27/12/18

UNIT-III

Types of instructions:- Arithmetic & logical instructions, branch instructions, Addresing modes, Input output operations

Introduction:- From unit-II we concluded that we need an instruction to provide a communication between memory, processor, Input/output devices.

→ Mainly there are 3 different ways of organising instructions :- (i) ARM (Advanced Risk machine instruction) → It follows 32 bit format

(ii) Motrolal processor architechture

(iii) Intel processor architechture.

→ these 3 have diff instruction formats & diff addressing modes. & it's own (i.e; diff) Assembly language notation.

29/12/18 → Speed of processor depends on instruction format design.

ARM (Advanced Risk machine instruction):-

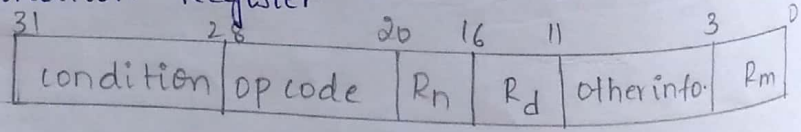
→ ARM processor, mostly will be used in mobiles

→ It follows 32-bit instruction format.

Eg:- opcode R_d, R_n, R_m

R_n, R_m are operands

R_d - destination Register.



Arithmetic & logic Instructions

→ The basic assembly language expression for arithmetic instruction is:

opcode R_d, R_n, R_m

→ operation is specified by the opcode.

→ Rm, Rn are general purpose registers used for operands.

→ The result is placed in register r_d

opcode.
Eg:- ADD R0, R2, R4

"opcodes are written in capital letters in ARM"

$R_0 \leftarrow [R_2] + [R_4]$ → RTN Notation.

Here [ ] (square braces) means addresses

→ Registers can have addresses.

Eg:- SUB R0, R6, R5.

$R_0 \leftarrow [R_6] - [R_5]$

Eg:- ADD R0, R1, R5, LSL #4

$R_0 \leftarrow [R_1] + [R_5] * 2^4$

Eg:- ADD R0, R3 #17

$R_0 \leftarrow [R_3] + 17$.

Eg:- MUL R0, R1, R2

$R_0 \leftarrow [R_1] * [R_2]$

Eg:- MLA R0, R1, R2, R3.

$R_0 \leftarrow [R_1] * [R_2] + [R_3]$

MLA - multiply accumulate operation (combining together)

→ In ARM 3 to 4 instructions can be written in a single instruction.

→ "MLA" instruction is used for ~~dio~~ "Analog digital signal processing"

Advantages of ARM:-

→ In other instructions sets shift operations are seperately written.

Eg:- ADD R0, R1, R2, LSL #2

$R_0 \leftarrow [R_1] + [R_2] * 2^2$

→ It saves code space and can potentially improve performance.

→ Barrel shifter circuit

2/1/19

ARM logical Instructions:-

→ The instructions are AND, OR, XOR, BIC (Bit clear), MVN

i) AND:-

Eg:- AND Rd, Rn, RM

$R_d \leftarrow [R_n] \wedge [R_m]$

→ AND R_0, R_0, R_1

$R_0 \leftarrow [R_0] \wedge [R_1]$

→ Eg:- R_0 - 02FA62CA

R_1 - 0000FFFF

AND R_0, R_0, R_1

$R_0 \leftarrow [R_0] \wedge [R_1]$ = 62CA

$R_0 \leftarrow$ 000062CA.

OR:-

Eg:- $R_0 \leftarrow$ 02FA62CA

R_1 — 0000FFFF] 02FA

ORR R_0, R_0, R_1

$R_0 \leftarrow$ 02FAFFFF

=

BIC (Bit clear):-

→ BIC R_0, R_0, R_1

→ This instruction is closely related to AND instruction. It complements each bit in operand R_m before doing AND operation with the bits in R_n

