MODULE – 02

INTRODUCTION TO THE ARM INSTRUCTION SET

INTRODUCTION TO THE ARM INSTRUCTION SET

Different ARM architecture revisions support different instructions. However, new revisions usually add instructions and remain backwardly compatible. Code you write for architecture ARMv4T should execute on an ARMv5TE processor.

The following Table provides a complete list of ARM instructions available in the *ARMv5E instruction set architecture* (*ISA*). This ISA includes all the core ARM instructions as well as some of the newer features in the ARM instruction set.

Table: 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

Mnemonics	ARM ISA	Description
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
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

Mnemonics	ARM ISA	Description
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 × 16) + 32 = 32-bit)
SMLAL	v3M	signed multiply accumulate long ((32 × 32) + 64 = 64-bit)
SMLAL*xy*	v5E	signed multiply accumulate long ((16 × 16) + 64 = 64-bit)
SMLAW*y*	v5E	signed multiply accumulate instruction (((32 × 16) ≫ 16) + 32 = 32-bit)
SMULL	v3M	signed multiply long (32 × 32 = 64-bit)

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)

In the following sections, the hexadecimal numbers are represented with the prefix *0x* and binary numbers with the prefix *0b*. The examples follow this format:

PRE *<pre-conditions>*

<instruction/s>

POST *<post-conditions>*

In the pre- and post-conditions, memory is denoted as

mem<data_size>[address]

This refers to *data_size* bits of memory starting at the given byte address. For example, *mem32[1024]* is the 32-bit value starting at address 1 KB.

ARM instructions process data held in registers and memory is accessed only with load and store instructions.

ARM instructions commonly take two or three operands. For instance, the ADD instruction below adds the two values stored in registers *r1* and *r2* (the source registers). It writes the result to register *r3* (the destination register).

Instruction Syntax	Destination register (*Rd*)	Source register 1 (*Rn*)	Source register 2 (*Rm*)
ADD r3, r1, r2	*r3*	*r1*	*r2*

ARM instructions classified as—data processing instructions, branch instructions, load-store instructions, software interrupt instruction, and program status register instructions.

DATA PROCESSING INSTRUCTIONS:

The data processing instructions manipulate data within registers. They are—

- ✓ move instructions, arithmetic instructions, logical instructions, comparison instructions, and multiply instructions.

Most data processing instructions can process one of their operands using the barrel shifter.

If you use the S suffix on a data processing instruction, then it updates the flags in the *cpsr*.

Move and logical operations update the carry flag *C*, negative flag *N*, and zero flag *Z*.

- The *C* flag is set from the result of the barrel shift as the last bit shifted out.
- The *N* flag is set to bit 31 of the result.
- The *Z* flag is set if the result is zero.

MOVe Instructions:

Move instruction copies *N* into a destination register *Rd*, where *N* is a register or immediate value. This instruction is useful 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	$Rd = \sim N$

Example: This example shows a simple move instruction. The MOV instruction takes the contents of register *r5* and copies them into register *r7*, in this case, taking the value *5*, and overwriting the value *8* in register *r7*.

```
PRE   r5 = 5
      r7 = 8
MOV   r7, r5   ; let r7 = r5
POST  r5 = 5
      r7 = 5
```

Barrel Shifter:

In above Example, we showed a MOV instruction where *N* is a simple register. But *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.

- ✓ Data processing instructions are processed within the arithmetic logic unit (ALU).
- ✓ A unique and powerful feature of the ARM processor is the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU.
- ✓ Pre-processing or shift occurs within the cycle time of the instruction.
  - o This shift increases the power and flexibility of many data processing operations.
  - o This is particularly useful for loading constants into a register and achieving fast multiplies or division by a power of 2.
- ✓ There are data processing instructions that do not use the barrel shift, for example, the MUL (multiply), CLZ (count leading zeros), and QADD (signed saturated 32-bit add) instructions.

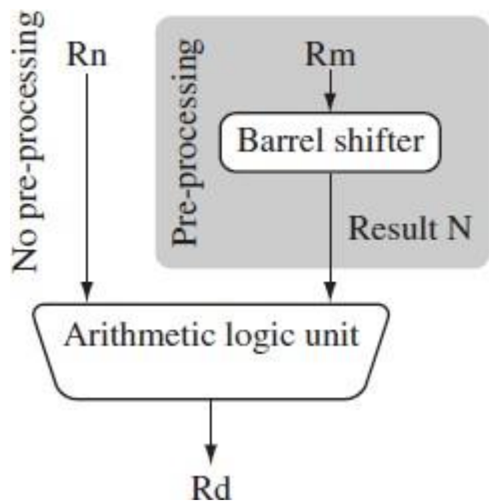

Figure: Barrel Shifter and ALU

- ✓ Figure shows the data flow between the ALU and the barrel shifter.
- ✓ Register *Rn* enters the ALU without any pre- processing of registers.
- ✓ We apply a logical shift left (LSL) to register *Rm* before moving it to the destination register. This is the same as applying the standard *C* language shift operator « to the register.

