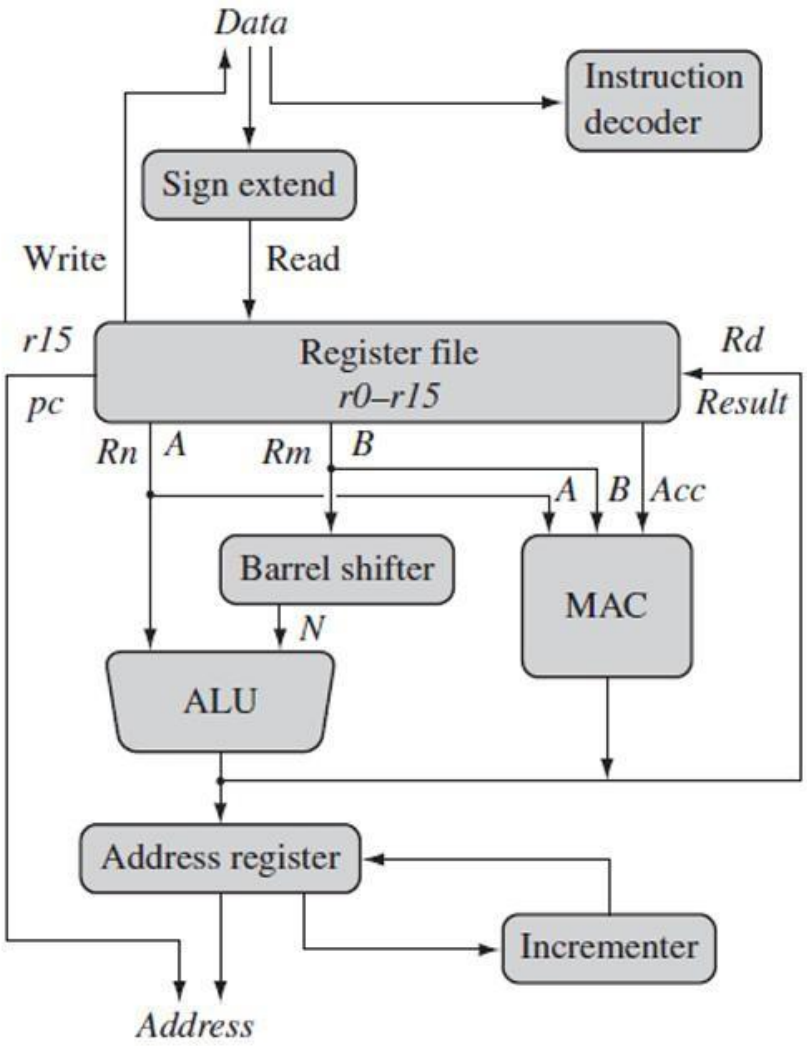
UNIT 4
ARM PROCESSOR



The arrows represent the flow of data, the lines represent the buses, and the boxes represent either an operation unit or a storage area.

- Data enters the processor core through the Data bus. The data may be an instruction to execute or a data item.
- Figure shows a Von Neumann implementation of the ARM—data items and instructions share the same bus. (In contrast, Harvard implementations of the ARM use two different buses).
- The instruction decoder translates instructions before they are executed. Each instruction executed belongs to a particular instruction set.

The ARM processor, like all RISC processors, uses load-store architecture—means it has two instruction types for transferring data in and out of the processor:

load instructions copy data from memory to registers in the core
store instructions copy data from registers to memory
 There are no data processing instructions that directly manipulate data in memory. Thus, data processing is carried out in registers.
 Data items are placed in the register file—a storage bank made up of 32-bit registers. Since the ARM core is a 32-bit processor, most instructions treat the registers as holding signed or unsigned 32-bit values. The sign extend hardware converts signed 8-bit and 16 bit numbers to 32-bit values as they are read from memory and placed in a register.
 ARM instructions typically have two source registers, Rn and Rm, and a single result or destination register, Rd. Source operands are read from the register file using the internal buses A and B, respectively.
 The ALU (arithmetic logic unit) or MAC (multiply-accumulate unit) takes the register values Rn and Rm from the A and B buses and computes a result. Data processing instructions write the result in Rd directly to the register file.
 Load and store instructions use the ALU to generate an address to be held in the address register and broadcast on the Address bus.
One important feature of the ARM is that register Rm alternatively can be preprocessed in the barrel shifter before it enters the ALU. Together the barrel shifter and ALU can calculate a wide range of expressions and addresses.
 After passing through the functional units, the result in Rd is written back to the register file using the Result bus.
 For load and store instructions the Incrementer updates the address register before the core reads or writes the next register value from or to the next sequential memory location.
 The processor continues executing instructions until an exception or interrupt Changes the normal execution flow.

