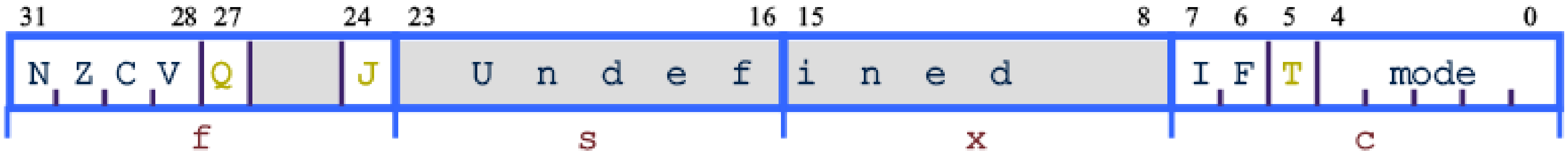
ARM Processor – Instruction Set

Lecture on ARM7 Instruction set

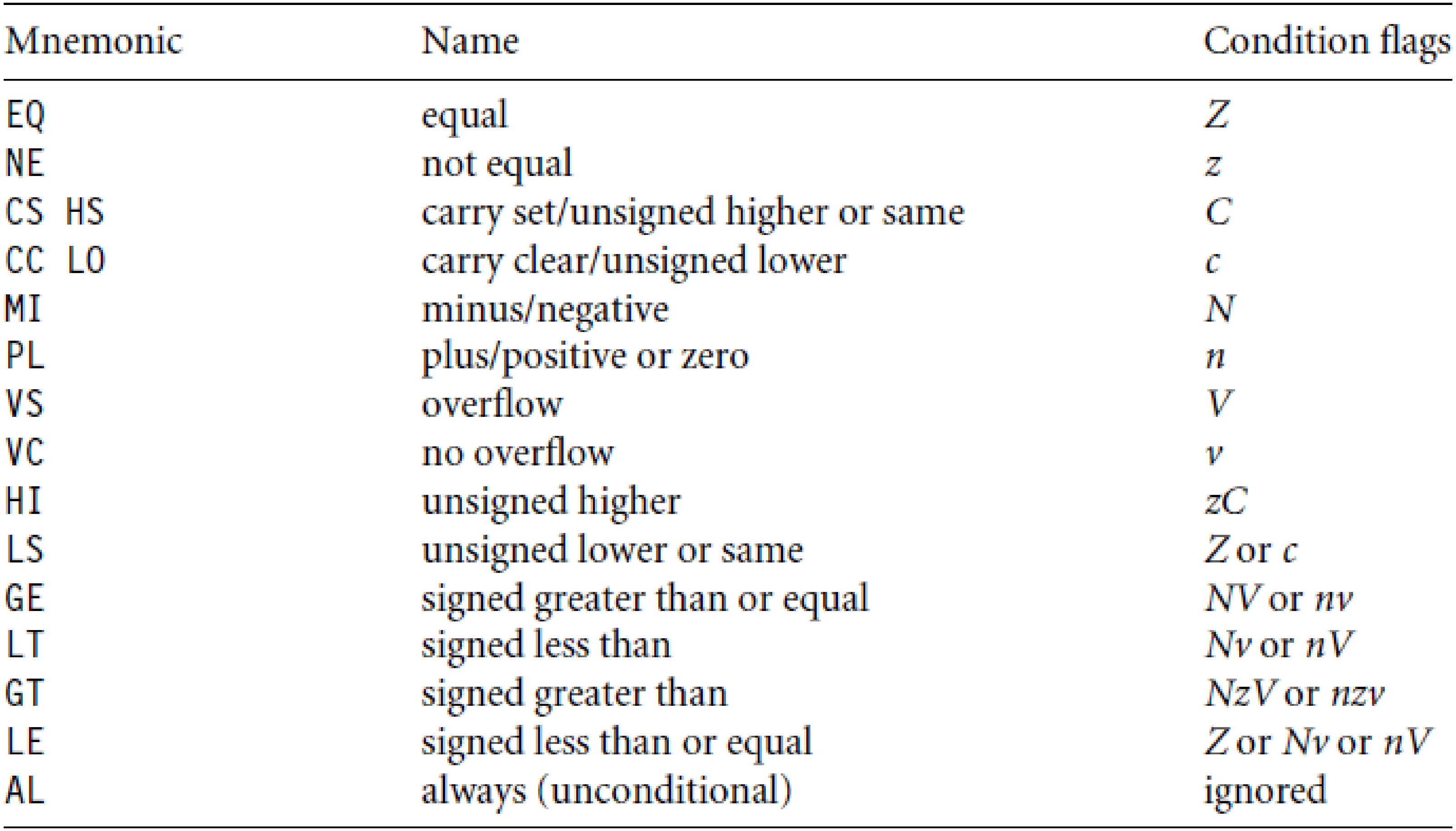
By
Harish V. Mekali
Assistant Professor, Dept. of ECE
BMSCE, Bangalore - 19

Register Organization - CPSR



- Condition code flags
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation o Verflowed
- Sticky Overflow flag - Q flag
  - Architecture 5TE/J only
  - Indicates if saturation has occurred
- J bit
  - Architecture 5TEJ only
  - J = 1: Processor in Jazelle state
- Interrupt Disable bits.
  - I = 1: Disables the IRQ.
  - F = 1: Disables the FIQ.
- T Bit
  - Architecture xT only
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
- Mode bits
  - Specify the processor mode

