The ARM Instruction Set

Minsoo Ryu

Department of Computer Science and Engineering
Hanyang University

msryu@hanyang.ac.kr

Topics Covered

- Data Processing Instructions
- Branch Instructions
- Load-Store Instructions
- Software Interrupt Instruction
- Program Status Register Instructions
- Loading Constants
- ARMv5E Extensions
- Conditional Execution

Introduction

- Different ARM architecture revisions support different instructions
  - New revisions usually add instructions and remain backward compatible
  - Code you write for architecture ARMv4T should execute on an ARMv5TE processor

ARM Instruction Set

Mnemonics	ARM ISA	Description
ADC	v1	add two 32-bit values and carry
ADD	v1	add two 32-bit values
AND	v1	logical bitwise AND of two 32-bit values
B	v1	branch relative +/− 32 MB
BIC	v1	logical bit clear (AND NOT) of two 32-bit values
BKPT	v5	breakpoint instructions
BL	v1	relative branch with link
BLX	v5	branch with link and exchange
BX	v4T	branch with exchange
CDP CDP2	v2 v5	coprocessor data processing operation
CLZ	v5	count leading zeros
CMN	v1	compare negative two 32-bit values
CMP	v1	compare two 32-bit values
EOR	v1	logical exclusive OR of two 32-bit values
LDC LDC2	v2 v5	load to coprocessor single or multiple 32-bit values
LDM	v1	load multiple 32-bit words from memory to ARM registers
LDR	v1 v4 v5E	load a single value from a virtual address in memory
MCR MCR2 MCRR	v2 v5 v5E	move to coprocessor from an ARM register or registers
MLA	v2	multiply and accumulate 32-bit values
MOV	v1	move a 32-bit value into a register

ARM Instruction Set

MRC MRC2 MRRC	v2 v5 v5E	move to ARM register or registers from a coprocessor
MRS	v3	move to ARM register from a status register (*cpsr* or *spsr*)
MSR	v3	move to a status register (*cpsr* or *spsr*) from an ARM register
MUL	v2	multiply two 32-bit values
MVN	v1	move the logical NOT of 32-bit value into a register
ORR	v1	logical bitwise OR of two 32-bit values
PLD	v5E	preload hint instruction
QADD	v5E	signed saturated 32-bit add
QDADD	v5E	signed saturated double and 32-bit add
QDSUB	v5E	signed saturated double and 32-bit subtract
QSUB	v5E	signed saturated 32-bit subtract
RSB	v1	reverse subtract of two 32-bit values
RSC	v1	reverse subtract with carry of two 32-bit integers
SBC	v1	subtract with carry of two 32-bit values
SMLA*xy*	v5E	signed multiply accumulate instructions ($(16 \times 16) + 32 = 32$-bit)
SMLAL	v3M	signed multiply accumulate long ($(32 \times 32) + 64 = 64$-bit)
SMLAL*xy*	v5E	signed multiply accumulate long ($(16 \times 16) + 64 = 64$-bit)
SMLAW*y*	v5E	signed multiply accumulate instruction ($((32 \times 16) \gg 16) + 32 = 32$-bit)
SMULL	v3M	signed multiply long ($32 \times 32 = 64$-bit)

continued

ARM Instruction Set

Mnemonics	ARM ISA	Description
SMUL*xy*	v5E	signed multiply instructions (16 × 16 = 32-bit)
SMULW*y*	v5E	signed multiply instruction ((32 × 16) ≫ 16 = 32-bit)
STC STC2	v2 v5	store to memory single or multiple 32-bit values from coprocessor
STM	v1	store multiple 32-bit registers to memory
STR	v1 v4 v5E	store register to a virtual address in memory
SUB	v1	subtract two 32-bit values
SWI	v1	software interrupt
SWP	v2a	swap a word/byte in memory with a register, without interruption
TEQ	v1	test for equality of two 32-bit values
TST	v1	test for bits in a 32-bit value
UMLAL	v3M	unsigned multiply accumulate long ((32 × 32) + 64 = 64-bit)
UMULL	v3M	unsigned multiply long (32 × 32 = 64-bit)

Move Instructions

- Copy N into a destination register Rd, where N is a register or immediate value
  - For setting initial values and transferring data between registers

- Syntax: <instruction>{<cond>}{S} Rd, N

MOV	Move a 32-bit value into a register	Rd = N
MVN	Move the NOT of the 32-bit value into a register	Rd = ~N

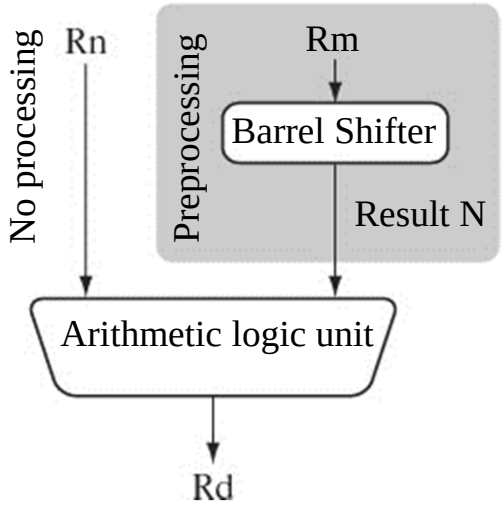
- PRE : r5 = 5, r7 = 8

  MOV r7, r5 ; let r7 = r5

- POST: r5 = 5, r7 = 5

Barrel Shifter

- In the MOV instruction, N can be more than just a register or immediate value
  - It can also be a register Rm that has been preprocessed by the barrel shifter prior to being used by a data processing instruction
  - The barrel shifter can shift a 32-bit binary pattern left or right by a specific number of positions



Barrel Shifter

- Example: we apply a logical shift left (LSL) to register Rm before moving it to the destination register

- PRE : r5 = 5, r7 = 8

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

- POST: r5 = 5, r7 = 20

Table 3.2 Barrel shifter operations.

Mnemonic	Description	Shift	Result	Shift amount y
LSL	logical shift left	*x*LSL *y*	$x \ll y$	#0–31 or *Rs*
LSR	logical shift right	*x*LSR *y*	(unsigned)$x \gg y$	#1–32 or *Rs*
ASR	arithmetic right shift	*x*ASR *y*	(signed)$x \gg y$	#1–32 or *Rs*
ROR	rotate right	*x*ROR *y*	((unsigned)$x \gg y$) | ($x \ll (32 - y)$)	#1–31 or *Rs*
RRX	rotate right extended	*x*RRX	(*c* flag $\ll 31$) | ((unsigned)$x \gg 1$)	none

Note: *x* represents the register being shifted and *y* represents the shift amount.

