Arquitetura e Organização de Computadores

Exercícios - Parte II

António José Araújo

João Canas Ferreira

Bruno Lima

Mestrado Integrado em Engenharia Informática e Computação

Novembro de 2018

Conteúdo

1 Linguagem *assembly*

1.1 Sumário das instruções ARM

As instruções ARMv7 utilizadas em AOCO estão indicadas na tabela seguinte.

Categoria	**Instrução**
Aritmética	Add
	Subtract
	Add with Carry
	Reverse Subtract
	Reverse Subtract with Carry
Lógica	Bitwise And
	Bitwise Exclusive Or
	Bitwise Clear
	Bitwise Or
	Logical Shift Right
	Arithmetic Shift Right
	Rotate Right
	Rotate Right and Extend
Transferência de dados	Move
	Move Negated
	Address Load
	LDR Psuedo-Instruction
	Load Register
	Store Register
	Load Multiple Registers
	Store Multiple Registers
Comparações e saltos	Compare
	Compare Negated
	Test Bit(s) Set
	Test Equals
	Branch
	Branch with Link

1.2 Exercícios resolvidos

Exercício 1
Para as seguintes expressões aritméticas (números inteiros de 32 bits), especifique um mapeamento de variáveis para registos e o fragmento de código *assembly* ARM que as implementa.

a) `f = g - (f + 5)`

b) `f = A[12] + 17`

O primeiro passo neste tipo de problemas é escolher uma atribuição de variáveis a registos. Cada variável é atribuída a um registo. Como a arquitetura ARM possui 12 registos de uso geral, trata-se de uma tarefa simples porque, neste caso, se pode usar um registo diferente para cada variável.

a) Uma possível atribuição de registos a variáveis é a seguinte:

`R0`: `f` `R1`: `g`

O fragmento de código que realiza os cálculos desejados é:

```
add     R0, R0, #5              ; Calcula f = f + 5
sub     R0, R1, R0              ; Calcula f = g - f
```

Após esta sequência de duas instruções, `R0` contém o novo valor associado a `f`. O cálculo da primeira parte da expressão (instrução `add`) pode guardar o resultado intermédio no registo `R0`, porque a segunda instrução estabelece o valor final correto.

b) Possível atribuição de variáveis a registos:

`R0`: `f` `R6`: endereço base de `A`

Como cada elemento de uma sequência de inteiros tem 4 bytes, o elemento de índice 12 da sequência `A` está guardado a partir da posição de memória cujo endereço é dado por:

$$\texttt{endereço base de A} + 12 \times 4$$

A primeira instrução deve ir buscar o valor guardado nessa posição.

```
ldr     R0, [R6, #48]           ; Carrega valor da posição R6+48
add     R0, R0, #17             ; Soma-lhe o valor 17
```

Exercício 2

Assuma as seguintes condições iniciais:

```
R0 = 0XBEADFEED          R1 = 0xDEADFADE
```

a) Determine o valor de `R2` após a execução da seguinte sequência de instruções:

```
lsl     R2, R0, #4
orr     R2, R2, R1
```

b) Determine o valor de `R2` após a execução da seguinte sequência de instruções:

```
lsr     R2, R0, #3
and     R2, R2, #0xFFFFFFEFH
```

Em binário, os valores iniciais dos registos são:

$$\texttt{R0} = 1011\ 1110\ 1010\ 1101\ 1111\ 1110\ 1110\ 1101_2$$
$$\texttt{R1} = 1101\ 1110\ 1010\ 1101\ 1111\ 1010\ 1101\ 1110_2$$

a) A primeira instrução desloca o valor de `R0` quatro bits para a esquerda. Os 4 bits mais significativos perdem-se; nos 4 menos significativos são introduzidos zeros. O resultado da operação é guardado em `R2`; o registo `R0` fica inalterado. O valor de `R2` depois da execução da primeira instrução é:

