

**AKSHAYA INSTITUTE OF  
TECHNOLOGY**  
Lingapura, Tumkur-Koratagere Road, Tumkur-572106.



# DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Prepared by: - Mrs. THIPPAMMA S  
Assistant Professor, Department of  
CSE. Akshaya Institute of  
Technology, Tumakuru

# AKSHAYA INSTITUTE OF TECHNOLOGY

Lingapura, Obalapura Post, Koratagere Road, Tumakuru - 572106

## DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

### VISION

To empower the students to be technically competent, innovative and self-motivated with human values and contribute significantly towards betterment of society and to respond swiftly to the challenges of the changing world.



### MISSION

**M1:** To achieve academic excellence by imparting in-depth and competitive knowledge to the students through effective teaching pedagogies and hands on experience on cutting edge technologies.

**M2:** To collaborate with industry and academia for achieving quality technical education and knowledge transfer through active participation of all the stake holders.

**M3:** To prepare students to be life-long learners and to upgrade their skills through Centre of Excellence in the thrust areas of Computer Science and Engineering.



### Program Specific Outcomes (PSOs)

*After Successful Completion of Computer Science and Engineering Program Students will be able to*

- \* Apply fundamental knowledge for professional software development as well as to acquire new skills.
- \* Implement disciplinary knowledge in problem solving, analyzing and decision-making abilities through different domains like database management, networking, algorithms, and programming as well as research and development.
- \* Make use of modern computer tools for creating innovative career paths, to become an entrepreneur or desire for higher studies.

### Program Educational Objectives (PEOs)

**PEO1:** Graduates expose strong skills and abilities to work in industries and research organizations.

**PEO3:** Graduates engage in team work to function as responsible professional with good ethical behavior and leadership skills.

**PEO3:** Graduates engage in life-long learning and innovations in multi disciplinary areas.

# Module -2

## Chapter 3

### Introduction to the ARM Instruction Set

- Data Processing Instructions
- Branch Instructions
- Software Interrupt Instructions
- Program Status Register Instructions
- Coprocessor Instructions
- Loading Constants.

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

# 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 instrs.**)

## Syntax:

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

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

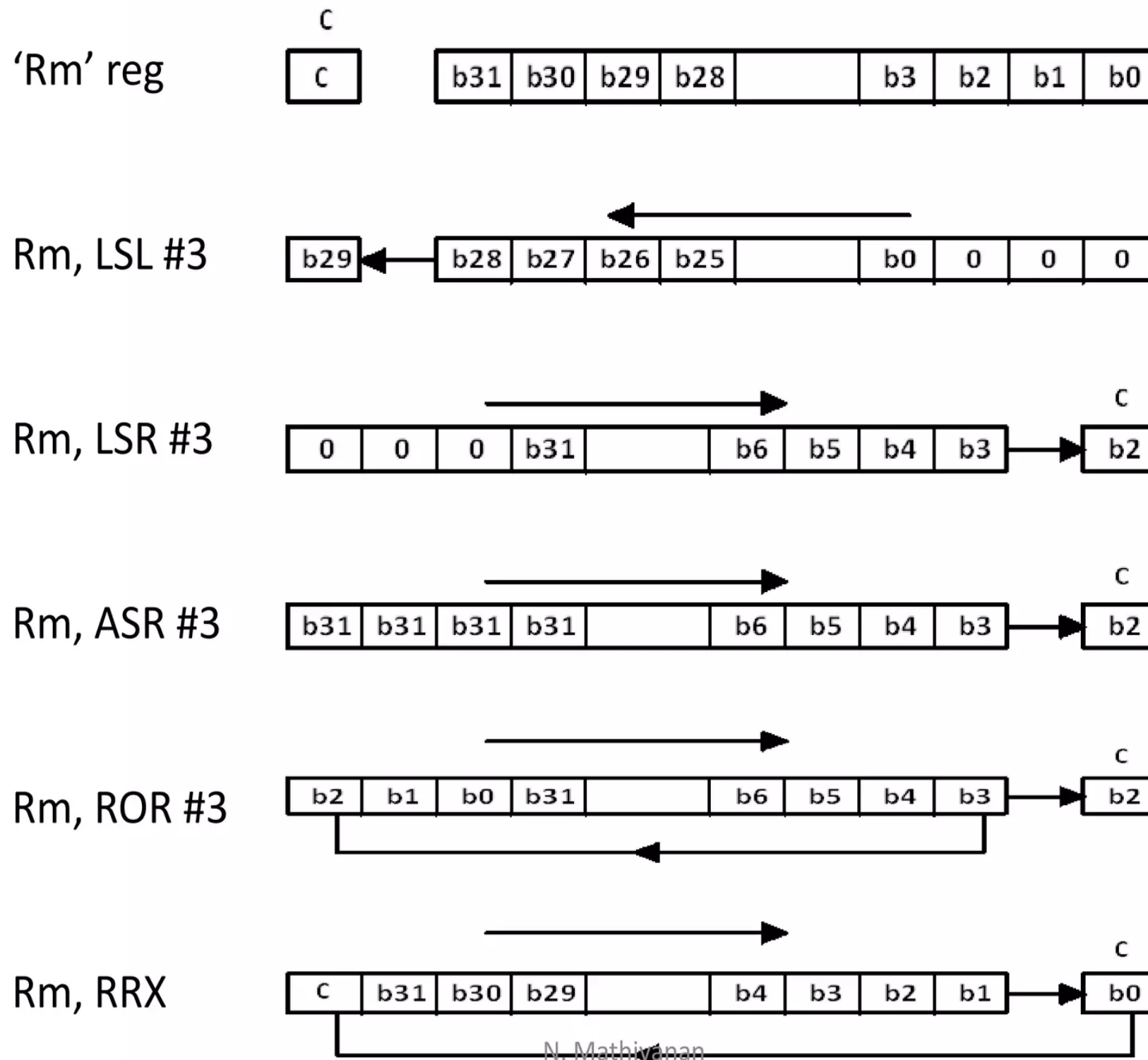
**'n'** may be **'Rm'**, **'#const'** or **'Rs, <shift|rotate> N'**

