Introdução à arquitetura AArch64

João Canas Ferreira

Março 2020

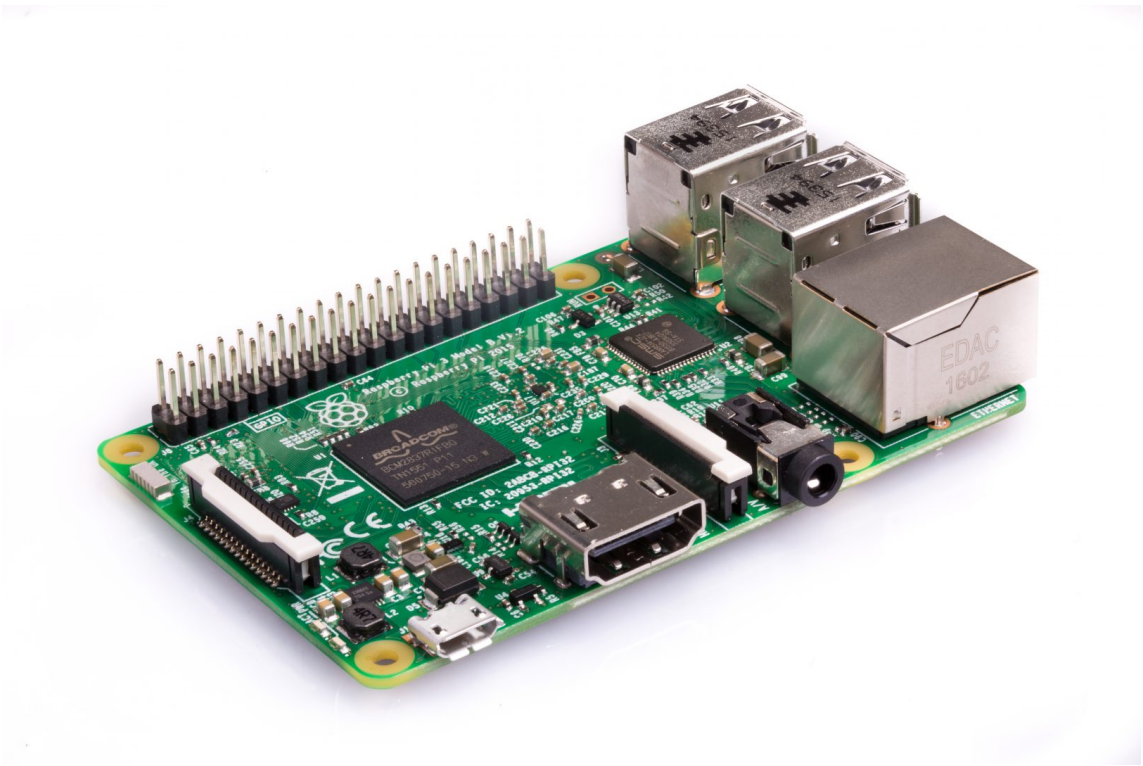
U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Assuntos

