Conjuntos de instruções de microprocessadores

Arquitetura ARMv7

João Canas Ferreira

Novembro de 2018

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Tópicos

1. Arquitetura do conjunto de instruções
2. Conjunto de instruções ARMv7
3. Programação em Assembly
4. Definição e utilização de sub-rotinas

Contém figuras de "Computer Organization and Design", D. Patterson & J. Hennessey, 3ª. ed., MKP

1. Arquitetura do conjunto de instruções
2. Conjunto de instruções ARMv7
3. Programação em Assembly
4. Definição e utilização de sub-rotinas

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, ARMv7):

```
023081E0
000095E5
046083E4
```

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

O mesmo exemplo:

```
add   R3, R2, R1
ldr   R0, [R5]
str   R6, [R3], #4
```

➠ 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 Xeon é diferente do código-máquina de um processador Cortex-A5.

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
- ARMv7: o registo R15 é o PC.

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.

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

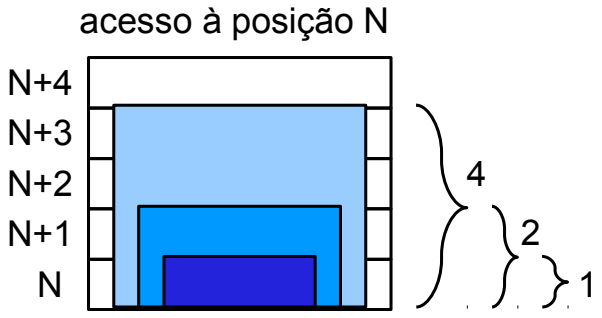
Tipos de operandos

- Tipos comuns de operandos:

1. números inteiros de:
   - 4 bytes (1 palavra)
   - 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 ARMv7

3 Programação em Assembly

4 Definição e utilização de sub-rotinas

Caraterísticas das instruções ARMv7

- Conjunto de instruções reduzido (RISC = **R**educed **I**nstruction **S**et **C**omputer)
- Organização **reg-reg**
- Acesso a memória:
  - apenas **ldr** (leitura: CPU $\leftarrow$ MEM) e **str** (escrita: MEM$\leftarrow$ CPU)
- 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
- Endereços válidos: 2^{32} bytes (2^{30} palavras)
- 16 registos (0-15) de 32 bits: `0`, `1`, etc.

Uso especial: 15 $\equiv$ PC, 14 $\equiv$ LR (Link register) para sub-rotinas

- Registo de estado: CPSR *(Current Processor Status Register)*

Utilização convencional dos registos

➠ Sub-rotinas devem ser escritas de forma independente da sua invocação/utilização (por programas **escritos separadamente**): modularidade.

➠ A **interoperabilidade** das sub-rotinas requer o uso de **convenções de utilização dos registos** *(calling conventions)*, que variam com o conjunto de instruções e, possivelmente, com o sistema operativo usado.
Estas regras fazem parte da *interface binária de programas* (ABI = *Application Binary Interface)*

➠ Convenções de utilização de registos numa sub-rotina para ARMv7:

- `0`-`3`: uso sem restrições; são usados para passar os argumentos de uma sub-rotina
  `0` contém o resultado da sub-rotina.
- 4-9: conteúdo deve ser preservado (valor inicial igual a valor final)
- 13: reservado para gestão de uma pilha de dados.
- LR guarda o endereço de retorno de uma sub-rotina

Subconjunto de instruções armV7a (I)

Operação	Sintaxe	Significado
adição	`add dest,op1,op2`	dest ←op1 + op2
subtração	`sub dest,op1,op2`	dest ←op1 - op2
subtração inversa	`rsub dest,op1,op2`	dest ←op2 - op1
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
E-lógico e negação bit-a-bit	`bic dest,op1,op2`	dest ←op1 AND (NOT 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`	
rotação para a direita com carry	`rrx dest,op1,op2`	
transferência	`mov dest,op1`	dest ←op1
transferência e negação	`mvn dest,op1`	dest ←NOT(op1)
carregar endereço	`adr dest,etiqueta`	dest ←etiqueta

Subconjunto de instruções ARMv7A (II)

Operação	Sintaxe	Significado
transf. de memória	`ldr dest,[op1{,offset}]`	dest ←Mem[op1+offset]
tranf. de memória (byte)	`ldrb dest,[op1{,offset}]`	dest ←Mem[op1+offset]
transf. para memória	`str fonte,[op1{,offset}]`	Mem[op1+offset] ←fonte
transf. para memória (byte)	`strb fonte,[op1{,offset}]`	Mem[op1+offset] ←fonte
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

➠ Os valores de {cond} estão na página seguinte. Os valores dos indicadores (*flags*)) são afetados pelas instruções de comparação.