**Rd** - destination, **Rn** - 1<sup>st</sup> operand, **Rn/Rm/Rs** remains unchanged

- Condition codes & flag states tested

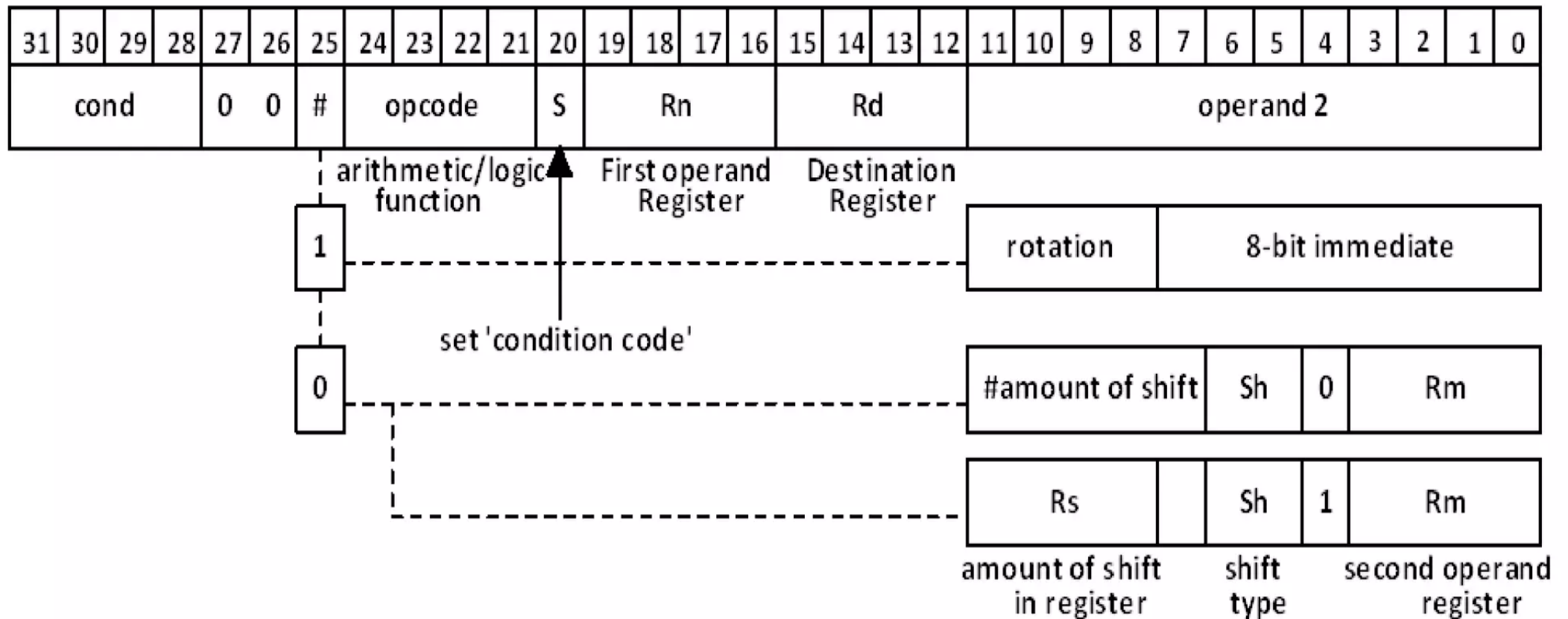
Mnemonic Extension	Condition Tested	Cond. Code	Flags Tested	Mnemonic Extension	Condition Tested	Cond. Code	Flags Tested
<b>EQ</b>	Equal	0000	Z = 1	<b>HI</b>	Unsigned higher	1000	C=1, Z=0
<b>NE</b>	Not Equal	0001	Z = 0	<b>LS</b>	Unsigned Lower or same	1001	C=0, Z=1
<b>CS/HS</b>	Carry Set / unsigned higher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

