

MODULE – 2ARM INSTRUCTION SET & ARM PROGRAMMINGINTRODUCTION 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 (<i>cpsr</i> or <i>spsr</i>)
MSR	v3	move to a status register (<i>cpsr</i> or <i>spsr</i>) 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 $((16 \times 16) + 32 = 32\text{-bit})$
SMLAL	v3M	signed multiply accumulate long $((32 \times 32) + 64 = 64\text{-bit})$
SMLAL _{xy}	v5E	signed multiply accumulate long $((16 \times 16) + 64 = 64\text{-bit})$
SMLAW _y	v5E	signed multiply accumulate instruction $((32 \times 16) \gg 16) + 32 = 32\text{-bit})$
SMULL	v3M	signed multiply long $(32 \times 32 = 64\text{-bit})$

Mnemonics	ARM ISA	Description
SMUL _{xy}	v5E	signed multiply instructions $(16 \times 16 = 32\text{-bit})$
SMULW _y	v5E	signed multiply instruction $((32 \times 16) \gg 16 = 32\text{-bit})$
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 $((32 \times 32) + 64 = 64\text{-bit})$
UMULL	v3M	unsigned multiply long $(32 \times 32 = 64\text{-bit})$

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 (<i>Rd</i>)	Source register 1 (<i>Rn</i>)	Source register 2 (<i>Rm</i>)
ADD <i>r3</i> , <i>r1</i> , <i>r2</i>	<i>r3</i>	<i>r1</i>	<i>r2</i>

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	<i>Rd</i> = <i>N</i>
MVN	move the NOT of the 32-bit value into a register	<i>Rd</i> = $\sim N$

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.
 - This shift increases the power and flexibility of many data processing operations.
 - 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.

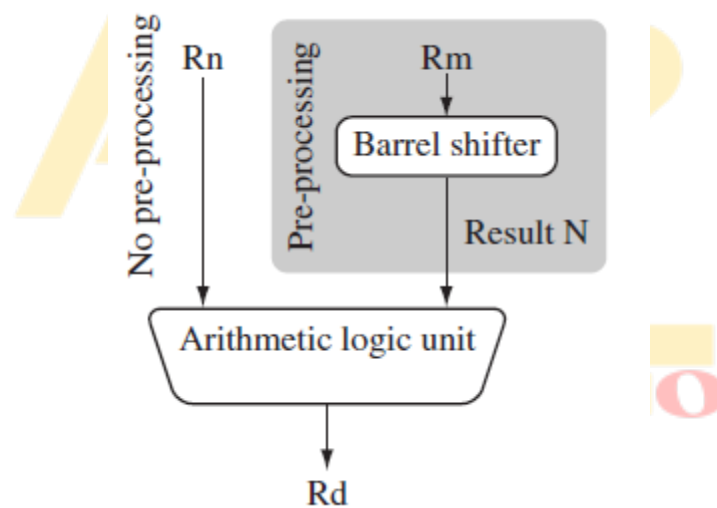


Figure: Barrel Shifter and ALU

- ✓ 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 \ll 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	$x \text{ LSL } y$	$x \ll y$	#0–31 or R_s
LSR	logical shift right	$x \text{ LSR } y$	(unsigned) $x \gg y$	#1–32 or R_s
ASR	arithmetic right shift	$x \text{ ASR } y$	(signed) $x \gg y$	#1–32 or R_s
ROR	rotate right	$x \text{ ROR } y$	$((\text{unsigned})x \gg y) (x \ll (32 - y))$	#1–31 or R_s
RRX	rotate right extended	$x \text{ RRX}$	$(c \text{ flag} \ll 31) ((\text{unsigned})x \gg 1)$	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 $r5 = 5$