- ✓ The MOV instruction copies the shift operator result *N* into register *Rd*. *N* represents the result of the LSL operation described in the following Table.

Table: Barrel Shifter Operations

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

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

- ✓ The five different shift operations that you can use within the barrel shifter are summarized in the above Table.

PRE *r5 = 5*

r7 = 8

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

POST *r5 = 5*

r7 = 20

- ✓ The above example multiplies register *r5* by four and then places the result into register *r7*.
- ✓ The following Figure illustrates a logical shift left by one.

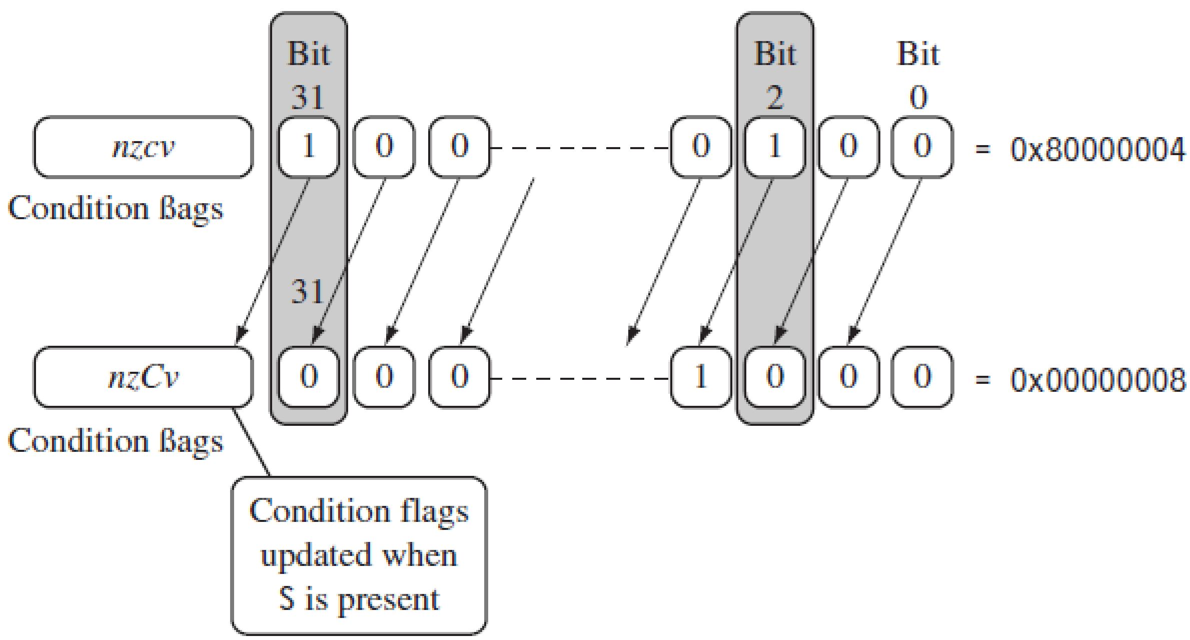

Figure: Logical Shift Left by One

- ✓ For example, the contents of bit *0* are shifted to bit *1*. Bit 0 is cleared. The C flag is updated with the last bit shifted out of the register. This is bit (32 - y) of the original value, where *y* is the shift amount. When *y* is greater than one, then a shift by *y* positions is the same as a shift by one position executed *y* times.

Example: This example of a MOVS instruction shifts register *r1* left by one bit. This multiplies register *r1* by a value 2^1. As you can see, the C flag is updated in the *cpsr* because the S suffix is present in the instruction mnemonic.

PRE *cpsr = nzcvqiFt_USER*

r0 = 0x00000000

r1 = 0x80000004

MOVS r0, r1, LSL #1

POST *cpsr = nzCvqiFt_USER*

r0 = 0x00000008

r1 = 0x80000004

The following Table lists the syntax for the different barrel shift operations available on data processing instructions. The second operand *N* can be an immediate constant preceded by #, a register value *Rm*, or the value of *Rm* processed by a shift.

Table: Barrel Shifter Operation Syntax for data Processing Instructions

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`

Arithmetic Instructions:

The arithmetic instructions implement addition and subtraction of 32-bit signed and unsigned values.

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

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 flag)`
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$ `!(carry flag)`
SUB	subtract two 32-bit values	$Rd = Rn - N$

N is the result of the shifter operation.

Example: The following simple subtract instruction subtracts a value stored in register *r2* from a value stored in register *r1*. The result is stored in register *r0*.

PRE *r0 = 0x00000000*

r1 = 0x00000002

r2 = 0x00000001

SUB r0, r1, r2

POST *r0 = 0x00000001*

Example: The following reverse subtract instruction (RSB) subtracts *r1* from the constant value #0, writing the result to *r0*. You can use this instruction to negate numbers.