• Examples of shift, rotate operations



N. Mathivanan

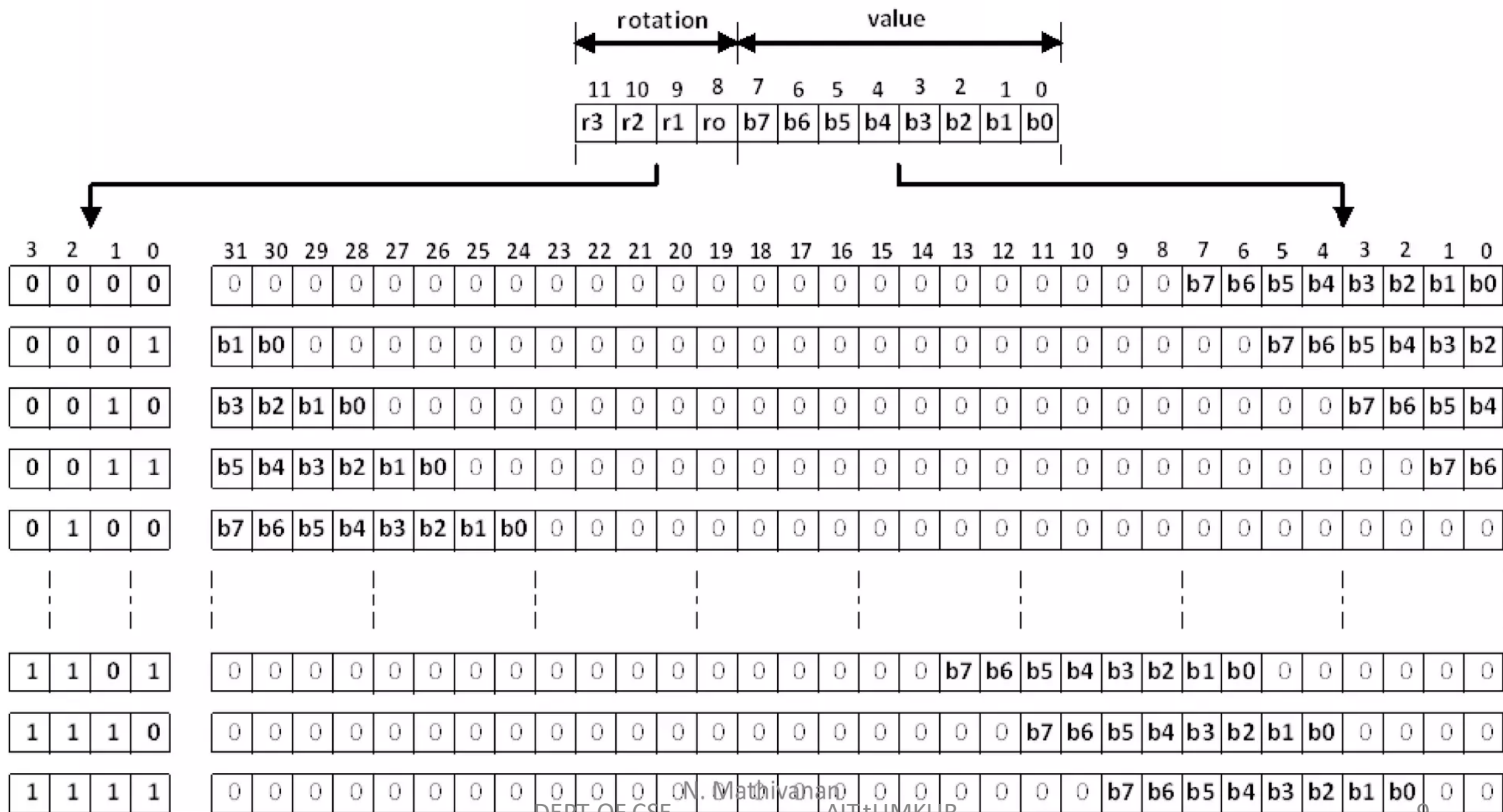
• Binary encoding of data processing instructions





# Immediate constants

- Construction of 32-bit const. using 8-bit value and 4-bit rotation
- Example: `MOV r0, #0xFF000000` is `MOV r0, #0xFF, 8`  
`MOV r0, #0xFFFFFFFF` is `MVN r0, #0x00`



Assembly instructions given below. (verify using Keil  $\mu$ Vision)

```
MOV r0, #0x00          MOV r0, #0xFF000000          MOV r0, #0xFFFFFFFF
MVN r1, #0x01         MOV r0, #0xFC000003          MOV r0, #0x55555555
```

**Solution:** (Demo: Prog01.s)

Assembly Instruction	Assembled equivalent
MOV r0, #0x00	MOV r0, #0x00
MOV r0, #0xFF000000	MOV r0, #0xFF, 8
MOV r0, #0xFFFFFFFF	MVN r0, #00
MVN r1, #0x01	MVN r1, #0x01
MOV r0, #0xFC000003	MOV r0, #0xFF, 6
MOV r0, #0x55555555	<b>Error generated</b>

