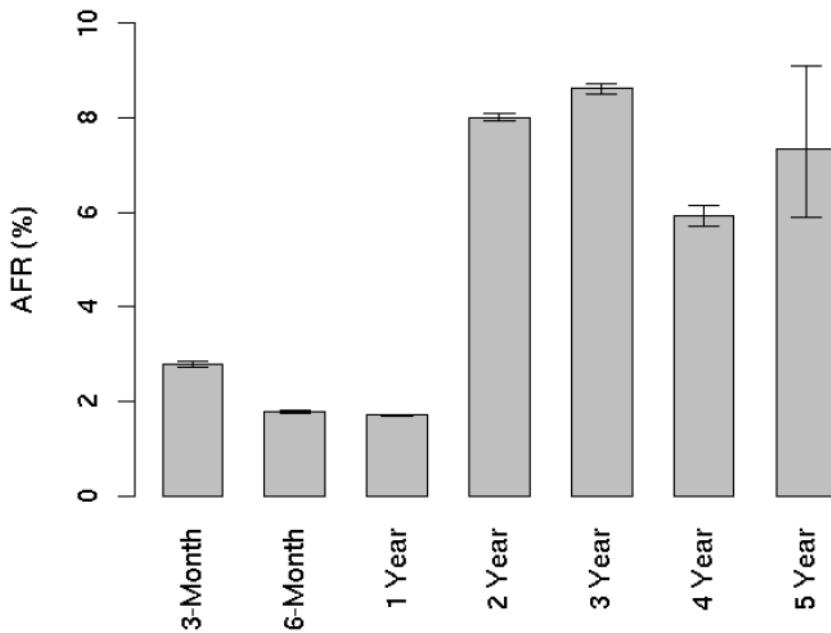


Figure 2: Annualized failure rates broken down by age groups

Fonte: E. Pinheiro et al, "Failure Trends in a Large Disk Drive Population", 5th USENIX Conference on File and Storage Technologies (FAST'07), Fev. 2007

1 Aspetos gerais

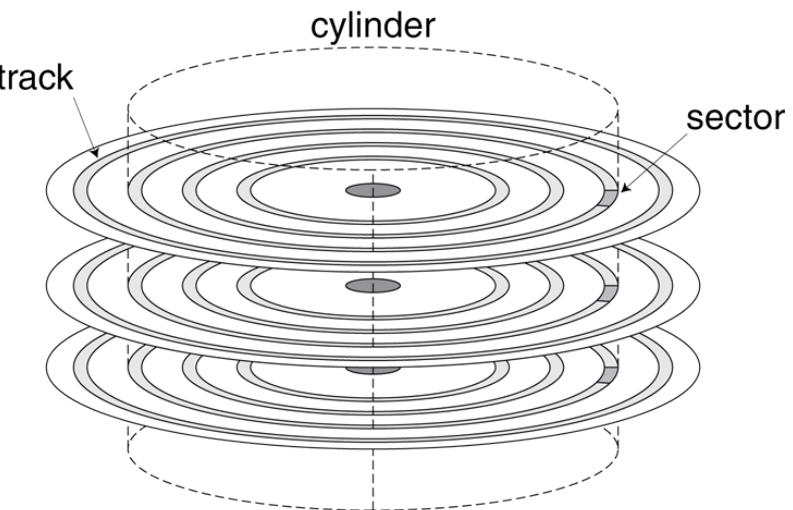
2 Armazenamento de informação

3 Comunicação com periféricos

4 Gestão de periféricos

5 Sistemas RAID

Discos magnéticos

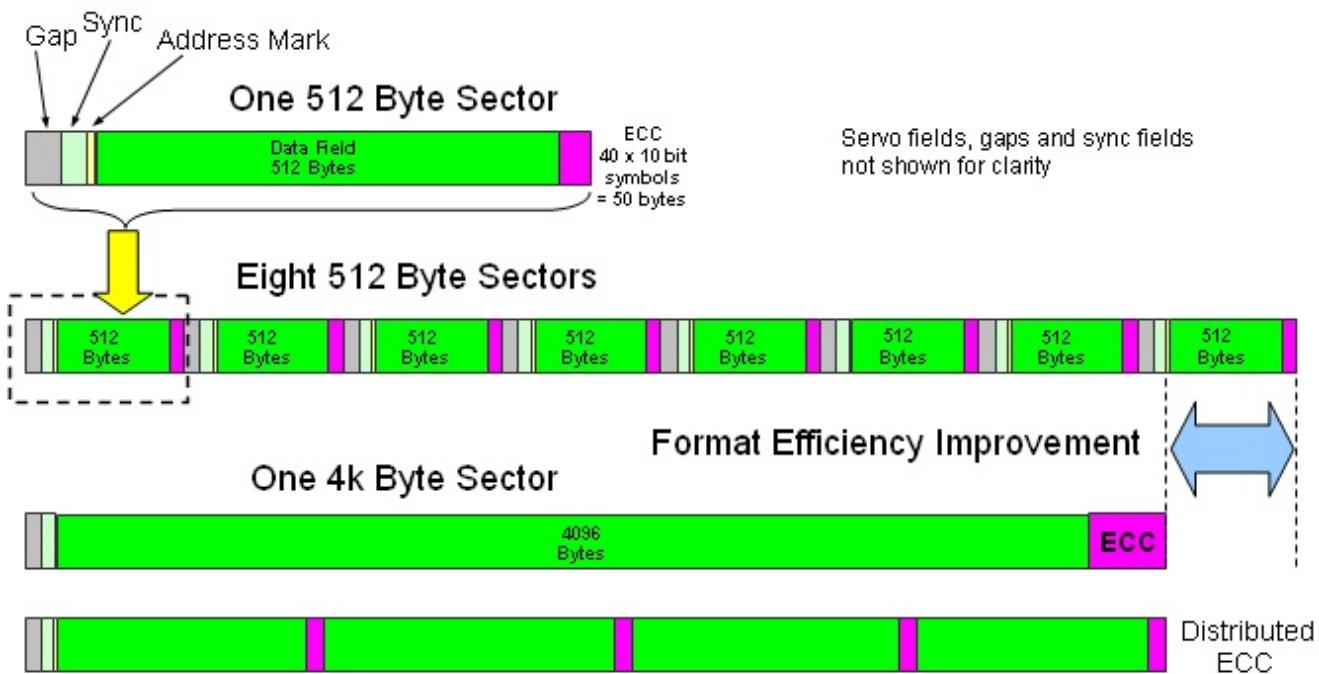


- Pista (*track*): coroa circular numa face de um prato
- Cilindro (*cylinder*): conjunto de pistas que podem ser simultaneamente lidas/escritas
- Setor: segmento de uma pista; unidade básica de armazenamento

Acesso a um setor

- Cada setor físico contém:
 - identificador do setor
 - dados
tradicional: 512 bytes
a partir de 2009: 4096 bytes
 - código corretor de erros (ECC)
permite “esconder” defeitos e problemas de gravação
 - informação de sincronização e lacunas
- Aceder a um setor envolve:
 - esperar se existirem acessos pendentes (fila de espera)
 - movimento da cabeça de leitura: *seek time*
 - latência rotacional: tempo até setor surgir debaixo da cabeça de leitura
 - tempo de transferência dos dados
 - *overhead* do controlador

Formato de setores



[Fonte: Wikipedia]

Exemplo de cálculo de tempo de acesso a disco

Calcular tempo médio de leitura:

- Setor: 512 B
- Velocidade de rotação: 15000 rpm
- Tempo médio de procura: 4 ms
- Taxa de transferência: 100 MB/s
- *Overhead* de controlador: 0,2 ms
- Disco inicialmente inativo

$$t_a = t_{\text{seek}} + t_{\text{rot}} + t_{\text{transf}} + t_{\text{ctrl}}$$

$$t_{\text{rot}} = 0,5 \times \frac{60}{15000} = 2 \text{ ms}$$

$$t_{\text{transf}} = \frac{512}{10^8} = 0.005 \text{ ms}$$

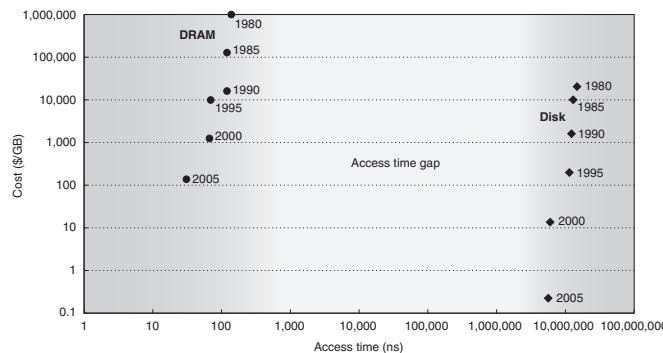
No total: $t_a = 4 + 2 + 0.005 + 0.2 = 6.2 \text{ ms}$

Fatores que afetam o cálculo do desempenho

- Fabricantes indicam o tempo médio de procura t_{seek} baseado na média de todos os acessos
- Na prática, setores sucessivos estão relativamente próximos (proximidade)
 - tempos de procura reais são tipicamente mais baixos que t_{seek}
- Controladores inteligentes determinam a posição física dos setores
 - Apresentam uma interface ao sistema hóspede baseada em setores “lógicos”
- Discos incluem memória *cache*
 - leitura por antecipação (*prefetch*)
 - podem evitar tempo de pesquisa e latência de rotação

Características de discos magnéticos

- ➡ Densidade por unidade de área: $\frac{\text{pistas}}{\text{polegada}} \times \frac{\text{bits}}{\text{polegada}}$
- ➡ Custo e tempo de acesso

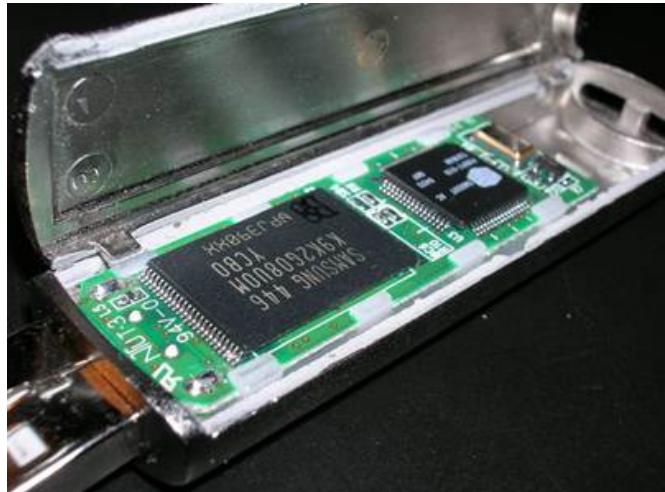


Fonte: [CAQA5]

- ➡ Potência dissipada: $P \propto \text{diâmetro}^{4.6} \times \text{RPM}^{2.8} \times \text{num. de pratos}$

	Capacity (GB)	Price	Platters	RPM	Diameter (inches)	Average seek (ms)	Power (watts)	I/O/sec	Disk BW (MB/sec)	Buffer BW (MB/sec)	Buffer size (MB)	MTTF (hrs)
SATA	2000	\$85	4	5900	3.7	16	12	47	45-95	300	32	0.6M
SAS	600	\$400	4	15,000	2.6	3-4	16	285	122-204	750	16	1.6M

Fonte: [CAQA5]



- Memória *não volátil*

- 100x – 1000x mais rápida que discos magnéticos
- menor consumo, maior robustez
- mais cara em €/GB (entre disco magnético e DRAM)

Tipos de memória Flash

- Flash do tipo NOR

- célula de armazenamento com estrutura semelhante a porta NOR
- acesso direto para leitura/escrita (como SRAM e DRAM)
- memória de instruções em sistemas embarcados

- Flash do tipo NAND

- célula de armazenamento com estrutura semelhante a porta NAND
- mais densa (bits/área), mas acesso é sequencial por blocos
- mais barata

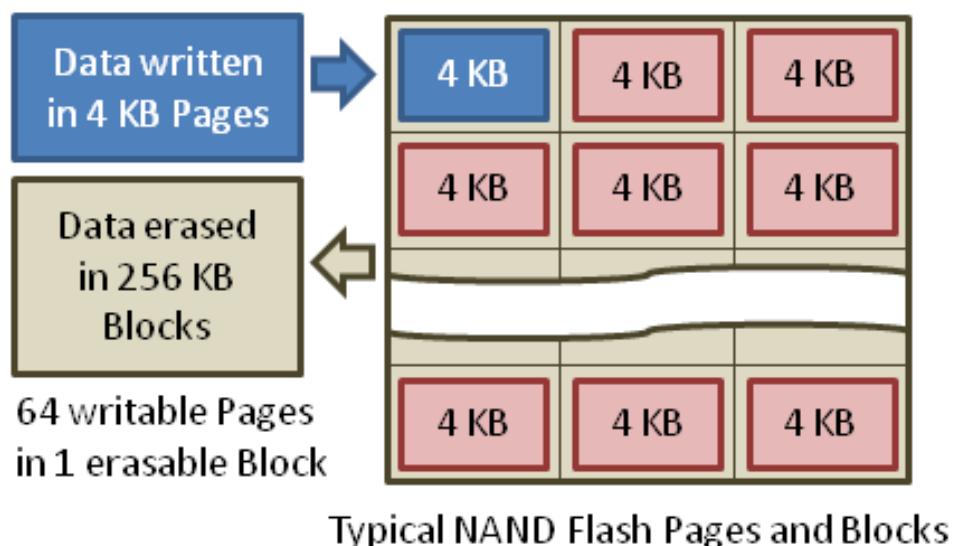
- células perdem capacidade de armazenamento

- gestão de desgaste: mapear dados para zonas menos usadas

Características de memórias Flash

Characteristics	NOR Flash Memory	NAND Flash Memory
Typical use	BIOS memory	USB key
Minimum access size (bytes)	512 bytes	2048 bytes
Read time (microseconds)	0.08	25
Write time (microseconds)	10.00	1500 to erase + 250
Read bandwidth (MBytes/second)	10	40
Write bandwidth (MBytes/second)	0.4	8
Wearout (writes per cell)	100,000	10,000 to 100,000
Best price/GB (2008)	\$65	\$4

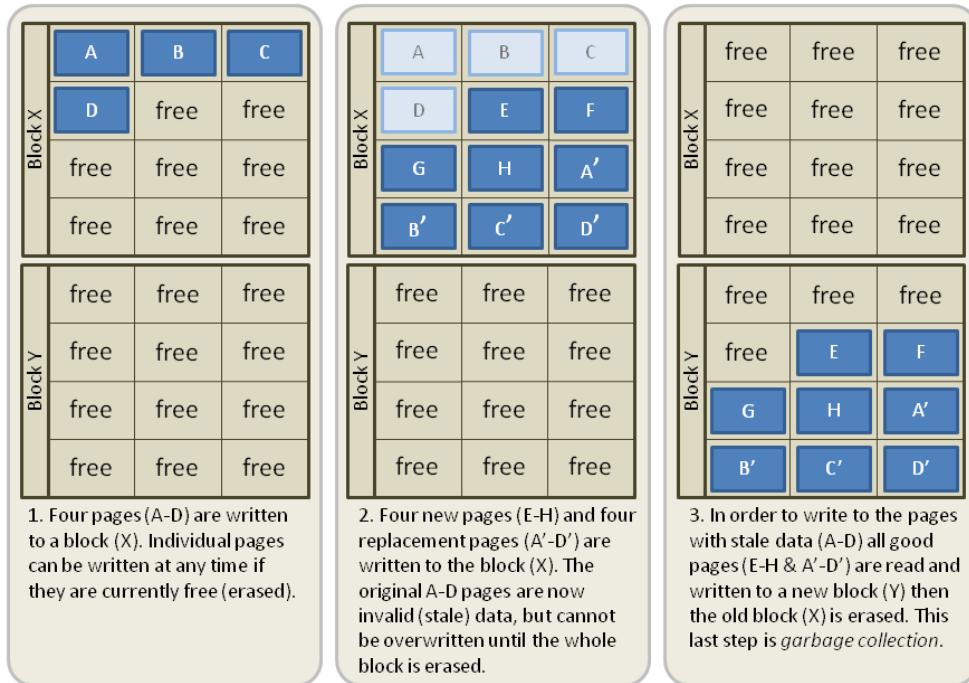
Estrutura interna de NAND Flash: páginas e blocos



[Fonte: Wikipedia]

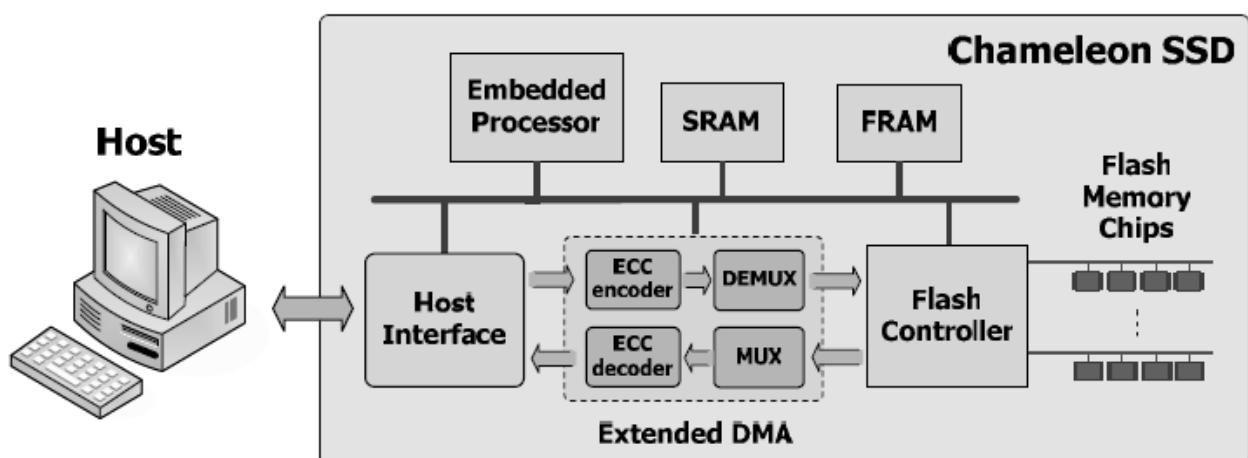
- Escrever por páginas
- Apagar por blocos

Gestão de espaço



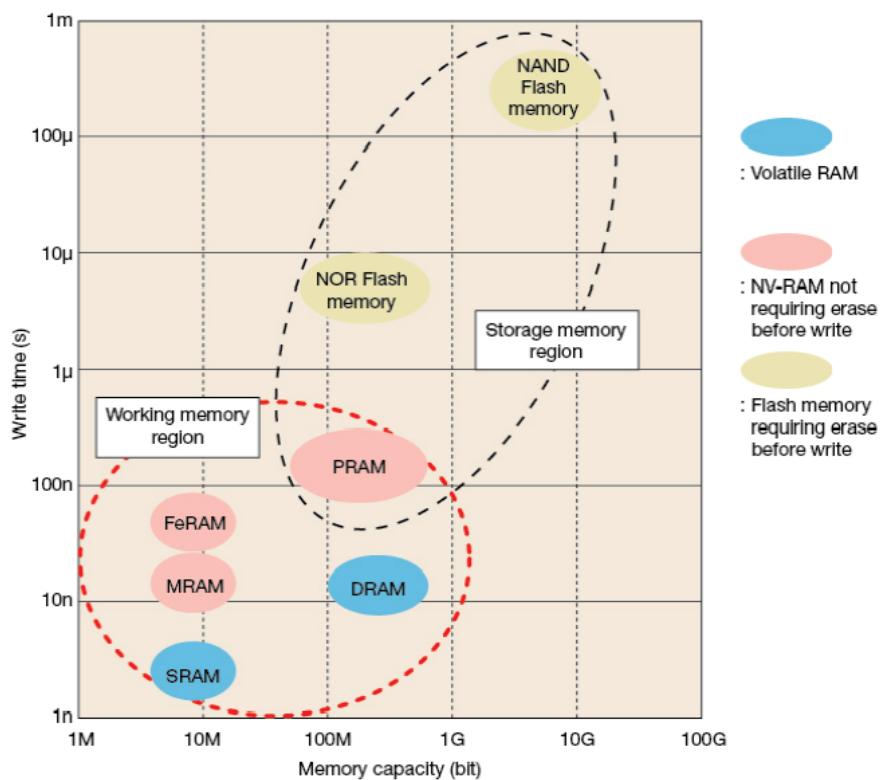
[Fonte: Wikipedia]

Exemplo: Arquitetura de um Solid State Drive



[Fonte: Chameleon: A High Performance Flash/FRAM Hybrid Solid State Disk Architecture, Soon et al., IEEE-CAL Vol.7, N.1, 2008]

Outras tecnologias para memórias não-voláteis



Courtesy: Motoyuki Ooishi

1 Aspetos gerais

2 Armazenamento de informação

3 Comunicação com periféricos

4 Gestão de periféricos

5 Sistemas RAID

Interligar dispositivos

➡ Como interligar CPU, memória e controladores de E/S ?

- Solução mais simples: barramento paralelo
- É um canal de comunicação *partilhado*
- Conjunto de ligações paralelas para dados e sincronização
- Problema: pode limitar o desempenho do sistema (“gargalo”)
- O desempenho de um barramento está limitado por fatores físicos
 - comprimento das ligações
 - número de dispositivos ligados (*slots*)
- Alternativa: ligações série de alta velocidade (ponto-a-ponto)

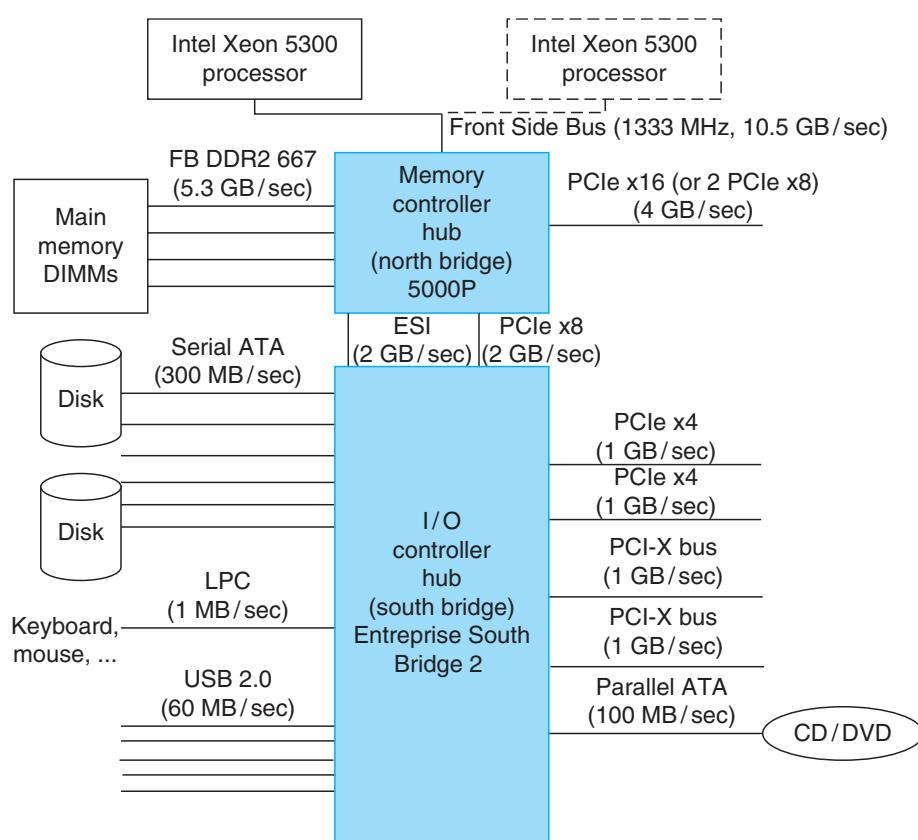
Tipos de barramentos

- **Barramentos processador/memória**
- Curto, alta velocidade
- Características adaptadas à organização da memória
- Exemplo: FSB: *Front Side Bus*
- **Barramento de E/S (periféricos)**
- Mais comprido
- Múltiplas ligações
- Normalizado, para garantir interoperabilidade
- Ligado ao CPU através de uma *ponte*
- Exemplos: PCI, USB, LPC, etc.