R_1 — 0000FFFF

~R_1 — FFFF0000

R_0 - 02FA62CA

$R_0 \leftarrow$ 02FA0000

MVN (Move Negate instruction):-

Eg:- MVN R_d, R_n

$R_d \leftarrow N[R_n]$

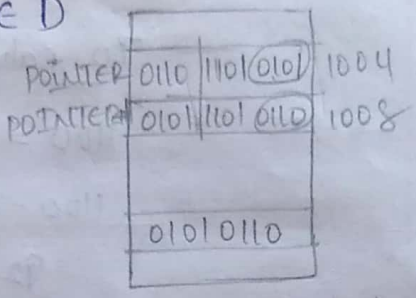
→ $R_3 \leftarrow$ 0F0F0F0F

MVN R_0, R_3

$R_0 \leftarrow$ F0F0F0F0

* ARM Digit packing example:-

problem statement: Extract low order 4-bits from ~~pointer~~ POINTER & POINTER+1 and concatenate them into a single byte location ~~packed~~ PACKED



→ source → destination

```
Move #POINTER, R0      1
MoveByte (R0)+, R1     2
LshiftL  #4, R1        3
MoveByte (R0), R2      4
AND      R2, 0FH       5
OR       R1, R2        6
MoveByte R2, PACKED    7
```

R0	R1	R2
1004	1101 0101	1101 0110
	0101 000	0000 0110
		0101 0110

Move #loc

In ARM

→ destination → source

```
LDR   R0, POINTER      1
LDRB  R0, [R0]         2
LDRB  R2, [R0,#1]      3
AND   R2, R2, #&F      4
ORR   R2, R2, R1, LSL#4  5
STRB  PACKED, R2       6
```

LDR - loader Register

STRB - Store Byte means 8 bits

$R_2 \leftarrow$ 1101 0110
#0F $\leftarrow$ 0000 1111
$R_2 \leftarrow$ 0000 0110

$R_1 -$ 1101 0101 $\leftarrow 4$
$R_1 -$ 0101 000
$R_2 -$ 0000 0110
(OR) R_2 0101 0110

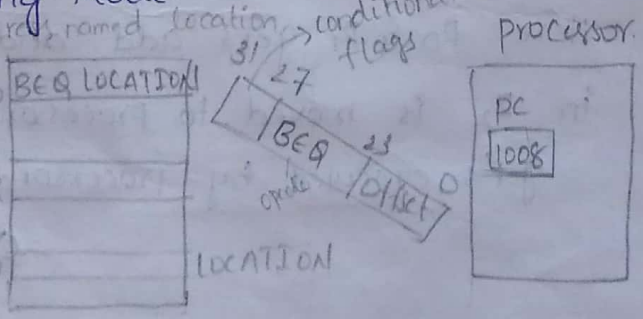
3/11/19

Branch Instructions:-

from Arithmetic & Logical instructions in ARM

→ For branch instructions instruction format is different.
→ It is 32-bit format.
→ Branch Instruction in ARM is effectively explained by relative addressing Mode.

eg:- BEQ LOCATION → Branch Equal to address named location



PC - always contain next instruction address
→ always offset is calculated with respect to pc address

'offset' is how much distance target address exists from Program counter value.

Effective Address = [pc] + offset

LOCATION = 1008 + 92

To caluclate offset

1100 = 1008 + offset

1100 - 1008 = offset

92 = offset.

→ Here BEQ Results in Jumping from one memory location to another memory location.

→ offset is caluclated as effective address - ~~content~~ [pc] (Contents of pc)

→ offset binary value gets stored in 0 - 23 bits.

→ 24 - 31 bits are conditional codes

↳ stores which is binary equivalents 0 or 1

If it is 1 → It wants to get executed. (i.e, memory is asking processor to execute)

In BEQS means set conditional flag to 1

If it is only BEQ it moves to processor but not executed in the ARM. BEQS is executed by processor (conditional flag) in ARM.

If 1 - moves to processor & gets executed by processor in ARM

0 - moves to processor & not get executed by processor in ARM.

Eg:- ADDS R_0, R_1, R_2

Here R_0 & R_1 gets add and the result is stored in R_2 is moved to processor as flag is 1 and get's executed by processor.

CMP Rn, Rm

[Rn] - [Rm]

→ caluclate sum of Array elements in ARM.