LSL and LSR

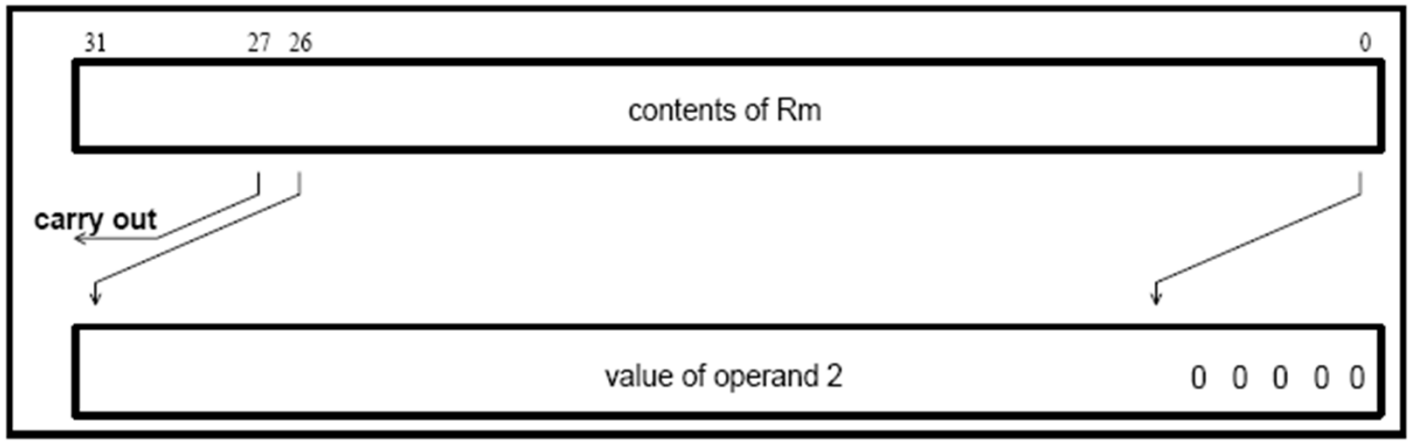


Figure 5-6: Logical shift left

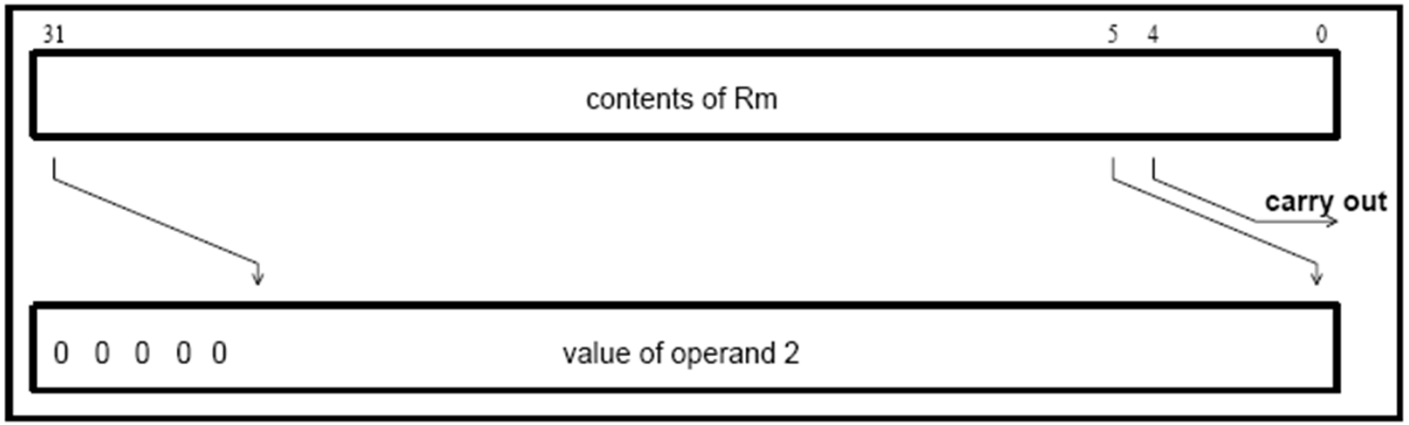


Figure 5-7: Logical shift right

ASR and ROR

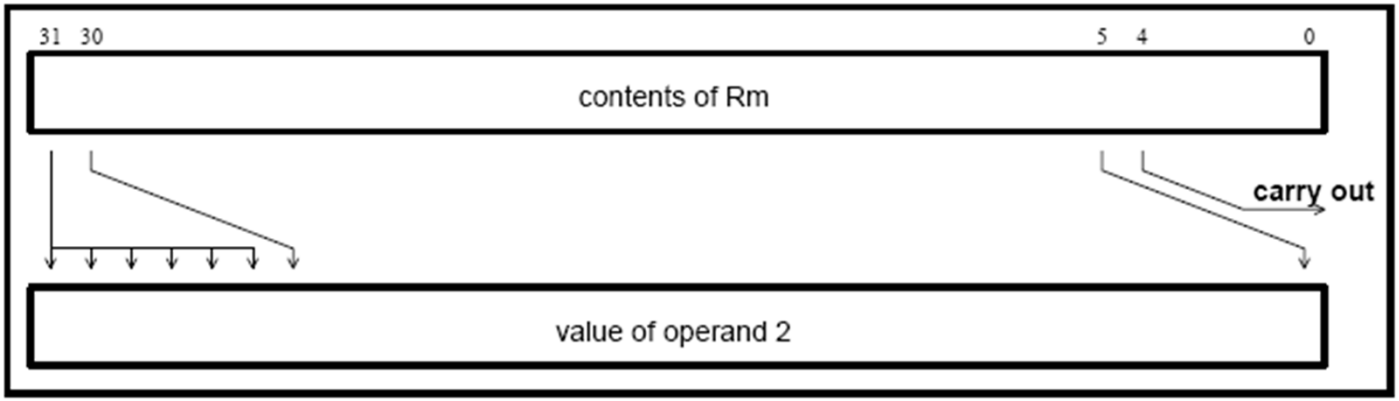


Figure 5-8: Arithmetic shift right

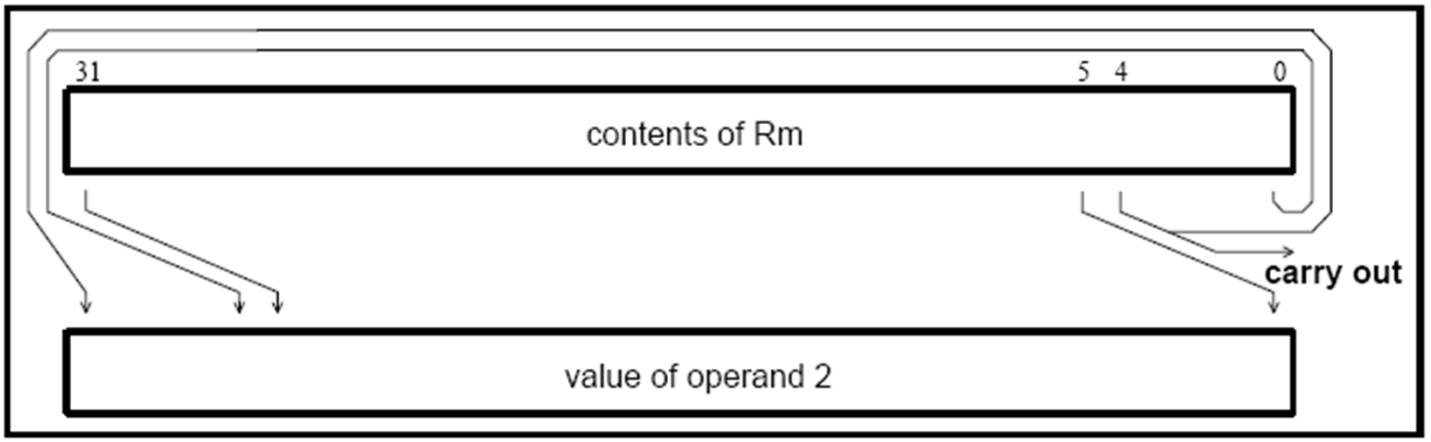


Figure 5-9: Rotate right

Arithmetic Instructions

- Implement addition and subtraction of 32-bit signed and unsigned values

- Syntax: <instruction>{<cond>}{S} Rd, Rn, N
  - N is the result of shift operation

ADC	Add two 32-bit values and carry	Rd = Rn + N + carry
ADD	Add two 32-bit values	Rd = Rn + N
RSB	Reverse subtract of two 32-bit values	Rd = N – Rn
RSC	Reverse subtract with carry of two 32-bit values	Rd = N - Rn -!(carry)
SBC	Subtract with carry of two 32-bit values	Rd = Rn - N -!(carry)
SUB	Subtract two 32-bit values	Rd = Rn - N

Two's Complement Representation

- **Two's complement**
  - **Complement bits, add 1, and ignore the carry out of MSB**

17_{10} = 00010001_2

$\Downarrow$

11101110_2

$+1$

$11101111_2 = -17_{10}$

sign bit									
0	1	1	1	1	1	1	1	=	127
0	0	0	0	0	0	1	0	=	2
0	0	0	0	0	0	0	1	=	1
0	0	0	0	0	0	0	0	=	0
1	1	1	1	1	1	1	1	=	−1
1	1	1	1	1	1	1	0	=	−2
1	0	0	0	0	0	0	1	=	−127
1	0	0	0	0	0	0	0	=	−128

8-bit two's complement integers

- **Ones' complement has a symmetry but an adder for ones' complement is somewhat trickier**

Two's Complement Addition and Subtraction

❒ Two's complement addition

- Two's complement numbers can be added by ordinary binary addition, ignoring any carries beyond the MSB

+3	0011	- 2	1110	+6	0110	+4	0100
+ +4	0100	+ - 6	1010	+ - 3	1101	+ - 7	1001
+7	0111	- 8	11000	+3	10011	- 3	1101

❒ Overflow detection

- If the signs of the addends are the same and the sign of the sum is different from the addends' sign

