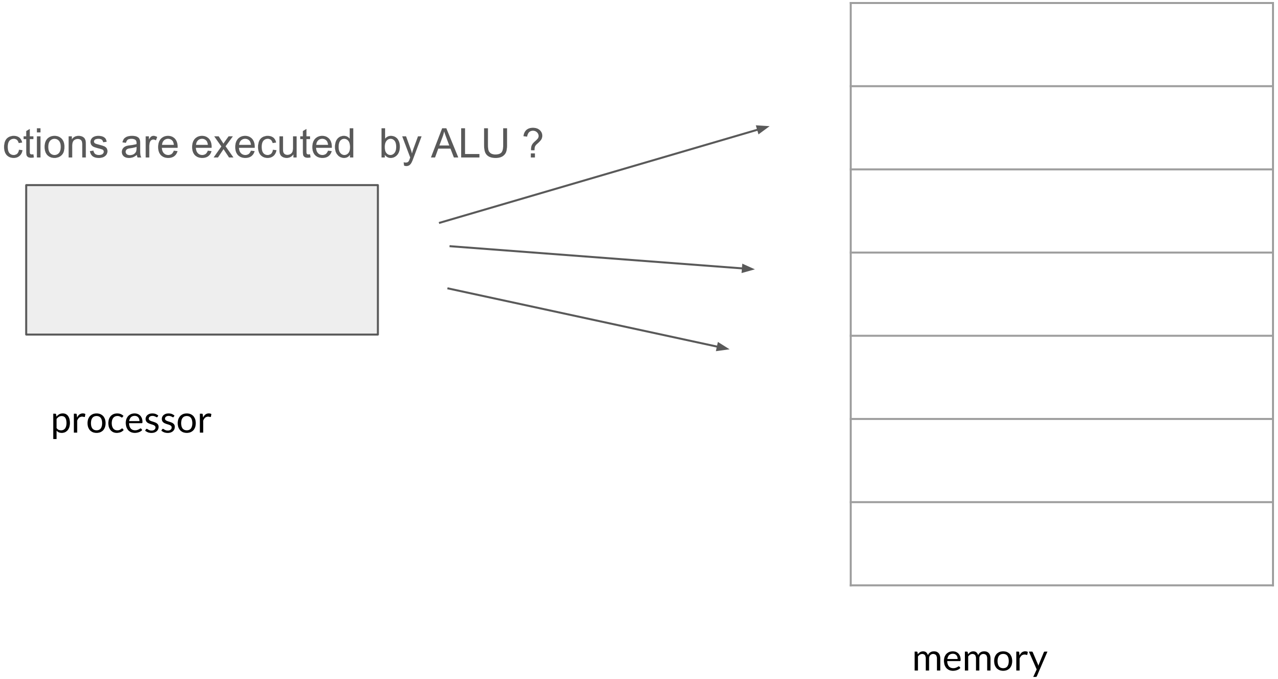
جامعة عين شمس

AIN SHAMS UNIVERSITY

Section 3

