Conjunto de Instruções

Universidade Federal do Ceará - Campus Quixadá

Roberto Cabral
rbcabral@ufc.br

16 de Dezembro de 2020

Arquitetura e Organização de Computadores I

Conjunto de Instruções

- Diferentes revisões de arquitetura ARM suportam instruções diferentes.
- No entanto, novas revisões geralmente adicionam instruções e permanecem compatíveis com versões anteriores.
- As instruções do ARM processam os dados mantidos nos registradores e acessam apenas a memória com instruções de carga (*load*) e armazenamento (*store*).
- As instruções do ARM geralmente levam dois ou três operandos.

Instruções de processamento de dados

- As instruções de processamento de dados manipulam dados dentro de registradores. Elas podem ser:
  - instruções de movimentação;
  - instruções aritméticas;
  - instruções lógicas;
  - instruções de comparação;
  - instruções de multiplicação.
- A maioria das instruções de processamento de dados pode processar um de seus operandos usando o *barril shift*.

Condicionais

cond	Mnemonic	Integer	Float	Condition Flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal	Z == 0
0010	CS	Carry set	Greater than, equal, or unordered	C == 1
0011	CC	Carry clear	Less than	C == 0
0100	MI	Negative	Less than	N == 1
0101	PL	Positive, or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C == 1 AND Z == 0
1001	LS	Unsigned lower or same	Less than or equal	C == 0 OR Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 AND N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 OR N != V
1110	none (AL)	Always	Always	Any

Instruções de processamento de dados

- Ao adicionar o sufixo S em uma instrução de processamento de dados, ele atualizará as flags no cpsr.
- Instruções de movimentação e lógicas podem atualizar as flags de carry C, negativo N e zero Z.
- A flag de carry é definida a partir do resultado do *barril shift* como valor do último bit deslocado.
- A flag N é definida como o bit 31 do resultado.
- A flag Z é ativada se o resultado for zero.

Instrução de Movimentação

- A instrução mais simples do ARM.
- Copia N em um registrador de destino Rd, onde N é um registrador ou valor imediato.

`sintaxe: <instrução> {cond} {S} Rd, N`

MOV	Move um valor de 32 bits para um registrador	$Rd = N$
MVN	Move a negação de um valor de 32 bits para um registrador	$Rd = \ N$

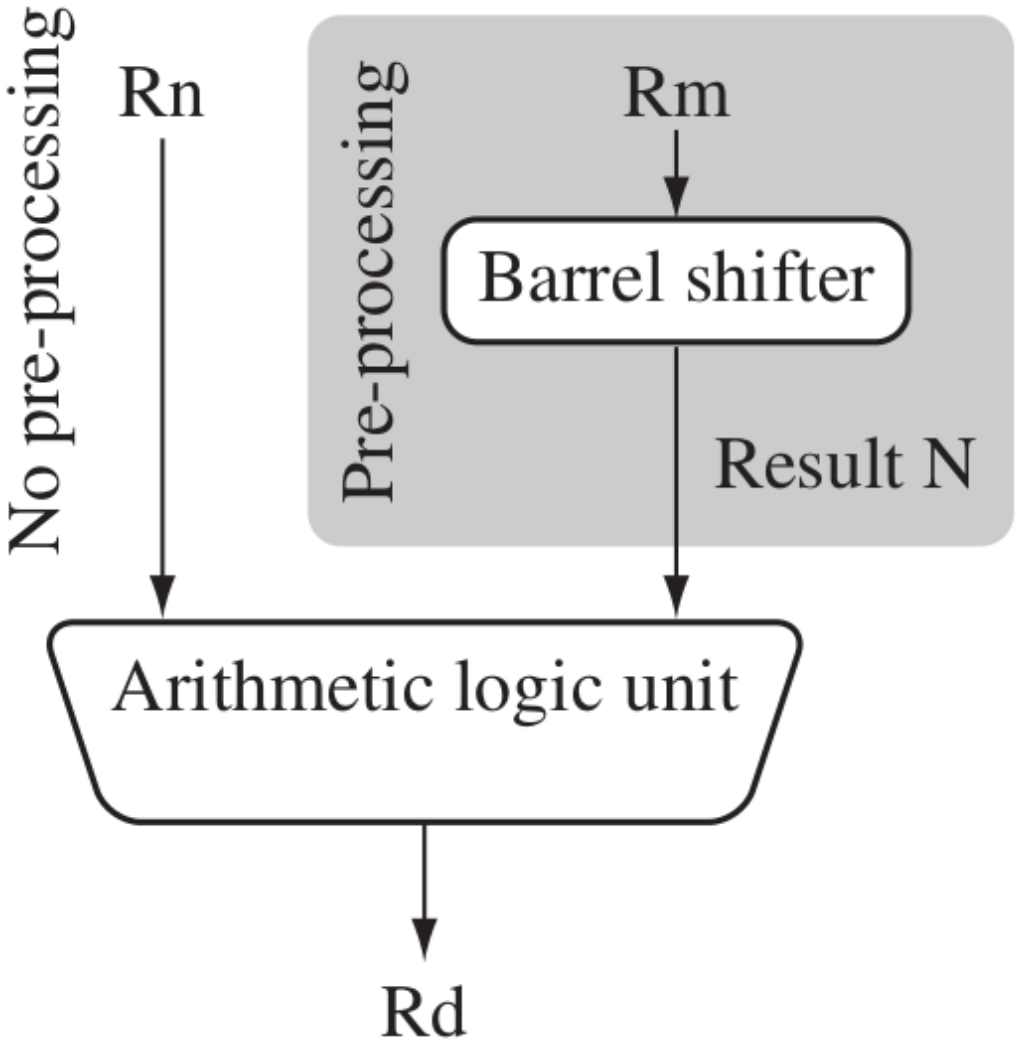
```
mov r7, r5
mov r1, #10
mvn r2, #5
```

