Arquitetura AArch64

Parte 2

João Canas Ferreira

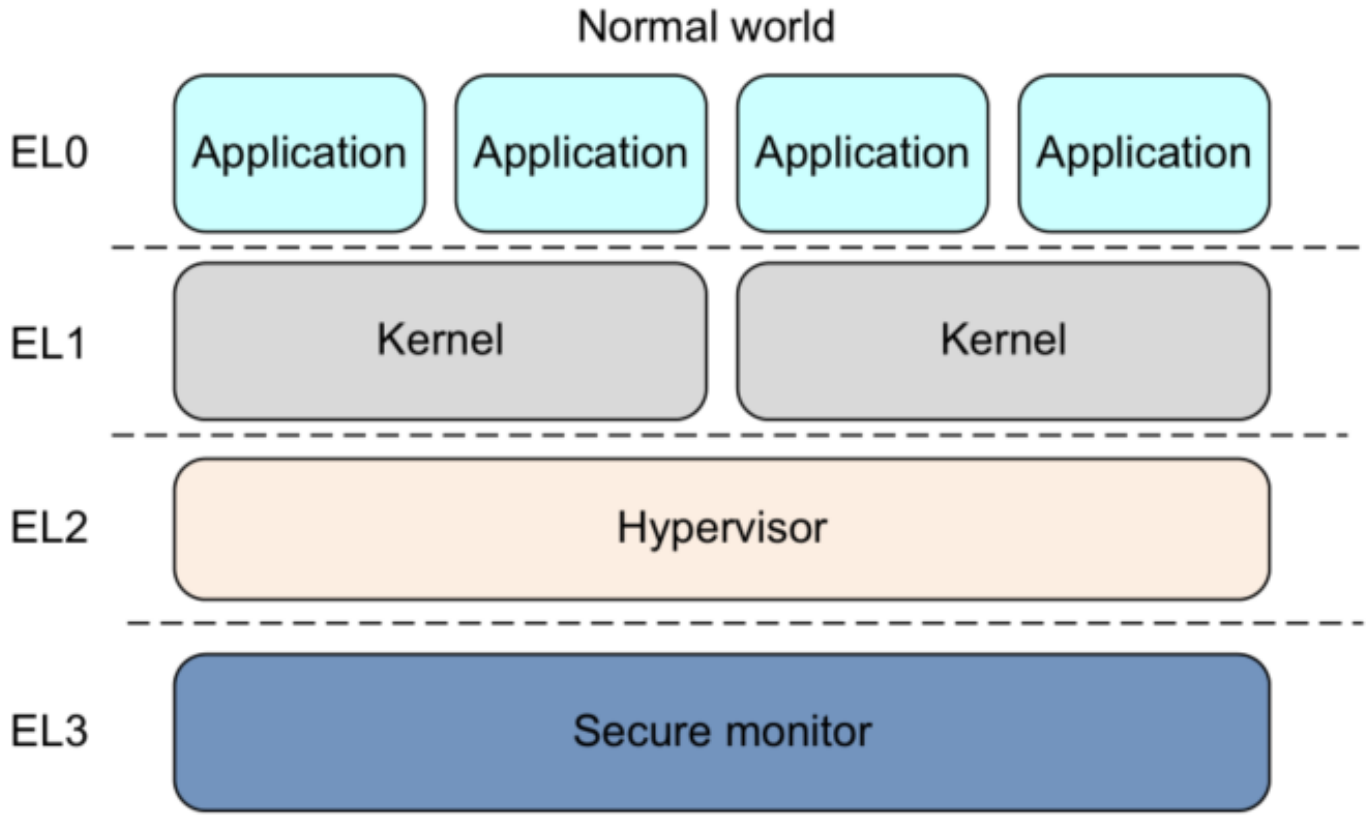
Março 2019

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Assuntos

Níveis de execução

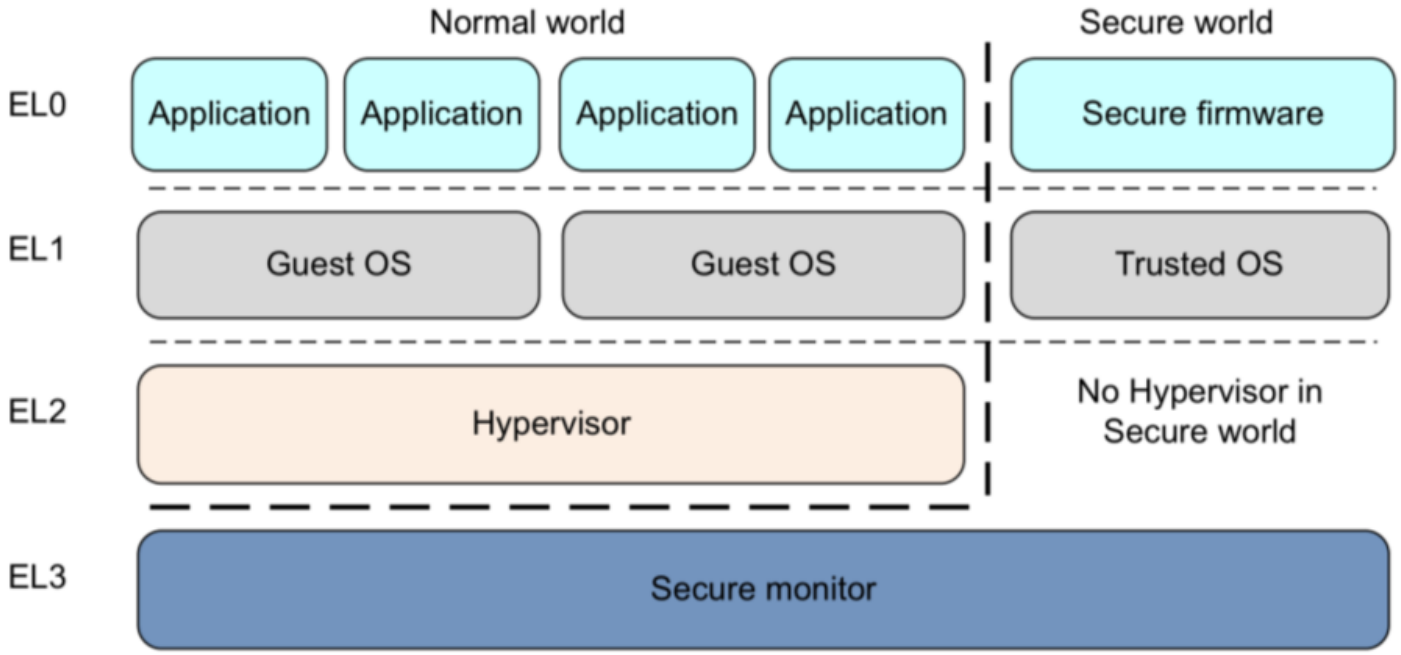
- Na arquitetura ARMv8, a execução de código ocorre num de quatro *níveis de exceção* (EL0–EL3), com o correspondente nível de privilégio.



- *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.

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

- 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.