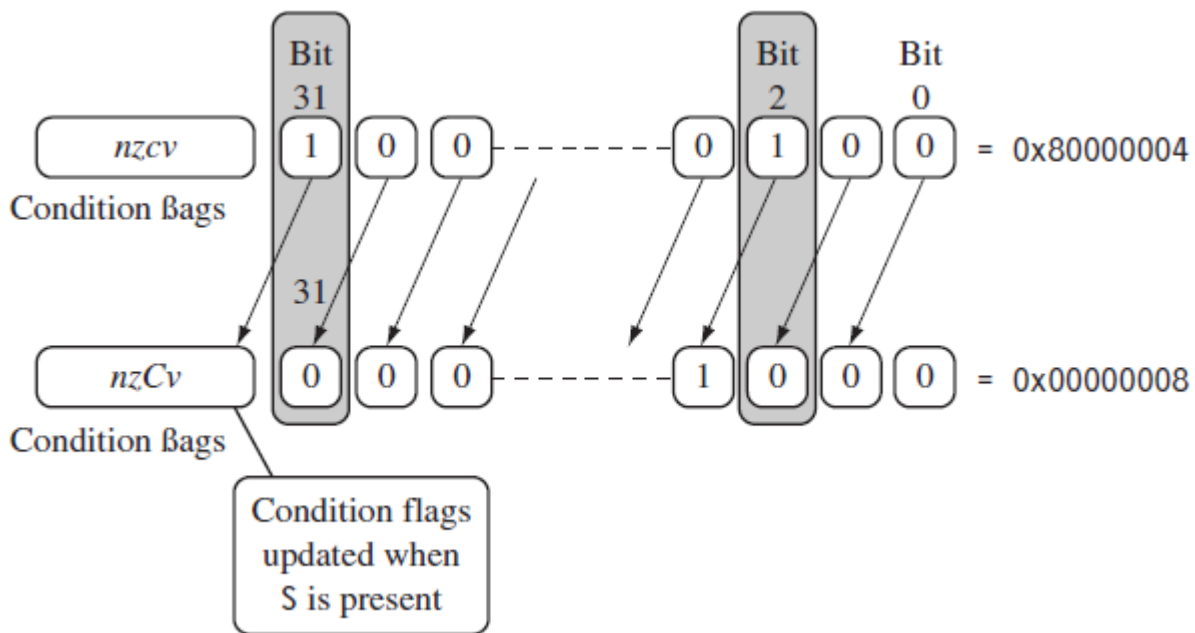
$r7 = 8$

MOV $r7, r5, \text{LSL} \#2$; let $r7 = r5 * 4 = (r5 \ll 2)$

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

<i>N</i> shift operations	Syntax
Immediate	#immediate
Register	<i>Rm</i>
Logical shift left by immediate	<i>Rm</i> , LSL #shift_imm
Logical shift left by register	<i>Rm</i> , LSL <i>Rs</i>
Logical shift right by immediate	<i>Rm</i> , LSR #shift_imm
Logical shift right with register	<i>Rm</i> , LSR <i>Rs</i>
Arithmetic shift right by immediate	<i>Rm</i> , ASR #shift_imm
Arithmetic shift right by register	<i>Rm</i> , ASR <i>Rs</i>
Rotate right by immediate	<i>Rm</i> , ROR #shift_imm
Rotate right by register	<i>Rm</i> , ROR <i>Rs</i>
Rotate right with extend	<i>Rm</i> , 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	$Rd = Rn + N + \text{carry}$
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	$Rd = N - Rn - !(\text{carry flag})$
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N - !(\text{carry flag})$
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* = 0xfffff89

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* = nzcqvqiFt_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	$Rd = Rn N$
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 – $Rd = Rn \text{ AND NOT } (N)$

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	$Rd = Rm * Rs$

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

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 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	$pc = label$ $lr = \text{address of the next instruction after the BL}$
BX	branch exchange	$pc = Rm \ \& \ 0xffffffffe, T = Rm \ \& \ 1$
BLX	branch exchange with link	$pc = label, T = 1$ $pc = Rm \ \& \ 0xffffffffe, T = Rm \ \& \ 1$ $lr = \text{address of the next instruction after the BLX}$

- ✓ 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	$Rd \leftarrow mem32[address]$
STR	save byte or word from a register	$Rd \rightarrow mem32[address]$
LDRB	load byte into a register	$Rd \leftarrow mem8[address]$
STRB	save byte from a register	$Rd \rightarrow mem8[address]$

LDRH	load halfword into a register	$Rd \leftarrow mem16[address]$
STRH	save halfword into a register	$Rd \rightarrow mem16[address]$
LDRSB	load signed byte into a register	$Rd \leftarrow SignExtend(mem8[address])$
LDRSH	load signed halfword into a register	$Rd \leftarrow SignExtend(mem16[address])$

- ✓ 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.
 - 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.
 - The postindex and preindex with writeback modes are useful for traversing an array.

Example:

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

LDR r0, [r1, #4]!

Preindexing with writeback:

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

LDR r0, [r1, #4]

Preindexing:

POST(2) $r0 = 0x02020202$
 $r1 = 0x00090000$

LDR r0, [r1], #4

Postindexing:

POST(3) $r0 = 0x01010101$
 $r1 = 0x00090004$

- ✓ 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	<i>r0</i> =	<i>r1</i> + =
Preindex with writeback	LDR <i>r0</i> , [<i>r1</i> , #0x4]!	mem32[<i>r1</i> + 0x4]	0x4
	LDR <i>r0</i> , [<i>r1</i> , <i>r2</i>]!	mem32[<i>r1</i> + <i>r2</i>]	<i>r2</i>
	LDR <i>r0</i> , [<i>r1</i> , <i>r2</i> , LSR #0x4]!	mem32[<i>r1</i> + (<i>r2</i> LSR 0x4)]	(<i>r2</i> LSR 0x4)
Preindex	LDR <i>r0</i> , [<i>r1</i> , #0x4]	mem32[<i>r1</i> + 0x4]	<i>not updated</i>
	LDR <i>r0</i> , [<i>r1</i> , <i>r2</i>]	mem32[<i>r1</i> + <i>r2</i>]	<i>not updated</i>
	LDR <i>r0</i> , [<i>r1</i> , - <i>r2</i> , LSR #0x4]	mem32[<i>r1</i> - (<i>r2</i> LSR 0x4)]	<i>not updated</i>
Postindex	LDR <i>r0</i> , [<i>r1</i>], #0x4	mem32[<i>r1</i>]	0x4
	LDR <i>r0</i> , [<i>r1</i>], <i>r2</i>	mem32[<i>r1</i>]	<i>r2</i>
	LDR <i>r0</i> , [<i>r1</i>], <i>r2</i> , LSR #0x4	mem32[<i>r1</i>]	(<i>r2</i> 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	<i>r1</i> +=
Preindex with writeback	STRH r0, [r1, #0x4]!	mem16[r1+0x4]=r0	0x4
Preindex	STRH r0, [r1, r2]!	mem16[r1+r2]=r0	r2
	STRH r0, [r1, #0x4]	mem16[r1+0x4]=r0	<i>not updated</i>
Postindex	STRH r0, [r1, r2]	mem16[r1+r2]=r0	<i>not updated</i>
	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.
 - 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.
 - 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	$r2 = 0x00000000$
$r0 = 0x80010 \rightarrow$	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	
$r0 = 0x8001c \rightarrow$	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.