As arquiteturas ARM

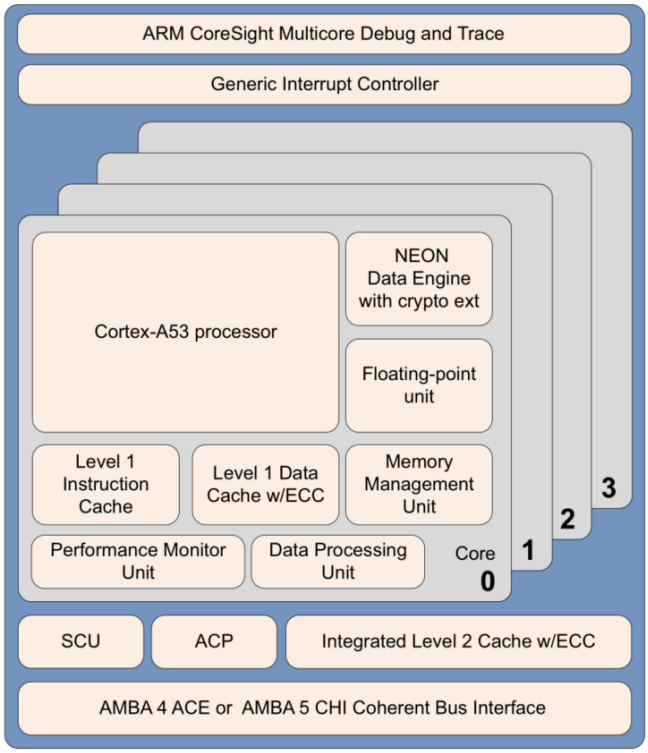
- A designação ARM cobre uma família de arquiteturas RISC.
- Arquiteturas predominantes em *smartphones*, *tablets* e dispositivos de IoT (Internet of Things)
- A companhia ARM licencia a sua propriedade intelectual a outras companhias (Samsung, NXP, etc.).
  O Raspberry PI 3 usa um circuito BCM2837 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.

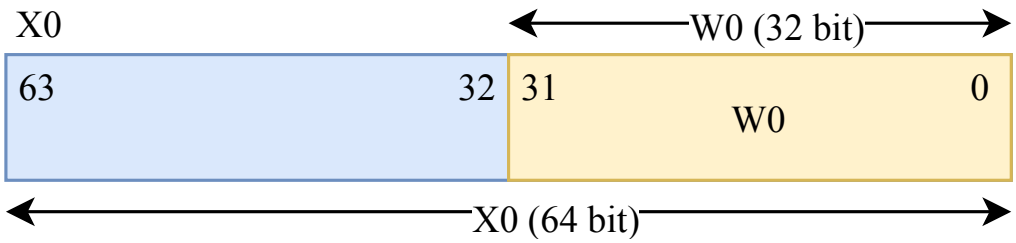
  A — Application Suporta memória virtual via uma MMU (*Memory Management Unit*); usada com sistemas operativos como Linux, iOS, Android.

  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→negativo)
Z	Zero	Z=1 se valor igual a 0
C	Carry	Fica a 1 se a operação faz *overflow* (sem sinal)
V	Overflow	Fica a 1 se a operação faz *overflow* (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

Instruções de processamento de dados

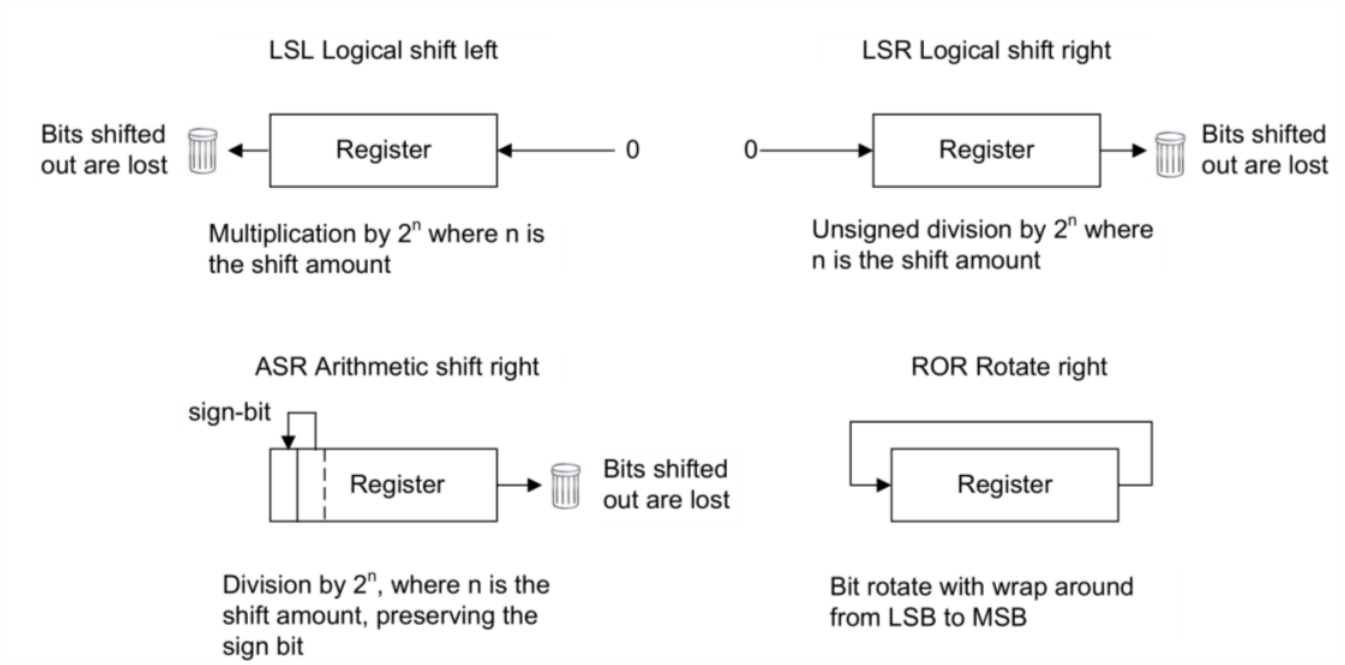
- Estas instruções têm o formato: **Instrução Rd, Rn, Op2**
  - Rd Registo de destino (X0,X1,...,X30 ou W0,W1,...,W30))
  - Rn Fonte de um dos valores usados na operação
  - Op2 Fonte do outro valor usada na operação.
- Op2 pode ser um registo, um registo *modificado* ou um valor imediato.