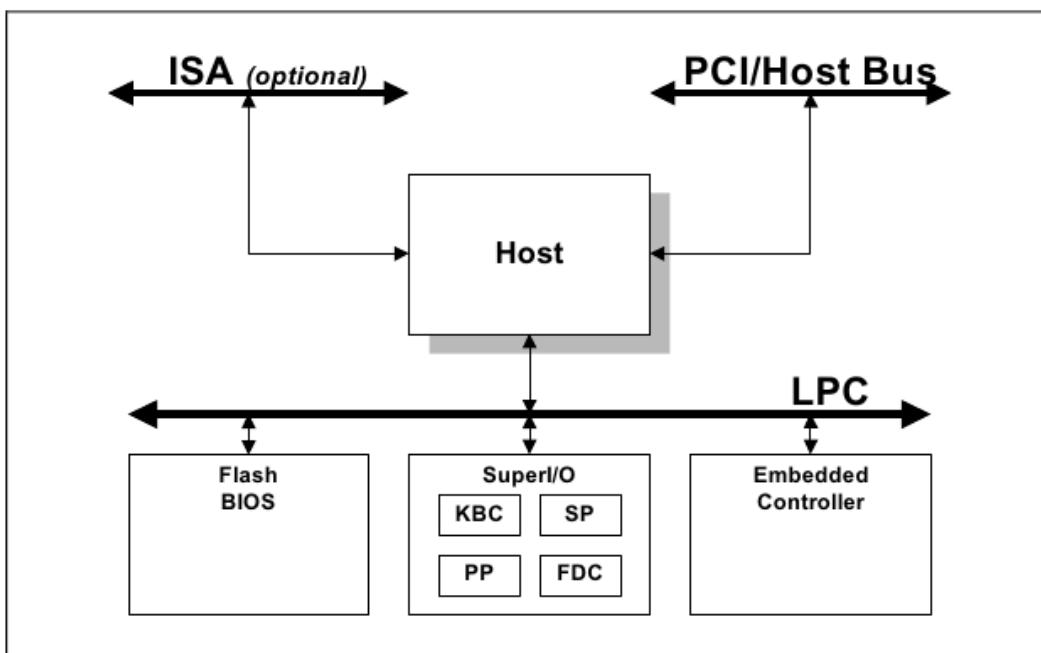
Alguns barramentos atuais

	Firewire	USB 2.0	PCI Express	Serial ATA	Serial Attached SCSI
Intended use	External	External	Internal	Internal	External
Devices per channel	63	127	1	1	4
Data width	4	2	2/lane	4	4
Peak bandwidth	50MB/s or 100MB/s	0.2MB/s, 1.5MB/s, or 60MB/s	250MB/s/lane 1x, 2x, 4x, 8x, 16x, 32x	300MB/s	300MB/s
Hot pluggable	Yes	Yes	Depends	Yes	Yes
Max length	4.5m	5m	0.5m	1m	8m
Standard	IEEE 1394	USB Implementers Forum	PCI-SIG	SATA-IO	INCITS TC T10

Sistema E/S típico (*x86*)



Exemplo: Barramento LPC (Low Pin Count)



KBC: controlador de teclado, SP: porto série(RS232), PP: porto paralelo ,
FDC: controlador de diskette (*floppy disk controller*)

1 Aspetos gerais

2 Armazenamento de informação

3 Comunicação com periféricos

4 Gestão de periféricos

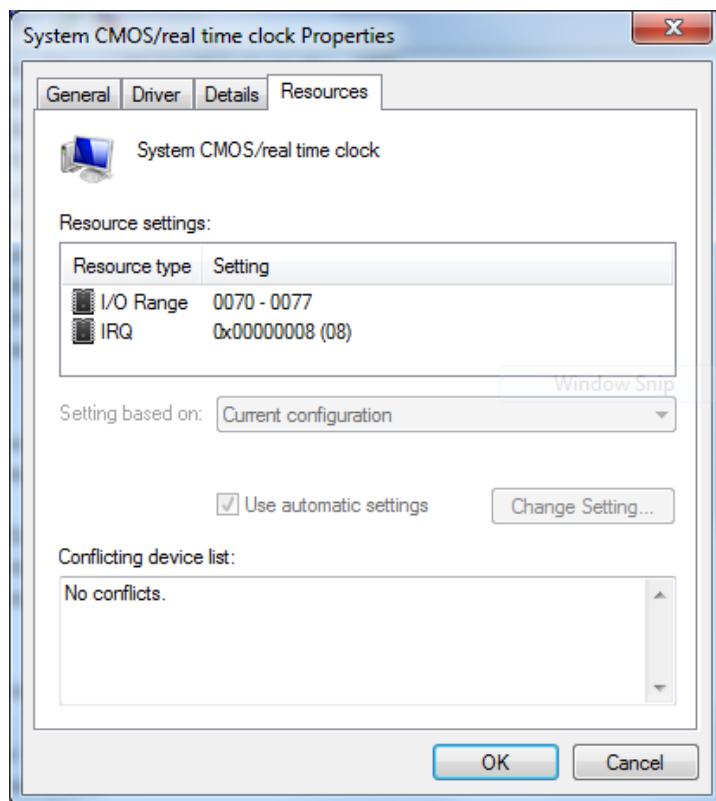
5 Sistemas RAID

- **E/S são geridas pelo sistema operativo**
- Vários programas partilham os recursos E/S
 - Proteção e sequenciamento da utilização
- E/S causam interrupções assíncronas
 - Gerir as rotinas de atendimento de interrupções
- Programação de E/S é “delicada”
 - Sistema operativo disponibiliza abstrações (p.ex. ficheiros)
- **Dispositivos são geridos por controladores de E/S em hardware**
 - Transferências entre memória (ou CPU) e dispositivos
- Controlador tem:
 - Registos de comandos: indicam a tarefa a executar
 - Registos de estado: indicam atividade em execução e situações de erro
 - Registos de dados:
 - escrita (CPU/memória para dispositivo)
 - leitura (do dispositivo para CPU/memória)

Acesso a dispositivos de E/S

- **Mapeamento no espaço de endereçamento de memória**
- Registos de E/S são mapeados em endereços de memória
- Circuito de descodificação de endereços distingue entre memória e dispositivos de E/S
- Sistema operativo usa unidade de gestão de memória virtual para “ocultar” os dispositivos dos programas do utilizador
- Pode ser usado com qualquer CPU
- **Instruções dedicadas de E/S**
- CPU dispõe de instruções separadas para acesso a dispositivos de E/S
- Instruções só podem ser executadas em modo privilegiado
- Exemplo: IA-32 (instruções IN e OUT)

Exemplo: Relógio de tempo real



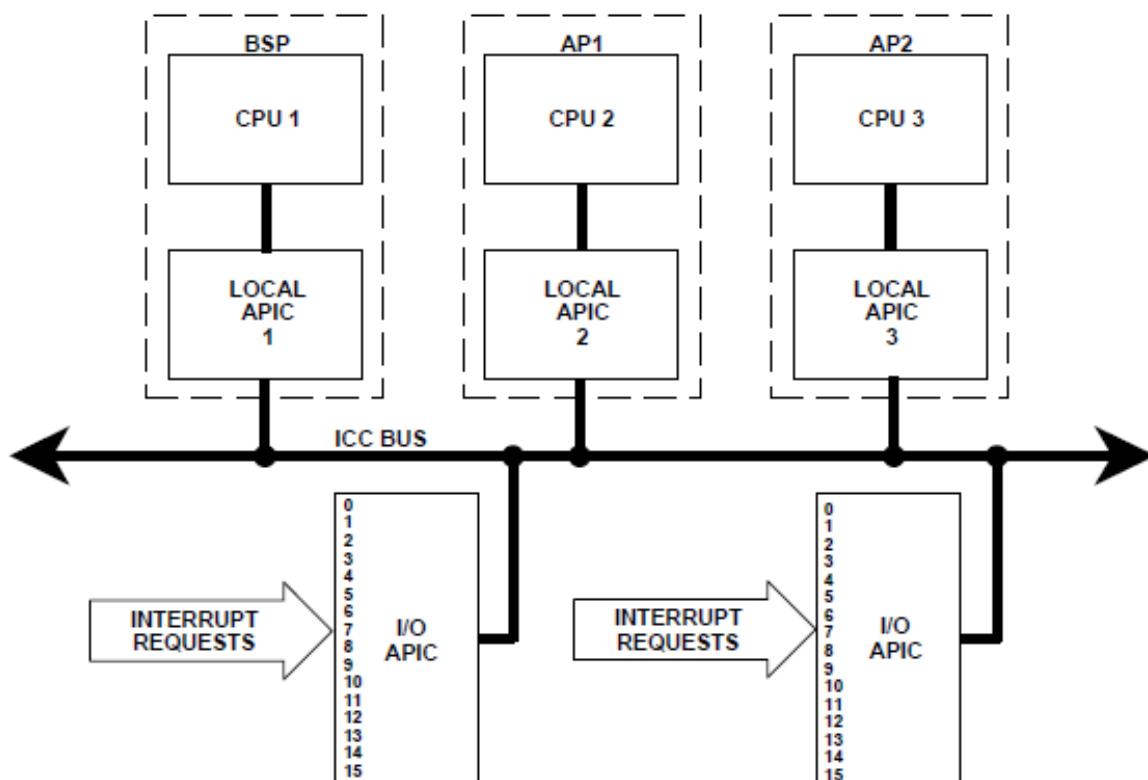
Técnica 1: Varriamento (polling)

- Repetir periodicamente:
 - ① Consultar registos de estado
 - ② Dispositivo pronto? Realizar operação (leitura/escrita)
 - ③ Dispositivo com erro? Recuperar
 - Técnica comum em sistemas pequenos ou de baixo desempenho
 - Comportamento temporal previsível
 - Baixo custo em *hardware*
 - Simples
- ➡ Em sistemas de elevado desempenho: desperdiça recursos de CPU
- ➡ Pode ser apropriado para periféricos lentos:
número de varrimentos necessários é pequeno

Técnica 2: Interrupções

- Dispositivo gera interrupção quando:
 - está pronto
 - ocorre um erro
- Atendimento de interrupção pelo CPU:
 - Interrompe execução do programa
 - Salta para rotina de atendimento
 - Quando a rotina termina, retoma execução “regular”
- CPU suporta diferentes interrupções
- Interrupções são hierarquizadas (prioridades)
 - Dispositivos de atendimento urgente usam interrupções de maior prioridade
 - Atendimento de interrupções de maior prioridade pode interromper atendimento de outras de menor prioridade
- Prioridades são, geralmente, fixas (definidas por hardware)
- Sistema inclui hardware específico para gerir as interrupções
 - Intel 8259 (PCs antigos), circuito integrado ou incluído em “southbridge”
 - APIC (Advanced Programmable Interrupt Controller)

Exemplo: Sistema com APIC



Técnica 3: Acesso directo a memória (DMA)

- Problema: Com as duas técnicas anteriores, a transferência de dados fica a cargo do CPU
 - O CPU é que transfere a informação dos registos de dados do periférico para memória e vice-versa
 - Com dispositivos de alta velocidade (discos, placas de rede), esta abordagem pode afetar gravemente o desempenho das outras tarefas
- Solução: usar um controlador para gerir as transferências
 - sistema operativo define zona de memória (endereço e tamanho) (configuração do controlador de DMA)
 - controlador realiza a transferência, enquanto CPU continua com as suas tarefas
 - quando transferência termina ou encontra um erro, controlador notifica CPU (interrupção)
- Fatores a ter em conta:
 - Interação com memória *cache*: DMA altera memória principal, cópia em *cache* fica desatualizada!
 - Interação com memória virtual: endereços virtuais consecutivos podem não corresponder a endereços físicos consecutivos!

Medidas de desempenho

- Medidas de desempenho de E/S dependem de:
 - Hardware: CPU, memória, controladores, barramentos
 - Software: sistema operativo, sistema de gestão de base de dados, aplicação
 - Carga: taxas e padrões de pedidos
- Projeto pode beneficiar tempo de resposta ou débito
- Frequentemente: medir débito com restrições do tempo de resposta (o tempo de resposta não pode exceder um limiar definido)
- Vários tipos de *benchmarks*
 - Bases de dados: Transaction Processing Council
<http://www.tpc.org>
 - Sistema de ficheiros: SPEC File System (SFS)
<http://www.spec.org/sfs2008>
 - Servidores Web: SPEC Web
<http://www.spec.org/web2009>

➡ Amdahl ataca de novo...

- Desempenho E/S pode comprometer ganhos obtidos por aumento do paralelismo.
- Exemplo:
 - Tarefa demora 90 s (de tempo de CPU) mais 10 s de tempo de E/S
 - Se o número de CPUs duplicar cada dois anos, como evolui o tempo total, admitindo que o sistema E/S não é alterado?

Year	CPU time	I/O time	Elapsed time	% I/O time
now	90s	10s	100s	10%
+2	45s	10s	55s	18%
+4	23s	10s	33s	31%
+6	11s	10s	21s	47%

Fonte: [COD4]

1 Aspetos gerais

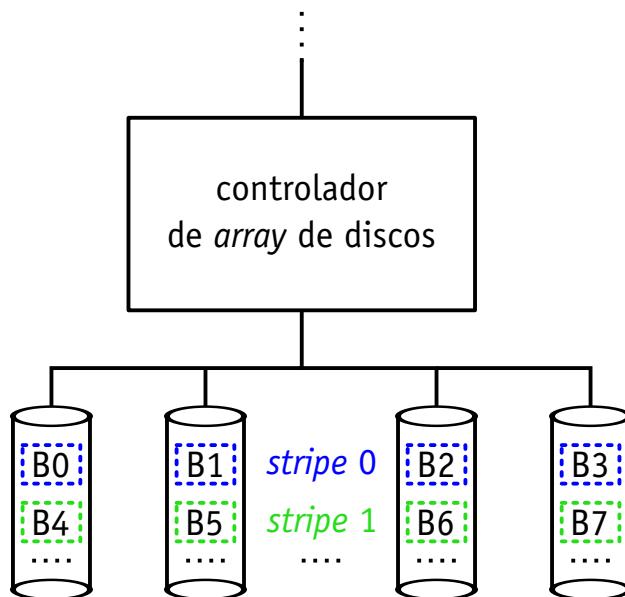
2 Armazenamento de informação

3 Comunicação com periféricos

4 Gestão de periféricos

5 Sistemas RAID

Conjuntos de discos



- Discos físicos em paralelo formam um disco virtual
- Blocos (B_0, \dots) de um ficheiro são "espalhados" pelos discos físicos
- 1 bloco = 1 ou mais setores (número fixo)
- Blocos correspondentes de cada disco formam uma "banda" (*stripe*)

Características de conjuntos de discos

➡ Vantagens:

- Melhor desempenho
 - Transferências "grandes": (array com bandas)
Acesso a D blocos de uma banda demora tanto como acesso a um só disco
 - Transferências "pequenas":
Podem efetuar-se D acessos independentes
- Menor custo
 - Discos individuais mais baratos (grande volume de vendas)
 - Controladores sofisticados em circuito integrado
Existem controladores capazes de lidar com 1000 discos individuais

➡ Desvantagem: Menor fiabilidade

- Basta falhar um disco físico para levar à falha do conjunto
- Para um conjunto de D discos, cada um com o mesmo MTTF:

$$MTTF_{array} = \frac{MTTF}{D}$$

➡ Solução: Introduzir redundância para recuperar de avaria de um disco

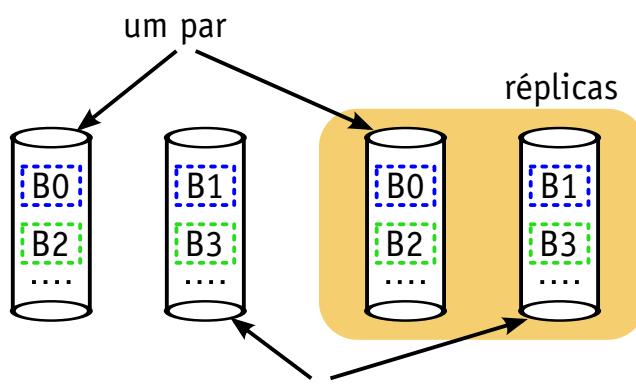
- RAID = *Redundant Array of Inexpensive Disks*
- Paralelismo aumenta a fiabilidade
É mais económico (para o mesmo nível de fiabilidade) ter redundância que aumentar a fiabilidade de um único disco
 - Aproveita economia de escala: existe uma mercado muito grande para discos baratos
- Aumenta MTBF, especialmente se for possível trocar discos sem parar o sistema (*hot swap*)
- Paralelismo aumenta o desempenho
 - Um ficheiro pode ser espalhado por vários discos
 - Leituras “paralelas” permitem obter vários blocos simultaneamente
- Desvantagem: mais discos que o estritamente necessário (aumenta do consumo de energia)
 - Exceção: RAID nível 0 corresponde a uma organização sem redundância

Impacto da introdução de redundância

- D : nº total de discos de dados
- G : nº de discos de dados por grupo
- C : nº de discos com informação de verificação por grupo
- $n_G = D/G$: nº de grupos de proteção
- $N = n_G \times (G + C)$
- falhas independentes; taxa de falhas $t_f = 1/\text{MTTF} = \text{constante}$
- MTTR: o tempo médio de reparação (substituição de um disco)

➡ Array falha se ocorrer pelo menos uma *outra* falha num grupo enquanto a *primeira falha é reparada*.

$$\text{MTTF}_{\text{RAID}} = \frac{\text{MTTF}_{\text{disco}}^2}{(D + C \times n_G) \times (G + C - 1) \times \text{MTTR}}$$



➡ RAID 1: espelho (duplicação concorrente de cada disco)

- Discos: $G = 1$, $C = 1$ (cada disco de dados é duplicado)
- Escrita simultânea no disco de dados e no “disco-espelho”
- Falha de disco: ler do espelho
- Pode ler blocos diferentes de cada disco
- Aproveitamento da capacidade dos discos de 50 %

Cálculo de paridade

- Paridade de D bits (o símbolo \oplus representa a operação “ou-exclusivo”):

$$\text{Paridade}(b_1, b_2, \dots, b_D) = b_1 \oplus b_2 \oplus \dots \oplus b_D = p$$

- Se número de $b_i = 1$ é par \rightarrow paridade = 0; senão paridade = 1
- Usar paridade para determinar um dos b_i em falta:

- ① Calcular a paridade q dos b_i disponíveis
- ② Determinar o valor em falta calculando $q \oplus p$

- (Exemplo) Para determinar b_1 calcula-se sucessivamente:

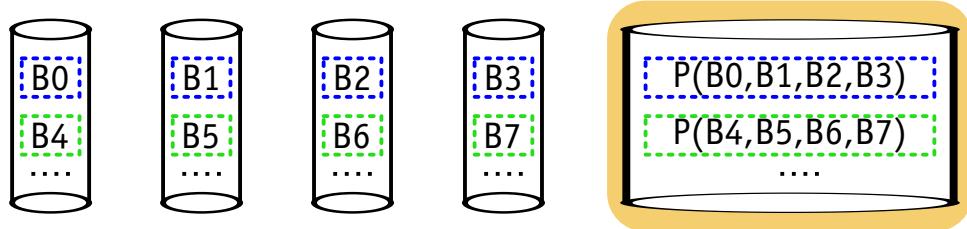
- ① $q = \text{Paridade}(b_2, \dots, b_D)$
- ② $b_1 = p \oplus q$

- Nota: $a \oplus (b \oplus c) = (a \oplus b) \oplus c = a \oplus b \oplus c$
 $a \oplus b = b \oplus a$

- Para elementos com vários bits, a paridade é calculada fazendo o “ou-exclusivo” dos bits em posições correspondentes:

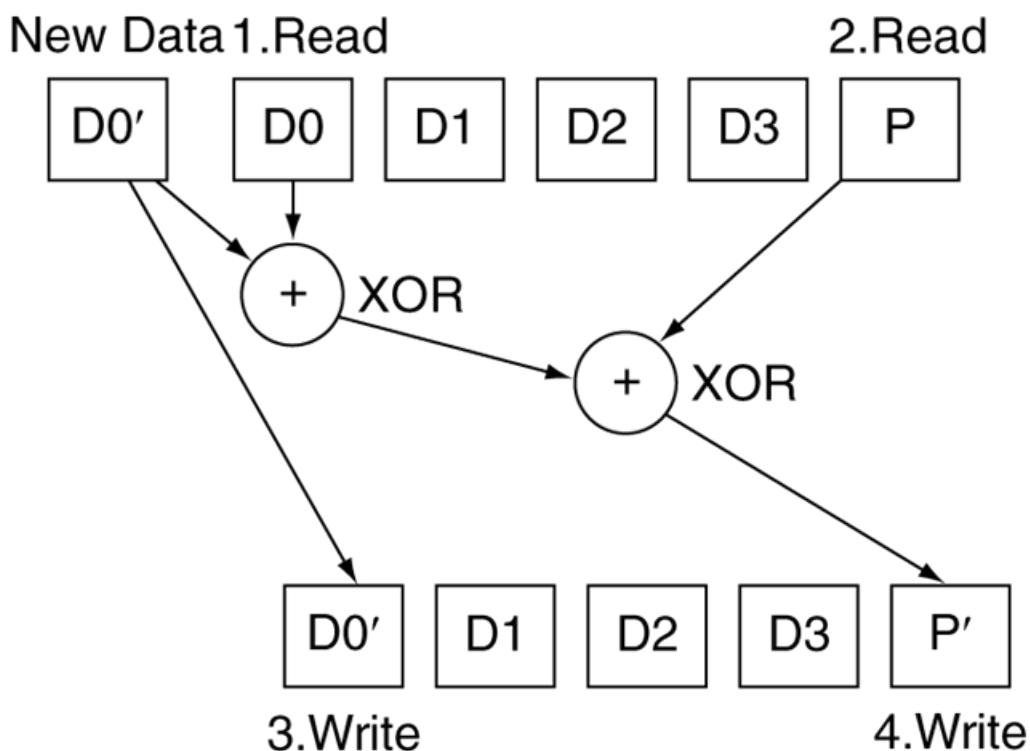
Se $X = [x_1, x_2, \dots, x_n]$ e $Y = [y_1, y_2, \dots, y_n]$, então
 $\text{Paridade}(X, Y) = [x_1 \oplus y_1, x_2 \oplus y_2, \dots, x_n \oplus y_n]$

RAID 4: Block-Interleaved Parity



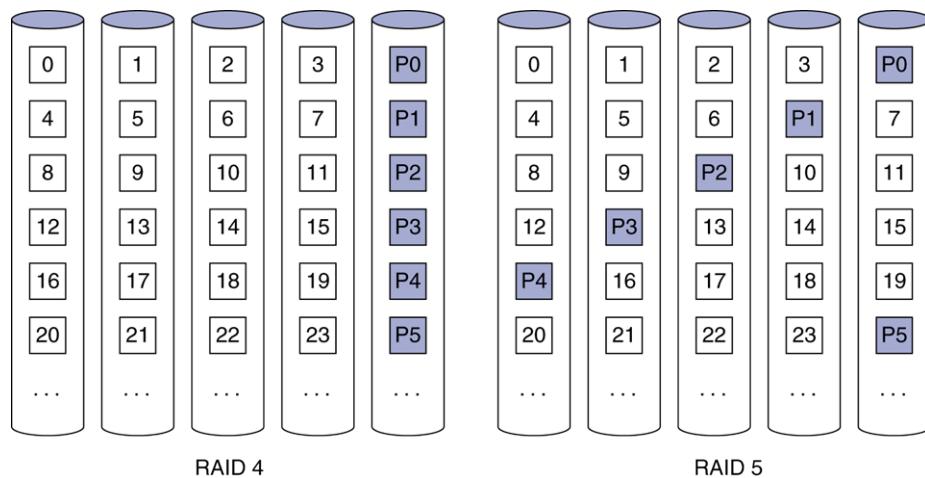
- RAID 4 tem C=1 (1 disco de paridade por grupo)
- Dados espalhados pelos discos ao nível do bloco
- Disco de verificação armazena paridade de blocos da mesma banda
- Leitura: acesso apenas ao disco que contém o bloco
- Escrita: alterar *um* disco de dados e atualizar o disco de paridade (ciclo leitura-modificação-escrita).
- Pouco usado: o acesso ao disco de paridade limita o desempenho na escrita.

RAID 4: leitura-alteração-escrita



RAID 5: Paridade distribuída por vários discos

- Dados espalhados pelos discos ao nível do bloco
- Semelhante a RAID 4, mas com blocos de paridade distribuídos por todos os discos do grupo
- Evita que o disco de paridade restrinja o desempenho do sistema (como acontece com RAID 4)
- Muito usado.



Fonte: [COD4]

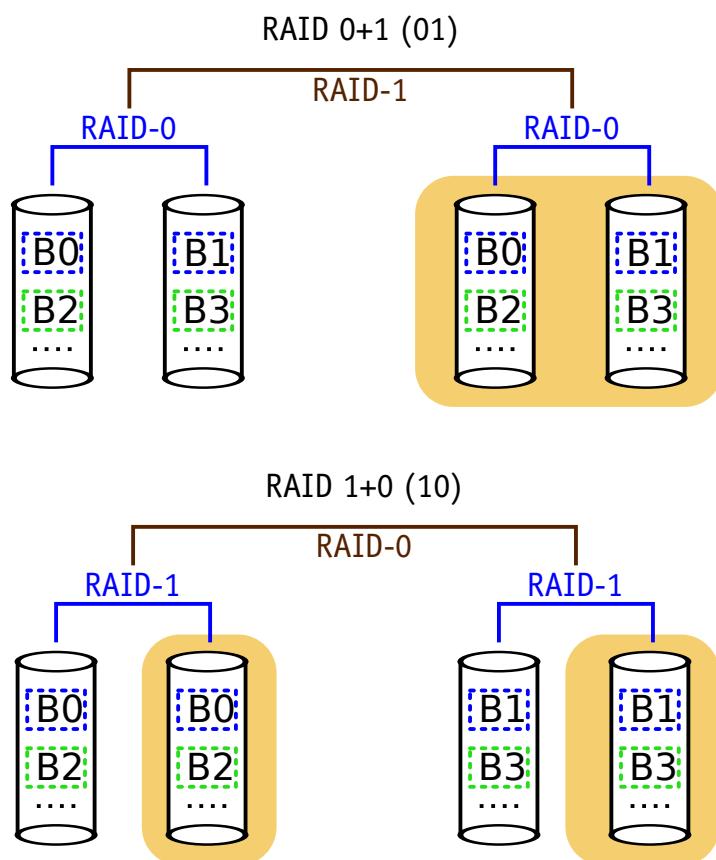
Outros tipos de RAID

➡ RAID 6: Redundância adicional

- Como RAID 5, mas com dois grupos de informação de redundância ($C=2$)
 - Paridade calculada para grupos formados de duas maneiras diferentes.
 - Um dos métodos é igual ao de RAID-5.
- Pode recuperar de situação com dois discos avariados por grupo
- Maior tolerância a falhas à custa de mais redundância

➡ RAID composto

- Composição de duas técnicas de RAID
- RAID 10 (1+0): Aplicar RAID 0 a conjuntos RAID 1 (em vez de discos individuais)
- RAID 01 (0+1): Aplicar RAID 1 a conjuntos RAID 0
- Mais variantes: RAID 50, RAID 60



RAID: conclusão

- ➡ Solução económica para necessidades de armazenamento significativas
É mais barato usar vários discos de fiabilidade normal que um disco de fiabilidade muito elevada.
- ➡ RAID aumenta disponibilidade (exceto RAID 0)
- ➡ Também pode aumentar o desempenho
É preciso ter em atenção o domínio de aplicação, porque o desempenho depende das características dos acessos
- ➡ Elevada disponibilidade requer trocas em funcionamento para reduzir o tempo médio de reparação (MTTR)
- ➡ Atenção: Assume-se que falhas de discos são independentes!
RAID não resolve situações em que vários discos são afetados simultaneamente pelo mesmo evento (p.ex. falha de fonte de alimentação).

Referências

- COD4** D. A. Patterson & J. L. Hennessey, Computer Organization and Design, 4 ed.
- CA5** D. A. Patterson & J. L. Hennessey, Computer Architecture: A Quantitative Approach, 5 ed.