Barril Shift

- Um recurso exclusivo e poderoso do processador ARM é a capacidade de deslocar o valor binário de 32 bits de um dos registradores de origem para a esquerda ou direita por um número específico de posições antes que ele entre na ALU.
- Essa mudança aumenta o poder e a flexibilidade de muitas operações de processamento de dados.
- Existem instruções de processamento de dados que não usam o deslocamento de barril, por exemplo, as instruções MUL (multiplicar), CLZ (contagem de zeros à esquerda) e QADD (adição de 32 bits saturada com sinal).

```
MOV r7, r5, LSL #2 ; let r7 = r5*4 = (r5 << 2)
```

Barril Shift



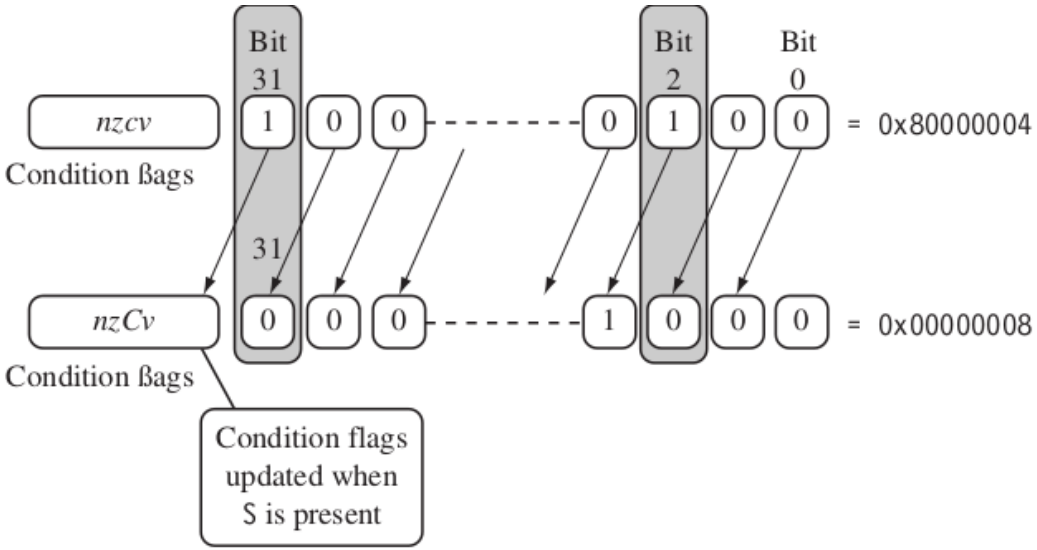
Barril Shift

O *Barril Shift* permite cinco operações diferentes:

Mnemonic	Descrição	Resultado	Deslocamento
`LSL`	Desl. lógico para esquerda	$x << y$	#0-31 ou Rs
`LSR`	Desl. lógico para direita	(com sinal) $x >> y$	#1-32 ou Rs
`ASR`	Desl. aritmético para direita	(sem sinal) $x >> y$	#1-32 ou Rs
`ROR`	Rotação para direita	$x >> y \mid (x << (32 - y))$	#1-31 ou Rs
`RRX`	Rotação estendida para direita	flag $c << 31 \mid (x >> 1)$	-