## • Basic data processing instructions

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

N. Mathivanan

# Data Processing Instructions - Examples

1. `ADDS r0,r1,r2, LSL #3; r0=r1+(r2*8), flags change`
2. `CMP r0,#5 ; If r0 == 5, set Z flag`  
`ADDEQ r1,r2,r3 ; If Z=1, r1=r2+r3 else skip`
3. `AND r0,r0,#0x1 ; status of bottom bit of r0`
4. `BIC r0,r1,r2 ; 1 in r2 bits clears r1 bits`
5. `CMN r0,#6400 ; flags change on r0+6400, r0 is same`
6. The following program fragment implements 'is the value of 'A' 1 or 5 or 8?' functionality. 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`

# Data Processing Instructions - Examples

1. `ADDS r0,r1,r2, LSL #3;` `r0=r1+(r2*8)`, flags change
2. `CMP r0,#5` ; If `r0 == 5`, set Z flag  
`ADDEQ r1,r2,r3` ; If `Z=1`, `r1=r2+r3` else skip
3. `AND r0,r0,#0x1` ; status of bottom bit of r0
4. `BIC r0,r1,r2` ; 1 in r2 bits clears r1 bits
5. `CMN r0,#6400` ; flags change on `r0+6400`, r0 is s
6. The following program fragment implements 'is the value of 'A' 1 or 5 or functionality. 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

```

## • Features of Conditional Execution instructions

Improves execution speed and offers high code density

Illustration:

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

# Questions

1. What is the result of execution of the following?

```
(i)  ADD      r9, r8, r8, LSL #2
      RSB      r10, r9, r9 LSL #3

(ii) MOVS     r7, r7
      RSBMI    r7, r7, #00
```

2. Place two's complement of -1 in r6.

3. Implement ARM assembly program for the 'C' loop:

```
if((a==b) && (c==d)) e++;
```

3. Write an ARM instruction that converts ASCII codes of **lower case alphabets to upper case**.

4. Implement (**if --- then ---else**) functionality using ARM instructions for "Test whether a value in a register is positive or negative, if positive save +1 else save -1 in a register".

## Branch instructions

- Divert sequential execution / CALL a subroutine,
- Range: **+/- 32 MB** from current position (i.e. **PC-8**), PC relative offset
- Instruction has **24 bits** allocated for specifying word **offset**
- With 24-bit offset **append '00'** at LSB, **extend sign** bit, place **into PC**
- **PC** is set to point to **new address**
- How is asm instruction '**here B here**' encoded? - Ans. **0xEAFFFFF**
  - Hint: [31:28] = 1110, [27:24] = 1010, [23:0] = 24-bit offset

<b>B</b>	Branch	<b>B label</b>	PC = label, ( <b>unconditional branch</b> )
<b>BL</b>	Branch and Link	<b>BL label</b>	LR = PC-4, PC = label, ( <b>CALL functionality</b> )
<b>BX</b>	Branch and Exchange	<b>BX Rm</b>	PC = Rm, 'T' bit of CPSR = 1 ( <b>to ARM state</b> )
<b>BLX</b>	Branch with Link	<b>BLX Rm</b>	LR = PC-4, PC = Rm, 'T' bit of CPSR = 1
	Exchange	<b>BLX label</b>	LR = PC-4, 'T' bit of CPSR = 1, PC = label



## 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

```

# 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
- Passing parameters between procedures: Methods -
  - Using Registers, Parameter Block, Stack

# Branch Instructions - Examples

## 3. Example of using 'BX' instruction

; ARM state codes

```

CODE32                ; 32-bit instructions follow
LDR    r0,=tcode+1    ; address of tcode to r0,
                    ; +1 to enter Thumb state
MOV    lr,pc          ; save return address
BX     r0              ; branch to thumb code
---
```

```

; ARM code continues
```

; Thumb state codes

```

CODE16                ; to begin Thumb code execution
tcode ADD    r0,#1     ; Thumb code halfword aligned
---
```

```

BX     lr              ; return to ARM code & state
```

## 4. Example of using 'BLX' instruction

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

N. Mathivanan

# Load-Store Instructions

## • Single Transfer Instruction

- Transfers **boundary aligned** Word/HW/Byte between mem & r
- **LDR** and **STR** instructions
- Address of mem loc. is given by **Base Addr. +/- Offset**
- **Addressing modes**: method of providing offset
  - Register addressing - A **register** contains offset
  - Immediate addressing - **Immediate constant** is offset
  - Scaled addressing - Offset in a reg 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>#`

## • Single register transfer instructions 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	<code>STR Rd, [Rn, Rm]</code>	$\text{mem32}[Rn+Rm] = Rd$
	Immediate addressing (with offset zero)	<code>STR Rd, [Rn]</code>	$\text{mem32}[Rn] = Rd$
	Immediate addressing	<code>STR Rd, [Rn, #offset]</code>	$\text{mem32}[Rn+\text{offset}] = Rd$
	Scaled addressing	<code>STR Rd, [Rn, Rm LSL #n]</code>	$\text{mem32}[Rn+(Rm \ll n)] = Rd$
LDR	Register addressing	<code>LDR Rd, [Rn, Rm]</code>	$Rd = \text{mem32}[Rn+Rm]$
	Immediate addressing (with offset zero)	<code>LDR Rd, [Rn]</code>	$Rd = \text{mem32}[Rn]$
	Immediate addressing	<code>LDR Rd, [Rn, #offset]</code>	$Rd = \text{mem32}[Rn+\text{offset}]$
	Scaled addressing	<code>LDR Rd, [Rn, Rm LSL #n]</code>	$Rd = \text{mem32}[Rn+(Rm \ll n)]$