REGISTERS:
General-purpose registers hold either data or an address. They are identified with the letter r prefixed to the register number. For example, register 4 is given the label r4.
The Figure shows the active registers available in user mode. (A protected mode is normally used when executing applications).
 The processor can operate in seven different modes.
 All the registers shown are 32 bits in size.

There are up to 18 active registers:
16 data registers and 2 processor status registers.
The data registers visible to the programmer are r0 to r15.
the ARM processor has three registers assigned to a particular task or special function: r13, r14, and r15. They are given with different labels to differentiate them from the other registers.
Register r13 is traditionally used as the stack pointer (sp) and stores the head of the stack in the current processor mode.

Register r14 is called the link register (lr) and is where the core puts the return address whenever it calls a subroutine.
Register r15 is the program counter (pc) and contains the address of the next instruction to be fetched by the processor.
In ARM state the registers r0 to r13 are orthogonal—any instruction that you can apply to r0 you can equally well apply to any of the other registers.
In addition to the 16 data registers, there are two program status registers: cpsr (current program status register) and spsr (saved program status register).

CURRENT PROGRAM STATUS REGISTER:

The ARM core uses the cpsr to monitor and control internal operations. The cpsr is a dedicated 32-bit register and resides in the register file. The following Figure shows the basic layout of a generic program status register. Note that the shaded parts are reserved for future expansion.

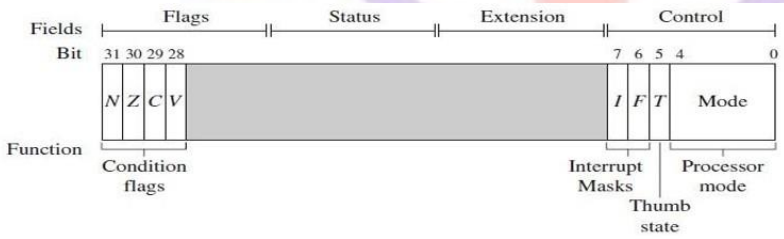


Figure: A Generic Program Status Register (psr)

The cpsr is divided into four fields, each 8 bits wide: flags, status, extension, and control. In current designs the extension and status fields are reserved for future use.

- The control field contains the processor mode, state, and interrupts mask bits.
- The flags field contains the condition flags.

Some ARM processor cores have extra bits allocated. For example, the J bit, which can be found in the flags field, is only available on Jazelle-enabled processors, which execute 8-bit instructions.
It is highly probable that future designs will assign extra bits for the monitoring and control of new features.

Processor Modes:

 The processor mode determines which registers are active and the access rights to the cpsr
register itself. Each processor mode is either privileged or non-privileged:
A privileged mode allows full read-write access to the cpsr.
A non-privileged mode only allows read access to the control field in the cpsr, but still allows read-write access to the condition flags.
 There are seven processor modes in total:
six privileged modes (abort, fast interrupt request, interrupt request, supervisor, system, and undefined)
 The processor enters abort mode when there is a failed attempt to access memory.
 Fast interrupt request and interrupt request modes correspond to the two interrupt levels available on the ARM processor.
 Supervisor mode is the mode that the processor is in after reset and is generally the mode that an operating system kernel operates in.
 System mode is a special version of user mode that allows full read-write access to the cpsr.
 Undefined mode is used when the processor encounters an instruction that is undefined or not supported by the implementation.
one non-privileged mode (user).
 User mode is used for programs and applications.

Banked Registers:

The following Figure shows all 37 registers in the register file.
 Of these, 20 registers are hidden from a program at different times.

These registers are called banked registers and are identified by the shading in the diagram.
They are available only when the processor is in a particular mode; for example, abort mode has banked registers r13_abt, r14_abt and spsr_abt.
 Banked registers of a particular mode are denoted by an underline character post-fixed to the mode mnemonic or _mode.
 Every processor mode except user mode can change mode by writing directly to the mode bits of the cpsr.
 All processor modes except system mode have a set of associated banked registers that are a subset of the main 16 registers.
 A banked register maps one-to-one onto a user mode register.
 If you change processor mode, a banked register from the new mode will replace an existing register.

For example, when the processor is in the interrupt request mode, the instructions you execute still access registers named r13 and r14. However, these registers are the banked registers r13_irq and r14_irq. The user mode registers r13_usr and r14_usr are not affected by the instruction referencing these registers. A program still has normal access to the other registers r0 to r12.