Os tópicos tratados nesta apresentação são abordados nas seguintes secções de [COD4]:

- 6.1–6.7

Introdução à arquitetura AArch64

João Canas Ferreira

Março 2019



Assuntos

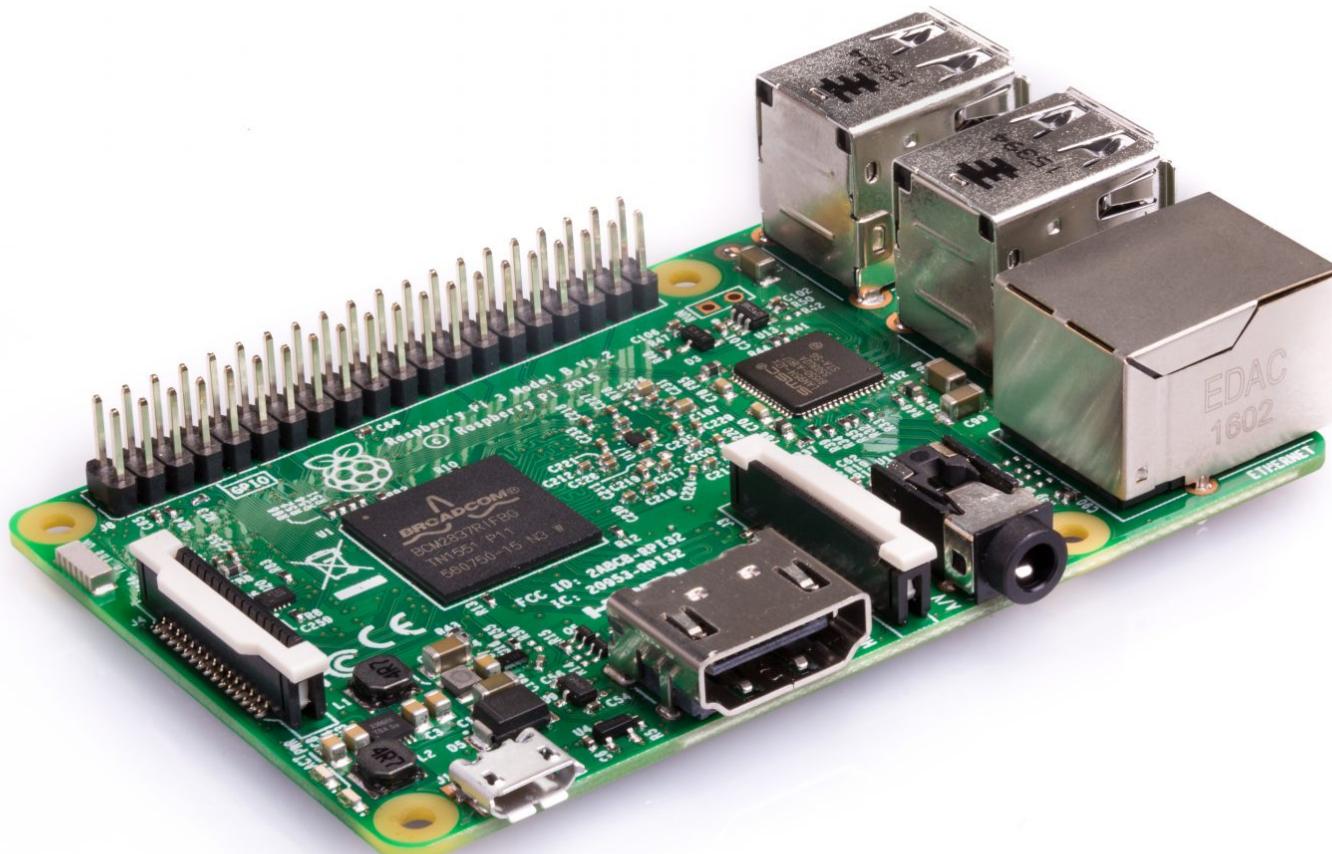
1 Microprocessadores ARM

2 Instruções básicas

As arquiteturas ARM

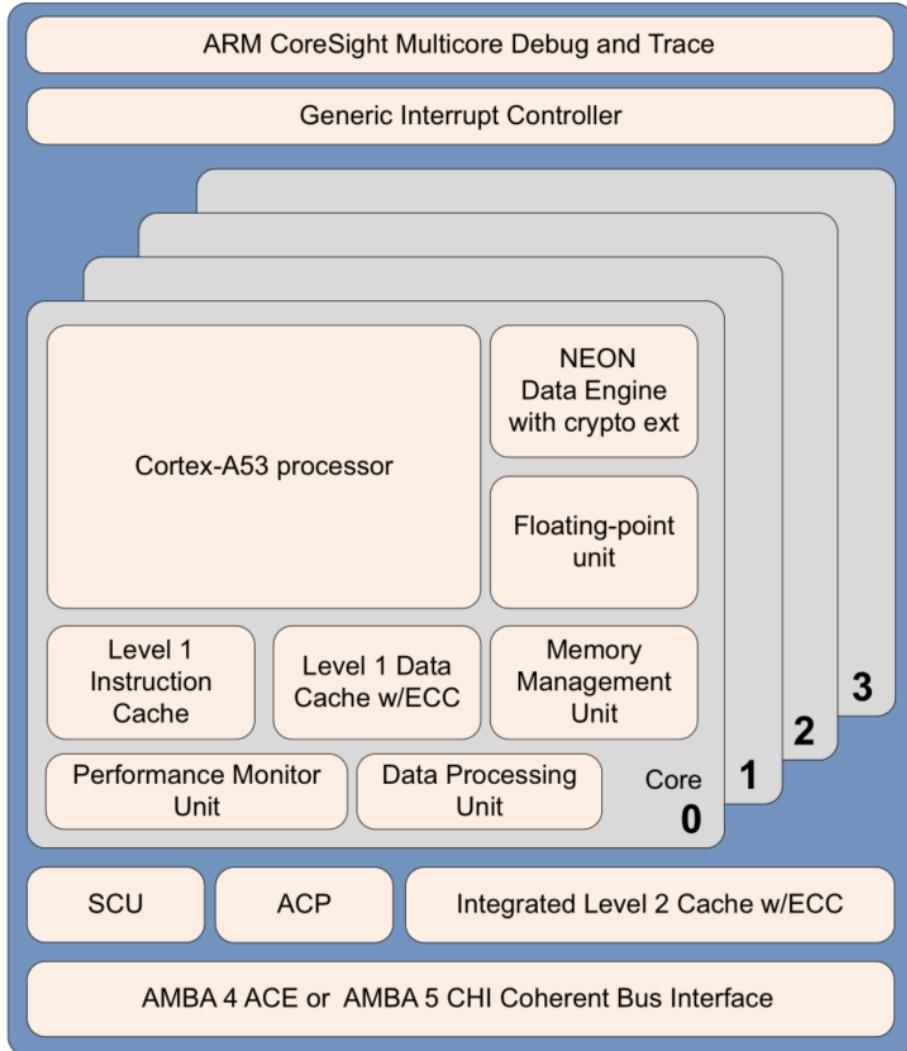
- A designação ARM cobre uma família de arquiteturas RISC.
- Arquiteturas predominantes em *smartphones*, *tablets* e dispositivos de IoT (Internet of Things)
- A companhia ARM licencia a sua propriedade intelectual a outras companhias (Samsung, NXP, etc.).
O Raspberry PI 3 usa um circuito BCM2837 da companhia Broadcomm, que inclui um CPU Cortex-A53 com 4 núcleos.
- Arquitetura ARMv8 tem dois modos de execução:
 - AArch64** Execução de aplicações de 64 bits
 - AArch32** Execução de aplicações de 32 bits, compatível com ARMv7-A.
- Vamos tratar a arquitetura do conjunto de instruções **AArch64**.

Raspberry PI 3



Dimensões: 85 mm×56 mm

Cortex-A53



- 4 núcleos
- unidade de vírgula flutuante
- instruções SIMD (NEON)
- controlador de interrupções
- memória *cache* L1 (por núcleo) e L2 (comum)
- ECC: código corretor de erros
- SCU, ACP: módulos de apoio (*cache*)
- AMBA: barramento interno

Perfis (variantes)

- Existem três variantes (perfis) da arquitetura ARM para mercados/aplicações diferentes.

A — Application Suporta memória virtual via uma MMU (*Memory Management Unit*); usada com sistemas operativos como Linux, iOS, Android.

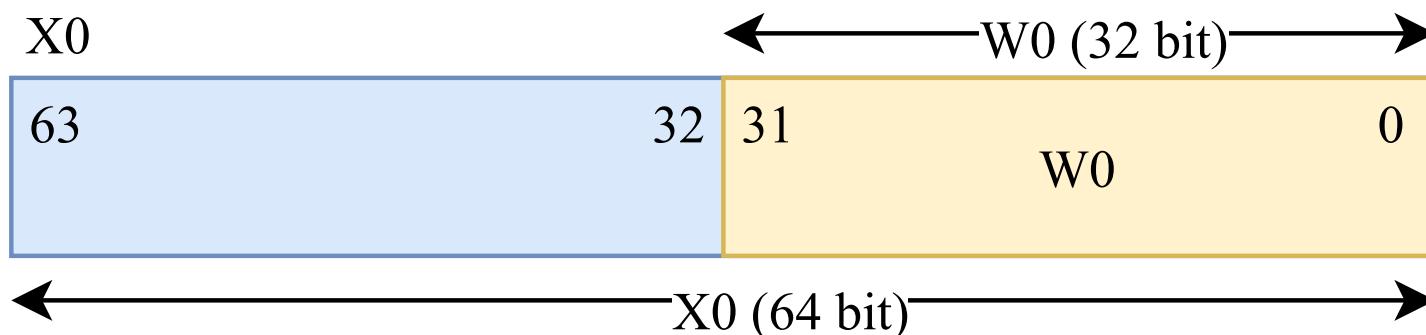
R — Real-time Para aplicações de tempo real (aplicações médicas, automóveis, aviação, robótica); baixa latência e elevado nível de segurança.

M — Microcontroller Proteção de memória; usado para gestão de energia, entradas/saídas, ecrãs táteis, controladores de sensores.

- Nesta UC focamos o perfil A, que é usado na maioria dos SBC (*Single-Board Computer*).
- ARM está a entrar no mercado de HPC (*High-Performance Computing*) (servidores e computação científica).

AArch64: Modelo de programação (1/3)

- Arquitetura do conjunto de instruções (ISA) **ortogonal** do tipo *load/store*; endereços de 64 bits.
- Todas as instruções têm o mesmo comprimento: 32 bits.
- 31 registos de 64 bits para uso geral: X0-X30.
- 1 registo “virtual” (X31) com dupla função: tem o valor 0 ou o **apontador para topo da pilha**.
- O *program counter* não é diretamente acessível.
- Os 32 bits menos significativos de cada registo de uso geral são designados por W0-W30.



AArch64: Modelo de programação (2/3)

- Geralmente, as instruções têm 3 operandos: destino, fonte1 e fonte2
- Os operandos podem ser de 32 ou 64 bits. Exemplo:

```
ADD W0,W1,W2 // adição de registos de 32 bits
```

```
ADD X0,X1,X2 // adição de registos de 64 bits
```

```
ADD X0,X1,#42 // adição de um valor imediato a registo de 64 bits
```

- Atribuições de um valor a registos $W<n>$ colocam os 32 bits mais significativos de $X<n>$ a zero.
- Na maioria das instruções, o registo 31 produz o valor 0 e não é alterado na escrita.
- Quando usado como registo de endereço em instruções *load/store* e em algumas operações aritméticas, o registo 31 dá acesso ao **apontador para o topo da pilha**.
- São permitidos acessos não-alinhados a memória num grande número de situações. Convenções de organização de código podem impor condições de alinhamento (*software*).

AArch64: Modelo de programação (3/3)

- Existem quatro indicadores (*flags*) NZCV com o seguinte significado:

Flag	Nome	Descrição
N	Negativo	N = bit mais significativo do resultado ($1 \rightarrow$ negativo)
Z	Zero	Z=1 se valor igual a 0
C	Carry	Fica a 1 se a operação faz <i>overflow</i> (sem sinal)
V	Overflow	Fica a 1 se a operação faz <i>overflow</i> (com sinal)

- Códigos de condições `<cond>` (para instruções condicionais):

Cód.	Significado	Cond.	Cód.	Significado	Cond.
EQ	igual a	Z=1	NE	diferente de	Z=0
CS	carry set	C=1	HS	igual ou maior (s/ sinal)	C=1
CC	carry clear	C=0	LO	menor que (s/ sinal)	C=0
MI	negativo	N=1	PL	positivo ou zero	N=1
VS	overflow (c/ sinal)	V=1	VC	sem overflow (c/ sinal)	V=0
HI	maior que (s/ sinal)	(C=1) e (Z=0)	LS	menor ou igual (s/ sinal)	(C=0) e (Z=1)
GE	maior ou igual (c/ sinal)	N=V	LT	menor que (c/sinal)	N!=V
GT	maior que (c/ sinal)	(Z=0) e (N=V)	LE	menor ou igual (c/sinal)	(Z=1) e (N!=V)

(Nota: códigos AL e NV: execução sem condições)

Assuntos

1 Microprocessadores ARM

2 Instruções básicas

Instruções de processamento de dados

- ➡ Estas instruções têm o formato: **Instrução Rd, Rn, Op2**
 - Rd** Registo de destino (X_0, X_1, \dots, X_{30} ou W_0, W_1, \dots, W_{30})
 - Rn** Fonte de um dos valores usados na operação
 - Op2** Fonte do outro valor usada na operação.
- ➡ Op2 pode ser um registo, um registo *modificado* ou um valor imediato.

Tipo	Instruções
Aritmética	ADD{S}, SUB{S}, ADC{S}, SBC{S}, NEG
Lógicas	AND{S}, BIC{S}, ORR, ORN, EOR, EON
Comparação	CMP, CMN, TST
Movimento	MOV, MOVN

- ➡ As variantes que terminam com S ({s}) afetam as *flags*.
- ➡ As instruções de comparação também afetam as *flags*.
- ➡ Valores imediatos têm 12 bits.

Instruções de deslocamento

➡ Instruções LSL, LSR, ASR e ROR

LSL Logical shift left



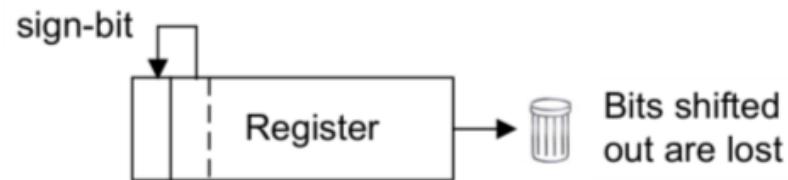
Multiplication by 2^n where n is the shift amount

LSR Logical shift right



Unsigned division by 2^n where n is the shift amount

ASR Arithmetic shift right



Division by 2^n , where n is the shift amount, preserving the sign bit

ROR Rotate right



Bit rotate with wrap around from LSB to MSB

Controlo de fluxo

➡ Instruções de salto / chamada de sub-rotina / retorno de sub-rotina

Instrução	Efeito
B etiqueta	salto para PC \pm 128 MiB
BL etiqueta	como B, mas guarda endereço de retorno em X30
B.<cond> etiqueta	salto condicional para PC \pm 1 MiB
BR X<n>	salto para posição com endereço no registo X<n>
BLR X<n>	como BR, mas guarda endereço de retorno em X30
RET {X<n>}	como BR, mas indica retorno de sub-rotina; por omissão n=30

➡ Instruções de salto condicional com comparação incluída

Instrução	Efeito
CBZ Reg, etiqueta	se Reg=0, salto para PC \pm 1 MiB
CBNZ Reg, etiqueta	se Reg=1, salto para PC \pm 1 MiB
TBZ Reg, bit, etiqueta	se Reg[bit]=0, salto para PC \pm 32 KiB
TBNZ Reg, bit, etiqueta	se Reg[bit]=1, salto para PC \pm 32 KiB

Reg pode ser X<n> ou W<n>

CB:Compare and branch TB: test and branch

Instruções de leitura de memória

► Formato: **LDR Reg, endereço**

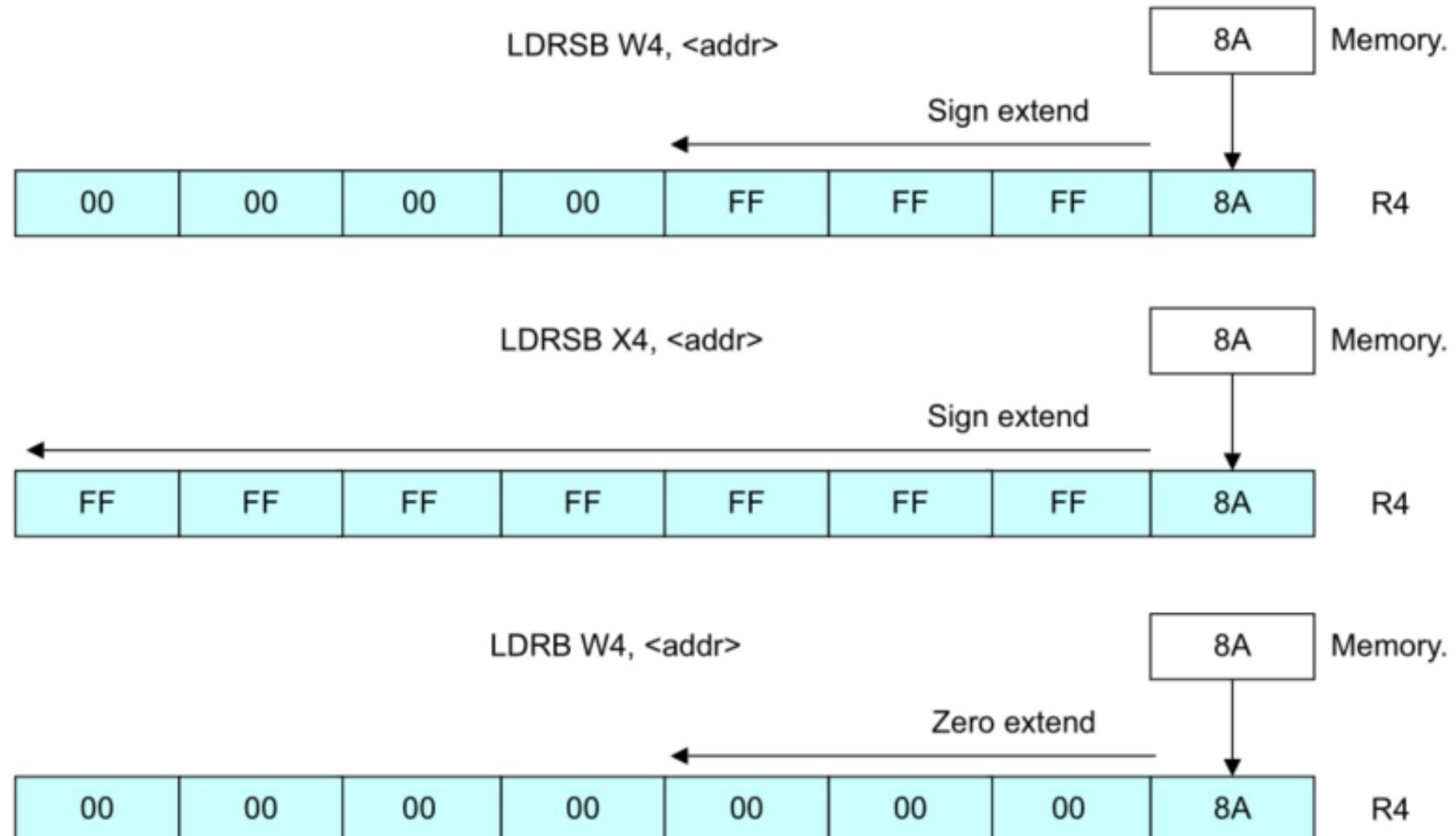
Instrução	Tamanho	Extensão	Registo
LDRB	8 bit	com 0	W<n>
LDRSB	8 bit	de sinal	W<n> ou X<n>
LDRH	16 bit	com 0	W<n>
LDRSH	16 bit	de sinal	W<n> ou X<n>
LDRSW	32 bit	de sinal	X<n>

► Formato de endereço (simples)

Exemplo	Descrição
LDR X0, [X1]	endereço dos dados é o valor de X1
LDR X0, [X1, #8]	endereço dos dados é o valor de X1+8
LDR X0, [X1,X2]	endereço dos dados é o valor de X1+X2

► Endereço é sempre um número de 64 bits.

Leitura com extensão de representação



Instruções de escrita em memória

- ➡ Formato: **STR Reg, endereço**
- ➡ As instruções seguintes escrevem em memória a parte menos significativa de um registo W<n>.

Instrução	Tamanho
STRB	8 bit
STRH	16 bit

- ➡ Formato de endereço é o mesmo que é usado com as instruções de leitura.
- ➡ Para valores de 32 ou 64 bits, usar os registos respetivos. Exemplos:
 - 32 bits: STR W0, [X1,#8]
 - 64 bits: STR X0, [X1, #8]

Arquitetura AArch64

Parte 2

João Canas Ferreira

Março 2019

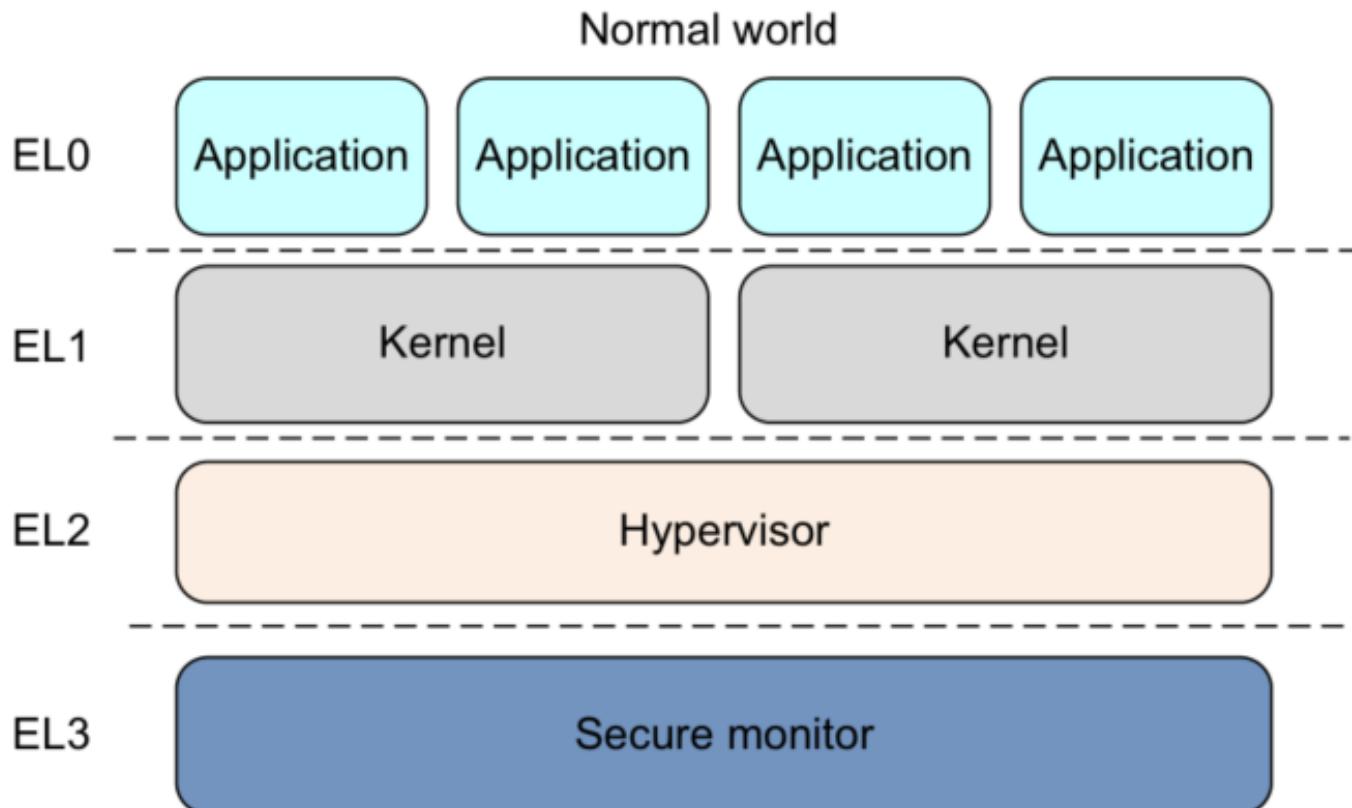


Assuntos

- 1 Aspetos de sistema
- 2 Manipulação de bits
- 3 Execução condicional
- 4 Multiplicação e divisão
- 5 Endereçamento

Níveis de execução

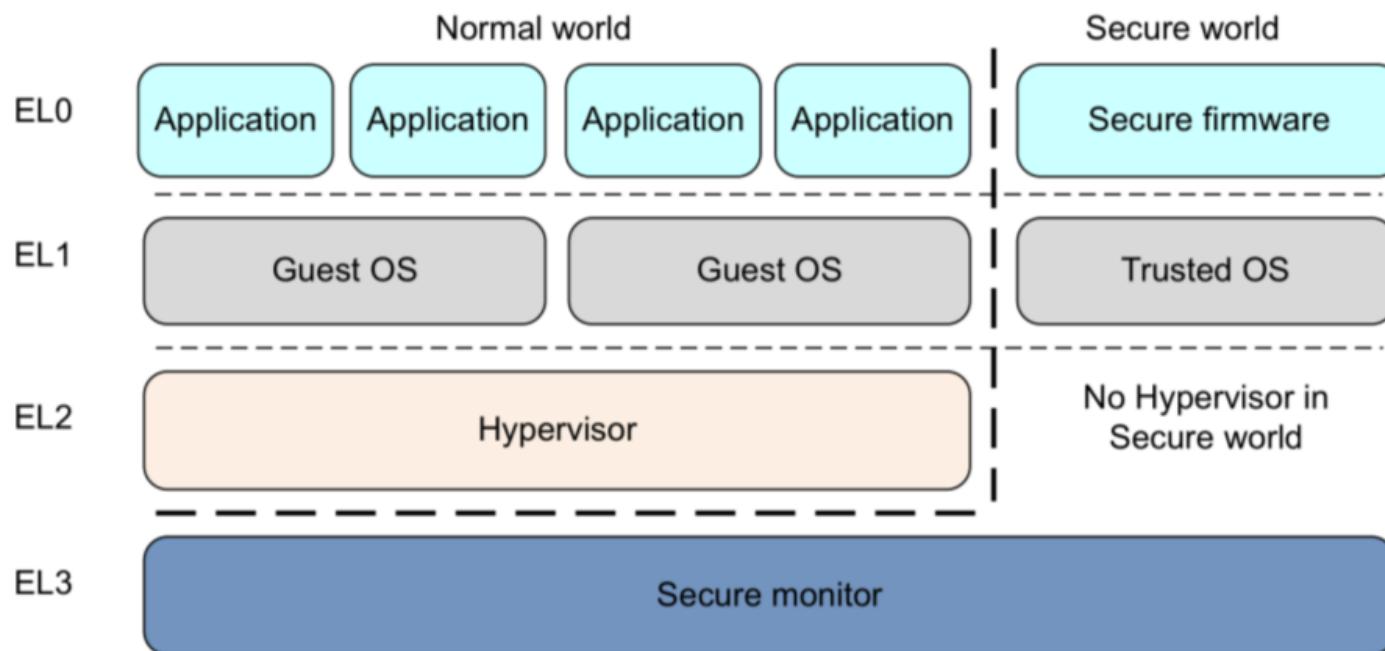
- Na arquitetura ARMv8, a execução de código ocorre num de quatro *níveis de exceção* (EL0–EL3), com o correspondente nível de privilégio.