- **Indexing methods** (base pointer update options)

- Preindexed:

$\langle \text{opcode} \rangle \{ \langle \text{cond} \rangle \} \{ \langle \text{type} \rangle \} \text{Rd}, [\text{Rn} \{ , \langle \text{offset} \rangle \}]$

- Preindexed with write back (note the exclamation symbol)

$\langle \text{opcode} \rangle \{ \langle \text{cond} \rangle \} \{ \langle \text{type} \rangle \} \text{Rd}, [\text{Rn} \{ , \langle \text{offset} \rangle \}] !$

- Postindexed

$\langle \text{opcode} \rangle \{ \langle \text{cond} \rangle \} \text{Rd}, [\text{Rn}] , \langle \text{offset} \rangle$

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

N. Mathivanan

## • 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`

N. Mathivanan

# Multiple Load-Store Instructions

- Transfers data between multiple regs & mem 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)

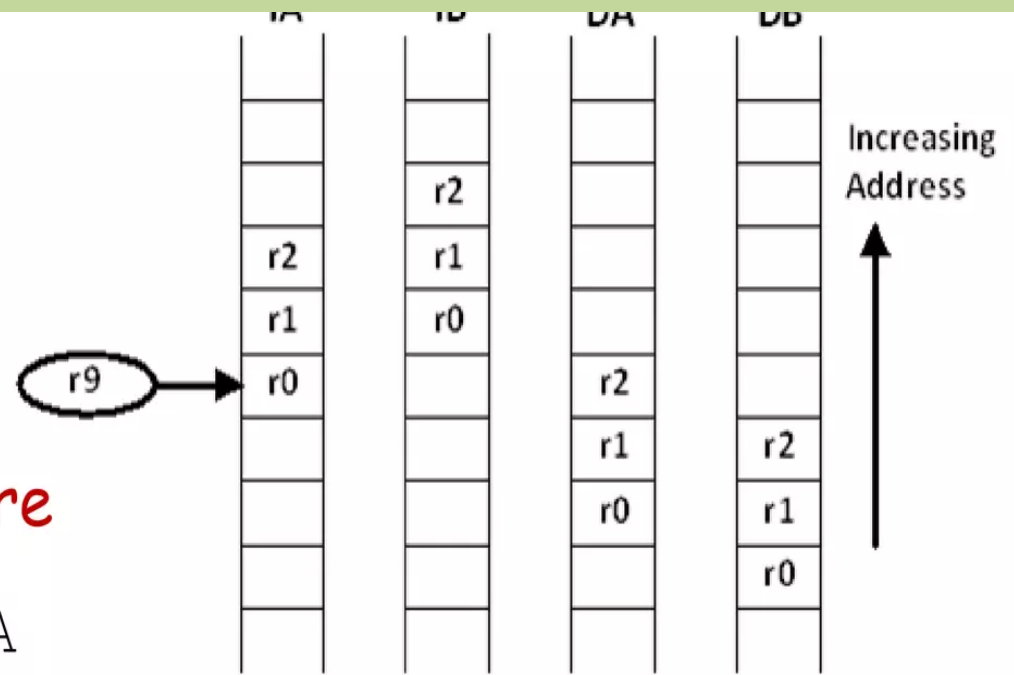
<code>&lt;mode&gt;</code>	Description	Start address	End address	Rn!
<b>IA</b>	Increment After	$R_n$	$R_n + N * 4 - 4$	$R_n + N * 4$
<b>IB</b>	Increment Before	$R_n + 4$	$R_n + N * 4$	$R_n + N * 4$
<b>DA</b>	Decrement After	$R_n - N * 4 + 4$	$R_n$	$R_n - N * 4$
<b>DB</b>	Decrement Before	$R_n - N * 4$	$R_n - 4$	$R_n - N * 4$



• Examples

```
STMxx r9, {r0-r2}
```

```
LDMxx r9, {r0-r2}
```



• Used in pairs for temp store/restore

```
STMIA - LDMDB, STMIB - LDMDA
```

```
STMDA - LDMIB, STMDB - LDMIA
```

• Store & restore operation by STMIB & LDMDA pair (testprog2)

Program	R0	r1	r2	r3	[r0+0x04]	[r0+0x08]	[r0+0x0C]
	0x00000100	0x00000011	0x00000022	0x00000033	0x00000000	0x00000000	0x00000000
STMIB r0!, {r1-r3}	0x0000010C	0x00000011	0x00000022	0x00000033	0x00000011	0x00000022	0x00000033
MOV r1,#00	0x0000010C	0x00000000	0x00000022	0x00000033	0x00000011	0x00000022	0x00000033
MOV r2,#00	0x0000010C	0x00000000	0x00000000	0x00000033	0x00000011	0x00000022	0x00000033
MOV r3,#00	0x0000010C	0x00000000	0x00000000	0x00000000	0x00000011	0x00000022	0x00000033
LDMDA r0!, {r1-r3}	0x00000100	0x00000011	0x00000022	0x00000033	0x00000011	0x00000022	0x00000033