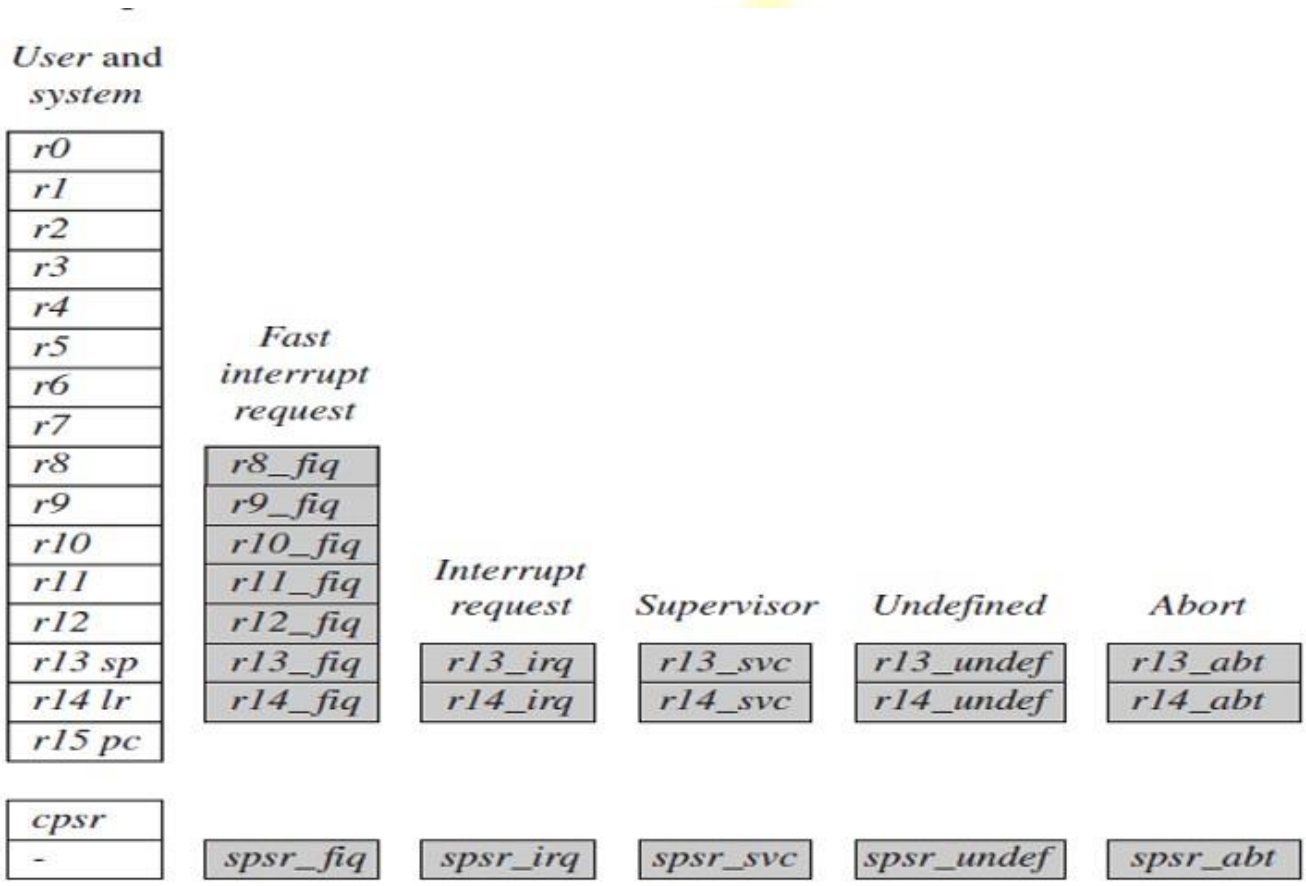


Figure: Complete ARM Register Set

The processor mode can be changed by a program that writes directly to the cpsr (the processor core has to be in privileged mode) or by hardware when the core responds to an exception or interrupt.

- The following exceptions and interrupts cause a mode change: reset, interrupt request, fast interrupt request, software interrupt, data abort, prefetch abort, and undefined instruction.
- Exceptions and interrupts suspend the normal execution of sequential instructions and jump to a specific location.
- The following Figure illustrates what happens when an interrupt forces a mode change.