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



Controlo de fluxo

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

Instrução	Efeito
B etiqueta	salto para PC ± 128 MiB
BL etiqueta	como B, mas guarda endereço de retorno em X30
B.<cond> etiqueta	salto condicional para PC ± 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 ± 1 MiB
CBNZ Reg, etiqueta	se Reg=1, salto para PC ± 1 MiB
TBZ Reg, bit, etiqueta	se Reg[bit]=0, salto para PC ± 32 KiB
TBNZ Reg, bit, etiqueta	se Reg[bit]=1, salto para PC ± 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**

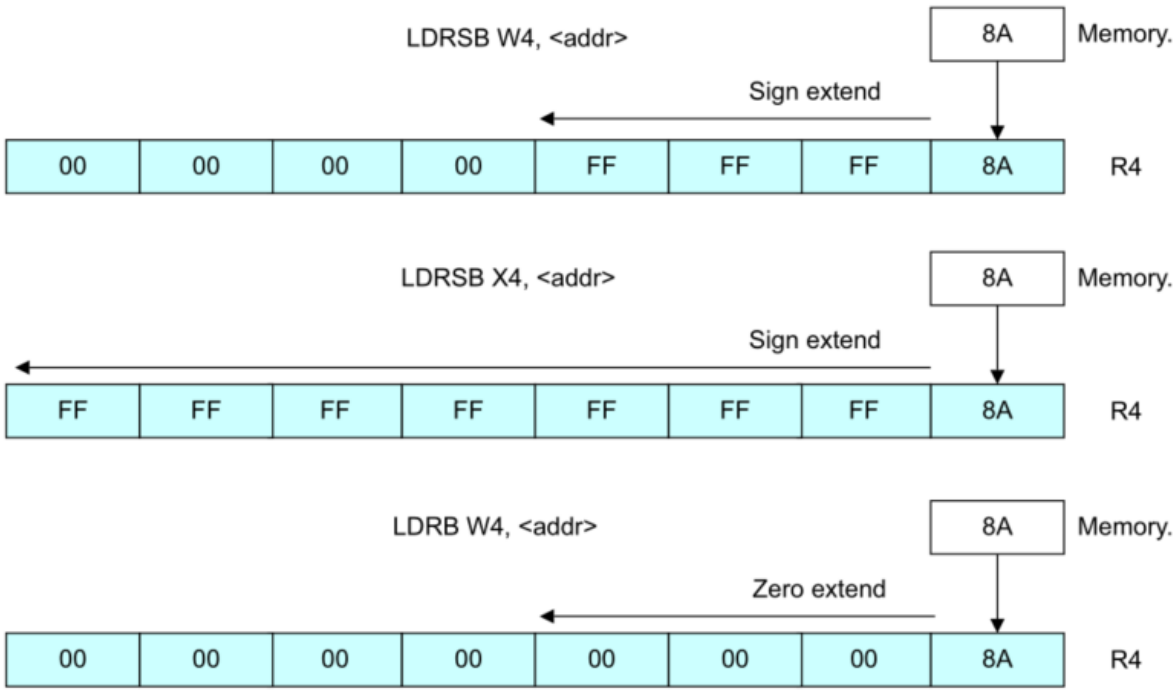
Instrução	Tamanho	Extensão	Registo
LDRB	8 bit	com 0	W<n>
LDRSB	8 bit	de sinal	W<n> ou X<n>
LDRH	16 bit	com 0	W<n>
LDRSH	16 bit	de sinal	W<n> ou X<n>
LDRSW	32 bit	de sinal	X<n>

