UNIT1 BASIC PROCESSOR ARCHITECTURE AND DESIGN

1.1 INTRODUCTION

- One of the most popular RISC instruction sets in use today
- Used by licensees of ARM Limited, UK
- ARM processors
- Some processors by Samsung, Qualcomm, and Apple
- Highly versatile instruction set
- Floating-point and vector (multiple operations per instruction) extensions

Characteristic of RISC

- Simpler instruction, hence simple instruction decoding.
- Instruction comes undersize of one word.
- Instruction takes a single clock cycle to get executed.
- More number of general-purpose registers.
- Simple Addressing Modes.
- Less Data types.
- Pipeline can be achieved.

The RISC philosophy is implemented with four major design rules:
1. Instructions. 2. Pipelines. 3. Register. 4. Load - Store architecture.

1. Instructions: – RISC processors have a reduced number of instruction classes. These classes provide simple operations that can each execute in a single cycle. The compiler or programmer synthesizes complicated operations (a divide operation) by combining several simple instructions. Each instruction is a fixed length to allow the pipeline to fetch future instructions before decoding the current instruction. In contrast, in CISC processors the instructions are often of variable size and take many cycles to execute.

2. Pipelines: —the processing of instructions is broken down into smaller units that can be executed in parallel by pipelines. Ideally the pipeline advances by one step on each cycle for maximum throughput. There is no need for an instruction to be executed by a mini program called microcode as on CISC processors.

3. Registers:—RISC machines have a large general-purpose register set. Any register can contain either data or an address. In contrast, CISC processors have dedicated registers for specific purposes.
4. Load-store architecture:—the processor operates on data held in registers. Separate load and store instructions transfer data between the register bank and external memory In contrast, with a CISC design the data processing operations can act on memory directly.

Characteristic of CISC

- Complex instruction, hence complex instruction decoding.
- Instructions are larger than one-word size.
- Instruction may take more than a single clock cycle to get executed.
- Less number of general-purpose registers as operation get performed in memory itself.
- Complex Addressing Modes.
- More Data types.

1.2 INTRODUCTION TO MICROPROCESSOR

Advanced RISC Machine (ARM) Processor is considered to be family of Central Processing Units that is used in music players, smartphones, wearables, tablets and other consumer electronic devices.
The architecture of ARM processor is created by Advanced RISC Machines, hence name ARM.
This needs very few instruction sets and transistors. It has very small size. This is reason that it is perfect fit for small size devices. It has less power consumption along with reduced complexity in its circuits. They can be applied to various designs such as 32-bit devices and embedded systems. They can even be upgraded according to user needs.
The main features of ARM Processor are mentioned below:

1. Multiprocessing Systems –
   ARM processors are designed so that they can be used in cases of multiprocessing systems where more than one processors are used to process information.

2. Thumb-2 Technology –
   Thumb-2 Technology was introduced in 2003 and was used to create variable length instruction set. It extends 16-bit instructions of initial Thumb technology to 32-bit instructions. It has better performance than previously used Thumb technology.
3. One cycle execution time –
   ARM processor is optimised for each instruction on CPU. Each instruction is of fixed length that allows time for fetching future instructions before executing present instruction. ARM has CPI (Clock per Instruction) of one cycle.
4. Pipelining –
   Processing of instructions is done in parallel using pipelines. Instructions are broken down and decoded in one pipeline stage. The pipeline advances one step at a time to increase throughput (rate of processing).
5. Large number of registers –
   Large number of registers are used in ARM processor to prevent large amount of memory interactions. Registers contain data and addresses. These act as local memory store for all operations.

1.3 INTRODUCTION TO ARM MICROPROCESSOR

- ARM stands for Advance RISC Machines.
- ARM was developed at Acorn Computer Limited of Cambridge, UK (between 1983 & 1985) ⁄ RISC concept introduced in 1980 at Stanford and Berkeley.
- ARM founded in November 1990 ⁄ Advanced RISC Machines.
- ARM does not manufacture silicon ⁄ Licensed to partners to develop and fabricate new microcontrollers.
- The ARM processor is a key component of many successful 32-bit embedded systems. These processors widely used in mobile phones, handled organizers and multitude of other everyday portable consumer devices.
- In fact the ARM core is not a single core, but a whole family of designs sharing similar design principles and a common instruction set.
- The ARM7TDMI is ARM’s most successful core processor.

- ARM is based upon RISC Architecture with enhancements to meet requirements of embedded applications
- A large uniform register file
- Load-store architecture
- Fixed length instructions
- 32-bit processor
- Good speed/Good Power
- High code density

Enhancement to Basic RISC

- Control over ALU and shifter for every data processing operations
- Auto-increment and auto-decrement addressing modes
- To optimize program loops
- Load/Store multiple data instructions
- To maximize data throughput
- Conditional execution of instructions
- To maximize execution throughput

Data Sizes and Instruction Sets

- ARM is a 32-bit load / store RISC architecture
- The only memory accesses allowed are loads and stores
- Most internal registers are 32 bits wide
- Most instructions execute in a single cycle
- When used in relation to ARM cores
    - Halfword means 16 bits (two bytes)
    - Word means 32 bits (four bytes)
    - Double word means 64 bits (eight bytes)
- ARM cores implement two basic instruction sets
    - ARM instruction set – instructions are all 32 bits long
    - Thumb instruction set – instructions are a mix of 16 and 32 bits

1.4 REGISTER BANK

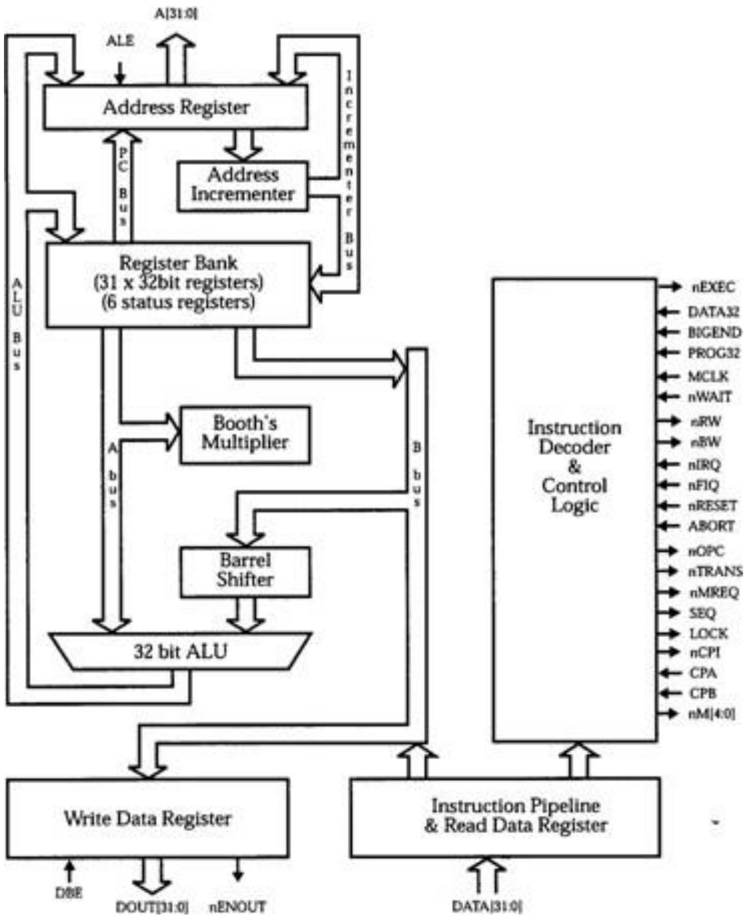


Figure 1 Basic ARM Organization

ARM processors provide general-purpose and special-purpose registers. Some additional registers are available in privileged execution modes. General Purpose registers hold either data or address. All registers are of 32 bits. In user mode 16 data registers and 2 status registers are visible

- Data registers: r0 to r15

r13, r14, and r15 perform special functions

- r13: stack pointer
- r14: link register
- r15: program counter

Depending upon context, registers r13 and r14 can also be used as GPR. Any instruction which use r0 can as well be used with any other GPR (r1-r13), two status registers

CPSR: Current Program Status Register, SPSR: Saved Program Status Register

The ARM Register Set

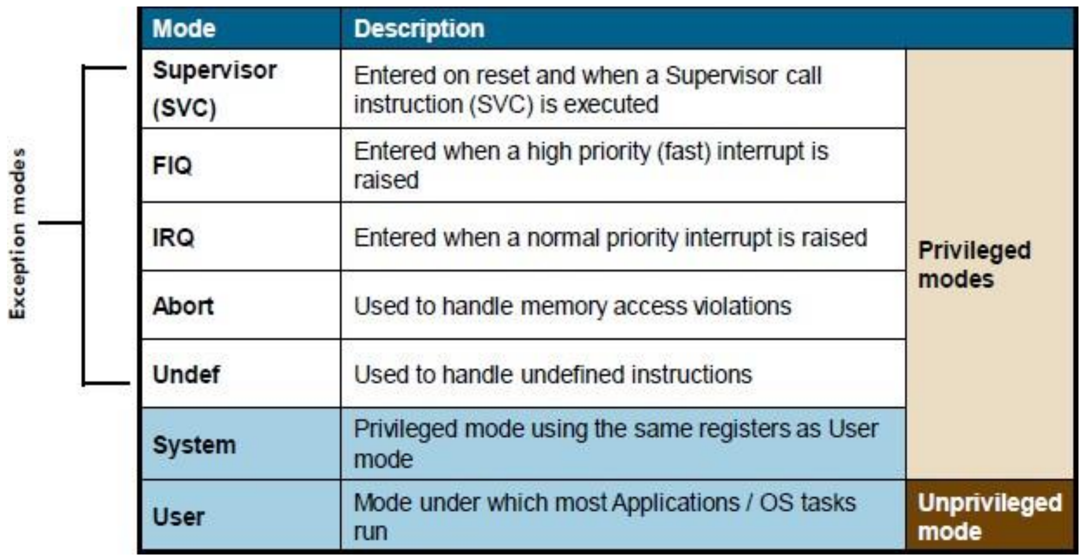
In all ARM processors, the following registers are available and accessible in any processor mode:

- 13 general-purpose register R0-R12.
- One Stack Pointer (SP)
- One Link Register (LR).
- One Program Counter (PC).
- One Current Program Status Register (CPSR).

ARM processors, with the exception of ARMv6-M and ARMv7-M based processors, have a total of 37 registers, with 3 additional registers if the Security Extensions are implemented, and in ARMv7-A only, 3 more if the Virtualization Extensions are implemented.

ARM Processor Modes

- Processor modes determine which registers are active
- Access right to CPSR registers itself
- Each processor mode is either
  - Privileged: full read-write access to the CPSR
  - Non-privileged: only read access to the control field of CPSR but read-write access to the condition flags
- ARM has seven basic operating modes
- Each mode has access to its own space and a different subset of registers
- Some operations can only be carried out in a privileged mode


Exception modes

Mode	Description	
Supervisor (SVC)	Entered on reset and when a Supervisor call instruction (SVC) is executed	Privileged modes
FIQ	Entered when a high priority (fast) interrupt is raised	
IRQ	Entered when a normal priority interrupt is raised	
Abort	Used to handle memory access violations	
Undef	Used to handle undefined instructions	
System	Privileged mode using the same registers as User mode	
User	Mode under which most Applications / OS tasks run	Unprivileged mode

Figure 2 ARM Processor Modes

Mode Changing

The registers are arranged in partially overlapping banks. There is a different register bank for each processor mode. The banked registers give rapid context switching for dealing with processor exceptions and privileged operations. The additional registers that are available in privileged software execution, with the exception of ARMv6-M and ARMv7-M, are:

- Two Supervisor mode registers for banked SP and LR.
- Two Abort mode registers for banked SP and LR.
- Two Undefined mode registers for banked SP and LR.
- Two Interrupt mode registers for banked SP and LR.
- Seven FIQ mode registers for banked R8-R12, SP and LR.