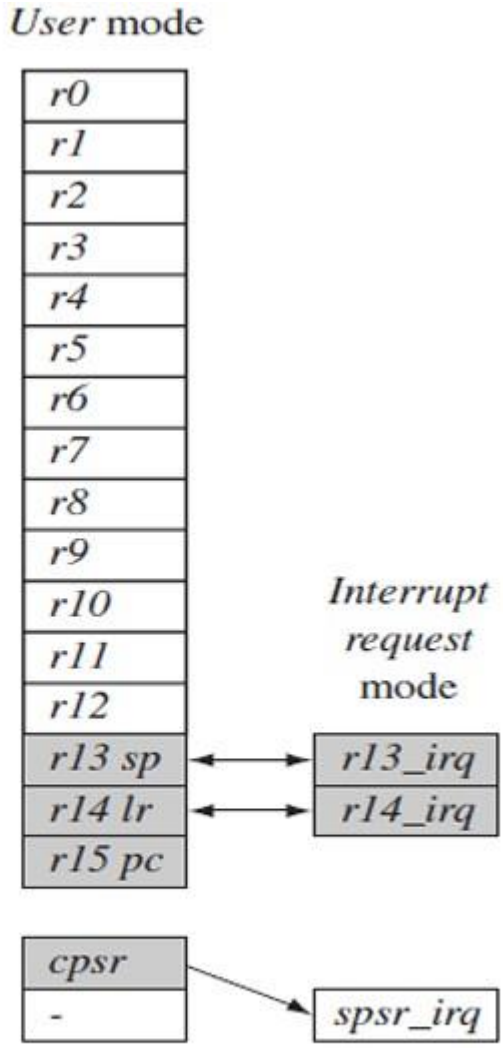


Figure: Changing Mode on an Exception

To return back to user mode, a special return instruction is used that instructs the core to restore the original cpsr from the spsr_irq and bank in the user registers r13 and r14.

 Note that, the spsr can only be modified and read in a privileged mode. There is no spsr available in user mode.

 Another important feature to note is that the cpsr is not copied into the spsr when a mode change is forced due to a program writing directly to the cpsr. The saving of the cpsr only occurs when an exception or interrupt is raised.

 When power is applied to the core, it starts in supervisor mode, which is privileged. Starting in a privileged mode is useful since initialization code can use full access to the

cpsr to set up the stacks for each of the other modes.

 The following Table lists the various modes and the associated binary patterns. The last column of the table gives the bit patterns that represent each of the processor modes in the cpsr

Table: Processor Mode

Mode	Abbreviation	Privileged	Mode[4:0]
Abort	abt	yes	10111
Fast Interrupt Request	fiq	yes	10001
Interrupt Request	irq	yes	10010
Supervisor	svc	yes	10011
System	sys	yes	11111
Undefined	und	yes	11011
User	usr	no	10000

State and Instruction Sets:

The state of the core determines which instruction set is being executed. There are three instruction sets:

- ARM
- Thumb
- Jazelle

.

 The ARM instruction set is only active when the processor is in ARM state.

 The Thumb instruction set is only active when the processor is in Thumb state. Once in Thumb state the processor is executing purely Thumb 16-bit instruction

You cannot inter-mingle sequential ARM, Thumb, and Jazelle instructions.

 The Jazelle J and Thumb T bits in the cpsr reflect the state of the processor.

When both J and T bits are 0, the processor is in ARM state and executes ARM instructions. This is the case when power is applied to the processor.

When the T bit is 1, then the processor is in Thumb state.

 To change states the core executes a specialized branch instruction.

The following Table compares the ARM and Thumb instruction set features.

Table: ARM and Thumb Instruction Set Features

-	ARM (*cspr T* = 0)	Thumb (*cspr T* = 1)
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution	most	only branch instructions
Data processing instructions	access to barrel shifter and ALU	separate barrel shifter and ALU instructions
Program status register	read-write in privileged mode	no direct access
Register usage	15 general-purpose registers +pc	8 general-purpose registers +7 high registers +pc

 The ARM designers introduced a third instruction set called Jazelle. Jazelle executes 8 bit instructions and is a hybrid mix of software and hardware designed to speed up the execution of Java byte-codes.

 To execute Java byte-codes, you require the Jazelle technology plus a specially modified version of the Java virtual machine.

The following Table gives the Jazelle instruction set features.

Table: Jazelle instruction set features

-	Jezelle (*cspr T* = 0, *J* – 1)
Instruction size	8-bit
Core Instructions	Over 60% of the Java byte-codes are implemented in hardware; the rest of the codes are implemented in software

Interrupt Masks:

 Interrupt masks are used to stop specific interrupt requests from interrupting the processor.

 There are two interrupt request levels available on the ARM processor core—

interrupt request (IRQ)

fast interrupt request (FIQ).

The cpsr has two interrupt mask bits, 7 and 6 (or I and F), which control the masking of IRQ and FIQ, respectively.

 The I bit masks IRQ when set to binary 1; and similarly, the F bit masks FIQ when set to binary

Condition Flags:

 Condition flags are updated by comparisons and the result of ALU operations that specify

the S instruction suffix.