➠ As instruções começam **sempre** em posições cujos endereços são **múltiplos de 4**.

Instruções de acesso a memória

- A arquitetura ARM tem um conjunto de instruções *load-store*.

- O segundo operando especifica o endereço de memória a usar.

Offset pode ser uma constante de 12 bits (com sinal) ou um registo.

O *endereço efetivo* é calculado como op1 + offset.

- Obter de memória (leitura, *load*): `ldr` e `ldrb`
  - A instrução `ldrb` lê 1 byte de memória e guarda-o no byte menos significativo de um registo (dest).
    Os demais bits do registo de destino são colocados a zero.

- Armazenar em memória (escrita, *store*): `str` e `strb`
  - A instrução `strb` escreve apenas o byte menos significativo em memória

- **Atenção:** para acessos a palavras, o endereço final deve ser múltiplo de 4.

- Instruções podem ter sufixo condicional, mas nunca alteram *flags*.

Operando flexível

➠ O operando op1 é sempre um registo.

➠ Em muitas instruções, o operando op2 (apenas!) pode ter várias formas: *operando flexível*:

- registo
- uma constante (valor imediato) de 32 bits que possa ser produzido a partir de 8 bits por um número par de rotações para a direita

➠ Exemplos de constantes permitidas
(as letras *a-h* representam os bits menos significativos):

- `00000000 00000000 000000000 abcdefgh`
- `gh000000 00000000 000000000 00abcdef`
- `efgh0000 00000000 000000000 0000abcd`
- ...
- `00000000 00000000 0000000ab cdefgh00`

O registo de indicadores

➠ O registo CPSR contém quatro bits que podem ser afetados pelo resultado de uma instrução. (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)

Execução condicional de instruções

- Muitas instruções podem ser executadas condicionalmente.
- Formato: <instrução><sufixo> <operandos>
- Exemplo: `addeq R1,R2,R3` apenas é executada se *flag* Z=1
- Qual é o valor final de R3?

```
mov     R3,#2
mov     R4,#3
cmp     R3,R4
addeq   R3,R4,#4
```

Resposta: 2

- As instruções aritméticas, lógicas, de comparação, de transferência de dados e saltos podem ser executadas condicionalmente.
- Instruções sem sufixo ou com o sufixo AL são sempre executadas (AL = always).

Alteração de indicadores

- Os indicadores podem ser alterados por:
  1. instruções de comparação e teste: `cmp`, `cmn`, `tst`, `teq`.
  2. instruções aritméticas e lógicas com `S` como 4ª letra da mnemónica: `adds`, `ands`, `...`
- Qual é o valor final de `R6` em cada um dos exemplos:

```
mov     R3,#2
mov     R4,#-3
adds    R6,R4,R3
addmi   R6,R6,#4
```

Reposta: R6=3
A última instrução é executada

```
mov     R3,#2
mov     R4,#-3
add     R6,R4,R3
addmi   R6,R6,#4
```

R6: não pode ser determinado
O valor do indicador N não é conhecido!

- Estas instruções podem ter sufixo condicional (`addsmi`, `addseq` ...).