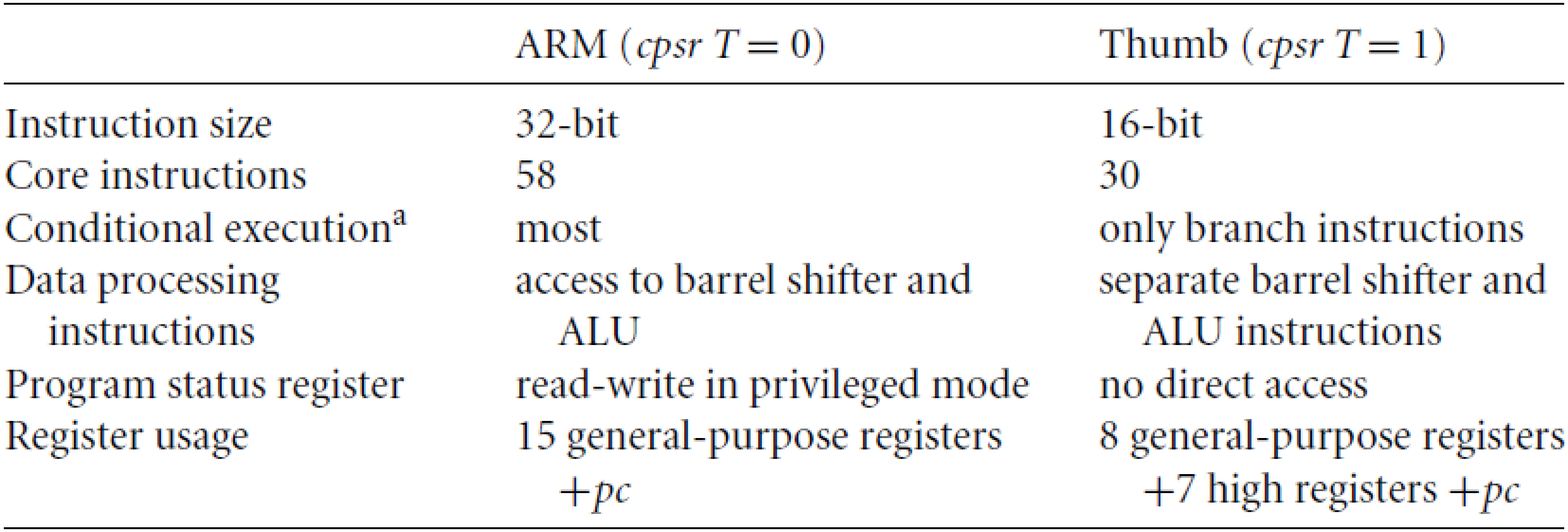
Condition Mnemonics



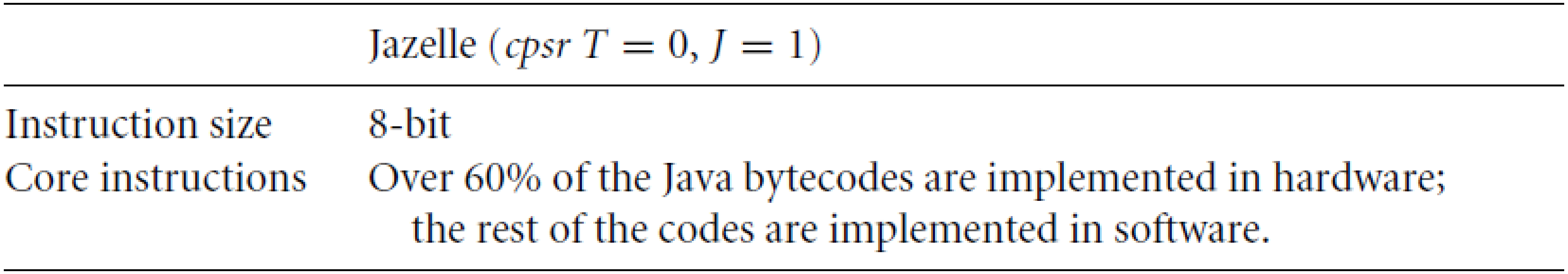
Mnemonic	Name	Condition flags
`EQ`	equal	*Z*
`NE`	not equal	*z*
`CS HS`	carry set/unsigned higher or same	*C*
`CC LO`	carry clear/unsigned lower	*c*
`MI`	minus/negative	*N*
`PL`	plus/positive or zero	*n*
`VS`	overflow	*V*
`VC`	no overflow	*v*
`HI`	unsigned higher	*zC*
`LS`	unsigned lower or same	*Z* 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

Fig : Condition mnemonics that can be added to basic instructions

Instruction Set & Processor State



	ARM (*cpsr* $T = 0$)	Thumb (*cpsr* $T = 1$)
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution[a]	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*



	Jazelle (*cpsr* $T = 0, J = 1$)
Instruction size	8-bit
Core instructions	Over 60% of the Java bytecodes are implemented in hardware; the rest of the codes are implemented in software.

Instruction set

Classification of instructions

1. Data Processing Instruction
   - Data movement , Arithmetic, Logical, Comparison, Multiply
2. Branch instructions
3. Load – store instructions
   - Single register, Multiple register, Stack operations, Swap
4. Software interrupt instructions
5. Program Status Register instructions

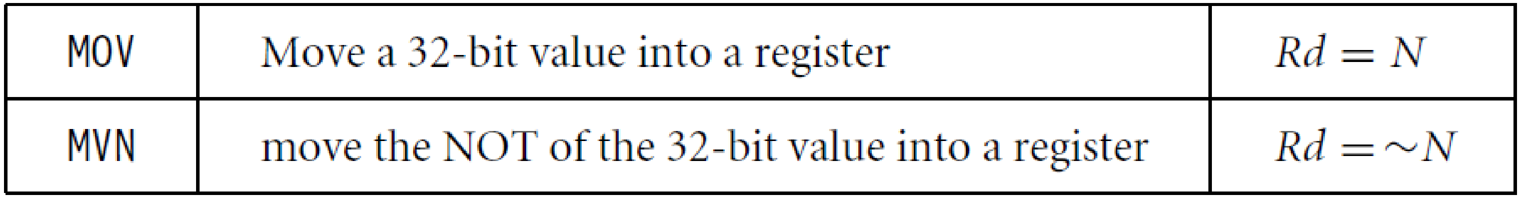
Data Processing Instructions

- Most of the data processing instructions can process one operand using barrel shifter

Data movement instructions

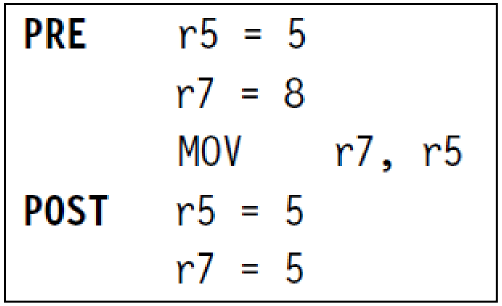
- **“S”** suffix with MOVE operations can update the C, Z, N flags in CPSR

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

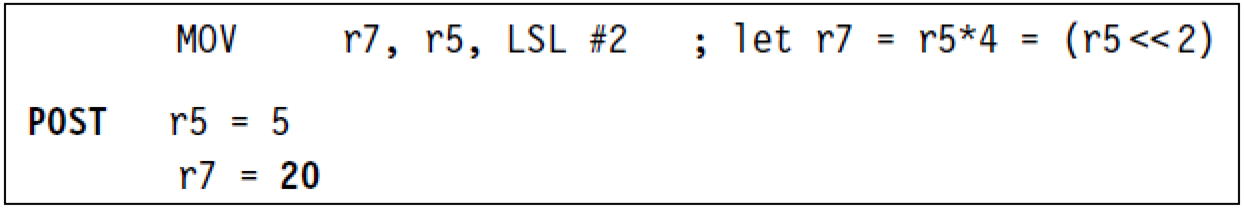


MOV	Move a 32-bit value into a register	$Rd = N$
MVN	move the NOT of the 32-bit value into a register	$Rd = \sim N$

- **“N”** can be register or immediate data
- Example : MOV instruction without and with barrel shifter operations

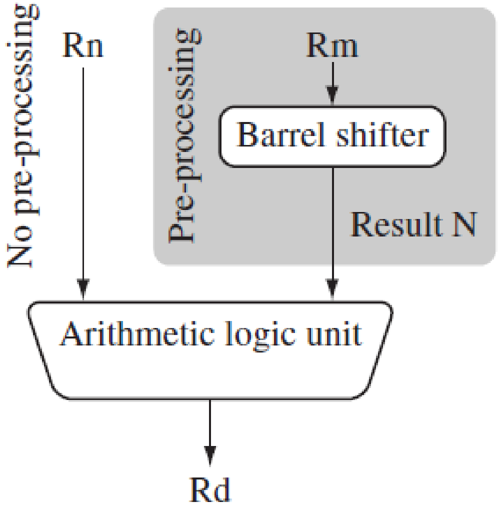


```
PRE     r5 = 5
        r7 = 8
        MOV     r7, r5
POST    r5 = 5
        r7 = 5
```