❒ Two's complement subtraction

- Negate the subtrahend by taking its two's complement, and then add it to the minuend (minuend + (- subtrahend))

Logical Instructions

- Perform bitwise logical operations on the two source registers

- Syntax: <instruction>{<cond>}{S} Rd, Rn, N

AND	Logical bitwise AND of two 32-bit values	Rd = Rn & N
ORR	Logical bitwise OR of two 32-bit values	Rd = Rn | N
EOR	Logical exclusive OR of two 32-bit values	Rd = Rn ^ N
BIC	Logical bit clear (AND NOT)	Rd = Rn & ~N

- PRE : r1 = 0b1111, r2 = 0b0101

  BIC r0, r1, r2

- POST: r0 = 0b1010

Comparison Instructions

- Compare or test a register with 32-bit values
  - No need to apply the S suffix to update the flag

- Syntax: <instruction>{<cond>}{S} Rn, N

CMN	Compare negated	Flags set as a result of Rn + N
CMP	Compare	Flags set as a result of Rn – N
TEQ	Test for equality of two 32-bit values	Flags set as a result of Rn ^ N
TST	Test bits of a 32-bit value	Flags set as a result of Rn & N

- PRE : cpsr = nzcvqFt_USER, r0 = 4, r9 = 4

  CMP r0, r9

- POST: cpsr = nZcvqFt_USER

Comparison Instructions

- The CMP is effectively a subtract instruction with the result discarded

- The TST instruction is a logical AND operation
- The TEQ is a logical exclusive OR operation

Multiply Instructions

- Multiply the contents of a pair of registers and, depending upon the instruction, accumulate the results in with another register

- Syntax: MLA{<cond>}{S} Rd, Rm, Rs, Rn
  MUL{<cond>}{S} Rd, Rm, Rs

MLA	Multiply and accumulate	Rd = (Rm*Rs) + Rn
MUL	Multiply	Rd = Rm*Rs

Multiply Instructions

- PRE : r0 = 0x00000000
  r1 = 0x00000002
  r2 = 0x00000002
  MUL r0, r1, r2 ; r0 = r1*r2
- POST: r0 = 0x00000004
  r1 = 0x00000002
  r2 = 0x00000002