- *Hypervisor:* gestor de máquinas virtuais (cada sistema operativo hóspede funciona de forma independente).

Estados de segurança

- A arquitetura Armv8-A pode estar em estados de segurança: *seguro* e *não-seguro* (“mundo normal”).



- *Hypervisor*: não existe em modo seguro.
- Núcleos como Linux e Windows correm no nível El1 não-seguro.

Registros de sistema

- A configuração do sistema é controlada pelos *registos de sistema*.
- Apenas alguns registos de sistema são acessíveis em EL0.
- Os registos de sistema são acedidos pelas instruções MRS e MSR

```
MRS    x0, CTR_EL0      // mover valor de CTR_EL0 para X0  
MSR    CTR_EL0, x0      // mover valor de X0 para CTR_EL0
```

- Por exemplo, o registo CTR_EL0 contém informação sobre a memória *cache*.
 - **bits[3:0]** \log_2 do número de palavras da D-cache
 - **bits[19:16]** \log_2 do número de palavras da I-cache
- O registo SCTRL_EL0 controla unidade de gestão de memória (MMU) e verificação de alinhamento.

Modelo de dados

- Tipos de dados suportados nativamente por ARMv8 e equivalência com tipos de dados em C/C++.

Tipo nativo	Tipo em C/C++	Tamanho (bits)
byte	char	8
halfword	short int	16
word	int	32
doubleword	long int long long int apontador	64
quadword	—	128

- A correspondência entre formatos nativos ↔ C/C++ não é universal.
- A tabela indica a correspondência para Linux & GCC.

Assuntos

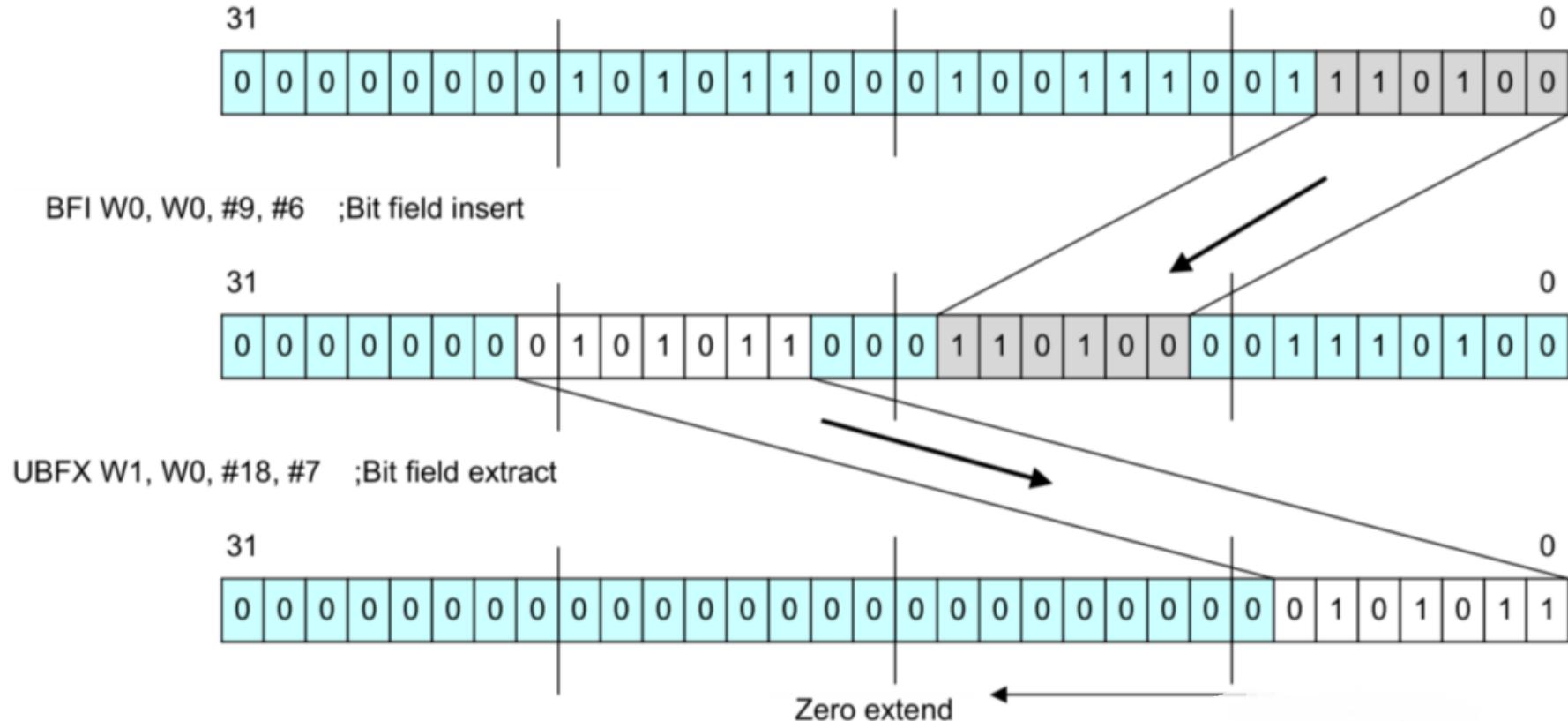
- 1 Aspetos de sistema
- 2 Manipulação de bits
- 3 Execução condicional
- 4 Multiplicação e divisão
- 5 Endereçamento

Manipulação de bits (1/2)

- Partes de um dado (designadas por *bitfields*) são manipuladas por instruções específicas.

Nome	Operação	Descrição
BFI	Bit Field Insert	Copia bits menos significativos de um registo para uma posição qualquer de outro registo
(S/U)BFX	Bit Field Extract	Copia um segmento de bits de um registo para os bits menos significativos de outro (com extensão de sinal ou com zero)
(S/U)BFIZ	Bit Field Insert in Zero	Copia bits menos significativos de um registo para uma posição qualquer da representação de zero
BFXIL	Bit Field Extract and Insert Low	Copia bits de um registo para a posição menos significativa de outro

Manipulação de bits (2/2)



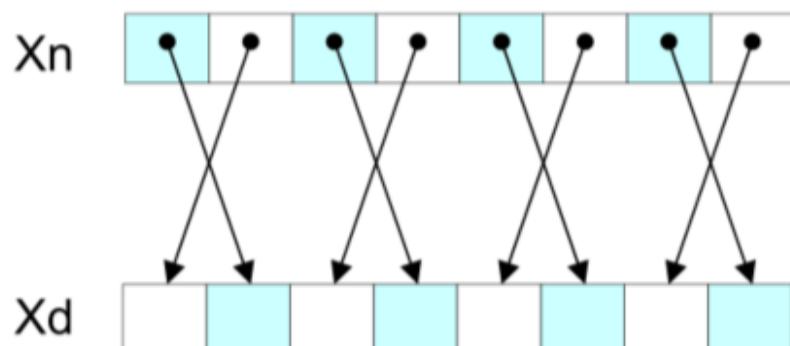
Extensão de operandos

- Existem instruções para alargar o valor de (parte de) um registo $W< n >$ para o registo inteiro $W< n >$ ou $X< n >$.
- Estas instruções são “sinónimos” das instruções de manipulação de bits apropriadas.

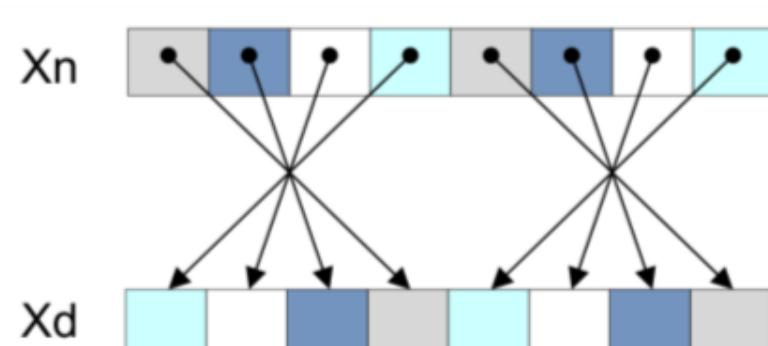
Nome	Exemplo	Descrição
SXTB	SXTB X0 , W1	Extensão de sinal (S) do byte (B) menos significativo de W1 para X0
SXTH	SXTH X0 , W1	Extensão de sinal (S) da halfword (H) menos significativa de W1 para X0
SXTW	SXTW X0 , W1	Extensão de sinal (S) da word de W1 para X0 (2º operando é sempre $W< n >$)
UXTB	UXTB X0 , W1	Extensão com 0 (U) do byte (B) menos significativo de W1 para X0
UXTH	UXTH X0 , W1	Extensão com 0 (U) da halfword (H) menos significativa de W1 para X0

Trocas de bytes, halfwords e words

Nome	Operação	Descrição
CLZ	Count leading zero bits	Número de zeros seguidos a contar da esquerda
CLS	Count leading sign bits	Número de bits iguais ao bit de sinal e que sucedem a este (da esquerda para a direita)
RBIT	Reverse all bits	Troca simétrica de todos os bits
REV	Reverse byte order	Troca a ordem de todos os bytes
REV16		Troca a ordem dos bytes em cada <i>halfword</i>
REV32		Troca a ordem em dos bytes em cada <i>word</i> (apenas registos X<n>)



REV16



REV32

Manipulação de bits: exemplos

- Resultado de várias operações para X0=0x0123456789ABCDEF.

Operação	Resultado
RBIT x0,x0	0xF7B3 D591 E6A2 C480
REV16 x0,x0	0x2301 6745 AB89 EFCD
REV32 x0,x0	0x6745 2301 EFCD AB89
REV x0,x0	0xEFCD AB89 6745 2301
UBFX x0,x0,#16,#4	0x0000 0000 0000 000B
SBFX x0,x0,#24,#8	0xFFFFFFFFFFFFFF89
CLZ x0,x0	7
CLS x0,x0	6

Manipulação lógica de “op2”

- O operando “op2” das operações aritméticas pode ser alterado por operações lógicas antes de ser usado nos cálculos.
- As operações são indicadas por “LSL”, “LSR” ou “ASR” seguidas de uma constante.

```
SUB    X0, X1, X2, ASR #2 // X0 = X1 - (X2 >> 2)  
ADD    X5, X2, #10, LSL #12 // X5 = X2 + (10 << 12)
```

Aplicam-se tanto a registos como valores imediatos.

- Nas operações aritméticas, “op2” também pode ser um registo com operação de extensão.

```
ADD    X0, X1, W2, SXTW // X0=(ext. sinal do valor de W2)+X1
```

- Estas últimas operações (SXTW no exemplo) podem incluir um valor imediato (0-4) que indica um deslocamento tipo LSL

```
ADD    X0, X1, W2, SXTB #2  
// X0 = ((ext.sinal do byte menos significativo de W2) << 2) + X1
```

Assuntos

- 1 Aspetos de sistema
- 2 Manipulação de bits
- 3 Execução condicional
- 4 Multiplicação e divisão
- 5 Endereçamento

Instruções condicionais (1/2)

- Uma instrução cujo resultado depende de uma dada condição é uma *instrução condicional*.
- AArch64 tem um conjunto pequeno deste tipo de instruções.
- CSEL seleciona o valor de um de dois registos. Exemplo:

CSEL X10,X11,X12,PL

- CSINC é similar, mas incrementa o valor do 2º operando. Exemplo:

CSINC, X0,X2,X3,NE

$$X_0 = \begin{cases} X_2 & \text{se } Z = 0 \\ X_3 + 1 & \text{se não} \end{cases}$$

- Coloca registo a 1 (se condição for verdadeira) ou a 0.

CSET W10,GE

Instruções condicionais (2/2)

- Também existem *operações de comparação condicionais*
- CCMP compara dois valores e afeta as *flags* se condição for verdadeira; senão coloca as *flags* conforme indicado pela valor imediato. Exemplo:

CCMP X1,X2,#3,NE

$$\text{NZCV} = \begin{cases} \text{efeito de } X_1 - X_2 & \text{se } Z = 0 \\ 0011_2 & \text{se não} \end{cases}$$

- Variante: operando imediato (21, neste caso) deve estar em [0;31].

CCMP X1,#21,#3,NE

- CCMN é a instrução complementar de CCMP. Exemplo:

CCMN X1,X2,#3,NE

$$\text{NZCV} = \begin{cases} \text{efeito de } X_1 + X_2 & \text{se } Z = 0 \\ 0011_2 & \text{se não} \end{cases}$$

Assuntos

- 1 Aspetos de sistema
- 2 Manipulação de bits
- 3 Execução condicional
- 4 Multiplicação e divisão
- 5 Endereçamento

Operação de multiplicação

- Em geral, o número de bits do resultado de uma multiplicação é igual à soma dos números de bits dos operandos.
- Em muitas linguagens de programação, não acontece sempre isso. Por exemplo em C++:

```
int a, b=250000, c=10000;  
a = b * c;  
cout << a;
```

pode imprimir **-1794967296**. [Porquê?]

- As operações de multiplicação de valores sem sinal e de valores com sinal são realizadas de modo diferente, possuindo mnemónicas diferentes.
- Multiplicações são frequentemente seguidas de adições: operação *multiply-and-add*.
- O conjunto de instruções ARMv8 inclui diversas instruções de multiplicação adaptadas a cada uma das situações.

Instruções de multiplicação

➡ Resumo das instruções de multiplicação

Nome	Formato	Operação
MUL	MUL Rd , Rn , Rm	$Rd \leftarrow Rn \times Rm$; ignora <i>overflow</i>
SMULL	SMULL Xd , Wn , Wm	$Xd \leftarrow Wn \times Wm$ (operандos com sinal)
UMULL	UMULL Xd , Wn , Wm	$Xd \leftarrow Wn \times Wm$ (operандos sem sinal)
SMULH	SMULH Xd , Xn , Xm	$Xd \leftarrow (Xn \times Xm) <127:64>$, com sinal
UMULH	UMULH Xd , Xn , Xm	$Xd \leftarrow (Xn \times Xm) <127:64>$, sem sinal
MADD	MADD Rd , Rn , Rm , Ra	$Xd \leftarrow Rd + (Rn \times Rm)$; ignora <i>overflow</i>
SMADDL	SMADDL Xd , Wn , Wm , Xa	$Xd \leftarrow Ra + (Rn \times Rm)$; operандos com sinal
UMADDL	UMADDL Xd , Wn , Wm , Xa	$Xd \leftarrow Ra + (Rn \times Rm)$; operандos sem sinal

- subtract-multiply: MSUB, SMSUBL, UMSUBL
- multiply-negate: MNEG, SMNEGL, UMNEGL

Xn: registo de 64 bits; Wn: registo de 32 bits;

Rn: Xn ou Wn (interpretação coerente na mesma instrução)

Operação e instruções de divisão

- A operação de divisão calcula (numerador÷denominador), produzindo o *quociente inteiro* (arredondado para zero).
- O resto pode ser calculado como numerador-(quociente×denominador) usando a instrução MSUB.
- Divisão por zero tem como “resultado” zero!
- Divisão do número com sinal mais negativo (INT_MIN) por (-1) produz *overflow* (Porquê?). Não é produzida indicação de *overflow* e o “resultado” é INT_MIN.
- As operações de divisão de valores sem sinal e de valores com sinal são diferentes.

Nome	Formato	Operação
SDIV	SDIV Rd , Rn , Rm	$Rd \leftarrow Rn \div Rm$; com sinal
UDIV	UDIV Rd , Rn , Rm	$Rd \leftarrow Rn \div Rm$; sem sinal

Assuntos

- 1 Aspetos de sistema
- 2 Manipulação de bits
- 3 Execução condicional
- 4 Multiplicação e divisão
- 5 Endereçamento

Modos de endereçamento (1/2)

- A arquitetura ARMv8 permite usar várias formas para especificar o endereço usado em instruções *load* e *store*.
- **base**: usar o endereço guardado num registo de 64 bits (base): **[Xb]**.
Endereço efetivo = Xb;
- **base e deslocamento** tem três variantes
 - 1 **[Xb, #imm]**: endereço efetivo = Xb + #imm
 - 2 **[Xb, Xm, {LSL #imm}]**:
endereço efetivo = Xb + Xm $\{\times 2^{\#imm}\}$
 - 3 **[Xb, Wm,(S|U)XTW {#imm}]**:
endereço efetivo = Xb + ((extended) Wm) $\{\times 2^{\#imm}\}$
- **Deslocamento relativo ao PC**: constante é um número N (com sinal) de 19 bits; geralmente representado por uma *etiqueta*:
endereço efetivo = PC $\pm N \times 4$ (signed word offset)

Modos de endereçamento (2/2)

- Dois modos de endereçamento combinam atualização do registo base com acesso.
 - **pré-indexado**: usar endereço efetivo $[Xb, \#imm]$ para acesso e atualizar registo base $Xb \leftarrow$ endereço efetivo
[Xb, #imm]!
 - **pós-indexado**: usar endereço efetivo $[Xb]$ para acesso e atualizar o registo base $Xb \leftarrow Xb + \#imm$
[base], #imm
- ➡ O registo **SP** (stack pointer) pode ser usado como base (registo virtual X31); não pode ser usado como registo de deslocamento (pág. anterior).

Endereçamento “scaled” e “unscaled”

- O endereçamento **[Xb, #imm]** pode ser “scaled” e “unscaled”.
- **scaled:** o valor codificado na instrução é $\#imm / (\text{tamanho do item})$; o valor deve caber em 11 bits (sem sinal)
 - ➡ LDR W1, [X2, #20]: valor imediato codificado: $20 \div 4 = 5$
 - ➡ LDR X2, [X2, #24]: valor imediato codificado: $24 \div 8 = 3$
 - ➡ LDRSH W1, [X2, #20]: valor imediato codificado: $20 \div 2 = 10$
- **unscaled:** $\#imm$ tem 9 bits (com sinal); no cálculo do endereço efetivo, o valor de $\#imm$ é codificado sem modificações.
As instruções têm a letra “U” antes do R final: LDUR, STUR, etc.
 - ➡ LDUR W1, [X2, #20]: valor imediato codificado: X2
 - ➡ LDUR X2, [X2, #20]: valor imediato codificado: X2
 - ➡ LDURSH W1, [X2, #20]: valor imediato codificado: X2
- Usar geralmente as versões “scaled”; o *assembler* usa automaticamente as versões “unscaled” se necessário.

Sub-rotinas

Arquitetura AARCH64

João Canas Ferreira

Março 2019



Assuntos

- 1 Sub-rotinas: aspectos gerais
- 2 Organização de sub-rotinas
- 3 Comunicação C ↔ Assembly
- 4 Exemplos

Decomposição funcional

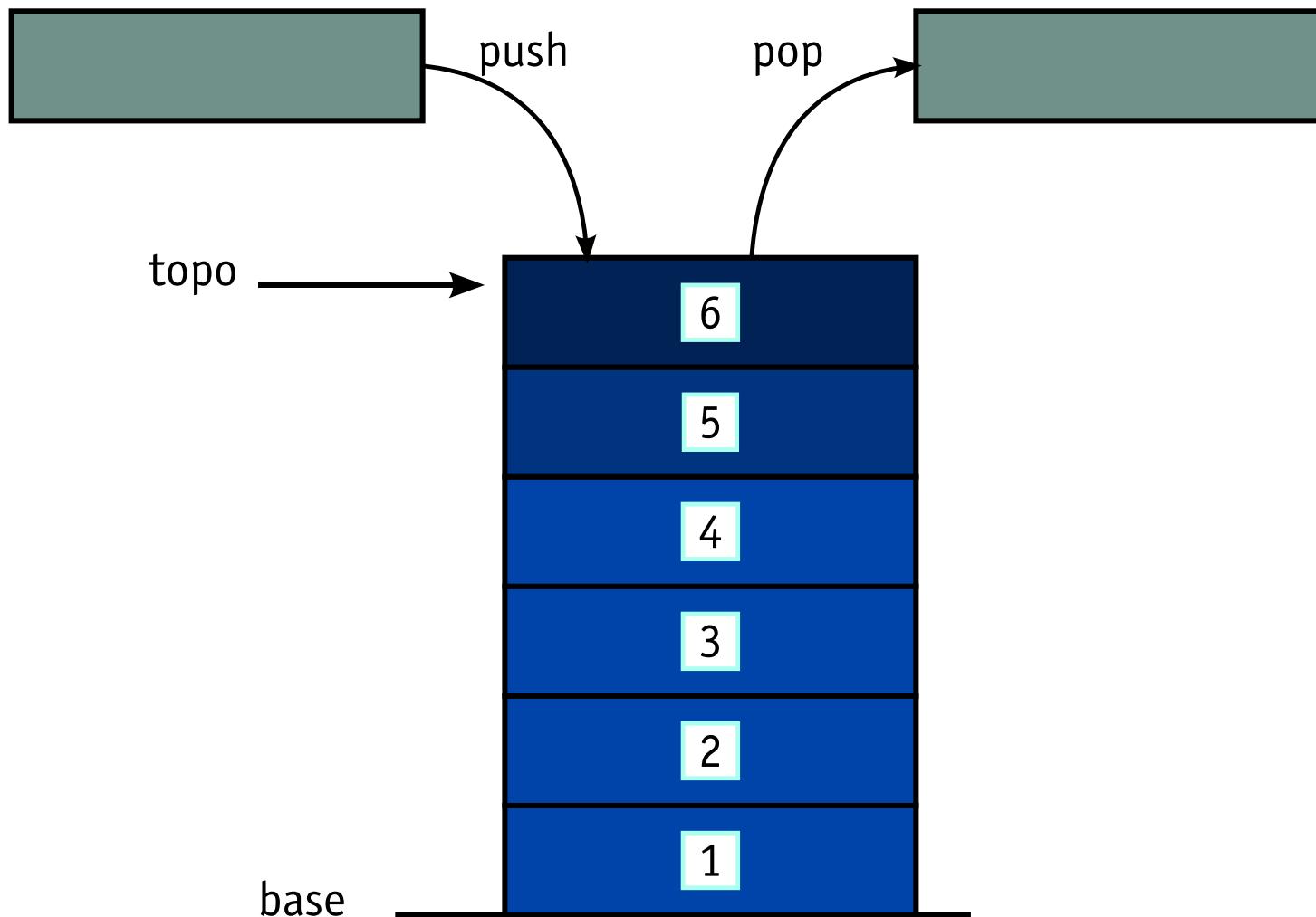
- Programar também é **gerir complexidade** (da especificação e da implementação)
- A decomposição funcional envolve:
 - projetar programa antes de iniciar a codificação
 - decompor tarefas maiores em tarefas mais pequenas (sub-rotinas)
 - criar uma estrutura hierárquica de sub-rotinas
 - testar sub-rotinas individualmente
- A utilização de sub-rotinas é uma forma de *reutilização de código*
- Sub-rotinas podem ser:
 - procedimentos: a sua invocação não produz um valor
 - funções: a sua invocação produz um valor
- Em *assembly* não existe distinção formal entre procedimentos e funções: a designação usada é *procedure* (procedimento)
- CPU suporta sub-rotinas através das instruções: BL / BLR e RET

Interoperabilidade: Convenção de invocação de sub-rotinas

- Uma convenção de invocação de sub-rotinas [CIS] (*Procedure Call Standard*) define como é que sub-rotinas *compiladas separadamente* podem “trabalhar” em conjunto.
- Faz parte da Interface Binária da Aplicação (ABI=Application Binary Interface)
- Respeitar a CIS definida pela ARM para a arquitetura AArch64 implica:
 - 1 Respeitar as restrições de alinhamento da pilha
 - 2 Respeitar o tipo de uso dos registos
 - 3 Respeitar regras na representação de dados em memória (*data layout rules*)

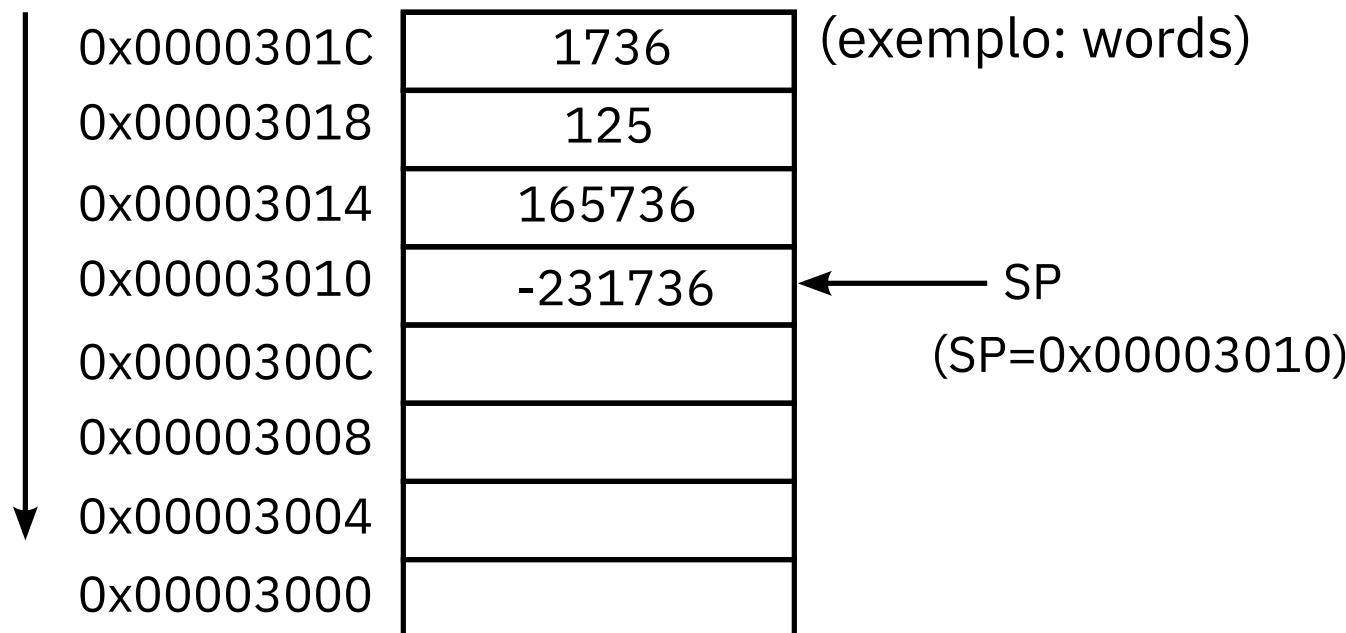
Pilha

- Durante a execução, os programas mantêm uma pilha de dados



Gestão da pilha

- Pilha: zona contígua de memória gerida segundo o princípio LIFO (*Last-In First-Out*).
- Usada para passar parâmetros (se não couberem em registos), guardar variáveis locais e preservar endereços de retorno.
- Pilha gerida através de um apontador para o topo da pilha: SP (registro reservado para esta função)



- O que está na posição 0x0000300C?