For example, if a SUBS subtract instruction results in a register value of zero, then the Z Flag in the cpsr is set. This particular subtract instruction specifically updates the cpsr.With processor cores that include the DSP extensions, the Q bit indicates if an overflow or saturation has occurred in an enhanced DSP instruction. The flag is “sticky” in the sense that the hardware only sets this flag. To clear the flag you need to write to the cpsr directly.

 In Jazelle-enabled processors, the J bit reflects the state of the core; if it is set, the core is in Jazelle state. The J bit is not generally usable and is only available on some processor cores. To take advantage of Jazelle, extra software has to be licensed from both ARM Limited and Sun Microsystems.

 Most ARM instructions can be executed conditionally on the value of the condition flags.

The following Table lists the condition flags and a short description on what causes them to be set.

Table: Condition Flags

Flag	Flag Name	Set When
Q	Saturation	the result causes an overflow and/or saturation
V	oVerflow	the result causes a signed overflow
C	Carry	the result causes an unsigned carry
Z	Zero	the result is zero
N	Negative	bit 31 of the result is a binary 1

These flags are located in the most significant bits in the cpsr. These bits are used for conditional execution. The following Figure shows a typical value for the cpsr with both DSP extensions and Jazelle.

For the condition flags a capital letter shows that the flag has been set. For interrupts a capital letter shows that an interrupt is disabled.

Figure: Example:* cspr = *nzCvqjiFt_SVC

In the cpsr example shown in above Figure, the C flag is the only condition flag set.
The rest nzvq flags are all clear.

- The processor is in ARM state because neither the Jazelle j nor Thumb t bits are set.

The IRQ interrupts are enabled, and FIQ interrupts are disabled.

- Finally, you can see from the Figure, the processor is in supervisor (SVC) mode, since the mode[4:0] is equal to binary 10011

Conditional Execution:

- Conditional execution controls whether or not the core will execute an instruction.
- Prior to execution, the processor compares the condition attribute with the condition flags in the cpsr. If they match, then the instruction is executed; otherwise the instruction is ignored.
- The condition attribute is post-fixed to the instruction mnemonic, which is encoded into the instruction.
- The following Table lists the conditional execution code mnemonics. When a condition mnemonic is not present, the default behavior is to set it to always (AL) execute.

Table: Condition Mnemonics

Mnemonic	Name	Condition flags
EQ	equal	*Z*
NE	not equal	*z*
CS HS	carry set/unsigned higher or same	*C*
CC LO	carry clear/unsigned lower	*c*
MI	minus/negative	*N*
PL	plus/positive or zero	*n*
VS	overflow	*V*
VC	no overflow	*v*
HI	unsigned higher	*zC*
LS	unsigned lower or same	*Z* or *c*
GE	signed greater than or equal	*NV* or *nv*
LT	signed less than	*Nv* or *nV*
GT	signed greater than	*NzV* or *nzv*
LE	signed less than or equal	*Z* or *Nv* or *nV*
AL	always (unconditional)	ignored

PIPELINE:

 A pipeline is the mechanism in a RISC processor, which is used to execute instructions.

 Pipeline speeds up execution by fetching the next instruction while other instructions are being decoded and executed.

Figure: ARM7 Three-stage Pipeline

The above Figure shows a three-stage pipeline:

o Fetch loads an instruction from memory.

o Decode identifies the instruction to be executed.

o Execute processes the instruction and writes the result back to a register.

The following Figure illustrates pipeline using a simple example.

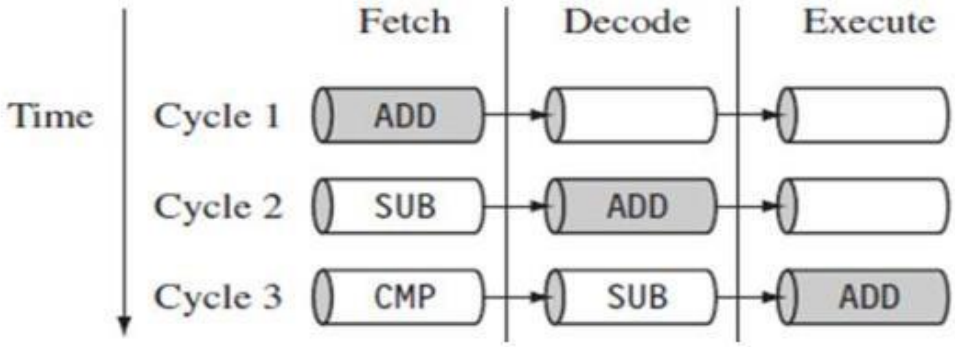


Figure: Pipelined Instruction Sequence

 The Figure shows a sequence of three instructions being fetched, decoded, and executed by the processor.

o The three instructions are placed into the pipeline sequentially.

o In the first cycle, the core fetches the ADD instruction from memory.

o In the second cycle, the core fetches the SUB instruction and decodes the ADD instruction.

o In the third cycle, both the SUB and ADD instructions are moved along the pipeline. The ADD instruction is executed, the SUB instruction is decoded, and The CMP instruction is fetched.