Binary Multiplication

- Multiplication can be performed by adding a list of shifted multiplicands computed according to the digits of the multiplier

```
          1011      multiplicand
X         1101      multiplier
----------------------
          1011  ┐
         0000   │
        1011    ├ shifted multiplicands
       1011     │
          ┘
----------------------
      10001111    product
```

Multiply Instructions

- **The long multiply instructions produce a 64-bit result**
  - **The result is too large to fit a single 32-bit register so the result is placed in two registers labeled RdLo and RdHi**

- **Syntax: <instruction>{<cond>}{S} RdLo, RdHi, Rm, Rs**

SMLAL	Signed multiply accumulate long	[RdHi, RdLo] = [RdHi, RdLo] + (Rm*Rs)
SMULL	Signed multiply long	[RdHi, RdLo] = (Rm*Rs)
UMLAL	Unsigned multiply accumulate long	[RdHi, RdLo] = [RdHi, RdLo] + (Rm*Rs)
UMULL	Unsigned multiply long	[RdHi, RdLo] = (Rm*Rs)

Multiply Instructions

- PRE : r0 = 0x00000000
  r1 = 0x00000000
  r2 = 0xf0000002
  r3 = 0x00000002
  UMULL r0, r1, r2, r3 ; [r1, r0] = r2*r3
- POST: r0 = 0xe0000004 ; = RdLo
  r1 = 0x00000001 ; = RdHi

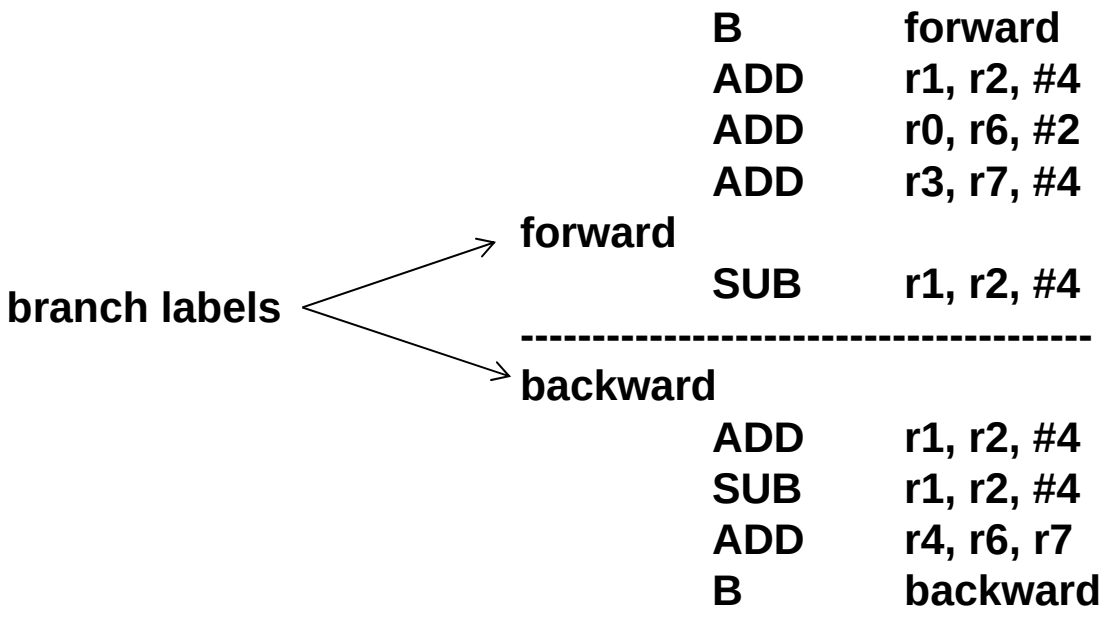
Branch Instructions

- Change the flow of execution or is used to call a routine
- Syntax: B{<cond>} label
  BL{<cond>} label
  BX{<cond>} Rm
  BLX{<cond>} label | Rm

B	Branch	pc = label
BL	Branch with link	pc = label lr = address of the next instruction after the BL
BX	Branch exchange	pc = Rm & 0xfffffffe, T = Rm & 1
BLX	Branch exchange with link	pc = label pc = Rm & 0xfffffffe, T = Rm & 1 lr = address of the next instruction after the BL

Branch Instructions

- The branch labels are placed at the beginning of the line and are used to mark an address that can be used later by the assembler to calculate the branch offset



Branch Instructions

- **The branch with link, or BL, instruction is similar to the B instruction but overwrites the link register lr with a return address**
  - **To return from a subroutine, the link register should be copied to the pc**

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

Branch Instructions

- The branch exchange (BX) and branch exchange with link (BLX) are the third type of branch instruction
  - The BX instruction uses an absolute address stored in register Rm
  - It is primarily used to branch to and from Thumb code
  - The T bit in the cpsr is updated by the least significant bit of the branch register

Load-Store Instructions

- Transfer data between memory and registers
- Syntax: <LDR|STR>{<cond>}{B} Rd, addressing[1]
  LDR {<cond>}{SB|H|SH} Rd, addressing[2]
  STR{<cond>}H Rd, addressing[2]

LDR	Load word into a register	Rd ← mem32[address]
STR	Save byte or word from a register	Rd → mem32[address]
LDRB	Load byte into a register	Rd ← mem8[address]
STRB	Save byte from a register	Rd → mem8[address]
LDRH	Load half word into a register	Rd ← mem16[address]
STRH	Save half word into a register	Rd → mem16[address]
LDRSB	Load signed byte into a register	Rd ← SignExtend (mem32[address])
LDRSH	Load signed half word into a register	Rd → SignExtend (mem32[address])

Load-Store Instructions 빈도 높아서 부가 기능 많다

- LDR and STR instructions can load and store data on a boundary alignment that is the same as the datatype size being loaded or stored
  - LDR can only load 32-bit words on a memory address that is a multiple of four bytes—0, 4, 8, and so on

```
; 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]
```

  - In the above, the offset from register r1 is zero
  - Register r1 is called the base address register

Single-Register Load-Store Addressing Modes

❒ Three addressing modes

Index methods.

Index method	Data	Base address register	Example
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`

Note: ! indicates that the instruction writes the calculated address back to the base address register.