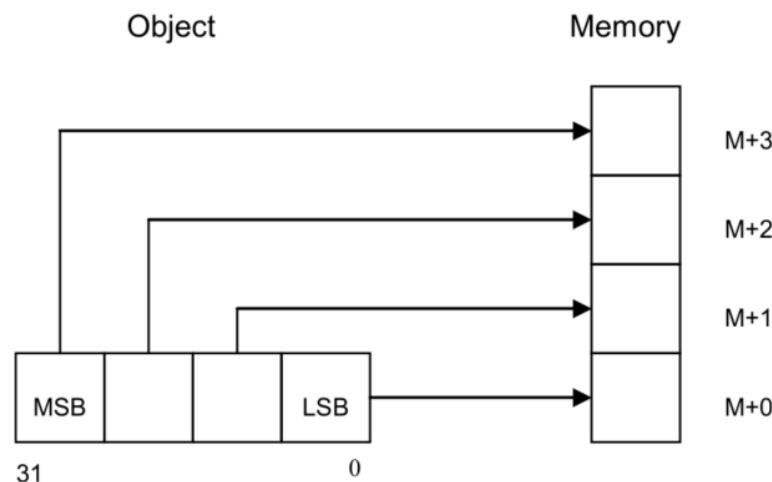
Regras para utilização da pilha

- Valor de SP deve ser sempre múltiplo de 16
- Usar modos de endereçamento apropriados

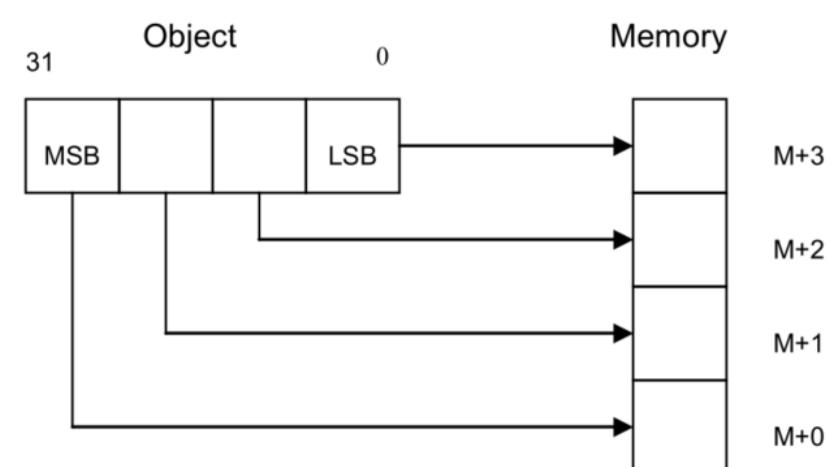
⇒ Qual é o valor de W10 nos seguintes casos?

- 1 ldr W10, [SP, #4]
- 2 ldrh W10, [SP, #8]

⇒ “Layout” de objetos



Little-endian



Big-endian

Assuntos

- 1 Sub-rotinas: aspectos gerais
- 2 Organização de sub-rotinas
- 3 Comunicação C ↔ Assembly
- 4 Exemplos

Colocação dos dados em memória

Type Class	Machine Type	Byte size	Natural Alignment (bytes)
Integral	Unsigned byte	1	1
	Signed byte	1	1
	Unsigned half-word	2	2
	Signed half-word	2	2
	Unsigned word	4	4
	Signed word	4	4
	Unsigned double-word	8	8
	Signed double-word	8	8
	Unsigned quad-word	16	16
	Signed quad-word	16	16
Pointer	Data pointer	8	8
	Code pointer	8	8

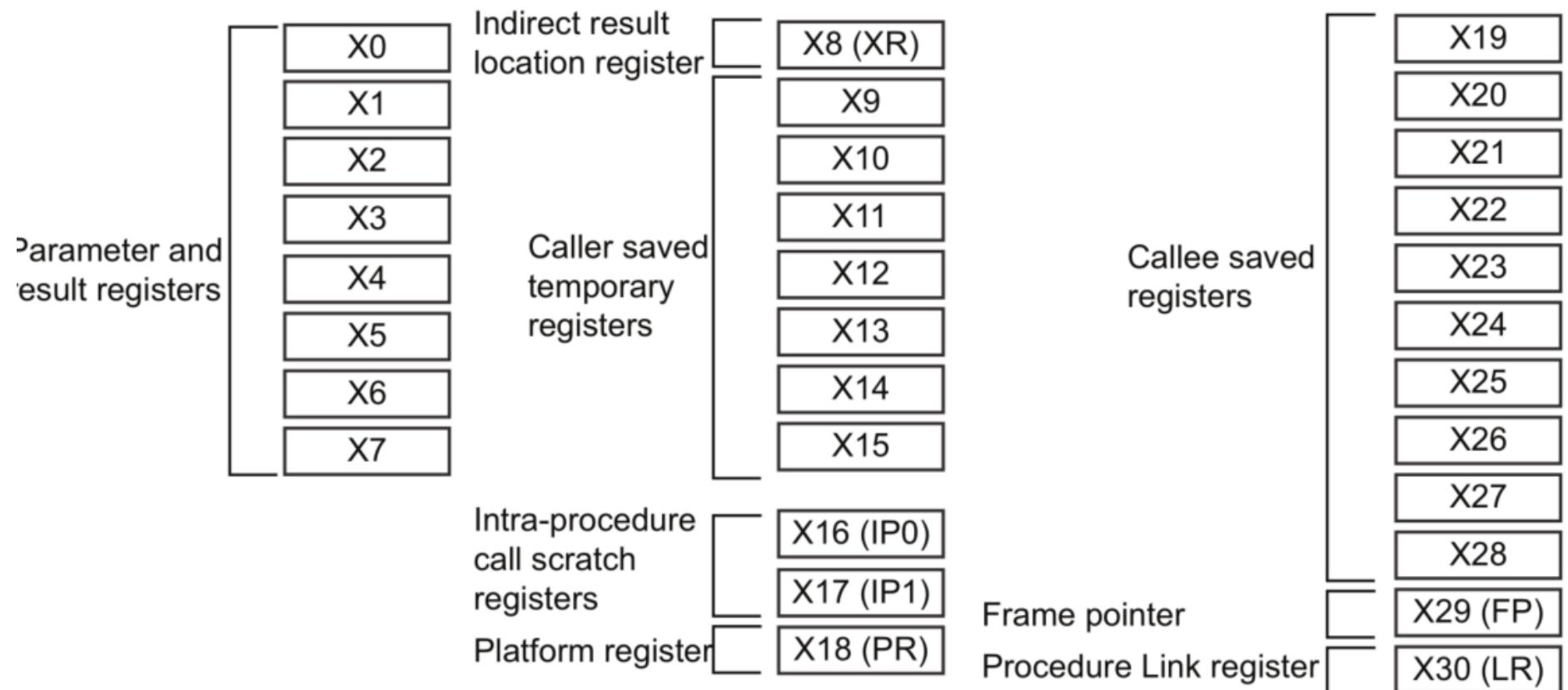
Utilização de registos (1/3)

Register	Special	Role in the procedure call standard
SP		The Stack Pointer.
r30	LR	The Link Register.
r29	FP	The Frame Pointer
r19...r28		Callee-saved registers
r18		The Platform Register, if needed; otherwise a temporary register. See notes.
r17	IP1	The second intra-procedure-call temporary register (can be used by call veneers and PLT code); at other times may be used as a temporary register.
r16	IP0	The first intra-procedure-call scratch register (can be used by call veneers and PLT code); at other times may be used as a temporary register.
r9...r15		Temporary registers
r8		Indirect result location register
r0...r7		Parameter/result registers

Utilização de registos (2/3)

- Para a arquitetura de 64 bits, $R<n>=X<n>$!
 - $x0-x7$: passar argumentos (na chamada) e resultados (no retorno); podem ser alterados pela sub-rotina.
 - $x9-x15$: podem ser usados livremente pela sub-rotina.
 - $x19-x28$: devem ser preservados pela sub-rotina (“callee”).
 - $x8, x16-x18$: não usar!
 - SP contém endereço do “topo” da pilha (endereço mais baixo);
 - LR contém endereço de retorno (link register, x30);
 - FP contém endereço para a *moldura* da sub-rotina que invocou esta (“caller”); registo x29;
 - *moldura*: região da pilha reservada por cada invocação de uma sub-rotina para guardar valores temporariamente.
- ➡ Casos não tratados nesta u.c.: argumentos ou resultado não cabem nos registos disponíveis.

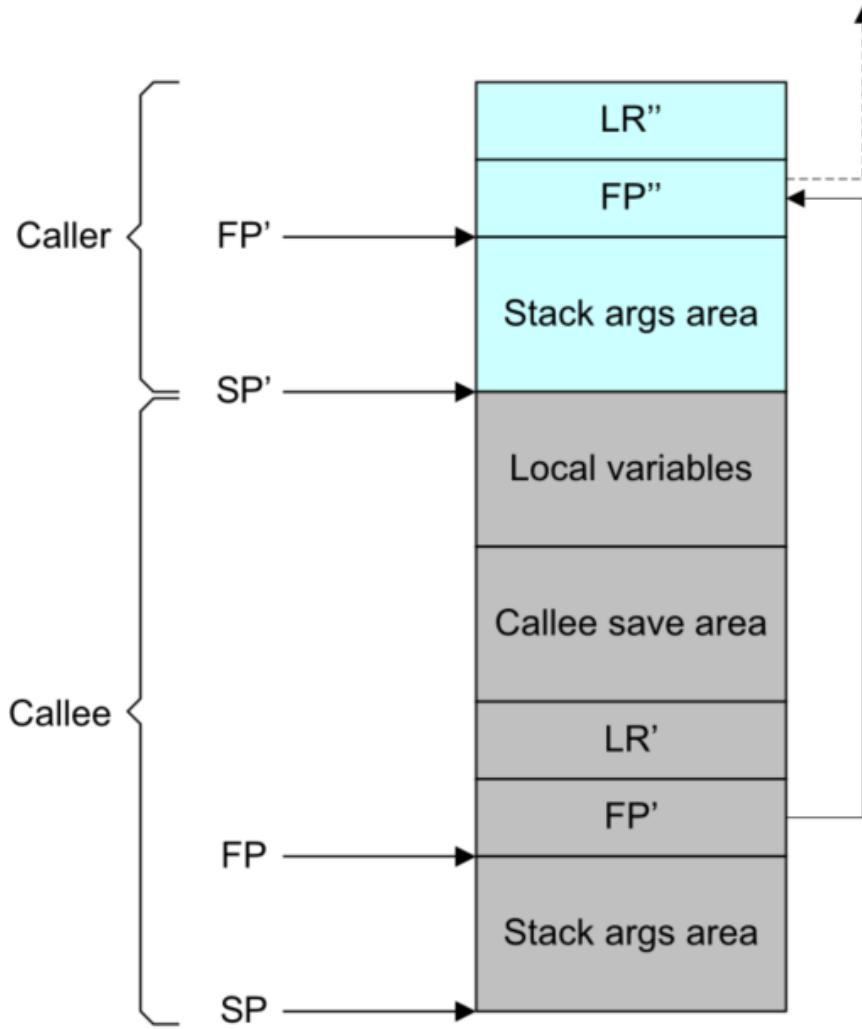
Utilização de registos (3/3)



Regras para invocação de uma sub-rotina

- Em “caller”:
 - ① Os argumentos da função são colocados por ordem (da esquerda para a direita) nos registos x0–x7.
 - ② A instrução BL é usada para invocar a sub-rotina.
- Na sub-rotina invocada (“callee”):
 - ① Construir a moldura (“frame record”).
 - ② Guardar valores de FP e LR na moldura.
 - ③ Fazer os cálculos observando as regras de utilização de registos.
 - ④ Colocar o resultado no registo x0.
 - ⑤ Recuperar os valores originais de FP e LR.
 - ⑥ Terminar a execução da sub-rotina com RET (equivalente: BR x30).
- Execução continua em “caller”.

Organização da moldura



⇒ “Stacks arg area” não é usada nesta u.c. (todos os argumentos são passados em registos)

Assuntos

- 1 Sub-rotinas: aspectos gerais
- 2 Organização de sub-rotinas
- 3 Comunicação C ↔ Assembly
- 4 Exemplos

Invocação a partir de C

- ⇒ Declarar a sub-rotina na sintaxe de C++ [sub-rotina externa] (possivelmente em “header file” com extensão*.h)

```
extern "C" tipo_resultado nome_func (arg1, arg2, ..., argN);
```

- ⇒ Usar normalmente como qualquer função de C++:

```
res = nome_func (12, x, ..., y);
```

- ⇒ Os tipos de dados devem respeitar a correspondência da página 18.

- ⇒ Para usar uma rotina de C:

- 1 Colocar argumentos nos registos corretos
- 2 Invocar a sub-rotina com BL
- 3 Usar o resultado

- ⇒ Variáveis globais podem ser declaradas em C++ ou em assembly

Correspondência de tipos

C/C++ Type	Machine Type
char	unsigned byte
unsigned char	unsigned byte
signed char	signed byte
[signed] short	signed halfword
unsigned short	unsigned halfword
[signed] int	signed word
unsigned int	unsigned word
[signed] long	signed word or signed double-word
unsigned long	unsigned word or unsigned double-word

► Apontadores são valores de 64 bits.

Assuntos

- 1 Sub-rotinas: aspectos gerais
- 2 Organização de sub-rotinas
- 3 Comunicação C ↔ Assembly
- 4 Exemplos

Invocar sub-rotina em assembly

Ficheiro: t1.c

```
#include <stdio.h>
extern int add2(int);

int main(void)
{
    int x = add2(10);
    printf("%d\n", x);
    return 0;
}
```

⇒ Novas instruções:

stp store pair of registers
ldp load pair of registers

Ficheiro: add2.s

```
.global add2
.type add2, %function

.text
add2:   stp x29,x30,[sp,#-16]!
        mov x29,sp
        add w0,w0,#2
        ldp x29,x30,[sp],#16
        ret
```

Versões em C e assembly (1/4)

Ficheiro: max.c

```
int func(int a, int b)
{
    int m;
    if (a>b)
        m=a;
    else
        m=b;
    return m;
}
```

Ficheiro: max.s

```
.text
.global func
.type func, %function
func:
    stp    x29, x30, [sp, #-16]!
    mov    x29, sp
    cmp    w0, w1
    csel   w0, w0, w1, ge
    ldp    x29, x30, [sp], #16
    ret
```

Versões em C e assembly (2/4)

Ficheiro: test.c

```
long test(long a,long b)
{
    long m;
    if (a>b)
        m=a/b;
    else
        m=b/a;
    return m;
}
```

gerado por compilador →

Ficheiro: test.s

```
.text
.global test
.type test, %function
test:
    stp x29, x30, [sp,#-16]!
    mov x29, sp
    cmp x0, x1
    bgt .L5
    sdiv x0, x1, x0
.L1:
    ldp x29, x30, [sp],#16
    ret
.L5:
    sdiv x0, x0, x1
    b .L1
```

Versões em C e assembly (3/4)

Ficheiro: loop.c

```
int loop(int n)
{
    int i, j;

    j = 0;
    for (i=1; i<=n; i++)
        j = j + i;
    return j;
}
```

gerado por compilador →

Ficheiro: loop.s

```
loop:
    stp x29, x30, [sp,#-16]!
    add x29, sp, #0 //?
    cmp w0, #0
    ble .L4
    mov w2, w0
    mov w0, #0
    mov w1, #1
.L3:
    add w0, w0, w1
    add w1, w1, #1
    cmp w2, w1
    bge .L3
.L1:
    ldp x29, x30, [sp],#16
    ret
.L4:
    mov w0, #0
    b .L1
```

Versões em C e assembly (4/4)

Ficheiro: loop2.c

```
long loop2(long *vect,
           int n)
{
    int i;
    long j=0;
    for (i=0; i<n; i++)
        j = j + vect[i];
    return j;
}
```

gerado por compilador →
Constantes não precisam de #

Ficheiro: loop2.s

```
loop2:
    stp x29, x30, [sp, -16]!
    add x29, sp, 0
    cmp w1, 0
    ble .L4

    mov x2, x0
    sub w1, w1, 1
    add x0, x0, 8
    add x3, x0, x1, lsl 3
    mov x0, 0

.L3:
    ldr x1, [x2], 8
    add x0, x0, x1
    cmp x2, x3
    bne .L3

.L1:
    ldp x29, x30, [sp], 16
    ret

.L4:
    mov x0, 0
    b .L1
```

Utilização de variáveis globais

- Sub-rotina que retorna letra da posição “n” de uma cadeia de caracteres.

```
.arch armv8-a
.global nome
.data
.align 3
//alinhamento 8
nome:
.string "MPCP 2018/19"
```

```
.text
.align 2 // alinhamento 4
.global get_letter
.type get_letter,%function
get_letter:
    stp    x29, x30, [sp,-16]!
    mov    x29, sp
    cmp    w0, 12
    bhi   L1
    ldr    x1, =msg
    ldrb   w0, [x1, w0, sxtw]
    b     Lfim
L1:
    mov    w0, 0
Lfim:
    ldp    x29, x30, [sp], 16
    ret
```

Diretivas de declaração de dados

Diretiva	Efeito
.byte valor{,valor}	espaço inicializado com valores de 1 byte
.hword valor{,valor}	espaço inicializado com valores de tipo halfword
.word valor {,valor}	espaço inicializado com valores de tipo word
.quad valor {,valor}	espaço inicializado com valores inteiros de 8 bytes
.string "str"	espaço inicializado com os caracteres de “str” com 0 no final
.space tamanho {,valor}	inicializa “tam” bytes com valor (ou 0 se valor for omitido)

Invocar funções de C

Ficheiro: print_msg_tb.c

```
#include <stdio.h>
extern void
print_msg(char *msg);

int main(void)
{
    char mensagem[]="ARMv8-A!";
    print_msg(mensagem);
    return 0;
}
```

→ Como exemplo, print_msg acrescenta 10 ao código da 1^a letra (M → K)

Ficheiro: print_msg.s

```
.text
.align 2
.global print_msg
.type print_msg,%function

print_msg:
    stp x29, x30,[sp,-16]!
    mov x29, SP
    ldrb w9, [x0]
    add w9, w9, 10
    strb w9, [x0]
        // invocar
    bl puts
    ldp x29, x30,[sp],16
    ret
```

Mais variáveis globais

Ficheiro: addr_tb.c _____ Ficheiro: addrs.s _____

```
#include <stdio.h>
unsigned long secret =
    0xaabbccdd12345678;
extern void func_addr(void);

int main(void)
{
    func_addr();
    printf("0x%lx\n", secret);
    return 0;
}
```

→ Este exemplo imprime
0xbcf023552468acf0

```
.data
.extern secret
.align 3
num: .quad 0x1234567812345678

.text
.align 2
.global func_addr
.type func_addr,%function

func_addr:
    ldr x0, =num
    ldr x1, =secret
    ldr x2, [x0]
    ldr x3, [x1]
    add x3, x3, x2
    str x3, [x1]
    ret
```

Operações em vírgula flutuante

João Canas Ferreira

Abril 2019



Assuntos

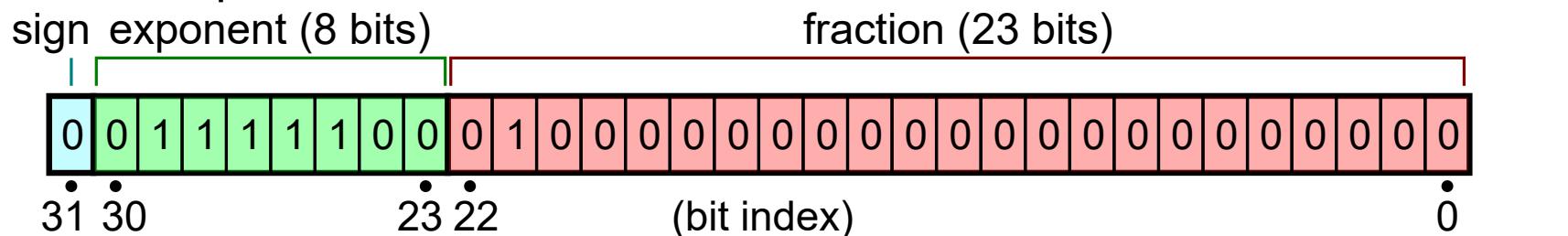
1 Vírgula flutuante: aspectos gerais

2 Categorias de instruções

Tipos de dados

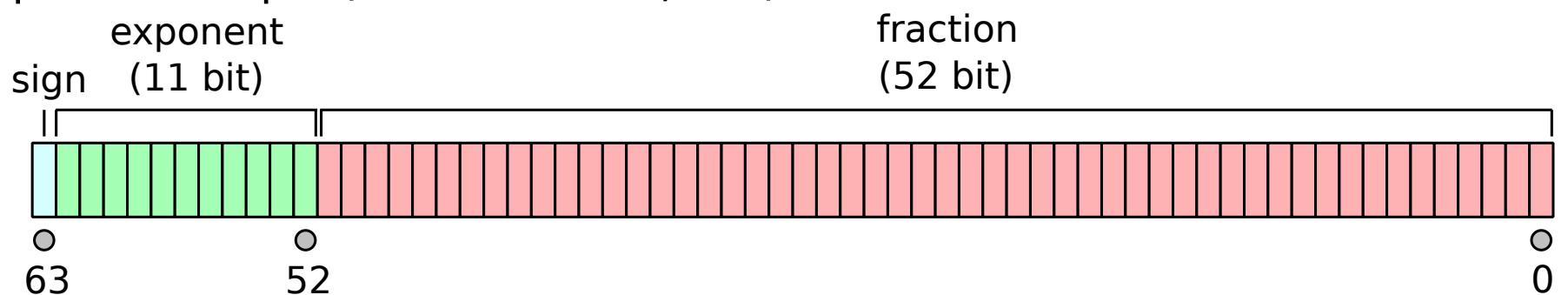
A arquitetura AArch64 suporta 3 tipos de dados em vírgula flutuante de acordo com a norma IEEE-754-2008:

- 1) precisão simples (float em C/C++)



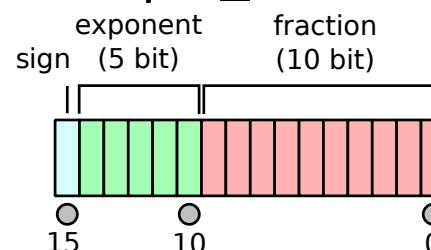
32 bits

- 2) precisão dupla (double em C/C++)



64 bits

- 3) meia precisão (extensão em C; tipo `_Float16`)



16 bits

Exemplo em C

⇒ Atenção: Em C, `_Float16` serve apenas para armazenamento; para cálculos é convertido em `float` ou `double`.

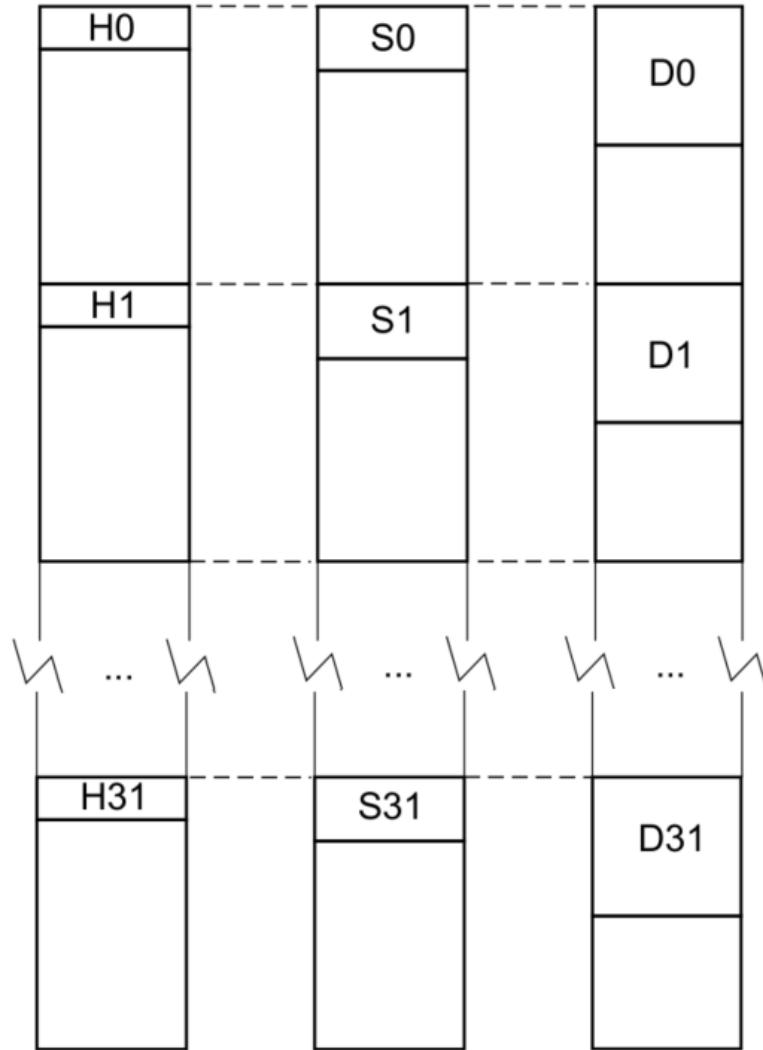
```
#include <stdio.h>
float    vFloat = 126.2346f;
double   vDouble = 124.23434556;
_Float16 vHalf[2];

int main (void)
{
    printf("%.8f %.8f\n", vFloat, vDouble);
    vHalf[0] = vFloat;    vHalf[1] = vDouble;
    vFloat   = vHalf[0];  vDouble = vHalf[1];
    printf("%.8f %.8f\n", vFloat, vDouble);
}
```

⇒ Resultado impresso

126.23459625	124.23434556
126.25000000	124.25000000

Registros dedicados



- Registros D0–D31: precisão dupla
- Registros S0–S31: precisão simples
 - 32 bits menos significativos de D0–D31
- Registros H0–H31: meia precisão
 - 16 bits menos significativos de D0–D31 (e de S0–S31)

Resultados e argumentos de sub-rotinas

- Os registos D0-D7 (S0-S7) são usados para passar argumentos e retornar valores (D0 ou S0).
- Registos são atribuídos aos parâmetros por ordem.
 - Por exemplo, se o 1º argumento for do tipo float e o 2º do tipo double, são passados em S0 e D1, respetivamente.
 - A atribuição de registos VF e de registos inteiros (Xn ou Wn) são independentes.
- Registos 8–15 (S ou D) devem ser preservados pela sub-rotina; os outros registos não são preservados (em geral).

