Conjuntos de instruções de microprocessadores

Arquitetura ARMv8 – AArch64

João Canas Ferreira

Novembro de 2019

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Tópicos

1. Arquitetura do conjunto de instruções
2. Conjunto de instruções ARMv8 (AArch64)
3. Programação em Assembly ARMv8
4. Definição e utilização de sub-rotinas

Contém figuras de "Computer Organization and Design: ARM edition", D. Patterson & J. Hennessey, MKP

Dois princípios

➠ Os computadores atuais seguem dois princípios-chave:

1. Instruções são representadas como números;
2. Programas (sequências de instruções) são guardados em memória, tal como dados.

➠ Programas podem ser fornecidos como ficheiros (de dados binários): os dados são as instruções do programa.

➠ Esses programas podem ser executados em computadores que aceitem o mesmo conjunto de instruções codificadas da mesma maneira: *compatibilidade binária*.

➠ Um programa (A) também pode ser executado por outro programa (V), que *interpreta* as instruções de A: V é um *simulador* ou uma *máquina virtual*.

➠ **Questão:** Como codificar as instruções?

- critérios (tipos de instruções, tipos de dados, modelo de execução)
- formatos

Código-máquina e código assembly

➠ O código de um programa pode ser representado por números: *código-máquina*.

Exemplo (em hexadecimal, ARMv8):

```
0x4300018b
0x220140f8
0x06c100f8
```

➠ Código simbólico para instruções (mnemónicas): *assembly code*

O mesmo exemplo:

```
add     X3,X2,X1
ldur    X2,[X9]
stur    X6,[X3,12]
```

➠ Conversão de código *assembly* para código-máquina também é feita por um programa: *assembler*

➠ O código-máquina difere entre processadores de famílias diferentes. O código-máquina de um Intel Xeon é diferente do de um ARM Cortex-A53.

Modelo de programação

- O modelo de programação de um microprocessador é definido por:
  1. modelo de execução
  2. conjunto de instruções
     1. classes (ou tipos) de instruções
     2. modos de especificação de operandos (endereçamento)
  3. registos
     1. de uso geral
     2. dedicados (de uso específico)
- Modelo de execução:
  1. inicializar PC *(program counter)*
  2. obter instrução da posição PC da memória
  3. executar instrução e atualizar PC
  4. repetir a partir de 2

Classes de instruções

- As classes de instruções mais comuns são:

1. Operações aritméticas com números inteiros
   - adição, subtração, multiplicação, divisão
2. Operações lógicas sobre conjuntos de bits (números sem sinal)
   - AND, OR, NOR, deslocamentos *(shift)*
3. Transferências de dados
   - leitura e escrita de dados em memória
4. Alteração do fluxo (sequencial) de execução
   - saltos condicionais e comparações
   - saltos incondicionais
   - execução de sub-rotinas

- As instruções de salto têm explicitamente a função de alterar o valor do PC.