In Processor

N- variable name
no of element s

```
Move N, R1
Move #POINTER, R2
   clear R0
→ ADD (R2), R0
  ADD #4, R2
  Decrement R1
  Branch >0 LOOP
  Move R0, SUM.
```

	5	N
LOOP		POINTER
	10	1000
	20	1004
	30	1008
	40	1012
	50	1016
		1020
SUM	150	

In ARM

```
    LDR  R1, N
    LDR  R2, POINTER
    MOV  R0, #0
LOOP LDR  R3, [R2], #4
    ADD  R0, R0, R3
    SUBS R1, R1, #1
    BGT  LOOP
    STR  R0, SUM
```

→ R2 ← [R2 + 4]

Still

R0	R1	R2	R3
0	5	1000	10
10	4	1004	20
30	3	1008	30
60	2	1012	40
100	1	1016	50
150	0	1020	

"BGT" means branch back if not done.

ARM Adressing Modes:-

→ ARM Adressing modes are combination of different addressing modes.

→ It mainly work's with Index mode (offset).

→ Three types of addressing modes:-

(1) Immediate offset.
   (i) pre-Indexed
   (ii) pre-Indexed with write back. Rn gets updated.
   (iii) post Indexed.

(2) offset as magnitude.
   (i) pre-Indexed.
   (ii) pre-Indexed with write back
   (iii) post Indexed.

3. Relative Addressing mode:

RTN notation:

eg: LDR R_d, [Rn, #offset] (offset should be a value only.)

If register contain address, $R_d \leftarrow [R_n + offset]$ / $R_d \leftarrow [[R_n] + offset]$

→ LDR R_d, [R_n, R_m]

If register contain address, $R_n \leftarrow [R_n + R_m]$ / $R_n \leftarrow ([R_n] + [R_m])$ (here R_m is offset)

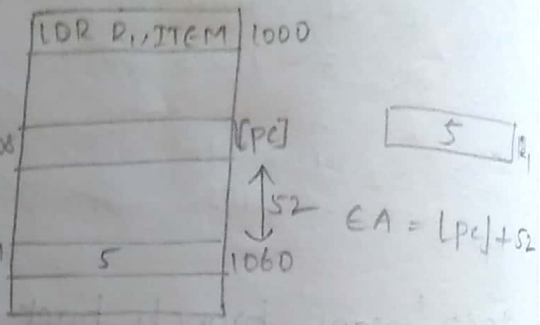
→ STR R_d, [Rn] (source, destin)

$[R_n] \leftarrow [R_d]$ (If register contain address that contain another address that contain value then we use this formulae)

→ LDR R_d, [R_n] $R_d \leftarrow [[R_n]]$ (R_n contents are updated with contents of R_d.)

1. Relative Addressing mode

LDR R_1, ITEM



→ suppose ITEM is at 1060 location. we want to move R_1 to ITEM for that we need to locate PC. Assum pc at 1008 memory location with an offset 52. the the value 5 in ITEM gets stored in R_1

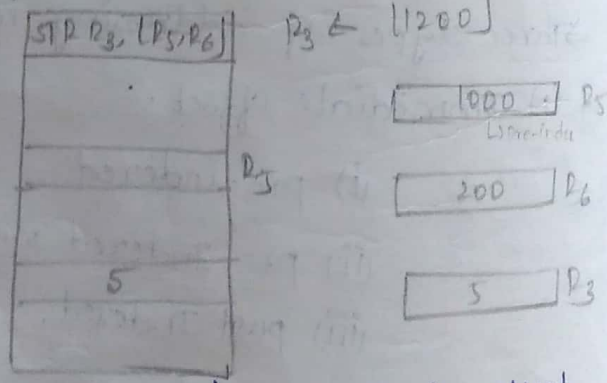
→ Different processor's follow different addressing mode with diff. meaning.

STR R_3, [R_5, R_6] (R_d, R_n, R_m → offset register.)

$R_5 \leftarrow [R_5 + R_6]$

(This is pre-index with write back)

[R_5 + R_6] → pre index.