PRE *r0 = 0x00000000*

r1 = 0x00000077

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

POST *r0 = -r1 = 0xffffff89*

Example: The SUBS instruction is useful for decrementing loop counters. In this example, we subtract the immediate value one from the value one stored in register *r1*. The result value zero is written to register *r1*. The *cpsr* is updated with the ZC flags being set.

PRE *cpsr = nzcvqiFt_USER*

r1 = 0x00000001

SUBS r1, r1, #1

POST *cpsr = nZCvqiFt_USER*

r1 = 0x00000000

Using the Barrel Shifter with Arithmetic Instructions:

The wide range of second operand shifts available on arithmetic and logical instructions is a very powerful feature of the ARM instruction set. The following Example illustrates the use of the inline barrel shifter with an arithmetic instruction. The instruction multiplies the value stored in register *r1* by three.

Example: Register *r1* is first shifted one location to the left to give the value of twice *r1*. The ADD instruction then adds the result of the barrel shift operation to register *r1*. The final result transferred into register *r0* is equal to three times the value stored in register *r1*.

PRE *r0 = 0x00000000*

r1 = 0x00000005

ADD r0, r1, r1, LSL #1

POST *r0 = 0x0000000f*

r1 = 0x00000005

Logical Instructions:

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	$Rd = Rn \mid N$
EOR	logical exclusive OR of two 32-bit values	$Rd = Rn \wedge N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \& \sim N$

Example: This example shows a logical OR operation between registers *r1* and *r2*. Register *r0* holds the result.

PRE *r0 = 0x00000000*

r1 = 0x02040608

r2 = 0x10305070

ORR r0, r1, r2

POST *r0 = 0x12345678*

Example: This example shows a more complicated logical instruction called BIC, which carries out a logical bit clear.

PRE *r1 = 0b1111*

r2 = 0b0101

BIC r0, r1, r2

POST *r0 = 0b1010*

This is equivalent to – *Rd = Rn* ***AND NOT*** *(N)*

In this example, register *r2* contains a binary pattern where every binary *1* in *r2* clears a corresponding bit location in register *r1*.

This instruction is particularly useful when clearing status bits and is frequently used to change interrupt masks in the *cpsr*.

NOTE: The logical instructions update the *cpsr* flags only if the S suffix is present. These instructions can use barrel-shifted second operands in the same way as the arithmetic instructions.

Comparison Instructions:

- ✓ The comparison instructions are used to compare or test a register with a 32-bit value.
- ✓ They update the *cpsr* flag bits according to the result, but do not affect other registers.

- ✓ After the bits have been set, the information can then be used to change program flow by using conditional execution.
- ✓ It is not required to apply the S suffix for comparison instructions to update the flags.

```
Syntax: <instruction>{<cond>} 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 $Rn \wedge N$
TST	test bits of a 32-bit value	flags set as a result of $Rn \,\&\, N$

N is the result of the shifter operation.

Example: This example shows a CMP comparison instruction. You can see that both registers, *r0* and *r9*, are equal before executing the instruction. The value of the *Z* flag prior to execution is *0* and is represented by a lowercase *z*. After execution the *Z* flag changes to *1* or an uppercase *Z*. This change indicates equality.

PRE *cpsr = nzcvqiFt_USER*

r0 = 4

r9 = 4

CMP r0, r9

POST *cpsr = nZcvqiFt_USER*

- ✓ The CMP is effectively a subtract instruction with the result discarded; similarly the TST instruction is a logical AND operation, and TEQ is a logical exclusive OR operation.
- ✓ For each, the results are discarded but the condition bits are updated in the *cpsr*.
- ✓ It is important to understand that comparison instructions only modify the condition flags of the *cpsr* and do not affect the registers being compared.

Multiply Instructions:

The multiply instructions multiply the contents of a pair of registers and, depending upon the instruction, accumulate the results in with another register.

The long multiplies accumulate onto a pair of registers representing a 64-bit value. The final result is placed in a destination register or a pair of registers.

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

MLA	multiply and accumulate	$Rd = (Rm^{*}Rs) + Rn$
MUL	multiply	$Rd = Rm^{*}Rs$

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	$[RdHi, RdLo] = Rm * Rs$

The number of cycles taken to execute a multiply instruction depends on the processor implementation. For some implementations the cycle timing also depends on the value in *Rs*.

Example: This example shows a simple multiply instruction that multiplies registers *r1* and *r2* together and places the result into register *r0*. In this example, register *r1* is equal to the value *2*, and *r2* is equal to *2*. The result, *4*, is then placed into register *r0*.

PRE *r0 = 0x00000000*

r1 = 0x00000002

r2 = 0x00000002

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

POST *r0 = 0x00000004*

r1 = 0x00000002

r2 = 0x00000002

The long multiply instructions (SMLAL, SMULL, UMLAL, and UMULL) 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*. *RdLo* holds the lower 32 bits of the 64-bit result, and *RdHi* holds the higher 32 bits of the 64-bit result. The following shows an example of a long unsigned multiply instruction.

Example: The instruction multiplies registers *r2* and *r3* and places the result into register *r0* and *r1*. Register *r0* contains the lower 32 bits, and register *r1* contains the higher 32 bits of the 64-bit result.

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:

A branch instruction changes the flow of execution or is used to call a routine. This type of instruction allows programs to have subroutines, if-then-else structures, and loops.

The change of execution flow forces the program counter *pc* to point to a new address. The ARMv5E instruction set includes four different branch instructions.