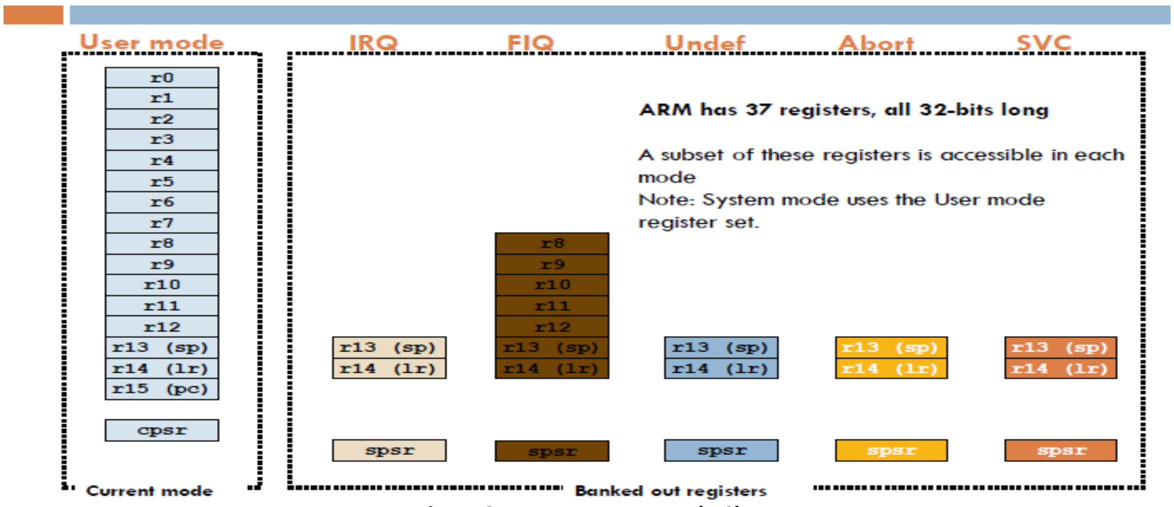

User mode
IRQ
FIQ
Undef
Abort
SVC
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr
spsr
ARM has 37 registers, all 32-bits long
A subset of these registers is accessible in each mode
Note: System mode uses the User mode register set.
Current mode
Banked out registers

Figure 3 ARM Processor Mode Change

- Mode changes by writing directly to CPSR or by hardware when the processor responds to exception or interrupt
- To return to user mode a special return instruction is used that instructs the core to restore the original CPSR and banked registers

Program Counter (r15)

When the processor is executing in ARM state:

- All instructions are 32 bits wide
- All instructions must be word aligned
- Therefore the PC value is stored in bits [31:2] with bits [1:0] undefined (as instruction cannot be half word or byte aligned)

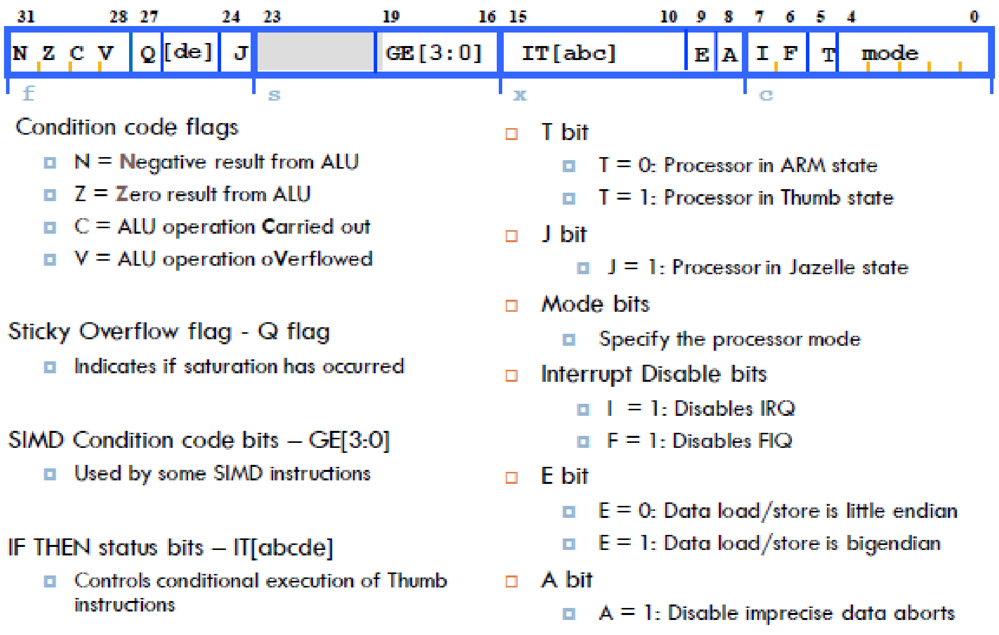
When the processor is executing in Thumb state:

- All instructions are 16 bits wide
- All instructions must be halfword aligned
- Therefore the pc value is stored in bits [31:1] with bit [0] undefined (as instruction cannot be byte aligned)

When the processor is executing in Jazelle state:

- All instructions are 8 bits wide
- Processor performs a word access to read 4 instructions at once

Program Status Registers



- Condition code flags
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed
- Sticky Overflow flag - Q flag
  - Indicates if saturation has occurred
- SIMD Condition code bits – GE[3:0]
  - Used by some SIMD instructions
- IF THEN status bits – IT[abcde]
  - Controls conditional execution of Thumb instructions
- T bit
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
- J bit
  - J = 1: Processor in Jazelle state
- Mode bits
  - Specify the processor mode
- Interrupt Disable bits
  - I = 1: Disables IRQ
  - F = 1: Disables FIQ
- E bit
  - E = 0: Data load/store is little endian
  - E = 1: Data load/store is bigendian
- A bit
  - A = 1: Disable imprecise data aborts

Figure 4 Current Program Status Register

Current Program Status Register: The CPSR holds

- The CPSR flags.
- The processor mode.
- The interrupt disable flags.
- The instruction set state (ARM, Thumb, ThumbEE, or Jazelle®).
- The endianness state (on ARMv4T and later).
- The execution state bits for the IT block (on ARMv6T2 and later).

ARM Memory Organization

Can be configured as

- Little Endian
- Big Endian
- Addresses are for each byte

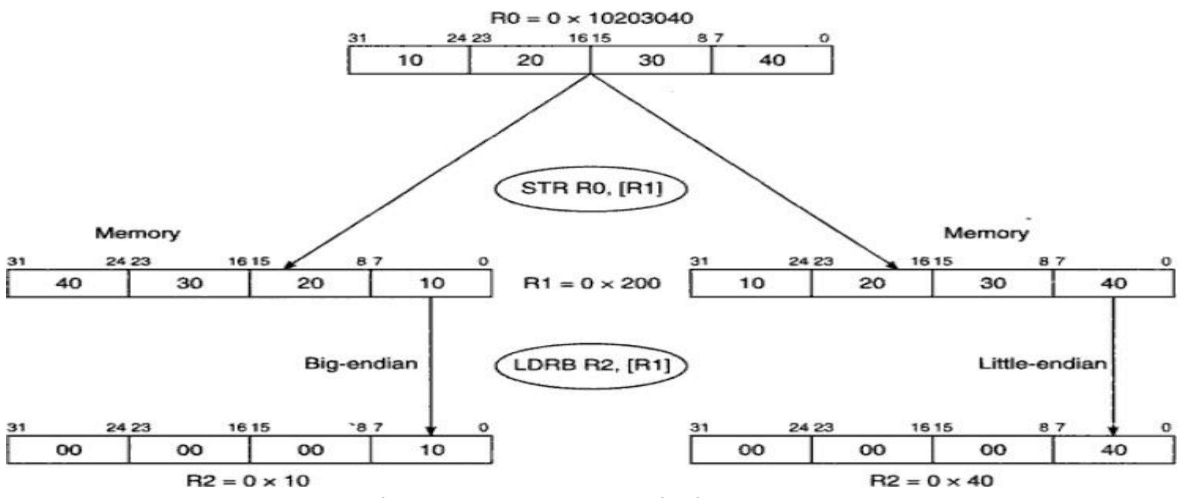


Figure 5 ARM Memory Organisation

1.5 ARM PROGRAM STRUCTURE AND INSTRUCTION FORMAT

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

Figure 6 ARM instruction Format

In instructions data will be represent hexadecimal numbers 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 Data Types

- Word is 32 bits long
- Word can be divided into four 8-bit bytes
- ARM addresses can be 32 bit long
- Address refers to byte
- Can be configured at power-up as either little- or big-endian mode

ARM Instruction Set

ARM instructions process data held in registers and only access memory 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).

Features of the ARM Assembly Language Instruction Set

- 3-address data processing instructions
- Conditional execution of each instruction
- Shift and ALU operations in single instruction
- Load-Store and Load-Store multiple instructions
- Single cycle execution of all instructions
- Instruction set extension through coprocessor instructions
- ARM instructions are all 32 -bit long (except for Thumb mode) Thumb mode).
- There are 232 possible machine instructions. Fortunately, they are structured.

Instructions process data held in registers and access memory with load and store instructions

Classes of instructions

- Data processing instructions
- Branch instructions
- Load-Store instructions
- Software interrupt instructions
- Program status register instructions
- Coprocessor instructions

1.6 DATA PROCESSING INSTRUCTION VARIANTS

Data Processing Instructions

- Perform move, arithmetic, logical, compare and multiply operations
- All operations except multiply instructions are carried out in ALU
- Multiply instructions are carried out in multiplier block
- Data processing instructions do not access memory
- Instructions operate on two 32-bit operands, produce 32-bit result.
- Instructions can pre-process one operand using barrel shifter
- No. of basic instructions: 16 (excluding two basic multiply instructions.)

  Syntax:

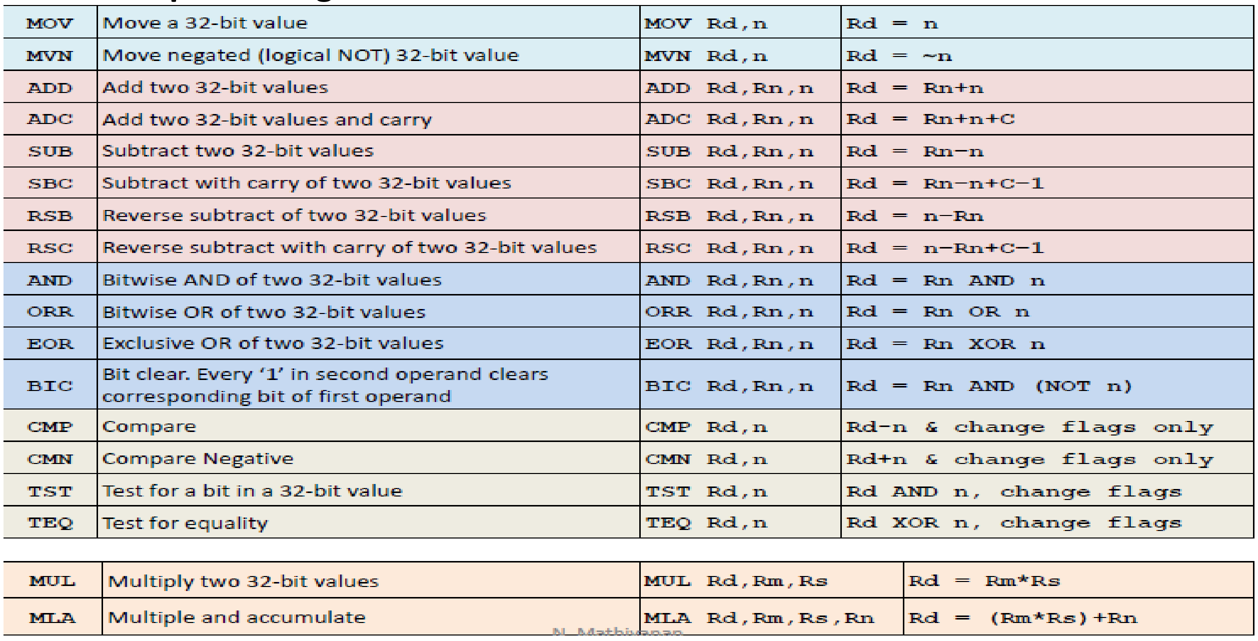
  <opcode> {<cond>}{S} Rd, Rn, n

  'cond'- indicates flags to test, 'S' – set condition flags in CPSR

  'n' may be 'Rm', '#const' or 'Rs, <shift|rotate> N'

  Rd – destination, Rn – 1st operand, Rn/Rm/Rs remains unchanged

Basic data processing instructions



MOV	Move a 32-bit value	MOV Rd,n	Rd = n
MVN	Move negated (logical NOT) 32-bit value	MVN Rd,n	Rd = ~n
ADD	Add two 32-bit values	ADD Rd,Rn,n	Rd = Rn+n
ADC	Add two 32-bit values and carry	ADC Rd,Rn,n	Rd = Rn+n+C
SUB	Subtract two 32-bit values	SUB Rd,Rn,n	Rd = Rn−n
SBC	Subtract with carry of two 32-bit values	SBC Rd,Rn,n	Rd = Rn−n+C−1
RSB	Reverse subtract of two 32-bit values	RSB Rd,Rn,n	Rd = n−Rn
RSC	Reverse subtract with carry of two 32-bit values	RSC Rd,Rn,n	Rd = n−Rn+C−1
AND	Bitwise AND of two 32-bit values	AND Rd,Rn,n	Rd = Rn AND n
ORR	Bitwise OR of two 32-bit values	ORR Rd,Rn,n	Rd = Rn OR n
EOR	Exclusive OR of two 32-bit values	EOR Rd,Rn,n	Rd = Rn XOR n
BIC	Bit clear. Every '1' in second operand clears corresponding bit of first operand	BIC Rd,Rn,n	Rd = Rn AND (NOT n)
CMP	Compare	CMP Rd,n	Rd-n & change flags only
CMN	Compare Negative	CMN Rd,n	Rd+n & change flags only
TST	Test for a bit in a 32-bit value	TST Rd,n	Rd AND n, change flags
TEQ	Test for equality	TEQ Rd,n	Rd XOR n, change flags

MUL	Multiply two 32-bit values	MUL Rd,Rm,Rs	Rd = Rm*Rs
MLA	Multiple and accumulate	MLA Rd,Rm,Rs,Rn	Rd = (Rm*Rs)+Rn

Figure 7 Basic Data Processing Instructions

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

General rules:

- All operands are 32-bit, coming from registers or literals.
- The result, if any, is 32-bit and placed in a register (with the exception of long multiply which produces a 64-bit result)
- 3-address format

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

- These instructions only work on registers, NOT memory.
- Comparisons set flags only - they do not specify Rd
- Data movement does not specify Rn
- Second operand is sent to the ALU via barrel shifter.
- Suffix S on data processing instructions updates flags in CPSR
- Operands are 32-bit wide
- 32-bit result placed in register
- Long multiply instruction produces 64-bit result

The data processing instructions manipulate data within registers. They are

- ➢ Move instructions
- ➢ Arithmetic instructions
- ➢ Logical instructions
- ➢ Comparison instructions
- ➢ Multiply instructions

The data processing operations may be classified as logical or arithmetic. The logical operations (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) perform the logical action on all corresponding bits of the operand or operands to produce the result. If the S bit is set (and Rd is not R15, see below) the V flag in the CPSR will be unaffected, the C flag will be set to the carry out from the barrel shifter (or preserved when the shift operation is LSL #0), the Z flag will be set if and only if the result is all zeros, and the N flag will be set to the logical value of bit 31 of the result.