Assuntos

1 Vírgula flutuante: aspectos gerais

2 Categorias de instruções

Operações aritméticas

- Operandos são do mesmo tipo, que é também o tipo do resultado
- Formato geral: FXXXX Rdest, Rn, Rm $R \in \{H,S,D\}$
- Operações combinadas: FXXXX Rdest, Rn,Rm,Ra $R \in \{H,S,D\}$

Instrução	Operação
FADD	$R_{dest} = R_n + R_m$
FSUB	$R_{dest} = R_n - R_m$
FDIV	$R_{dest} = R_n / R_m$
FMUL	$R_{dest} = R_n \times R_m$
FNMUL	$R_{dest} = -(R_n \times R_m)$
FMADD	$R_{dest} = (R_n \times R_m) + R_a$
FNMADD	$R_{dest} = -(R_n \times R_m) - R_a$
FMSUB	$R_{dest} = -(R_n \times R_m) + R_a$
FNMSUB	$R_{dest} = (R_n \times R_m) - R_a$

Funções matemáticas

- Operandos são do mesmo tipo, que é também o tipo do resultado
- Formato geral (1 operando): FXXXX Rdest, Rn $R \in \{H,S,D\}$
- Formato geral (2 operandos): FXXXX Rdest, Rn, Rm $R \in \{H,S,D\}$

Instrução	Operação
FABS	$R_{dest} = R_n $
FMAX	$R_{dest} = \max(R_n; R_m)$
FMIN	$R_{dest} = \min(R_n; R_m)$
FNEG	$R_{dest} = -R_n$
FSQRT	$R_{dest} = \sqrt{R_n}$
FRINTI	$R_{dest} = \text{arredondar}(R_n)$

Movimentação de dados

- Entre registos VF, sem conversão (FMOV)
 - FMOV Rd, Rn $Rd \leftarrow Rn$
- Entre registos VF e registos de uso geral *sem conversão* (FMOV)
 - $Wd \leftarrow \{Hn, Sn\}$
 - $Xd \leftarrow \{Hn, Dn\}$
 - $Hd \leftarrow \{Wn, Xn\}$
 - $Sd \leftarrow Wn$
 - $Dd \leftarrow Xn$
- Entre registos e memória: LDR, LDUR, STR, STUR, LDP e STP
- Condicional: FCSEL Rd, Rn, Rm, cond
 - Se (cond=true) $Rd \leftarrow Rn$, senão $Rd \leftarrow Rm$

Exemplo de operações básicas em VF

```
#include <stdio.h>
#include <math.h>
float VarF = 34.56f;
double VarG = -M_PI;
double vect[]={1.0, -1.23, 7.56};
extern double vf_func(double *v,
                      int n, double coef);

int main(void) {
    const int n = 3;
    double res;
    extern double myVar1;
    res = vf_func(vect, n, 2.5);
    for (int i = 0; i<n; i++)
        printf("%f ",vect[i]);
    printf("\nres=%f VarF=%f VarG=%f
          myVar1=%f\n",
          res, VarF, VarG, myVar1);
    return 0;
}

2.500000 -3.075000 7.560000
res=3.560000 VarF=34.560001
VarG=1.780000 myVar1=1.780000
```

```
.data
.global myVar1
myVar1: .double 1.78
.extern VarG

.text
.global vf_func
.type vf_func, %function
// X0: ponteiro para vetor
// W1: número de elementos
// D0: coeficiente
vf_func:
    ldr    D1, [X0]
    fmov   D2, D0
    fmul   D0, D1, D2
    str    D0, [X0]
    ldr    D1, [X0, 8]
    fmul   D0, D1, D2
    str    D0, [X0, 8]
    ldr    D2, myVar1
    ldr    X12, =VarG
    str    D2, [X12]
    fadd   D0, D2, D2
    ret
```

Operações de comparação

- As instruções de comparação afetam os indicadores N, Z, C e V.
- Se os operandos não puderem ser comparados, então N=0, Z=0, C=1 e V=1.
- Todas as operações de VF podem afetar os indicadores.
- Os indicadores podem ser acedidos via registo especial NZCV (onde ocupam os bits 31:28)

Instrução	Operação
FCMP Rn, RM	NZCV=comparação(Rn; Rm)
FCMP Rn, #0.0	NZCV=comparação(Rn; Rm)
FCCMP Rn, Rm, #nzcv, cond	se cond NZCV=comparação(Rn; Rm) senão NZVC=#nzvc

#nzvc: valor entre 0-15, composto pelos quatro bits de NZCV

Desvio: Manipulação direta dos indicadores

⇒ Exemplo de manipulação direta dos indicadores

MRS x1, NZCV

MOV x2, 0x30000000

BIC x1, x1, x2 // C e V colocados a 0 (bits 29 e 28)

ORR x1, x2, 0xC0000000 // N e Z colocados a 1 (bits 31 e 30)

MSR NZCV, x1 // atualizar indicadores

Conversão entre formatos VF

➡ Instruções para converter entre formatos de precisão diferente.

Instrução	Operação
FCVT Sd, Hn	meia precisão para precisão simples
FCVT Dd, Hn	meia precisão para precisão dupla
FCVT Hd, Sn	precisão simples para meia precisão
FCVT Dd, Sn	precisão simples para precisão dupla
FCVT Hd, Dn	dupla precisão para meia precisão
FCVT Sd, Dn	dupla precisão para precisão simples

- Conversão de formato de precisão mais baixa para precisão mais alta: o valor numérico não é afetado
- Conversão de formato de precisão mais alta para precisão mais baixa: pode ocorrer arredondamento ou produzir NaN.

Conversão VF → inteiros

- A conversão pode gerar uma exceção, se o valor não for representável no formato de destino.
- Para números inteiros em complemento para 2 [com sinal] ($R \in \{H;S;D\}$):

FCVTNS Wd , Rn arredondar para inteiro 32 bits $[-2^{31}; 2^{31} - 1]$

FCVNTS Xd , Rn arredondar para inteiro 64 bits $[-2^{63}; 2^{63} - 1]$

- Para números inteiros sem sinal ($R \in \{H;S;D\}$):

FCVTNU Wd , Rn arredondar para inteiro 32 bits $[0; 2^{32} - 1]$

FCVNTU Xd , Rn arredondar para inteiro 64 bits $[0; 2^{64} - 1]$

Conversão inteiros → VF

- A conversão pode gerar uma exceção, se o valor não for representável no formato de destino.
- De números inteiros em complemento para 2 [com sinal] ($R \in \{H;S;D\}$):

SCVTF Rd , Wn converter inteiro 32 bits para VF

SCVTF Rd , Xn converter inteiro 64 bits para VF

- De números inteiros sem sinal [binário simples] ($R \in \{H;S;D\}$):

UCVTF Wd , Rn converter inteiro 32 bits (binário simples) para VF

UCVTF Xd , Rn converter inteiro 64 bits (binário simples) para VF

Processamento paralelo de dados

Instruções SIMD da arquitetura AArch64

João Canas Ferreira

Maio 2019



Assuntos

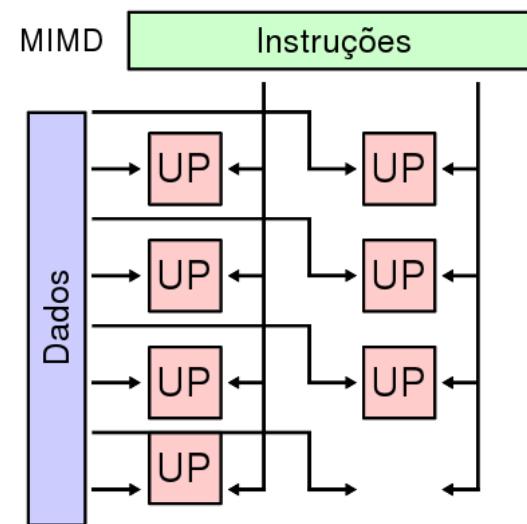
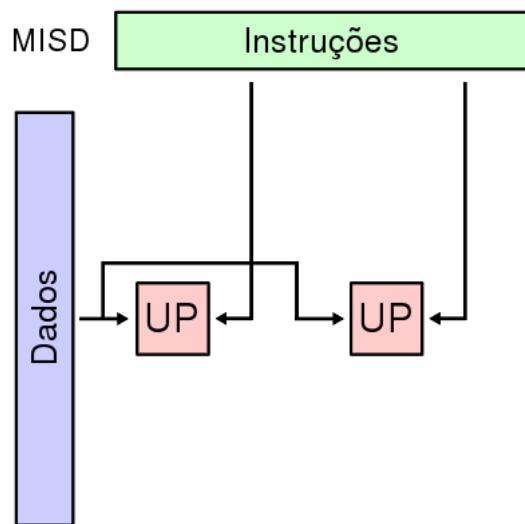
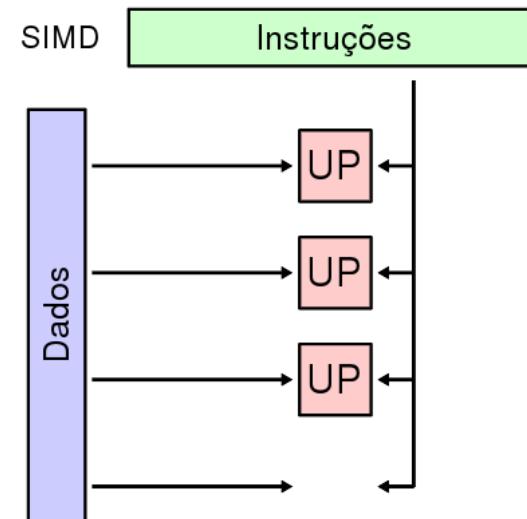
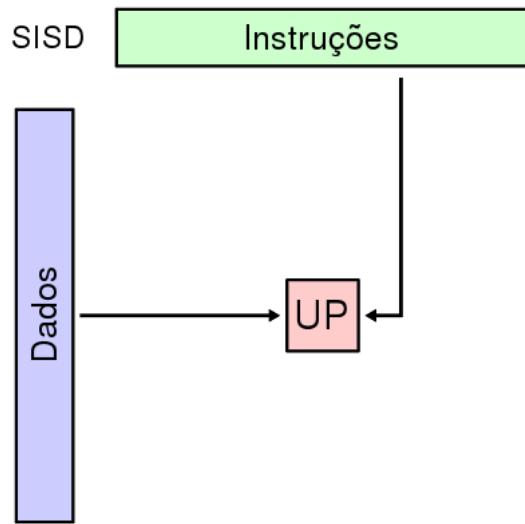
- 1 Processamento paralelo: introdução
- 2 Tecnologia NEON
- 3 Princípios gerais das instruções NEON
- 4 Principais instruções SIMD

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:

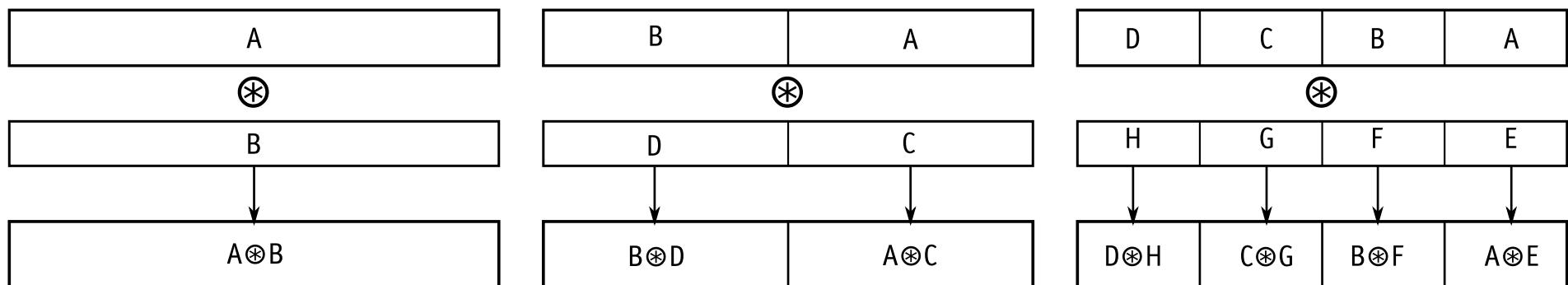
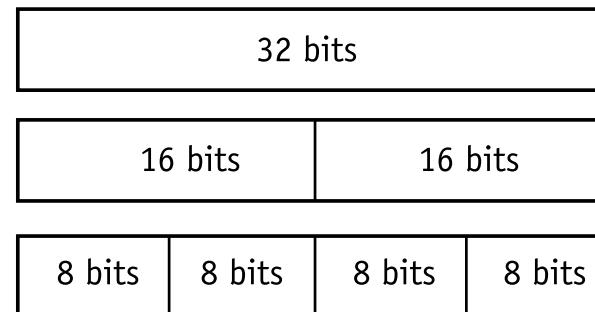
- **SISD:** *Single instruction stream, single data stream*
processador convencional: 1 instrução processa 1 conjunto de dados
- **SIMD:** *Single instruction stream, multiple data streams*
1 instrução processa vários conjuntos de dados (processamento vetorial, também usado em GPUs)
- **MISD:** *Multiple 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

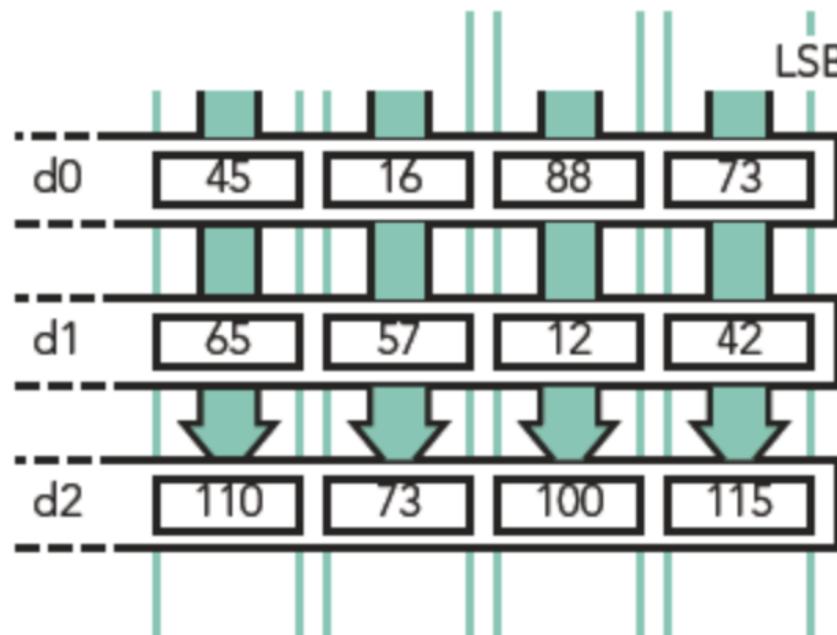
- Requer algoritmos com processamento de dados “uniforme” (todos os elementos do vetor são processados da mesma forma)
- Necessidade de adaptar a codificação do algoritmo
- Alguns compiladores têm dificuldade em aproveitar bem este tipo de instruções: recurso a linguagem *assembly*.

Assuntos

- 1 Processamento paralelo: introdução
- 2 Tecnologia NEON
- 3 Princípios gerais das instruções NEON
- 4 Principais instruções SIMD

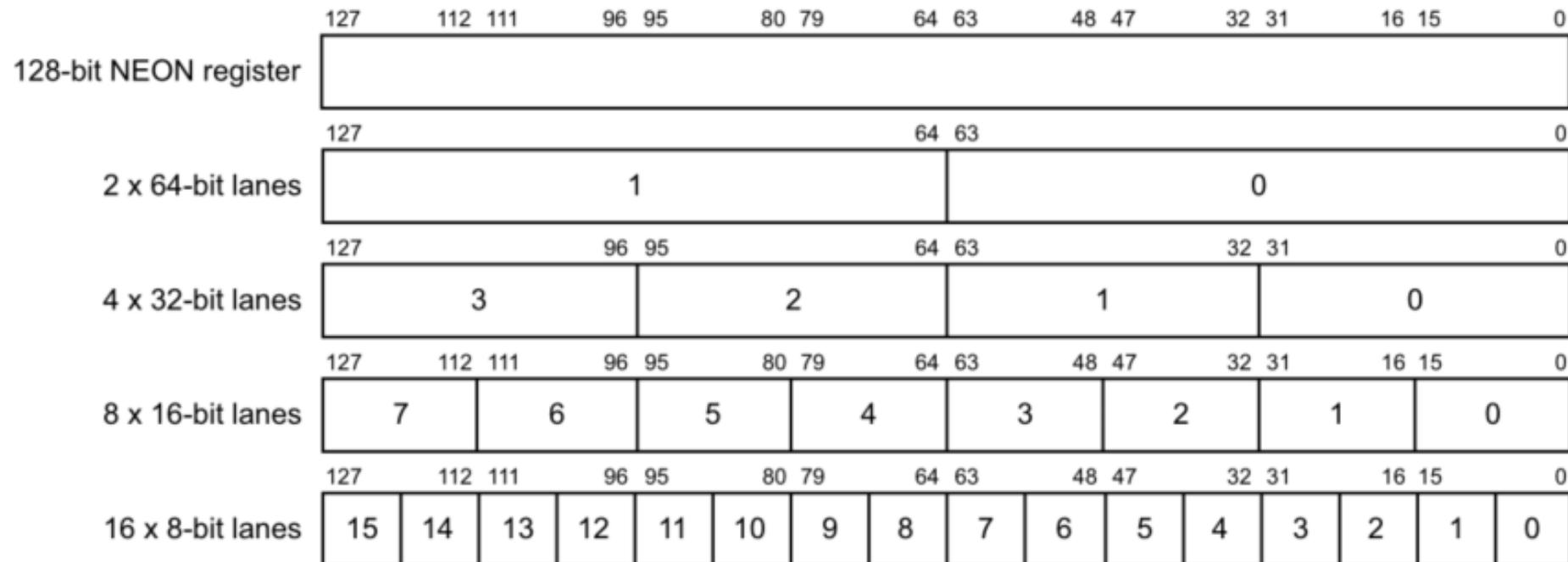
Princípios básicos

- 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 realizados por faixa (*lane*)



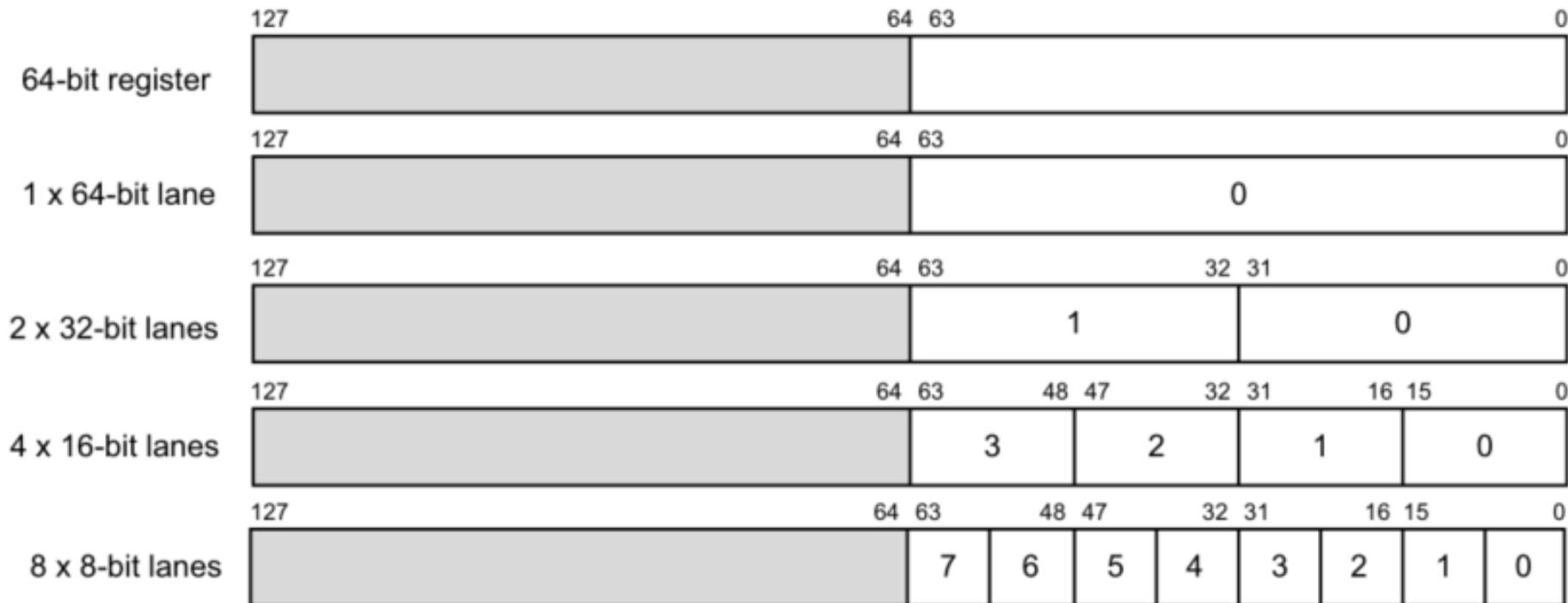
Banco de registo NEON

- 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.

$\langle \text{Instrução} \rangle \ Vd.\text{TS}[i], \ Vn.\text{Ts}[j]$

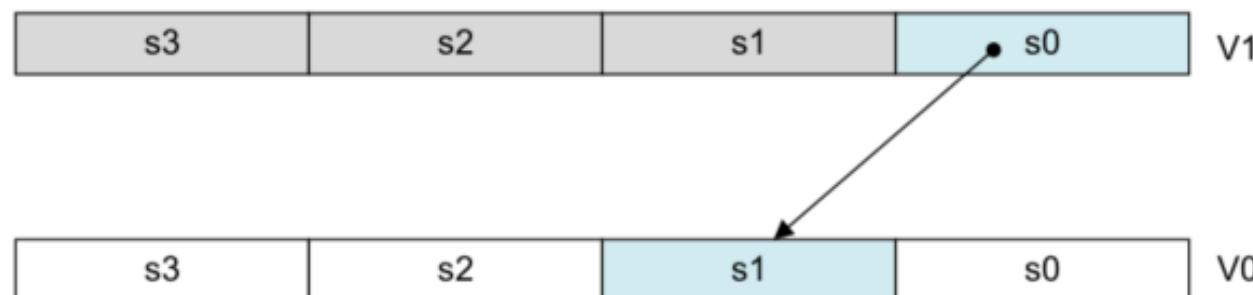
Vd registo de destino

Vn registo de origem

Ts especificador de tipo

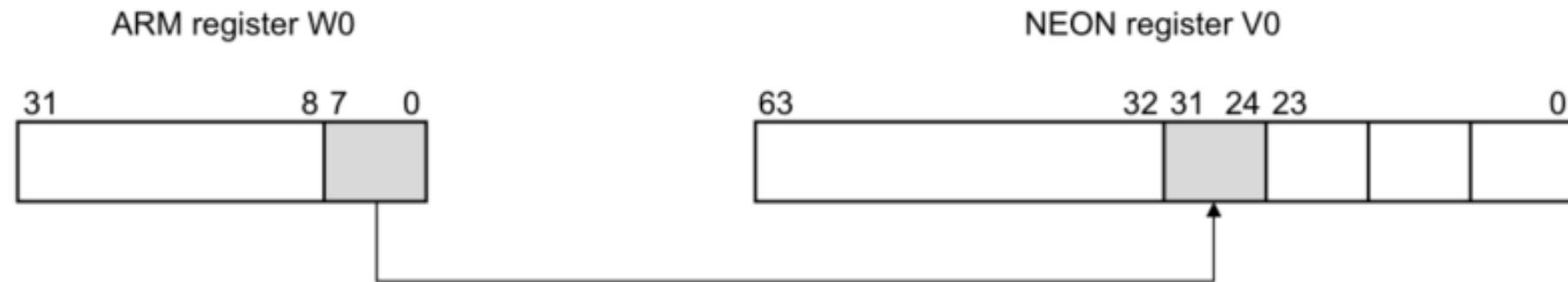
i, j índices dos elementos

- 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

1 Processamento paralelo: introdução

2 Tecnologia NEON

3 Princípios gerais das instruções NEON

4 Principais instruções SIMD

Mnemônicas 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.