```
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*, *T* = 1 *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 *lr* = address of the next instruction after the BLX

- ✓ The address *label* is stored in the instruction as a signed *pc*-relative offset and must be within approximately 32 MB of the branch instruction.
- ✓ *T* refers to the Thumb bit in the *cpsr*. When instructions set *T*, the ARM switches to Thumb state.

Example: This example shows a forward and backward branch. Because these loops are address specific, we do not include the pre- and post-conditions. The forward branch skips three instructions. The backward branch creates an infinite loop.

```
        B       forward
        ADD r1, r2, #4
        ADD r0, r6, #2
        ADD r3, r7, #4
forward
        SUB r1, r2, #4
 ---------------------------------------------------
backward
        ADD r1, r2, #4
        SUB r1, r2, #4
        ADD    r4, r6, r7
        B      backward
```

In this example, *forward* and *backward* are the labels. 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.

- ✓ The branch with link, or BL, instruction is similar to the B instruction but overwrites the link register *lr* with a return address. It performs a subroutine call.

Example: This example shows a simple fragment of code that, branches to a subroutine using the BL instruction. To return from a subroutine, you copy the link register 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
        <subroutine code>
        MOV pc, lr              ; return by moving pc = lr
```

- ✓ 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.
- ✓ Similarly the BLX instruction updates the *T* bit of the *cpsr* with the least significant bit and additionally sets the link register with the return address.

LOAD-STORE INSTRUCTIONS:

Load-store instructions transfer data between memory and processor registers. There are three types of load-store instructions: single-register transfer, multiple-register transfer, and swap.

Single-Register Transfer:

- ✓ These instructions are used for moving a single data item in and out of a register.
- ✓ The data types supported are signed and unsigned words (32-bit), half-words (16-bit), and bytes.

Here are the various load-store single-register transfer instructions.

```
Syntax: <LDR|STR>{<cond>}{B} Rd,addressing1
        LDR{<cond>}SB|H|SH Rd, addressing2
        STR{<cond>}H Rd, addressing2
```

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 halfword into a register	*Rd <- mem16[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*
LDRSB	load signed byte into a register	*Rd <- SignExtend (mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend (mem16[address])*

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

Example: This example shows a load from a memory address contained in register *r1*, followed by a store back to the same address in memory.

```
;
; 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]
```

The first instruction loads a word from the address stored in register *r1* and places it into register *r0*. The second instruction goes the other way by storing the contents of register *r0* to the address contained in register *r1*. The offset from register *r1* is zero. Register *r1* is called the *base address register*.

Single-Register Load-Store Addressing Modes:

The ARM instruction set provides different modes for addressing memory. These modes incorporate one of the indexing methods: preindex with writeback, preindex, and postindex.

Table: 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.

- ✓ *Preindex with writeback* calculates an address from a base register plus address offset and then updates that address base register with the new address.
- ✓ *Preindex* offset is the same as the preindex with writeback but does not update the address base register.
    - o The preindex mode is useful for accessing an element in a data structure.
- ✓ *Postindex* only updates the address base register after the address is used.
    - o The postindex and preindex with writeback modes are useful for traversing an array.

Example:

PRE *r0 = 0x00000000*

r1 = 0x00090000

mem32[0x00009000] = 0x01010101

mem32[0x00009004] = 0x02020202

LDR r0, [r1, #4]!

Preindexing with writeback:

POST(1) *r0 = 0x02020202*

r1 = 0x00009004

LDR r0, [r1, #4]

Preindexing:

POST(2) *r0 = 0x02020202*

r1 = 0x00009000

LDR r0, [r1], #4

Postindexing:

POST(3) *r0 = 0x01010101*

r1 = 0x00009004

- ✓ The above Example used a preindex method. This example shows how each indexing method affects the address held in register *r1*, as well as the data loaded into register *r0*.

The addressing modes available with a particular load or store instruction depend on the instruction class. The following Table shows the addressing modes available for load and store of a 32-bit word or an unsigned byte.

Table: 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`

- ✓ A signed offset or register is denoted by "+/-", identifying that it is either a positive or negative offset from the base address register *Rn*. The base address register is a pointer to a byte in memory, and the offset specifies a number of bytes.
- ✓ Immediate means the address is calculated using the base address register and a 12-bit offset encoded in the instruction.
- ✓ Register means the address is calculated using the base address register and a specific register's contents.
- ✓ Scaled means the address is calculated using the base address register and a barrel shift operation.

The following Table provides an example of the different variations of the LDR instruction.

Table: 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)`

The following Table shows the addressing modes available on load and store instructions using 16-bit halfword or signed byte data.

Table: Single-Register Load-Store Addressing, Halfword, Signed Halfword, Signed Byte and Doubleword

Addressing[2] mode and index method	Addressing[2] syntax
Preindex immediate offset	[Rn, #+/-offset_8]
Preindex register offset	[Rn, +/-Rm]
Preindex writeback immediate offset	[Rn, #+/-offset_8]!
Preindex writeback register offset	[Rn, +/-Rm]!
Immediate postindexed	[Rn], #+/-offset_8
Register postindexed	[Rn], +/-Rm

These operations cannot use the barrel shifter. There are no STRSB or STRSH instructions since STRH stores both a signed and unsigned halfword; similarly STRB stores signed and unsigned bytes.

The following Table shows the variations for STRH instructions.

Table: Variations of STRH Instructions

	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

Multiple-Register Transfer:

- ✓ Load-store multiple instructions can transfer multiple registers between memory and the processor in a single instruction.
- ✓ The transfer occurs from a base address register *Rn* pointing into memory.
  - o Multiple-register transfer instructions are more efficient from single-register transfers for
    - ▪ moving blocks of data around memory and
    - ▪ saving and restoring context and stacks.
- ✓ Load-store multiple instructions can increase interrupt latency.
- ✓ ARM implementations do not usually interrupt instructions while they are executing.
  - o For example, 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.