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

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 $r2, \#2$

Dr. MAHESH PRASANNA K., VCET, PUTTUR

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.

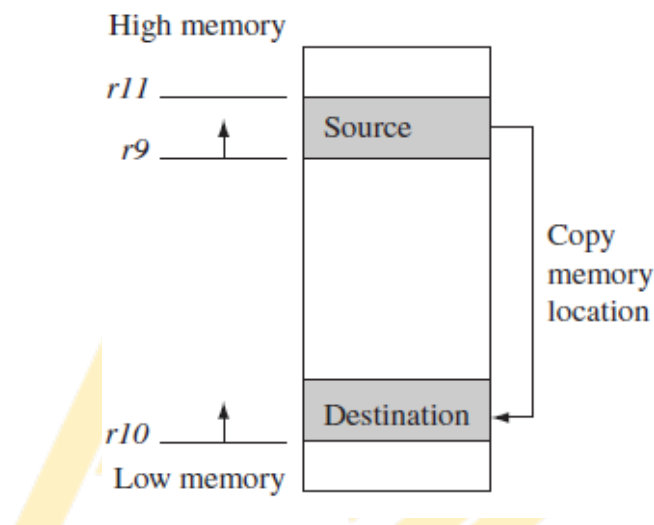


Figure: Block Memory Copy in the Memory map

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.

Stack 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.

PRE	Address	Data	POST	Address	Data
<i>sp</i> →	0x80018	0x00000001	<i>sp</i> →	0x80018	0x00000001
	0x80014	0x00000002		0x80014	0x00000002
	0x80010	Empty		0x80010	0x00000003
	0x8000c	Empty		0x8000c	0x00000002

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	POST	Address	Data
	0x80018	0x00000001		0x80018	0x00000001
	0x80014	0x00000002		0x80014	0x00000002
<i>sp</i> →	0x80010	Empty		0x80010	0x00000003
	0x8000c	Empty		0x8000c	0x00000002
	0x80008	Empty	<i>sp</i> →	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	$lr_svc = \text{address of instruction following the SWI}$ $spsr_svc = cpsr$ $pc = \text{vectors} + 0x8$ $cpsr \text{ mode} = SVC$ $cpsr I = 1$ (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

$$SWI_Number = \langle SWI\ instruction \rangle \text{ AND NOT } (0xff000000)$$

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.

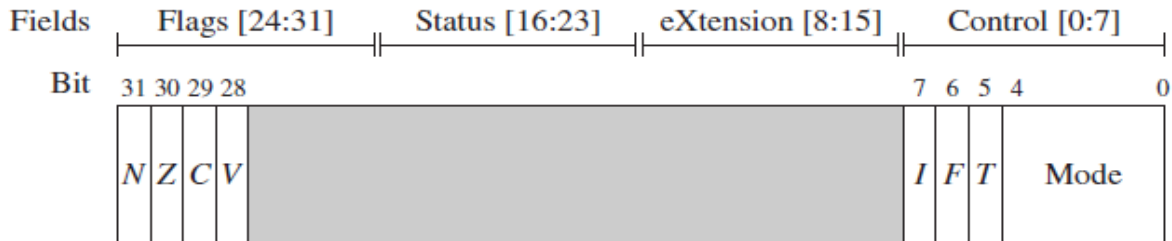


Figure: *psr* Byte Fields

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	<i>Rd</i> = 32-bit constant
ADR	load address pseudoinstruction	<i>Rd</i> = 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.

ARM PROGRAMMING USING ASSEMBLY LANGUAGEWRITING ASSEMBLY CODE:

This section gives examples showing how to write basic assembly code. Also, this section uses the *ARM macro assembler armasm* for examples.

Example 1:

<p>This example shows how to convert a C function to an assembly function—usually the first stage of assembly optimization. Consider the simple C program <i>main.c</i> following that prints the squares of the integers from 0 to 9:</p>	<p>Let's see how to replace square by an assembly function that performs the same action. Remove the C definition of square, but not the declaration (the second line) to produce a new C file <i>main1.c</i>. Next add an <i>armasm</i> assembler file <i>square.s</i> with the following contents:</p>
<pre>#include <stdio.h> int square(int i); int main(void) { int i; for (i=0; i<10; i++) { printf("Square of %d is %d\n", i, square(i)); } } int square(int i) { return i*i; }</pre>	<pre>AREA .text , CODE, READONLY EXPORT square ; int square(int i) square MUL r1, r0, r0 ; r1 = r0 * r0 MOV r0, r1 ; r0 = r1 MOV pc, lr ; return r0 END</pre>

- The *AREA* directive names the area or code section that the code lives in. If you use non-alphanumeric characters in a symbol or area name, then enclose the name in vertical bars. Many non-alphanumeric characters have special meanings otherwise. In the previous code we define a read-only code area called *.text*.
- The *EXPORT* directive makes the symbol square available for external linking. At line six we define the symbol square as a code label. Note that *armasm* treats non-indented text as a label definition.
- When square is called, the parameter passing is defined by the *ARM-Thumb procedure call standard (ATPCS)*. The input argument is passed in register *r0*, and the return value is returned in register *r0*. The multiply instruction has a restriction that the destination register must not be the same as the first argument register. Therefore we place the multiply result into *r1* and move this to *r0*.
- The *END* directive marks the end of the assembly file. Comments follow a semicolon.