## Stack operations

- Alternative suffixes to IA, IB, DA and DB
- Full/Empty, Ascending/Descending (FD, FA, ED, EA) (8085 stack-FD)
- E.g.: `STMFD r13, {r2, r0, r3-r8, r1}`

Stack type	Store multiple instruction (Push operation)	Load multiple instruction (Pop operation)
Full Descending	STMFD (STMDB)	LDMFD (LDMIA)
Full Ascending	STMFA (STMIB)	LDMFA (LDMDA)
Empty Descending	STMED (STMEDA)	LDMED (LDMIB)
Empty Ascending	STMEA (STMIA)	LDMEA (LDMDB)

## Swap instructions (also known as semaphore instructions)

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

# 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:** `MOVS PC, r14`
- **Examples - Semihosting**
  - Mechanism to communicate & use I/O facilities on host computer running debugger

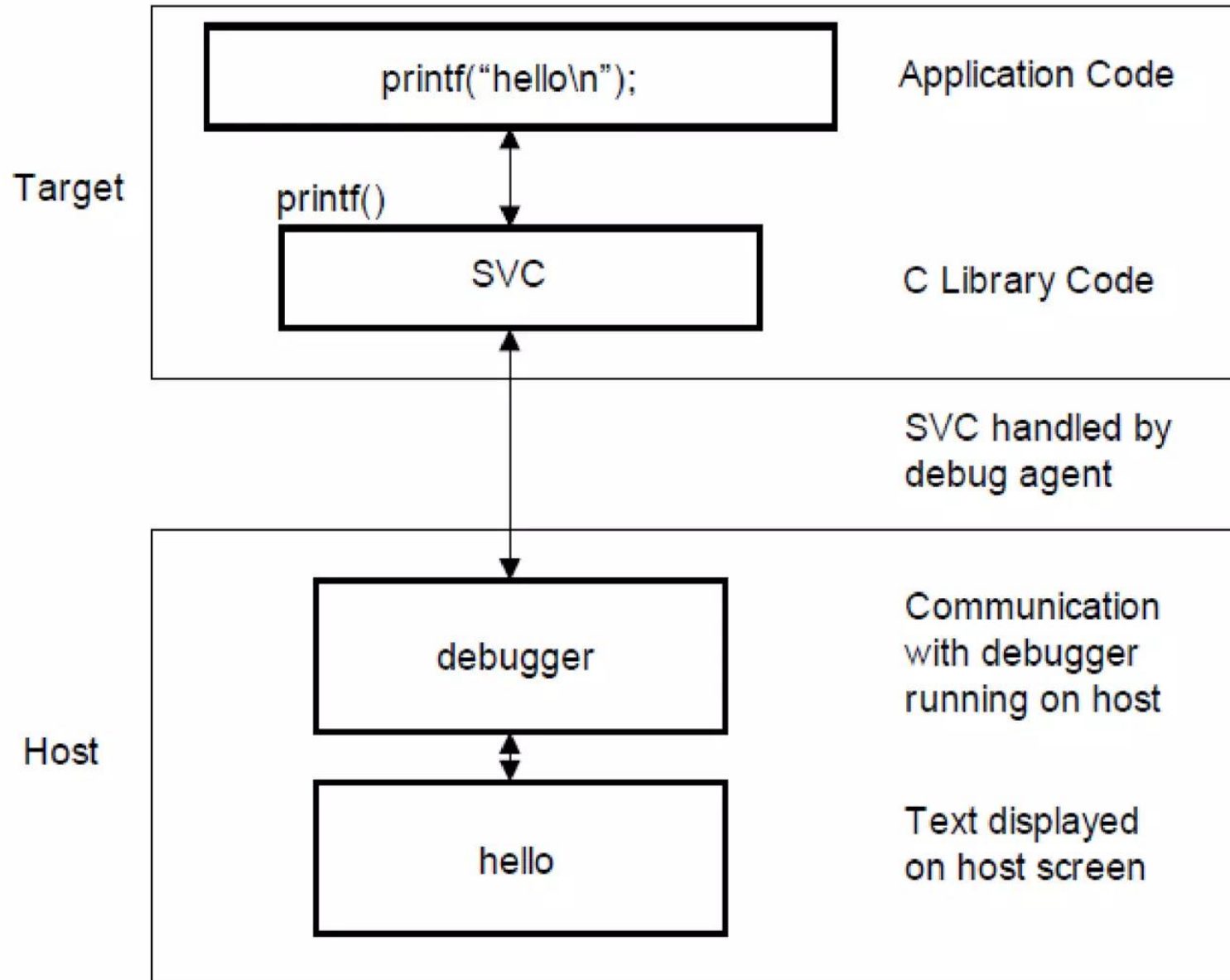
```
MOV    r0, #0x18    ; program termination
```

```
LDR    r1, =0x20026 ;
```

```
SVC    #0x123456    ; semihosting CALL
```