```
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

The following Table shows the different addressing modes for the load-store multiple instructions. Here *N* is the number of registers in the list of registers.

Table: 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	$Rn - 4^*N$	$Rn - 4$	$Rn - 4^*N$

- ✓ 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 for a load-store multiple instruction. This register can be optionally updated following the transfer. This occurs when register *Rn* is followed by the ! character, similar to the single-register load-store using preindex with writeback.

Example: In this example, register *r0* is the base register *Rn* and is followed by !, indicating that the register is updated after the instruction is executed. You will notice within the load multiple instruction that the registers are not individually listed. Instead the "-" character is used to identify a range of registers. In this case the range is from register *r1* to *r3* inclusive.

Each register can also be listed, using a comma to separate each register within "{" and "}" brackets.

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

The following Figure shows a graphical representation.

Address pointer	Memory address	Data	
	0x80020	0x00000005	
	0x8001c	0x00000004	
	0x80018	0x00000003	*r3* = 0x00000000
	0x80014	0x00000002	*r2* = 0x00000000
r0 = 0x80010 →	0x80010	0x00000001	*r1* = 0x00000000
	0x8000c	0x00000000	

Figure: Pre-condition for LDMIA Instruction

- ✓ The base register *r0* points to memory address 0x80010 in the PRE condition.
- ✓ Memory addresses 0x80010, 0x80014, and 0x80018 contain the values 1, 2, and 3 respectively.
- ✓ After the load multiple instruction executes, registers *r1*, *r2*, and *r3* contain these values as shown in the following Figure.

Address pointer	Memory address	Data	
	0x80020	0x00000005	
r0 = 0x8001c →	0x8001c	0x00000004	
	0x80018	0x00000003	*r3* = 0x00000003
	0x80014	0x00000002	*r2* = 0x00000002
	0x80010	0x00000001	*r1* = 0x00000001
	0x8000c	0x00000000	

Figure: Post Condition for LDMIA Instruction

- ✓ The base register *r0* now points to memory address 0x8001c after the last loaded word.
- ✓ Now replace the LDMIA instruction with a load multiple and increment before LDMIB instruction and use the same PRE conditions.
- ✓ The first word pointed to by register *r0* is ignored and register *r1* is loaded from the next memory location as shown in the following Figure.

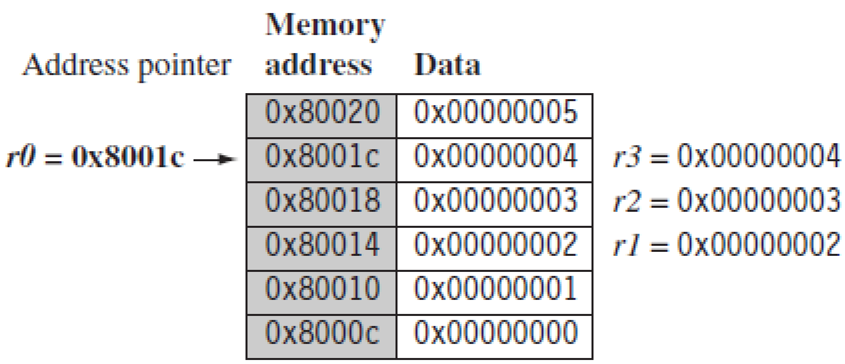

Figure: Post Condition for LDMIB Instruction

- ✓ After execution, register *r0* now points to the last loaded memory location. This is in contrast with the LDMIA example, which pointed to the next memory location.