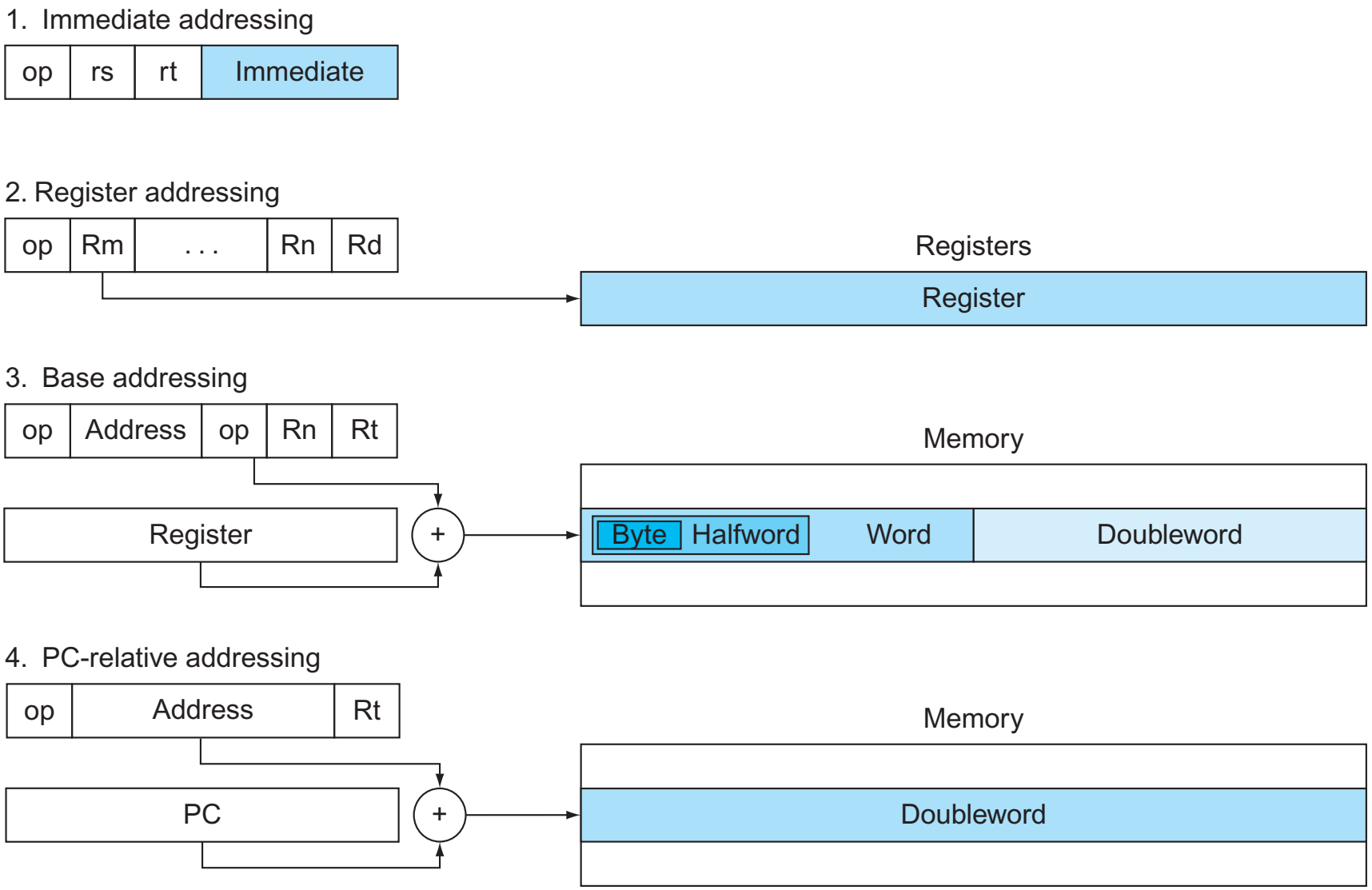
Modos de endereçamento

➠ Modos de endereçamento = modos de especificação dos operandos

Os mais comuns são:

1. **imediato:** o valor (constante) está incluído na instrução.
2. **registo:** o valor está num registo; a instrução inclui a especificação do registo.
3. **direto:** a instrução inclui o endereço da posição de memória.
4. **indireto** (via registo): o registo contém o endereço da posição de memória onde está o valor; a instrução especifica o registo.
5. **indireto** com deslocamento constante: instrução especifica registo e um valor constante: a posição de memória é obtida por soma do valor constante com o conteúdo do registo.
   (É uma generalização da categoria anterior.)
6. **relativo ao PC**: a instrução inclui constante a adicionar ao valor de PC.