$R_3 \leftarrow [[R_5], 200]$

$R_3 \leftarrow [1200]$

$R_3 \leftarrow [[R_5] + [R_6]]$ EA = [R_5] + offset

$\leftarrow [1000 + 200]$ = 1000 + 200

$R_3 \leftarrow [1200]$ = 1200.

pre Indexed [Rn, #offset]

EA = [Rn] + offset

pre Indexed with write back

EA = [Rn] + offset

Rn ← [Rn] + offset

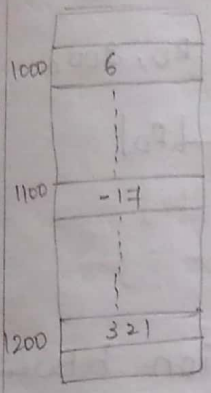
[Rn, #offset]! means write back

Q19 → write Assembly language code for storing two -dimensional Array. caluclate sum of elements in first row

Rd Rn Rm shift

LDR R1, [R2], R10, LSL #2

eg:



1100 base 2

1000

Base register

25 R10

offset register

6 -17 321 ... 25×25

EA = [R2] + [R10] shifted

= 1000 + 100

= 1100

[R2], R10, LSL #2

R2 ← [[R2] + Rm shifted]

R2 ← [1100]

R2 ← 17

→ Generally in memory processor elements of matrix, store row-wise but where as for ARM processor elements of matrix store is column-wise.

→ This is post-index with write back., Here offeset as magnitude.

[Rn], Rm, shift post, index

EA = [Rn]

Rn ← [Rn] + [Rm] shifted

For abve eg:- EA = [R2] = 1000

R2 ← [R2] + [R10] shifted

1000 + 100

R2 ← 1100

EA = [R2] = 1100

$R_2 \leftarrow [R_3] + [R_{10}]$ shifted

$\leftarrow 1100 + 100$

$R_2 \leftarrow 1200$

Eg: STR $R_0, [R_5, \#-4]!$ → write back.

R_n — $R_n(+)$ offset

Immediate offset pre-indexed write back

$EA = [R_5] + \text{offset}$

$= 2012 + (-4)$

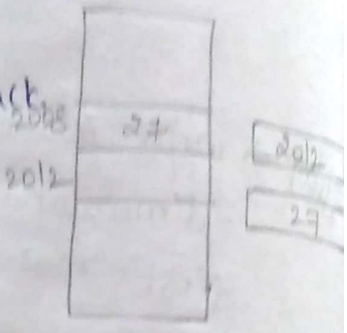
$= 2008$

$R_n \leftarrow [R_n] + \text{offset}$

$2012 + (-4)$

$R_n \leftarrow 2008$

$R_5 \leftarrow 2008$



⇒ STR R_0, 2008.

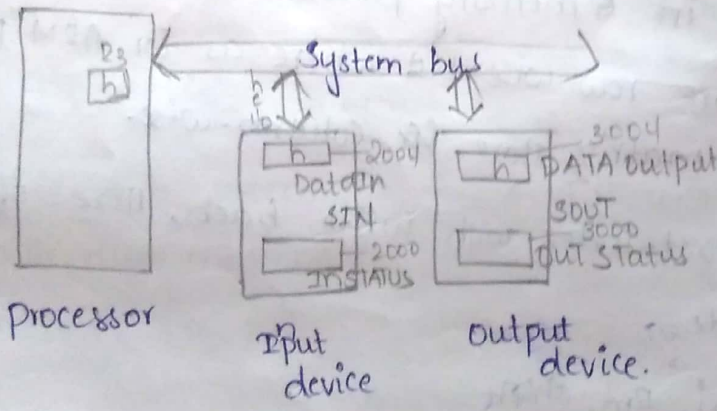
R_n, PTN

$[2008] \leftarrow [R_0]$

STR $R_0, [R_5]$

$[[R_5]] \leftarrow [R_0]$

9/11/19 **Input - output operations:-** communication between input output devices with other hardware components like processor.