- The decrement versions DA and DB of the load-store multiple instructions decrement the start address and then store to ascending memory locations.
- This is equivalent to descending memory but accessing the register list in reverse order.
- With the increment and decrement load multiples; you can access arrays forwards or backwards.
- They also allow for stack push and pull operations.

The following Table shows a list of load-store multiple instruction pairs.

Table: Load-Store Multiple Pairs when Base Update used

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

- 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.

Example: This example shows an STM *increment before* instruction followed by an LDM *decrement after* instruction.

PRE *r0 = 0x00009000*

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 = 0x00009000*

r1 = 0x00000009

r2 = 0x00000008

r3 = 0x00000007

The STMIB instruction stores the values 7, 8, 9 to memory. We then corrupt register *r1* to *r3*. The LDMDA reloads the original values and restores the base pointer *r0*.

Example: We illustrate the use of the load-store multiple instructions with a block memory copy example. This example is a simple routine that copies blocks of 32 bytes from a source address location to a destination address location.

The example has two load-store multiple instructions, which use the same increment after addressing mode.

```
; 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
```

- ✓ This routine relies on registers *r9*, *r10*, and *r11* being set up before the code is executed.
- ✓ Registers *r9* and *r11* determine the data to be copied, and register *r10* points to the destination in memory for the data.
- ✓ LDMIA loads the data pointed to by register *r9* into registers *r0* to *r7*. It also updates *r9* to point to the next block of data to be copied.
- ✓ STMIA copies the contents of registers *r0* to *r7* to the destination memory address pointed to by register *r10*. It also updates *r10* to point to the next destination location.

- ✓ CMP and BNE compare pointers *r9* and *r11* to check whether the end of the block copy has been reached.
- ✓ If the block copy is complete, then the routine finishes; otherwise the loop repeats with the updated values of register *r9* and *r10*.
- The BNE is the branch instruction B with a condition mnemonic NE (not equal). If the previous compare instruction sets the condition flags to not equal, the branch instruction is executed.

The following Figure shows the memory map of the block memory copy and how the routine moves through memory.

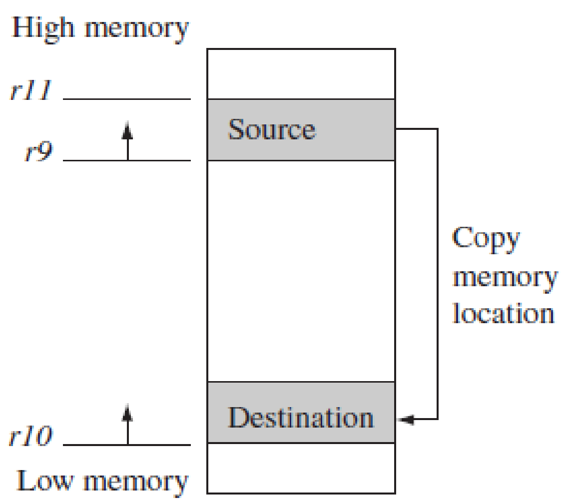

Figure: Block Memory Copy in the Memory map

Theoretically this loop can transfer 32 bytes (8 words) in two instructions, for a maximum possible throughput of 46 MB/second being transferred at 33 MHz. These numbers assume a perfect memory system with fast memory.

Stock Operation: The ARM architecture uses the load-store multiple instructions to carry out stack operations.

- The *pop operation* (removing data from a stack) uses a load multiple instruction.
- The *push operation* (placing data onto the stack) uses a store multiple instruction.

- ✓ When using a stack you have to decide whether the stack will grow up or down in memory.
  - A stack is either –
    - *ascending (A)* – stacks grow towards higher memory addresses or
    - *descending (D)* – stacks grow towards lower memory addresses.
- ✓ When you use a *full stack (F)*, the stack pointer *sp* points to an address that is the last used or full location (i.e., *sp* points to the last item on the stack).

- ✓ If you use an *empty stack (E)* the *sp* points to an address that is the first unused or empty location (i.e., it points after the last item on the stack).

- There are number of load-store multiple addressing mode aliases available to support stack operations (see the following Table).

Table: 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

- Next to the *pop* column is the actual load multiple instruction equivalent.
    - For example, a full ascending stack would have the notation FA appended to the load multiple instruction—LDMFA. This would be translated into an LDMDA instruction.
- ARM has specified an *ARM-Thumb Procedure Call Standard (ATPCS)* that defines how routines are called and how registers are allocated. In the ATPCS, stacks are defined as being full descending stacks. Thus, the LDMFD and STMFD instructions provide the *pop* and *push* functions, respectively.

Example: The STMFD instruction pushes registers onto the stack, updating the *sp*. The following Figure shows a *push* onto a full descending stack.