Resumo dos modos de endereçamento AARCH64



Classificação segundo a origem dos operandos

Mem.	Max. ops.	Arquitetura	Exemplos
0	3	reg-reg	ARM, MIPS, SPARC
1	2	reg-mem	IBM 360/370, Intel 80x86
2	2	mem-mem	VAX
3	3	mem-mem	VAX

Tipo	Vantagens	Desvantagens
reg-reg	Codificação simples, comprimento único. Geração de código simplificada. Duração similar.	Número de instruções elevado. Programas mais compridos.
reg-mem	Acesso a dados sem "load" em separado. Tendem a ter boa densidade de codificação.	Operandos não são equivalentes. Duração varia com a localização dos operandos. Pode restringir o número de registos codificáveis.
mem-mem	Programas compactos. Não ocupa registos com resultados temporários.	Comprimento de instruções muito variável. Complexidade de instruções muito variável. Acesso a memória é crítico.

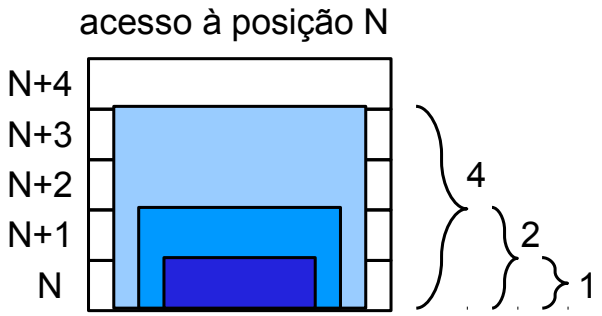
- As duas principais caraterísticas que diferenciam arquiteturas com registos de uso genérico são:
  1. número de operandos: 2 ou 3
  2. quantos operandos podem residir em memória: de 0 a 3

Tipos de operandos

- Tipos comuns de operandos:
  1. números inteiros de:
     - 8 bytes (palavra dupla, *doubleword*)
     - 4 bytes (1 palavra, *word*)
     - 2 bytes (meia palavra, *half-word*)
     - 1 byte
  2. números de vírgula flutuante:
     - 4 bytes (precisão simples *single, float*)
     - 8 bytes (precisão dupla, *double*)

- A interpretação dos dados e o seu tamanho são definidos pela instrução usada para os processar. O programador e/ou o compilador são responsáveis pela utilização coerente das instruções.

- Endereço de memória do item especifica **a posição do primeiro byte**.



- Regras de **alinhamento** típicas:
  - palavra: só endereços múltiplos de 4
  - meia palavra: só endereços múltiplos de 2
  - byte: qualquer endereço

1 Arquitetura do conjunto de instruções

2 Conjunto de instruções ARMv8 (AArch64)

3 Programação em Assembly ARMv8

4 Definição e utilização de sub-rotinas

Características das instruções ARMv8

- Conjunto de instruções reduzido (RISC = **R**educed **I**nstruction **S**et **C**omputer)
- Organização **reg-reg**
- Acesso a memória: apenas instruções para "loads" e "stores"
- Instruções lógicas e aritméticas com 3 registos (2 operandos e 1 resultado)
- Conjunto de instruções "ortogonal":
  - Onde pode ser usado um registo, pode ser usado qualquer outro (quase sempre).
- Todas as instruções têm 32 bits de comprimento e têm endereços que são múltiplos de 4.
- Endereços válidos: 2^{64} bytes (2^{62} palavras)
- 32 registos (0-31) de 64 bits bits: `X0`, `X1`, etc.

Uso especial: `X28` – `X30` `X31` ≡ `XZR` tem sempre o valor 0 (zero)

Registo especial de 64 bits: `PC` *(program counter)*

Subconjunto de instruções AARCH(64) (I)