$$\texttt{R2} = 1110\ 1010\ 1101\ 1111\ 1110\ 1110\ 1101\ 0000_2$$

A instrução `orr` calcula a função ou-inclusivo de cada bit de `R2` com o bit de `R0` situado na mesma posição. O resultado é guardado em `R2`. O resultado da operação `orr` é 1 sempre que pelo menos um dos operandos seja 1. Logo:

$$\texttt{R2} = 1111\ 1110\ 1111\ 1111\ 1111\ 1110\ 1101\ 1110_2 = \texttt{0xFEFFFEDE}$$

b) A instrução `lsr` desloca o valor de `R0` três posições para a direita, introduzindo zeros pela esquerda. O valor de `R2` depois da execução da primeira instrução é:

$$\texttt{R2} = 000\ 1\ 0111\ 1101\ 0101\ 1011\ 1111\ 1101\ 1101_2$$

A instrução `and` calcula a função e-lógico de cada bit de `R2` com o bit correspondente da constante `0xFFFFFFEF`

$$\text{FFFFFFEF}_{16} = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 1111_2$$

O resultado é guardado em `R2`. O e-lógico de dois bits tem resultado 1 apenas se ambos os operandos forem 1. Neste caso, os operandos são dados pelo conteúdo de `R2` e pela constante indicada. O valor final de `R2` é:

$$\texttt{R2} = 0001\ 0111\ 1101\ 0101\ 1011\ 1111\ 1100\ 1101_2 = \texttt{0x17D5BFCD}$$

Exercício 3

Assuma as seguintes condições iniciais:

$$\texttt{R0} = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2$$

$$\texttt{R1} = 0011\ 1111\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000_2$$

Determine o valor de `R1` após a execução do fragmento seguinte:

```
        cmp   R0, R1
        bge   ELSE
        b     DONE
ELSE    add   R1, R1, #2
DONE    ...
```

A primeira instrução compara os conteúdos de `R0` e `R1` alterando o valor de *flags* de acordo com o resultado da comparação. A operação realizada é equivalente a:

```
R0 - R1
```

Neste caso os valores dos indicadores (*flags)*) são alterados para N=1, Z=0, C=1 e V=0.

O salto condicional (segunda instrução) é tomado, se as *flags* N e V apresentarem o mesmo valor, o que se verifica quando `0` é maior ou igual a `1` (a condição `ge` interpreta os valores dos registos como sendo números em complemento para 2). Como neste caso o conteúdo de `R0` é negativo e o conteúdo de `R1` é positivo, o salto não será tomado.

Em consequência, a terceira instrução a ser executada é a de salto incondicional (a instrução `b`). Esta instrução leva o fluxo de execução a passar para o fim do fragmento apresentado (etiqueta `DONE`). A instrução `add` não é executada.

Como o conteúdo de `R1` não foi alterado, o seu valor são sofre alteração.

1.3 Exercícios propostos

Exercício 4

Para as seguintes expressões aritméticas (números inteiros de 32 bits), especifique um mapeamento de variáveis para registos e o fragmento de código *assembly* ARMv7 que as implementa.

a) `f = g + ( j + 2)`

b) `k = a + b - f + d - 30`

c) `f = g + h + A[4]`

d) `f = g - A[B[10]]`

e) `f = k - g + A[h + 9]`

f) `f = g - A[B[2] + 4]`

Exercício 5

Para os seguintes fragmentos de código *assembly* ARMv7, indique as expressões simbólicas correspondentes, bem como o correspondente mapeamento entre registos e variáveis.

a)
```
add     r0,r0,r1
add     r0,r0,r2
add     r0,r0,r3
add     r0,r0,r4
```

b)
```
ldr     r0,[r6,#4]
```

c) Assumir que `R6` contém o endereço-base da sequência `A[]`.

```
add     r6, r6, #-20
lsl     r10, r1, #2
add     r6, r6, r10
ldr     r0, [r6, #8]
```