The following script illustrates how to build this example using command line tools.

```
armcc -c main1.c
armasm square.s
armlink -o main1.axf main1.o square.o
```

Example 1 only works if you are compiling your *C* as ARM code. If you compile your *C* as Thumb code, then the assembly routine must return using a *BX* instruction.

Example 2: When calling ARM code from *C* compiled as Thumb, the only change required to the assembly in *Example 1* is to change the return instruction to a *BX*. *BX* will return to ARM or Thumb state according to bit 0 of *lr*. Therefore this routine can be called from ARM or Thumb. Use *BX lr* instead of *MOV pc, lr* whenever your processor supports *BX* (ARMv4T and above). Create a new assembly file *square2.s* as follows:

```
AREA    |.text|, CODE, READONLY
EXPORT  square

; int square(int i)
square
    MUL    r1, r0, r0    ; r1 = r0 * r0
    MOV    r0, r1        ; r0 = r1
    BX     lr            ; return r0
END
```

With this example we build the *C* file using the Thumb *C* compiler *tcc*. We assemble the assembly file with the interworking flag enabled so that the linker will allow the Thumb *C* code to call the ARM assembly code. You can use the following commands to build this example:

```
tcc -c main1.c
armasm -apcs /interwork square2.s
armlink -o main2.axf main1.o square2.o
```

Example 3: This example shows how to call a subroutine from an assembly routine. We will take Example 1 and convert the whole program (including main) into assembly. We will call the *C* library routine *printf* as a subroutine. Create a new assembly file *main3.s* with the following contents:

```

        AREA    |.text|, CODE, READONLY

        EXPORT  main

        IMPORT  |Lib$$Request$$armlib|, WEAK
        IMPORT  __main    ; C library entry
        IMPORT  printf    ; prints to stdout

i        RN 4

        ; int main(void)

main
        STMFD   sp!, {i, lr}
        MOV     i, #0

loop
        ADR     r0, print_string
        MOV     r1, i
        MUL     r2, i, i
        BL      printf
        ADD     i, i, #1
        CMP     i, #10
        BLT     loop
        LDMFD   sp!, {i, pc}

print_string
        DCB     "Square of %d is %d\n", 0

        END

```

- The *IMPORT* directive is used to declare symbols that are defined in other files.
- The imported symbol *Lib\$\$Request\$\$armlib* makes a request that the linker links with the standard ARM C library.
 - The *WEAK* specifier prevents the linker from giving an error if the symbol is not found at link time. If the symbol is not found, it will take the value zero.
- The second imported symbol *__main* is the start of the C library initialization code.

You only need to import these symbols if you are defining your own main; a main defined in C code will import these automatically for you. Importing *printf* allows us to call that C library function.

- The *RN* directive allows us to use names for registers. In this case we define *i* as an alternate name for register *r4*.

- Using register names makes the code more readable. It is also easier to change the allocation of variables to registers at a later date. Recall that *ATPCS* states that a function must preserve registers *r4* to *r11* and *sp*. We corrupt *i* (*r4*), and calling *printf* will corrupt *lr*. Therefore we stack these two registers at the start of the function using an *STMF* instruction. The *LDMFD* instruction pulls these registers from the stack and returns by writing the return address to *pc*.
- The *DCB* directive defines byte data described as a string or a comma-separated list of bytes.

To build this example you can use the following command line script:

```
armasm main3.s
armlink -o main3.axf main3.o
```

Note that Example 3 also assumes that the code is called from ARM code. If the code can be called from Thumb code as in Example 2 then we must be capable of returning to Thumb code. For architectures before *ARMv5* we must use a *BX* to return. Change the last instruction to the two instructions:

```
LDMFD    sp!, {i, lr}
BX       lr
```

Example 4: This example defines a function *sumof* that can sum any number of integers. The arguments are the number of integers to sum followed by a list of the integers. The *sumof* function is written in assembly and can accept any number of arguments. Put the C part of the example in a file *main4.c*:

```
#include <stdio.h>

/* N is the number of values to sum in list ... */
int sumof(int N, ...);

int main(void)
{
    printf("Empty sum=%d\n", sumof(0));
    printf("1=%d\n", sumof(1,1));
    printf("1+2=%d\n", sumof(2,1,2));
    printf("1+2+3=%d\n", sumof(3,1,2,3));
    printf("1+2+3+4=%d\n", sumof(4,1,2,3,4));
    printf("1+2+3+4+5=%d\n", sumof(5,1,2,3,4,5));
    printf("1+2+3+4+5+6=%d\n", sumof(6,1,2,3,4,5,6));
}
```

Next define the *sumof* function in an assembly file *sumof.s*:

```

AREA    |.text|, CODE, READONLY

EXPORT  sumof

N       RN 0    ; number of elements to sum
sum     RN 1    ; current sum

; int sumof(int N, ...)
sumof
    SUBS    N, N, #1        ; do we have one element
    MOVLT   sum, #0         ; no elements to sum!
    SUBS    N, N, #1        ; do we have two elements
    ADDGE   sum, sum, r2
    SUBS    N, N, #1        ; do we have three elements
    ADDGE   sum, sum, r3
    MOV     r2, sp          ; top of stack
loop
    SUBS    N, N, #1        ; do we have another element
    LDMGEFD r2!, {r3}       ; load from the stack

    ADDGE   sum, sum, r3
    BGE     loop
    MOV     r0, sum
    MOV     pc, lr          ; return r0

END