Operação	Sintaxe	Significado
adição	`add dest,op1,op2`	dest ←op1 + op2
subtração	`sub dest,op1,op2`	dest ←op1 - op2
multiplicação (unsigned)	`mul dest,op1,op2`	dest ←op1 × op2
divisão (unsigned)	`udiv dest,op1,op2`	dest ←op1 ÷ op2
E-lógico bit-a-bit	`and dest,op1,op2`	dest ←op1 AND op2
OU-lógico bit-a-bit	`orr dest,op1,op2`	dest ←op1 OR op2
OU exclusivo bit-a-bit	`eor dest,op1,op2`	dest ←op1 XOR op2
deslocamento lógico para a esquerda	`lsl dest,op1,op2`	dest ←op1 << op2
deslocamento lógico para a direita	`lsr dest,op1,op2`	dest ←op1 >> op2
deslocamento aritmético para a direita	`asr dest,op1,op2`	dest ←op1 >> op2 (sinal)
rotação para a direita	`ror dest,op1,op2`	
transferência	`mov dest,op1`	dest ←op1
transferência e negação	`mvn dest,op1`	dest ←NOT(op1)
transferência de constante (*)	`movk dest,imm16`	dest[15:0] ←imm16
transferência de constante	`movz dest,imm16`	dest ←imm16 (extensão com 0)
carregar endereço	`adr dest,etiqueta`	dest ←etiqueta

(*) preserva restantes bits do destino; <dest> e <op1> são registos; <op2> é um registo ou valor imediato de 12 bits

Subconjunto de instruções AArch64 (II)

- Instruções de acesso a memória

Operação	Sintaxe	Significado
transf. de memória	`ldur dest,[op1{,offset}]`	dest ←Mem[op1+offset]
tranf. de memória (word)	`ldursw dest,[op1{,offset}]`	dest ←Mem[op1+offset]
transf. para memória	`stur fonte,[op1{,offset}]`	Mem[op1+offset] ←fonte

O segundo operando especifica o endereço de memória a usar.

`{offset}` é uma constante opcional de 9 bits (com sinal).

O *endereço efetivo* é calculado como $\boxed{\text{op1} + \text{offset}}$.

A instrução ldursw converte o valor de origem (32 bits, com sinal) para 64 bits por extensão de sinal.

O registo de indicadores

➠ O registo especial NZVC contém quatro bits que podem ser afetados pelo resultado de uma instrução. Esses bits designam-se por "indicadores de condição" ou *flags*)

Nome	Comportamento
N	N ←1 quando os resultado da operação é negativo, senão N ←0.
Z	Z ←1 quando os resultado da operação é 0, senão Z ←0.
C	C ←1 quando a operação resulta em transporte do MSB, senão C ←0.
V	V ←1 se a operação resulta em overflow, senão V ←0.

Os sufixos {cond} correspondem às seguintes condições:

Sufixo	*Flags*	Significado	Sufixo	*Flags*	Significado
EQ	Z= 1	igual	VC	V=0	sem overflow
NE	Z=0	diferente	HI	C=1 e Z=0	maior (sem sinal)
CS ou HS	C=1	maior ou igual (s/s)	LS	C=0 ou Z=1	menor ou igual (s/s)
CC ou LO	C = 0	menor que (s/s)	GE	N=V	maior ou igual (c/s)
MI	N=1	negativo	LT	N!=V	menor que (c/s)
PL	N=0	positivo ou 0	T	Z=0 e N=V	maior que (c/s)
VS	V=1	overflow (c/s)	LE	Z=1 e N!=V	menor ou igual (c/s)

Subconjunto de instruções AArch64 (III)

- Instruções de teste e transferência de fluxo

Operação	Sintaxe	Significado
comparação aritmética	`cmp op1,op2`	flags como em "op1-op2"
comparação negada	`cmn op1,op2`	flags como em "op1+op2"
comparação lógica	`tst op1,op2`	flags como em "op1 AND op2"
comparação igualdade lógica	`teq op1,op2`	flags como em "op1 XOR op2"
salto incondicional	`b alvo`	PC ←alvo
salto condicional	`b{.cond} alvo`	se {cond}=verdade, PC ←alvo
comparação e salto	`cbz op1, alvo`	se op1=0, PC ←alvo
	`cbnz op1, alvo`	se op1 != 0, PC ←alvo

