1
Arm: características gerais

A arquitetura ARM, inicialmente projetada na década de 1980, evoluiu ao longo dos anos, gerando diferentes versões, com pequenas variações no repertório de instruções e poder de processamento, para serem usadas em máquinas com diferentes fins. A versão original usa palavras de 32 bits, a versão mais recente usa palavras de 64 bits. Estudaremos neste livro uma versão recente de 32 bits, mais especificamente a versão ARMv7. Várias companhias licenciam e produzem diferentes versões da arquitetura ARM, utilizada tanto em servidores e computadores pessoais como em dispositivos móveis como tablets e telefones celulares.

1.1 Modos de Operação

No Faíska há dois modos de operação: supervisor e usuário. No ARMv7, com o objetivo de tornar mais rápido o atendimento a interrupções e exceções, há sete modos de operação. Os modos de operação na versão ARMv7 são mostrados na Tabela 1.1.

Os modos de operação são definidos em função de interrupções e exceções. O modo *User* é o modo em que aplicações de usuário devem ser executadas. Neste modo apenas instruções seguras são permitidas de executar. Os modos *FIQ* e *Interrupt* são associados a interrupções externas. No Arm, há dois tipos de interrupções externas: uma interrupção "rápida" (denominada *FIQ*, do inglês *fast interrupt request* e uma interrupção "normal" (denominada *IRQ*, do inglês *interrupt request*). A interrupção do tipo *FIQ* é reservada para interrupções mais prioritárias, que necessitem de um tempo de atendimento mais curto. Quando uma interrupção é aceita, o processador entra no modo correspondente (*FIQ* ou *Interrupt*).

O processador entra no modo de operação *Abort* quando uma exceção de acesso à memória ocorre. O Arm possui um módulo gerenciador de memória que permite que regiões de memória sejam protegidas para escrita ou para execução. Quando uma instrução

Nome	Descrição
User	Modo usuário, único modo sem nenhum privilégio de execução.
FIQ	Modo de interrupção rápida, entra neste modo quando uma interrupção do tipo FIQ é aceita.
Interrupt	Modo de interrupção, entra neste modo quando uma interrupção do tipo IRQ é aceita.
Supervisor	Modo supervisor, entra neste modo quando o processador sofre um reset ou uma instrução de chamada ao sistema (SWI) é executada.
Abort	Modo aborto de acesso, entra neste modo quando uma exceção de acesso a memória é disparada (acesso desalinhado de palavra, por exemplo).
Undefined	Modo instrução indefinida, entra neste modo quando uma exceção de instrução indefinida é disparada.
System	Modo sistema, único modo em que o processador entra por programa (ligando um bit específico na palavra de estado).

Tabela 1.1: Modos de operação do processador ARMv7.

viola uma proteção de memória uma exceção de tipo *abort* é gerada. Exceções de tipo *abort* também são geradas em caso de memória inexistente, ou *falta de página* em sistemas operacionais que utilizam paginação.

O processador entra no modo de operação *Undefined* quando tenta executar uma palavra que não contém um código válido de instrução. O modo de operação *Supervisor* é o modo em que o processador entra ao executar uma instrução de chamada ao sistema operacional (similar ao que ocorre no Faíska).

1.2 Registradores

O processador ARM tem 37 registradores, mas apenas 17 (ou 18, em alguns modos de operação) são acessíveis a cada momento. Dos registradores acessíveis, 13 são registradores de propósito geral (`r0` a `r12`). Os outros quatro registradores têm funções específicas:

- `sp` (do inglês *stack pointer*), apontador de pilha, similar ao registrador homônimo do Faíska, também acessado pelo nome `r13`.

- `lr` (do inglês *link register*), registrador de ligação, também acessado pelo nome `r14`. Como veremos, esse registrador recebe o endereço de retorno em chamadas de procedimento.

- `pc` (do inglês *program counter*), contador de programa, também acessado pelo nome `r15`. Similar ao registrador `ip` do Faíska, indica o endereço da próxima instrução a ser executada.

- `CPSR` (do inglês *current program status register*, registrador de estado corrente do programa), similar ao registrador de bits de estado do Faíska.

Os registradores acessíveis em um dado momento formam o "banco" de registradores disponíveis ao programador. O banco de registradores, em momentos diferentes de execução, é constituído por diferentes registradores físicos. O modo de operação corrente determina a composição dos registradores que formam o banco. A Figura 1.1 ilustra a composição do banco de registradores para cada modo de operação do processador. Na Figura, os registradores estão numerados de 0 a 36 (canto superior direito de cada registrador), indicando os 37 registradores físicos.

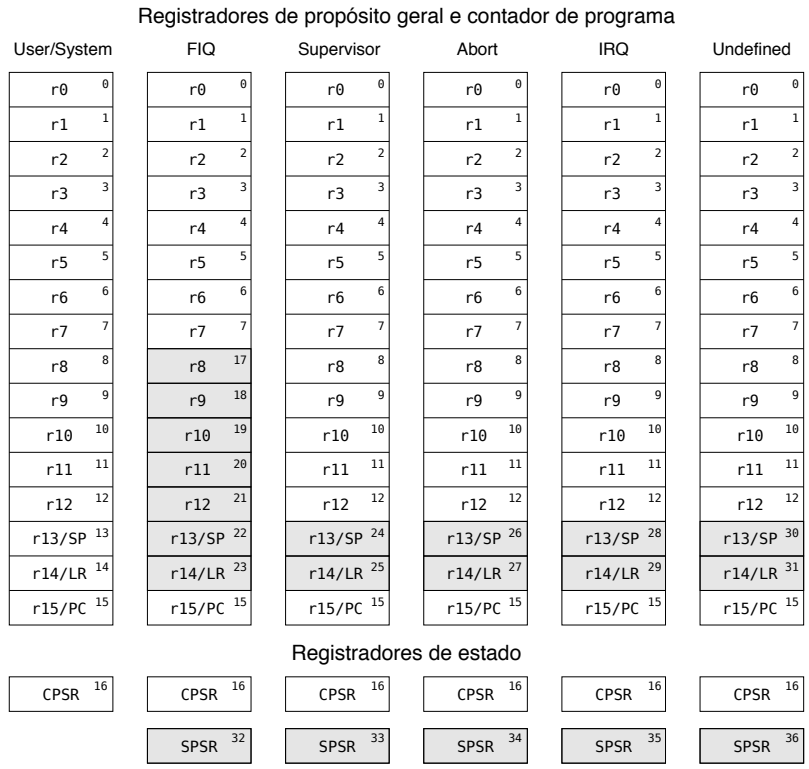

Figura 1.1: Mapa dos registradores ARM.

Os bancos de registradores nos modos de operação *User* e *System* são compostos pelo mesmo conjunto de registradores. No modo de operação *FIQ*, os registradores de `r0` a `r7`, e o registrador `15`, também são os mesmos utilizados no modo *User/System*. Mas os registradores de `r8` a `r14` são fisicamente distintos dos registradores com esses nomes no modo *User/System*. O modo *FIQ* também utiliza o mesmo registrador de estado que o modo *User/System*, mas um registrador fisicamente distinto armazena os bits de estado do programa inter-

rompido (registrador `SPSR`, abreviatura do inglês *saved program status register*). Como os registradores são distintos dos registradores do modo *User*, quando ocorre uma interrupção do tipo *FIQ*, se o tratador da interrupção utilizar apenas os registradores de `r8` a `r14`, nenhum registrador (nem mesmo o registrador de estado) necessita ser salvo na pilha, agilizando o tratamento da interrupção.

Os modos de operação *Supervisor*, *Abort*, *IRQ* e *Undefined* compartilham os registradores de `r0` a `r12`, e `r15`, com o modo *User/System*, mas têm registradores `r13` e `r14` fisicamente distintos. Nesses modos o registrador de estado `SPSR` armazena o registrador de estado do programa que foi interrompido.

1.2.1 Registradores de estado

Os registradores de estado mantêm bits de estado que indicam o resultado de operações lógicas e aritméticas, controlam interrupções e o modo de operação do processador. A Figura 1.2 mostra os bits de estado nos registradores de estado do processador ARM.

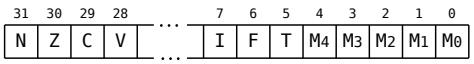

Figura 1.2: Bits de estado.

Os bits de estado são divididos em bits de condição (bits 28 a 31 do registrador de estado) e bits de controle (bits 0 a 7 do registrador de estado).

Os bits de condição são similares aos bits de condição do Faíska. Eles são ligados ou desligados como resultado de operações aritméticas e lógicas, e podem também ser alterados por instruções especiais. Os bits de condição no ARMv7 são:

- N: sinal. Cópia do bit mais significativo do resultado; considerando aritmética com sinal, se N igual a zero, o resultado é maior ou igual a zero. Se N igual a 1, resultado é negativo.
- Z: zero. Ligado se o resultado foi zero, desligado caso contrário.
- C: vai-um (*carry*). Ligado se a operação causou vai-um (*carry-out*) ou empresta-um (*carry-in*), desligado caso contrário.
- V: estouro de campo (*overflow*). Ligado se ocorreu estouro de campo; calculado como o ou-exclusivo entre o vai-um do bit mais significativo do resultado e o vai-um do segundo bit mais significativo do resultado.

Os bits de controle determinam condições de operação do processador. No ARM eles são:

- I: interrupção. Quando I é igual a 1, interrupções do tipo IRQ estão desabilitadas.
- F: interrupção rápida. Quando F é igual a 1, interrupções do tipo FIQ estão desabilitadas.
- T: estado *Arm/Thumb*. O processador ARMv7 pode executar dois conjuntos de instruções: o conjunto normal, em que instruções têm 32 bits (denominado *arm*), e um conjunto reduzido (denominado *thumb*), em que instruções têm 16 bits, permitindo um código mais compacto. O bit de controle T reflete o estado em que o processador está operando; quando T é igual a 1, o processador está executando no estado *thumb*, quando T é igual a 0 o processador está executando no estado *arm*.
- M[4:0]: modo de operação. Determinam o modo de operação, conforme descrito na Tabela 1.2.

Os demais bits dos registradores de estado (bits 8 a 27) não são utilizados, mas são reservados para uso futuro. Seu programa não deve alterar esses bits, para evitar problemas de compatibilidade com novas versões do processador.

M[4:0]	Modo de Operação
`10000`	*User*
`10001`	*FIQ*
`10010`	*IRQ*
`10011`	*Supervisor*
`10111`	*Abort*
`11011`	*Undefined*
`11111`	*System*

Tabela 1.2: Modos de operação definidos pelos bits de controle M[4:0].

1.2.2 *Uso de registradores em instruções*

Todos os registradores de `r0` a `r14` podem ser usados como operandos em instruções. A maioria das instruções também permite que o registrador `r15` (`pc`) seja usado como operando. Os registradores de estado podem ser carregados de e para registradores de propósito geral com instruções específicas. E quando o processador está executando em um modo privilegiado, é possível também carregar ou armazenar os registradores do modo *User*.

1.3 *Instruções*

Todas as instruções do processador ARM são codificadas em uma palavra de 32 bits, e praticamente todas executam em apenas um ciclo do relógio.

As instruções devem ter endereços múltiplos de quatro (alinhadas por palavras). Assim, os dois bits menos significativos do registrador `pc` são sempre zero. Tentativa de acesso a uma instrução não alinhada gera uma exceção.

Uma característica marcante no processador ARMv7 é que (praticamente) *todas* as instruções são executadas condicionalmente. Nas instruções, os quatro bits mais significativos (bits 28 a 31) são usados para codificar o *campo de condição* da instrução. As instruções são

executadas condicionalmente, dependendo dos bits especificados no campo de condição da instrução e dos bits do registrador de estado CPSR. Como o campo de condição tem quatro bits, é possível especificar dezesseis condições distintas. No entanto, na prática, devido à codificação das instruções, um dos valores do campo de condição (1111) não pode ser usado, de forma que podemos especificar quinze condições, descritas na Tabela 1.3.