- 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



Exemplos de leitura

Número: 690528571926 = 0xA0C6B5D216 **X0: 10200**

	MEM
10208	...
10207	**00**
10206	**00**
10205	**00**
10204	**A0**
10203	**C6**
10202	**B5**
10201	**D2**
10200	**16**

endereço (base 10) conteúdo (base 16)

LDR X2, [X0]

00	00	00	A0	C6	B5	D2	16	X2

LDR W2, [X0]

00	00	00	00	C6	B5	D2	16	X2

LDRSW X2, [X0]

FF	FF	FF	FF	C6	B5	D2	16	X2

LDRSH W2, [X0]

00	00	00	00	FF	FF	D2	16	X2

LDRSH X2, [X0]

FF	FF	FF	FF	FF	FF	D2	16	X2

LDRH W2, [X0]

00	00	00	00	00	00	D2	16	X2

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

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Assuntos

Registos 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 e interoperabilidade

- 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	64
	long long int	
	apontador	
quadword	—	128

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

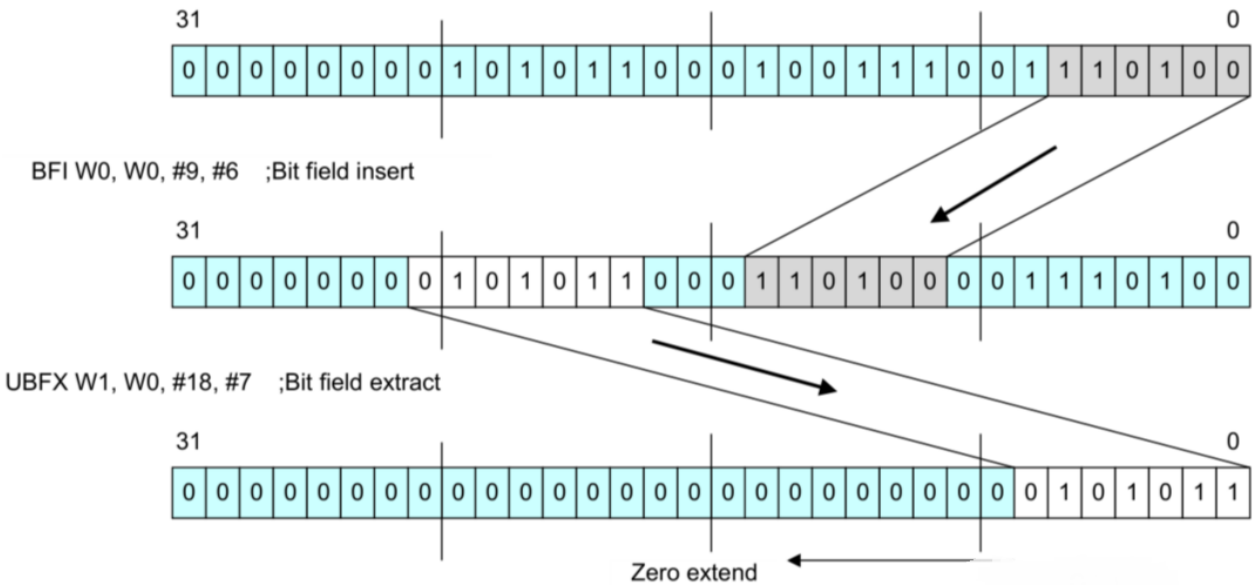
Assuntos

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)



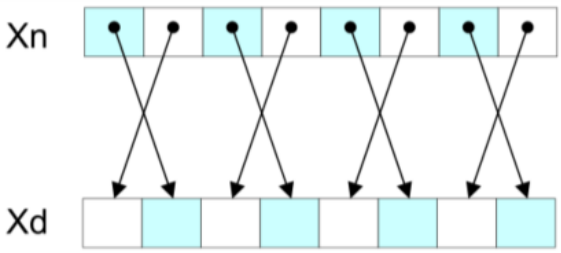
Extensão de operandos

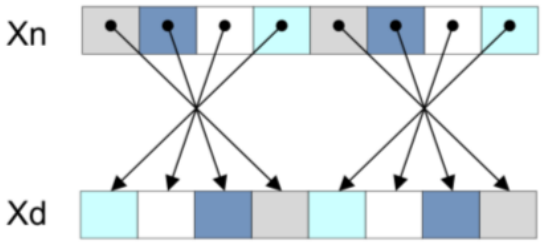
- Existem instruções para alargar o valor de (parte de) um registo W<n> para o registo inteiro W<n> ou X<n>.
- Estas instruções são "sinónimos" das instruções de manipulação de bits apropriadas.

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 *halfword*
REV32		Troca a ordem em dos bytes em cada *word* (apenas registos X<n>)