- Alteração dos indicadores:
  1. instruções de comparação e teste;
  2. algumas instruções aritméticas e lógicas com S como última letra da mnemónica: adds, ands, subs

Resumo dos saltos condicionais

- Principais saltos condicionais

	Signed numbers		Unsigned numbers	
Comparison	Instruction	CC Test	Instruction	CC Test
=	B.EQ	Z=1	B.EQ	Z=1
≠	B.NE	Z=0	B.NE	Z=0
<	B.LT	N!=V	B.LO	C=0
≤	B.LE	~(Z=0 & N=V)	B.LS	~(Z=0 & C=1)
>	B.GT	(Z=0 & N=V)	B.HI	(Z=0 & C=1)
≥	B.GE	N=V	B.HS	C=1

Instruções condicionais

- 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` X10←X11 se condição verdadeira, senão X10 ←X12
- csinc é similar, mas incrementa o valor do 2º operando. Exemplo:

  `csinc X0,X2,X3,NE`

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

- Outras instruções similares: csneg (2º operando com sinal trocado), csinv (2º operando complementado).
- cset Coloca registo a 1 (se condição for verdadeira) ou a 0.

  `cset X10,GE`

1 Arquitetura do conjunto de instruções

2 Conjunto de instruções ARMv8 (AArch64)

3 Programação em Assembly ARMv8

4 Definição e utilização de sub-rotinas

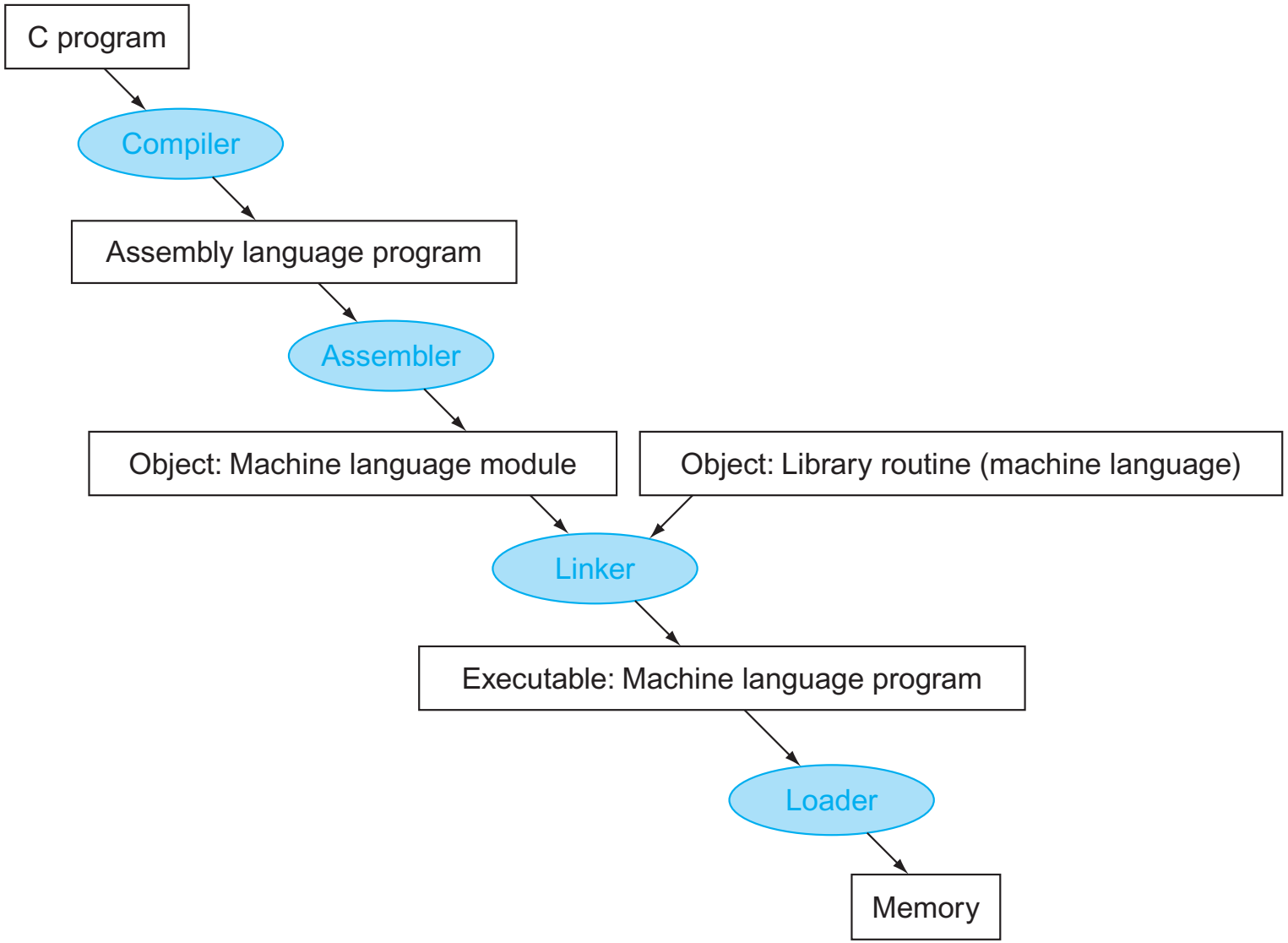
Fluxo (simplificado) de criação de programas