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

Barrel Shifter Operations



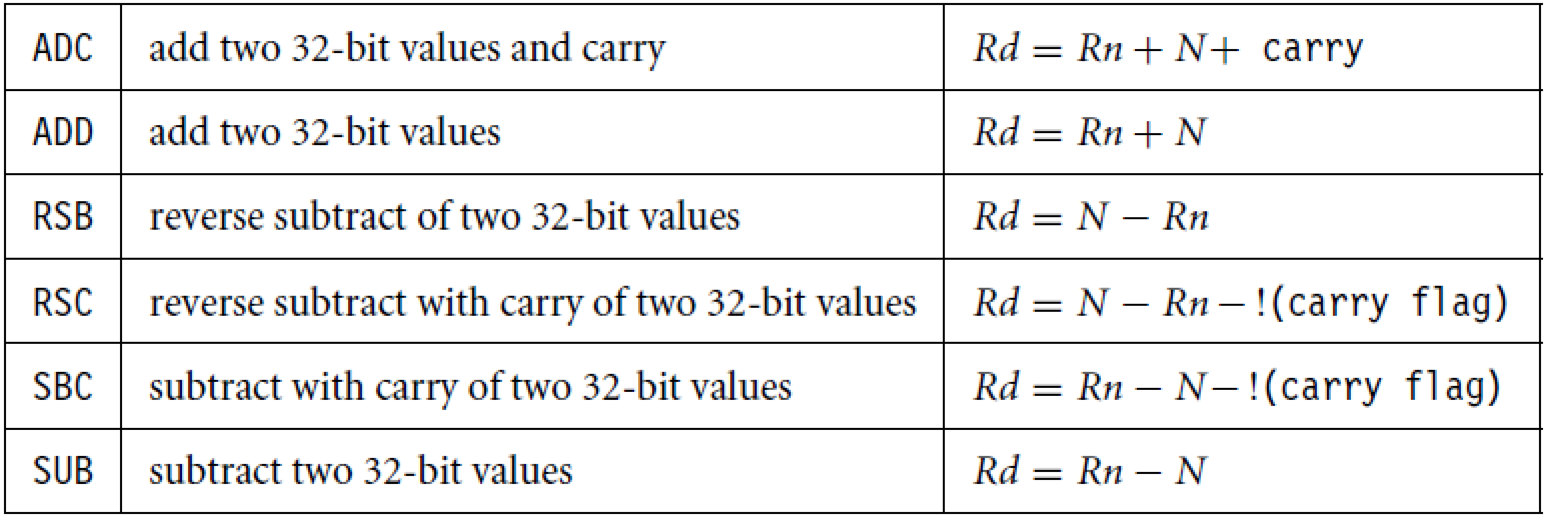
N shift operations	Syntax
Immediate	`#immediate`
Register	`Rm`
Logical shift left by immediate	`Rm, LSL #shift_imm`
Logical shift left by register	`Rm, LSL Rs`
Logical shift right by immediate	`Rm, LSR #shift_imm`
Logical shift right with register	`Rm, LSR Rs`
Arithmetic shift right by immediate	`Rm,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

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

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

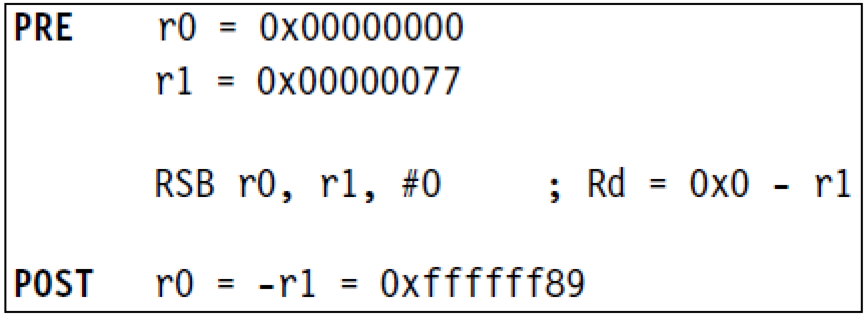
Arithmetic Instructions

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

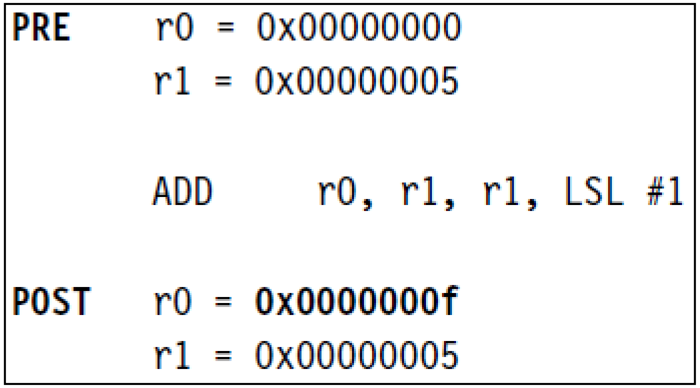


ADC	add two 32-bit values and carry	$Rd = Rn + N +$ carry
ADD	add two 32-bit values	$Rd = Rn + N$
RSB	reverse subtract of two 32-bit values	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-bit values	$Rd = N - Rn -$!(carry flag)
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$!(carry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

- **“N”** is the result of shift operation
- Example :



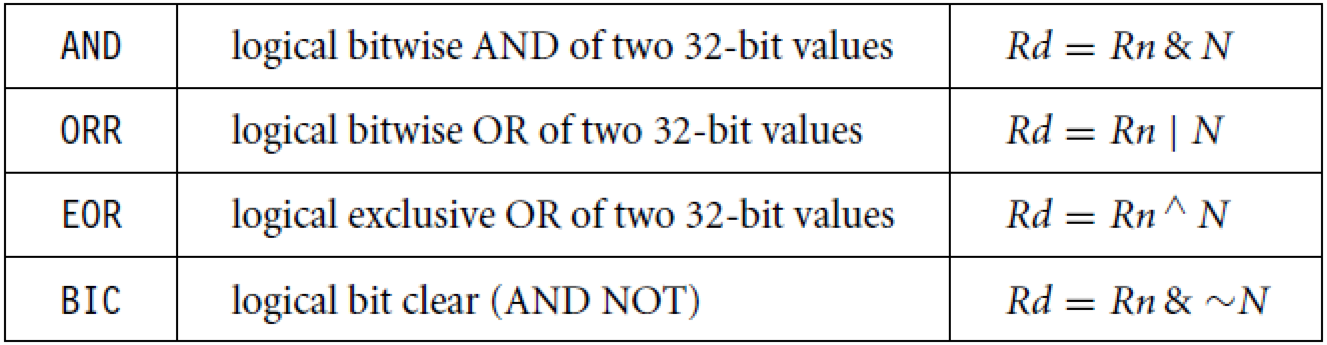
```
PRE     r0 = 0x00000000
        r1 = 0x00000077

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

POST    r0 = -r1 = 0xffffff89
```



```
PRE     r0 = 0x00000000
        r1 = 0x00000005

        ADD     r0, r1, r1, LSL #1

POST    r0 = 0x0000000f
        r1 = 0x00000005
```

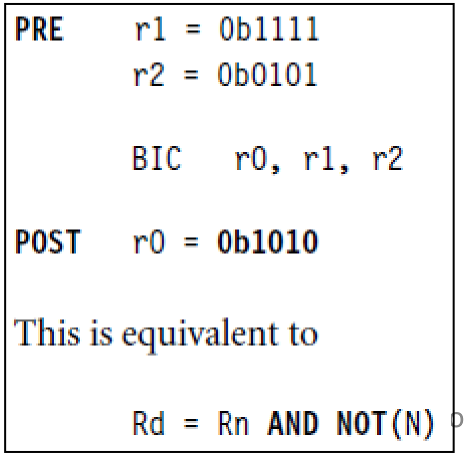
Logical Instructions

```
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)	$Rd = Rn \& \sim N$

- Updates CPSR only if suffix “S” is added and logical instructions can also use barrel shifter in the same way as MOVE and arithmetic
- Example :



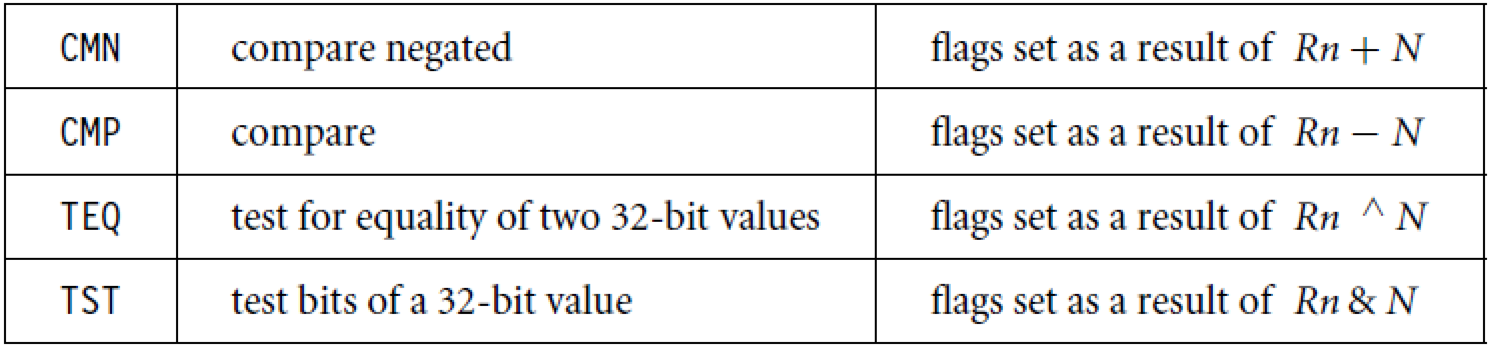
```
PRE     r1 = 0b1111
        r2 = 0b0101