This procedure is called filling the pipeline.

 The pipeline allows the core to execute an instruction every cycle.

o As the pipeline length increases, the amount of work done at each stage is reduced, which allows the processor to attain a higher operating frequency. This in turn increases the performance.

o The increased pipeline length also means increased system latency and there can be data dependency between certain stages.

o The pipeline design for each ARM family differs. For example, The ARM9 core increases the pipeline length to five stages, as shown in Figure.

Figure: ARM9 Five-stage Pipeline

The ARM9 adds a memory and writeback stage, which allows the ARM9 to –

 process on average 1.1 Dhrystone MIPS per MHz

 increase the instruction throughput in ARM9 by around 13% compared with an ARM7.

Table:ARM Instruction Set

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

Mnemonics	ARM ISA	Description
MCR MCR2 MCRR	v2 v5 v5E	move to coprocessor from an ARM register or registers
MLA	v2	multiply and accumulate 32-bit values
MOV	v1	move a 32-bit value into a register
MRC MRC2 MRRC	v2 v5 v5E	move to ARM register or registers from a coprocessor
MRS	v3	move to ARM register from a status register (*cpsr* or *spsr*)
MSR	v3	move to a status register (*cpsr* or *spsr*) from an ARM register
MUL	v2	multiply two 32-bit values
MVN	v1	move the logical NOT of 32-bit value into a register

Mnemonics	ARM ISA	Description
ORR	v1	logical bitwise OR of two 32-bit values
PLD	v5E	preload hint instruction
QADD	v5E	signed saturated 32-bit add
QDADD	v5E	signed saturated double and 32-bit add
QDSUB	v5E	signed saturated double and 32-bit subtract
QSUB	v5E	signed saturated 32-bit subtract
RSB	v1	reverse subtract of two 32-bit values
RSC	v1	reverse subtract with carry of two 32-bit integers
SBC	v1	subtract with carry of two 32-bit values
SMLA*xy*	v5E	signed multiply accumulate instructions ($(16 \times 16) + 32 = 32$-bit)
SMLAL	v3M	signed multiply accumulate long ($(32 \times 32) + 64 = 64$-bit)
SMLAL*xy*	v5E	signed multiply accumulate long ($(16 \times 16) + 64 = 64$-bit)
SMLAW*y*	v5E	signed multiply accumulate instruction ($((32 \times 16) \gg 16) + 32 = 32$-bit)
SMULL	v3M	signed multiply long ($32 \times 32 = 64$-bit)

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

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,theADD instruction

below adds the two values store din registers r1and r2(the source registers).It writes the result to register r3 (the destination register).

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

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

DATA PROCESSING INSTRUCTIONS:

The data processing instructions manipulated within registers.They are—
 move instructions, arithmetic instructions, logical instructions, comparison instructions, and multiply instructions.
Most data pocessing 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.
o The C flag is set from the result of the barrel shift as the last bit shifted out.
o The N flag is set to bit 31of the result.
o 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

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	$Rd = \sim N$

This instruction is useful for setting initial values and transferring data between registers.
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 with in 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 with in the cycle time of the instruction.

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

o This is particularly useful for loading constants into a register and achieving fast multiplies or division by a power of 2.

 There are data processing Instructions that donot use the barrelshift, forexample, the MUL (multiply), CLZ (count leading zeros), and QADD (signed saturated 32-bit add)instructions.

Figure:BarrelShifterand

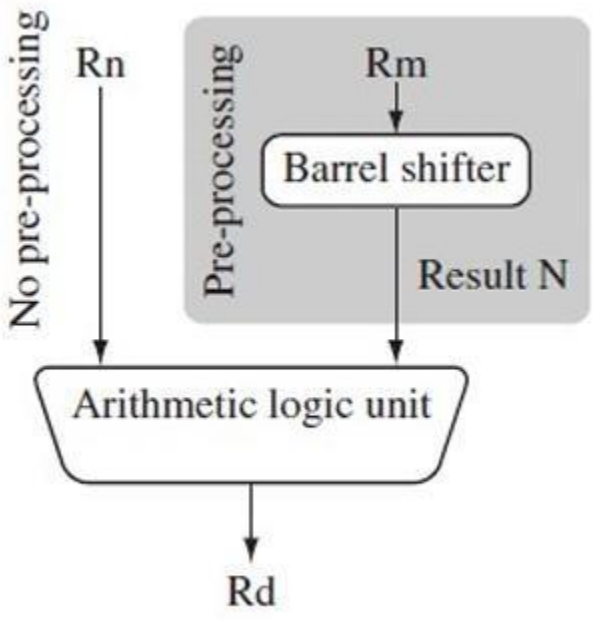


Figure:BarrelShifterand ALU

Figure shows the data flow between the ALU and the barrel shifter.