- Distinguish subroutine call and `SWI` instruction execution

- Semihosting overview



# 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
```

# Coprocessor Instructions

- ARM supports **16 coprocessors**
- Coprocessor implemented in **hardware, software or both**
- Coprocessor **contains instruction pipeline, instruction decoding logic, handshake logic, register bank**, special processing logic with its own data path.
- It is also **connected to data bus** like ARM core processor
- Instructions in program memory are available for coprocessors also.
- **Works in conjunction with ARM core** and process same instruction stream as ARM, but **executes only instructions meant for coprocessor** and ignores ARM and other coprocessor's instructions
- **3 types of instructions:** data processing, register transfer, memory transfer

- Coprocessor instructions

CDP	Coprocessor data processing	CDP p5, 2, c12, c10, c3, 4	Coprocessor number 5, opcode1 – 2, opcode2 – 4, Coprocessor destination register – 12, Coprocessor source registers – 10 & 3
MCR	Move to coprocessor from ARM register	MCR p14, 1, r7, c7, c12, 6	Coprocessor number 14, opcode1 – 2, opcode2 – 4, ARM source register – r7, Coprocessor destination registers – 10 & 3
MRC	Move to ARM register from coprocessor	MRC p15, 5, r4, c0, c2, 3	Coprocessor number 15, opcode1 – 5, opcode2 – 3, ARM destination register – r4, Coprocessor source registers – 0 & 2
MCRR	Move to coprocessor from two ARM registers	MCRR p10, 3, r4, r5, c2	Coprocessor number 10, opcode1 – 3, ARM source registers – r4, r5 Coprocessor destination register – 2
MRRC	Move to two ARM registers from coprocessor	MRCC p8, 4, r2, r3, c3	Coprocessor number 8, opcode1 – 4, ARM destination registers – r2, r3 Coprocessor source register – 3



- Coprocessor instructions *continued.....*

LDC	Load coprocessor register	LDC p6, c1, [r4]	Coprocessor number 6, Coprocessor register c1 is loaded with data from memory address in r4.
		LDC p6, c4, [r2, #4]	Coprocessor number 6, Coprocessor register c4 is loaded with data from memory address in r2 +4.
STC	Store coprocessor register	STC p8, c8, [r2, #4]!	Coprocessor number 8, Memory address is [r2] + 4 Store c8 in memory and then r2 =r2+4
		STC p8, c9, [r2], #-16	Coprocessor number 8, Memory address is [r2] Store c9 in memory and then r2 =r2-16

# Thumb State Instruction Set

- **16-bit instructions** (compressed form), executable in Thumb state.
- Instruction decoder **decompresses to 32-bit** ARM equivalent
- **Features:**
  - Works on **32-bit values**, produces **32-bit addr** for mem access
  - Access to **low registers** (r0-r7) similar to ARM state
  - **Restricted access to high registers** (r8-r15), `MOV`, `ADD`, `CMP` can access
  - Thumb state **enabled by 'T' bit** (if set) of CPSR.
  - If enabled, **fetches 16-bit code** from **HW aligned** addresses, **PC** is incremented by **two bytes**
  - **Instructions removed:** `MLA`, `RSB`, `RSC`, `MSR`, `MRS`, `SWP`, `SWPB` and coprocessor instructions;
  - **New Instructions:** `LSL`, `LSR`, `ASR`, `ROR` (barrel shifter can't be combined with any instruction)
  - **Not conditionally executable** (except 'B' branch instruction)

- **Data processing instructions**

- 2-addr format, No conditional execution, No 'S' suffix (always ON)
- MOV, MVN, ADD, ADC, SUB, SBC, MUL, AND, ORR, EOR, BIC,
- NEG, LSL, LSR, ASR, ROR, CMP, CMN, TST

- **Branch instructions**

- Branch instruction ('B') is conditionally executable
- B{cond}, B, BL, BX, BLX

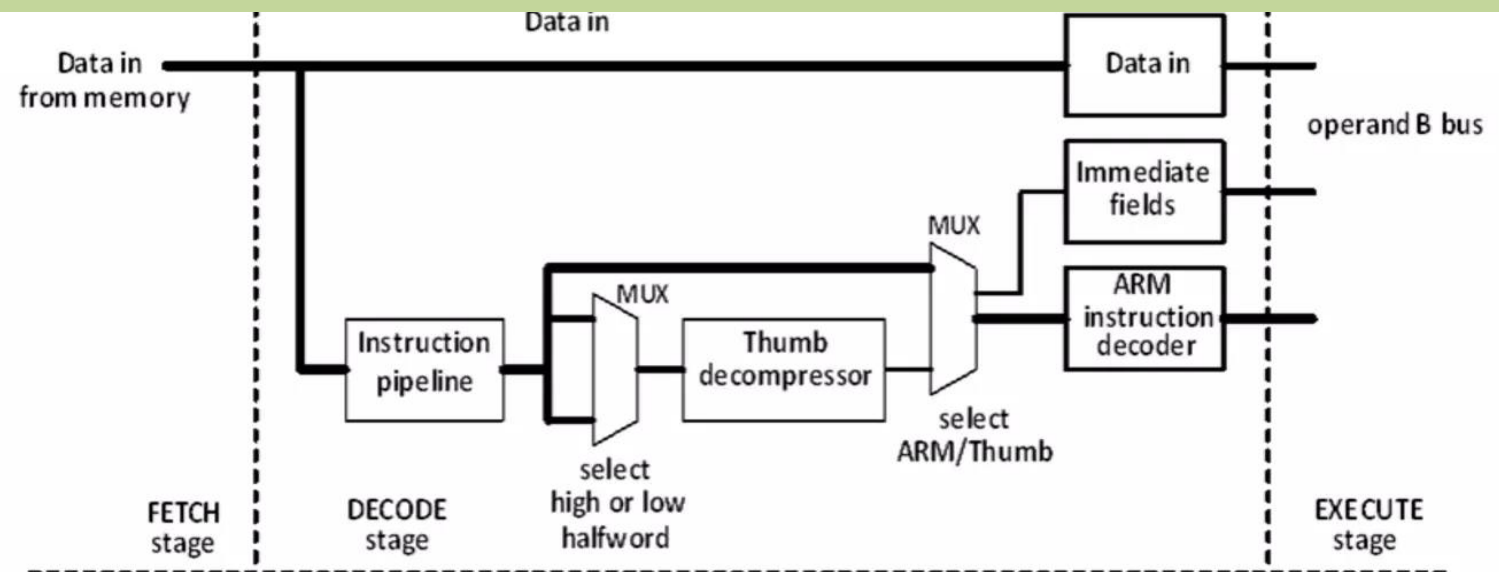
- **Load-Store instructions**