```

The code keeps count of the number of remaining values to *sum*, *N*. The first three values are in registers *r1*, *r2*, *r3*. The remaining values are on the stack (Recall that *ATPCS* places the first four arguments in registers *r0* to *r3*. Subsequent arguments are placed on the stack). You can build this example using the commands –

```

armcc -c main4.c
armasm sumof.s
armlink -o main4.axf main4.o sumof.o

```

PROFILING AND CYCLE COUNTING:

- ✓ The first stage of any optimization process is to identify the critical routines and measure their current performance. A **profiler** is a tool that measures the proportion of time or processing cycles spent in each subroutine. You use a profiler to identify the most critical routines.

- ✓ A **cycle counter** measures the number of cycles taken by a specific routine. You can measure your success by using a cycle counter to benchmark a given subroutine before and after an optimization.
- ✓ The ARM simulator used by the *ADS1.1 debugger* is called the *ARMulator* and provides profiling and cycle counting features.
 - The *ARMulator profiler* works by sampling the program counter *pc* at regular intervals. The profiler identifies the function the *pc* points to and updates a hit counter for each function it encounters. Another approach is to use the trace output of a simulator as a source for analysis.
 - The accuracy of a *pc*-sampled profiler is limited, as it can produce meaningless results if it records too few samples.
- ✓ ARM implementations do not normally contain cycle-counting hardware; so to easily measure cycle counts you should use an ARM debugger with ARM simulator.
 - You can configure the *ARMulator* to simulate a range of different ARM cores and obtain cycle count benchmarks for a number of platforms.

INSTRUCTION SCHEDULING:

The time taken to execute instructions depends on the implementation pipeline. For this section, we assume *ARM9TDMI* pipeline timings. The following rules summarize the cycle timings for common instruction classes on the *ARM9TDMI*.

Instructions that are conditional on the value of the ARM condition codes in the *cpsr* take one cycle if the condition is not met. If the condition is met, then the following rules apply:

- ✓ *ALU* operations such as addition, subtraction, and logical operations take one cycle.
- ✓ This includes a shift by an immediate value. If you use a register-specified shift, then add one cycle. If the instruction writes to the *pc*, then add two cycles.
- Load instructions that load *N* 32-bit words of memory such as *LDR* and *LDM* take *N* cycles to issue, but the result of the last word loaded is not available on the following cycle.
 - The updated load address is available on the next cycle. This assumes zero-wait-state memory for an un-cached system, or a cache hit for a cached system. An *LDM* of a single value is exceptional, taking two cycles. If the instruction loads *pc*, then add two cycles.
 - Load instructions that load 16-bit or 8-bit data such as *LDRB*, *LDRSB*, *LDRH*, and *LDRSH* take one cycle to issue. The load result is not available on the following two cycles. The updated load address is available on the next cycle. This assumes zero-wait-state memory for an un-cached system, or a cache hit for a cached system.
- Branch instructions take three cycles.

- Store instructions that store N values take N cycles. This assumes zero-wait-state memory for an un-cached system, or a cache hit or a write buffer with N free entries for a cached system. An *STM* of a single value is exceptional, taking two cycles.
- Multiply instructions take a varying number of cycles depending on the value of the second operand in the product.

To understand how to schedule code efficiently on the ARM, we need to understand the ARM pipeline and dependencies. The *ARM9TDMI* processor performs five operations in parallel:

- **Fetch:** Fetch from memory the instruction at address pc . The instruction is loaded into the core and then processes down the core pipeline.
- **Decode:** Decode the instruction that was fetched in the previous cycle. The processor also reads the input operands from the register bank if they are not available via one of the forwarding paths.
- **ALU:** Executes the instruction that was decoded in the previous cycle. Note this instruction was originally fetched from address $pc - 8$ (ARM state) or $pc - 4$ (Thumb state).
 - Normally this involves calculating the answer for a data processing operation, or the address for a load, store, or branch operation.
 - Some instructions may spend several cycles in this stage. For example, multiply and register-controlled shift operations take several ALU cycles.
- **LS1:** Load or store the data specified by a load or store instruction. If the instruction is not a load or store, then this stage has no effect.
- **LS2:** Extract and zero- or sign-extend the data loaded by a byte or half-word load instruction. If the instruction is not a load of an 8-bit byte or 16-bit half-word item, then this stage has no effect.

The following Figure shows a simplified functional view of the five-stage *ARM9TDMI* pipeline.

Instruction address	pc	$pc-4$	$pc-8$	$pc-12$	$pc-16$
Action	Fetch	Decode	ALU	LS1	LS2

Note that multiply and register shift operations are not shown in the figure.

After an instruction has completed the five stages of the pipeline, the core writes the result to the register file. Note that pc points to the address of the instruction being fetched. The ALU is executing the instruction that was originally fetched from address $pc - 8$ in parallel with fetching the instruction at address pc .

How does the pipeline affect the timing of instructions? Consider the following examples. These examples show how the cycle timings change because an earlier instruction must complete a stage before the current instruction can progress down the pipeline.