        BIC   r0, r1, r2

POST    r0 = 0b1010
```

This is equivalent to

```
        Rd = Rn AND NOT(N)
```

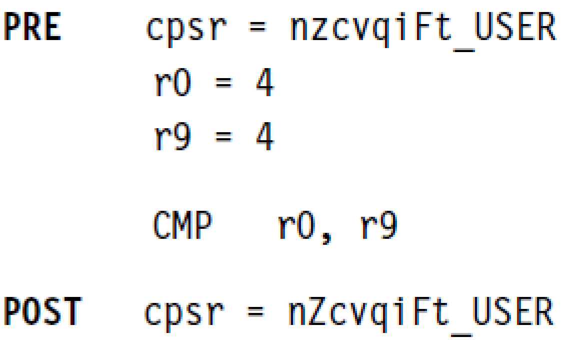
Comparison Instructions

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



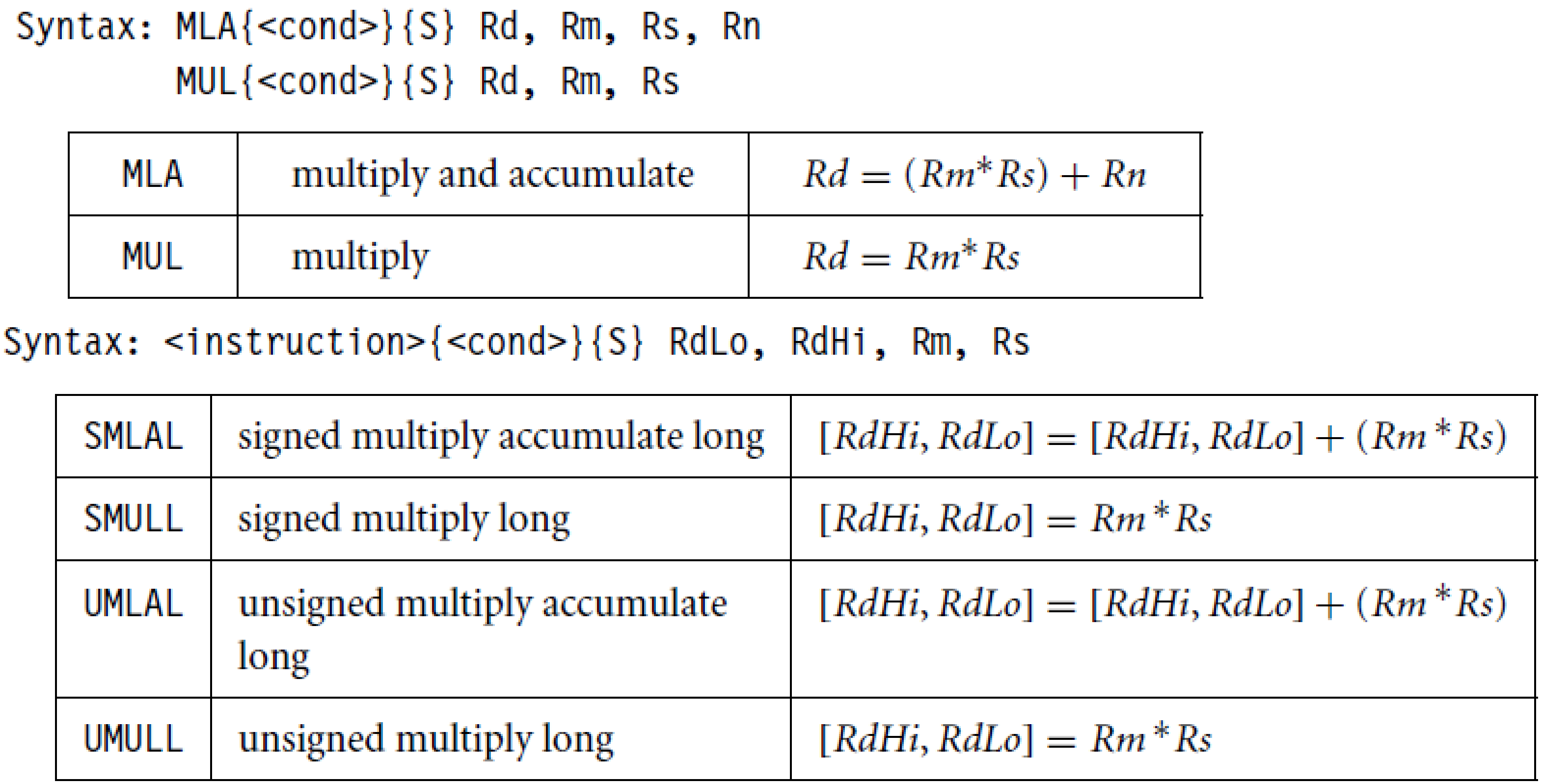
CMN	compare negated	flags set as a result of $Rn + N$
CMP	compare	flags set as a result of $Rn - N$
TEQ	test for equality of two 32-bit values	flags set as a result of $Rn \wedge N$
TST	test bits of a 32-bit value	flags set as a result of $Rn \& N$

- Updates CPSR only without affecting the register content and this can be used in conditional execution
- TST is logical AND and TEQ is a logical XOR operation
- Example :