- Offset: register, immediate (0-124 words), relative to PC/SP
- LDR, STR, LDMIA, STMIA, PUSH, POP

- **Software interrupt instructions**

- Generates exception, Use just 8-bit interpreted value
- Handler enters in ARM mode

- Thumb decompressor



- Illustration:** Thumb uses less memory space than ARM though more instructions are used - Let r1, r2 have value and divisor. The program produces quotient and remainder in r3 and r2.

<pre> MOV    r3, #0 loop  SUB    r0, r0, r1       ADDGE  r3, r3, #1       BGE   loop       ADD   r2, r0, r1 </pre>	<pre> MOV    r3, #0 loop  ADD    r3, #1       SUB    r0, r1       BGE   loop       SUB    r3, #1       ADD   r2, r0, r1 </pre>
<p><b>ARM code</b></p> <p>Instructions: 5</p> <p>Memory space: 20 bytes</p>	<p><b>Thumb code</b></p> <p>Instructions: 6</p> <p>Memory space: 12 bytes</p>

# Review Questions

1. What are the features of 32-bit ARM state instructions?
2. Illustrate the features of conditional execution of ARM instruction with suitable examples.
3. List the classes of ARM state instructions.
4. Write down the syntax of data processing class of instructions and explain.
5. Implement the multiplication of a register by 35 using 'ADD' and 'RSB' instructions.
6. What are the functions of 'TST' and 'BIC' instructions?
7. What is result of execution of 'MOV r0, 0x55555555' instruction?

Why?

8. Write down the instructions providing 'JUMP' and 'CALL' functionalities.
9. What is the branching range of branch instructions?
10. Write down the syntax of single transfer Load-Store class of instructions and explain.
11. Describe the addressing modes of Load-Store instructions.
12. Write down an instruction that transfers a word size data into 'r0' register from a memory location at an offset of 16 bytes from a word aligned memory location whose address is in 'r1' register.
13. Explain the indexing methods supported by ARM7 Load-Store class of instruction.
14. Write down the syntax of Load-Store Multiple instructions. Explain the use of the instructions with examples.
15. What are the advantages & disadvantages of Load-Store Multiple instructions?

16. What is 'FD' stack? Explain with suitable schematics.
17. Write down the pair of load and store multiple instructions that is used in 'EA' type stack.
18. An array of words with 25 elements is in memory and the address of first element is in r1 register. Implement the following 'C' statement using ARM instructions. The value 'const' is in r2 register.  
`array[10] = array[5] + const`
19. Write an ARM7 ALP fragment that implements 'block move' functions assuming the elements of the block are words, the starting address of source block is in 'r9' register, the destination address is in 'r10' register and the size of the block is 8 words.
20. What is the basic task of 'SWP' instruction?
21. What for the PSR instructions are provided?

16. What is 'FD' stack? Explain with suitable schematics.
17. Write down the pair of load and store multiple instructions that is used in 'EA' type stack.
18. An array of words with 25 elements is in memory and the address of first element is in r1 register. Implement the following 'C' statement using ARM instructions. The value 'const' is in r2 register.  
`array[10] = array[5] + const`
19. Write an ARM7 ALP fragment that implements 'block move' functions assuming the elements of the block are words, the starting address of source block is in 'r9' register, the destination address is in 'r10' register and the size of the block is 8 words.
20. What is the basic task of 'SWP' instruction?
21. What for the PSR instructions are provided?