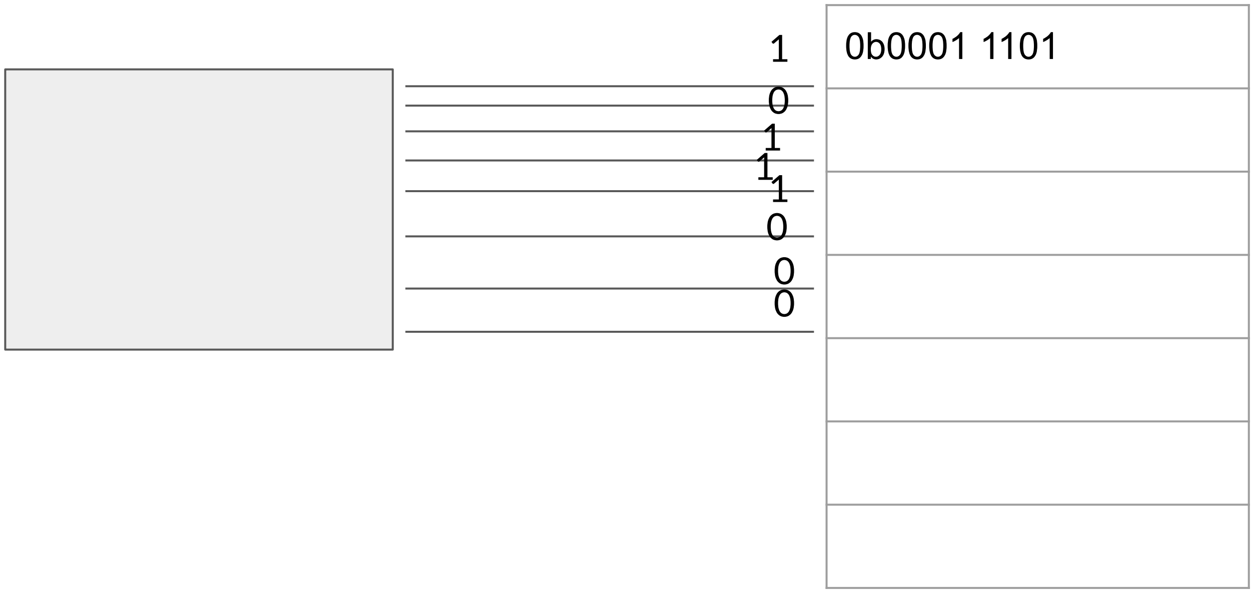
Sheet 2

ALL the instructions are executed by ALU ?