- **In pre-indexed instructions, the offset is calculated and added to the base, and the resulting address is used for the transfer. If writeback is selected, the transfer address is written back into the base register**
- **In post-indexed instructions the offset is calculated and added to the base after the transfer. The base register is always updated by post-indexed instructions.**

Single-Register Load-Store Addressing Modes

```
PRE     r0 = 0x00000000
        r1 = 0x00090000
        mem32[0x00090000] = 0x01010101
        mem32[0x00090004] = 0x02020202

        LDR     r0, [r1, #4] !

POST    r0 = 0x02020202
        r1 = 0x00090004
```

```
PRE     r0 = 0x00000000
        r1 = 0x00090000
        mem32[0x00090000] = 0x01010101
        mem32[0x00090004] = 0x02020202

        LDR     r0, [r1, #4]

POST    r0 = 0x02020202
        r1 = 0x00090000
```

dest. base

```
PRE     r0 = 0x00000000
        r1 = 0x00090000
        mem32[0x00090000] = 0x01010101
        mem32[0x00090004] = 0x02020202

        LDR     r0, [r1], #4

POST(1) r0 = 0x01010101
        r1 = 0x00090004
```

Single-Register Load-Store Addressing Modes

Single-register load-store addressing, word or unsigned byte.

Addressing[1] mode and index method	Addressing[1] syntax
Preindex with immediate offset	`[Rn, #+/-offset_12]`
Preindex with register offset	`[Rn, +/-Rm]`
Preindex with scaled register offset	`[Rn, +/-Rm, shift #shift_imm]`
Preindex writeback with immediate offset	`[Rn, #+/-offset_12]!`
Preindex writeback with register offset	`[Rn, +/-Rm]!`
Preindex writeback with scaled register offset	`[Rn, +/-Rm, shift #shift_imm]!`
Immediate postindexed	`[Rn], #+/-offset_12`
Register postindex	`[Rn], +/-Rm`
Scaled register postindex	`[Rn], +/-Rm, shift #shift_imm`

- Scaled means the address is calculated using the base address register and a barrel shift operation

Single-Register Load-Store Addressing Modes

Table 3.6 Examples of LDR instructions using different addressing modes.

	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)`

Multiple Register Transfer

- **Transfer multiple registers between memory and the processor in a single instruction**

- **Syntax:**

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

LDM	Load multiple registers	{Rd}^{*N} ← mem32[start address + 4*N] optional Rn updated
STM	Store multiple registers	{Rd}^{*N} → mem32[start address + 4*N] optional Rn updated

Multiple Register Transfer

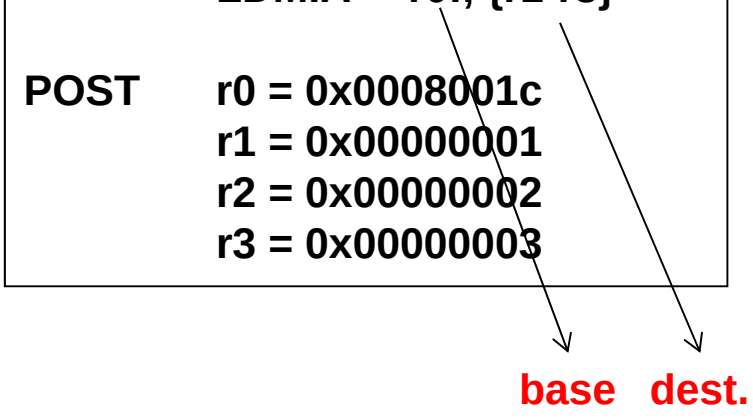
- Any subset of the current bank of registers can be transferred to memory or fetched from memory
  - The base register Rn determines the source or destination address
  - Rn can be updated following the transfer when register Rn is followed by !

Addressing mode for load-store multiple instructions.

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	$Rn - 4*N$	$Rn - 4$	$Rn - 4*N$

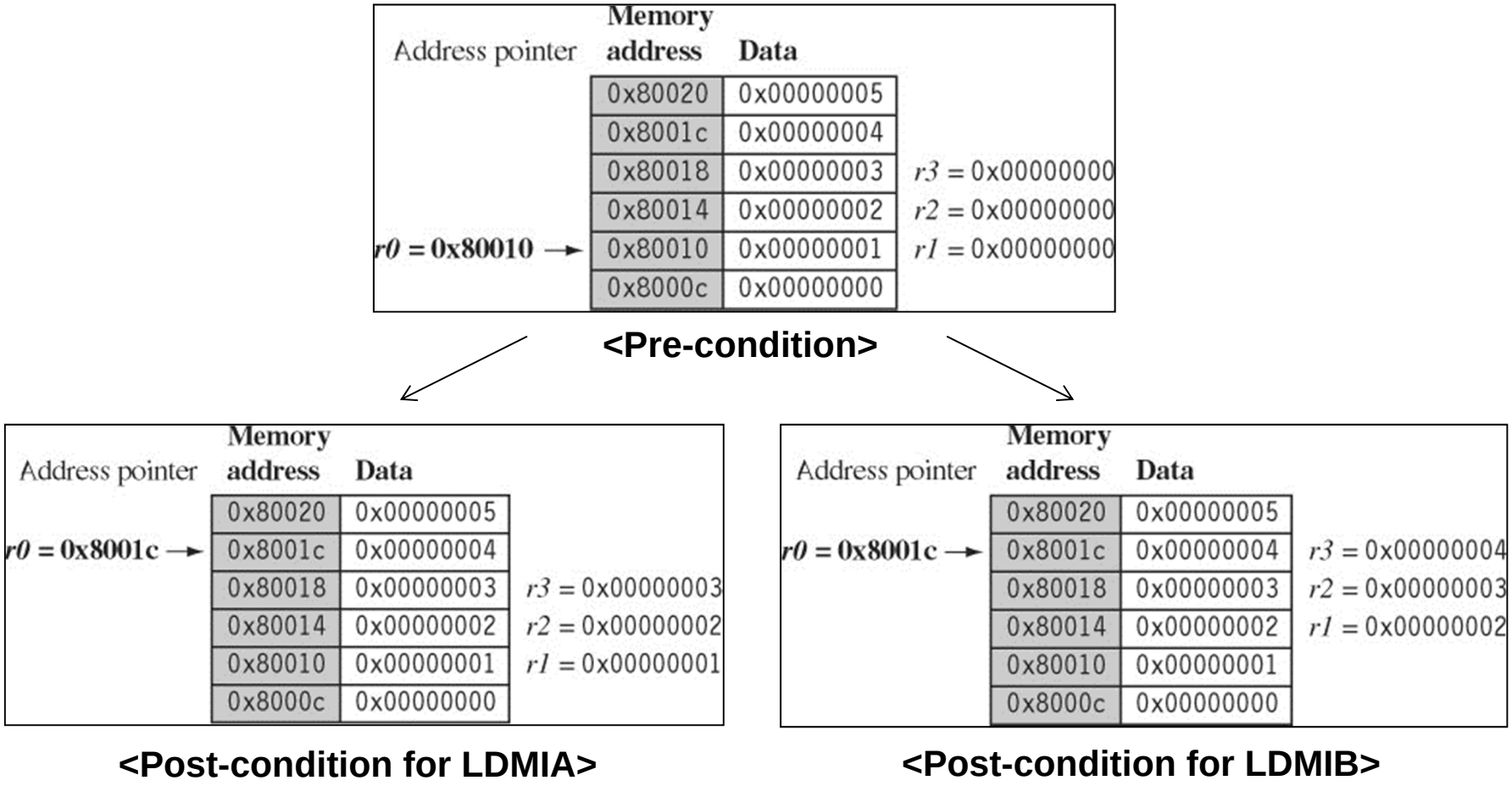
Multiple Register Transfer

```
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
```