```
PRE     cpsr = nzcvqiFt_USER
        r0 = 4
        r9 = 4

        CMP    r0, r9

POST    cpsr = nZcvqiFt_USER
```

Multiply Instructions



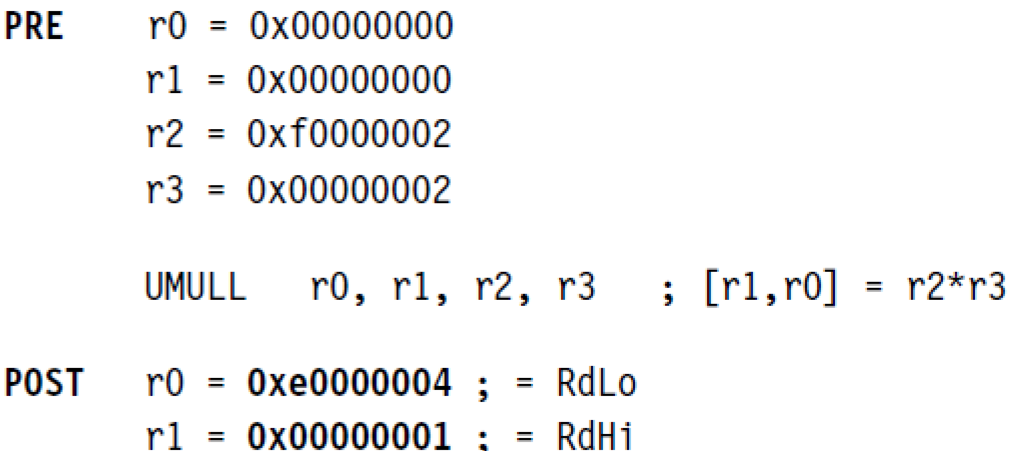
```
Syntax: MLA{<cond>}{S} Rd, Rm, Rs, Rn
        MUL{<cond>}{S} Rd, Rm, Rs
```

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

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

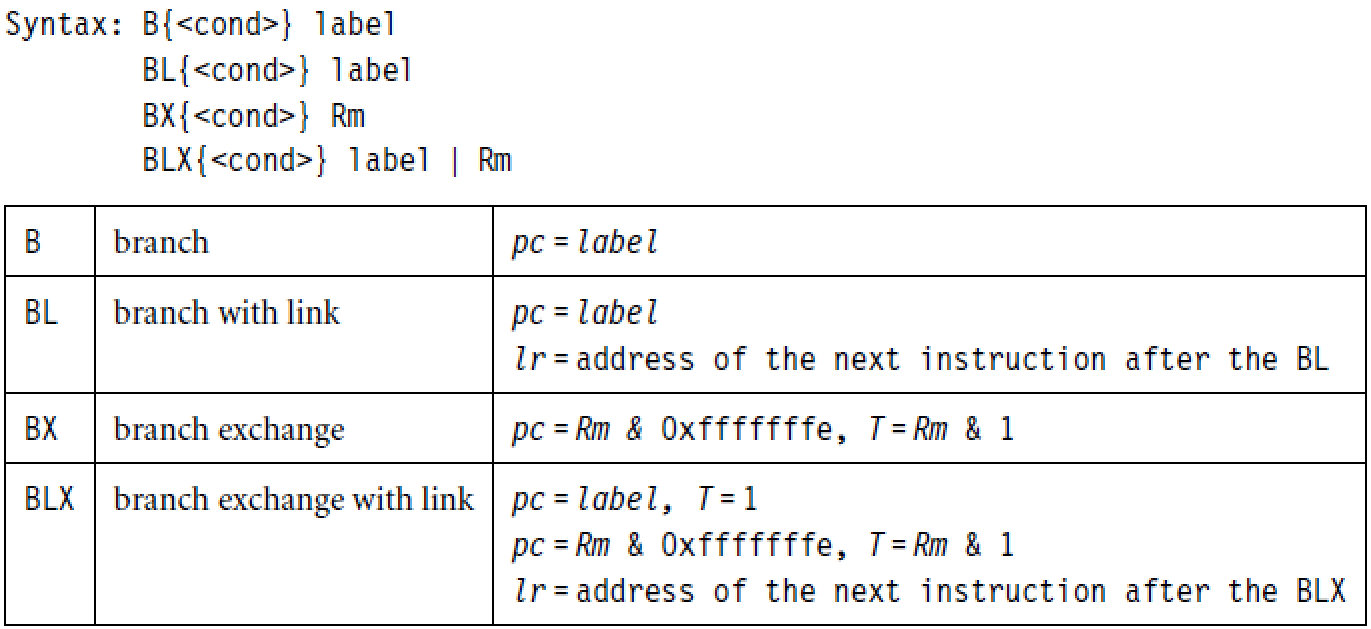
SMLAL	signed multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
SMULL	signed multiply long	$[RdHi, RdLo] = Rm * Rs$
UMLAL	unsigned multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
UMULL	unsigned multiply long	$[RdHi, RdLo] = Rm * Rs$

- Example :



```
PRE     r0 = 0x00000000
        r1 = 0x00000000
        r2 = 0xf0000002
        r3 = 0x00000002

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

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

Branch Instructions



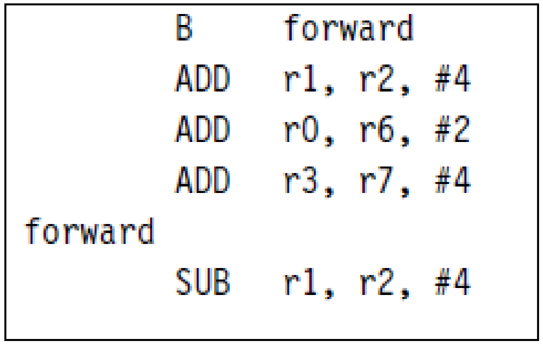
```
Syntax: B{<cond>} label
        BL{<cond>} label
        BX{<cond>} Rm
        BLX{<cond>} label | Rm
```

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

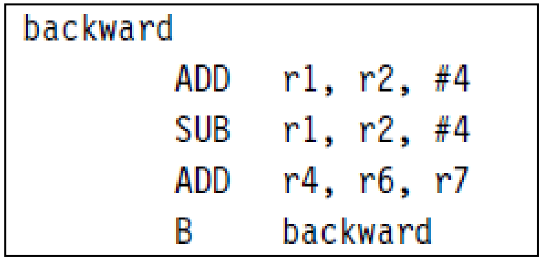
- **Address label** is stored with instruction as a PC relative offset and it must be approximately 32 MB
- **BX** and **BLX** are primarily used for branch to and from Thumb code . “**T**” bit in CPSR is updated by least significant bit of branch register

Branch Instructions

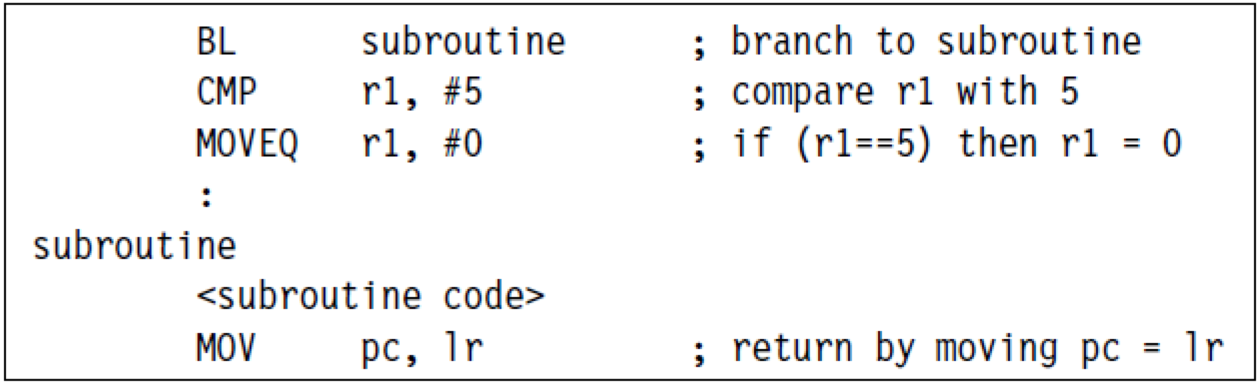
- Example :



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

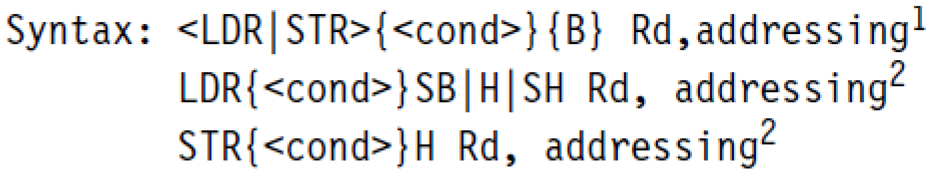


```
backward
        ADD     r1, r2, #4
        SUB     r1, r2, #4
        ADD     r4, r6, r7
        B       backward
```

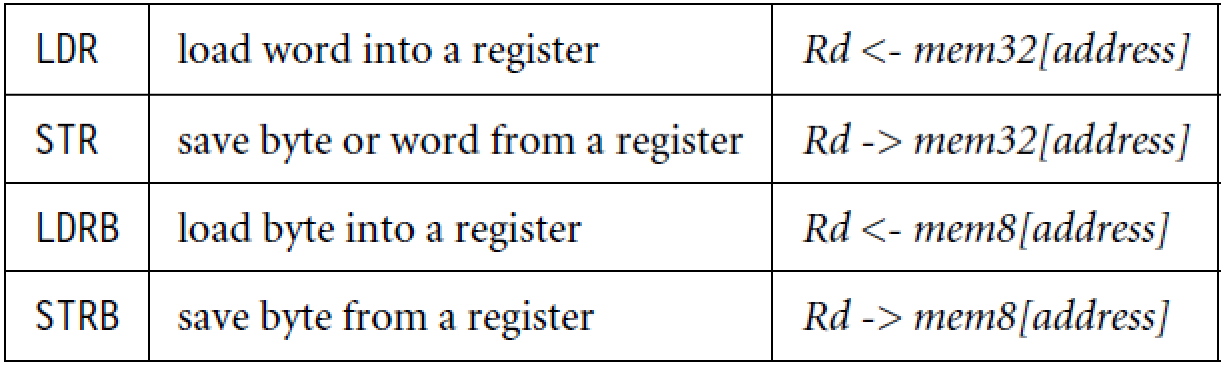


```
        BL      subroutine      ; branch to subroutine
        CMP     r1, #5          ; compare r1 with 5
        MOVEQ   r1, #0          ; if (r1==5) then r1 = 0
        :
subroutine
        <subroutine code>
        MOV     pc, lr          ; return by moving pc = lr
```