1 Arquitetura do conjunto de instruções

2 Conjunto de instruções ARMv7

3 Programação em Assembly

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

➠ Emulador VisUAL: `https://salmanarif.bitbucket.io/visual/index.html`

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 →R0, ..., R4

```
add R8, R1, R2
add R9, R3, 4
sub R0, R8, R9
```

- Código para calcular a expressão

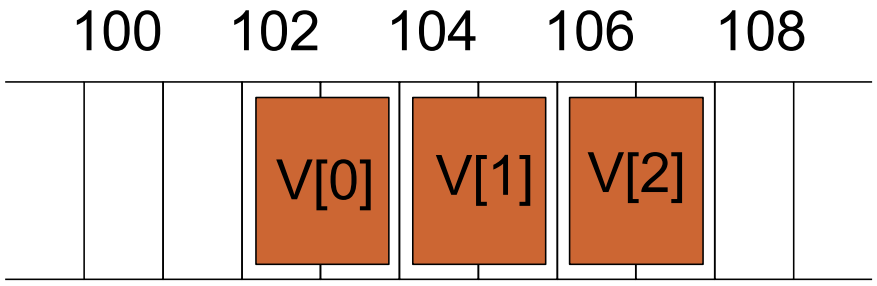
```
f = (g + h - 100) - (i + 320)
```

- Atribuição de variáveis a registos: f, ...i →R0, ..., R3

```
add R8, R1, R2
add R9, R3, #320
sub R8, R8, #100
sub R0, R8, R9
```

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: R1, h: R2, endereço-base de A: R3

```
ldr R9,[R3,#32]
add R1,R2, R9
```

- Porque é que o valor do deslocamento é 32?
- Código correspondente a:

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

- Atribuição de variáveis a registos: h:R2, endereço-base de A: R3

```
ldr R9, [R3,#32]
add R9, R2,R9
str R9, [S3,#4]
```

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 $\rightarrow$ R0, R1, ..., R4

```
        cmp     R3, R4
        bne     Alt
        add     R0, R1, R2
        b       Cont
Alt     sub     R0, R1, R2
Cont    ....
```

- Alternativa com instruções de execução condicional:

```
cmp     R3, R4
addeq   R0, R1, R2
subne   R0, R1, R2
```

Ciclos (repetição de grupos de instruções)

- Código *assembly* correspondente a:

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

- Atribuição de variáveis a registos: i $\rightarrow$R3, k $\rightarrow$R5, base de V[] $\rightarrow$R6

```
Ciclo     lsl   R1, R3, #2
          ldr   R0, [R6,R1]
          cmp   R0, R5
          addeq R3, R3, #1
          beq   Ciclo
Cont      . . .
```

1 Arquitetura do conjunto de instruções

2 Conjunto de instruções ARMv7

3 Programação em Assembly

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 UC)
  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:
  - `R0`–`R3` : argumentos
    Os registos devem ser preenchidos **por ordem**
  - `R0`, `R1`: resultado
    Resultados de 1 palavra em `V0`, de 2 palavras em `R0` e `R1`
  - `LR`: guarda o endereço de retorno
    endereço da instrução a executar quando a a sub-rotina terminar
  - R4–R9: conteúdo inicial e final **deve ser o mesmo**

Instruções para sub-rotinas

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

```
bl etiqueta_sub_rotina
```

- Retorno da sub-rotina por atribuição ao registo PC
  - 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 de resultado `R0` e (eventualmente) `R1` devem já conter o resultado
  - O registo `LR` (cujo conteúdo não deve ser alterado pelo código da sub-rotina) contém o endereço desejado (lá colocado pela instrução `bl`)

```
mov PC,BL
```

- 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 4 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
  R0: g R1: h R2: i R3; j
- No final da execução da sub-rotina, o registo R0 deve conter o valor de x.
  O conteúdo de R1 não é relevante.

Exemplo: sub-rotina terminal (2)

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

```
; Início da sub-rotina marcado
; por uma etiqueta

exemplo  add     R10, R0, R1
         add     R11, R2, R3
         subs    R0, R10, R11   ; afeta flags
         ; resultado negativo?
         rsblt   R0, R0, #0
         mov     PC, LR

; Fim do código da sub-rotina
; sem marcas especiais
```

Exemplo: sub-rotina terminal (3)

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

```
mov    R0, #10
mov    R1, #12
mov    R2, #15
mov    R3, #15
bl     exemplo
;  R0 tem agora o valor 8
cmp    R0, #10
blt    L1
```

- Salto para L1 é tomado?

Sim

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

```
mov    R0, #-3
mov    R1, #-13
mov    R2, #1
mov    R3, #2
bl     exemplo
;  R0 tem agora o valor 19
cmp    R0, #10
blt    L1
```

- Salto para L1 é tomado?

Não