If an instruction requires the result of a previous instruction that is not available, then the processor stalls. This is called a *pipeline hazard* or *pipeline interlock*.

Example 5: This example shows the case where there is no interlock.

```
ADD  r0, r0, r1
ADD  r0, r0, r2
```

This instruction pair takes two cycles. The ALU calculates $r0 + r1$ in one cycle. Therefore this result is available for the ALU to calculate $r0 + r2$ in the second cycle.

Example 6: This example shows a one-cycle interlock caused by load use.

```
LDR  r1, [r2, #4]
ADD  r0, r0, r1
```

This instruction pair takes three cycles. The ALU calculates the address $r2 + 4$ in the first cycle while decoding the *ADD* instruction in parallel. However, the *ADD* cannot proceed on the second cycle because the load instruction has not yet loaded the value of $r1$. Therefore the pipeline stalls for one cycle while the load instruction completes the *LS1* stage. Now that $r1$ is ready, the processor executes the *ADD* in the ALU on the third cycle.

The following Figure illustrates how this interlock affects the pipeline.

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	...	ADD	LDR	...	
Cycle 2		...	<i>ADD</i>	LDR	...
Cycle 3		...	ADD	—	LDR

The processor stalls the *ADD* instruction for one cycle in the ALU stage of the pipeline while the load instruction completes the *LS1* stage. Figure denotes this stall by *italic ADD*. Since the *LDR* instruction proceeds down the pipeline, but the *ADD* instruction is stalled, a gap opens up between them. This gap is sometimes called a pipeline *bubble*. We've marked the bubble with a *dash*.

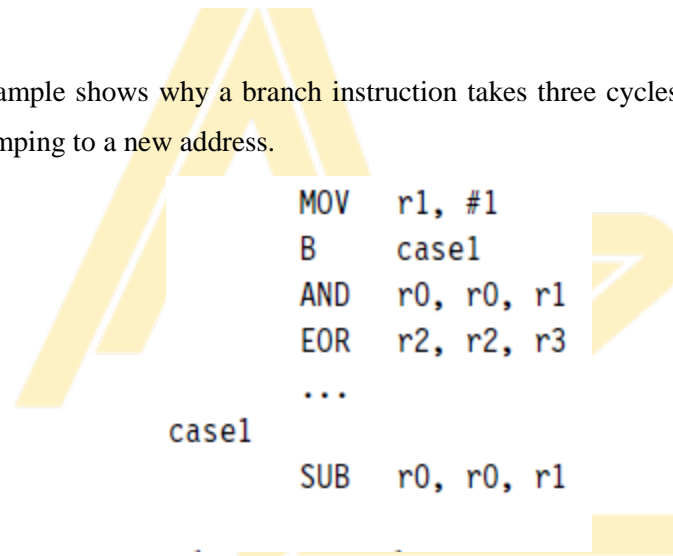
Example 7: This example shows a one-cycle interlock caused by delayed load use.

```
LDRB  r1, [r2, #1]
ADD    r0, r0, r2
EOR    r0, r0, r1
```

This instruction triplet takes four cycles. Although the *ADD* proceeds on the cycle following the load byte, the *EOR* instruction cannot start on the third cycle. The *r1* value is not ready until the load instruction completes the *LS2* stage of the pipeline. The processor stalls the *EOR* instruction for one cycle. Note that the *ADD* instruction does not affect the timing at all. The sequence takes four cycles whether it is there or not! The following Figure shows how this sequence progresses through the processor pipeline. The *ADD* doesn't cause any stalls since the *ADD* does not use *r1*, the result of the load.

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	EOR	ADD	LDRB
Cycle 2	...	EOR	ADD	LDRB	...
Cycle 3		...	EOR	ADD	LDRB
Cycle 4		...	EOR	—	ADD

Example 8: This example shows why a branch instruction takes three cycles. The processor must flush the pipeline when jumping to a new address.



The three executed instructions take a total of five cycles. The *MOV* instruction executes on the first cycle. On the second cycle, the branch instruction calculates the destination address. This causes the core to flush the pipeline and refill it using this new *pc* value. The refill takes two cycles. Finally, the *SUB* instruction executes normally. The following Figure illustrates the pipeline state on each cycle. The pipeline drops the two instructions following the branch when the branch takes place.

Pipeline	Fetch	Decode	ALU	LS1	LS2
Cycle 1	AND	B	MOV
Cycle 2	EOR	AND	B	MOV	...
Cycle 3	SUB	—	—	B	MOV
Cycle 4	...	SUB	—	—	B
Cycle 5		...	SUB	—	—

Scheduling of Load Instructions:

Load instructions occur frequently in compiled code, accounting for approximately one-third of all instructions. Careful scheduling of load instructions so that pipeline stalls don't occur can improve performance. The compiler attempts to schedule the code as best it can, but the aliasing problem of C limits the available optimizations. The compiler cannot move a load instruction before a store instruction unless it is certain that the two pointers used do not point to the same address.

Consider an example of a memory-intensive task. The following function, *str_tolower*, copies a zero-terminated string of characters from *in* to *out*. It converts the string to lowercase in the process.