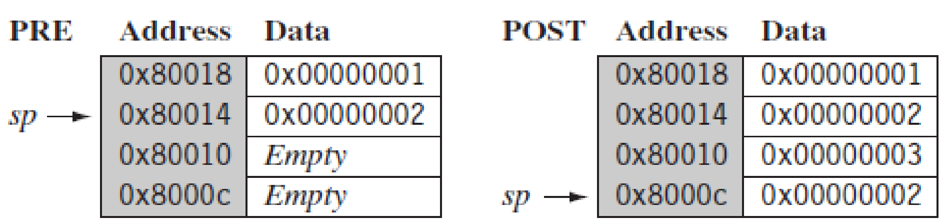

Figure: STMFD Instruction – Full Stack *push* Operation

You can see that when the stack grows the stack pointer points to the last full entry in the stack.

PRE *r1 = 0x00000002*

r4 = 0x00000003

sp = 0x00080014

STMFD sp!, {r1, r4}

POST *r1 = 0x00000002*

r4 = 0x00000003

sp = 0x0008000c

Example: The following Figure shows a *push* operation on an empty stack using the STMED instruction.

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*

Figure: STMED Instruction – Empty Stack *push* Operation

The STMED instruction pushes the registers onto the stack but updates register *sp* to point to the next empty location.

PRE *r1 = 0x00000002*

r4 = 0x00000003

sp = 0x00080010

STMED sp!, {r1, r4}

POST *r1 = 0x00000002*

r4 = 0x00000003

sp = 0x00080008

- ✓ When handling a checked stack there are three attributes that need to be preserved: the stack base, the stack pointer, and the stack limit.
- ✓ The stack base is the starting address of the stack in memory.
- ✓ The stack pointer initially points to the stack base; as data is pushed onto the stack, the stack pointer descends memory and continuously points to the top of stack. If the stack pointer passes the stack limit, then a stack overflow error has occurred.
- ✓ Here is a small piece of code that checks for stack overflow errors for a descending stack:

```
; check for stack overflow
        SUB sp, sp, #size
        CMP sp, r10
        BLLO _stack_overflow        ; condition
```

- ATPCS defines register *r10* as the stack limit or *sl*. This is optional since it is only used when stack checking is enabled.
- The BLLO instruction is a branch with link instruction plus the condition mnemonic LO.
  - If *sp* is less than register *r10* after the new items are pushed onto the stack, then *stack overflow* error has occurred.
  - If the stack pointer goes back past the stack base, then a *stack underflow* error has occurred.

Swap Instruction:

The swap instruction is a special case of a load-store instruction. It swaps the contents of memory with the contents of a register.

This instruction is an *atomic operation*—it reads and writes a location in the same bus operation, preventing any other instruction from reading or writing to that location until it completes.

```
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 cannot be interrupted by any other instruction or any other bus access. We say the system "holds the bus" until the transaction is complete. Also, swap instruction allows for both a word and a byte swap.

Example: The swap instruction loads a word from memory into register *r0* and overwrites the memory with register *r1*.

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

Example: This example shows a simple data guard that can be used to protect data from being written by another task. The SWP instruction "holds the bus" until the transaction is complete.

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

The address pointed to by the semaphore either contains the value *0* or *1*. When the semaphore equals *1*, then the service in question is being used by another process. The routine will continue to loop around until the service is released by the other process—in other words, when the semaphore address location contains the value *0*.

SOFTWARE INTERRUPT INSTRUCTION:

A *software interrupt instruction (SWI)* causes a software interrupt exception, which 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 interrupts)

When the processor executes an SWI instruction, it sets the program counter *pc* to the offset *0x8* in the vector table. The instruction also forces the processor mode to SVC, which allows an operating system routine to be called in a privileged mode.

Each SWI instruction has an associated SWI number, which is used to represent a particular function call or feature.

Example: Here we have a simple example of an SWI call with SWI number 0x123456, used by ARM toolkits as a debugging SWI. Typically the SWI instruction is executed in user mode.

```
PRE     cpsr = nzcVqift_USER
        pc = 0x00008000
        lr = 0x003fffff        ;lr = r14
        r0 = 0x12
0x00008000    SWI    0x123456
POST cpsr = nzcVqIft_SVC
        spsr = nzcVqift_USER
        pc = 0x00000008
        lr = 0x00008004
        r0 = 0x12
```

Since SWI instructions are used to call operating system routines, you need some form of parameter passing. This is achieved using registers. In this example, register *r0* is used to pass the parameter *0x12*. The return values are also passed back via registers.

Code called the ***SWI handler*** is required to process the SWI call. The handler obtains the SWI number using the address of the executed instruction, which is calculated from the link register *lr*.

The SWI number is determined by

SWI_Number = <SWI instruction> ***AND NOT*** *(0xff000000)*

Here the *SWI instruction* is the actual 32-bit SWI instruction executed by the processor.

Example: This example shows the start of an SWI handler implementation. The code fragment determines what SWI number is being called and places that number into register *r10*.

You can see from this example that the load instruction first copies the complete SWI instruction into register *r10*. The BIC instruction masks off the top bits of the instruction, leaving the SWI number. We assume the SWI has been called from ARM state.