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

 We apply a logical shift left (LSL) to register Rm before moving it to the destination register.This is the same as applying the standard C language shift operator « to the register.

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

Table:Barrel Shifter Operations

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

X represents the register being shifted and y represents shift amount

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

```
PRE r5=5
r7 =8
MOV r7,r5, LSL#2
;let r7=r5*4 =(r5 <<2)
POSTr5=5
r7 =20
```

The above example multiplies register r5 by four and then places the result in to register r7.
The following Figure illustrates a logical shift left by one.

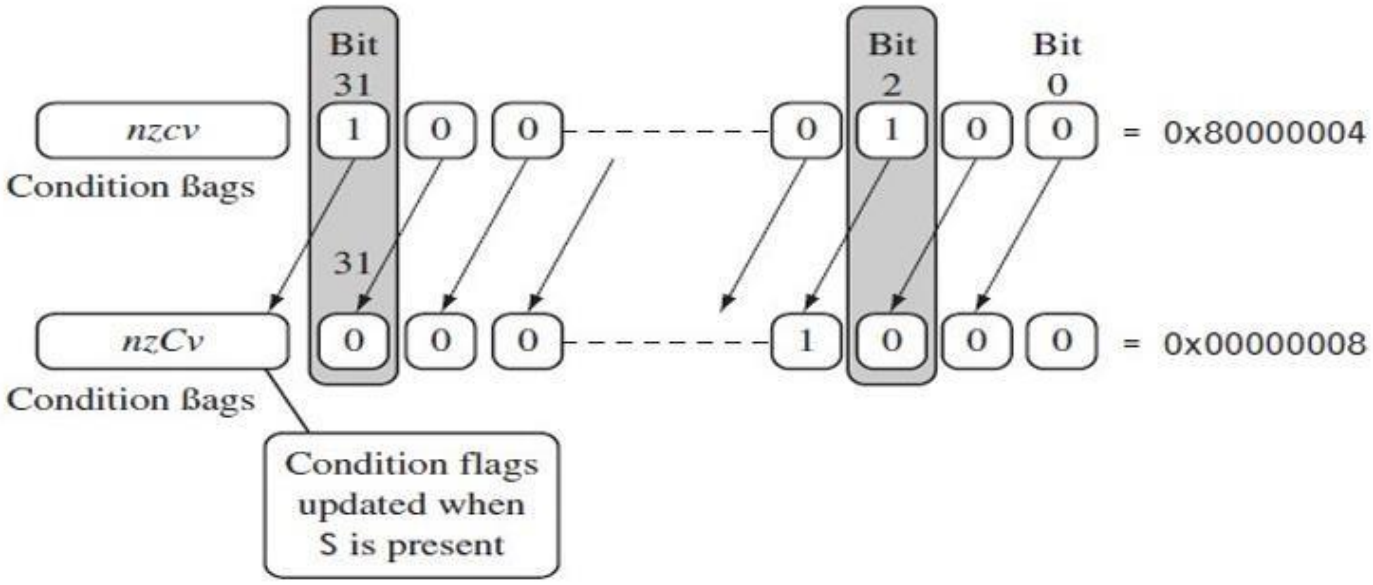


Figure:Logical Shift Left by One

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 r1left by one bit.This multiplies register r1 by a value 21.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:BarrelShifterOperationSyntaxfordataProcessingInstructions

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

Arithmetic Instructions:

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

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

ADC	add two 32-bit values and carry	$Rd = Rn + N +$ carry
ADD	add two 32-bit values	$Rd = Rn + N$
RSB	reverse subtract of two 32-bit values	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-bit values	$Rd = N - Rn -$!(carry flag)
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$!(carry flag)
SUB	subtract two 32-bit values	$Rd = Rn - N$

N is the result of the shifter operation.