<pre> void str_tolower(char *out, char *in) { unsigned int c; do { c = *(in++); if (c>='A' && c<='Z') { c = c + ('a' - 'A'); } *(out++) = (char)c; } while (c); } </pre>	<pre> str_tolower LDRB r2,[r1],#1 ; c = *(in++) SUB r3,r2,#0x41 ; r3 = c - 'A' CMP r3,#0x19 ; if (c <= 'Z' - 'A') ADDLS r2,r2,#0x20 ; c += 'a' - 'A' STRB r2,[r0],#1 ; *(out++) = (char)c CMP r2,#0 ; if (c!=0) BNE str_tolower ; goto str_tolower MOV pc,r14 ; return </pre>
---	--

The compiler generates the above compiled output. Notice that the compiler optimizes the condition ($c \geq 'A' \ \&\& \ c \leq 'Z'$) to the check that $0 \leq c - 'A' \leq 'Z' - 'A'$. The compiler can perform this check using a single unsigned comparison.

Unfortunately, the *SUB* instruction uses the value of *c* directly after the *LDRB* instruction that loads *c*. Consequently, the *ARM9TDMI* pipeline will stall for two cycles. The compiler can't do any better since everything following the load of *c* depends on its value.

However, there are two ways you can alter the structure of the algorithm to avoid the cycles by using assembly. We call these methods load scheduling by *preloading* and *unrolling*.

» *Load Scheduling by Preloading & Load Scheduling by Unrolling – Self Study.*

REGISTER ALLOCATION:

You can use 14 of the 16 visible ARM registers to hold general-purpose data. The other two registers are the *stack pointer*, *r13*, and the *program counter*, *r15*. For a function to be *ATPCS* compliant it must preserve the callee values of registers *r4* to *r11*. *ATPCS* also specifies that the stack should be eight-byte aligned; therefore you must preserve this alignment if calling subroutines. Use the following template for optimized assembly routines requiring many registers:


```

routine_name
    STMFD sp!,      {r4-r12, lr}      ; stack saved registers
    ; body of routine
    ; the fourteen registers r0-r12 and lr are available
    LDMFD sp!,      {r4-r12, pc}      ; restore registers and return

```

The only purpose in stacking *r12* is to keep the stack eight-byte aligned. You need not stack *r12* if your routine doesn't call other *ATPCS* routines. For *ARMv5* and above you can use the preceding template even when being called from Thumb code. If your routine may be called from Thumb code on an *ARMv4T* processor, then modify the template as follows:

```

routine_name
    STMFD sp!,      {r4-r12, lr}      ; stack saved registers
    ; body of routine
    ; registers r0-r12 and lr available
    LDMFD sp!,      {r4-r12, lr}      ; restore registers
    BX              lr                ; return, with mode switch

```

Allocating Variables to Register Numbers:

When you write an assembly routine, it is best to start by using names for the variables, rather than explicit register numbers. This allows you to change the allocation of variables to register numbers easily. You can even use different register names for the same physical register number when their use doesn't overlap. Register names increase the clarity and readability of optimized code.

For the most part ARM operations are orthogonal with respect to register number. In other words, specific register numbers do not have specific roles. If you swap all occurrences of two registers *Ra* and *Rb* in a routine, the function of the routine does not change.

However, there are several cases where the physical number of the register is important:

- ✓ *Argument registers:* The *ATPCS* convention defines that the first four arguments to a function are placed in registers *r0* to *r3*. Further arguments are placed on the stack.
 - The return value must be placed in *r0*.
- ✓ *Registers used in a load or store multiple:* Load and store multiple instructions *LDM* and *STM* operate on a list of registers in order of ascending register number. If *r0* and *r1* appear in the register list, then the processor will always load or store *r0* using a lower address than *r1* and so on.
- ✓ *Load and store double word:* The *LDRD* and *STRD* instructions introduced in *ARMv5E* operate on a pair of registers with sequential register numbers, *Rd* and *Rd + 1*. Furthermore, *Rd* must be an even register number.

Using More Than 14 Local Variables:

If you need more than 14 local 32-bit variables in a routine, then you must store some variables on the stack. The standard procedure is to work outwards from the innermost loop of the algorithm, since the innermost loop has the greatest performance impact.

Making the Most of Available Registers:

On load-store architecture such as the ARM, it is more efficient to access values held in registers than values held in memory. There are several tricks you can use to fit several sub-32-bit length variables into a single 32-bit register and thus can reduce code size and increase performance.

CONDITIONAL EXECUTION:

The processor core can conditionally execute most ARM instructions. This conditional execution is based on one of 15 condition codes. If you don't specify a condition, the assembler defaults to execute always condition (AL). The other 14 conditions split into seven pairs of complements. The conditions depend on the four condition code flags *N*, *Z*, *C*, *V* stored in the *cpsr* register.

By default, ARM instructions do not update the *N*, *Z*, *C*, *V* flags in the ARM *cpsr*. For most instructions, to update these flags you append an *S* suffix to the instruction mnemonic.

Exceptions to this are comparison instructions that do not write to a destination register. Their sole purpose is to update the flags and so they don't require the *S* suffix.

By combining conditional execution and conditional setting of the flags, you can implement simple if statements without any need for branches. This improves efficiency since branches can take many cycles and also reduces code size.

Example 17: The following C code converts an unsigned integer $0 \leq i \leq 15$ to a hexadecimal character *c*:

<pre> if (i<10) { c = i + '0'; } else { c = i + 'A'-10; } </pre>	<p>We can write this in assembly using conditional execution rather than conditional branches:</p>	<pre> CMP i, #10 ADDLO c, i, #'0' ADDHS c, i, #'A'-10 </pre>
---	--	---

The sequence works since the first *ADD* does not change the condition codes. The second *ADD* is still conditional on the result of the compare.