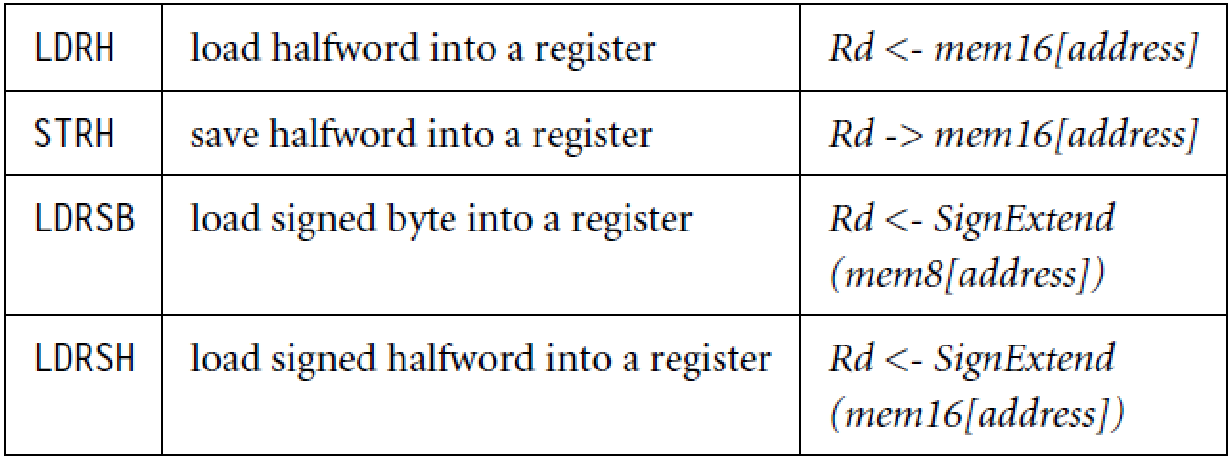
Load and Store Instructions



```
Syntax: <LDR|STR>{<cond>}{B} Rd,addressing[1]
        LDR{<cond>}SB|H|SH Rd, addressing[2]
        STR{<cond>}H Rd, addressing[2]
```

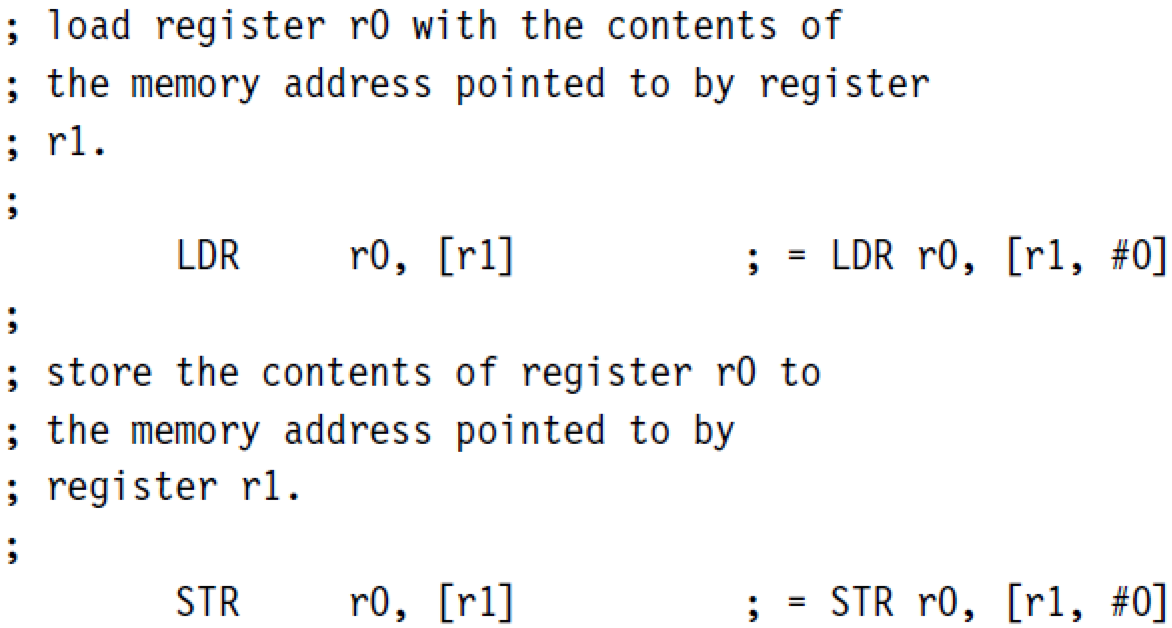


LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*



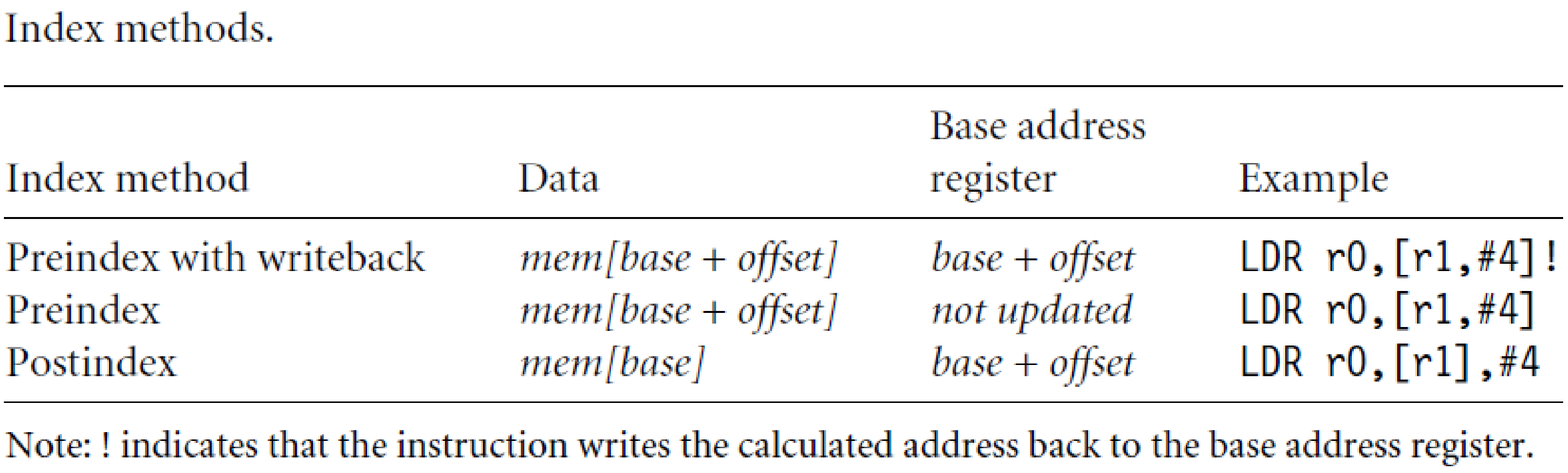
LDRH	load halfword into a register	*Rd <- mem16[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*
LDRSB	load signed byte into a register	*Rd <- SignExtend (mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend (mem16[address])*