Exercício 6

Assuma as seguintes condições iniciais:

`R0 = 0x55555555` `R1 = 0x12345678`

Determine o valor de `R2` após a execução das sequências de instruções seguintes.

a)
```
lsl     r2, r0, #4
orr     r2, r2, r1
```

b)
```
lsl     r2, r0, #4
and     r2, r2, #-1
```

c)
```
lsr     r2, r0, #3
and     r2, r2, #0x00EF
```

Exercício 7

Os processadores RISC como o ARM implementam apenas instruções muito simples. Este problema aborda exemplos de hipotéticas instruções mais complexas.

a) Considere uma instrução hipotética `abs` que coloca num registo o valor absoluto de outro registo.

`abs  T2, T1` é equivalente a `T2 ← |T1|`

Apresente a mais curta sequência de instruções ARMv7 que realiza esta operação.

b) Repita a alínea anterior para a instrução `sgt`, em que `sgt  T1, T2, T3` é equivalente a `se T2 > T3 então T1 ← 1 senão T1 ← 0`.

Exercício 8

Considere o seguinte fragmento de código *assembly* ARMv7:

```
L1      str   R4,[R5]
        lsl   R4,R4,#4
        add   R5,R5,#4
        cmp   R4, #0
        bne   L1
```

Assuma os seguintes valores iniciais:

```
R4 = 0x12345678          R5 = 2000
```

Explique como é alterada a memória durante a execução do programa. Apresente um diagrama do conteúdo da memória alterada pela execução do programa.

Soluções dos exercícios propostos

1 Linguagem *assembly*

Exercício 4
É necessário definir uma atribuição arbitrária de variáveis a registos.

a) Atribuição: `f` → `r0`, `g` → `r1`, `j` → `r2`.

```
add     r0, r2, #2
add     r0, r1, r0
```

b) Atribuição: `a` → `r0`, `b` → `r1`, `d` → `r2`, `f` → `r3`, `k` → `r4`.

```
add     r4, r0, r1
sub     r4, r4, r3
add     r4, r4, r2
add     r4, r4, #-30
```

c) Atribuição: `f` → `r0`, `g` → `r1`, `h` → `r2`, `A` → `r7`.

```
ldr     r0, [r7, #16]
add     r0, r0, r1
add     r0, r0, r2
```

d) Atribuição: `f` → `r0`, `g` → `r2`, `A` → `r6`, `B` → `r7`.

```
ldr     r5, [r7, #40]
lsl     r5, r5, #2
add     r5, r5, r6
ldr     r0, [r5]
sub     r0, r2, r0
```

e) Atribuição: `f` → `r0`, `g` → `r1`, `h` → `r2`, `k` → `r3`, `A` → `r6`.

```
add     r10, r2, #9
lsl     r10, r10, #2
add     r10, r6, r10
ldr     r0, [r10]
add     r0, r0, r3
sub     r0, r0, r1
```

f) Atribuição: `f` → `r0`, `g` → `r1`, `A` → `r6`, `B` → `r7`.

```
ldr     r10, [r7, #8]
add     r10, r10, #4
lsl     r10, r10, #2
add     r10, r6, r10
ldr     r10, [r10]
sub     r0, r1, r10
```

Exercício 5

a) Atribuição: `f` → `r0`, `g` → `r1`, `h` → `r2`, `i` → `r3`, `j` → `r4`

A expressão correspondente é: `f = f + g + h + i + j`

b) Atribuição: `f` → `r0`, `A` → `r6`

A expressão correspondente é: `f = A[1]`

c) Atribuição: `f` → `r0`, `g` → `r1`, `A` → `r6`

A expressão correspondente é: `f = A[g-3]`

Exercício 6

a) `0x57755778`

b) `0x55555550`

c) `0x000000AA`

Exercício 7

a) ...

b) ...

Exercício 8

...