1. Preparar programa com editor de texto (1 ou mais ficheiros)
2. Invocar *assembler* para converter ficheiros para código-máquina
3. "Ligar" programa às sub-rotinas do sistema *(linker)*
4. Executar (talvez usando um emulador)
   O programa deve ser carregado previamente para memória *(loader)*
5. Depurar e voltar a 1.

➠ O que é preciso saber sobre o "ambiente de execução"?

1. organização de memória (sistema operativo e aplicação)
2. onde fica colocado o código e as zonas de dados
3. sub-rotinas disponíveis (sistema ou bibliotecas de funções)
4. como invocar serviços do sistema operativo (se existirem) e como aceder a periféricos

Etapas da produção de um programa executável



Assembler

- Função principal: código *assembly* $\rightarrow$ código-máquina.
- Facilitar a programação:
  1. verificar a "legalidade" das instruções
     - sintaxe das instruções, tamanho das constantes, ...
  2. nomes para posições de memória: etiquetas
  3. reserva de zonas de memória para dados (alocação de memória)
  4. especificação de valores iniciais para zonas de memória
  5. síntese de instruções úteis (pseudo-instruções) ou de "sinónimos"
  6. ajuste de saltos, dependendo da distância ao destino
  7. definir procedimentos para geração de grupos de instruções (macro-instruções)
     - O próprio *assembler* é programável!
- Opcionalmente produzir listagens anotadas do código gerado.

Exemplos: expressões numéricas

- Código para calcular a expressão

```
f = (g + h) - (i + j)
```

- Atribuição de variáveis a registos: f, ...j $\longrightarrow$ X0, ..., X4 (exemplo)

```
add X8, X1, X2
add X9, X3, X4
sub X0, X8, X9
```

- Código para calcular a expressão

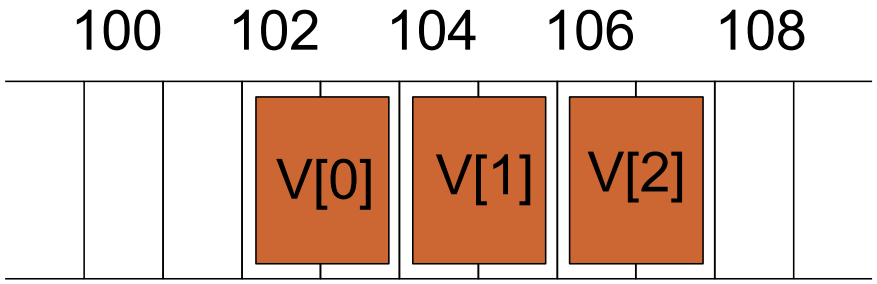
```
f = (g + h - 100) - (i + 120)
```

- Atribuição de variáveis a registos: f, ...i $\longrightarrow$ X0, ..., X3 (exemplo)

```
add X8, X1, X2
add X9, X3, 120
sub X8, X8, 100
sub X0, X8, X9
```