The data processing instruction is only executed if the condition is true. The instruction produces a result by performing a specified arithmetic or logical operation on one or two operands. The first operand is always a register (Rn).The second operand may be a shifted register (Rm) or a rotated 8 bit immediate value (Imm) according to the value of the I bit in the instruction.

The condition codes in the CPSR may be preserved or updated as a result of this instruction, according to the value of the S bit in the instruction.

Certain operations (TST, TEQ, CMP, CMN) do not write the result to Rd. They are used only to perform tests and to set the condition codes on the result and always have the S bit set.

Data Processing Instructions - Examples

1.	MOVCS R0, R1	@ if carry is set @ then R0:=R1
2.	MOVS R0, #0	@ R0:=0 @ Z=1, N=0 @ C, V unaffected
3.	ADDS r0, r1, r2, LSL #3	; r0=r1+ (r2*8), flags change
4.	CMP r0, #5	; if r0 == 5, set Z flag
5.	ADDEQ r1, r2, r3	; If Z=1, r1=r2+r3 else skip
6.	AND r0, r0, #0x1	; status of bottom bit of r0
7.	BIC r0, r1, r2	; 1 in r2 bits clears r1 bits
8.	CMN r0, #6400	; flags change on r0+6400, r0 is same

Register movement

MOV R0, R2	@ R0 = R2
MVN R0, R2	@ R0=~R2

Addressing modes

- Register operands ADD R0, R1, R2
- Immediate operands a literal

  ADD R3, R3, #1 @ R3:=R3+1

  AND R8, R7, #0xff @ R8=R7 [7:0]

- Shifted register operands

One operand to ALU is routed through the Barrel shifter. Thus, the operand can be modified before it is used. Useful for fast multiplication and dealing with lists, table and other complex data structure.

Some instructions (e.g. MUL, CLZ, QADD) do not read barrel shifter.

Move Instructions:

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

- It 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	$Rd = \sim N$

Figure 8 MOV and MVN

MOV Rd, N
Rd: destination register, N: can be an immediate value or source register
Example: MOV r7, r5
MVN Rd, N; Move into Rd not of the 32-bit value from source

The second operand *N* for all data processing instructions is a register R_m or a constant preceded by #.

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 thevalue 5, and overwriting the value 8 in register *r7*.

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

Using the 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 pre-processed 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 ofthe 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 is particularly useful for loading constants into a register and achieving fast multiplies or division by a power of 2.
- Enables shifting 32-bit operand in one of the source registers left or right by a specific number of positions within the cycle time of instruction
- Facilitate fast multiply, division and increases code density

The below figure shows the data flow between the ALU and the barrel shifter.

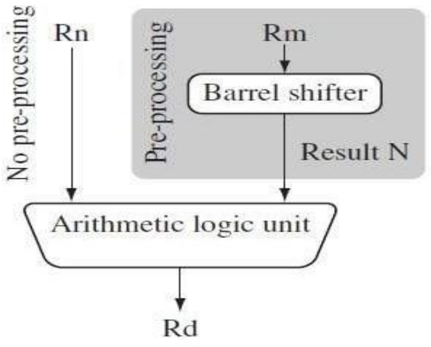


Figure 9 Barrel Shifter and ALU

Register Rn enters the ALU without any pre-processing of registers.

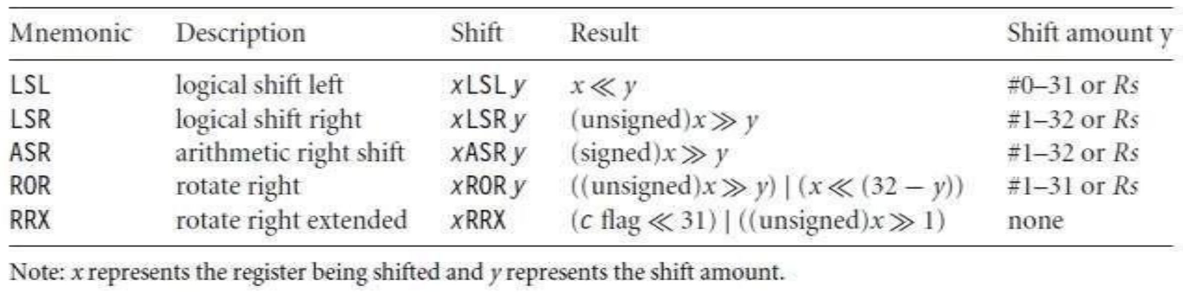
- Basic Barrel shifter operations-Shift left, shift right, rotate right

Example: Apply a logical shift left (LSL) to register Rm before moving it to the destination register. The MOV instruction copies the shift operator result N into register Rd. N represents the result of the LSL operation.

```
PRE r5 = 5 r7 = 8
MOV r7, r5, LSL #2            ; let r7 = r5*4 = (r5 << 2)
POST r5 = 5 r7 = 20
```

The example multiplies register *r5* by four and then places the result into register *r7*.

The five different shift operations that you can use within the barrel shifter are summarized in below table.

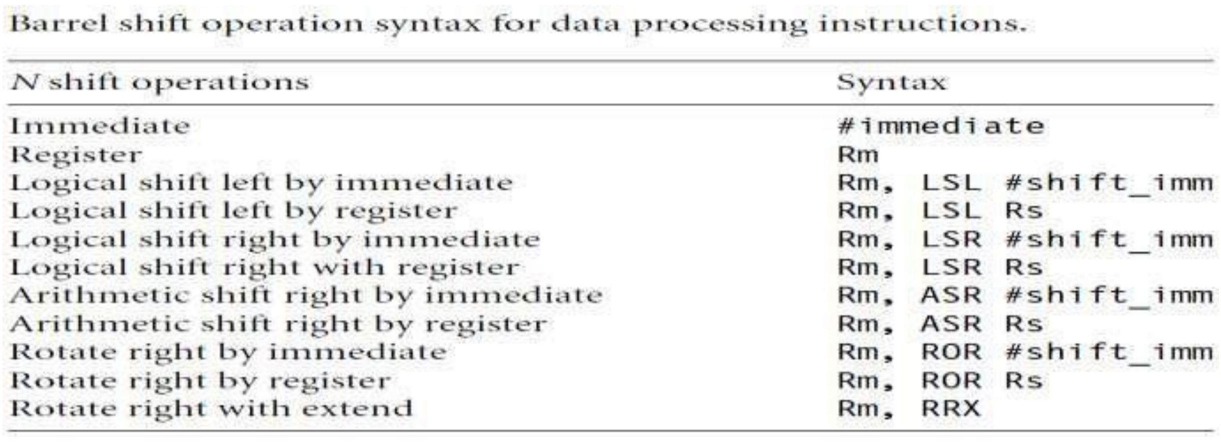


Mnemonic	Description	Shift	Result	Shift amount y
LSL	logical shift left	*x*LSL *y*	$x \ll y$	#0–31 or *Rs*
LSR	logical shift right	*x*LSR *y*	$(\text{unsigned})x \gg y$	#1–32 or *Rs*
ASR	arithmetic right shift	*x*ASR *y*	$(\text{signed})x \gg y$	#1–32 or *Rs*
ROR	rotate right	*x*ROR *y*	$((\text{unsigned})x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

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

Figure 10 Barrel Shifter related Shift Operations

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



Barrel shift 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`

Figure 11 Barrel Shifter Operation Syntax for Data Processing Instructions

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

Using a Barrel Shifter: The 2nd Operand

- Register, optionally with shift operation
- Shift value can be either be:
  - 5 bit unsigned integer
  - Specified in bottom byte of another register.
- Used for multiplication by constant

  Immediate value
  - 8 bit number, with a range of 0-255.
  - Rotated right through even number of positions
  - Allows increased range of 32-bit constants to be loaded directly into registers

Arithmetic Instructions:

The arithmetic instructions implement addition and subtraction of 32-bit signed and unsigned values. 3-operand form

Examples

- SUB r0, r1, r2

  Subtract value stored in r2 from that of r1 and store in r0
- SUBS r1, r1, #1

  Subtract 1 from r1 and store result in r1 and update Z and C flags

Example: This simple subtract instruction subtracts a value stored in register *r2* from a value store 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: This reverse subtract instruction (RSB) subtracts r1 from the constant value #0, writing the result to r0. This instruction use 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 = nzcviFt_USER r1 = 0x00000001

SUBS r1, r1, #1

POST cpsr = nZCviFt_USER r1 = 0x00000000

Using the Barrel Shifter with Arithmetic Instructions:

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)	$Rd = Rn \& \sim N$

Figure 12 Logical Instructions

Example: Shows a logical OR operation between registers r1 and r2. r0 holds the result.

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

ORR r0, r1, r2

POST r0 = 0x12345678

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)

R2 contains a binary pattern where every binary 1 in r2 clears a corresponding bit location in register r1 .Useful in manipulating status flags and interrupt masks.

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. For these instructions no needs to apply the S suffix for update the flags.

- Enables comparison of 32 bit values
- Updates CPSR flags but do not affect other registers

Examples

CMP r0, r9 ;–Flags set as a result of r0 - r9

TEQ r0, r9 ;–Flags set as a result r0 ex-or r9

TST r0, r9 ;–Flags set as a result of r0 & r9

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 = nzcviFt_USER r0 = 4; r9 = 4

CMP r0, r9 POST cpsr = nZcviFt_USER

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

TST instruction is a logical AND operation

TEQ is a logical exclusive OR operation.

For each, the results are discarded but the condition bits are updated in the cpsr.

Multiply Instructions:

There are 2 classes of multiply - producing 32-bit and 64-bit results

- 32-bit versions on an ARM7TDMI will execute in 2 - 5 cycles
- MUL r0, r1, r2 ; r0 = r1 * r2
- MLA r0, r1, r2, r3 ; r0 = (r1 * r2) + r3

64-bit multiply instructions offer both signed and unsigned versions. For this instruction there are 2 destination registers

```
[U|S]MULL r4, r5, r2, r3        ; r5:r4 = r2 * r3

[U|S]MLAL r4, r5, r2, r3        ; r5:r4 = (r2 * r3) + r5:r4
```

Most ARM cores do not offer integer divide instructions. Division operations will be performed by C library routines or inline shifts

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 multiply 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.

Example: This example shows a simple multiply instruction that multiplies registers r1 and r2 together and places the result into register r0.

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

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

POST r0 = 0x00000004 r1 = 0x00000002r2 = 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.

Example: Shows an example of a long unsigned multiply instruction. 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.

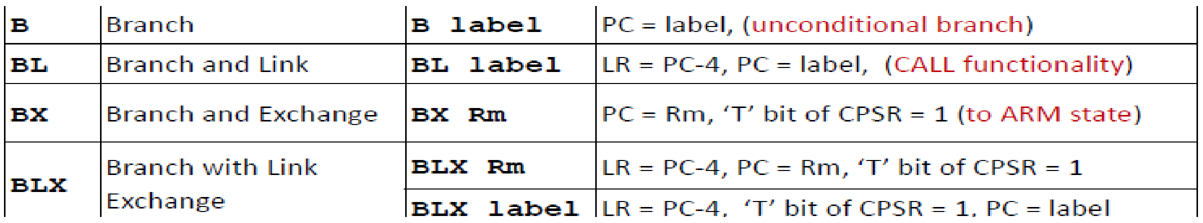
PRE r0 = 0x00000000 r1 = 0x00000000

r2 = 0xf0000002 r3 = 0x00000002

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

POST r0 = 0xe0000004; = RdLo r1 = 0x00000001; = RdHi

1.7 BRANCH INSTRUCTION



B	Branch	**B label**	PC = label, (unconditional branch)
BL	Branch and Link	**BL label**	LR = PC-4, PC = label, (CALL functionality)
BX	Branch and Exchange	**BX Rm**	PC = Rm, ‘T’ bit of CPSR = 1 (to ARM state)
BLX	Branch with Link Exchange	**BLX Rm**	LR = PC-4, PC = Rm, ‘T’ bit of CPSR = 1
		BLX label	LR = PC-4, ‘T’ bit of CPSR = 1, PC = label

Figure 13 Types of branch instructions

Address label is stored in the instruction as a signed pc-relative offset
Branch instruction: B label
Example: B forward
Conditional Branch: B<cond> label
Example: BNE loop

Branch has a condition associated with it and executed if condition codes have the correct value
Branch and Exchange (BX)-This instruction is only executed if the condition is true. This instruction performs a branch by copying the contents of a general register, Rn, into the program counter, PC. The branch causes a pipeline flush and refill from the address specified by Rn. This instruction also permits the instruction set to be exchanged. When the instruction is executed, the value of Rn[0] determines whether the instruction stream will be decoded as ARM or THUMB instructions.

Branch and Branch with Link (B, BL)-The instruction is only executed if the condition is true. Branch instructions contain a signed 2's complement 24 bit offset. This is shifted left two bits, sign extended to 32 bits, and added to the PC. The instruction can therefore specify a branch of +/- 32Mbytes. The branch offset must take account of the prefetch operation, which causes the PC to be 2 words (8 bytes) ahead of the current instruction.