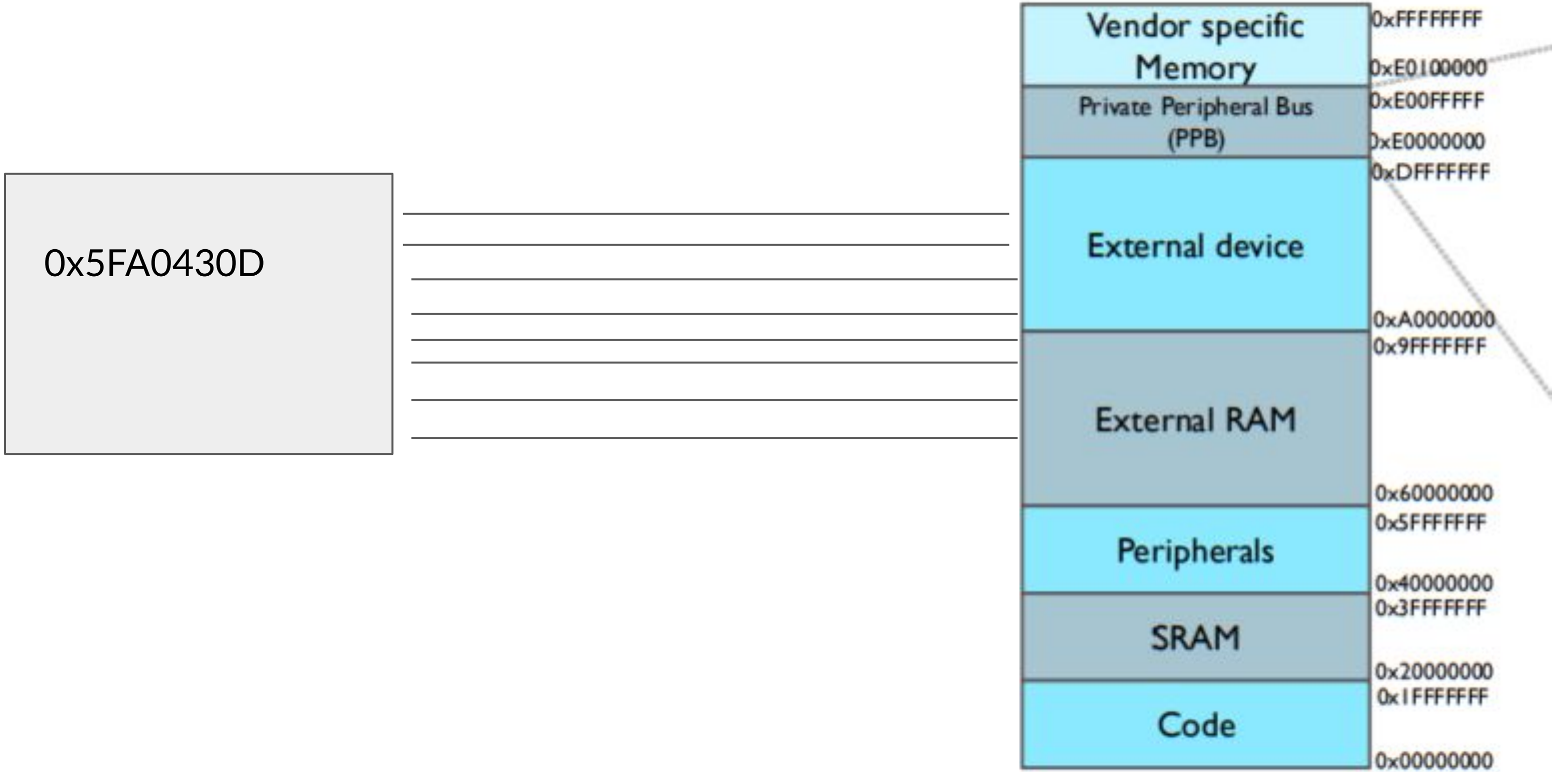


System Bus.

- Data bus .
- Control bus.
- Address bus.



Address bus size can be 8 , 16 ,32
The numbers mean numbers of locations processor can access.
Ex : 32 bits mean 2^32 locations mean 4GB



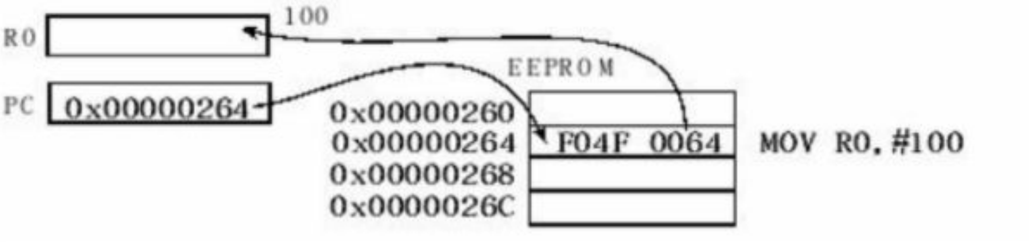
0x5FA0430D

Vendor specific Memory
0xFFFFFFFF
0xE0100000
Private Peripheral Bus (PPB)
0xE00FFFFF
0xE0000000
0xDFFFFFFF
External device
0xA0000000
0x9FFFFFFF
External RAM
0x60000000
0x5FFFFFFF
Peripherals
0x40000000
0x3FFFFFFF
SRAM
0x20000000
0x1FFFFFFF
Code
0x00000000