Tabela 1.3: Execução condicional

Código	Sufixo	Condição	Descrição
0000	EQ	$Z = 1$	Igual
0001	NE	$Z = 0$	Diferente
0010	CS	$C = 1$	Maior ou igual, valor sem sinal
0011	CC	$C = 0$	Menor, valor sem sinal
0100	MI	$N = 1$	Negativo
0101	PL	$N = 0$	Positivo ou zero
0110	VS	$V = 1$	Estouro de campo, valor com sinal
0111	VC	$V = 0$	Não estouro de campo, valor com sinal
1000	HI	$C = 1 \wedge Z = 0$	Maior, valor sem sinal
1001	LS	$C = 0 \vee Z = 1$	Menor ou igual, valor sem sinal
1010	GE	$N = V$	Maior ou igual, valor com sinal
1011	LT	$N \neq V$	Menor, valor com sinal
1100	GT	$Z = 0 \wedge (N = V)$	Maior, valor com sinal
1101	LE	$Z = 1 \vee (N \neq V)$	Menor ou igual, valor com sinal
1110	AL	–	Sempre

A Tabela 1.3 mostra, para cada código, um sufixo. Os sufixos são utilizados em comandos de linguagem de montagem para especificar o campo de condição da instrução correspondente, codificado nos bits 28 a 31. Um comando sem sufixo é o mesmo que um comando com o sufixo AL, significando que a instrução correspondente é sempre executada. No momento da execução de uma instrução, o processador verifica se os bits de condição do registrador de estado CPSR estão de acordo com a condição codificada nos bits 28 a 31 da instrução. Se a condição é satisfeita, a instrução é executada; caso contrário a instrução não é executada.

Uma instrução de desvio incondicional, cujo comando em linguagem de montagem é B (do inglês *branch*, desvio), como por exemplo

```
b longe
```

se torna um desvio condicional pela adição de um sufixo, como EQ:

```
beq longe
```

Nesse caso, o desvio é tomado somente se o bit de condição Z estiver ligado. O mesmo ocorre com todas as outras instruções. Como outro exemplo, uma instrução de transferência de dados, como cópia entre registradores, cujo comando em linguagem de montagem é MOV

```
mov r1,r2
```

se torna uma instrução condicional pela adição de um sufixo de condição:

```
movne r1,r2
```

Nesse caso, como o sufixo usado é NE, no momento da execução, se o bit de condição Z estiver desligado a instrução é executada e o valor do registrador `r2` é copiado para o registrador `r1`. Se o bit de condição Z estiver ligado, a instrução não é executada.

O fato de que toda instrução pode ser executada condicionalmente permite o desenvolvimento de códigos muito compactos e eficientes, e tornam desvios condicionais para endereços maiores do que o endereço corrente praticamente desnecessários.

1.3.1 Pipeline

Para conseguir que, em regime, uma instrução seja executada a cada ciclo de relógio, a arquitetura ARM utiliza uma técnica de implementação conhecida como *pipeline* de execução. Nessa implementação, a execução de uma instrução é dividida em *estágios*, cada um responsável por uma tarefa independente. A *pipeline* do processador ARMv7 tem três estágios: busca da instrução, decodificação e execução. Como cada estágio executa uma tarefa independente, eles não precisam ser executados sequencialmente. Na implementação em *pipeline* os estágios são executados concorrentemente, como ilustra o diagrama da Figura 1.3, em que os estágios são designados por B (busca), D (decodificação) e X (execução).

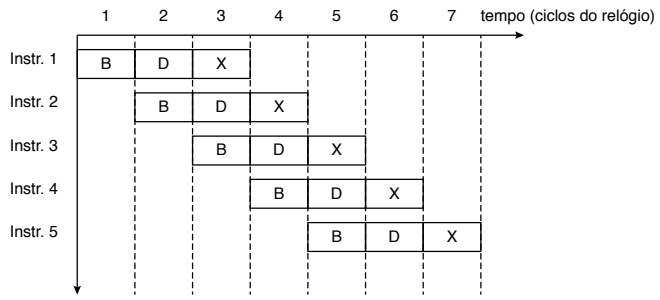

Figura 1.3: Execução em *pipeline*, com três estágios.

No ciclo de relógio 1, apenas a Instrução 1 executa, utilizando o estágio B. No ciclo de relógio 2, a Instrução 1 passa a utilizar o estágio D, enquanto a Instrução 2 utiliza o estágio B. A partir do ciclo 3, todos os estágios da *pipeline* estão ocupados, cada um executando uma instrução diferente. O resultado é que, se não houver instruções de desvios, o processador termina de executar uma instrução a cada ciclo do relógio. Quando há um desvio, a *pipeline* é esvaziada, tem que

recomeçar, e demora três ciclos para voltar a ficar em regime, o que retarda a execução. Por isso, o fato de todas as instruções poderem ser condicionais tende a agilizar a execução de um programa, pois a *pipeline* está sempre cheia. Obviamente, se um grupo grande de instruções consecutivas não é executado por conta da execução condicional, a penalidade de não execução pode chegar a se sobrepor à penalidade de usar um desvio condicional e causar um esvaziamento do *pipeline*. Essa é uma escolha que deve ser feita pelo programador, caso a caso.

É interessante notar que, devido à execução em *pipeline*, quando uma instrução está no estágio de execução, o registrador contador de programa `pc` aponta na realidade para a instrução que está no estágio de busca de instrução. Ou seja, quando uma instrução no endereço K está sendo executada, o registrador `pc` tem o valor $K + 8$. Isso é importante no cálculo do endereço alvo de desvios, por exemplo, como veremos na seção **??**.

1.4 Unidade de deslocamento

O processador ARMv7 inclui em sua Unidade Lógica e Aritmética uma unidade de deslocamento (em inglês *barrel shifter*) que pode ser acionada para efetuar operações de deslocamento ou rotação em um operando *antes* de o operando ser usado em instruções aritméticas, lógicas ou de transferência de dados. Na maioria das instruções o deslocamento é feito dentro do mesmo ciclo de relógio em que a instrução é executada.

Já que qualquer instrução pode utilizar a unidade de deslocamento, o processador ARMv7 não inclui instruções específicas de deslocamento e rotação, como ocorre o Faíska, já que uma instrução de transferência de dados pode ser usada para efetuar deslocamentos ou rotações, utilizando a unidade de deslocamento em um de seus operando.

O uso de descolamentos e rotações em qualquer instrução permite uma grande versatilidade na construção de constantes utilizadas em endereçamento imediato, no cálculo de endereços de elementos de vetores, além de permitir multiplicação por um valor imediato. Veremos, nos próximos capítulos, exemplos de uso da unidade de deslocamento.

1.5 Convenções do montador

Vamos utilizar nos exemplos o montador gnu-arm, por ser software livre disponível para diversos sistemas operacionais.

1.5.1 Diretivas

As diretivas da linguagem de montagem Arm são muito similares às diretivas da linguagem de montagem Faíska:

- Comentários são iniciados pelo caractere '@' e se estendem até o final da linha.
- Para modificação do ponto de montagem utilizamos a diretiva .ORG:

  `.org` *expressão_inteira* [@ *comentário*]

  que altera o ponto de montagem corrente para o valor *expressão_inteira*.
- Para definição de constantes utilizamos a diretiva .EQU:

  `.equ` *nome* *expressão_inteira*

  que associa o valor da *expressao_inteira* ao símbolo *nome*.
- Para reservar espaço em memória, sem inicialização, usamos a diretiva .SKIP:

  [*rótulo:*] `.skip` [*expressão_inteira*] [@ *comentário*]

  que reserva *expressão_inteira* bytes de memória; se *rótulo* é definido, é associado ao endereço do primeiro byte reservado.
- Para reservar e inicializar espaço em memória, sem inicialização, usamos as diretivas .BYTE (reserva e inicialização de bytes) e .WORD (reserva e inicialização de palavras):

  [*rótulo:*] `.byte` [*lista_de_valores*] [@ *comentário*] [*rótulo:*] `.word` [*lista_de_valores*] [@ *comentário*]

  em que *lista_de_valores* é uma lista de valores de 8 ou 32 bits, separada por vírgulas.
- Para reservar e inicializar uma cadeia de caracteres usamos a diretiva .ASCII:

  [*rótulo:*] `.ascii` "*cadeia_de_caracteres*" [@ *comentário*]

  em que *cadeia_de_caracteres* é uma cadeia de caracteres limitada por aspas duplas.
- Como instruções no ARM devem ser montadas em endereços multiplos de quatro (limites de palavras), muitas vezes é necessário "alinhar" dados e instruções. Para isso utilizamos a diretiva .ALIGN:

`.align` *expressão_inteira*

que altera o ponto de montagem para o próximo endereço divisível por $2^{expressao\_inteira}$.

O Exemplo 1.1 ilustra as diretivas da linguagem de montagem ARM.

Exemplo 1.1: Uso de diretivas.

```
1  @ definição de constante
2            .equ MAXVAL, 256
3
4  @ alteração do ponto de montagem
5            .org   0x1000
6
7  @ reserva de espaço sem inicialização
8  vetor:    .skip  0x200          @ reserva 512 bytes
9  um:       .skip  1              @ reserva um byte;
10
11 @ ponto de montagem aqui é 0x1201
12 @ alinha em palavra
13           .align 2              @ alinha em endereço múltiplo de 4 (2^2)
14 @ ponto de montagem aqui é 0x1204
15
16 @ reserva de espaço e inicialização
17 var1:     .byte  0x01           @ um byte, valor hexadecimal (8 bits)
18 var2:     .word  1000,2000      @ duas palavras (32 bits cada), valores decimais
19 mensagem: .ascii "uma linha\n"  @ cadeia de caracteres
```

1.5.2 Operandos com endereçamento imediato

No montador gnu-arm, valores imediatos utilizados como operandos devem ser prefixados com o caractere '#', como por exemplo no comando carrega registrador com valor imediato:

```
mov r5, #0x5
```

2
Transferência entre um registrador e memória

O processador ARM usa uma arquitetura em que acessos à memória são feitos através de instruções específicas, como carrega registrador de memória e armazena registrador de memória. Instruções de processamento de dados não fazem acesso a operandos na memória (por exemplo não é possível somar o valor de um registrador ao valor de uma palavra na memória com uma instrução de processamento de dados).

As instruções de transferência entre registrador e memória do ARM podem transferir uma única palavra (ou byte) de ou para um registrador, ou um grupo de palavras (ou bytes) de ou para um grupo de registradores. Neste capítulo vamos estudar instruções e modos de endereçamento para transferir um único valor (palavra ou byte) de ou para um registrador. As instruções que transferem grupos de palavras serão vistas no Capítulo **??**.

A Figura2.1 mostra a codificação das instruções de carrega registrador com palavra de memória e armazena registrador em palavra de memória, cujos comandos em linguagem de montagem são respectivamente LDR (do inglês *load register* e STR (do inglês *store register*).

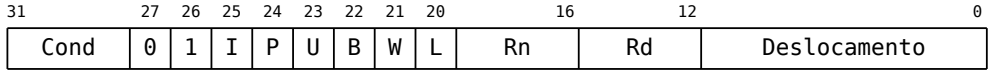

Figura 2.1: Codificação das instruções LDR e STR.

Na Figura 2.1, o campo Cond é o campo de condição, que controla a execução condicional da instrução, conforme descrito na Seção 1.3. Na instrução LDR o campo Rd indica o *registrador destino* (onde o valor da palavra de memória é carregado). Na instrução STR o campo Rd indica o *registrador fonte* (valor que é armazenado na memória). Em ambas as instruções, o campo Rn indica o *registrador base*, que contém o endereço base do operando, utilizado no cálculo do endereço efetivo do operando, que varia de acordo com o modo de endereçamento utilizado pela instrução.

O formato geral do comando LDR em linguagem de montagem é

```
ldr   Rd, <endereço>
```

O formato geral do comando STR em linguagem de montagem é

```
str   Rd, <endereço>
```

Nos dois formatos acima, <*endereço*> assume diferentes formas dependendo do modo de endereçamento.