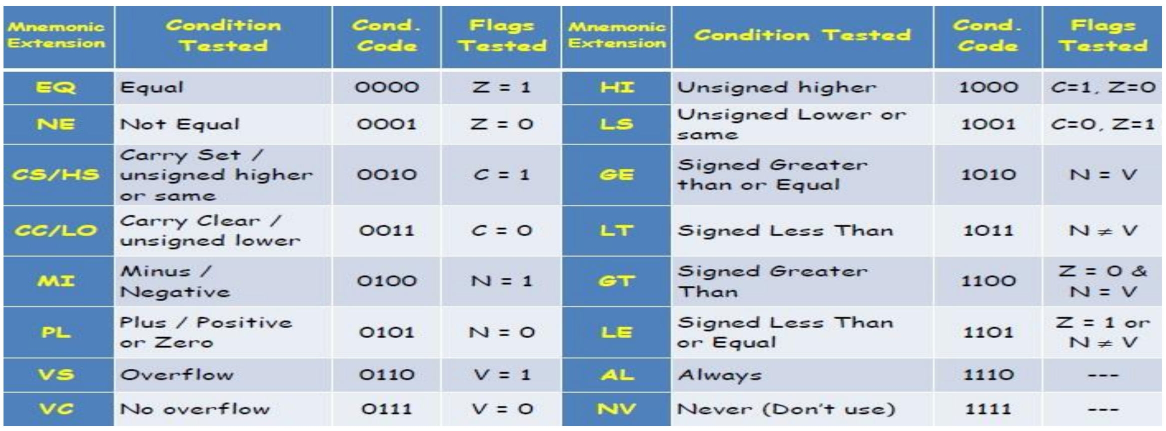
Branches beyond +/- 32Mbytes must use an offset or absolute destination which has been previously loaded into a register. In this case the PC should be manually saved in R14 if a Branch with Link type operation is required.

The link bit Branch with Link (BL) writes the old PC into the link register (R14) of the current bank. The PC value written into R14 is adjusted to allow for the prefetch, and contains the address of the instruction following the branch and link instruction.

Note that the CPSR is not saved with the PC and R14 [1:0] are always cleared. To return from a routine called by Branch with Link use MOV PC,R14 if the link register is still valid or LDM Rn!,{..PC} if the link register has been saved onto a stack pointed to by Rn.

Features of Conditional Execution instructions

Almost all ARM instructions have a condition field which allows it to be executed conditionally. Improves execution speed and offers high code density



Mnemonic Extension	Condition Tested	Cond. Code	Flags Tested	Mnemonic Extension	Condition Tested	Cond. Code	Flags Tested
EQ	Equal	0000	Z = 1	HI	Unsigned higher	1000	C=1, Z=0
NE	Not Equal	0001	Z = 0	LS	Unsigned Lower or same	1001	C=0, Z=1
CS/HS	Carry Set / unsigned higher or same	0010	C = 1	GE	Signed Greater than or Equal	1010	N = V
CC/LO	Carry Clear / unsigned lower	0011	C = 0	LT	Signed Less Than	1011	N ≠ V
MI	Minus / Negative	0100	N = 1	GT	Signed Greater Than	1100	Z = 0 & N = V
PL	Plus / Positive or Zero	0101	N = 0	LE	Signed Less Than or Equal	1101	Z = 1 or N ≠ V
VS	Overflow	0110	V = 1	AL	Always	1110	---
VC	No overflow	0111	V = 0	NV	Never (Don't use)	1111	---

Figure 14 Condition Codes

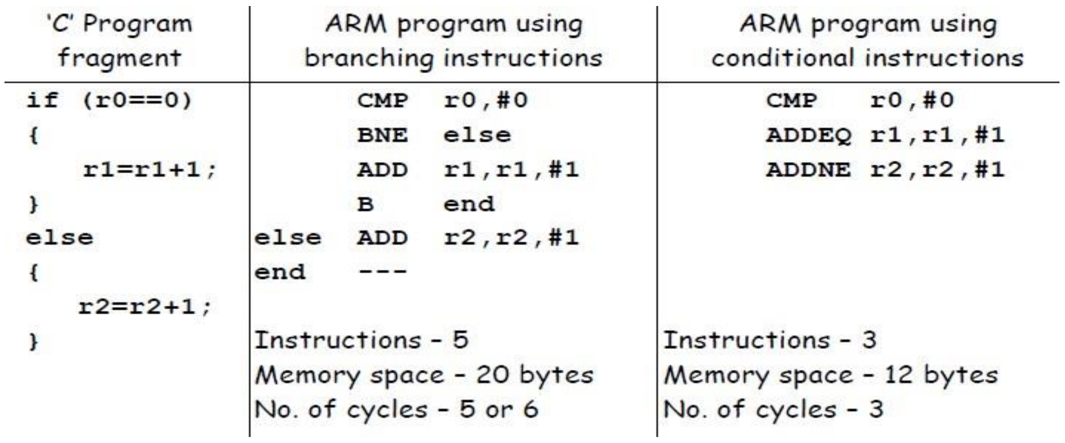
Conditional Execution and Flags

In ARM state, all instructions are conditionally executed according to the state of the CPSR condition codes and the instruction's condition field. This field (bits 31:28) determines the circumstances under which an instruction is to be executed. If the state of the C, N, Z and V flags fulfills the conditions encoded by the field, the instruction is executed, otherwise it is ignored. There are sixteen possible conditions, each represented by a two-character suffix that can be appended to the instruction's mnemonic.

For example, a Branch (B in assembly language) becomes BEQ for "Branch if Equal", which means the Branch will only be taken if the Z flag is set. In practice, fifteen different conditions may be used. The sixteenth (1111) is reserved, and must not be used. In the absence of a suffix, the condition field of most instructions is set to "Always" (suffix AL). This means the instruction will always be executed regardless of the CPSR condition codes.

By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using "S". CMP does not need "S".

Illustration:



'C' Program fragment	ARM program using branching instructions	ARM program using conditional instructions
if (r0==0)	CMP r0,#0	CMP r0,#0
{	BNE else	ADDEQ r1,r1,#1
r1=r1+1;	ADD r1,r1,#1	ADDNE r2,r2,#1
}	B end	
else	else ADD r2,r2,#1	
{	end ---	
r2=r2+1;		
}	Instructions - 5	Instructions - 3
	Memory space - 20 bytes	Memory space - 12 bytes
	No. of cycles - 5 or 6	No. of cycles - 3

Figure 15 ARM Vs C code

Branch Instructions - Examples

1. Example of using ‘B’ instruction:

```
CMP r0,#0              ; check if r0 == 0
BNE r2inc              ; if r0 !=0 branch to ‘r2inc’
ADD r1,r1,#1           ; r1 += 1
B next                 ; unconditional branch to ‘next’
r2inc :ADD r2,r2,#1          ; r2 += 1
next -----------------------; continue
```

2. Example of using ‘BL’ instruction

```
BL funct1                    ; save return addr. & subroutine
CMP r0,#5                    ; next instruction
func1 ADD r0,r0,#1           ; subroutine codes
MOV pc,lr                    ; return to program
```

3. Example of using ‘BLX’ instruction

In the above example replace ‘MOV lr, pc’ and ‘BX r0’ by ‘BLX r0’

Subroutines

•Called from main program using ‘BL’ instruction
•The instruction places PC-4 in LR and addr of subroutine in PC
•Last instruction in subroutine is MOV PC, LR or BX LR
•If not a leaf routine (nested subroutine):
Store LR in stack using STMxx r13, {……, r14} at entry
Restore LR from stack using LDMxx r13, {…, r14} before exit
The following program fragment implements ‘is the value of ‘A’ 1 or 5 or 8?’
Let the value of ‘A’ be in register r0.
TEQ r0, #1 ; if r0 == 0, then Z = 1
TEQNE r0, #5; if Z! = 1 & if r0==5, then Z=1
TEQNE r0, # 8 ; if Z != 1 & if r0==8, then Z=1
BNE error; if Z! = 1 branch to report error

1.8 Data Transfer Instructions 1

- The ARM Architecture is a Load/Store architecture
- No direct manipulation of memory contents
- Memory must be loaded into the CPU to be modified, then written back out
- Cores are either in ARM state or Thumb state
- This determines which instruction set is being executed
- An instruction must be executed to switch between states
- The architecture allows programmers and compilation tools to reduce branching through the use of conditional execution
- Method differs between ARM and Thumb, but the principle is that most (ARM) or all (Thumb) instructions can be executed conditionally.

Load-Store Instructions

Load-store instructions transfer data between memory and processor registers. There are three types of load-store instructions:

- Single register transfer

Data types supported are signed and unsigned words (32 bits), half-word, bytes

- Multiple-register transfer

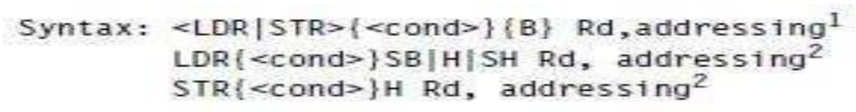
Transfer multiple registers between memory and the processor in a single instruction

- Swap

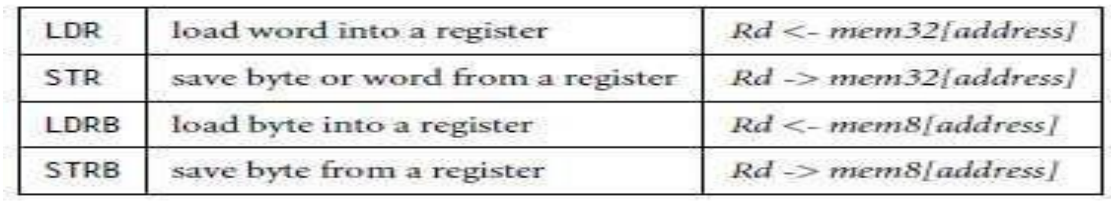
Swaps content of a memory location with the contents of a register

Single-Register Transfer:

- Transfers boundary aligned Word/HW/Byte between memory & register
- LDR and STR instructions
- Address of memory loc. is given by Base Addr. +/- Offset
- Addressing modes: method of providing offset
- Register addressing – A register contains offset
- 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.

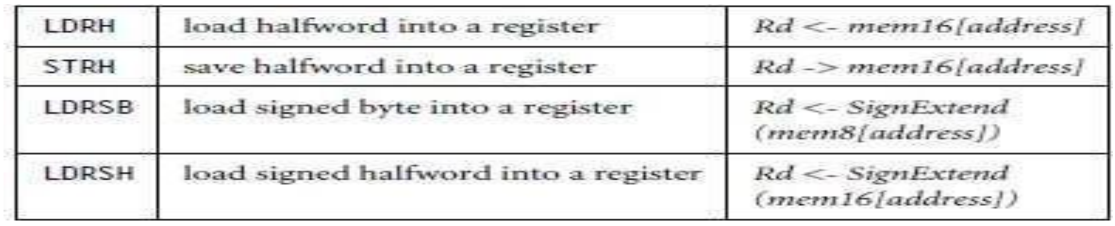


```
Syntax: <LDR|STR>{<cond>}{B} Rd,addressing1
        LDR{<cond>}SB|H|SH Rd, addressing2
        STR{<cond>}H Rd, addressing2
```



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]*

Figure 16 Load and Store data on a boundary alignment



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])*

Load & Store data on a boundary alignment

- LDR, LDRH, LDRB
  Load (word, half-word, byte)
- STR, STRH, STRB
  Store (word, half-word, byte)
  Supports different addressing modes
  Register indirect: LDR r0, [r1]
  Immediate: LDR r0, [r1, #4]
  12-bit offset added to the base register
  Register operation: LDR r0, [r1,-r2]
  Address calculated using base register and another register

Example: LDR r0, [r1]
STR r0, [r1]
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.Register r1 is called the base address register.
Immediate addressing – Immediate constant is offset
Scaled addressing – Offset in a register is scaled using shift operation
Syntax: <opcode> {<condition>}{<type>}Rd,[Rn{,<offset>}]
<type> - H, HS, B, BS
Rd – source/destination register
Rn – Base address
<offset> - ‘Rm’ or # (0-4095) or ‘Rm, <shift>#n’

Single-Register Load-Store Addressing Modes:

Examples

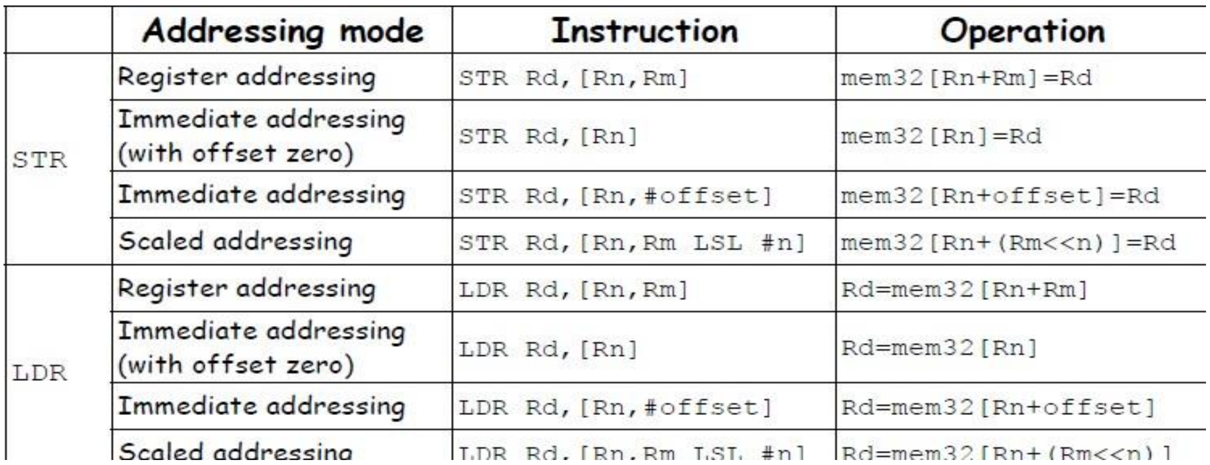
1. LDRB r3, [r8, #3] ; load at bottom byte of r3 from mem8[r8+3]