Barril Shift

```
PRE
  cpsr = nzcvqiFt_USER
  r0 = 0x00000000
  r1 = 0x80000004

  MOVS r0, r1, LSL #1

PROT
  cpsr = nzCvqiFt_USER
  r0 = 0x00000008
  r1 = 0x80000004
```



Barril Shift

Sintaxe das operações de *Barril Shift* para instruções de processamento de dados

N shift operations	Syntax
Immediate	`#immediate`
Register	`Rm`
Logical shift left by immediate	`Rm, LSL #shift_imm`
Logical shift left by register	`Rm, LSL Rs`
Logical shift right by immediate	`Rm, LSR #shift_imm`
Logical shift right with register	`Rm, LSR Rs`
Arithmetic shift right by immediate	`Rm, ASR #shift_imm`
Arithmetic shift right by register	`Rm, ASR Rs`
Rotate right by immediate	`Rm, ROR #shift_imm`
Rotate right by register	`Rm, ROR Rs`
Rotate right with extend	`Rm, RRX`

Instruções Aritméticas

- As instruções aritméticas implementam adição e subtração de valores de 32 bits com e sem sinal.

`sintaxe: <instrução> {cond} {S} Rd, Rn, N`

`ADD`	Soma dois valores de 32 bits	$Rd = Rn + N$
`ADC`	Soma dois valores de 32 bits mais o carry	$Rd = Rn + N + carry$
`SUB`	Subtração de dois valores de 32 bits	$Rd = Rn - N$
`SBC`	Subtração de dois valores de 32 bits (carry)	$Rd = Rn - N - !(carry)$
`RSB`	Sub. reversa de dois valores de 32 bits	$Rd = N - Rn$
`RSC`	Sub. reversa de dois valores de 32 bits (carry)	$Rd = N - Rn - !(carry)$

Instruções Aritméticas

```
PRE
  r0 = 0x00000000
  r1 = 0x00000002
  r2 = 0x00000001

  SUB r0, r1, r2

PROT
  r0 = 0x00000001
```

```
PRE
  r0 = 0x00000000
  r1 = 0x00000077

  RSB r0, r1, #0 ; Rd = 0x0 - r1

PROT
  r0 = -r1 = 0xffffff89
  r1 = 0x00000077
```

```
PRE
  cpsr = nzcvqiFt_USER
  r1 = 0x00000001

  SUBS r1, r1, #1

PROT
  cpsr = nZCvqiFt_USER
  r1 = 0x00000000
```

```
PRE
  r1 = #0x80000005

  MOVS  r0, r1, lsl #1
  MOV   r2, #10
  ADC   r0, r0, r2

PROT
  r0 = 0x15 = 21
```

Usando Barril Shift com Instruções Aritméticas

- A grande possibilidade de valores do segundo operando em instruções lógicas e aritméticas disponíveis na arquitetura ARM é uma característica muito poderosa.

```
PRE
    r0 = 0x00000000
    r1 = 0x00000005

ADD r0, r1, r1, LSL #1

POST
    r0 = 0x0000000f
    r1 = 0x00000005
```

Instruções Lógicas

- As lógicas aplicam operações bitwise sobre dois valores.

`sintaxe: <instrução> {cond} {S} Rd, Rn, N`

`AND`	AND entre dois valores de 32 bits	$Rd = Rn$ `AND` N
`ORR`	OR entre dois valores de 32 bits	$Rd = Rn$ `OR` N
`EOR`	XOR entre dois valores de 32 bits	$Rd = Rn$ `XOR` N
`BIC`	AND NOT entre dois valores de 32 bits	$Rd = Rn$ `ANDNOT` N

Instruções Aritméticas

```
PRE
  r0 = 0x00000000
  r1 = 0x02040608
  r2 = 0x10305070

  ORR     r0, r1, r2

PROT
  r0 = 0x12345678
```

```
PRE
  r1 = 0b1111
  r2 = 0b0101

  BIC r0, r1, r2

PROT
  r0 = 0b1010
```

Instruções de Comparação

- As instruções de comparação são usadas para comparar ou testar um registrador com um valor de 32 bits.
- Elas atualizam os bits de flag $cpsr$ de acordo com o resultado, mas não afetam outros registradores.