- Register r0 is the base register Rn and is followed by !, indicating that the register is updated after the instruction is executed
- "—" character is used to identify a range of registers
- Each register can also be listed, using a comma to separate each register within "{" and "}" brackets

Multiple Register Transfer



Multiple Register Transfer

Load-store multiple pairs when base update used.

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

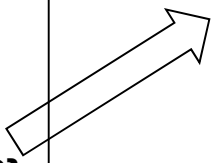
memory copy
커널과 유저 영역 사이에 많이 일어난다.

유저 buffer <- 커널 buffer <- interrupt NIC 데이터 발생
block copy

- If you use a store with base update, then the paired load instruction of the same number of registers will reload the data and restore the base address pointer
  - This is useful when you need to temporarily save a group of registers and restore them later

Multiple Register Transfer

```
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 = 0x0009000c
          r1 = 0x00000001
          r2 = 0x00000002
          r3 = 0x00000003
```



```
mem32[0x00090004] = 0x00000009
mem32[0x00090008] = 0x00000008
mem32[0x0009000c] = 0x00000007
```

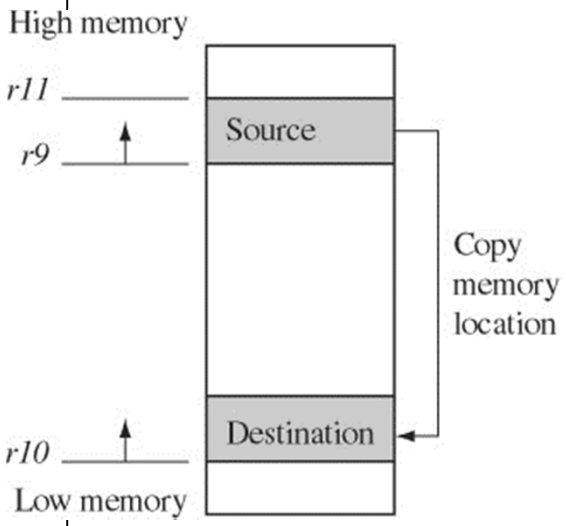
```
          LDMDA   r0!, {r1-r3}

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

The decrement versions DA and DB decrement the start address and then store to ascending memory locations. With the increment and decrement load multiples, you can access arrays forwards or backwards.

Block Memory Copy

- A simple routine that copies blocks of 32 bytes from a some address location to a destination address location

```
; 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                   ; if not, go to loop
```



Multiple Register Transfer

- ARM implementations do not usually interrupt instructions while they are executing
  - On an ARM7, a load multiple instruction takes 2 + Nt cycles, where N is the number of registers to load and t is the number of cycles required for each sequential access to memory
  - If an interrupt has been raised, then it has no effect until the load-store multiple instruction is complete
  - Compilers, such as armcc, provide a switch to control the maximum number of registers being transferred on a load-store, which limits the maximum interrupt latency

Stack Operations

- The ARM architecture uses the load-store multiple instructions to carry out stack operations
  - The pop operation uses a load multiple instruction
  - The push operation uses a store multiple instruction

- Ascending (A) or descending (D)
  - Ascending stacks grow towards higher memory addresses
  - Descending stacks grow towards lower memory addresses
- Full (F) or empty (E)
  - Full stack: stack pointer sp points to an address that is the last used or full location
  - Empty stack: the sp points to an address that is the first unused or empty location

Stack Operations

❒ Addressing methods

Addressing methods for stack operations.

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

Stack Operations

```
PRE     r1  = 0x00000002
        r4  = 0x00000003
        sp = 0x00080014

        STMFD   sp!, {r1, r4}

POST    r1  = 0x00000002
        r4  = 0x00000003
        sp = 0x0008000c
```

PRE

	Address	Data
	0x80018	0x00000001
sp →	0x80014	0x00000002
	0x80010	*Empty*
	0x8000c	*Empty*

POST

	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
sp →	0x8000c	0x00000002

```
PRE     r1  = 0x00000002
        r4  = 0x00000003
        sp = 0x00080010

        STMED   sp!, {r1, r4}

POST    r1  = 0x00000002
        r4  = 0x00000003
        sp = 0x00080008
```

PRE

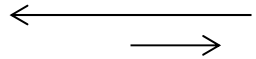
	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
sp →	0x80010	*Empty*
	0x8000c	*Empty*
	0x80008	*Empty*

POST

	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
	0x8000c	0x00000002
sp →	0x80008	*Empty*

Swap Instructions

- Swap the contents of memory with the contents of a register
  - A special case of a load-store instruction
  - And atomic operation



- 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

Swap Instructions

- Swap cannot be interrupted by any other instruction or any other bus access
  - The system "holds the bus" until the transaction is complete

- Swap is particularly useful when implementing semaphores and mutual exclusion in an operating system

```
PRE     mem32[0x9000] = 0x12345678
        r0 = 0x00000000
        r1 = 0x11112222
        r2 = 0x00009000