2. STRB r10, [r7,-r4] ; store bottom byte of r10 at mem8[r7-r4]

3. LDRH r1, [r0] ; load at bottom halfword of r1 from mem16[r0]

4. STRH r10, [r7,-r4] ; store bottom halfword of r10 at mem16 [r7-r4]

5. LDR r0, [r1, r2] ; r0=mem32[r1+r2]



	Addressing mode	Instruction	Operation
STR	Register addressing	`STR Rd, [Rn,Rm]`	`mem32[Rn+Rm]=Rd`
	Immediate addressing (with offset zero)	`STR Rd, [Rn]`	`mem32[Rn]=Rd`
	Immediate addressing	`STR Rd, [Rn,#offset]`	`mem32[Rn+offset]=Rd`
	Scaled addressing	`STR Rd, [Rn,Rm LSL #n]`	`mem32[Rn+(Rm<<n)]=Rd`
LDR	Register addressing	`LDR Rd, [Rn,Rm]`	`Rd=mem32[Rn+Rm]`
	Immediate addressing (with offset zero)	`LDR Rd, [Rn]`	`Rd=mem32[Rn]`
	Immediate addressing	`LDR Rd, [Rn,#offset]`	`Rd=mem32[Rn+offset]`
	Scaled addressing	`LDR Rd, [Rn,Rm LSL #n]`	`Rd=mem32[Rn+(Rm<<n)]`

Figure 17 Single Register Transfer Instructions Addressing Modes

The ARM instruction set provides different modes for addressing memory. These modes incorporate one of the indexing methods:

Preindexed:

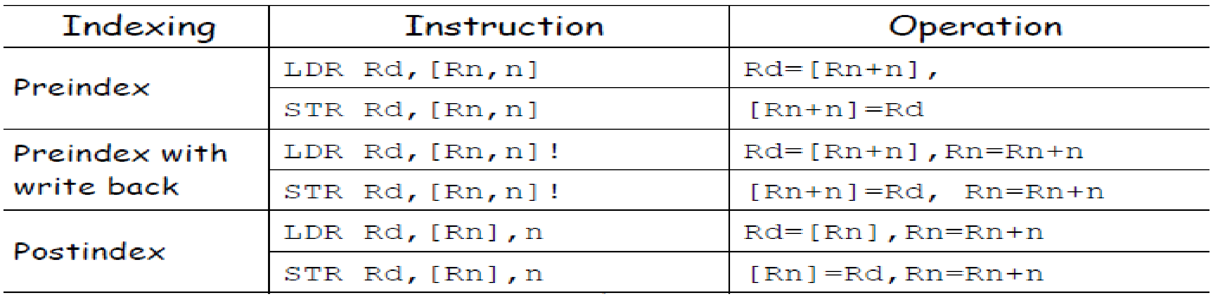
<opcode> {<cond>}{<type>}Rd,[Rn{,<offset>]

Preindexed with write back (note the exclamation symbol)

<opcode> {<cond>}{<type>}Rd,[Rn{,<offset>]!

Postindexed

<opcode> {<cond>}Rd,[Rn],<offset>



Indexing	Instruction	Operation
Preindex	`LDR Rd,[Rn,n]`	`Rd=[Rn+n],`
	`STR Rd,[Rn,n]`	`[Rn+n]=Rd`
Preindex with write back	`LDR Rd,[Rn,n]!`	`Rd=[Rn+n],Rn=Rn+n`
	`STR Rd,[Rn,n]!`	`[Rn+n]=Rd, Rn=Rn+n`
Postindex	`LDR Rd,[Rn],n`	`Rd=[Rn],Rn=Rn+n`
	`STR Rd,[Rn],n`	`[Rn]=Rd,Rn=Rn+n`

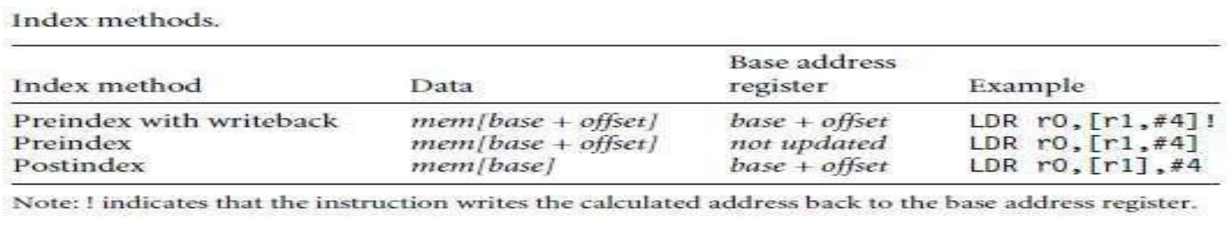
Figure 18 ARM Indexing Modes

Preindex with write back: It calculates an address from a base register plus address offset and then updates that address base register with the new address.

Preindex: It calculates an address from a base register plus address offset but does not update the address base register.

Postindex: It only updates the address base register after the address is used.

Note: The pre-index mode is useful for accessing an element in a data structure. The post index and pre index with write back modes are useful for traversing an array.



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.

Figure 19 ARM indexing modes syntax

The offset address can provide in the instructions in different types. They are

Immediate: It means the address is calculated using the base address register and a 12-bit offset encoded in the instruction.

Register: It means the address is calculated using the base address register and a specific register's contents.

Scaled: It means the address is calculated using the base address register and a barrel shift operation.

Example: Index addressing modes

PRE r0 = 0x00000000 r1 = 0x00090000

mem32 [0x00009000] = 0x01010101, mem32 [0x00009004] = 0x02020202

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

POST (1) r0 = 0x02020202 r1 = 0x00009004

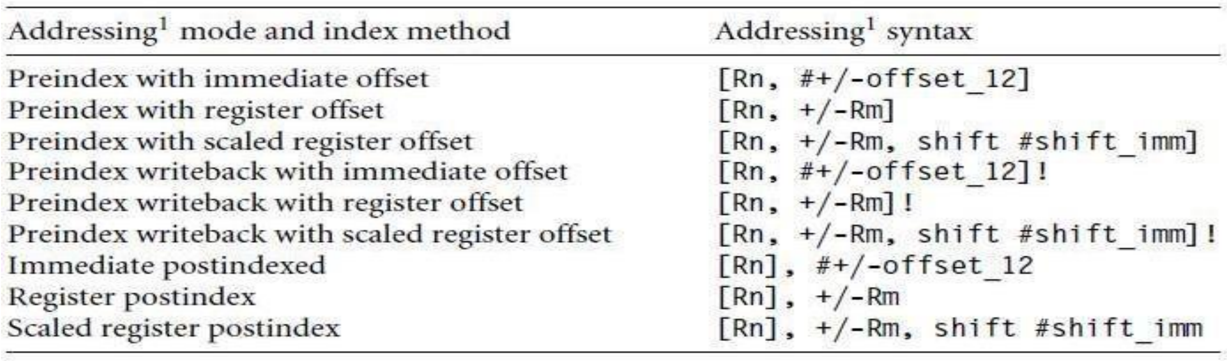
Preindexing:LDR r0, [r1, #4] POST (2) r0 = 0x02020202, r1 = 0x00009000

Postindexing: LDR r0, [r1], #4

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

Table below shows the addressing modes available for load and store of a 32-bit word or an unsigned byte.

- 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.



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

Figure 20 Single Register Load Store Addressing

Table below provides an example of the different variations of the LDR instruction.

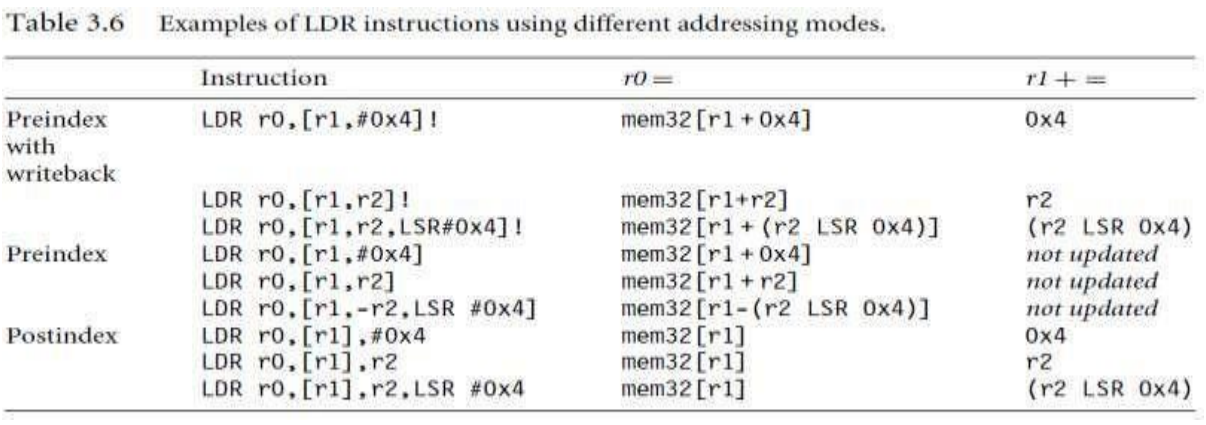
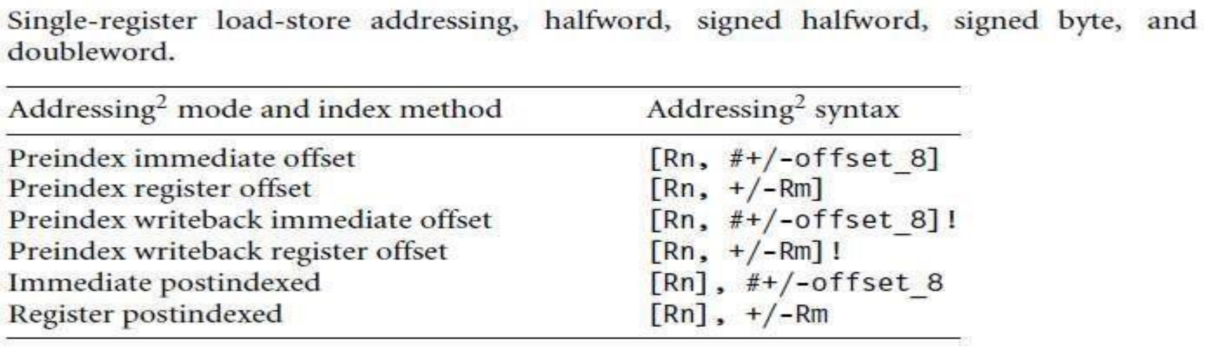


Table 3.6 Examples of LDR instructions using different addressing modes.

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

Figure 21 Examples of LDR instructions using different addressing modes

Table below shows the addressing modes available on load and store instructions using 16-bit half word orsigned byte data.

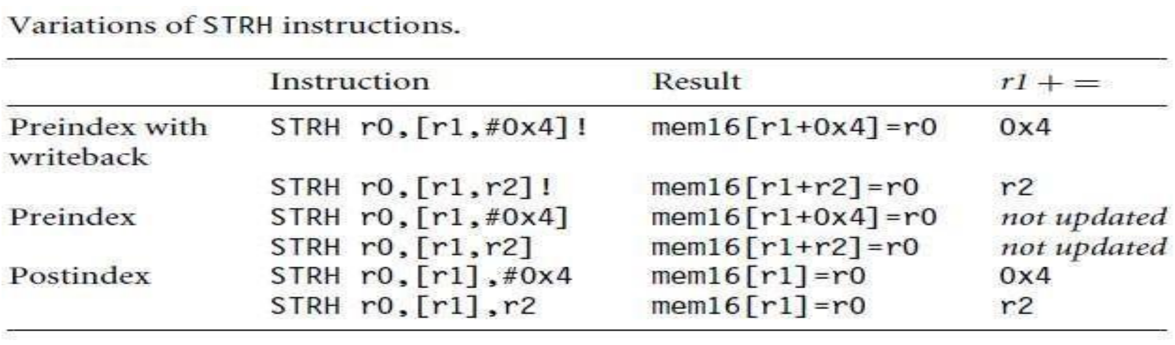


Single-register load-store addressing, halfword, signed halfword, signed byte, and doubleword.

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`

Figure 22 Single register load store addressing

There are no STRSB or STRSH instructions since STRH store both a signed and unsigned half word; similarly STRB stores signed and unsigned bytes. Table below shows the variations for STRH instructions.