O processador ARM possui um número muito maior de modos de endereçamento do que o Faíska. Para as instruções LDR e STR memória os modos de endereçamento se dividem em duas categorias: *pré-indexados* ou *pós-indexados*. Os nomes se referem à capacidade de incrementar ou decrementar o registrador base antes (pré) ou depois (pós) do cálculo do endereço efetivo.

2.1 *Endereçamento direto*

No modo de endereçamento direto, o endereço do operando é dado somente pelo campo Deslocamento da Figura 2.1, ou seja, o registrador base Rn não é utilizado.

O formato dos comandos LDR e STR em linguagem de montagem no modo de endereçamento direto é:

instr{*cond*}{B} Rd, *endereço*

onde um componente entre chaves (como {*cond*} indica um componente opcional e

- *instr* especifica a instrução e pode ser LDR ou STR.
- *cond* é um sufixo de condição da Tabela 1.3.
- B, se presente, faz com que o montador coloque o valor 1 no bit 22 da instrução (bit B na Figura 2.1), indicando que a operação deve carregar ou armazenar bytes; o byte menos significativo do registrador Rd é carregado ou armazenado (na instrução LDR os outros 24 bits do registrador são zerados). Se não presente, o bit B da instrução é 0 e a operação carrega ou armazena palavras.
- Rd é o registrador fonte (contém valor a ser armazenado na memória) para instruções LDR e destino (onde o valor lido da memória é armazenado) para instrucões STR.
- *endereço* é o endereço do operando na memória (normalmente especificado por um rótulo do programa). O montador monta no campo Deslocamento da instrução (Figura 2.1) a diferença entre o ponto corrente de montagem e o endereço do operando.

No momento da execução, o processador calcula o endereço efetivo adicionando o valor do registrador `pc` ao valor do campo Deslocamento com o sinal estendido para 32 bits. Um erro é gerado pelo montador se a diferença entre o endereço do operando e o ponto corrente de montagem não puder ser codificado em 12 bits.

O Exemplo 4.2 ilustra a sintaxe dos comandos LDR e STR com endereçamento direto.

Exemplo 2.1: Instruções LDR e STR com endereçamento direto.

```
1       .org 0x1000
2       ldr    r1, var1       @ carrega palavra em r1, incondicional
3       ldrgt  r2, var1       @ carrega palavra em r2, condicional
4       ldreqb r3, var2       @ carrega byte em r3, condicional
5
6       str    r4, var1       @ armazena r4 em  palavra, incondicional
7       strcc  r5, var1       @ armazena r5 em palavra, condicional
8       strleb r6, var2       @ armazena r6 em byte, condicional
9
10      ldr    r7, var_longe  @ gera erro, endereço muito distante
11
12      .org 0x2000
13  var1:
14      .word 0x12341234
15  var2:
16      .byte 0xff
17
18      .org 0x20000
19  var_longe:
20      .word 0x0
21
```

2.2 *Modo de endereçamento indireto por registrador*

No modo de endereçamento indireto por registrador, o endereço do operando é dado pelo valor do registrador base Rn.

O formato dos comandos LDR e STR em linguagem de montagem no modo de endereçamento indireto por registrador é:

instr{*cond*}{B} Rd, [Rn]

onde um componente entre chaves (como {*cond*} indica um componente opcional e

- *instr* especifica a instrução e pode ser LDR ou STR.
- *cond* é um sufixo de condição da Tabela 1.3.
- B, se presente, faz com que o montador coloque o valor 1 no bit 22 da instrução (bit B na Figura 2.1), indicando que a operação deve carregar ou armazenar bytes; o byte menos significativo do registrador Rd é carregado ou armazenado (na instrução LDR os

outros 24 bits do registrador são zerados). Se não presente, o bit B da instrução é 0 e a operação carrega ou armazena palavras.

- Rd é o registrador fonte para instruções LDR e destino para instruções STR.
- Rn é o registrador base. O endereço efetivo do operando na memória é dado pelo valor do registrador Rn.

O Exemplo 2.2 ilustra a sintaxe dos comandos LDR e STR com endereçamento indireto por registrador.

```
1       ldr    r1, [r0]       @ carrega palavra em r1, incondicional
2       ldrmi  r2, [r8]       @ carrega palavra em r2, condicional
3       ldrplb r3, [r9]       @ carrega byte em r3, condicional
4
5       str    r4, r[10]      @ armazena r4 em  palavra, incondicional
6       strlt  r5, r[11]      @ armazena r5 em palavra, condicional
7       strgtb r6, r[12]      @ armazena r6 em byte, condicional
```

Exemplo 2.2: Instruções LDR e STR com endereçamento indireto por registrador.

2.3 *Modos de endereçamento pré-indexados*

Nos modos de endereçamento pré-fixados, o endereço efetivo é formado pelo valor do registrador base mais (ou menos) um *offset*, codificado no campo Offset da instrução (Figura 2.1). A forma do cálculo desse *offset* dá origem a diferentes modos de endereçamento. O *offset* pode ser dado por:

- um valor imediato, constante, dando origem ao modo de endereçamento *indireto por registrador mais constante*.
- o valor de um registrador, dando origem ao modo de endereçamento *indireto por registrador base e registrador índice*.
- o valor de um registrador deslocado com o auxílio da unidade de deslocamento (*barrel shifter*), dando origem ao modo de endereçamento *indireto por registrador e registrador índice escalado*.

Nos modos de endereçamento pré-indexados, há ainda a possibilidade de atualizar o valor do registrador base com o valor do endereço efetivo, dessa forma fazendo com que o registrador avance (ou recue) automaticamente por um valor controlado, dentro do mesmo ciclo de instrução, permitindo uma flexibilidade e eficiência muito grande para implementação de comandos de repetição e para varredura de estruturas de dados sequenciais (como vetores, por exemplo).

2.3.1 *Endereçamento indireto por registrador mais constante*

O formato dos comandos LDR e STR em linguagem de montagem no modo de endereçamento indireto por registrador mais constante é:

instr{*cond*}{B} Rd, [Rn, #*expr*12] {!}

onde um componente entre chaves (como {*cond*} indica um componente opcional e

- *instr* especifica a instrução e pode ser LDR ou STR.
- *cond* é um sufixo de condição da Tabela 1.3.
- B, se presente, faz com que o montador coloque o valor 1 no bit 22 da instrução (bit B na Figura 2.1), indicando que a operação deve carregar ou armazenar bytes; o byte menos significativo do registrador Rd é carregado ou armazenado (na instrução LDR os outros 24 bits do registrador são zerados). Se não presente, o bit B da instrução é 0 e a operação carrega ou armazena palavras.
- Rd é o registrador fonte para instruções LDR e destino para instruções STR.
- Rn é o registrador base.
- *expr*12 é uma expressão que o montador codifica no campo *Deslocamento* da instrução (Figura 2.1). O endereço efetivo do operando na memória é dado pelo valor do registrador Rn mais o valor de *expr*12 com o sinal estendido para 32 bits.
- ! é um indicador se o registrador base deve ser atualizado. Se presente, o montador monta o valor 1 no bit 21 da instrução (bit W, do inglês *write-back*, na Figura 2.1), indicando que o registrador Rn deve ter o valor atualizado, ao final da instrução, com o valor do endereço efetivo. Se ausente, o valor do registrador Rn permanece inalterado.

O Exemplo 2.3 ilustra a sintaxe dos comandos LDR e STR com endereçamento indireto por registrador mais constante, pré-indexado.

2.3.2 *Endereçamento indireto por registrador base e registrador índice*

O formato dos comandos LDR e STR em linguagem de montagem no modo de endereçamento pré-indexado indireto por registrador base e registrador índice é:

instr{*cond*}{B} Rd, [Rn, {+|-} Rm] {!}

onde um componente entre chaves (como {*cond*} indica um componente opcional e

```
1       ldr    r1, [r0,#1]      @ carrega palavra em r1, incondicional
2       ldrmi  r2, [r8,#4]!     @ carrega palavra em r2, condicional
3                               @ r2 é incrementado de 4
4       ldrplb r3, [r9,#2]      @ carrega byte em r3, condicional
5
6       str    r4, [r10,#-2]!   @ armazena r4 em  palavra, incondicional
7                               @ r10 é decrementado de 2
8       strlt  r5, [r11,#4]     @ armazena r5 em palavra, condicional
9       strgtb r6, [r12,#1]!    @ armazena r6 em byte, condicional
10                              @ r12 é incrementado de 1
```

Exemplo 2.3: Instruções LDR e STR com modo de endereçamento pré-indexado indireto por registrador mais constante.

- *instr* especifica a instrução e pode ser LDR ou STR.
- *cond* é um sufixo de condição da Tabela 1.3.
- B, se presente, faz com que o montador coloque o valor 1 no bit 22 da instrução (bit B na Figura 2.1), indicando que a operação deve carregar ou armazenar bytes; o byte menos significativo do registrador Rd é carregado ou armazenado (na instrução LDR os outros 24 bits do registrador são zerados). Se não presente, o bit B da instrução é 0 e a operação carrega ou armazena palavras.
- Rd é o registrador fonte para instruções LDR e destino para instruções STR.
- Rn é o registrador base.
- Rm é o registrador índice.
- +|- indica se o registrador índice deve ser adicionado ou subtraído do registrador base para formar o endereço efetivo do operando. O sinal + é opcional.
- ! é um indicador se o registrador base deve ser atualizado. Se presente, o montador monta o valor 1 no bit 21 da instrução (bit W, do inglês *write-back*, na Figura 2.1), indicando que o registrador Rn deve ter o valor atualizado, ao final da instrução, com o valor do endereço efetivo. Se ausente, o valor do registrador Rn permanece inalterado.

O Exemplo 2.4 ilustra a sintaxe dos comandos LDR e STR com endereçamento pré-fixado indireto por registrador base e registrador índice.

2.3.3 *Endereçamento indireto por registrador base e registrador índice escalado*

Para este modo de endereçamento o campo Offset da instrução (Figura 2.1) codifica um registrador índice e o tipo da operação de deslocamento utilizada, conforme a Figura 2.2.

```
1       ldr    r1, [r0,r2]      @ carrega palavra em r1, incondicional
2                               @ endereço efetivo é r0+r2
3                               @ r0 não é alterado
4       ldrvs  r2, [r8,-r3]!    @ carrega palavra em r2, condicional
5                               @ endereço efetivo é r8-r3
6                               @ r8 tem valor alterado ao final
7       ldrvcb r3, [r9,+r10]    @ carrega byte em r3, condicional
8                               @ endereço efetivo é r9+r10
9                               @ r9 não é alterado
10      str    r4, [r10,-r0]!   @ armazena r4 em  palavra, incondicional
11                              @ endereço efetivo é r10-r0
12                              @ r10 tem valor alterado ao final
13      strhi  r5, [r11,r0]     @ armazena r5 em palavra, condicional
14                              @ endereço efetivo é r11+r0
15                              @ r11 não é alterado
16      strlsb r6, [r12,r13]!   @ armazena r6 em byte, condicional
17                              @ endereço efetivo é r12+r13
18                              @ r12 tem valor alterado ao final
```

Exemplo 2.4: Instruções LDR e STR com endereçamento indireto por registrador mais constante, pré-indexado.

O formato dos comandos LDR e STR em linguagem de montagem no modo de endereçamento pré-indexado indireto por registrador base e registrador índice escalado é:

instr{*cond*}{B} Rd, [Rn, {+|-} Rm, *shift #expr5*] {!}

onde um componente entre chaves (como {*cond*} indica um componente opcional e

- *instr* especifica a instrução e pode ser LDR ou STR.
- *cond* é um sufixo de condição da Tabela 1.3.
- B, se presente, faz com que o montador coloque o valor 1 no bit 22 da instrução (bit B na Figura 2.1), indicando que a operação deve carregar ou armazenar bytes; o byte menos significativo do registrador Rd é carregado ou armazenado (na instrução LDR os outros 24 bits do registrador são zerados). Se não presente, o bit B da instrução é 0 e a operação carrega ou armazena palavras.
- Rd é o registrador fonte para instruções LDR e destino para instruções STR.
- Rn é o registrador base.
- Rm é o registrador índice.
- +|- indica se o valor registrador índice, após escalado, deve ser adicionado ou subtraído do registrador base para formar o endereço efetivo do operando. Codificado no bit 23 da instrução (bit U, do inglês *Up or down*) na Figura 2.1). O sinal + é opcional.