`sintaxe: <instrução> {cond} Rn, N`

`CMN`	Compara negativo	flag seta o resultado de $Rn + N$
`CMP`	compara	flag seta o resultado de $Rn - N$
`TEQ`	Testa igualdade de dois inteiros	flag seta o resultado de Rn `XOR` N
`TST`	testa os bits	flag seta o resultado de Rn `AND` N

Instruções Aritméticas

```
PRE
  cpsr = nzcvqiFt_USER
  r0 = 4
  r9 = 4

  CMP      r0, r9

PROT
  cpsr = nZcvqiFt_USER
```

Instrução de Multiplicação

- As instruções de multiplicação multiplicam o conteúdo de um par de registradores e, dependendo da instrução, acumulam os resultados com outro registrador.
- As multiplicações longas se acumulam em um par de registradores representando um valor de 64 bits. O resultado final é colocado em um registrador de destino ou em um par de registradores.

```
sintaxe:  MLA {cond} {S} Rd, Rm, Rs, Rn
sintaxe:  MUL {cond} {S} Rd, Rm, Rs
```

`MLA`	Multiplica e acumula	$Rd = (Rm * Rs) + Rn$
`MUL`	Multiplica	$Rd = Rm * Rs$

Instrução de Multiplicação

`sintaxe: <instrução> {cond} {S} RdLo, RdHi, Rm, Rs`

`SMLAL`	Mul. e acumula (signed long)	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
`SMULL`	Mul. (signed long)	$[RdHi, RdLo] = (Rm * Rs)$
`UMLAL`	Mul. e acumula (unsigned long)	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
`UMULL`	Mul. (unsigned long)	$[RdHi, RdLo] = (Rm * Rs)$

Instrução de Multiplicação

```
PRE
  r0 = 0x00000000
  r1 = 0x00000002
  r2 = 0x00000002

  MUL r0, r1, r2; r0 = r1*r2

PROT
  r0 = 0x00000004
  r1 = 0x00000002
  r2 = 0x00000002
```

```
PRE
  r0 = 0x00000000
  r1 = 0x00000000
  r2 = 0xf0000002
  r3 = 0x00000002

  UMULL   r0,r1,r2,r3 ;[r1,r0] = r2*r3

PROT
  r0 = 0xe0000004 ; = RdLo
  r1 = 0x00000001 ; = RdHi
```

Instruções de Salto

Instruções de salto modificam o fluxo de execução.

`sintaxe: <instrução>{cond} label`

`B`	salto	pc = label
`BL`	salto com link	pc = label lr = endereço da próxima instrução depois do BL

Instruções de Salto

```
    B forward
    ADD r1, r2, #4
    ADD r0, r6, #2
    ADD r3, r7, #4
foward
    SUB r1, r2, #4

backward
    ADD r1, r2, #4
    SUB r1, r2, #4
    ADD r4, r6, r7
    B backward
```

```
    BL  subroutine  ; branch to subroutine
    CMP r1, #5      ; compare r1 with 5
    MOVEQ   r1, #0  ; if (r1==5) then r1 = 0
    .
    .
subroutine
    <subroutine code>
    MOV pc, lr      ; return by moving pc = lr
```

Instruções de Carga e Armazenamento

Transferem dados da memória para os registradores e vice versa.

`sintaxe: <LDR||STR>{cond} {B} Rd, endereço sintaxe: <LDR>{cond} SB||HS||H Rd, endereço sintaxe: <STR>{cond} H Rd, endereço`

LDR	carrega palavra no registrador	Rd = mem32
STR	salva byte ou palavra de um registrador	mem32 = Rd
LDRB	carrega byte no registrador	Rd = mem8
STRB	salva byte de um registrador	mem8 = Rd
LDRH	carrega meia palavra no registrador	Rd = mem16
STRH	salva meia palavra de um registrador	mem16 = Rd
LDSB	carrega um byte com sinal no registrador	Rd = mem8
LDSH	carrega meia palavra com sinal no registrador	Rd = mem16

Instruções de Carga e Armazenamento

- As instruções `LDR` e `STR` só podem carregar e armazenar dados em memória alinhada que tenha o mesmo tamanho do tipo de dados que está sendo carregado ou armazenado.
- Por exemplo, o `LDR` só pode carregar palavras de 32 bits em um endereço de memória que é um múltiplo de quatro bytes - 0, 4, 8 e assim por diante.

```
; load register r0 with the contents of
; the memory address pointed to by register
; r1.
;
    LDR r0, [r1] ; = LDR r0, [r1, #0]
;
; store the contents of register r0 to
; the memory address pointed to by
; register r1.
;
    STR r0, [r1] ; = STR r0, [r1, #0]
```

Modos de endereçamento de load/store de registo único

O conjunto de instruções ARM fornece modos diferentes para endereçamento de memória.