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

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

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←Rn × Rm; ignora *overflow*
SMULL	`SMULL Xd,Wn,Wm`	Xd←Wn × Wm (operandos com sinal)
UMULL	`UMULL Xd,Wn,Wm`	Xd←Wn × Wm (operandos sem sinal)
SMULH	`SMULH Xd,Xn,Xm`	Xd←(Xn×Xm)<127:64>, com sinal
UMULH	`UMULH Xd,Xn,Xm`	Xd←(Xn×Xm)<127:64>, sem sinal
MADD	`MADD Rd,Rn,Rm,Ra`	Xd←Rd + (Rn×Rm); ignora *overflow*
SMADDL	`SMADDL Xd,Wn,Wm,Xa`	Xd←Ra + (Rn×Rm); operandos com sinal
UMADDL	`UMADDL Xd,Wn,Wm,Xa`	Xd←Ra + (Rn×Rm); operandos 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←Rn ÷Rm; com sinal
UDIV	`UDIV Rd,Rn,Rm`	Rd←Rn ÷Rm; sem sinal

Assuntos

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 mais 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×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 ←endereço efetivo
  [Xb, #imm]!

  Ex: `ldr X1,[X2,40]!` $\Longrightarrow$ `X2 = X2+20; ldr X1,[X2]`

- **pós-indexado**: usar endereço efetivo [Xb] para acesso e atualizar o registo base Xb ←Xb + #imm
  [base], #imm

  Ex: `ldr X1,[X2],40` $\Longrightarrow$ `ldr X1,[X2]; X2=X2+40`

➠ 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/(tamanho do item); o valor deve caber em 12 bits (sem sinal)
  - ➠ `LDR W1,[X2,20]`: valor imediato codificado: 20 ÷4 = 5
  - ➠ `LDR X2,[X2,24]`: valor imediato codificado: 24 ÷8 = 3
  - ➠ `LDRSH W1,[X2,20]`: valor imediato codificado: 20 ÷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.