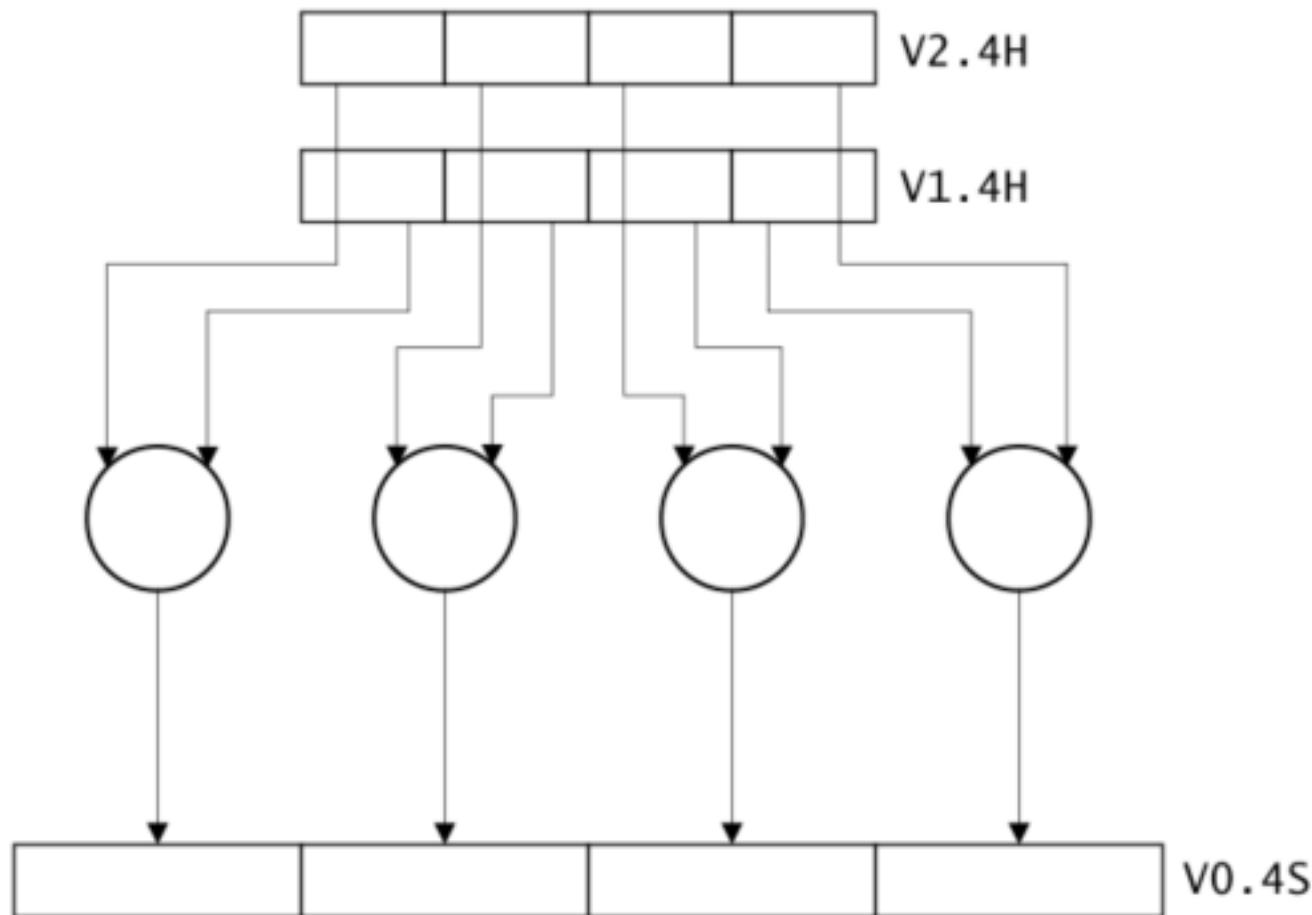
- A variante *Normal* opera sobre qualquer tipo de vetor e produz vetores da mesma dimensão (número de componentes) e (geralmente) do mesmo tipo.

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

Estas instruções usam o sufixo **P**.

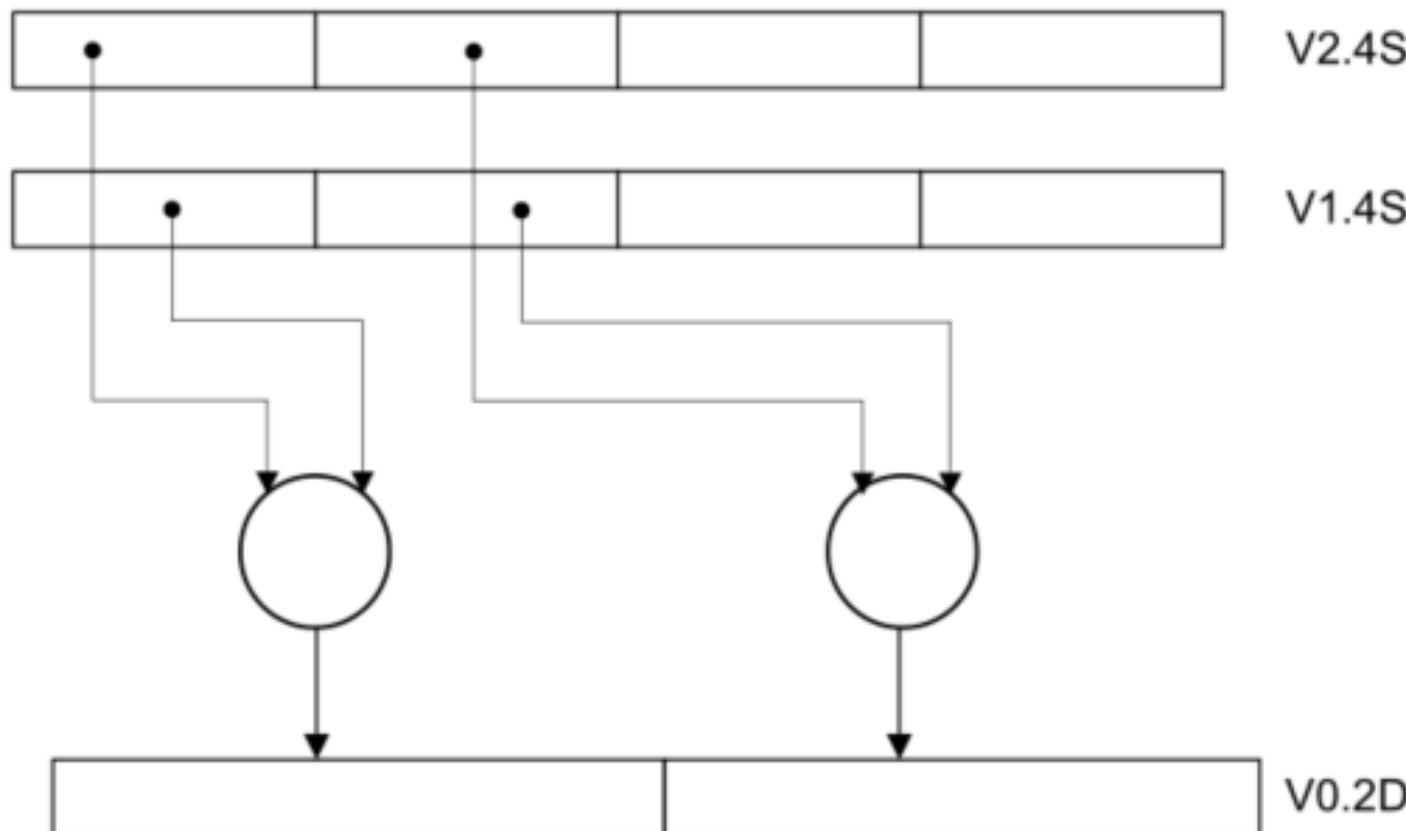
Variante “long”

- 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**



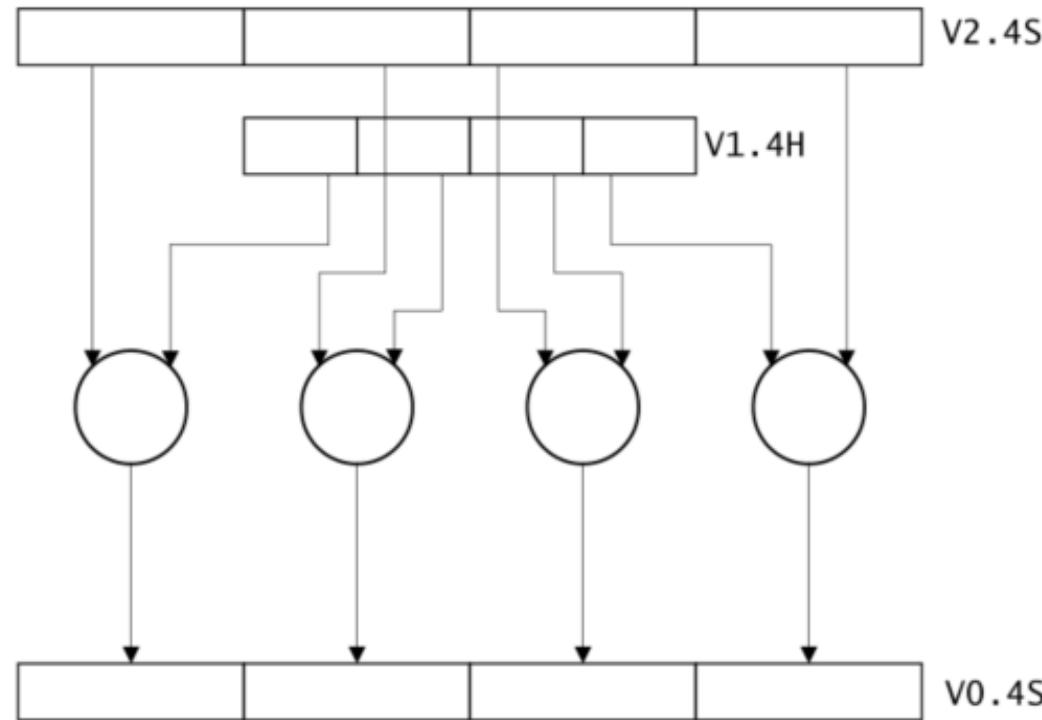
Variante “long” para elementos mais significativos

- 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**



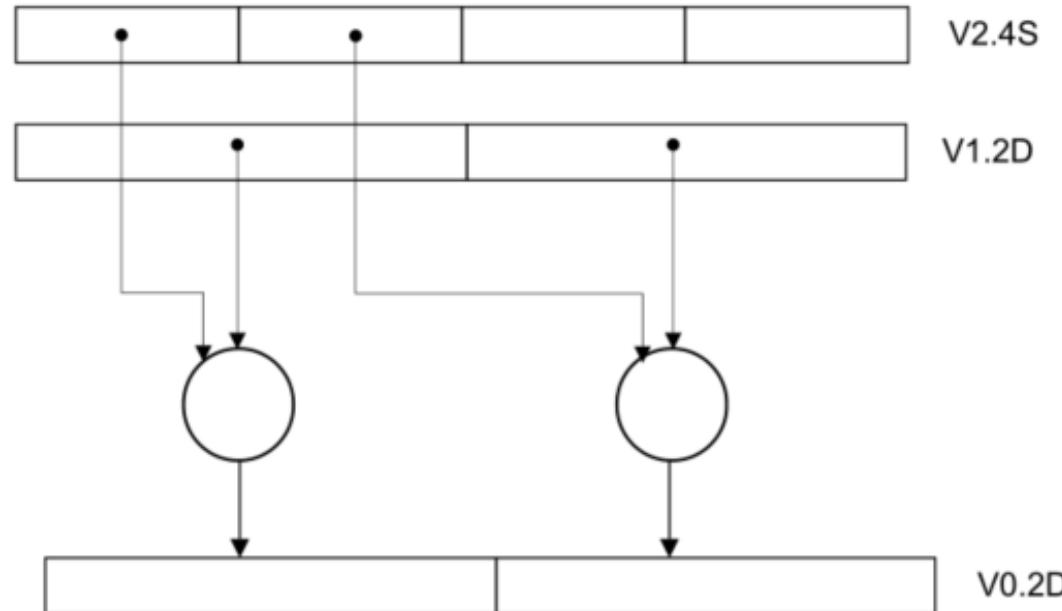
Variante “wide”

- Operam sobre um vetor de 64 bits e 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**



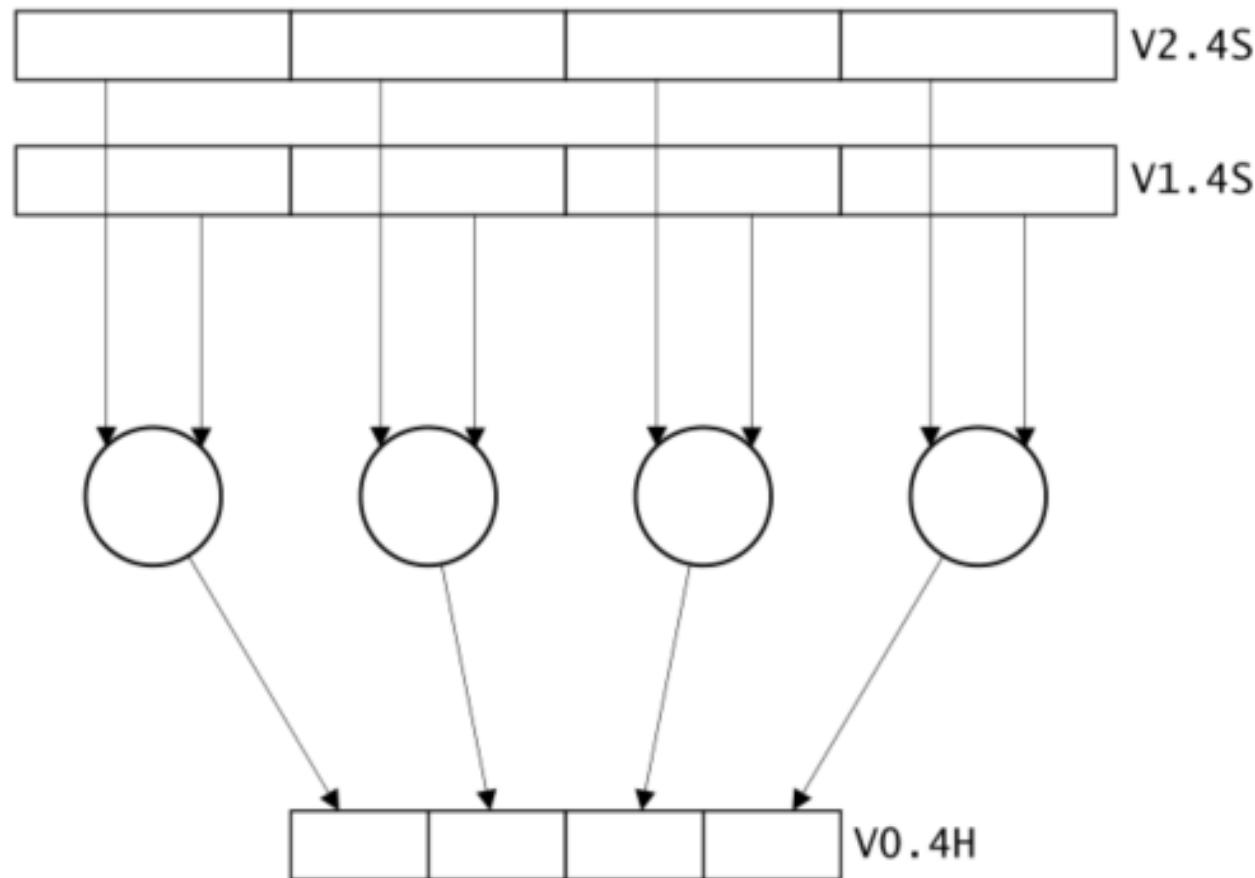
Variante “wide” para pistas superiores

- 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**



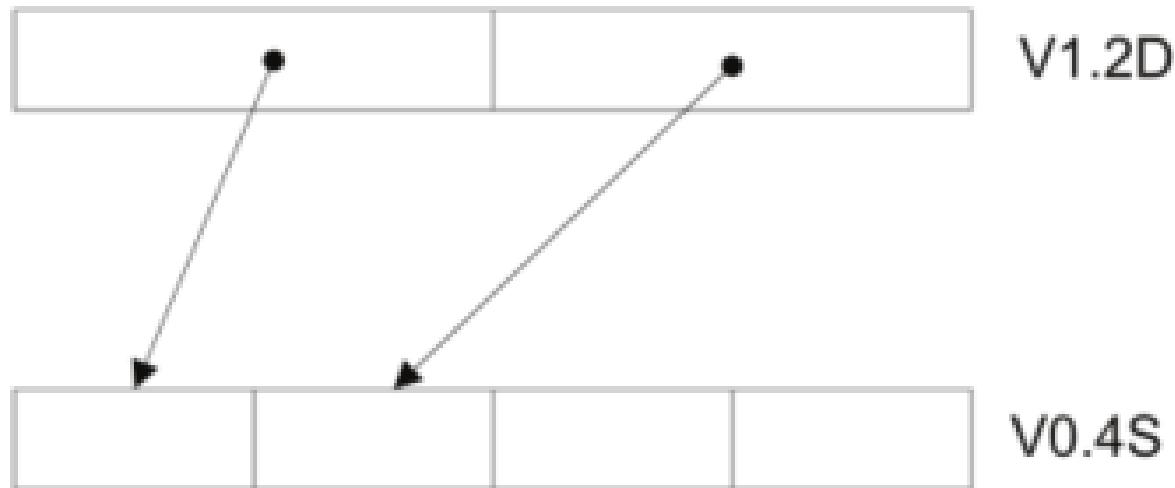
Variante “narrow”

- 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)



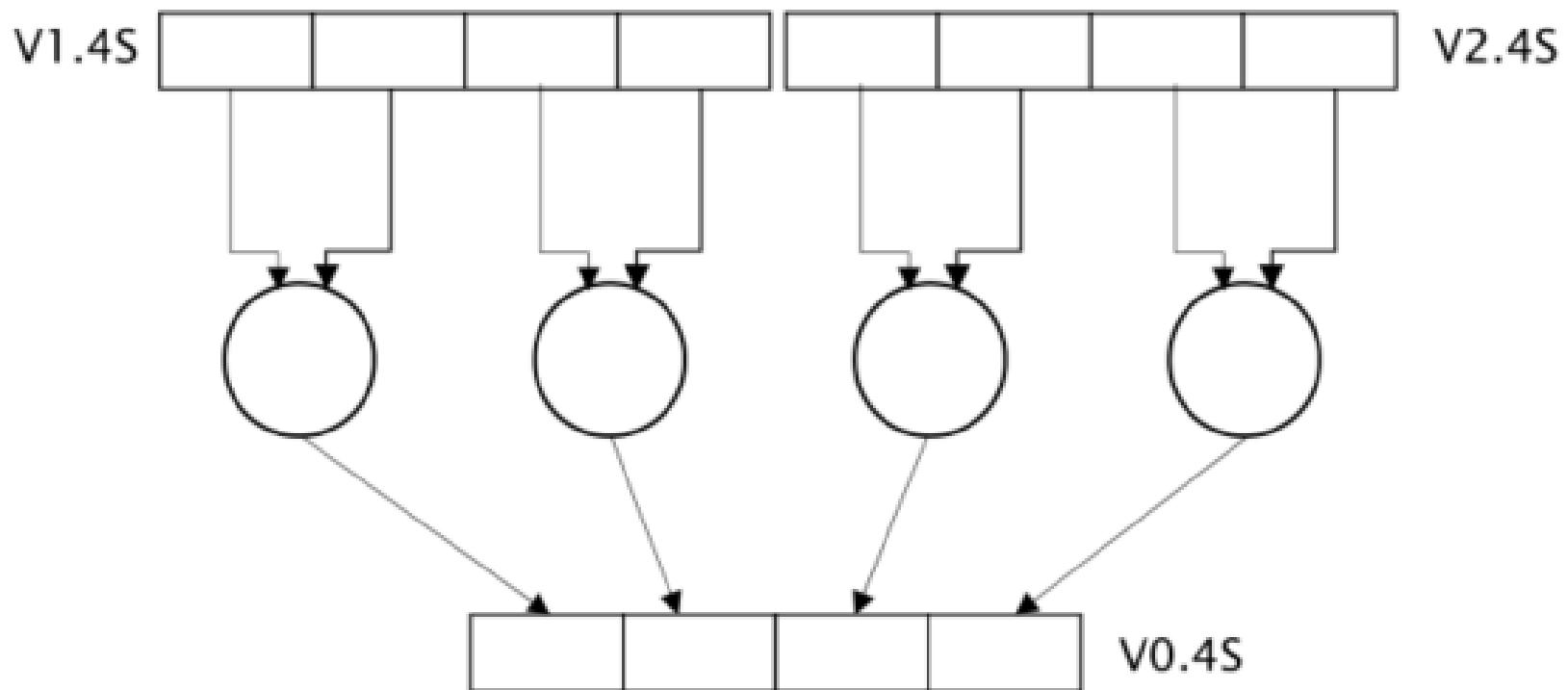
Operações com saturação

- 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)	$-2^7 \leq x < 2^7$
Signed halfword (S16)	$-2^{15} \leq x < 2^{15}$
Signed word (S32)	$-2^{31} \leq x < 2^{31}$
Signed doubleword (S64)	$-2^{63} \leq x < 2^{63}$
Unsigned byte (U8)	$0 \leq x < 2^8$
Unsigned halfword (U16)	$0 \leq x < 2^{16}$
Unsigned word (U32)	$0 \leq x < 2^{32}$
Unsigned doubleword (U64)	$0 \leq x < 2^{64}$

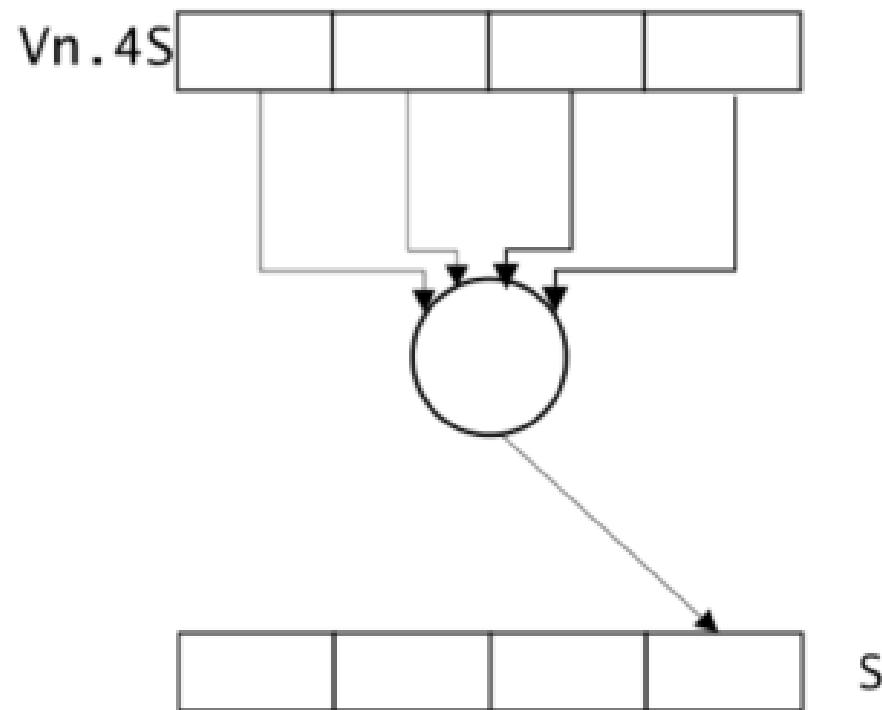
Operações para operações sobre pares de registos

- 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

- 1 Processamento paralelo: introdução
- 2 Tecnologia NEON
- 3 Princípios gerais das instruções NEON
- 4 Principais instruções SIMD

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_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

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

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_smax4:

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

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_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 → 32 bits, extensão com 0)
SMOV X10, V2.S[3] (32 bits → 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 uchar_b[16];

int main(void)
{
    func_replicate_byte(uchar_b, -12);
    print16(uchar_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 -12

Instruções de conversão inteiro ↔ VF

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

UCVT_F H<d>, H<n> meia precisão

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

- Para números com sinal, usar a instrução SCVT_F
- Para conversões no sentido inverso, usar FCVTNU e FCVTNS, respectivamente

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 números de precisão simples
// X1 endereço de vetor de 4 números 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 ($a \cdot \bar{b}$), EOR, NOT, ORN ($a + \bar{b}$), 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];

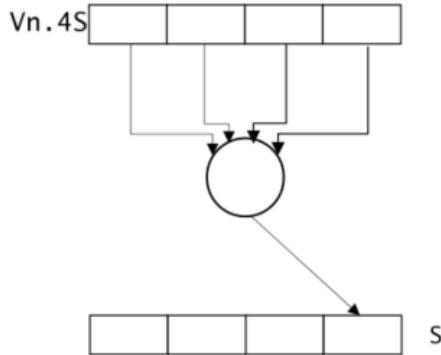
int main(void)
{
    manter_se_maiores(vint_a, vint_b, 10);
    print4(vint_b);
}

// 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_advv16(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_advv16(vchar);
    printf("Soma=%hd\n", r);
}