Método de indexação	Dado	Reg.base	Exemplo
Preindex with writeback	mem[base + offset]	base + offset	LDR r0,[r1,#4]!
Preindex	mem[base + offset]	not updated	LDR r0,[r1,#4]
Postindex	mem[base]	base + offset	LDR r0,[r1],#4

- O Preindex com writeback calcula um endereço de um registrador base mais o offset de endereço e então atualiza o registrador base com o novo endereço.
- O preindex funciona de forma análoga ao preindex com writeback, mas não atualiza o registrador base de endereços.
- O Postindex atualiza o registrador base de endereços somente após o endereço ser usado.

Modos de endereçamento de load/store de registo único

```
PRE
    r0 = 0x00000000
    r1 = 0x00090000
    mem32[0x00009000] = 0x01010101
    mem32[0x00009004] = 0x02020202

    LDR r0, [r1, #4]!
Preindexing with writeback:

POST(1) r0 = 0x02020202
        r1 = 0x00090004

    LDR r0, [r1, #4]

Preindexing:

POST(2) r0 = 0x02020202
        r1 = 0x00009000

    LDR r0, [r1], #4

Postindexing:

POST(3) r0 = 0x01010101
        r1 = 0x00009004
```

Exemplos de instruções LDR usando diferentes modos de endereçamento

	Instruction	*r0* =	*r1* +=
Preindex with writeback	`LDR r0,[r1,#0x4]!`	`mem32[r1+0x4]`	`0x4`
	`LDR r0,[r1,r2]!`	`mem32[r1+r2]`	`r2`
	`LDR r0,[r1,r2,LSR#0x4]!`	`mem32[r1+(r2 LSR 0x4)]`	`(r2 LSR 0x4)`
Preindex	`LDR r0,[r1,#0x4]`	`mem32[r1+0x4]`	*not updated*
	`LDR r0,[r1,r2]`	`mem32[r1+r2]`	*not updated*
	`LDR r0,[r1,-r2,LSR #0x4]`	`mem32[r1-(r2 LSR 0x4)]`	*not updated*
Postindex	`LDR r0,[r1],#0x4`	`mem32[r1]`	`0x4`
	`LDR r0,[r1],r2`	`mem32[r1]`	`r2`
	`LDR r0,[r1],r2,LSR #0x4`	`mem32[r1]`	`(r2 LSR 0x4)`

Variações da instrução STRH

	Instruction	Result	*r1* +=
Preindex with writeback	`STRH r0,[r1,#0x4]!`	`mem16[r1+0x4]=r0`	`0x4`
	`STRH r0,[r1,r2]!`	`mem16[r1+r2]=r0`	`r2`
Preindex	`STRH r0,[r1,#0x4]`	`mem16[r1+0x4]=r0`	*not updated*
	`STRH r0,[r1,r2]`	`mem16[r1+r2]=r0`	*not updated*
Postindex	`STRH r0,[r1],#0x4`	`mem16[r1]=r0`	`0x4`
	`STRH r0,[r1],r2`	`mem16[r1]=r0`	`r2`

Transferência entre múltiplos registradores

- Instruções múltiplas de load/store podem transferir vários registradores entre a memória e o processador em uma única instrução.
- A transferência ocorre a partir de um registrador de endereço base Rn apontando para a memória.
- As instruções de transferência de múltiplos registradores são mais eficientes que instruções simples para mover blocos de dados pela memória e salvar e restaurar contexto e pilhas.

```
Syntax: <LDM|STM>{<cond>}<addressing mode> Rn{!},<registers>{^}
```

LDM	load multiple registers	{Rd}^{*N} <- mem32[start address + 4*N] optional Rn updated
STM	save multiple registers	{Rd}^{*N} -> mem32[start address + 4*N] optional Rn updated