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`

Figure 23 Variations of STRH instructions

Indexing methods – Examples

1. LDR r0,[r1,#04]!	; preindex with write back,
	; r0 = mem32[r1+04], r1 += 04
2. LDR r0,[r1,r2]	; preindex,
	; r0 = mem32[r1+r2], r1 not updated
3. LDR r0,[r1],r2 LSR #04	; postindex,
	; r0 = mem32[r1], r1 += r2 >> 04
4. LDRB r7,[r6,#-1]!	; preindex with write back
	; bottom byte of r7 = mem8[r6-1],
	; then r6 = r6-1
5. STRH r0,[r1,r2]!	; preindex with write back,
	; mem16[r1+r2] = bottom halfword of r0,
	; r1 += r2
6. STRH r0,[r1],#04	; postindex,
	; mem16[r1] = r0, r1 += 04

1.9 Block Transfer Instructions (Multiple Register Load/Store)

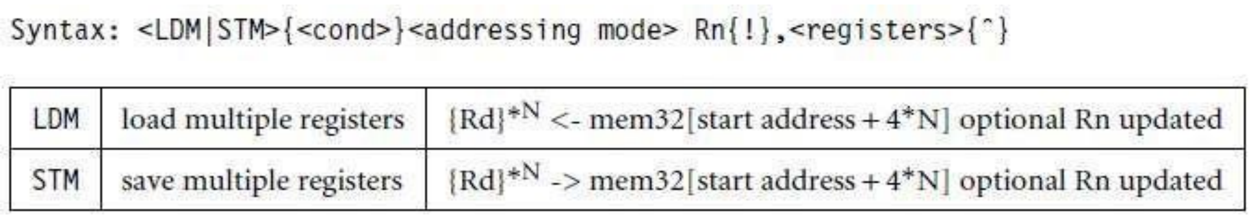
•Transfers data between multiple registers & memory in single instruction
•Instructions: LDM and STM
•Use: stack, block move, temporary store & restore
•Advantages: small code size, single instruction fetch from memory
•Disadvantages: can't be interrupted, increases interrupt latency
•Syntax:
<Opcode> {<cond>} <mode>Rn{!}, <registers>
•Rn – Base register, '!' update base reg. after data transfer (option)

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. Load-store multiple instructions can increase interrupts latency. ARM implementations do not usually interrupt instructions while they are executing. If an interrupt has been raised, then it has no effect until the load-store multiple instruction is complete.

Any subset of current bank of registers can be transferred to memory or fetched from memory

- LDM
- SDM

The base register Rn determines source or destination address
More efficient for moving blocks of memory and saving and restoring context and stack. These instructions can increase interrupt latency. Instruction executions are not interrupted by ARM



```
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

Figure 24 LDM and STM syntax

Stack Processing

A stack is implemented as a linear data structure which grows up (ascending) or down (descending) .Stack pointer hold the address of the current top of the stack.

Modes of Stack Operation

ARM multiple register transfer instructions support

- Full ascending: grows up, SP points to the highest address containing a valid item
- Empty ascending: grows up, SP points to the first empty location above stack
- Full descending: grows down, SP points to the lowest address containing a valid data
- Empty descending: grows down, SP points to the first location below the stack

Some Stack Instructions

Full Ascending

- LDMFA: translates to LDMIA (POP)
- STMFA: translates to STMIB (PUSH)
- SP points to last item in stack

Empty Descending

- LDMED: translates to LDMIB (POP)
- STMED: translates to STMIA (PUSH)
- SP points to first unused location

Addressing Modes

LDMIA|IB|DA|DB ex: LDMIA Rn! , {r1-r3}

STMIA|IB|DA|DB

Stack Example

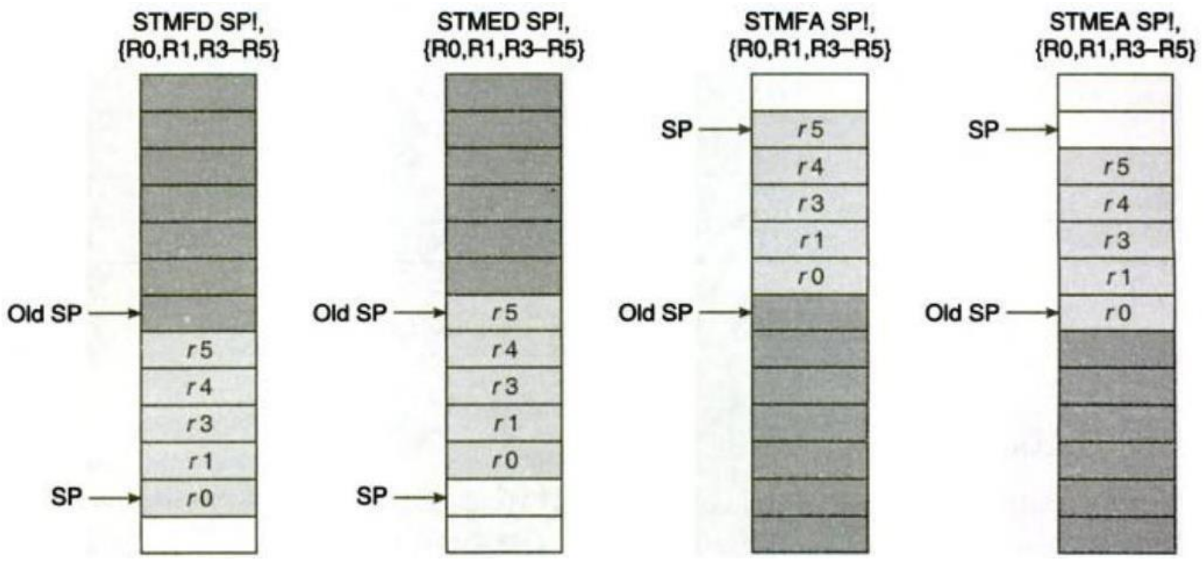


Figure 25 Stack Example

Table below shows the different addressing modes for the load-store multiple instructions.

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

Figure 26 Addressing Mode for load store multiple instructions

Here N is the number of registers in the list of registers. 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 when register Rn is followed by the '!' character.
Example: Register r0 is the base register Rn and is followed by! indicating that the register is updated after the instruction is executed.

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

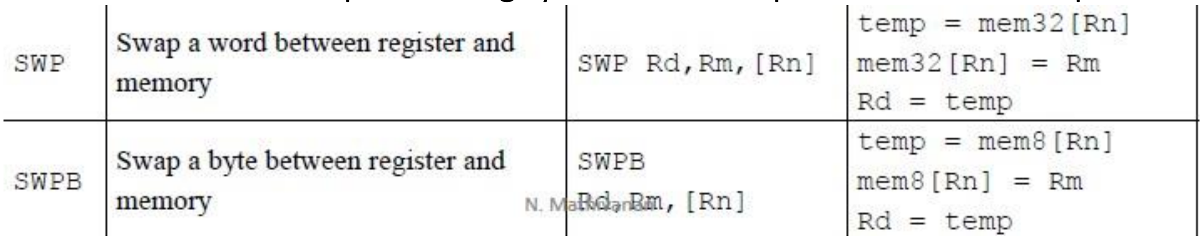
Moving a large data block

```
; R12 points to the start if the source data
; R14 points to the end of the source data
; R13 points to the start of the destination data
Loop LDMIA R12!, {R0-R11}       ; load 48 bytes...
STMIA R13!, {R0-R11}           ; ...and store them
CMP R12, R14                    ; check for the end
 BNE Loop                       ; and loop until done
```

1.10 Multiplication, Swap, PSR Instructions

Special case of load store instruction
Swap instructions (also known as semaphore instructions)

- SWP: swap a word between memory and register
- SWPB: swap a byte between memory and register
- Useful for implementing synchronization primitives like semaphore



SWP	Swap a word between register and memory	SWP Rd,Rm,[Rn]	temp = mem32[Rn]
			mem32[Rn] = Rm
			Rd = temp

SWPB	Swap a byte between register and memory	SWPB Rd,Rm,[Rn]	temp = mem8[Rn]
			mem8[Rn] = Rm
			Rd = temp

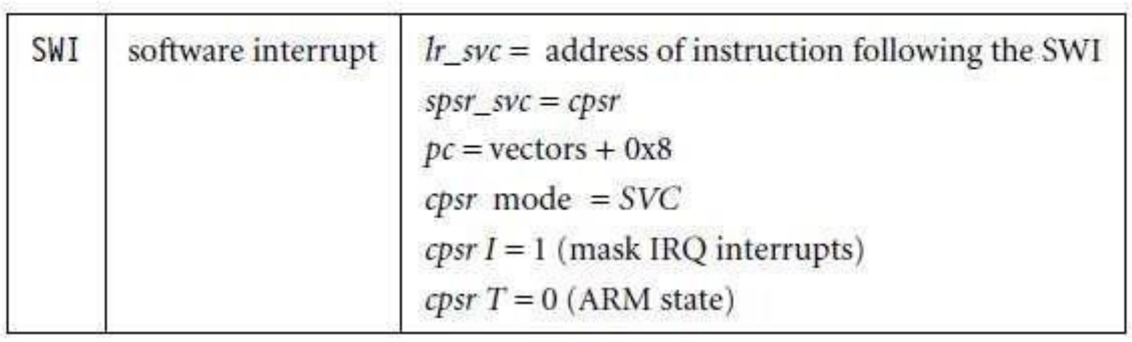
Figure 27 Swap Instruction

Program Status Register Instructions
•Instructions to read/write from/to CPSR or SPSR
•Instructions: MRS, MSR
Syntaxes:
MRS{<cond>} Rd,<CPSR|SPSR>
MSR{<cond>} <CPSR|SPSR>,Rm
MSR{<cond>} <CPSR|SPSR>_<fields>,Rm
MSR{<cond>} <CPSR|SPSR>_<fields>,#immediate
•Modifying CPSR, SPSR: Read, Modify and Write back technique

Read CPSR/SPSR using MRS
Modify relevant bits
Transfer to CPSR/SPSR using MSR
•Note: In user mode all fields can be read, but flags alone can be modified
Examples:
Program to enable FIQ (executed in svc mode)
MRS r1, cpsr ; copies CPSR into r1
BIC r1, #0x40 ; clears B6, i.e. FIQ interrupt mask bit
MSR cpsr, r1 ; copies r1 into CPSR
Program to change mode (from svc mode to fiq mode)
MRS r0, cpsr ; get CPSR into r0
BIC r0,r0, 0x1F ; clear the mode bits, i.e. 5 LSB bits - B[4:0]
ORR r0,r0, 0x11 ; set to FIQ mode
MSR cpsr,r0 ; write r0 into CPSR

1.11 Interrupts and Programming Examples
Software Interrupt Instructions
•Use: User mode applications to execute OS routines
•When executed, mode changes to supervisor mode
•Syntax: SWI {cond} SWI_number
  Example: SWI 0x123456
•Return instruction from SWI routine: MOV PC, r14
The Thumb software interrupt (SWI) instruction causes a software interrupt exception. If any interrupt or exception flag is raised in Thumb state, the processor automatically reverts back to ARM state to handle the exception. The Thumb SWI instruction has the same effect and nearly the same syntax as the ARM equivalent. It differs in that the SWI number is limited to the range 0 to 255 and it is not conditionally executed.