Example:The following simple subtract instruction subtracts avalue stored in register r2from a value stored in register r1. The result is stored in register r0.
PRE r0=0x00000000
r1=0x00000002
r2 =0x00000001
SUB r0,r1,r2
POST r0 =0x00000001

Example: The following reverse subtract instruction (RSB) subtracts r1 from the constant value#0, writing the result to r0. You can use this instruction to negate numbers.
PRE r0 =0x00000000
r1 =0x00000077
RSB r0, r1,#0 ;Rd=0x0-r1
POST r0=-r1=0xffffff89
Example: The SUBS instruction is useful for decrementing loop counters.In this example, we subtract the immediate value one from the value one stored in register r1. The result value zero is written to register r1. The cpsr is updated with the ZC flags being set.
PRE cpsr= nzcvqiFt_USER
r1 = 0x00000001
SUBS r1,r1, #1
POST cpsr= nZCvqiFt_USER
r1 = 0x00000000

Logical Instructions:

Logical instructions perform bitwise logical operations on the two source registers.
Example:

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

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

Example: This example shows a logical OR operation between registers r1 and r2. Register r0 holds the result.
PRE r0 =0x00000000
r1=0x02040608
r2 =0x10305070
ORR r0,r1,r2
POSTr0=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=RnANDNOT(N)
In this example, register r2 contains a binary pattern where every binary1 in r2clears 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 theS suffix is present. These instructions can use barrel-shif ted 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 isrepresented 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 resultd is carded;similarly the TS T 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. Itisimportanttounderstandthatcomparisoninstructionsonlymodifytheconditionflagsofthe Cpsr and donot 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 on to a pair of registers representing a 64-bit value.The final resultis 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 in to 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
MULr0,r1, r2 ;r0 =r1*r2
POSTr0=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
UMULLr0,r1,r2,r3 ;[r1,r0]=r2*r3
POSTr0 = 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.

```
Syntax: B{<cond>} label
        BL{<cond>} label
        BX{<cond>} Rm
        BLX{<cond>} label | Rm
```

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

BL subroutine ; branch to subroutine
CMP r1, #5 ; compare r1 with 5
MOVEQ r1, #0 ;if(r1==5)thenr1=0
Subroutine:
<subroutinecode>
MOVpc,lr ;returnbymovingpc=lr

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.

LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*

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

LDR and STR instructions can load and store data on a boundary alignment that is the same as the data type size being loaded or stored.

o 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

Single-Register Load-Store Addressing Modes:

The ARM instruction set provides different modes for addressing memory. These modes incorporate one of the indexing methods: pre index with write back, pre index, and post index.

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.

Pre index with write back calculates an address from a base register plus address offset and then updates that address base register with the new address.
Pre index offset is the same as the pre index with write back but does not update the address base register.
o The pre index mode is useful for accessing an element in a data structure.
Post index only updates the address base register after the address is used.
o Thepostindexandpreindexwithwritebackmodesareusefulfortraversingan array.

EXCEPTIONS, INTERRUPTS AND THE VECTOR TABLE:

When an exception or interrupt occurs, the processor sets the pc to a specific memory address. The address is within a special address range called the vector table.
The entries in the vector table are instructions that branch to specific routines designed to handle a particular exception or interrupt.
The memory map address 0x00000000 (or in some processors starting at the offset 0xffff0000) is reserved for the vector table, a set of 32-bit words.

When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the exception vector table (see the following Table).

TabTable: The Vector Table

Exception/Interrupt	Shorthand	Address	High Address
Reset	RESET	0x00000000	0x00000000
Undefined instruction	UNDEF	0x00000004	0xffff0004
Software interrupt	SWI	0x00000008	0xffff0008
Prefetch abort	PABT	0x0000000c	0xffff000c
Data abort	SABT	0x00000010	0xffff0010
Reserved	–	0x00000014	0xffff0014
Interrupt request	IRQ	0x00000018	0xffff0018
Fast interrupt request	FIQ	0x0000001c	0xffff001c

Each vector table entry contains a form of branch instruction pointing to the start of a specific routine:
Reset vector is the location of the first instruction executed by the processor when

power is applied. This instruction branches to the initialization code.
Undefined instruction vector is used when the processor cannot decode an instruction.
Software interrupt vector is called when you execute a SWI instruction. The SWI instruction is frequently used as the mechanism to invoke an operating system routine.
Prefetch abort vector occurs when the processor attempts to fetch an instruction from an address without the correct access permissions. The actual abort occurs in the decode stage.
Data abort vector is similar to a prefetch abort, but is raised when an instruction attempts to access data memory without the correct access permissions.
Interrupt request vector is used by external hardware to interrupt the normal execution flow of the processor. It can only be raised if IRQs are not masked in the *cpsr*.
Fast interrupt request vector is similar to the interrupt request, but is reserved for hardware requiring faster response times. It can only be raised if FIQs are not masked in the *cpsr*.

LOADING CONSTANTS:

You might have noticed that there is no ARM instruction to move a 32-bit constant into a register. Since ARM instructions are 32 bits in size, they obviously cannot specify a general 32-bit constant. To aid programming there are two pseudo-instructions to move a 32-bit value into a register.

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

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

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

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

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

This example involves a memory access to load the constant, which can be expensive for timecritical 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 pcrelative 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.