Transferência entre múltiplos registradores - Modo de endereçamento

Addressing mode	Description	Start address	End address	$Rn!$
IA	increment after	Rn	$Rn + 4^*N - 4$	$Rn + 4^*N$
IB	increment before	$Rn + 4$	$Rn + 4^*N$	$Rn + 4^*N$
DA	decrement after	$Rn - 4^*N + 4$	Rn	$Rn - 4^*N$
DB	decrement before	$Rn - 4^*N$	$Rn - 4$	$Rn - 4^*N$

Modos de endereçamento de load/store de Múltiplos registradores único

```
PRE
    mem32[0x80018] = 0x03
    mem32[0x80014] = 0x02
    mem32[0x80010] = 0x01
    r0 = 0x00080010
    r1 = 0x00000000
    r2 = 0x00000000
    r3 = 0x00000000

    LDMIA r0!, {r1-r3}

POST
    r0 = 0x0008001c
    r1 = 0x00000001
    r2 = 0x00000002
    r3 = 0x00000003
```

Transferência entre múltiplos registradores

- As instruções de armazenamento funcionam de forma análoga às instruções de carga.
- Se usarmos um store com atualização de base, a instrução de load configurada com o mesmo número de registradores recarregará os dados e restaurará o ponteiro de endereço base.
- Isso é útil quando precisamos salvar temporariamente um grupo de registros e restaurá-los posteriormente.
- As instruções de decremento acessam os registradores na ordem inversa.

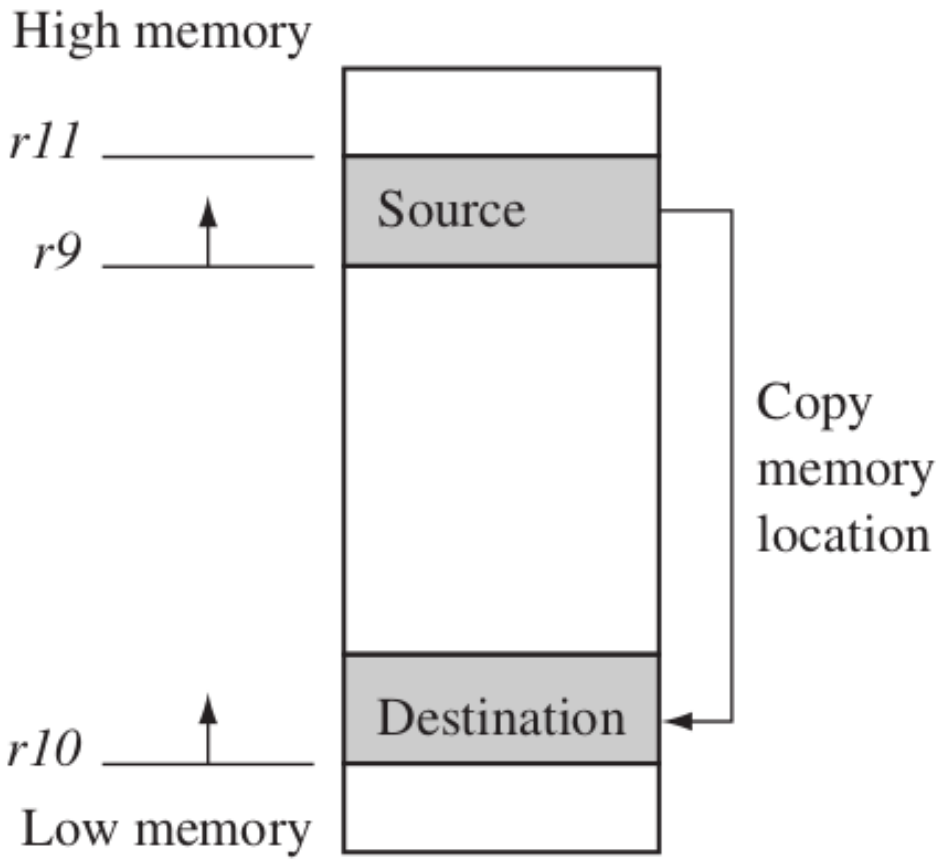
Store multiple	Load multiple
STMIA	LDMDB
STMIB	LDMDA
STMDA	LDMIB
STMDB	LDMIA

Modos de endereçamento de load/store de Múltiplos registradores único

```
PRE
    r0 = 0x00090000
    r1 = 0x00000009
    r2 = 0x00000008
    r3 = 0x00000007

    STMIB r0!, {r1-r3}

    MOV r1, #1
    MOV r2, #2
    MOV r3, #3

PRE(2)
    r0 = 0x0000900c
    r1 = 0x00000001
    r2 = 0x00000002
    r3 = 0x00000003

    LDMDA r0!, {r1-r3}

POST
    r0 = 0x00090000
    r1 = 0x00000009
    r2 = 0x00000008
    r3 = 0x00000007
```