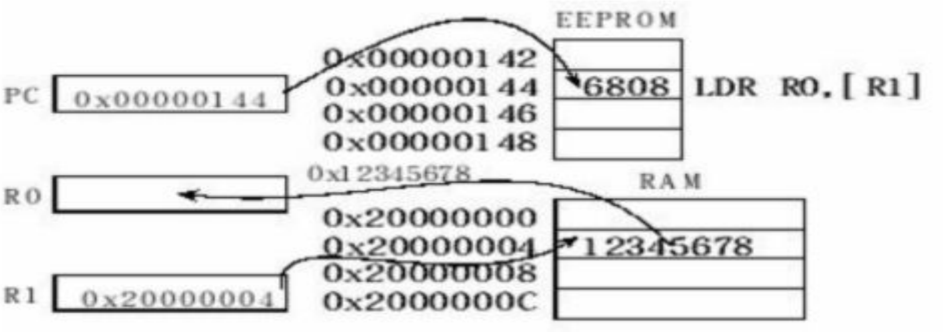
Addressing Mode

❑ Immediate addressing

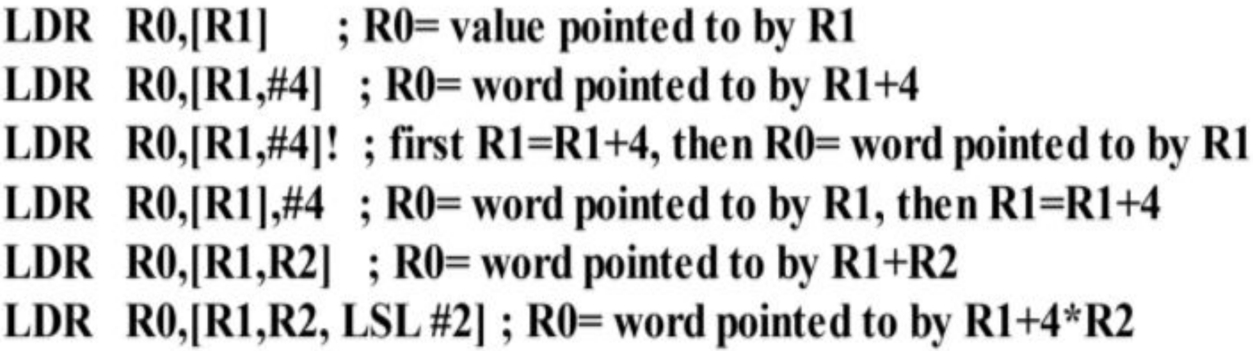
MOV R0,#100 ; R0=100, immediate addressing



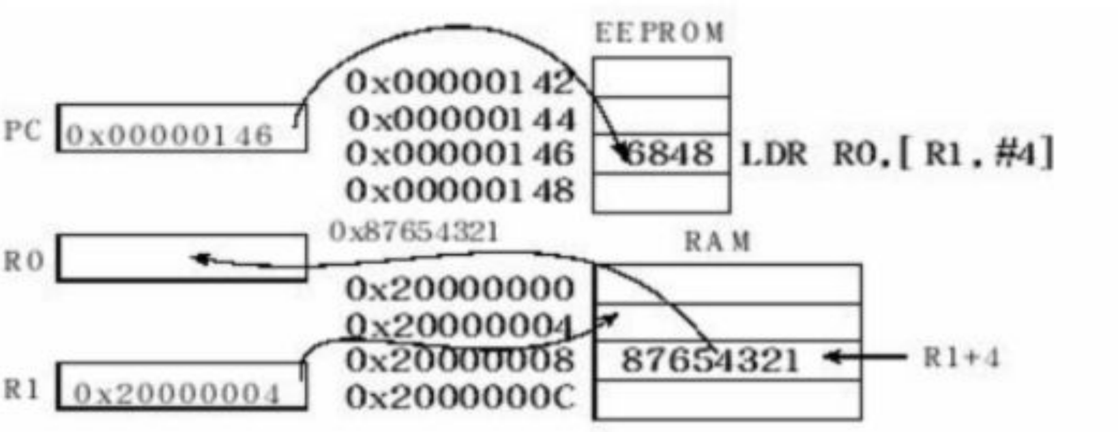
❑ Indexed addressing.



Addressing Mode

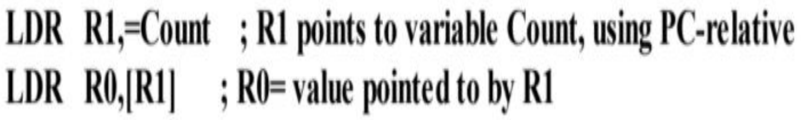


```
LDR   R0,[R1]       ; R0= value pointed to by R1
LDR   R0,[R1,#4]    ; R0= word pointed to by R1+4
LDR   R0,[R1,#4]!   ; first R1=R1+4, then R0= word pointed to by R1
LDR   R0,[R1],#4    ; R0= word pointed to by R1, then R1=R1+4
LDR   R0,[R1,R2]    ; R0= word pointed to by R1+R2
LDR   R0,[R1,R2, LSL #2] ; R0= word pointed to by R1+4*R2
```