```
Syntax: SWI immediate
```

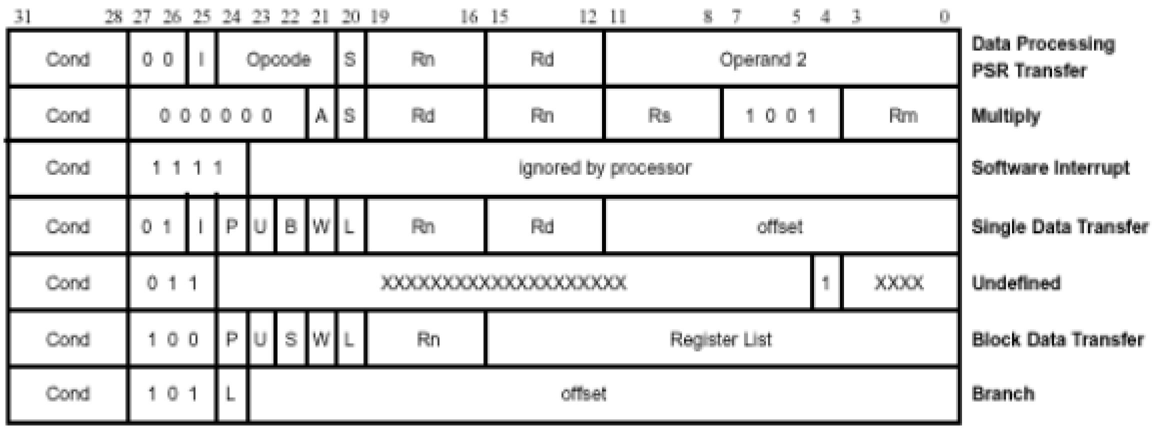


SWI	software interrupt	$lr\_svc =$ address of instruction following the SWI $spsr\_svc = cpsr$ $pc =$ vectors + 0x8 $cpsr$ mode $= SVC$ $cpsr\ I = 1$ (mask IRQ interrupts) $cpsr\ T = 0$ (ARM state)

Figure 28 SWI Instruction

1.12 Instruction Encoding-Data Processing Instructions

A summary of the ARM7 instruction set is shown in Figure 29.



31 28 27 26 25 24 23 22 21 20 19 16 15 12 11 8 7 5 4 3 0

Cond	0 0	I	Opcode	S	Rn	Rd	Operand 2				Data Processing PSR Transfer
Cond	0 0 0 0 0 0	A	S	Rd	Rn	Rs	1 0 0 1	Rm			Multiply
Cond	1 1 1 1	ignored by processor									Software Interrupt
Cond	0 1	I	P	U	B	W	L	Rn	Rd	offset	Single Data Transfer
Cond	0 1 1	XXXXXXXXXXXXXXXXXXXX	1	XXXX							Undefined
Cond	1 0 0	P	U	S	W	L	Rn	Register List			Block Data Transfer
Cond	1 0 1	L	offset								Branch

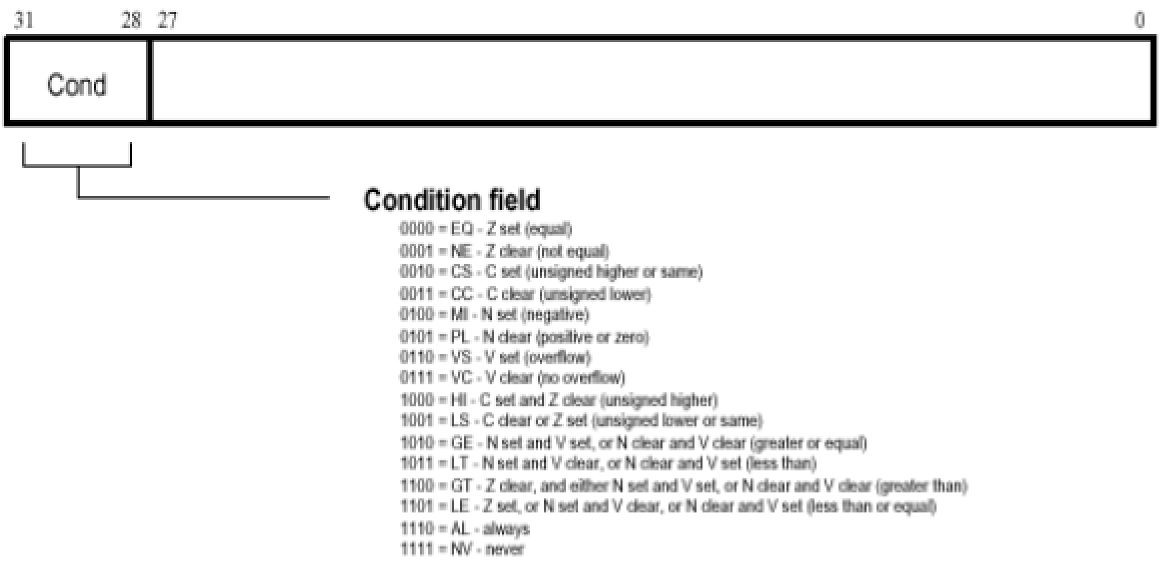
Figure 29 Instruction Set Summary

The Condition Field

All ARM instructions are conditionally executed, which means that their execution may or may not take place depending on the values of the N, Z, C and V flags in the CPSR. The condition encoding is shown in Figure 29.

If the always (AL) condition is specified, the instruction will be executed irrespective of the flags. The never (NV) class of condition codes shall not be used as they will be redefined in future variants of the ARM architecture.

If a NOP is required it is suggested that MOV R0, R0 be used. The assembler treats the absence of a condition code as though always had been specified. The other condition codes have meanings as detailed in Figure 30.
For instance code 0000 (EQual) causes the instruction to be executed only if the Z flag is set. This would correspond to the case where a compare (CMP) instruction had found the two operands to be equal. If the two operands were different, the compare instruction would have cleared the Z flag and the instruction will not be executed.



0000 = EQ - Z set (equal)
0001 = NE - Z clear (not equal)
0010 = CS - C set (unsigned higher or same)
0011 = CC - C clear (unsigned lower)
0100 = MI - N set (negative)
0101 = PL - N clear (positive or zero)
0110 = VS - V set (overflow)
0111 = VC - V clear (no overflow)
1000 = HI - C set and Z clear (unsigned higher)
1001 = LS - C clear or Z set (unsigned lower or same)
1010 = GE - N set and V set, or N clear and V clear (greater or equal)
1011 = LT - N set and V clear, or N clear and V set (less than)
1100 = GT - Z clear, and either N set and V set, or N clear and V clear (greater than)
1101 = LE - Z set, or N set and V clear, or N clear and V set (less than or equal)
1110 = AL - always
1111 = NV - never

Figure 30 Condition Codes

Branch and Branch with link (B, BL)

The instruction is only executed if the condition is true. The instruction encoding is shown in Figure 31.Branch instructions contain a signed 2's complement 24 bit offset. This is shifted left two bits, sign extended to 32 bits, and added to the PC. The instruction can therefore specify a branch of +/- 32Mbytes.

The branch offset must take account of the prefetch operation, which causes the PC to be 2 words (8 bytes) ahead of the current instruction. Branches beyond +/- 32Mbytes must use an offset or absolute destination that has been previously loaded into a register. In this case the PC should be manually saved in R14 if a Branch with Link type operation is required.

The link bit

Branch with Link (BL) writes the old PC into the link register (R14) of the current bank. The PC value written into R14 is adjusted to allow for the prefetch, and contains the address of the instruction following the branch and link instruction. Note that the CPSR is not saved with the PC. To return from a routine called by Branch with Link use MOV PC,R14 if the link register is still valid or LDM Rn!,{..PC} if the link register has been saved onto a stack pointed to by Rn.

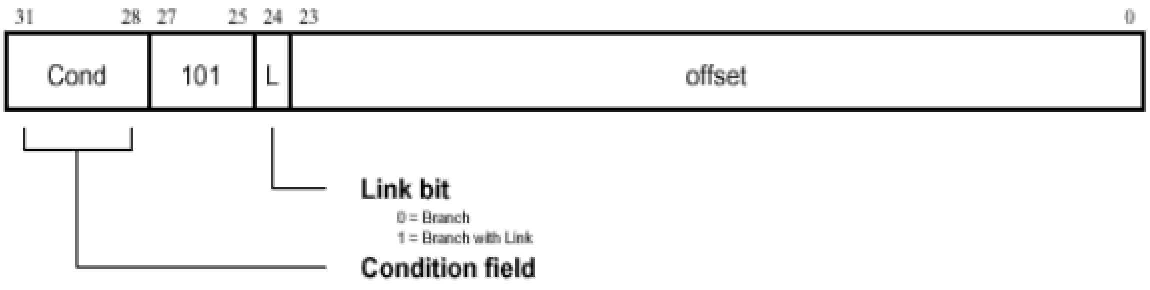
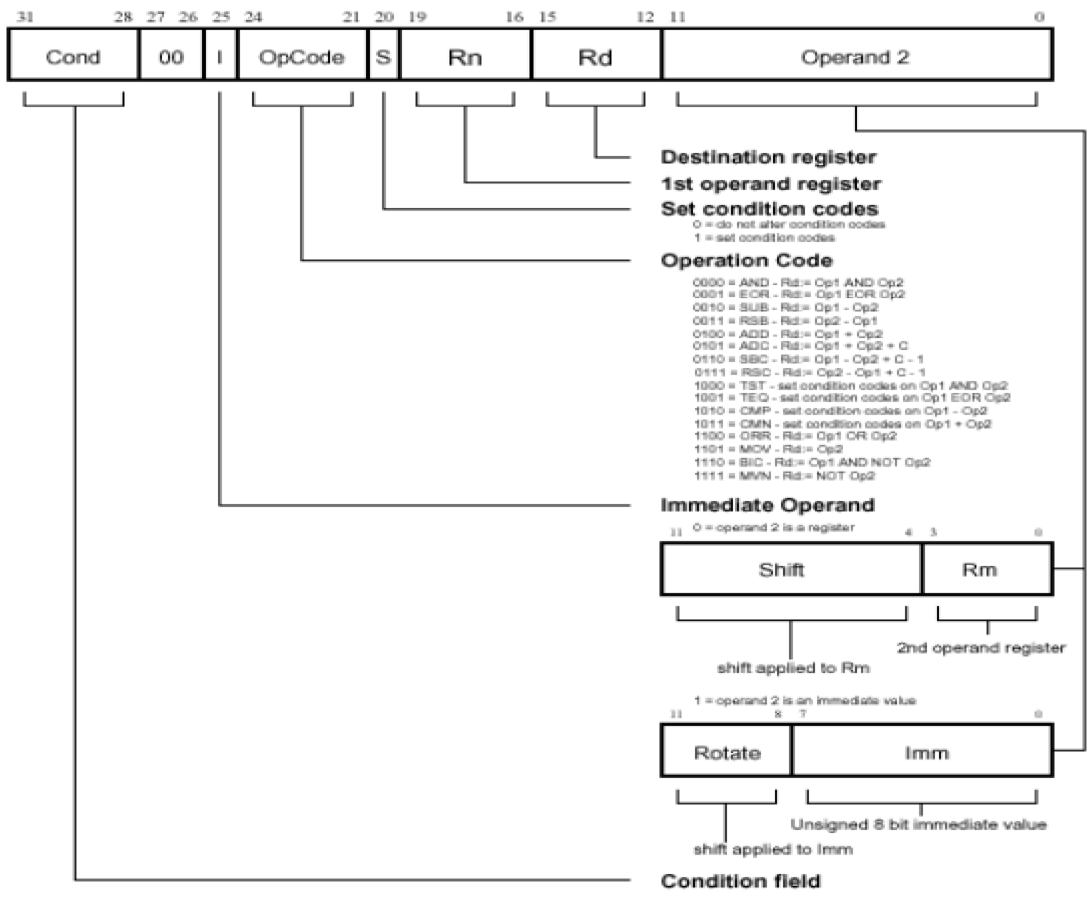


Figure 31 Branch Instructions

Data processing

The instruction is only executed if the condition is true. The instruction encoding is shown in Figure 32.The instruction produces a result by performing a specified arithmetic or logical operation on one or two operands. The first operand is always a register (Rn). The second operand may be a shifted register (Rm). The condition codes in the CPSR may be preserved or updated as a result of this instruction, according to the value of the S bit in the instruction. Certain operations (TST, TEQ, CMP, CMN) do not write the result to Rd. They are used only to perform tests and to set the condition codes on the result and always have the S bit set.



31 28 27 26 25 24 21 20 19 16 15 12 11 0
Cond
00
I
OpCode
S
Rn
Rd
Operand 2
Destination register
1st operand register
Set condition codes
0 = do not alter condition codes
1 = set condition codes
Operation Code
0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2
Immediate Operand
0 = operand 2 is a register
11 4 3 0
Shift
Rm
2nd operand register
shift applied to Rm
1 = operand 2 is an immediate value
11 8 7 0
Rotate
Imm
Unsigned 8 bit immediate value
shift applied to Imm
Condition field

Figure 32 Data Processing Instructions

Shifts