Assume like the ~~resin~~ register of IN status is loaded with register of R_1

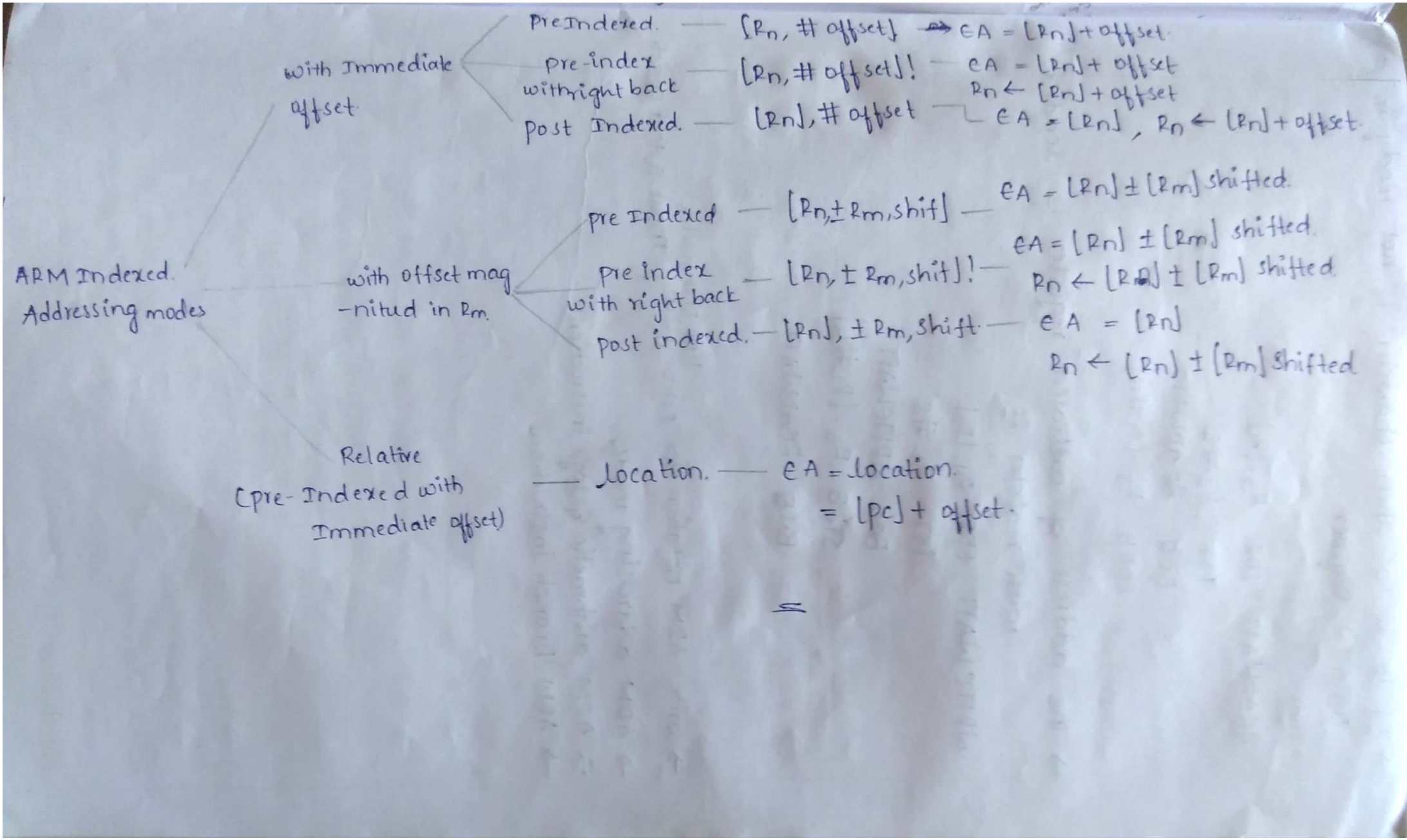
00001000

00000000

Eg: ~~LDR R3, [R1]~~

~~TST R3, #8~~.

→ If 3rd index value is 1 then result is postive. if it is zero then the result is negative.



→ If it is '0' then character is not ready in DATAIN Register

code:- we give 'h' as input

```
READWAIT  LDR   R3,[R1]
          TST   R3,#8
          BEQ   READWAIT
          LDRB  R3,[R1,#4]
```

→ The address of outstated register is loaded with register R_2

→ should get 'h' as o/p

```
WRITEWAIT  LDR   R4,[R2]
           TST   R4,#8
           BEQ   WRITEWAIT
           STRB  R3,[R2,#4]
           BNQ   READWAIT
```

→ we should add this stmt for more than one character.

→ write ARM notation for echo line of characters using Input, Output operator

→ ARM addressing modes

→ ARM arithmetic & logic instructions

→ ARM Branch instructions