Copiar bloco de memória

```
;r9 points to start of source data

;r10 points to start of destination data

;r11 points to end of the source

loop
    ;load 32 bytes from source and
         update r9 pointer

    LDMIA r9!, {r0-r7}

    ;store 32 bytes to destination and
         update r10 pointer

    STMIA r10!, {r0-r7} ; and store them

    ;have we reached the end

    CMP r9, r11

    BNE loop
```



Operações de Pilha

- A operação pop (remover dados de uma pilha) usa uma instrução de carga múltipla;
- A operação push (colocando dados na pilha) usa uma instrução de armazenamento múltiplo.
- Ao usar uma pilha, você deve decidir se a pilha aumentará ou diminuirá na memória.
  - A pilha é crescente (A) quando cresce para endereços de memória maiores.
  - A pilha é decrescente (D) quando cresce para endereços de memória menores.

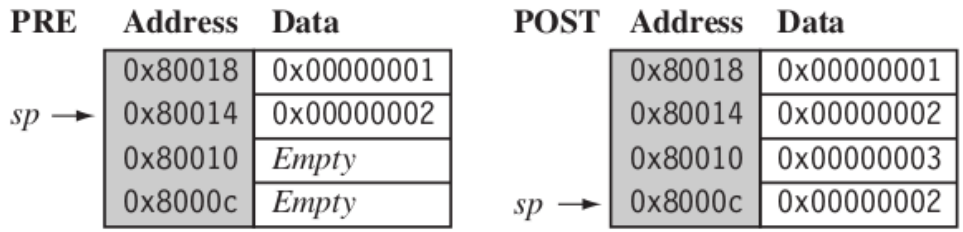
Operações de Pilha

- Quando você usa uma pilha completa (*full stack* - F), o ponteiro da pilha sp aponta para um endereço que é o último local usado ou completo (ou seja, sp aponta para o último item da pilha).
- Por outro lado, se você usar uma pilha vazia (*empty stack* - E), o sp aponta para um endereço que é o primeiro local não utilizado ou vazio (ou seja, aponta para o último item da pilha).

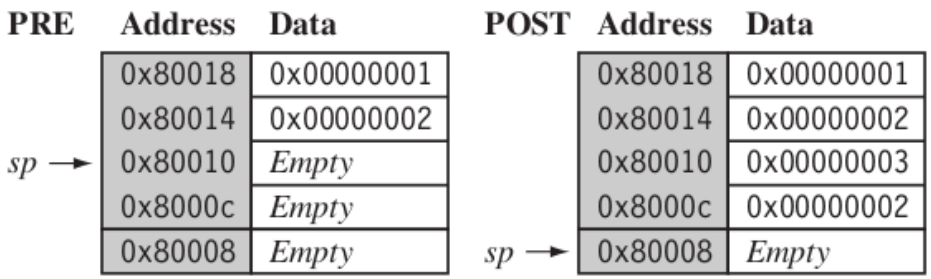
Addressing mode	Description	Pop	= LDM	Push	= STM
FA	full ascending	LDMFA	LDMDA	STMFA	STMIB
FD	full descending	LDMFD	LDMIA	STMFD	STMDB
EA	empty ascending	LDMEA	LDMDB	STMEA	STMIA
ED	empty descending	LDMED	LDMIB	STMED	STMDA

Copiar bloco de memória

```
;push onto a full descending stack.
PRE
     r1 = 0x00000002
     r4 = 0x00000003
     sp = 0x00080014

     STMFD sp!, {r1,r4}
```




```
;push operation on an empty stack
PRE
     r1 = 0x00000002
     r4 = 0x00000003
     sp = 0x00080010

     STMED sp!, {r1,r4}
```



Operações de Pilha

- O ARM especificou um ATPCS (Standard Procedure Call Standard) que define como as rotinas são chamadas e como os registradores são alocados.
- No ATPCS, as pilhas são definidas como sendo pilhas descendentes completas.
- Assim, as instruções LDMFD e STMFD fornecem as funções pop e push, respectivamente.