Armazenamento de sequências de valores em memória

- Sequências homogéneas (todos os elementos são do mesmo tipo):
  1. Sequência `V[]` com N elementos: `V[0]`, , . . . , `V[N-1]`.
  2. Todos os elementos têm o mesmo tamanho S (em bytes).
  3. Uma sequência de N elementos, cada um de S bytes, ocupa N$\times$S bytes.
  4. Cada elemento de uma sequência pode ser especificado pelo par de números (b, d):
     1. *endereço-base da sequência b*: endereço do primeiro elemento;
     2. *deslocamento d*: distância do elemento ao início da da sequência.
  5. O deslocamento associado a `V[i]` é: $d = i \times S$
- Exemplo: Disposição de uma sequência de 3 meias-palavras em memória:



- $b = 102$
- endereço de V[1]:

$$b + 1 \times 2 = 104$$

Exemplos: Acesso a memória

- Operandos em memória:

```
g = h + A[8]
```

- Atribuição de variáveis a registos: g: X1, h: X2, endereço-base de A[]: X3

```
ldur   X9,[X3, 64]
add    X1,X2,X9
```

- Porque é que o valor do deslocamento é 64?
- Código correspondente a:

```
A[12] = h + A[8]
```

- Atribuição de variáveis a registos: h:X2, endereço-base de A[]: X3

```
ldur   X9,[X3, 64]
add    X9,X2,X9
stur   X9,[X3, 96]
```

Saltos condicionais

- Qual é o código *assembly* correspondente a:

```
se (i = j) f = g + h
senão      f = g - h
```

- Atribuição de variáveis a registos: f, g, ... j $\rightarrow$ X0, X1, ..., X4

```
        cmp     X3, X4
        b.ne    Alt
        add     X0, X1, X2
        b       Cont
Alt:    sub     X0, X1, X2
Cont:   ....
```

Ciclos (repetição de grupos de instruções)

- Código *assembly* correspondente a:

```
enquanto (V[i] = k) i = i + 1
```

- Atribuição de variáveis a registos: i $\rightarrow$ X3, k $\rightarrow$ X5, base de V[] $\rightarrow$ X6

```
ciclo:  lsl     X1,X3,3
        add     X7,X6,X1
        ldur    X0,[X7]
        cmp     X0,X5
        b.ne    cont
        add     X3,X3,1
        b       ciclo
cont:   ....
```

1 Arquitetura do conjunto de instruções

2 Conjunto de instruções ARMv8 (AArch64)

3 Programação em Assembly ARMv8

4 Definição e utilização de sub-rotinas

Sequência de ações

- Para a utilização correta de uma sub-rotina, as tarefas a realizar são:
  1. Colocar argumentos em registos
  2. Passar o fluxo de execução para o código da sub-rotina (invocar)
  3. Reservar espaço para dados da sub-rotina (não tratado nesta u.c.)
  4. Realizar as operações da sub-rotina
  5. Colocar o resultado (se houver) em registo
  6. Retomar o fluxo de execução a partir do ponto de invocação (retornar)
- Regras para o uso dos registos:
  - `X0`–`X7` : argumentos
    Os registos devem ser preenchidos **por ordem**
  - `X0`, `X1`: resultado
    Resultados até 1 *double word* em `X0`, de 2 *doublewords* em `X0` e `X1`
  - Registo `X30` ou `LR`: guarda o endereço de retorno
    endereço da instrução a executar quando a a sub-rotina terminar
  - `X19`–`X27`: conteúdo inicial e final **deve ser o mesmo** (valores *preservados*)