When the second operand is specified to be a shifted register, the Shift field in the instruction controls the operation of the barrel shifter. This field indicates the type of shift to be performed (logical left or right, arithmetic right or rotate right). The amount by which the register should be shifted may be contained in an immediate field in the instruction, or in the bottom byte of another register (other than R15). The encoding for the different shift types is shown in Figure 33.

Assembler Mnemonic	**OpCode**	**Action**
AND	0000	operand1 AND operand2
EOR	0001	operand1 EOR operand2
SUB	0010	operand1 - operand2
RSB	0011	operand2 - operand1
ADD	0100	operand1 + operand2
ADC	0101	operand1 + operand2 + carry
SBC	0110	operand1 - operand2 + carry - 1
RSC	0111	operand2 - operand1 + carry - 1
TST	1000	as AND, but result is not written
TEQ	1001	as EOR, but result is not written
CMP	1010	as SUB, but result is not written
CMN	1011	as ADD, but result is not written
ORR	1100	operand1 OR operand2
MOV	1101	operand2 (operand1 is ignored)
BIC	1110	operand1 AND NOT operand2 (Bit clear)
MVN	1111	NOT operand2 (operand1 is ignored)

Table 1 ARM Data Processing Instructions

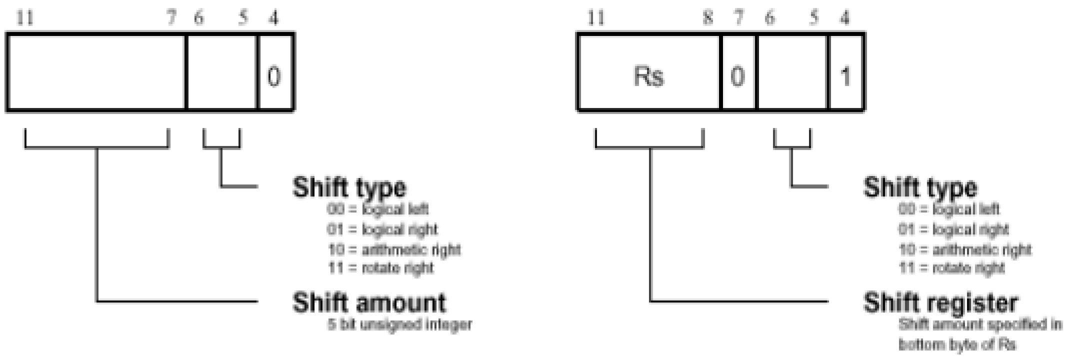


Figure 33 ARM Shift Operations

Immediate operand rotates

The immediate operand rotate field is a 4 bit unsigned integer which specifies a shift operation on the 8 bit immediate value. This value is zero extended to 32 bits, and then subject to a rotate right by twice the value in the rotate field. This enables many common constants to be generated, for example all powers of 2.

Writing to R15

When Rd is a register other than R15, the condition code flags in the CPSR may be updated from the ALU flags as described above.
When Rd is R15 and the S flag in the instruction is not set the result of the operation is placed in R15 and the CPSR is unaffected.
When Rd is R15 and the S flag is set the result of the operation is placed in R15 and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes that atomically restore both PC and CPSR. This form of instruction shall not be used in User mode.

Using R15 as an operand

If R15 (the PC) is used as an operand in a data processing instruction the register is used directly. The PC value will be the address of the instruction, plus 8 bytes due to instruction prefetching.

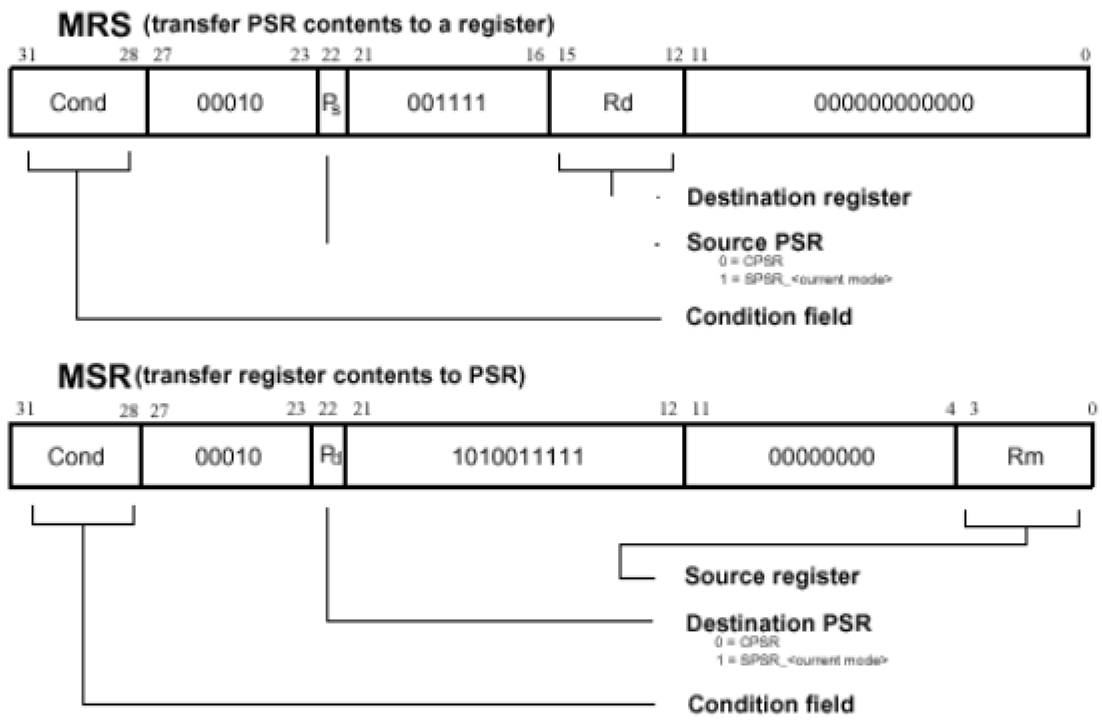


Figure 34 PSR Transfer

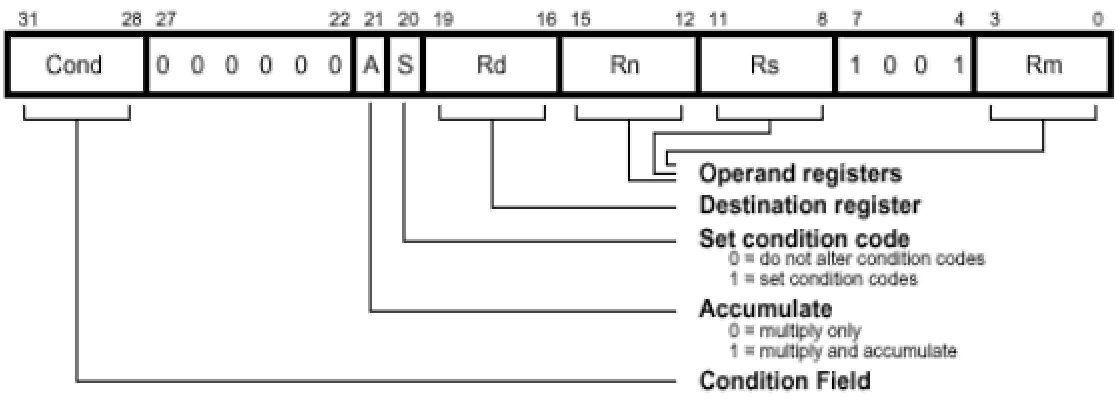
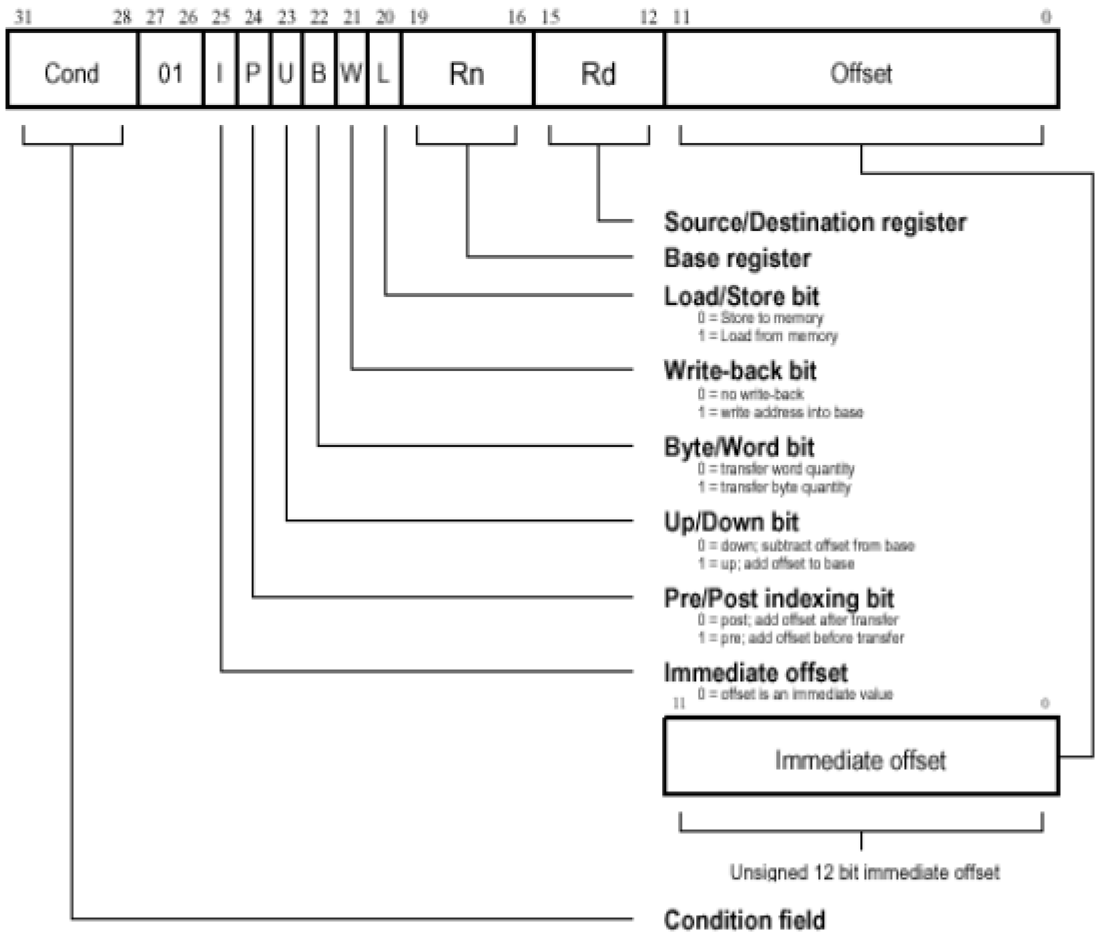


Figure 35 Multiply Instructions

1.13 Instruction Encoding-Data Transfer Instructions

Single data transfer (LDR, STR)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in Figure 36. The single data transfer instructions are used to load or store single bytes or words of data. The memory address used in the transfer is calculated by adding an offset to a base register.



31 28 27 26 25 24 23 22 21 20 19 16 15 12 11 0
Cond
01
I
P
U
B
W
L
Rn
Rd
Offset
Source/Destination register
Base register
Load/Store bit
0 = Store to memory
1 = Load from memory
Write-back bit
0 = no write-back
1 = write address into base
Byte/Word bit
0 = transfer word quantity
1 = transfer byte quantity
Up/Down bit
0 = down; subtract offset from base
1 = up; add offset to base
Pre/Post indexing bit
0 = post; add offset after transfer
1 = pre; add offset before transfer
Immediate offset
0 = offset is an immediate value
11 0
Immediate offset
Unsigned 12 bit immediate offset
Condition field

Figure 36 Single Data Transfer Instructions

Use of R15

When using R15 as the base register you must remember it contains address 8 bytes on from the address of the current instruction. When R15 is the source register (Rd) of a register store (STR) instruction, the stored value will be address of the instruction plus 8.

Block data transfer (LDM, STM)

The instruction is only executed if the condition is true. The instruction encoding is shown in Figure 37.

Block data transfer instructions are used to load (LDM) or store (STM) any subset of the currently visible registers. They support all possible stacking modes, maintaining full or empty stacks that can grow up or down memory, and are very efficient instructions for saving or restoring context, or for moving large blocks of data around main memory.

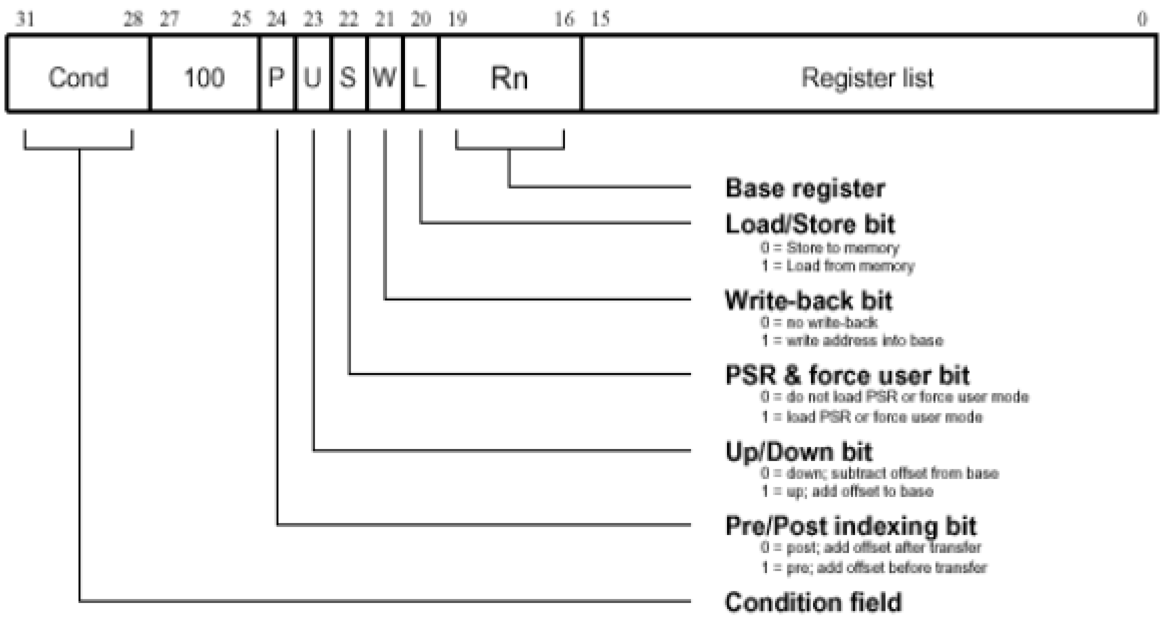

31
28
27
25
24
23
22
21
20
19
16
15
0
Cond
100
P
U
S
W
L
Rn
Register list
Base register
Load/Store bit
0 = Store to memory
1 = Load from memory
Write-back bit
0 = no write-back
1 = write address into base
PSR & force user bit
0 = do not load PSR or force user mode
1 = load PSR or force user mode
Up/Down bit
0 = down; subtract offset from base
1 = up; add offset to base
Pre/Post indexing bit
0 = post; add offset after transfer
1 = pre; add offset before transfer
Condition field

Figure 37 Block Data Transfer Instructions

Software interrupts (SWI)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in Figure 38.
The software interrupt instruction is used to enter Supervisor mode in a controlled manner. The instruction causes the software interrupt trap to be taken, which effects the mode change. The PC is then forced to a fixed value (0x08) and the CPSR is saved in SPSR_svc.

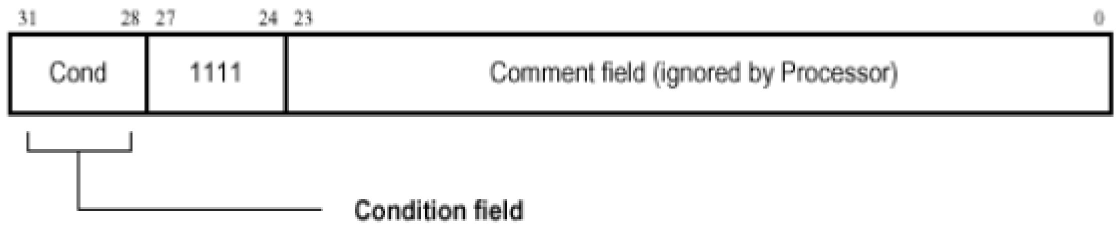


Figure 38 Software interrupt instruction