Load and Store Instructions



```
; load register r0 with the contents of
; the memory address pointed to by register
; r1.
;
        LDR     r0, [r1]            ; = LDR r0, [r1, #0]
;
; store the contents of register r0 to
; the memory address pointed to by
; register r1.
;
        STR     r0, [r1]            ; = STR r0, [r1, #0]
```

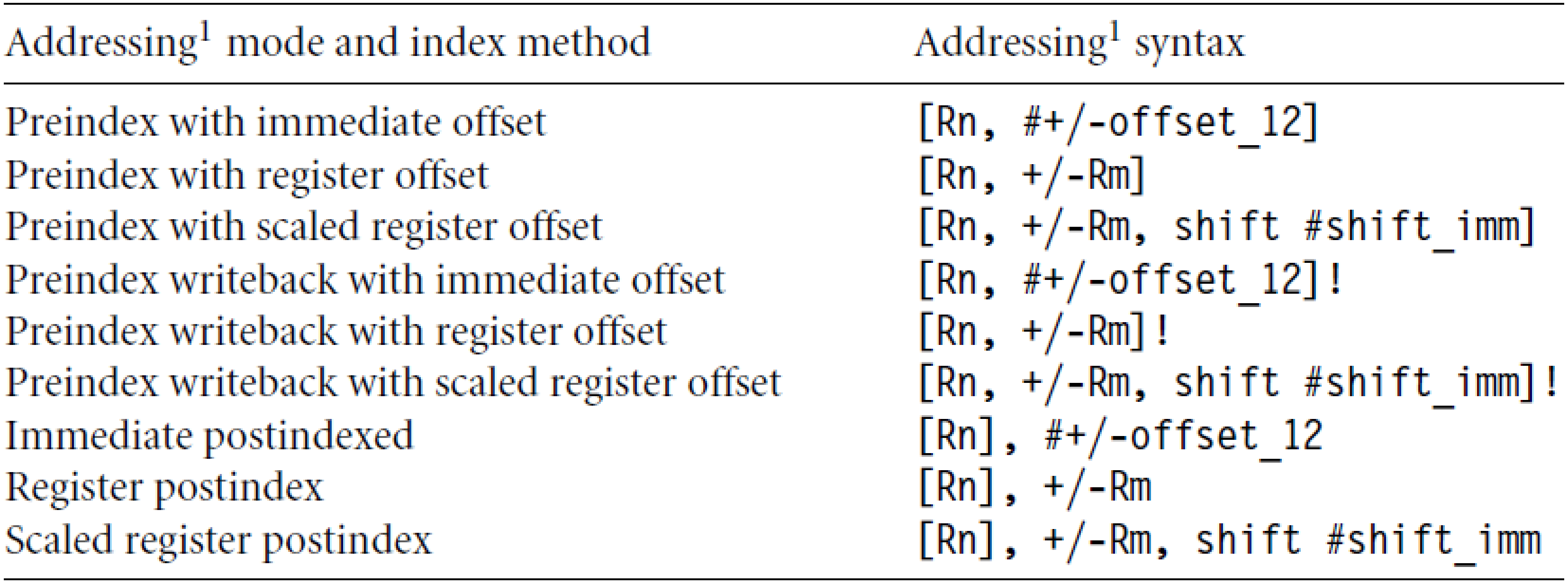
Load and Store Instructions



Index methods.

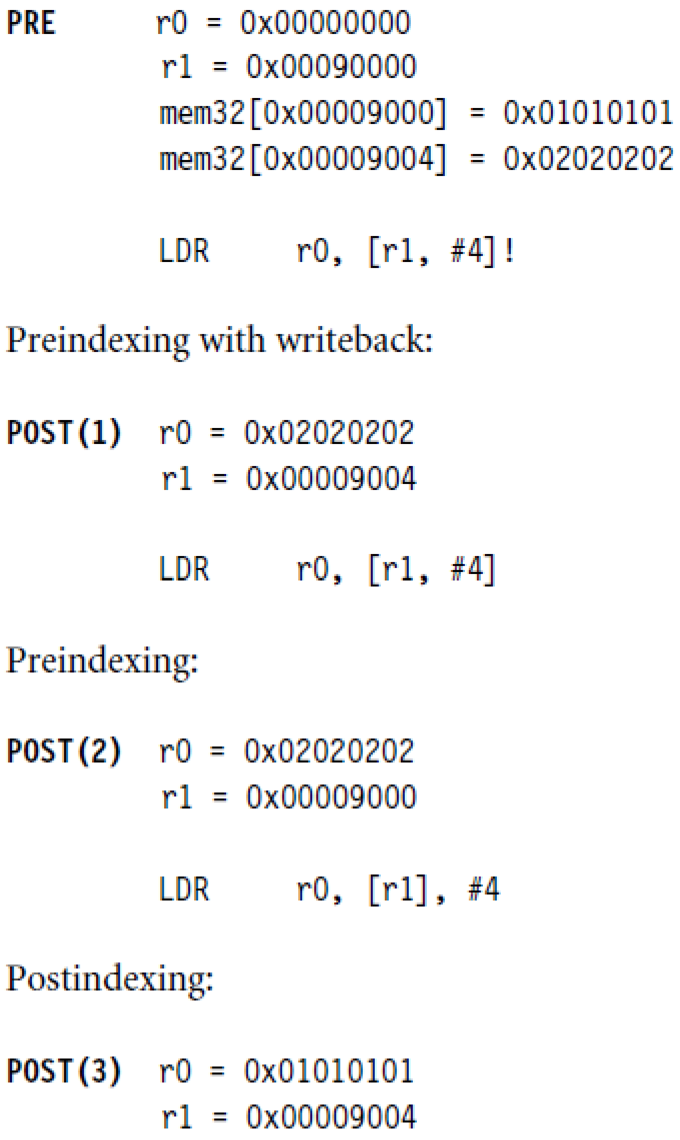
Index method	Data	Base address register	Example
Preindex with writeback	*mem[base + offset]*	*base + offset*	`LDR r0,[r1,#4]!`
Preindex	*mem[base + offset]*	*not updated*	`LDR r0,[r1,#4]`
Postindex	*mem[base]*	*base + offset*	`LDR r0,[r1],#4`

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



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`

Load and Store Instructions



```
PRE      r0 = 0x00000000
         r1 = 0x00090000
         mem32[0x00009000] = 0x01010101
         mem32[0x00009004] = 0x02020202

         LDR     r0, [r1, #4]!
```

Preindexing with writeback:

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

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

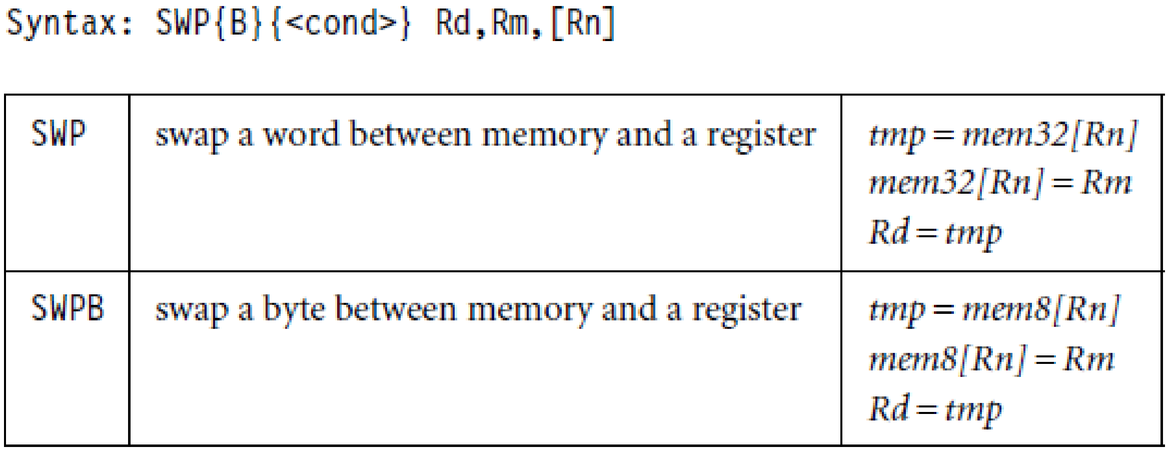
Preindexing:

```
POST(2)  r0 = 0x02020202
         r1 = 0x00009000

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

Postindexing:

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

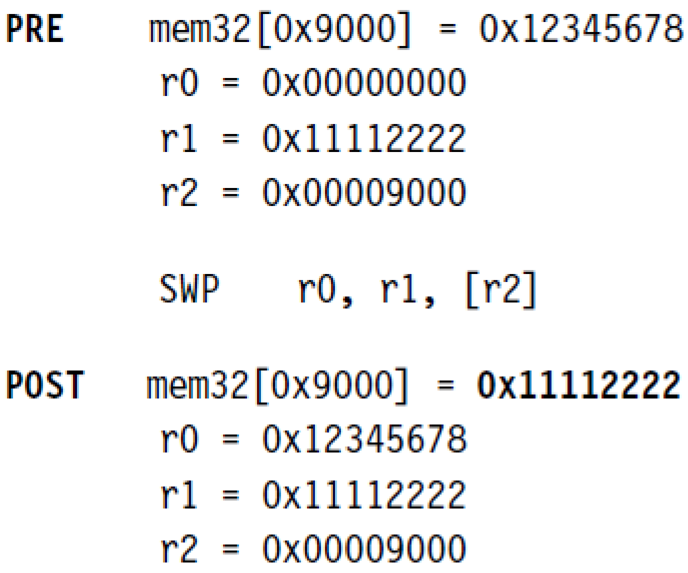
Swap Instructions



```
Syntax: SWP{B}{<cond>} Rd,Rm,[Rn]
```

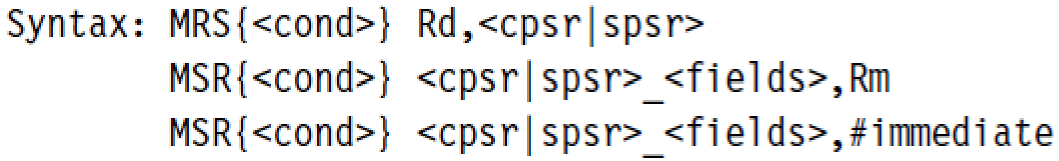
SWP	swap a word between memory and a register	$tmp = mem32[Rn]$ $mem32[Rn] = Rm$ $Rd = tmp$
SWPB	swap a byte between memory and a register	$tmp = mem8[Rn]$ $mem8[Rn] = Rm$ $Rd = tmp$

- Its an atomic operation – It reads and writes location in the same bus operation preventing any other instruction from reading and writing to the location until it completes

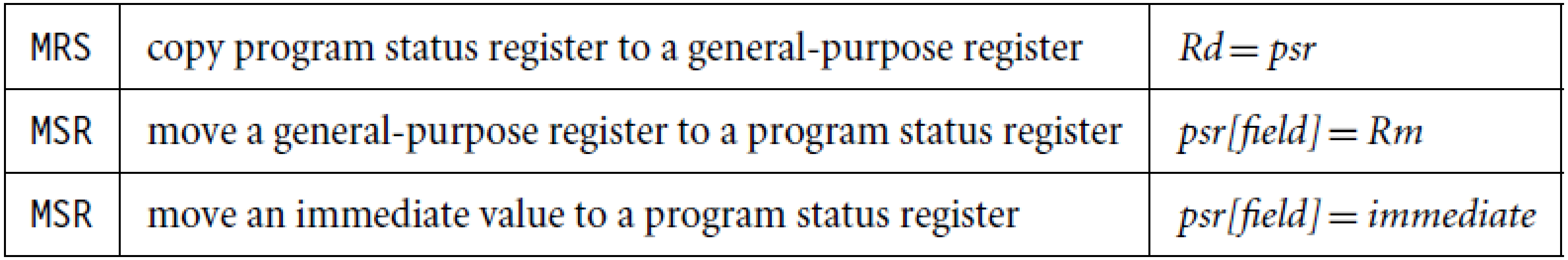


```
PRE     mem32[0x9000] = 0x12345678
         r0 = 0x00000000
         r1 = 0x11112222
         r2 = 0x00009000

         SWP    r0, r1, [r2]

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

Program Status Register Instructions



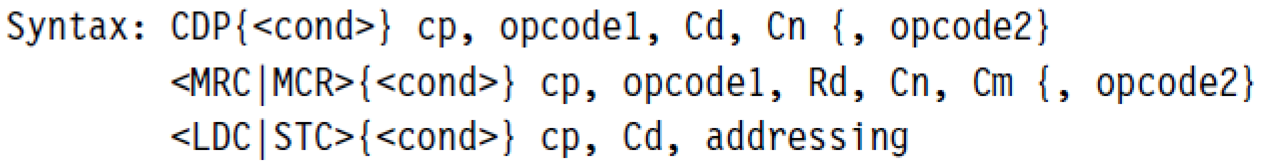
```
Syntax: MRS{<cond>} Rd,<cpsr|spsr>
        MSR{<cond>} <cpsr|spsr>_<fields>,Rm
        MSR{<cond>} <cpsr|spsr>_<fields>,#immediate
```



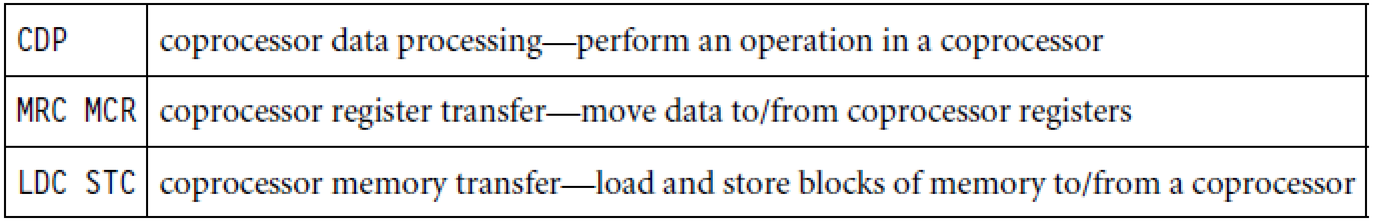
MRS	copy program status register to a general-purpose register	*Rd* = *psr*
MSR	move a general-purpose register to a program status register	*psr[field]* = *Rm*
MSR	move an immediate value to a program status register	*psr[field]* = *immediate*

- **CPSR_<fields>** refer to Control (C) , Extension (x) , Status (s) , Flags (f)

Coprocessor Instructions



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



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

- **"cp"** field in the syntax refer to coprocessor number p0 to p15.
- **"opcode"** field refer to operation to take place on coprocessor
-

Thank you

Harish V. Mekali
Assistant Professor, Dept. of ECE, BMSCE

hvm.ece@bmsce.ac.in / harishmekali@gmail.com

+91-9538765141

http://harishvmekali.blogspot.in/p/technology.html

I acknowledge and appreciate ARM University Program(AUP) and ARM Embedded Systems pvt. Ltd. for their continuous support.

References

Video lectures :

1. Mr. Chrish Shore, ARM Training Manager, UK
The ARM University Program, ARM Architecture Fundamentals
(https://www.youtube.com/watch?v=7LqPJGnBPMM)

2. Dr. Santanu Chaudhury, Dept. of Electrical Engineering, IIT Delhi
Lecture - 5 ARM : (https://www.youtube.com/watch?v=4VRtujwa_b8)

Website :

3. http://infocenter.arm.com/help/index.jsp

Textbooks:

1. ARM system developers guide, Andrew N Sloss, Dominic Symes and Chris Wright, Elsevier, Morgan Kaufman publishers, 2008.
2. ARM System-on-Chip Architecture, Steve Furber, Second Edition, Pearson, 2015