// X0 endereço de vetor de 16 bytes
// W0 resultado (byte menos significativo)
func_advv16:
    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
```

Introdução aos multiprocessadores

João Canas Ferreira

Abril 2018



Assuntos

- 1 Processamento paralelo
- 2 Multiprocessadores de memória partilhada
- 3 Modelo simplificado de desempenho

Multiprocessadores

- Objetivo: criar computadores potentes pela interligação de computadores mais simples
 - programas devem aproveitar um número variável de processadores
 - utilização de processadores mais simples e eficientes (em vez de complexos e ineficientes)
 - aumento da disponibilidade (robustez em face de falhas)
- execução simultânea de programas independentes: *paralelismo ao nível do processo*
- programa único em execução simultânea em vários processadores: *programa paralelo*
- **cluster:** computador paralelo composto por muitos processadores alojados em computadores independentes
- **processador multinúcleo:** vários processadores (núcleos) num único circuito integrado
- vários processadores (multinúcleo ou não) podem ser reunidos num único computador: multiprocessador

Concorrência e paralelismo

		Software	
		Sequencial	Concorrente
Hardware	“Serial”	Multiplicação de matrizes em Matlab (Pentium 4)	Windows a correr no processador Pentium 4
	Paralelo	Multiplicação de matrizes em Matlab (Xeon e5345)	Windows a correr no processador Xeon e5345

► Grandes desafios:

- Como executar programas sequenciais eficientemente em *hardware paralelo*?
- Como executar programas concorrentes eficientemente à medida que o número de processadores aumenta? (Escalabilidade)

Desafio 1: Maior rapidez de processamento

■ Pretende-se obter um aumento de rapidez (*speed-up*) de 90 usando um computador com 100 processadores. Que percentagem dos cálculos pode ser sequencial?

■ Lei de Amdahl:

f : fator de aumento de rapidez

p : percentagem de tempo que o melhoramento pode ser aplicado:

$$S = \frac{1}{(1 - p) + \frac{p}{f}}$$

■ Fazendo:

$$90 = \frac{1}{(1 - p) + \frac{p}{100}}$$

■ Resultado: $p \approx 0,999$

■ Para obter um aumento de rapidez de 90 usando 100 processadores, apenas 0,1 % do tempo de execução pode ser sequencial.

Desafio 2: Escalabilidade (1)

- Pretende-se somar duas matrizes de dimensão 10×10 e realizar 10 somas com variáveis simples (com dependências). Qual é o aumento de rapidez obtido usando um multiprocessador com 10 ou com 100 elementos? (Seja t o tempo que demora fazer uma multiplicação uma adição).
- Tempo necessário num único processador sequencial: $110 \times t$
- Tempo com multiprocessador 10: $20 \times t \Rightarrow S = 5,5$ (55,5 % do ideal)
- Tempo com multiprocessador 100: $11 \times t \Rightarrow S = 10$ (10 % do ideal)

E se as matrizes forem 100×100 (10000 adições)?

- Tempo necessário num único processador sequencial: $10010 \times t$
- Tempo com multiprocessador 10: $1010 \times t \Rightarrow S = 9,9$ (99 % do ideal)
- Tempo com multiprocessador 100: $110 \times t \Rightarrow S = 91$ (>90 % do ideal)

Desafio 2: Escalabilidade (2)

⇒ No problema anterior, é mais fácil aproveitar os recursos no caso de problemas de maiores dimensões.

⇒ Escalabilidade de um multiprocessador

Escalabilidade forte aumento de rapidez obtido mantendo a dimensão do problema.

Escalabilidade fraca aumento de rapidez obtido aumentando o problema de forma proporcional ao número de processadores.

⇒ Seja M o espaço de memória e P o número de processadores:

- escalabilidade forte: memória por processador M/P (decresce)
- escalabilidade fraca: memória por processador M (mantém-se)

⇒ O tipo de escalabilidade relevante depende do problema.

Desafio 3: Equilíbrio

➡ Até agora, assumiu-se que os cálculos podiam ser distribuídos de forma equilibrada por todos os processadores. Para 100 processadores, cada um executa 1 % dos cálculos.

O que sucede quando um dos processadores tratar de 2 % dos cálculos (no caso de matrizes 100×100)

➡ $2\% \times 10000 = 200$ adições são feitas por um processador; os restantes 99 repartem 9800 adições.

➡ Tempo com multiprocessador 100:

$$\max\left(\frac{9800 \times t}{99}, \frac{200 \times t}{1}\right) + 10t = 210t \quad \Rightarrow S = 48$$

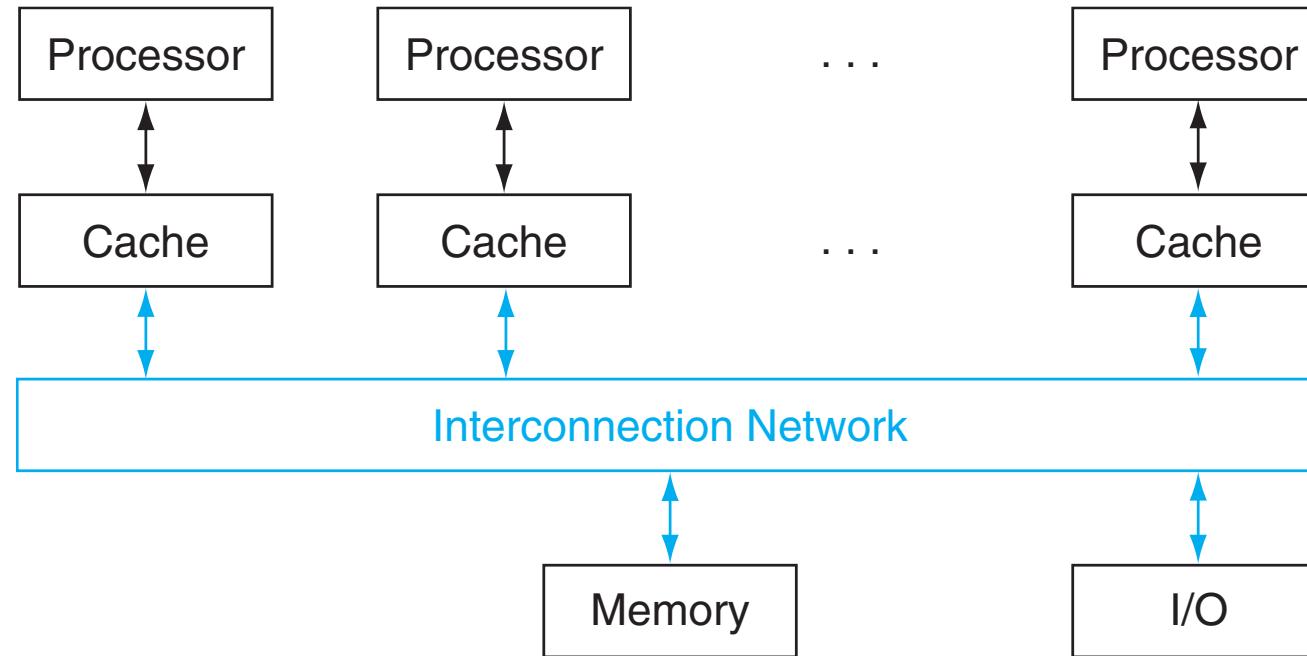
Cerca de metade da situação equilibrada!

1 Processamento paralelo

2 Multiprocessadores de memória partilhada

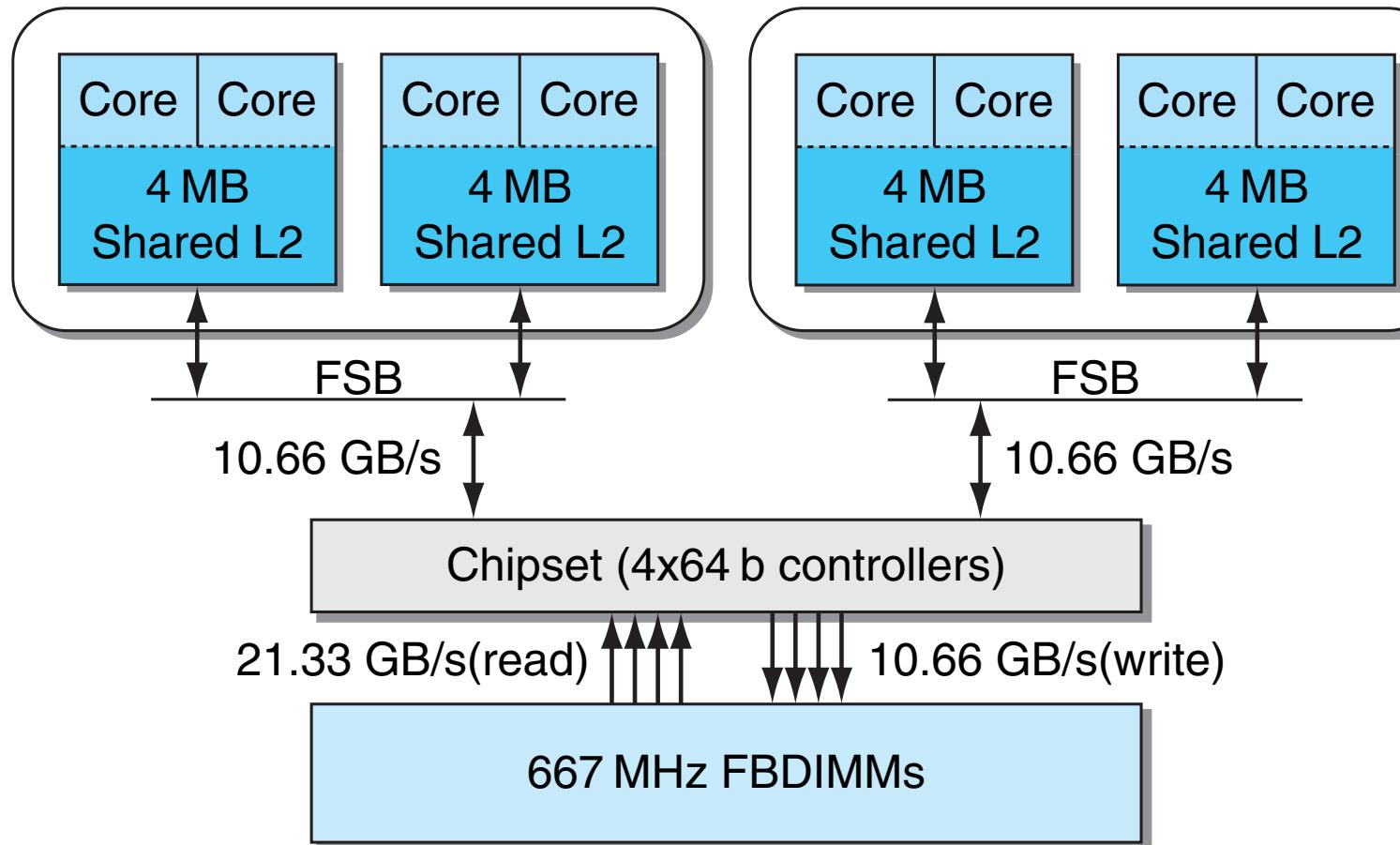
3 Modelo simplificado de desempenho

Organização



Fonte: [COD4]

Exemplo: Xeon e5345



Fonte: [COD4]

Características

- Espaço físico de endereços **único**
- Processadores comunicam através de variáveis partilhadas em memória: todos os processadores podem aceder a qualquer posição de memória via instruções *load/store*
- Dois tipos de organização:
 - o tempo de acesso a memória é idêntico para todos os processadores e todas as posições de memória: UMA (*uniform memory access*)
 - alguns acessos a memória são mais rápidos que outros (dependendo do processador e da posição de memória acedida): NUMA (*non-uniform memory access*)
- Coordenação no acesso aos dados partilhados por código em execução em processadores diferentes: **sincronização**.

Programa para memória partilhada (1)

- ➡ Tarefa: Somar 100000 num multiprocessador NUMA com 100 processadores.
- ➡ Seja P_n o identificador do processador ($0 \leq P_n \leq 99$)
- ➡ Todos os processadores começam por executar o seguinte ciclo, que processa uma parte dos dados ($100000/100 = 1000$ números)

```
sum[Pn] = 0;  
  
for (i = 1000*Pn; i < 1000*(Pn+1) ; i=i+1)  
    sum[Pn] = sum[Pn] + A[i];
```

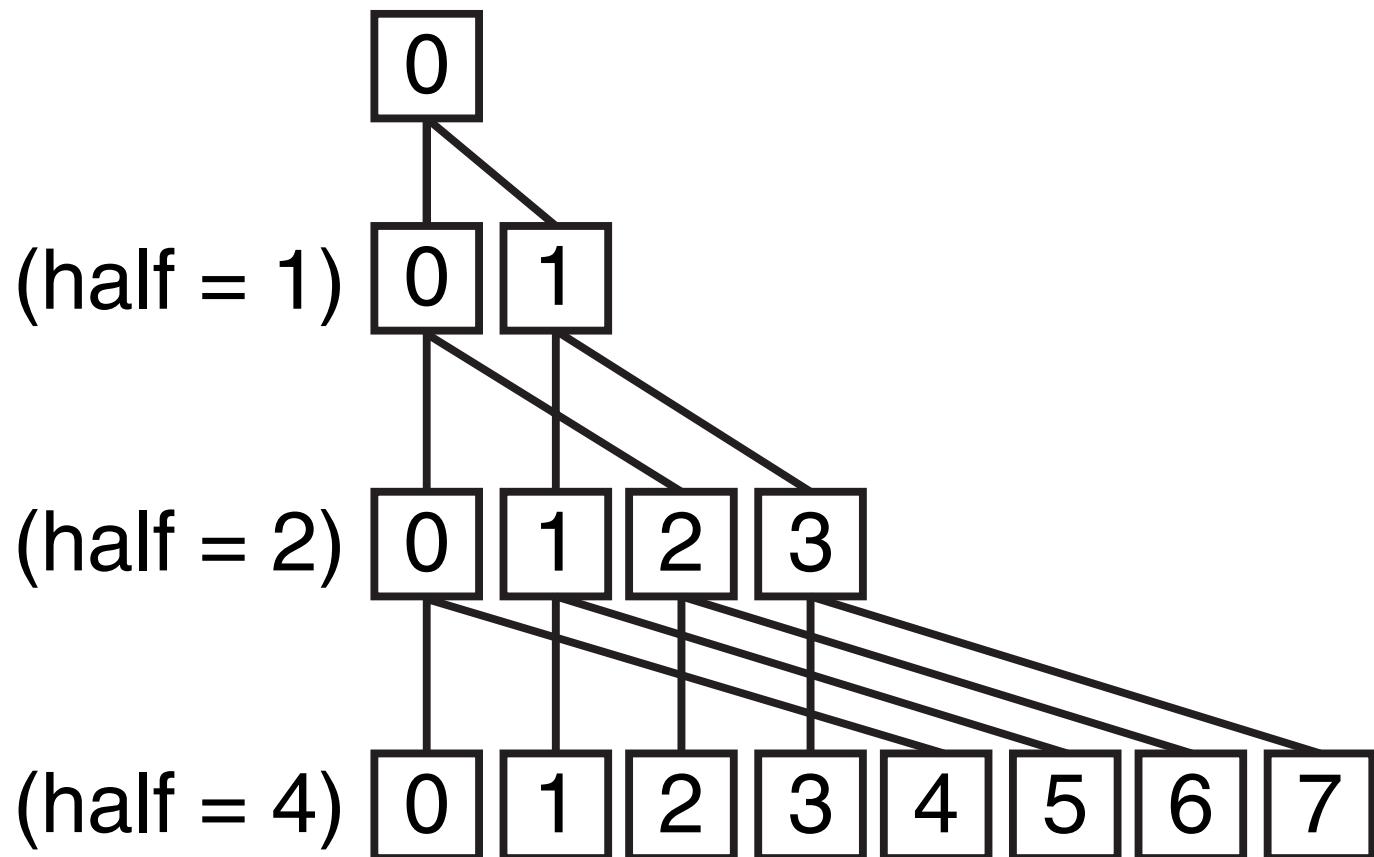
Programa para memória partilhada (2)

⇒ O segundo passo faz a adição das somas parciais (redução)

- 1/2 dos processadores somam pares de números
- 1/4 dos processadores somam as novas somas parciais
- 1/8 dos processadores somam as novas somas parciais, etc.

```
half = 100;  
repeat  
    synch(); // esperar pela conclusão das somas parciais  
    if (half % 2 != 0 && Pn == 0) // half é ímpar  
        sum[0] = sum[0] + sum[half-1];  
    half = half / 2;  
    if (Pn < half)  
        sum[Pn] = sum[Pn] + sum[Pn+half];  
until (half == 1);
```

Programa para memória partilhada (3)



Fonte: [COD4]

- Os quatro últimos níveis da operação de *redução* que soma os resultados de cada processador (da base para o topo).

1 Processamento paralelo

2 Multiprocessadores de memória partilhada

3 Modelo simplificado de desempenho

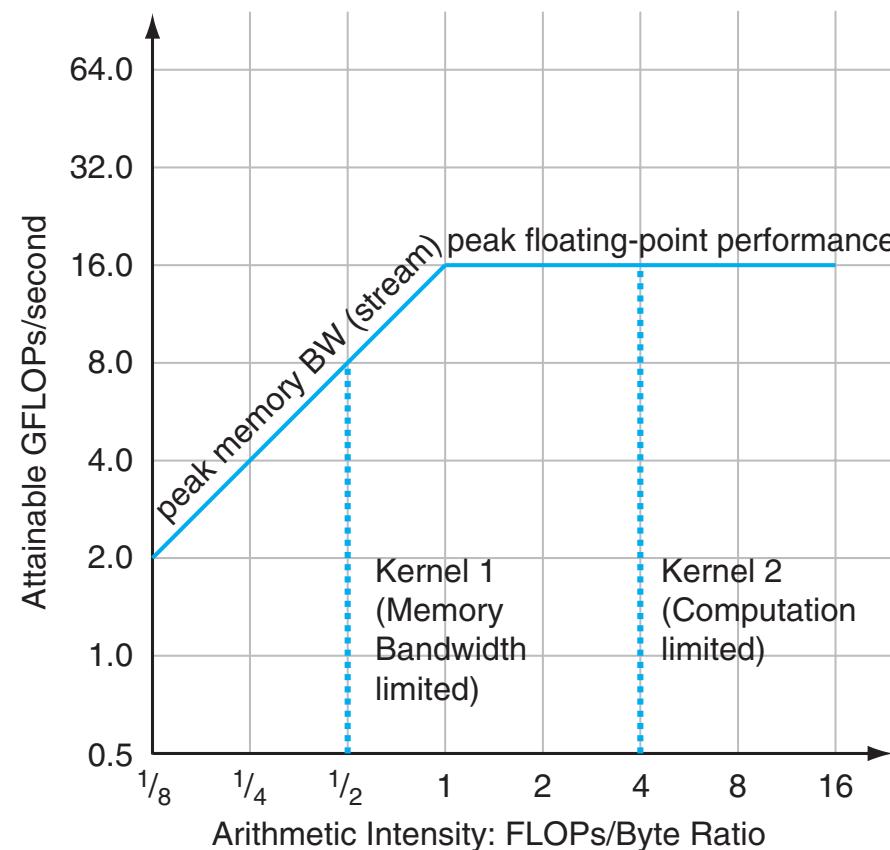
Simplificações

- Domínio: cálculo científico
- Dados em vírgula flutuante (VF): FLOP (*floating-point op per second*)
- Problemas de grandes dimensões compostos apenas por cálculos VF e transferências de dados
- **Intensidade aritmética:**
Operações vírgula flutuante / bytes transferidos de memória
- Impacto sobre o sistema de memória:

$$\frac{\text{Operações vírgula flutuante / segundo}}{\text{Operações de vírgula flutuante/byte}} = \text{bytes/segundo}$$

- bytes/segundo: taxa de transferência de dados que o sistema de memória deve suportar (*memory bandwidth*)
- Existe um valor máximo para o desempenho (valor de pico)

Modelo simplificado

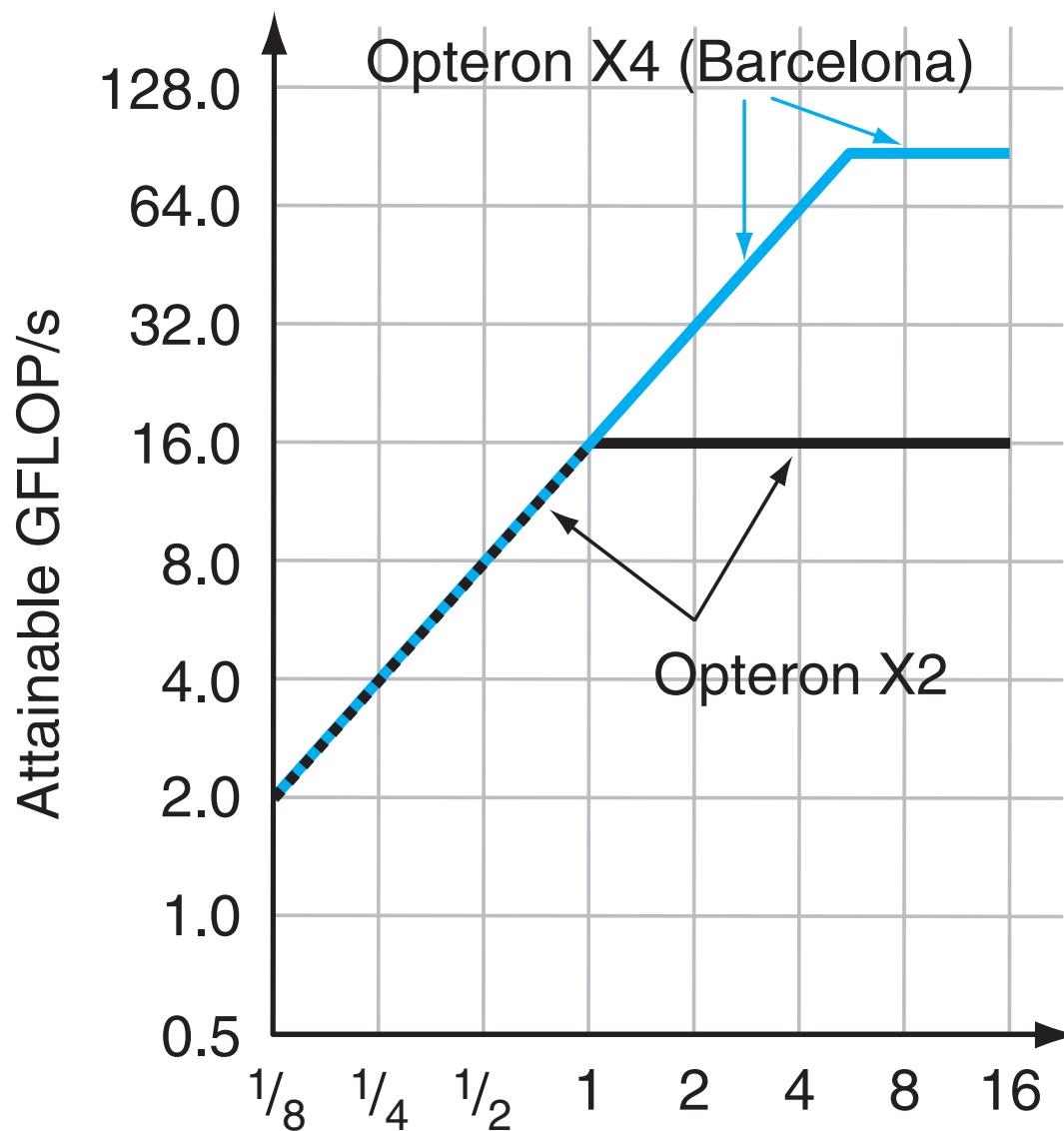


Fonte: [COD4]

➡ Desempenho é o valor mínimo entre:

- taxa de transferência de dados × intensidade aritmética
- valor máximo do desempenho assumindo que todas as unidades de cálculo estão permanentemente ocupadas (desempenho de pico)

Comparação de multiprocessadores



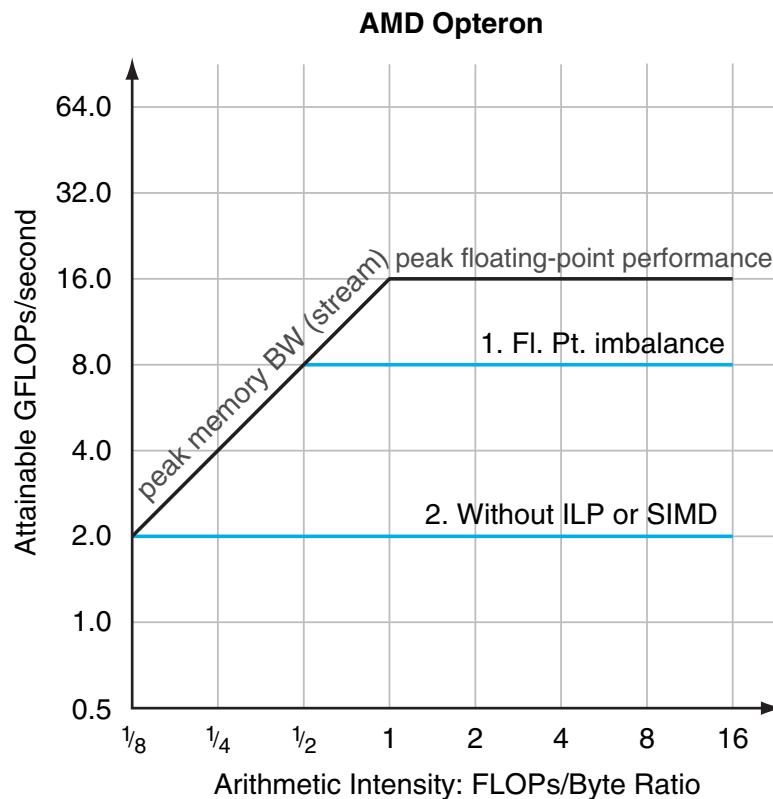
Fonte: [COD4]

Otimizações úteis

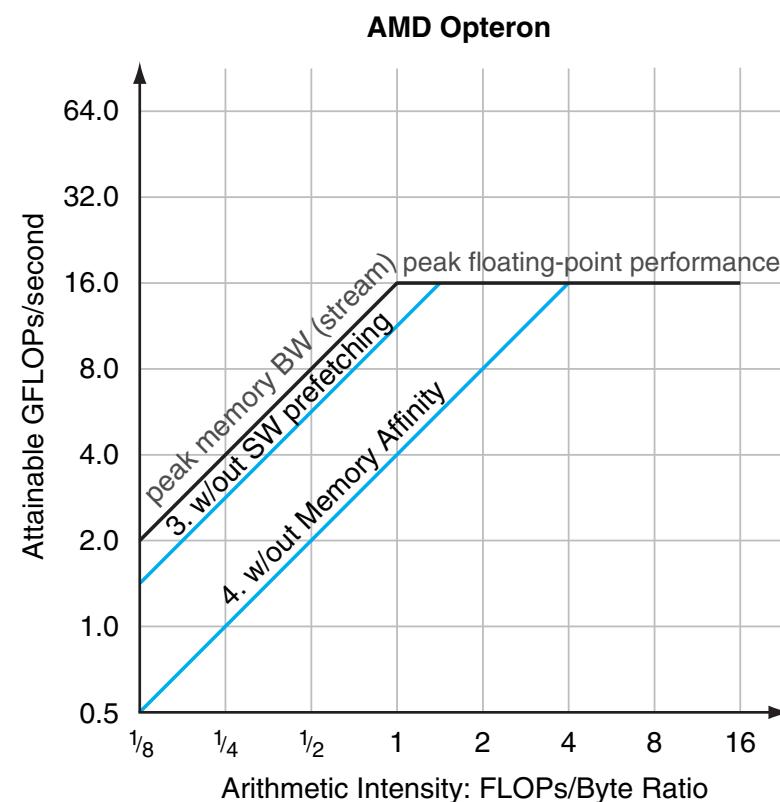
- ➡ As seguintes otimizações (realizadas pelo compilador) permitem frequentemente aumentar o desempenho.
- ➡ Computação:
 - ① Otimizar combinações de operações de vírgula flutuante
Tipicamente, CPUs requerem um número quase igual de somas e multiplicações VF simultâneas para atingir o desempenho de pico
 - ② Execução simultânea de instruções (*ILP=instruction level parallelism*)
O aproveitamento eficaz desta possibilidade também depende da organização do código
- ➡ Acesso a memória:
 - ① Usar instruções especiais para pré-carregar dados de memória (antes de serem necessários): *software prefetching*
 - ② Aproveitar afinidade de memória: colocar os dados em locais de memória com acesso mais eficiente para o processador que os trata.

Modelo com indicação de otimizações

Otimizações de cálculo



Otimizações de memória



Fonte: [COD4]

Referências

COD4 D. A. Patterson & J. L. Hennessy, Computer Organization and Design, 4 ed.

Os tópicos tratados nesta apresentação são abordados nas seguintes secções de [COD4]:

- 7.1–7.3, 7.10