Name	Register number	Usage	Preserved on call?
X0–X7	0–7	Arguments/Results	no
X8	8	Indirect result location register	no
X9–X15	9–15	Temporaries	no
X16 (IP0)	16	May be used by linker as a scratch register; other times used as temporary register	no
X17 (IP1)	17	May be used by linker as a scratch register; other times used as temporary register	no
X18	18	Platform register for platform independent code; otherwise a temporary register	no
X19–X27	19–27	Saved	yes
X28 (SP)	28	Stack Pointer	yes
X29 (FP)	29	Frame Pointer	yes
X30 (LR)	30	Link Register (return address)	yes
XZR	31	The constant value 0	n.a.

Instruções para sub-rotinas

- Invocação da sub-rotina com a instrução `bl` (*branch and link*):
  1. guardar o endereço da instrução seguinte no registo `X31` (LR)
  2. saltar para a primeira instrução da sub-rotina (endereço representado por uma *etiqueta*)

```
bl etiqueta_sub_rotina
```

- Retorno da sub-rotina: instrução `ret` (sem argumentos)
  - Retornar da sub-rotina = retomar a execução a partir da instrução colocada em memória a seguir à instrução `bl` que foi usada para a invocação.
  - Os registos `X0` e (eventualmente) `X1` devem já conter o resultado.
  - O registo `X30` (cujo conteúdo deve ser igual ao que tinha no início da sub-rotina) deve conter o endereço da próxima instrução a executar e que é o endereço da instrução situada em memória imediatamente a seguir à instrução `bl` usada para invocação.
- Distinção entre sub-rotinas:

função sub-rotina que devolve um valor como resultado

procedimento sub-rotina que **não** devolve resultados

Exemplo: sub-rotina terminal (1)

➠ Uma sub-rotina **terminal não invoca outras sub-rotinas**.

➠ **Importante:** Sub-rotinas não-terminais devem preservar o valor de LR antes de invocarem outras sub-rotinas.

➠ Nem todos os aspetos da utilização de sub-rotinas são abordados. Por exemplo: Como reservar espaço para preservar o valor de LR? Como implementar sub-rotinas com mais de 8 parâmetros?

➠ Exemplo de um algoritmo a implementar

Sub-rotina *exemplo*(g, h, i, j)

$x \leftarrow (g + h) - (i + j)$;
se $x < 0$ **então**
$x \leftarrow -x$
Resultado: x

- Associação entre parâmetros e registos é feita por ordem
  X0: g X1: h X2: i X3; j
- No final da execução da sub-rotina, o registo X0 deve conter o valor de x.
  O conteúdo de X1 não é relevante.

Exemplo: sub-rotina terminal (2)

- Código *assembly* para a sub-rotina *exemplo*

```
// Início da sub-rotina marcado
// por uma etiqueta (nome da sub-rotina)

exemplo:  add     X10, X0, X1
          add     X11, X2, X3
          subs    X0, X10, X11    afeta flags
         // resultado negativo?
         csneg    X1, X0, X0, PL
         ret
// Fim do código da sub-rotina
// sem marcas especiais
```

Exemplo: sub-rotina terminal (3)

- Calcular *exemplo(10, 12, 5, 15)*

```
mov    X0,10
mov    X1,12
mov    X2,5
mov    3,5
bl     exemplo
//   X0 tem agora o valor 2
cmp    X0, 10
b.lt    L1
```

- Salto para L1 é tomado?

Sim

- Calcular *exemplo(-3, -10, 1, 2)*

```
mov    X0, -3
mov    X1, -10
mov    X2, 1
mov    X3, 2
bl     exemplo
//   X0 tem agora o valor 16
cmp    X0, 10
b.lt    L1
```

- Salto para L1 é tomado?

Não