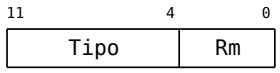

Figura 2.2: Codificação do campo Offset para o modo de endereçamento indireto por registrador base e registrador índice escalado.

- *shift* indica o tipo de operação de deslocamento realizada no valor do registrador `Rm` antes de ser utilizado no cálculo do endereço efetivo. O tipo pode ser:
  - LSL: deslocamento lógico para a esquerda (abreviatura do inglês *Logical Shift Left*).
  - LSR: deslocamento lógico para a direita (abreviatura do inglês *Logical Shift Right*).
  - ASR: deslocamento aritmético para a direita (abreviatura do inglês *Arithmetic Shift Right*).
  - ROR: rotação para a direita (abreviatura do inglês *Rotate Right*).
- *expr5* indica a quantidade do deslocamento a ser aplicado no registrador `Rm`.
- `!` é um indicador se o registrador base deve ser atualizado. Se presente, o montador monta o valor 1 no bit 21 da instrução (bit W, do inglês *write-back*, na Figura 2.1), indicando que o registrador `Rn` deve ter o valor atualizado, ao final da instrução, com o valor do endereço efetivo. Se ausente, o valor do registrador `Rn` permanece inalterado.

O Exemplo 2.5 ilustra a sintaxe dos comandos LDR e STR com endereçamento pré-fixado indireto por registrador base e registrador índice escalado.

Exemplo 2.5: Instruções LDR e STR com endereçamento pré-fixado indireto por registrador base e registrador índice escalado.

```
1       ldr    r1, [r0,r2,lsl#1]    @ carrega palavra em r1, incondicional
2                                   @ endereço efetivo é r0+(r2<<1)
3                                   @ r0 não é alterado
4       ldrgt  r2, [r8,-r3,lsr#2]!  @ carrega palavra em r2, condicional
5                                   @ endereço efetivo é r8-(r3>>2)
6                                   @ r8 tem valor alterado ao final
7       ldrltb r3, [r9,+r10,asr#4]  @ carrega byte em r3, condicional
8                                   @ endereço efetivo é r9+(r10 >>arit. 4)
9                                   @ r9 não é alterado
10      str    r4, [r10,-r0,ror#5]! @ armazena r4 em  palavra, incondicional
11                                  @ endereço efetivo é r10-(r0 rotacionado 5 posições)
12                                  @ r10 tem valor alterado ao final
```

2.4 *Modos de endereçamento pós-indexados*

Os modos de endereçamento pós-indexados são muito similares aos modos de endereçamento pré-indexados. A diferença é que, nos modos de endereçamento pós-indexados, o endereço efetivo do operando é o valor do registrador base, mas após o acesso à memória, o registrador base é (sempre) atualizado, adicionando ou subtraindo o valor do operando indexador (constante, registrador índice ou

registrador índice escalado). Como o registrador base é sempre atualizado no caso de endereçamento pós-indexado (pois não teria sentido ter o operando pós-indexado se não fosse para atualização do registrador base), o indicador "!" não é utilizado, e o bit 21 da instrução (bit W, do inglês *write-back*, na Figura 2.1), tem sempre o valor 1.

O formato dos comandos LDR e STR em linguagem de montagem nos modos de endereçamento pós-indexados é descrito a seguir, onde um componente entre chaves (como {*cond*} indica um componente opcional e os outros campos são iguais aos apresentados para os modos de endereçamento pré-indexados.

Endereçamento indireto por registrador mais constante

instr{*cond*}{B} Rd, [Rn], #*expr*12

Endereçamento pré-indexado indireto por registrador base e registrador índice

instr{*cond*}{B} Rd, [Rn], {+|-} Rm

Endereçamento pré-fixado indireto por registrador base e registrador índice escalado:

instr{*cond*}{B} Rd, [Rn], {+|-} Rm, *shift #expr5*

O montador diferencia entre os modos pré-fixado e pós-fixado pela sintaxe do comando, que é ligeiramente diferente (note a posição dos operandos em relação às chaves). O Exemplo 2.6 ilustra as instruções LDR e STR com modos de endereçamento pós-indexados.

```
1  @ modo de endereçamento registrador base e constante
2       ldrplb r3,[r9],#2         @ carrega byte em r3, condicional
3
4       str    r4,[r10],#-2       @ armazena r4 em  palavra, incondicional
5
6  @ modo de endereçamento registrador base e registrador índice
7       ldrvs  r2,[r8],-r3        @ carrega palavra em r2, condicional
8                                 @ endereço efetivo é r8-r3
9                                 @ r8 tem valor alterado ao final
10      ldrvcb r3,[r9],+r10       @ carrega byte em r3, condicional
11                                @ endereço efetivo é r9+r10
12                                @ r9 não é alterado
13      str    r4,[r10],-r0       @ armazena r4 em  palavra, incondicional
14                                @ endereço efetivo é r10-r0
15                                @ r10 tem valor alterado ao final
16
17 @ modo de endereçamento registrador base e registrador índice escalado
18
19      strhi  r5,[r11],r0        @ armazena r5 em palavra, condicional
20
21      ldr    r1,[r0],r2,lsl#1   @ carrega palavra em r1, incondicional
22                                @ endereço efetivo é r0
23                                @ r0 é alterado para r0+(r2<<1)
24      ldrgt  r2,[r8],-r3,lsr#2  @ carrega palavra em r2, condicional
25                                @ endereço efetivo é r8-(r3>>2)
26                                @ r8 tem valor alterado ao final
27      strltb r3,[r9],+r10,asr#4 @ armazena r3 em byte, condicional
28                                @ endereço efetivo é r9
29                                @ r9 é alterado para r9+(r10 >>arit. 4)
```

Exemplo 2.6: Instruções LDR e STR com modos de endereçamento pós-indexados.

Problema 2.1. Traduzir para linguagem de montagem ARM o seguinte trecho de programa C:

```
1  char b,vetor_char[100];
2  int a,i,vetor_int[100];
3  ...
4      vetor_char[i]=b;
5      vetor_int[i]=a;
6  ...
```

O Exemplo 2.7 mostra uma solução para o Problema 2.1. Note que para carregar o endereço da variável `vetor_char` utilizamos uma instrução LDR cujo operando é uma palavra de memória (`ender_vetor_char`) que contém o endereço da variável. Isso porque, como todas as intruções no ARM são codificadas em uma palavra, não há espaço na instrução para que qualquer valor imediato seja codificado. Dessa forma, como veremos no Capítulo **??**, muitas vezes é necessário utilizar essa abordagem para carregar uma constante em um registrador.

Problema 2.2. Traduzir para linguagem de montagem ARM o seguinte trecho de programa C:

```
1  int *p,*q;
2  ...
3    *p++=*--q;
```

Exemplo 2.7: Solução para o Problema 2.1.

```
vetor_int:
   .skip 100
a:
   .skip 4
i:
   .skip 4
vetor_char:
   .skip 100
b:
   .skip 1
   .align 2
ender_vetor_char:
   .word vetor_char
ender_vetor_int:
   .word vetor_int
@...
   ldr   r0,a                @ carrega valor de a
   ldr   r1,i                @ índice
   ldr   r3,ender_vetor_char  @ endereço de vetor_char
   str   r0,[r3,r1]          @ armazena em vetor_char[i]
   ldr   r0,b                @ carrega valor de b
   ldr   r3,ender_vetor_int   @ endereço base
   str   r0,[r3,r1,LSL#2]    @ armazena em vetor_int[i]


...
```

O Exemplo 2.8 mostra uma solução para o Problema 2.2.

Exemplo 2.8: Solução para o Problema 2.2.

```
p:
   .skip 4
q:
   .skip 4
...
   ldr   r1,q                @ carrega valor de q
   ldr   r0,[r1,#-4]!        @ carrega valor para atribuição,
                             @ pré-decremento de q com write-back
   ldr   r2,p                @ carrega valor de p
   str   r0,[r2],#4          @ armazena valor da atribuição,
                             @ pós-incremento de p com write-back
   str   r2,p                @ armazena valor atualizado de p
   str   r1,q                @ armazena valor atualizado de q
```

2.5 *Exercícios*

2.5.1. Traduza para linguagem de montagem ARM o seguinte trecho de programa em C:

```
int *p,i,k;
...
  *p++=i;
  *++p=k;
...
```

2.5.2. Traduza para linguagem de montagem ARM o seguinte trecho de programa em C:

```
1  char b,vetor_char[100];
2  int a,i,vetor_int[100];
3  ...
4      b=vetor_char[i++];
5      a=vetor_int[i++];
```

3
Desvios e Processamento de dados

3.1 Desvios

No processador ARM a as instruções de desvio podem ter endereçamento imediato ou endereçamento indireto por registrador.

3.1.1 Desvios com endereçamento imediato

A Figura3.1 mostra a codificação das instruções de desvio com endereçamento imediato.

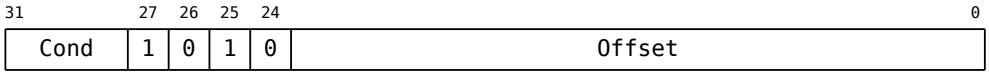

Figura 3.1: Codificação das instruções de desvio com endereçamento imediato.

O formato geral do comando de desvio com endereçamento imediato em linguagem de montagem é

`B`{*cond*} *endereço*

onde

- *cond* é um sufixo de condição da Tabela 1.3.
- *endereço* é um endereço que deve estar dentro do intervalo

$$[\texttt{pc} - (2^{24} - 1), \texttt{pc} + 2^{24}]$$

O endereço alvo do desvio é calculado relativo ao registrador contador de programa, `pc`. O montador calcula a diferença, em complemento de dois, do ponto de montagem e do endereço alvo. Como no processador ARM todos os endereços devem ser múltiplos de quatro, os dois bits menos significativos de um endereço são zero, e portanto não precisam ser armazenados no campo Offset. Assim, o montador desloca, de dois bits para a direita, o valor da diferença

encontrada entre o ponto de montagem e o endereço alvo, e monta o valor resultante no campo Offset.

No momento da execução, o processador extrai o valor do campo Offset, estende o bit de sinal para 32 bits, desloca esse valor que dois bits para a esquerda, adiciona com o valor corrente do registrador contador de programa e finalmente atualiza o valor do registrador contador de programa com o valor calculado. A pipeline é esvaziada e o processador inicia a execução da instrução no endereço alvo.

O Exemplo 5.1 ilustra a sintaxe do comando B.

```
1     bne  loop              @ desvio condicional
2     b    fim               @ desvio incondicional
3  loop:                    @ um rótulo
4     ...
5  fim:                     @ outro rótulo
6     ...
```

Exemplo 3.1: Instruções de desvio com endereçamento imediato.

3.1.2 *Desvios com endereçamento por registrador*

A Figura3.2 mostra a codificação das instruções de desvio com endereçamento por registrador.

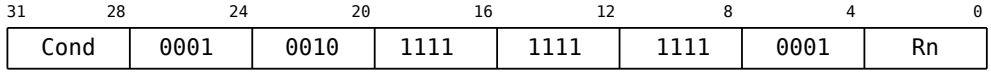

Figura 3.2: Codificação das instruções de desvio com endereçamento por registrador.

No momento da execução, se o campo de condição é satisfeito, o conteúdo do registrador Rd é copiado para o registrador pc.

O formato geral do comando de desvio com endereçamento imediato em linguagem de montagem é

BX{*cond*} Rd

onde

- *cond* é um sufixo de condição da Tabela 1.3.
- Rd é um registrador cujo valor será usado como endereço alvo.