        SWP     r0, r1, [r2]

PRE     mem32[0x9000] = 0x11112222
        r0 = 0x12345678
        r1 = 0x11112222
        r2 = 0x00009000
```

Software Interrupt Instruction

- Causes a software interrupt exception
  - Provides a mechanism for applications to call operating system routines
- Syntax: SWI{<cond>} SWI_number

SWI	Software interrupt	lr_svc = address of instruction following the SWI spsr_svc = cpsr pc = vectors + 0x8 cpsr mode = SVC cpsr I = 1 (mask IRQ interrupt)

Software Interrupt Instruction

Before and after SWI instruction

```
PRE     cpsr = nzcVqift_USER
        pc = 0x00008000
        lr  = 0x003fffff        ; lr = r14
        r0 = 0x12               ; parameter
                                  parameter convention이 os마다 다르다
        0x00008000 SWI    0x123456
                                  comment field에 들어간다(23-0)
                                  software interrupt number  (system call number)
POST    cpsr = nzcVqIft_SVC
        spsr = nzcVqift_USER
        pc = 0x00000008
        lr  = 0x00008004
        r0 = 0x12
```

SWI_handler

```
; Store registers r0-r12
; and the link register
                STMFD   sp!, {r0-r12, lr}
; Read the SWI instruction
                LDR     r10, [lr, #-4]
; Mask off top 8 bits
                BIC     r10, r10, #0xff000000
; r10 – contains the SWI number
                BL      service_routine
; Return from SWI handler
                LDMFD   sp!, {r0-r12, pc}^
```

Software Interrupt Instruction

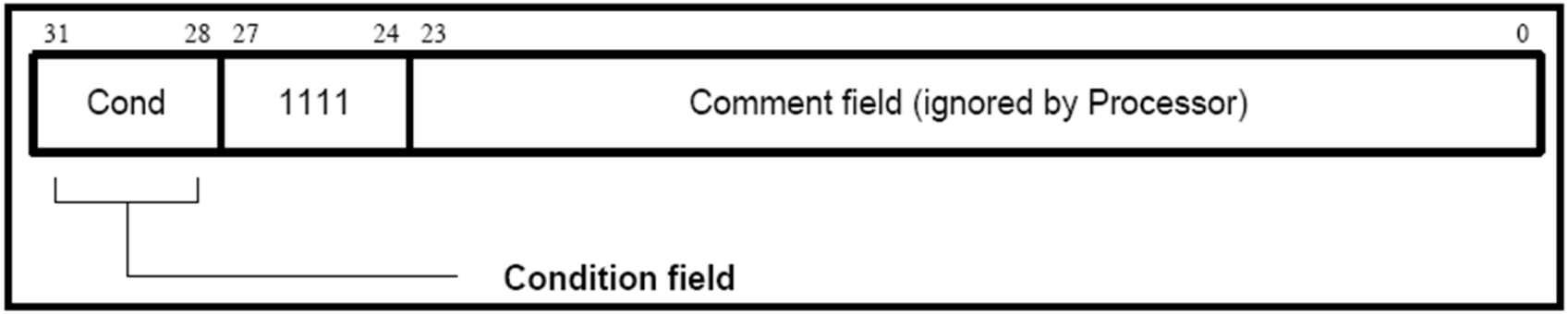


Figure 5-22: Software interrupt instruction

- **The bottom 24 bits of the instruction are ignored by the processor, and may be used to communicate information to the supervisor code**
- **For instance, the supervisor may look at this field and use it to index into an array of entry points for routines which perform the various supervisor functions**

Program Status Register Instructions

- Two instructions to directly control a program status register
  - The MRS instruction transfers the contents of either the cpsr or spsr into a register
  - The MSR instruction transfers the contents of a register into either the cpsr or spsr

- Syntax:

move register spsr

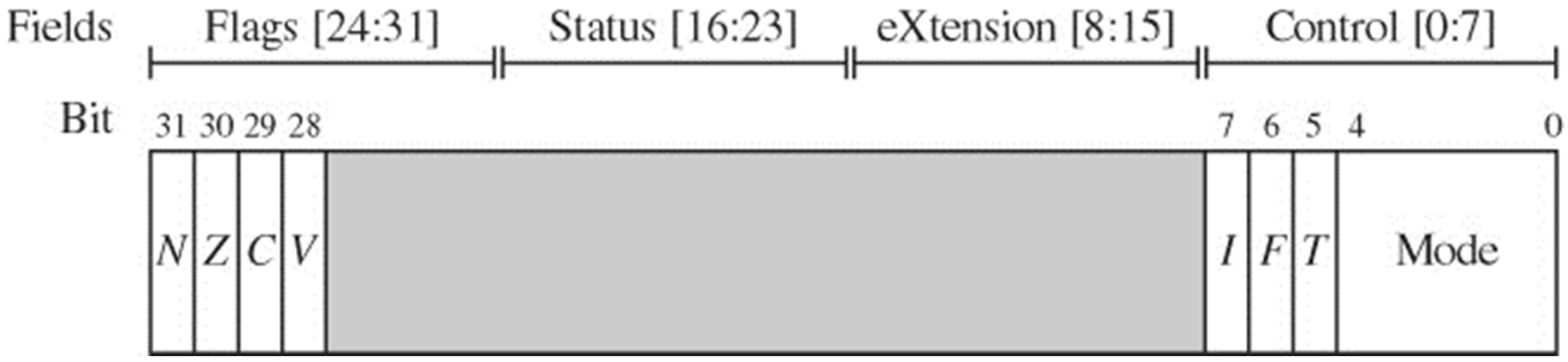
MRS{<cond>} Rd, <cpsr|spsr>

MSR{<cond>} <cpsr|spsr>_<fields>, Rm

MSR{<cond>} <cpsr|spsr>_<fields>, #immediate

Program Status Register Instructions

MRS	Copy program status register to a general-purpose register	Rd = psr
MSR	Move a general-purpose register to a program status register	Psr[field] = Rm
MSR	Move an immediate value to a program status register	Psr[field] = immediate



Program Status Register Instructions

- The MSR first copies the cpsr into register r1
- The BIC instruction clears bit 7 of r1
- Register r1 is then copied back into the cpsr, which enables IRQ interrupts