Conditional execution is even more powerful for cascading conditions.

Example 18: The following C code identifies if *c* is a vowel:

```
if (c=='a' || c=='e' || c=='i' || c=='o' || c=='u')
{
    vowel++;
}
```

In assembly you can write this using conditional comparison:

```
TEQ    c, #'a'
TEQNE  c, #'e'
TEQNE  c, #'i'
TEQNE  c, #'o'
TEQNE  c, #'u'
ADDEQ  vowel, vowel, #1
```

As soon as one of the *TEQ* comparisons detects a match, the Z flag is set in the *cpsr*. The following *TEQNE* instructions have no effect as they are conditional on $Z = 0$. The next instruction to have effect is the *ADDEQ* that increments *vowel*. You can use this method whenever all the comparisons in the if statement are of the same type.

Example 19: Consider the following code that detects if *c* is a letter:

```
if ((c>='A' && c<='Z') || (c>='a' && c<='z'))
{
    letter++;
}
```

To implement this efficiently, we can use an addition or subtraction to move each range to the form $0 \leq c \leq \text{limit}$. Then we use unsigned comparisons to detect this range and conditional comparisons to chain together ranges. The following assembly implements this efficiently:

```
SUB    temp, c, #'A'
CMP    temp, #'Z'-'A'
SUBHI  temp, c, #'a'
CMPHI  temp, #'z'-'a'
ADDLS  letter, letter, #1
```

Note that the logical operations *AND* and *OR* are related by the standard logical relations as shown in the following Table. You can invert logical expressions involving *OR* to get an expression involving *AND*, which can often be useful in simplifying or rearranging logical expressions.

Inverted expression	Equivalent
<code>!(a && b)</code>	<code>(!a) (!b)</code>
<code>!(a b)</code>	<code>(!a) && (!b)</code>

LOOPING CONSTRUCTS:

Most routines critical to performance will contain a loop. Note that, ARM loops are fastest when they count down towards zero. This section describes how to implement these loops efficiently in assembly. We also look at examples of how to unroll loops for maximum performance.

Decrementing Counted Loops:

For a decrementing loop of N iterations, the loop counter i counts down from N to 1 inclusive. The loop terminates with $i = 0$. An efficient implementation is

```

MOV i, N
loop
; loop body goes here and i=N,N-1,...,1
SUBS i, i, #1
BGT loop

```

The loop overhead consists of a subtraction setting the condition codes followed by a conditional branch. On ARM7 and ARM9 this overhead costs four cycles per loop. If i is an array index, then you may want to count down from $N-1$ to 0 inclusive instead so that you can access array element zero. You can implement this in the same way by using a different conditional branch:

```

SUBS i, N, #1
loop
; loop body goes here and i=N-1,N-2,...,0
SUBS i, i, #1
BGE loop

```

In this arrangement the Z flag is set on the last iteration of the loop and cleared for other iterations. If there is anything different about the last loop, then we can achieve this using the *EQ* and *NE* conditions. For example, if you preload data for the next loop, then you want to avoid the preload on the last loop. You can make all preload operations conditional on *NE*.

There is no reason why we must decrement by one on each loop. Suppose we require $N/3$ loops; rather than attempting to divide N by three, it is far more efficient to subtract three from the loop counter on each iteration:

```

MOV i, N
loop
; loop body goes here and iterates (round up)(N/3) times
SUBS i, i, #3
BGT loop

```

Unrolled Counted Loops:

Loop unrolling reduces the loop overhead by executing the loop body multiple times. However, there are problems to overcome.

Multiple Nested Loops:

How many loop counters does it take to maintain multiple nested loops? Actually, one will suffice—or more accurately, one provided the sum of the bits needed for each loop count does not exceed 32. We can combine the loop counts within a single register, placing the innermost loop count at the highest bit positions.

Other Counted Loops:

You may want to use the value of a loop counter as an input to calculations in the loop. It's not always desirable to count down from N to 1 or $N - 1$ to 0 . For example, you may want to select bits out of a data register one at a time; in this case you may want a power-of-two mask that doubles on each iteration.

The following subsections show useful looping structures that count in different patterns. They use only a single instruction combined with a branch to implement the loop.

Negative Indexing: This loop structure counts from $-N$ to 0 (inclusive or exclusive) in steps of size *STEP*.

```

RSB    i, N, #0      ; i=-N
loop
; loop body goes here and i=-N,-N+STEP,...,
ADDS   i, i, #STEP
BLT    loop          ; use BLT or BLE to exclude 0 or not

```

Logarithmic Indexing: This loop structure counts down from 2^N to 1 in powers of two. For example, if $N = 4$, then it counts 16, 8, 4, 2, 1.

```

MOV    i, #1
MOV    i, i, LSL N
loop
    ; loop body
    MOVS    i, i, LSR#1
    BNE     loop

```

The following loop structure counts down from an N -bit mask to a one-bit mask. For example, if $N = 4$, then it counts 15, 7, 3, 1.

```

MOV    i, #1
RSB    i, i, i, LSL N ; i=(1<<N)-1
loop
    ; loop body
    MOVS    i, i, LSR#1
    BNE     loop

```

By: DR. MAHESH PRASANNA K.,
DEPT. OF CSE, VCET.