O Exemplo 5.1 ilustra a sintaxe do comando B.

```
1     bxmi r10               @ desvio condicional
2     bx   r8                @ desvio incondicional
```

Exemplo 3.2: Instruções de desvio com endereçamento por registrador.

A instrução BX é utilizada também para passar para o modo de execução *THUMB*: se o bit menos significativo do registrador Rd é igual a 1, o processador passa a executar no modo *THUMB* a partir

do endereço alvo. Caso o bit menos significativo do registrador Rd seja igual a 0, o processador continua executando no modo *ARM* a partir do endereço alvo.

3.2 *Processamento de dados*

As instruções de processamento de dados do ARM englobam todas as instruções aritméticas, lógicas e de transferência entre registradores. Elas compartilham o mesmo formato básico de codificação, mostrado na Figura 3.3, embora tenham diferenças quanto à sintaxe dos comandos em linguagem de montagem.

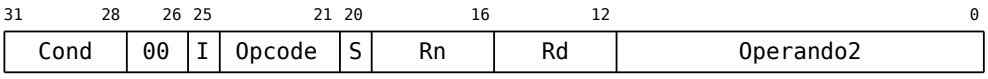

Figura 3.3: Codificação das instruções de processamento de dados.

Na Figura 3.3, o campo Cond é o campo de condição, que controla a execução condicional da instrução, conforme descrito na Seção 1.3.

O campo Opcode codifica a instrução; como esse campo tem 4 bits, há 16 instruções distintas, descritas na Tabela 3.1.

Tabela 3.1: Instruções de processamento de dados.

Opcode	Comando	Nome	Operação
0000	AND	E-lógico	Rd ← Rn ∧ Operando2
0001	EOR	Ou-exclusivo	Rd ← Rn ⊕ Operando2
0010	SUB	Subtração	Rd ← Rn − Operando2
0011	RSB	Subtração reversa	Rd ← Operando2 − Rn
0100	ADD	Adição	Rd ← Rn + Operando2
0101	ADC	Adição com vai-um	Rd ← Rn + Operando2 + C
0110	SBC	Subtração com empresta-um	Rd ← Rn − Operando2 + C − 1
0111	RSC	Subtração reversa com empresta-um	Rd ← Operando2 − Rn + C − 1
1000	TST	Testa bits	Rn ∧ Operando2
1001	TEQ	Testa equivalência	Rn ⊕ Operando2
1010	CMP	Comparação	Rn − Operando2
1011	CMN	Comparação negativa	Rn + Operando2
1100	ORR	Ou-lógico	Rd ← Rn ∨ Operando2
1101	MOV	Move registrador	Rd ← Operando2
1110	BIC	Desliga bit	Rd ← Rn ∧ *not* Operando2
1111	MVN	Move registrador negado	Rd ← *not* Operando2

Diferentemente do Faíska, em que toda instrução aritmética ou lógica atualiza os bits de condição, no ARM é possível escolher se a instrução de processamento de dados deve ou não atualizar os bits de condição. O bit S da instrução (Figura 3.3) determina se o processador deve ou não atualizar os bits de condição.

O bit I define como o campo Operando2 está codificado. Se I é zero, Operando2 é um registrador `Rm`, que pode ainda ser deslocado com a Unidade de Deslocamentos; nesse caso, o campo Operando2 é codificado como mostra a Figura 3.4.

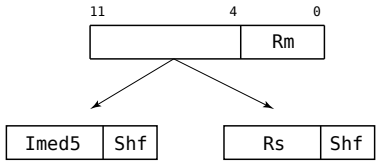

Figura 3.4: Codificação do campo Operando2 quando o operando é um registrador.

Os bits de 4 a 11 especificam o tipo de deslocamento e o a quantidade de bits que o registrador `Rm` deve ser deslocado (Shf). A quantidade de bits que o registrador `Rm` deve ser deslocado pode ser especificada como um valor imediato de cinco bits (Imed5) ou através de um outro registrador `Rs`.

Se o bit I é igual a 1, Operando2 é um valor imediato de oito bits, `Imed8`, que pode ainda ser deslocado ou rotacionado com a Unidade de Deslocamentos; nesse caso, o campo Operando2 é codificado como mostra a Figura 3.5.

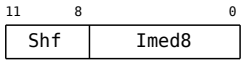

Figura 3.5: Codificação do campo Operando2 quando o operando é um valor imediato.

O campo Shf, tanto quando Operando2 é um valor imediato ou um registrador especifica o tipo de deslocamento a ser aplicado. Os tipos de deslocamento possíveis são os mesmos já vistos na Seção 2.3.3, cujos nomes em linguagem de montagem são mostrados na Tabela 3.2.

Comando	Operação
LSL	Deslocamento lógico para a esquerda
LSR	Deslocamento lógico para a direita
ASR	Deslocamento aritmético para a direita
ROR	Rotação para a direita

Tabela 3.2: Operações de deslocamento.

3.2.1 *Instruções de transferência entre registradores*

As instruções de transferência entre registradores são MOV (copia registrador) e MVN (copia registrador negado). Elas podem efetuar

- a carga de um valor imediato para um registrador,
- a cópia de um registrador para outro,
- o deslocamento de um registrador.

Os formatos dos comandos MOV e MVN em linguagem de montagem são:

instr{*cond*}{S} `Rd`, *Operando2*

onde

- *instr* especifica a instrução e pode ser MOV ou MVN.
- *cond* é um sufixo de condição da Tabela 1.3.

- S, se presente, faz com que o montador coloque o valor 1 no bit 20 da instrução (bit S na Figura 3.3), indicando que a operação deve atualizar o valor dos bits de condição N e Z de acordo com o valor resultante do registrador Rd. Se não presente, o bit S da instrução é 0 e a operação não altera o valor de nenhum bit de condição.
- Rd é o registrador destino para a operação.
- *Operando2* pode ser
  - um valor imediato; nesse caso a sintaxe em linguagem de montagem para Operando2 é #*expr*32. O montador procura encontrar um valor de oito bits e uma operação de deslocamento para montar respectivamente nos campos Imed8 e Shf da Figura 3.5 que produzam o valor *expr*32. Se não encontra, um erro é gerado (o valor *expr*32 não pode ser carregado de forma imediata em um registrador).
  - um registrador (possivelmente deslocado pela Unidade de deslocamento); nesse caso a sintaxe em linguagem de montagem para Operando2 é Rm {, *Tiposhift* Rs, onde *Tiposhift* é um dos comandos da Tabela 3.2.

O Exemplo 3.3 ilustra a sintaxe dos comando MOV e MVN.

Exemplo 3.3: Instruções MOV e MVN.

```
1       mov    r1,r2              @ cópia simples, incondicional
2                                 @ r1 <-- r2
3       mvn    r3,r4              @ cópia com negação, incondicional
4                                 @ r3 <-- -r4
5       mvneqs r10,#0             @ carga de registrador com valor imediato
6                                 @ r10 <-- -1
7                                 @ condicional; se executada, bits N=1 e Z=0
8       mov    r12,r8,lsr#2       @ cópia com deslocamento, incondicional
9                                 @ r12 <-- r8 / 4
10      movgt  r9,r9,lsl#2        @ deslocamento, condicional
11                                @ r9 <-- r9 * 4
12      movcc  r13,r10,ror r2     @ cópia com deslocamento, incondicional
13                                @ r9 <-- r10 rotacionado r2 bits
```

3.2.2 *Instruções que não armazenam o resultado*

As instruções CMP (compara), CMN (compara com negação), TEQ (testa equivalência) e TST (testa bits) não armazenam o resultado da operação em um registrador, conforme a descrição da operação na Tabela 3.1. O formato dos comandos correspondentes em linguagem de montagem é:

instr{*cond*} Rn, *Operando2*

onde

- *instr* especifica a instrução e pode ser CMP, CMN, TEQ ou TST.
- *cond* é um sufixo de condição da Tabela 1.3.
- Rn é um registrador, usado como primeiro da operação.
- *Operando2* especifica o segundo operando da operação, no mesmo formato descrito na Seção 3.2.1. .

Note que o sufixo S não é necessário, pois é implícito que essas instruções atualizam os bits de condição. Todos os bits de condição (C, N, V e Z) são atualizados de acordo com a operação e o resultado.

O Exemplo 3.4 ilustra a sintaxe dos comandos que não armazenam o resultado em um registrador.

```
1       cmp    r1,r2              @ comparação simples, incondicional
2                                 @ [C,N,V,Z] <-- r1-r2
3       cmnmi  r10,r12            @ comparação com negação, condicional
4                                 @ [C,N,V,Z] <-- r10-(-r12)
5       teqeq  r10,#1             @ testa bits com valor imediato
6                                 @ [C,N,V,Z] <-- r10 ou_exclusivo 1
7                                 @ condicional
8       tst    r9,r8,lsl#3        @ testa bits com registrador, incondicional
9                                 @ [C,N,V,Z] <-- r10 e_logico r8*8
10      tstcs  r0,r11,ror r1      @ testa bits com registrador, condicional
11                                @ [C,N,V,Z] <-- r0 e_logico (r11 rotacionado r1 bits)
```

Exemplo 3.4: Instruções de processamento de dados que não armazenam o resultado.

3.2.3 *Instruções que armazenam o resultado*

As instruções aritméticas e lógicas no ARM são

- ADD (adição),
- SUB (subtração),
- ADC (adição com vai-um),
- SBC (subtração com empresta-um),
- RSB (subtração reversa),
- AND (e-lógico),
- ORR (ou-lógico),
- EOR (ou-exclusivo)
- RSC (subtração reversa com empresta-um)
- BIC (desliga bit).

O formato geral dos comandos correspondentes em linguagem de montagem é

instr{*cond*}{S} Rd, Rn, *Operando2*

onde

- *instr* especifica a instrução e pode ser AND, EOR, SUB, RSB, ADD, ADC, SBC, RSC, ORR ou BIC.
- *cond* é um sufixo de condição da Tabela 1.3.
- S, se presente, faz com que o montador coloque o valor 1 no bit 20 da instrução (bit S na Figura 3.3), indicando que a operação deve atualizar o valor dos bits de condição N e Z de acordo com o valor resultante do registrador Rd. Se não presente, o bit S da instrução é 0 e a operação não altera o valor de nenhum bit de condição.
- Rd é o registrador destino, onde o resultado da operação é armazenado.
- Rn é um registrador, usado como primeiro operando da operação.
- *Operando2* especifica o segundo operando da operação, no mesmo formato descrito na Seção 3.2.2.

O Exemplo 3.5 ilustra a sintaxe de alguns comandos de processamento de dados que armazenam o resultado em um registrador.

```
1       add    r1,r2,r3          @ adição, incondicional
2                                @ r1 <-- r2 + r3
3                                @ bits de condição não são alterados
4       subeqs r3,r7,r9,lsr r10  @ subtração, condicional
5                                @ r3 <-- r7 + (r9 desloc. direita por r10)
6                                @ bits de condição são atualizados
7       rsbpls r4,r6,r8,lsl #4   @ subtração reversa, condicional
8                                @ r4 <-- (r8 * 16) - r6
9                                @ bits de condição são atualizados
10      ands   r14,r14,#2048     @ e-lógico, incondicional
11                               @ r14 <-- r14 & 0x800
```

Exemplo 3.5: Instruções de processamento de dados que armazenam o resultado.