```
PRE     cpsr = nzcvqIFt_SVC

        MRS     r1, cpsr
        BIC     r1, r1, #0x80       ; r1 = r1 ^ ~0b10000000
        MSR     cpsr, r1

PRE     cpsr = nzcvqIFt_SVC -> nzcvqiFt_SVC
```

Coprocessor Instructions

- **Extend the instruction set**
  - **The most common use of a coprocessor is the system coprocessor to control on-chip functions such as the cache and memory management unit on the ARM720**
  - **A floating-point ARM coprocessor also has been developed, and application-specific coprocessors are a possibility**

CDP	Coprocessor data processing—perform an operation in a coprocessor
MRC **MCR**	Coprocessor register transfer—move data from/to coprocessor registers
LDC **STC**	Coprocessor memory transfer—load and store blocks of memory to/from a coprocessor

Coprocessor Instructions

- **Syntax:**

  CDP{<cond>} cp, opcode1, Cd, Cn {,opcode2}

  <MRC|MCR>{<cond>} cp, opcode1, Rd, Cn, Cm {,opcode2}

  <LDC|STC> {<cond>} cp, Cd, addressing

  - **The cp field represents the coprocessor number between p0 and p15**
  - **The opcode fields describe the operation to take place on the coprocessor**
  - **The Cn, Cm, and Cd fields describe registers within the coprocessor**
  - **The interpretation of the opcode and register fields are coprocessor-dependent**

Coprocessor Instructions

- **The coprocessor operations and registers depend on the specific coprocessor you are using**
  - **Coprocessor (CP15) is reserved for system control purposes, such as memory management, write buffer control, cache control, and identification registers**

```
; transferring the contents of CP15 register c0 to register r10

        MRC     p15, 0, r10, c0, c0, 0
```

  - **CP15 register-0 contains the processor identification number**
    - Copied into the general-purpose register r10

Loading Constants pseudo instruction

- There is no ARM instruction to move a 32-bit constant into a register
  - Since ARM instructions are 32 bits in size, they obviously cannot specify a general 32-bit constant
- There are two pseudoinstructions
  - Syntax: LDR Rd, =constant
    ADR Rd, label

LDR	Load constant	Rd = 32-bit constant
ADR	Load address	Rd = 32-bit relative address

LDR pseudoinstruction conversion.

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

ARMv5E Extensions

- The ARMv5E extensions provide many new instructions

New instructions provided by the ARMv5E extensions.

Instruction	Description
`CLZ {<cond>} Rd, Rm`	count leading zeros
`QADD {<cond>} Rd, Rm, Rn`	signed saturated 32-bit add
`QDADD{<cond>} Rd, Rm, Rn`	signed saturated double 32-bit add
`QDSUB{<cond>} Rd, Rm, Rn`	signed saturated double 32-bit subtract
`QSUB{<cond>} Rd, Rm, Rn`	signed saturated 32-bit subtract
`SMLAxy{<cond>} Rd, Rm, Rs, Rn`	signed multiply accumulate 32-bit (1)
`SMLALxy{<cond>} RdLo, RdHi, Rm, Rs`	signed multiply accumulate 64-bit
`SMLAWy{<cond>} Rd, Rm, Rs, Rn`	signed multiply accumulate 32-bit (2)
`SMULxy{<cond>} Rd, Rm, Rs`	signed multiply (1)
`SMULWy{<cond>} Rd, Rm, Rs`	signed multiply (2)

ARMv5E Extensions

- One of the most important addition is the signed multiply accumulate instructions that operate on 16-bit data
  - These operations are single cycle on many ARMv5E implementations
- ARM5vE provides greater flexibility and efficiency when manipulating 16-bit values, which is important for applications such as 16-bit digital audio processing

Count Leading Zeros Instruction

- Counts the number of zeros between the most significant bit and the first bit set to 1

```
PRE     r1 = 0b00000000000000000000000000010000

        CLZ     r0, r1

POST    r0 = 27
```

Saturated Arithmetic

- Normal ARM instructions wrap around when you overflow an integer value
  - 0x7fffffff + 1 = - 0x80000000
  - Note that 0x7fffffff is the maximum positive value you can store in 32 bits

```
PRE     cpsr = nzcvqiFt_SVC
        r0 = 0x00000000
        r1 = 0x70000000 (positive)
        r2 = 0x7fffffff (positive)

        ADDS    r0, r1, r2

POST    cpsr = NzcVqiFt_SVC
        r0 = 0xefffffff (negative)
```

Saturated Arithmetic

- Using the ARMv5E instructions, you can saturate the result—once the highest number is exceeded the results remain at the maximum value of 0x7fffffff

Saturation instructions.

Instruction	Saturated calculation
QADD	$Rd = Rn + Rm$
QDADD	$Rd = Rn + (Rm*2)$
QSUB	$Rd = Rn - Rm$
QDSUB	$Rd = Rn - (Rm*2)$

```
PRE     cpsr = nzcvqiFt_SVC
        r0 = 0x00000000
        r1 = 0x70000000 (positive)
        r2 = 0x7fffffff (positive)

        QADD    r0, r1, r2

POST    cpsr = nzcvQiFt_SVC
        r0 = 0x7fffffff
```

ARMv5E Multiply Instructions

Table 3.15 Signed multiply and multiply accumulate instructions.

Instruction	Signed Multiply [Accumulate]	Signed result	Q flag updated	Calculation
SMLAxy	(16-bit *16-bit)+ 32-bit	32-bit	yes	$Rd = (Rm.x * Rs.y) + Rn$
SMLALxy	(16-bit *16-bit)+ 64-bit	64-bit	—	$[RdHi, RdLo] += Rm.x * Rs.y$
SMLAWy	((32-bit *16-bit) ≫ 16)+ 32-bit	32-bit	yes	$Rd = ((Rm * Rs.y) \gg 16) + Rn$
SMULxy	(16-bit *16-bit)	32-bit	—	$Rd = Rm.x * Rs.y$
SMULWy	((32-bit *16-bit)≫ 16)	32-bit	—	$Rd = (Rm * Rs.y) \gg 16$

- x and y select which 16 bits of a 32-bit register are used for the first and second operands, respectively
  - These fields are set to a letter T for the top 16-bits, or the letter B for the bottom 16 bits
  - For multiply accumulate operations with a 32-bit result, the Q flag indicates if the accumulate overflowed a signed 32-bit value