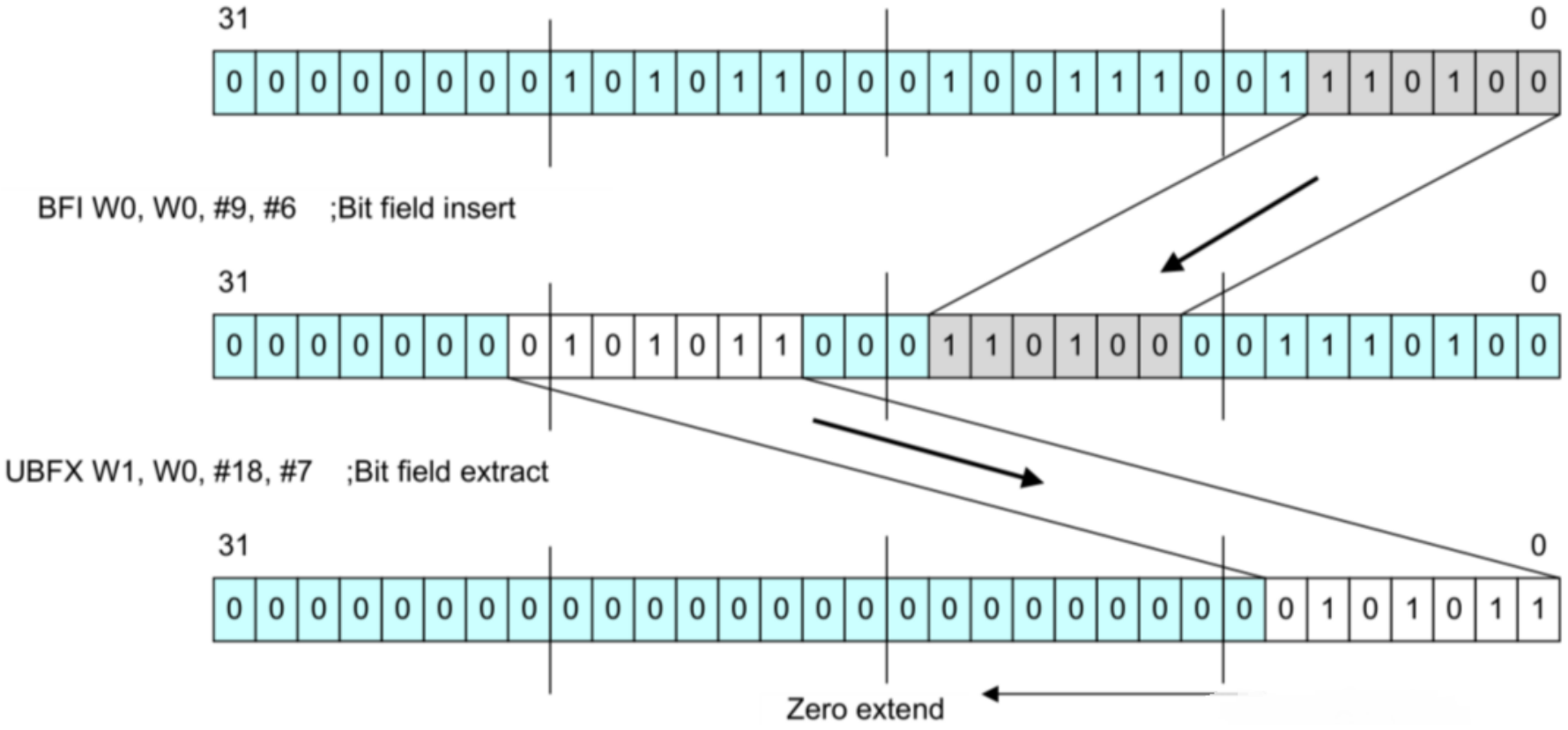
Assuntos

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)FIZ	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)



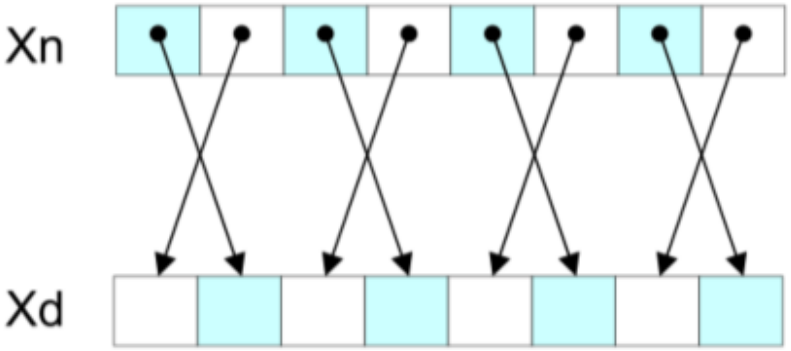
Extensão de operandos

- 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.

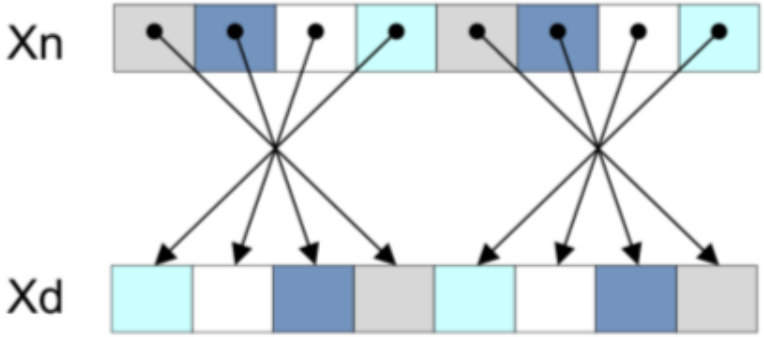
Nome	Exemplo	Descrição
SXTB	`SXTB X0,W1`	Extensão de sinal (S) do byte (B) menos significativo de `W1` para `X0`
SXTH	`SXTH X0,W1`	Extensão de sinal (S) da halfword (H) menos significativa de `W1` para `X0`
SXTW	`SXTW X0,W1`	Extensão de sinal (S) da word de `W1` para `X0` (2º operando é sempre W<n>)
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 direita para a esquerda)
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>)



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

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×Xn)<127:64>, com sinal
UMULH	`UMULH Xd,Xn,Xm`	Xd←(Xn×Xn)<127:64>, sem sinal
MADD	`MADD Rd,Rn,Rm,Ra`	Xd←Ra + (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 três variantes
  (sendo s=1,2,4,8 o tamanho de um item em bytes):