Problema 3.1. Escrever um trecho de programa em linguagem de montagem ARM para substituir, em uma cadeia de caracteres, toda ocorrência do caractere 'espaço em branco' (byte 0x20*) pelo caractere '-' (byte* 0x2d*). Suponha que o endereço da cadeia seja dado em* r0 *e o número de elementos em* r1*.*

O Exemplo 3.6 mostra uma solução para o Problema 3.1. O trecho das linhas 8 a 19 substitui os caracteres conforme especificado. Esse exemplo serve também para ilustrar a necessidade do uso de instruções LDR para carregar algumas constantes em registradores. Como nem todo valor pode ser carregado em um registrador com endereçamento imediato no ARM (com instruções MOV), se torna

Exemplo 3.6: Uma solução para o Problema 3.1.

```
1  @ Substitui caracteres ' ' (espaço) por '-' em cadeia de caracteres
2
3     .org 0x1000
4  inicio:
5      ldr    r0,ender_cadeia    @ carrega endereço de cadeia em r0, e
6      ldr    r1,tamanho_cadeia  @ número de elementos da cadeia em r1
7
8  substitui:
9      cmp    r1,#0              @ se comprimento é zero,retorna
10     ble    final
11     add    r2,r0,r1           @ r2 marca final da cadeia
12     mov    r1,#0x2d           @ r1 contém caractere '-'
13 loop:
14     ldrb   r3,[r0],#1         @ examina caractere corrente
15                               @ pós-indexada, r0 é atualizado
16     cmp    r3,#0x20           @ caractere é espaço?
17     streqb r1,[r0,#-1]        @ se sim, substitui
18     cmp    r0,r2              @ chegou ao final da cadeia?
19     bne    loop               @ não, continua
20 final:
21     b      continua           @ continua o programa (não mostrado)
22
23 cadeia:
24    .ascii   "Uma cadeia de caracteres"
25 tamanho_cadeia:
26     .word   . - cadeia
27 ender_cadeia:
28     .word   cadeia
29     .end
30 ...
```

necessário definir uma variável inicializada com o valor da constante, e usar uma instrução LDR. Assim, para carregar o endereço do rótulo `cadeia` no registrador `r0` (linha 5), usamos uma instrução LDR cujo operando é a variável `ender_cadeia` declarada nas linhas 27 e 28, inicializada pelo montador com o endereço do rótulo `cadeia`. Esse método de declarar uma variável inicializada com uma constante é utilizado com tanta frequência que o montador do ARM oferece uma facilidade para o programador. Se o operando de uma instrução LDR for uma expressão constante precedido do caractere '=', o montador declara automaticamente uma variável temporária, armazena a expressão constante no rótulo da variável temporária, e substitui o operando da instrução LDR pelo rótulo da variável temporária. Assim, para carregar no registrador `r0` o valor 1005 (0x3ed, que não pode ser montado como valor imediato em uma instrução MOV), podemos utilizar o pseudo-comando

```
ldr    r0,=1005          @ carrega valor 1005 em r0
```

Portanto, como um rótulo é uma expressão constante para o montador, no Exemplo 3.6 podemos eliminar as linhas 27 e 28 e substituir a linha 5 por

```
5       ldr    r0,=cadeia          @ carrega endereço de cadeia em r0
```

O Exemplo 3.6 ilustra ainda outra facilidade oferecida pelo montador ARM: o símbolo especial '.' tem o valor do ponto de montagem corrente. Isso é útil por exemplo para calcular o número de bytes de uma cadeia de caracteres, como na linha 26: o valor da expressão

$$. - \texttt{cadeia}$$

é o número de bytes da cadeia declarada na linha 24.

Problema 3.2. Traduzir o comando condicional do Exemplo 3.7, em C, para linguagem de montagem do ARM.

Exemplo 3.7: Comando condicional com duas expressões lógicas em C.

```
1  int x, y, z;
2  ...
3  if (x==y && x==z)
4     x=0;
5  else
6     x=100;
```

O Exemplo 3.8 mostra uma solução para o Problema 3.2. Note que não é necessário utilizar instruções de desvio.

Exemplo 3.8: Uma solução para o Problema 3.1.

```
1  x:  .skip 4
2  y:  .skip 4
3  z:  .skip 4
4  ...
5       ldr    r0,x              @ carrega endereço de cadeia em r0, e
6       ldr    r1,y              @ carrega endereço de cadeia em r0, e
7       ldr    r2,z              @ carrega endereço de cadeia em r0, e
8
9       cmp    r0,r1             @ primeira expressão do if
10      cmpeq  r0,r1             @ executa segunda expressão se primeira não verdadeira
11      moveq  r3,#0             @ prepara registrador r3 com constante
12      movne  r3,#100           @ uma delas será atribuída a x
13      str    r3,x              @ armazena constante em x
14
15 ...
```

Uma implementação do comando *switch* do Exemplo **??** utilizando uma tabela de desvios é apresentada no Exemplo 3.9.

```
...
    ldr   r0,val           @ carrega variavel de seleção
                           @ primeiro verificamos os limites
    cmp   r0,#1000         @ menor que menor entrada na tabela?
    blt   casedefault      @ sim, desvia
    ldr   r1,=1005         @ maior valor, não pode ser montado imediato
    cmpge r0,r1            @ compara com maior valor
    bgt   casedefault      @ val é maior que a maior entrada na tabela
                           @ r0 será o índice na tabela
    sub   r0,r0,#1000      @ primeiro valor tem índice zero
    ldr   r1,=tab_switch   @ carrega endereço da tabela de desvios
    ldr   pc,[r1,r0,lsl#2] @ desvia para seleção  (pc = r1 + r0*4)
tab_switch:
    .word case1000
    .word case1001
    .word casedefault
    .word casedefault
    .word case1004
    .word case1005
case1000:
    ldr   r0,y
    str   r0,x          @ x = y
    b     final         @ break
case1001:
    ldr   r0,x
    str   r0,y          @ y = x
    b     final         @ break
case1004:
    ldr   r0,x
    str   r0,t          @ t = x
                        @ note que não há break
case1005:
    mov   r0,#0
    str   r0,t          @ t = 0
    b     final         @ break
casedefault:
    mov   r0,#0
    str   r0,t          @ t = 0
    str   r0,x          @ x = 0
    str   r0,y          @ y = 0
final:
...
```

Exemplo 3.9: Implementação do comando *switch* do Exemplo **??** usando uma tabela de desvios.

4
Instruções de pilha e transferência múltipla entre registradores e memória

As instruções de transferência múltipla transferem um grupo de registradores da memória para o processador ou do processador para a memória.

A Figura4.1 mostra a codificação das instruções de transferência múltipla.

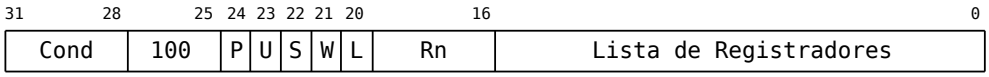

Figura 4.1: Codificação das instruções de transferência múltipla.

Na Figura 2.1, o campo Cond é o campo de condição, que controla a execução condicional da instrução, conforme descrito na Seção 1.3.

O campo Rn indica o *registrador base*, que contém o endereço base do operando, utilizado no cálculo do endereço efetivo do operando, que varia de acordo com o modo de endereçamento utilizado pela instrução. Os bits P e U controlam o modo de endereçamento: P (*pre/post*) controla se o registrador base deve ser pré (P = 0) ou pós (P = 1) indexado; o campo U (*up/down*) controla se a indexação envolve decremento (U = 0) ou incremento (U = 1) do registrador base. O bit W (*write-back*) indica se o registrador base deve ser atualizado com a indexação; o bit L indica se a operação é carrega registradores ou armazena registradores.

O campo Lista de registradores contém um bit para cada registrador visível, e controla quais registradores devem ser transferidos (se o bit correspondente for 1 na instrução, o registrador deve ser transferido). Todos, ou um grupo de registradores, podem ser transferidos, mas a lista não pode ser vazia (ou seja, pelo menos um bit deve ter valor 1 na instrução). Os registradores são armazenados ou carregados de endereços consecutivos de memória.

Cada modo de endereçamento tem um comando específico em

linguagem de montagem, mostrados na Tabela 4.1. Para conveniência, cada modo de endereçamento tem dois nomes: um para ser usado quando a instrução está sendo utilizada para implementar uma pilha e outro nome para outros usos.

Tabela 4.1: Modos de endereçamento das instruções de transferência múltipla.

Pilha	Outros	Descrição	L	U	P
LDMED	LDMIB	carrega com pré incremento	1	1	1
LDMFD	LDMIA	carrega com pós incremento	1	1	0
LDMEA	LDMDB	carrega com pré decremento	1	0	1
LDMFA	LDMDA	carrega com pós decremento	1	0	0
STMFA	STMIB	armazena com pré incremento	0	1	1
STMEA	STMIA	armazena com pós decremento	0	1	0
STMFD	STMDB	armazena com pré incremento	0	0	1
STMED	STMDA	armazena com pós decremento	0	0	0

Os nomes base dos comandos em linguagem de montagem são LDM (do inglês *load multiple*) e STM (do inglês *store multiple*). Os sufixos IA, IB, DA e DB, usados para instruções "normais" (que não se referem a pilhas), significam respectivemente incrementa depois (do inglês *increment after*), incrementa antes (do inglês *increment before*), decrementa depois (do inglês *decrement after*) e decrementa antes (do inglês *decrement before*).

Nos comandos relativos a operações de pilha, são utilizados os sufixos FD, ED, FA e EA, que se referem ao tipo da pilha implementada. As letras F e E se referem se o ponteiro da pilha deve ser

- pré-indexado (pilha "cheia", do inglês *Full*, significando que o espaço corrente que o apontador de pilha aponta está ocupado, e portanto é necessário pré-indexar o apontador) ou
- pós-indexado (pilha "vazia", do inglês *Empty*, significando que o espaço corrente que o apontador de pilha aponta não está ocupado).

As letras A e D se referem a uma pilha que cresce para endereços crescentes (do inglês *Ascending*) ou que cresce para endereços decrescentes (do inglês *Empty*). Se a pilha é do tipo *Ascending*, uma instrução STM incrementa o registrador base, e uma instrução LDM decrementa o registrador base. Se a pilha é do tipo *Descending*, uma instrução STM decrementa o registrador base, e uma instrução LDM incrementa o registrador base.

O formato dos comandos LDM e STM em linguagem de montagem é:

LDM*modo*{*cond*} Rn{!}, *lista_de_registradores*
STM*modo*{*cond*} Rn{!}, *lista_de_registradores*

onde um componente entre chaves indica um componente opcional e

- *modo* indica o modo de endereçamento, e pode ser FD, ED, FA, EA, IA, IB, DA ou DB. O montador monta os bits P e U de acordo com o modo de endereçamento, como mostrado na Tabela 4.1.
- *cond* é um sufixo de condição da Tabela 1.3.
- Rn é o registrador base.
- ! é um indicador se o registrador base deve ser atualizado. Se presente, o montador monta o valor 1 no bit 21 da instrução (bit W, do inglês *write-back*, na Figura 4.1), indicando que o registrador Rn deve ter o valor atualizado, ao final da instrução, pelo número de bytes transferidos (para mais ou para menos, dependendo do modo de endereçamento). Se ausente, o valor do registrador Rn permanece inalterado.
- *lista_de_registradores* é uma lista delimitada por chaves ({ e }), indicando quais registradores devem ser transferidos. Na lista, os elementos são separados por vírgulas, e podem ser registradores ou faixas de registradores (no formato *reg-reg*, como r2-r5). O registrador de menor número é armazenado ou carregado do menor endereço de memória, o registrador de maior número é armazenado ou carregado do maior endereço de memória.

O Exemplo 4.2 ilustra a sintaxe dos comandos LDR e STR com endereçamento direto.

```
1       stmia    r0,{r0-r15}     @ salva todos os registradores na memória
2       stmfd    sp!,{r0,r1,r3}  @ empilha três registradores
3       ldmfdeq  sp!,{r1-r3,r14} @ desempilha quatro registradores, condicional
```

Exemplo 4.1: Instruções LDM e STM.

Para simplificar a implementação de pilhas de sistema (que utilizam o registrador SP como base e o modo de endereçamento FD), o montador aceita também comandos PUSH e POP similares aos utilizados pelo processador Faíska. Nesse formato, o registrador base é implícito (SP e o modo de endereçamento, também implícito, é FD).

PUSH{*cond*} *lista_de_registradores*
POP{*cond*} *lista_de_registradores*

onde um componente entre chaves indica um componente opcional e

- *cond* é um sufixo de condição da Tabela 1.3.

- *lista_de_registradores* é uma lista delimitada por chaves ({ e }), indicando quais registradores devem ser transferidos, como especificado acima.