Operações de Pilha

- Ao manipular uma pilha há três atributos que precisam ser preservados:
  - a base da pilha (*stack base*);
  - o ponteiro da pilha (*stack pointer*);
  - o limite da pilha (*stack limit*).
- A base da pilha é o endereço inicial da pilha na memória.
- O ponteiro da pilha inicialmente aponta para a base da pilha e vai descendo na memória apontando continuamente para o topo da pilha.
- Se o ponteiro da pilha ultrapassar o limite da pilha, ocorreu um erro de estouro de pilha.

Instruções de swap

- A instrução swap é um caso especial de uma instrução de armazenamento de carga.
- Ele troca o conteúdo da memória pelo conteúdo de um registrador.
- Esta instrução é uma operação atômica - lê e grava um local na mesma operação de barramento, impedindo que qualquer outra instrução leia ou grave nesse local até que seja concluída.

```
Syntax: SWP{B}{<cond>} Rd,Rm,[Rn]
```

SWP	swap a word between memory and a register	$tmp = mem32[Rn]$ $mem32[Rn] = Rm$ $Rd = tmp$
SWPB	swap a byte between memory and a register	$tmp = mem8[Rn]$ $mem8[Rn] = Rm$ $Rd = tmp$

Instrução Swap

Exemplo insturção swap

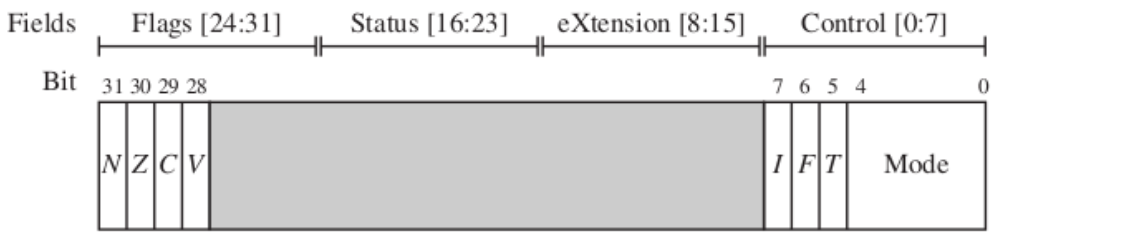
```
PRE
    mem32[0x9000] = 0x12345678

    r0 = 0x00000000
    r1 = 0x11112222
    r2 = 0x00009000

    SWP r0, r1, [r2]

POST
    mem32[0x9000] = 0x11112222

    r0 = 0x12345678
    r1 = 0x11112222
    r2 = 0x00009000
```

```
spin

    MOV r1, =semaphore
    MOV r2, #1
    SWP r3, r2, [r1] ; hold the bus until complete
    CMP r3, #1
    BEQ spin
```

Instruções de registro do status do programa

- O conjunto de instruções ARM fornece duas instruções para controlar diretamente um registro de status do programa.
- A instrução MRS transfere o conteúdo do cpsr ou spsr para um registro;
- A instrução MSR transfere o conteúdo de um registro para o cpsr ou spsr.

```
Syntax: MRS{<cond>} Rd,<cpsr|spsr>
        MSR{<cond>} <cpsr|spsr>_<fields>,Rm
        MSR{<cond>} <cpsr|spsr>_<fields>,#immediate
```



Instruções de registro do status do programa

```
PRE
    cpsr = nzcvqIFt_SVC

    MRS r1, cpsr
    BIC r1, r1, #0x80 ; 0b01000000
    MSR cpsr_c, r1

POST
    cpsr = nzcvqiFt_SVC
```

Carregando constantes

- Não temos no ARM instruçõespara mover uma constante de 32 bits para um registrador.
- Para auxiliar na programação, existem duas pseudo-instruções para mover um valor de 32 bits para um registrador[1].

```
Syntax: LDR Rd, =constant
        ADR Rd, label
```

LDR	load constant pseudoinstruction	*Rd* = 32-bit constant
ADR	load address pseudoinstruction	*Rd* = 32-bit relative address

Table 3.12 LDR pseudoinstruction conversion.

Pseudoinstruction	Actual instruction
`LDR r0, =0xff`	`MOV r0, #0xff`
`LDR r0, =0x55555555`	`LDR r0, [pc, #offset_12]`

[1] mais informações: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/how-to-load-constants-in-assembly-for-arm-architecture

Conjunto de Instruções

Universidade Federal do Ceará - Campus Quixadá

Roberto Cabral
rbcabral@ufc.br

16 de Dezembro de 2020

Arquitetura e Organização de Computadores I