```
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}^
```

The number in register *r10* is then used by the SWI handler to call the appropriate SWI service routine.

PROGRAM STATUS REGISTER INSTRUCTIONS:

The ARM instruction set provides two instructions to directly control a *program status register (psr)*.

- ✓ 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 the *cpsr* or *spsr*.

Together these instructions are used to read and write the *cpsr* and *spsr*.

In the syntax we can see a *label* called fields. This can be any combination of *control (c)*, *extension (x)*, *status (s)*, and *flags (f)*.

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

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*

These fields relate to particular byte regions in a *psr*, as shown in the following Figure.

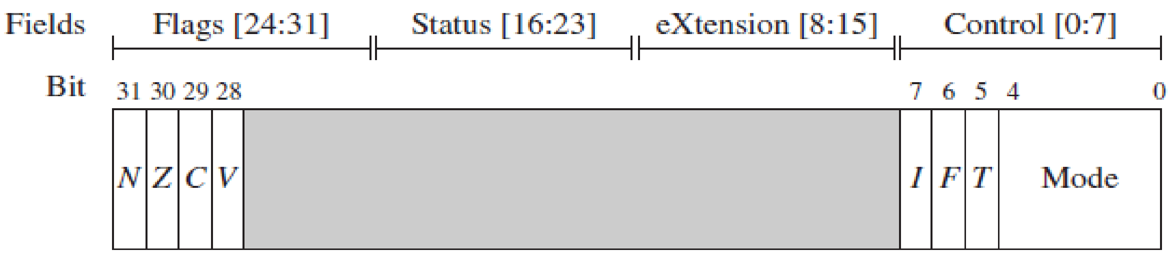

Figure: *psr* Byte Fields

The *c* field controls the interrupt masks, Thumb state, and processor mode.

The following Example shows how to enable IRQ interrupts by clearing the *I* mask. This operation involves using both the MRS and MSR instructions to read from and then write to the *cpsr*.

Example: 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. You can see from this example that this code preserves all the other settings in the *cpsr* and only modifies the *I* bit in the control field.

```
PRE cpsr = nzcvqIFt_SVC
MRS    r1,  cpsr
BIC r1, r1, #0x80         ; 0b01000000
MSR cpsr_c, r1
POST cpsr = nzcvqiFt_SVC
```

This example is in SVC mode. In user mode you can read all *cpsr* bits, but you can only update the condition flag field *f*.

Coprocessor Instructions:

Coprocessor instructions are used to extend the instruction set.

- ✓ A coprocessor can either provide additional computation capability or be used to control the memory subsystem including caches and memory management.
- ✓ The coprocessor instructions include data processing, register transfer, and memory transfer instructions.
- ✓ Note that these instructions are only used by cores with a coprocessor.

```
Syntax: CDP{<cond>} cp, opcode1, Cd, Cn {, opcode2}
        <MRC|MCR>{<cond>} cp, opcode1, Rd, Cn, Cm {, opcode2}
        <LDC|STC>{<cond>} cp, Cd, addressing
```

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

- ✓ In the syntax of the coprocessor instructions,
    - 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 coprocessor operations and registers depend on the specific coprocessor you are using.
- ✓ *Coprocessor 15 (CP15)* is reserved for system control purposes, such as memory management, write buffer control, cache control, and identification registers.

Example: This example shows a *CP15* register being copied into a general-purpose register.

; transferring the contents of CP15 register c0 to register r10

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

Here *CP15 register-0* contains the processor identification number. This register is copied into the general-purpose register *r10.*

LOADING CONSTANTS:

You might have noticed that 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.

To aid programming there are two pseudo-instructions to move a 32-bit value into a register.

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

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

- The first pseudo-instruction writes a 32-bit constant to a register using whatever instructions are available. It defaults to a memory read if the constant cannot be encoded using other instructions.
- The second pseudo-instruction writes a relative address into a register, which will be encoded using a pc-relative expression.

Example: This example shows an LDR instruction loading a 32-bit constant *0xff00ffff* into register *r0*.

```
LDR r0, [pc, #constant_number-8-{PC}]
:
constant_number
        DCD     0xff00ffff
```

This example involves a memory access to load the constant, which can be expensive for time-critical routines.

The following Example shows an alternative method to load the same constant into register *r0* by using an MVN instruction.

Example: Loading the constant *0xff00ffff* using an MVN.

PRE *none...*

MVN r0, #0x00ff0000

POST *r0 = 0xff00ffff*

As you can see, there are alternatives to accessing memory, but they depend upon the constant you are trying to load.

The LDR pseudo-instruction either inserts an MOV or MVN instruction to generate a value (if possible) or generates an LDR instruction with a *pc*-relative address to read the constant from a literal pool—a data area embedded within the code.

The following Table shows two pseudo-code conversions.

Table: LDR pseudo-instruction Conversion

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

The first conversion produces a simple MOV instruction; the second conversion produces a *pc*-relative load.

Another useful pseudo-instruction is the ADR instruction, or address relative. This instruction places the address of the given label into register *Rd*, using a *pc*-relative add or subtract.

****

****