  O comando

  PUSH *lista_de_registradores*

  equivale ao comando

  STMFD `sp!`, *lista_de_registradores*

e o comando

  POP *lista_de_registradores*

  equivale ao comando

  LDMFD `sp!`, *lista_de_registradores*

```
1       push    {r0-r8}      @ empilha registradores de r0 a r8, incondicional
2       popgt   {r0,r1,r3}   @ desempilha três registradores, condicional
3       pop     {r0-r3,r13}  @ desempilha cinco registradores: r0 a r3, e r13
```

Exemplo 4.2: Instruções PUSH e POP.

5
Procedimentos

Para agilizar a execução de chamadas de procedimento, o processador ARM não utiliza a pilha para armazenar o endereço de retorno em chamadas de procedimento. O endereço de retorno é armazenado, no momento da chamada, no registrador ligador (`r14`, ou `lr`).

A Figura5.1 mostra a codificação das instruções de chamada de procedimento.

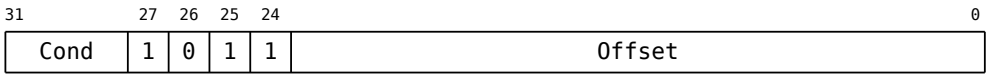

Figura 5.1: Codificação das instruções de chamada de procedimento.

O comando em linguagem de montagem para chamada de procedimentos é `BL` (do inglês *branch and link*).

O formato geral do comando em linguagem de montagem é igual ao comando de desvio com endereçamento imediato (Seção 3.1.1):

`BL`{*cond*} *endereço*

onde

- *cond* é um sufixo de condição da Tabela 1.3.
- *endereço* é um endereço que deve estar dentro do intervalo

$$[\texttt{pc} - (2^{24} - 1), \texttt{pc} + 2^{24}]$$

O endereço do procedimento é calculado relativo ao registrador contador de programa, `pc`. O montador calcula a diferença, em complemento de dois, do ponto de montagem e do endereço do procedimento. Como no processador ARM todos os endereços devem ser múltiplos de quatro, os dois bits menos significativos de um endereço são zero, e portanto não precisam ser armazenados no campo `Offset`. Assim, o montador desloca, de dois bits para a direita, o valor da diferença encontrada entre o ponto de montagem e o

endereço do procedimento, e monta o valor resultante no campo Offset.

No momento da execução, o processador armazena o valor do registrador contador de programa `pc` no registrador de ligação `lr`, extrai o valor do campo Offset, estende o bit de sinal para 32 bits, desloca esse valor que dois bits para a esquerda, adiciona com o valor corrente do registrador contador de programa e finalmente atualiza o valor do registrador contador de programa com o valor calculado. A pipeline é esvaziada e o processador inicia a execução da primeira instrução do procedimento no endereço alvo.

O Exemplo 5.1 ilustra a sintaxe do comando BL.

```
1     blpl Calcula            @ chamada de procedimento condicional
2     bl   Imprime            @ chamada de procedimento incondicional
3     ...
4   Calcula:                  @ um procedimento
5     ...
6   Imprime:                  @ outro procedimento
7     ...
```

Exemplo 5.1: Instruções de chamada de procedimento.

O processador ARM não necessita de uma instrução específica para o retorno de procedimento. Como o endereço de retorno está no registrador `lr`, para retornar do procedimento basta efetuar um desvio por registrador:

```
    bx    lr                @ retorna de procedimento, incondicional
```

Problema 5.1. Escreva um procedimento `Ordena` que ordene um vetor de inteiros com sinal. O endereço do vetor é dado no registrador `r0`, e o número de elementos no registrador `r1`.

O Exemplo 5.2 mostra uma solução para o Problema 5.1, usando o método de ordenação por inserção. Nesse método, iniciamos considerando que um sub-vetor de comprimento um (o primeiro elemento do vetor) está ordenado. A cada passo, aumentamos o comprimento do sub-vetor ordenado encontrando a posição correta de mais um elemento do vetor. O exemplo inclui um código em C, usado como referência na implementação.

Exemplo 5.2: Procedimento para ordenar um vetor de inteiros.

```
1  @ ******
2  @ Ordena
3  @ ******
4  @ Ordena vetor de inteiros com sinal
5  @   entrada: endereço do vetor em r0, número de bytes em r1 (r1>0)
6  @   saída: vetor ordenado crescentemente
7  @   destrói: r2,r3,r4,r5,r6,r7 e flags
8
9  Ordena:
10     mov    r2,#1              @ r2 é variável i
11 for_i:
12     cmp    r2,r1              @ todos os elementos foram processados?
13     bge    fim                @ então terminou
14     ldr    r4, [r0,r2,lsl#2]  @ val = a[i], r4 é variável val
15     mov    r5,r2              @ r5 é variável j
16 for_j:
17     subs   r5,r5,#1           @ passou do início do vetor?
18     bmi    end_for_j          @ sim, terminou este passo
19     ldr    r6,[r0,r5,lsl#2]   @ r6 é a[j]
20     cmp    r6,r4              @ if (a[j] <= val)
21     ble    end_for_j          @ encontramos a posição de val
22     add    r7,r5,#1           @ r7 é j+1
23     str    r6,[r0,r7,lsl#2]   @ a[j+1]=a[j]
24     b      for_j              @ continua o for j
25 end_for_j:
26     add    r5,r5,#1           @ r5 tem j+1
27     str    r4,[r0,r5,lsl#2]   @ a[j+1]=val, coloca elemento val na posição
28     add    r2,r2,#1           @ avança i
29     b      for_i              @ continua o for i
30 fim:
31     bx    lr                  @ quando terminou, retorna
32
33 @@@@@@@
34 @ void Ordena(int a[], int compr) {
35 @   int i,j,val;
36 @   for (i=1; i<compr; i++) {
37 @     val=a[i];
38 @     for (j=i-1; j>=0; j--) {
39 @       if (a[j]<=val) break;
40 @       a[j+1]=a[j];
41 @     }
42 @     a[j+1]=val;
43 @   }
44 @ }
45 @@@@@@@@
```

Problema 5.2. Escreva uma função `Divide` que efetue a operação de divisão de números inteiros. O dividendo é dado no registrador `r0`, o divisor no registrador `r1`. A função deve retornar o quociente no registrador `r0` e o resto no registrador `r1`.

O Exemplo 5.3 mostra uma solução para o Problema 5.2, usando divisões sucessivas. Esse método é simples de implementar mas não é eficiente: o número de subtrações realizadas é igual ao valor do quociente. Portanto, se o quociente é grande, a operação é muito cara.

Exemplo 5.3: Função para dividir dois inteiros sem sinal.

```
1  @ ******
2  @ DivideSuc
3  @ ******
4  @ Divide dois inteiros sem sinal pelo método de subtrações sucessivas
5  @   entrada: dividendo em r0, divisor em r1
6  @   saída: quociente em r0, resto em r1
7  @   destrói: r0,r1,r2 e flags
8
9  DivideSuc:
10         mov   r2,#0       @ r2 é o quociente
11 DivSuc1:
12         subs  r0,r0,r1    @ subtrai divisor do dividendo
13         addhi r2,r2,#1    @ incrementa quociente
14         bhi   DivSuc1     @ enquando r0 > r1
15                           @ armazena valores de retorno nos registradores
16         addne r1,r0,r1    @ se dividendo ficou menor que divisor
17         moveq r1,#0       @ senão resto é zero
18         mov   r0,r2       @ quociente em r0
19         bl    lr
```

Um método melhor para divisão é similar ao que normalmente usamos para fazer divisão com lápis e papel: a cada iteração da operação, subtraímos a maior parcela possível do dividendo. Por exemplo, para dividir 237 por 5, a primeira parcela que subtraímos é 200 (pois $200 = \mathit{divisor} \times 40$), resultando em um quociente parcial de 40 deixando um "resto" igual a 37 para ser ainda dividido. Dividimos então 37 por 5, e nessa iteracão a parcela que subtraímos é $30 = \mathit{divisor} \times 6$, que é somada à primeira parcela para formar o quociente parcial de 46, com resto 7. Dividimos então 7 por 5, e nessa iteração subtraímos a parcela $5 = \mathit{divisor} \times 1$, que é somada ao quociente parcial para formar o quociente final igual a 47 com resto 2. Note que a cada iteração a maior parcela que subtraímos é na forma de uma constante multiplicada por uma potência de 10 (2×10^2, 3×10^1, 5×10^0). Note ainda que o número de interações é igual a log_{10} do dividendo.

O Exemplo 5.4 mostra uma solução para o Problema 5.2 usando a abordagem descrita acima. Como a base é binária, a constante multiplicadora de cada parcela é 0 ou 1. A função inicialmente calcula a maior parcela que pode ser subtraída, multiplicando por 2 suces-

sivamente o divisor, enquanto esse for menor do que o dividendo (linhas 12 a 18). Então sucessivamente subtraímos as parcelas, da maior potência de 2 para a menor, acumulando o quociente parcial no registrador `r2` (linhas 20 a 27). O número de parcelas é log_2 do dividendo, fazendo com que em média a função do Exemplo 5.4 execute muito mais rápido do que a função do Exemplo 5.3.

Exemplo 5.4: Função para dividir dois inteiros sem sinal, otimizada.

```
1  @ ******
2  @ Divide
3  @ ******
4  @ Divide dois inteiros sem sinal pelo método de subtrações sucessivas
5  @ otimizada (número de ubtrações é log2 divisor)
6  @   entrada: dividendo em r0, divisor em r1
7  @   saída: quociente em r0, resto em r1
8  @   destrói: r0,r1,r2,r3 e flags
9
10 Divide:
11         mov   r3,#1          @ vai conter maior potência de 2
12 Div1:
13         cmp   r1,#0x80000000 @ enquanto bit mais signif. do divisor não é 1
14         cmpcc r1,r0          @ e enquanto divisor é menor que dividendo
15         movcc r1,r1,asl#1    @ desloca divisor, formando maior parcela
16         movcc r3,r3,asl#1    @ desloca potência de 2
17         bcc   Div1           @
18
19         mov   r2,#0          @ r2 vai ser o quociente, inicia com zero
20 Div2:
21         cmp   r0,r1          @ se dividendo maior que o divisor
22                              @ (na primeira e na última iteração não é)
23         subcs r0,r0,r1       @ subtrai parcela do dividendo
24         addcs r2,r2,r3       @ conta potência de dois no quociente
25         movs  r3,r3,lsr#1    @ agora diminiui a potência de 2
26         movne r1,r1,lsr#1    @ e divide a parcela por 2
27         bne   Div2
28
29         mov   r1,r0          @ terminou, armazena resto em r1
30         mov   r0,r2          @ e quociente em r0
31 Div3:
32         bx    lr
```

Problema 5.3. Escrever um procedimento `Subtrai64` que efetue a operação de subtração de dois valores inteiros de 64 bits, com sinal. Cada valor é armazenado em duas palavras de memória consecutivas: a palavra menos significativa de um valor tem o menor endereço, a palavra mais significativa tem o maior endereço. O endereço de um valor de 64 bits é o endereço da palavra menos significativa.

São dados como parâmetros o endereço do minuendo em `r0`, o endereço do subtraendo em `r1`, e o endereço de onde deve ser armazenada a diferença em `r2`.