  1. **[Xb, #imm]**: endereço efetivo = Xb + #imm ×s
  2. **[Xb, Xm, {LSL #imm}]**:
     endereço efetivo = Xb + Xm $\{\times 2^{\#imm}\}$ ×s
  3. **[Xb, Wm,(S|U)XTW {#imm}]**:
     endereço efetivo = Xb + ((extended) Wm) $\{\times 2^{\#imm}\}$ ×s
- **Deslocamento constante:** 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]!
- **pós-indexado**: usar endereço efetivo [Xb] para acesso e atualizar o registo base Xb ←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:** #imm tem 12 bits; no cálculo do endereço efetivo, o valor de #imm *é multiplicado pelo tamanho do item*.
  - `LDR W1,[X2,#20]`: endereço efetivo = X2 + 20 ×4 s=4
  - `LDR X2,[X2,#20]`: endereço efetivo = X2 + 20 ×8 s=8
  - `LDRSH W1,[X2,#20]`: endereço efetivo = X2+ 20 ×2 s=2
- **unscaled:** #imm tem 9 bits; no cálculo do endereço efetivo, o valor de #imm é usado sem modificações (s=1).

  As instruções têm a letra "U" antes do R final: LDUR, STUR, etc.
  - `LDUR W1,[X2,#20]`: endereço efetivo = X2 + 20
  - `LDUR X2,[X2,#20]`: endereço efetivo = X2 + 20
  - `LDURSH W1,[X2,#20]`: endereço efetivo = X2 + 20