Se o procedimento é do tipo *folha*, ou seja, não realiza nenhuma outra chamada de procedimento, como dos exemplos anteriores, a pilha não precisa ser utilizada para armazenar o endereço de retorno,

Exemplo 5.5: Função para subtrair dois inteiros de 64 bits com sinal.

```
1  @ **********
2  @ Subtrai64
3  @ **********
4  @ Subtrai dois inteiros de 64 bits. Cada inteiro é armazenado em
5  @ duas palavras consecutivas na memória.
6  @   entrada: endereço do minuendo em r0, endereço do subtraendo em r1
7  @            e endereço do resultado em r2
8  @   saída: diferença, armazenada no endereço do resultado
9  @   destrói: r3,r4 e flags
10
11 Subtrai64:
12         ldr   r3,[r0],#4     @ primeira palavra do minuendo, avança apontador
13         ldr   r4,[r1],#4     @ primeira palavra do subtraendo, avança apontador
14         subs  r3,r3,r4       @ calcula diferença
15         str   r3,[r2],#4     @ armazena primeira palavra do resultado,
16                              @ e avança apontador
17         ldr   r3,[r0],#-4    @ carrega segunda palavra do minuendo
18         ldr   r4,[r1],#-4    @ carrega segunda palavrado subtraendo
19         sbcs  r3,r3,r4       @ calcula diferença, com carry da primeira
20         str   r3,[r2],#-4    @ armazena segunda palavra do resultado
21         bx    lr
```

já que este é colocado no registrador `lr` no momento da chamada. Se o procedimento chama outro procedimento, o registrador `lr` deve ser armazenado na pilha antes da nova chamada.

Problema 5.4. Escrever um procedimento `DifAbs64` que efetue calcule o valor absoluto da diferença dois valores inteiros de 64 bits, com sinal, utilizando a função `Subtrai64`. São dados como parâmetros o endereço do primeiro valor em `r0`, o endereço do segundo valor em `r1`, e o endereço de onde deve ser armazenada o valor absoluto da difereça em `r2`.

```
1  @ ********
2  @ DifAbs64
3  @ ********
4  @ Calcula valor absoluto da diferença entre dois inteiros de 64 bits. Cada
5  @ inteiro é armazenado em duas palavras consecutivas na memória.
6  @   entrada: endereço do operando1 em r0, endereço do operando2 em r1
7  @            e endereço do resultado em r2
8  @   saída: valor absoluto da diferença, armazenado no endereço do resultado
9  @   destrói: r0,r1,r2,r3,r4 e flags
10
11 DifAbs64:
12         push  {lr}           @ salva endereço de retorno
13         bl    Subtrai64      @ efetua subtração
14         ldr   r3,[r2,#4]     @ palavra mais significativa do resultado
15         teq   r3,#0          @ testa bit de sinal
16         poppl   {lr}         @ se positivo terminou, recupera end. retorno
17         bxpl    lr           @ e retorna
18                              @ senão, resultado = -resultado
19         sub   sp,sp,#8       @ variável local na pilha
20         mov   r3,#0          @ para armazenar valor 0, com 64 bits
21         str   r3,[sp,#0]     @ primeira palavra 0
22         str   r3,[sp,#4]     @ segunda palavra 0
23         mov   r0,sp          @ primeiro parâmetro é valor 0
24         mov   r1,r2          @ segundo parâmetro é resultado
25         bl    Subtrai64      @ inverte sinal do resultado
26         add   sp,sp,#8       @ desaloca variável local da pilha
27         pop   {lr}           @ recupera endereço de retorno
28         bx    lr             @ retorna
```

Exemplo 5.6: Função para calcular o valor absoluto da diferença de dois inteiros de 64 bits com sinal.

6
Entrada, Saída e Interrupções

6.1 Entrada e saída

No processador ARM a entrada e saída é mapeada na memória, ou seja, o processador ARM não tem instruções especiais para acessar entrada e saída. Para comunicação do processador com os dispositivo são usadas as instruções normais de acesso à memória (LDR e STR).

6.2 Interrupções e Exceções

Interrupções e exceções no ARM são associadas aos modos de operação do processador. O ARM possui dois tipos interrupções externas: FIQ (*fast interrupt requests*, interrupções rápidas) e IRQ (*interrupt requests, interrupções normais*), três tipos de exceções: *Reset*, *Data abort*, *Pre-fetch abort* e *Undefined instruction*.

A Tabela 6.2 mostra as prioridades e os modos de operação que o processador entra quando uma determinada exceção ou interrupção ocorre.

Tabela 6.1: Interrupções e exceções, e respectivos modos de operação.

Exceção	Modo	Descrição
Reset	*Supervisor*	Processador está iniciando ou pino de reset do processador foi ativado.
Data Abort	*Abort*	Instrução de transferência de dados tentou acesso em endereço ilegal.
FIQ	*FIQ*	Pino de interrupção rápida do processador foi ativado.
IRQ	*IRQ*	Pino de interrupção do processador foi ativado.
Prefetch Abort	*Abort*	Processador tentou executar instrução em endereço ilegal.
SVC	*Supervisor*	Instrução de chamada a sistema é executada.
Undefined Instruction	*Undefined*	Processador não reconhece instrução corrente.

Os tipos de interrupções e exceções têm prioridades: se um tipo mais prioritário está sendo atendido os eventos de outros tipos ficam

pendentes. Além disso, para as interrupções externas FIQ e IRQ, a interrupção somente é tratada se o bit de controle o registrador CPSR é igual a zero, indicando que a interrupção está habilitada (bit I e F na Figura 1.2). Caso o bit seja um a interrupção fica pendente.

Tabela 6.2: Prioridades de interrupções e números no vetor de interrupção.

Exeção	Prior.	Tipo para o vetor de interrupções
Reset	1	0
Data Abort	2	4
FIQ	3	7
IRQ	4	6
Prefetch Abort	5	3
SVC	6	2
Undefined Instruction	6	1

Quando uma interrupção é aceita, o processador

- muda o modo de operação para o modo correspondente ao evento;
- copia o registrador CPSR no registrador SPSR do modo correspondente;
- armazena o endereço endereço de retorno da interrupção no registrador `lr` (`r14`) do modo correspondente;
- desvia para o endereço *tipo_da_interrupção* $\times 4$ (elemento do vetor de interrupção correspondente ao tipo da interrupção ou exceção, mostrados na Tabela 6.2.

O vetor de interrupções no ARM tem oito palavras, cada uma correspondendo a uma interrupção ou exceção, e normalmente está localizado no endereço 0 da memória (pode ser mudado por programa, acessando registradores especiais do processador). Algumas implementações possuem um controlador de interrupções integrado, que recebe pedidos dos dispositivos, e informa o tipo de pedido da interrupção mais prioritário ao processador; nesse caso, o vetor de interrupções tem mais palavras (o número de interrupções distintas depende do controlador de interrupções).

O Exemplo 6.1 mostra um trecho de programa com a inicialização do vetor de interrupções. Como FIQ é a última entrada do vetor, o seu tratador pode ser colocado imediatamente após o vetor de interrupções. Nesse exemplo, duas formas de inicializar o vetor de interrupções são mostradas, uma usando uma instrução de desvio (`B`), e outra usando uma instrução de carga (`LDR`) do registrador `pc`. A diferença é que com uma instrução de desvio o tratador da interrupção não pode estar muito distante, pois o endereço alvo é

relativo, codificado como um valor imediato de 24 bits. Já na carga direta ao registrador `pc`, o tratador pode estar em qualquer endereço da memória (mas uma palavra a mais é reservada pelo montador para armazenar o endereço do tratador).

Exemplo 6.1: Exemplo de vetor de interrupções.

```
 1     .org 0
 2  vetor_int:
 3     b      trata_reset            @ tipo 0, Reset
 4     ldr    pc,=trata_undef        @ tipo 1, Undefined Instruction
 5     b      trata_svc              @ tipo 2, SVC
 6     ldr    pc,=trata_prefetch     @ tipo 3, Prefetch abort
 7     ldr    pc,=trata_abort        @ tipo 4, Data abort
 8     .skip  4                      @ tipo 5, reservado
 9     ldr    pc,=trata_irq          @ tipo 6, IRQ
10  trata_fiq:                       @ tipo 7, FIQ
11     ...                           @ tratador FIQ pode ser colocado aqui
```

O Exemplo 6.1 também ilustra o caso em que a implementação não possui controlador de interrupções (ou possui e não é necessário), de modo que FIQ é a interrupção de tipo mais alto que existe. Nesse caso, como o vetor tem apenas oito posições, o código tratador de interrupção FIQ pode ser colocado logo em seguida ao vetor, não necessitando de uma instrução de desvio.

Para retornar do tratamento de uma interrupção ou exceção deve ser utilizada a instrução

```
movs   pc,lr
```

que restaura o registrador de estado, copiando o registrador de estado SPSR para o registrador de estado CPSR do modo em que o processador estava quando a interrupção foi aceita, e desvia para o endereço de retorno armazenado no registrador `lr`, voltando a executar o código interrompido.

Como cada modo de operação tem registradores `sp` e `lr`, específicos do modo, e o registrador de estado CPSR é guardado no registrador SPSR do modo, todo o mecanismo de interrupção é executado com apenas um acesso à memória, para carregar a instrução que desvia para o tratador. Além disso, no modo FIQ os registradores de `r8` a `r12` também são específicos do modo, de forma que o tratador pode utilizar esses registradores sem se preocupar em salvar registradores de usuário, agilizando muito o processamento de interrupções.

Note ainda que, como cada modo tem o seu próprio apontador de pilha, os tratadores podem utilizar pilhas independentes da pilha de usuário. Obviamente, para isso é necessário que as pilhas sejam inicializadas antes de as interrupções serem habilitadas.

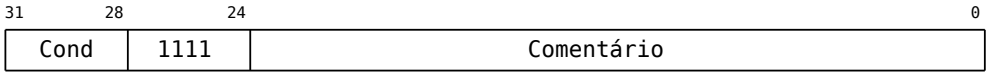

Figura 6.1: Codificação da instrução de chamada de chamada a sistema.

6.3 *Chamada a Sistema*

O comando em linguagem de montagem para chamada de procedimentos é SVC (do inglês *supervisor call*), também chamada de SWI (do inglês *software interrupt*).

O formato geral do comando em linguagem de montagem é:

SVC{*cond*} *expr*24

onde

- *cond* é um sufixo de condição da Tabela 1.3.
- *expr*24 é uma expressão que deve resultar em um valor inteiro de 24 bits sem sinal. O montador avalia a expressão e monta o valor neste campo.

O Exemplo 5.1 ilustra a sintaxe do comando SVC.

```
1     .equ   WRITE, 0x04
2     ...
3     svc    WRITE          @ chamada a sistema, incondicional
4     svceq  0x7            @ chamada a sistema, condicional
5     ...
```

Exemplo 6.2: Instruções de chamada a sistema.

No momento da execução, o processador dispara o mecanismo de interrupção: muda para o modo *Supervisor*, armazena o valor do registrador contador de programa `pc` no registrador de ligação `lr` no novo modo (este é o endereço de retorno), salva o registrador de estado CPSR no registrador SPSR, desabilita interrupções do tipo IRQ e executa a instrução presente na posição 2 no vetor de interrupção.

O processador ignora o valor do campo *expr*24 presente na instrução. Ele pode ser utilizado para passar um código (por exemplo a razão da chamada ao sistema), mas o programador deve consultar a instrução para obter o código. O programador deve nesse caso utilizar o registrador `lr` para acessar a instrução (a instrução de chamada está no endereço `lr-4`). Uma maneira mais eficiente de comunicar o tipo da chamada de sistema é utilizar os registradores. A convenção mais recente, para aplicações embarcadas, é utilizar o registrador `r7` para indicar o tipo da chamada a sistema. Os registradores de `r0` a `r4` são utilizados para passar os parâmetros para a chamada.

O Exemplo

```
1  @ Programa que imprime uma mensagem na tela e termina
2  @ convenção Embbeded Applications Binary Interface (EABI)
3  msg:
4      .ascii      "Hello, I'm ARM!\n"
5  len = . - msg
6
7  main:
8      @ syscall write(int fd, const void *buf, size_t count)
9      mov     r0, #1     @ descritor de arquivo (1 é stdout)
10     ldr     r1, =msg   @ endereço do buffer
11     ldr     r2, =len   @ número de bytes
12     mov     r7, #4     @ write é chamada a sistema de tipo 4
13     svc     #0         @ executa chamada
14
15     @ syscall exit(int status)
16     mov     r0, #0     @ status de retorno é 0
17     mov     r7, #1     @ exit é chamada a sistema do tipo #1
18     svc     #0         @ executa chamada, terminando o programa
19     .end
```

Exemplo 6.3: Programa que imprime mensagem e termina.