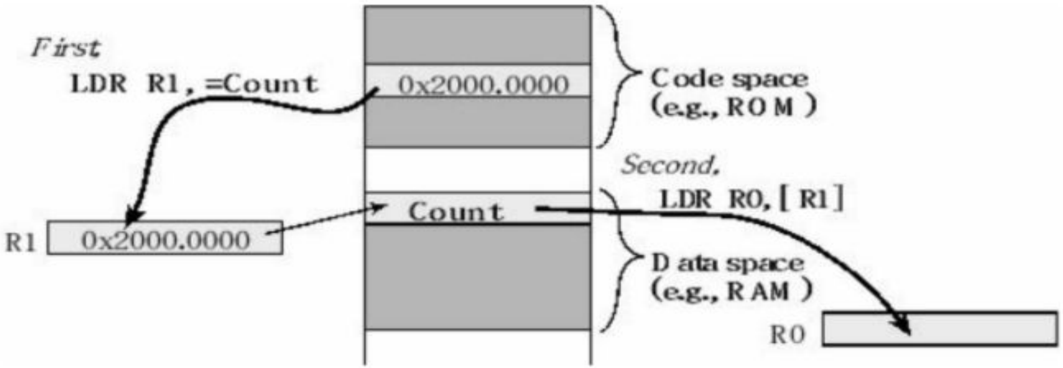


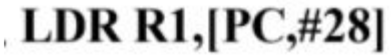
Addressing Mode

- PC-relative addressing

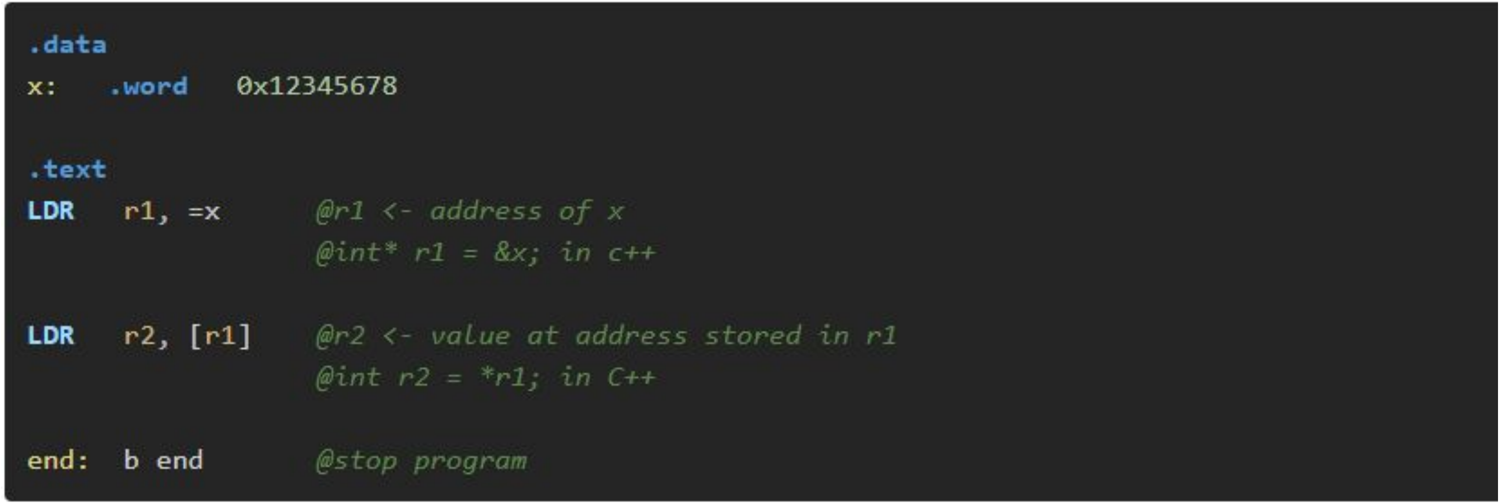


```
LDR  R1,=Count   ; R1 points to variable Count, using PC-relative
LDR  R0,[R1]     ; R0= value pointed to by R1
```



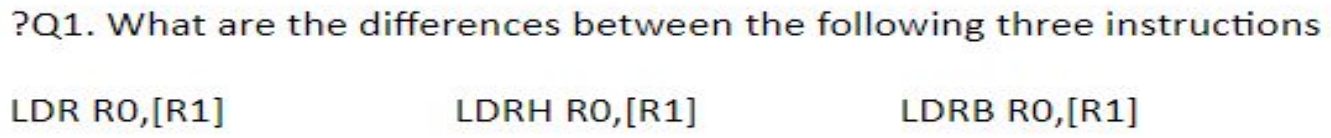


```
LDR R1,[PC,#28]
```



```
.data
x:   .word   0x12345678

.text
LDR   r1, =x        @r1 <- address of x
                    @int* r1 = &x; in c++

LDR   r2, [r1]      @r2 <- value at address stored in r1
                    @int r2 = *r1; in C++

end:  b end         @stop program
```

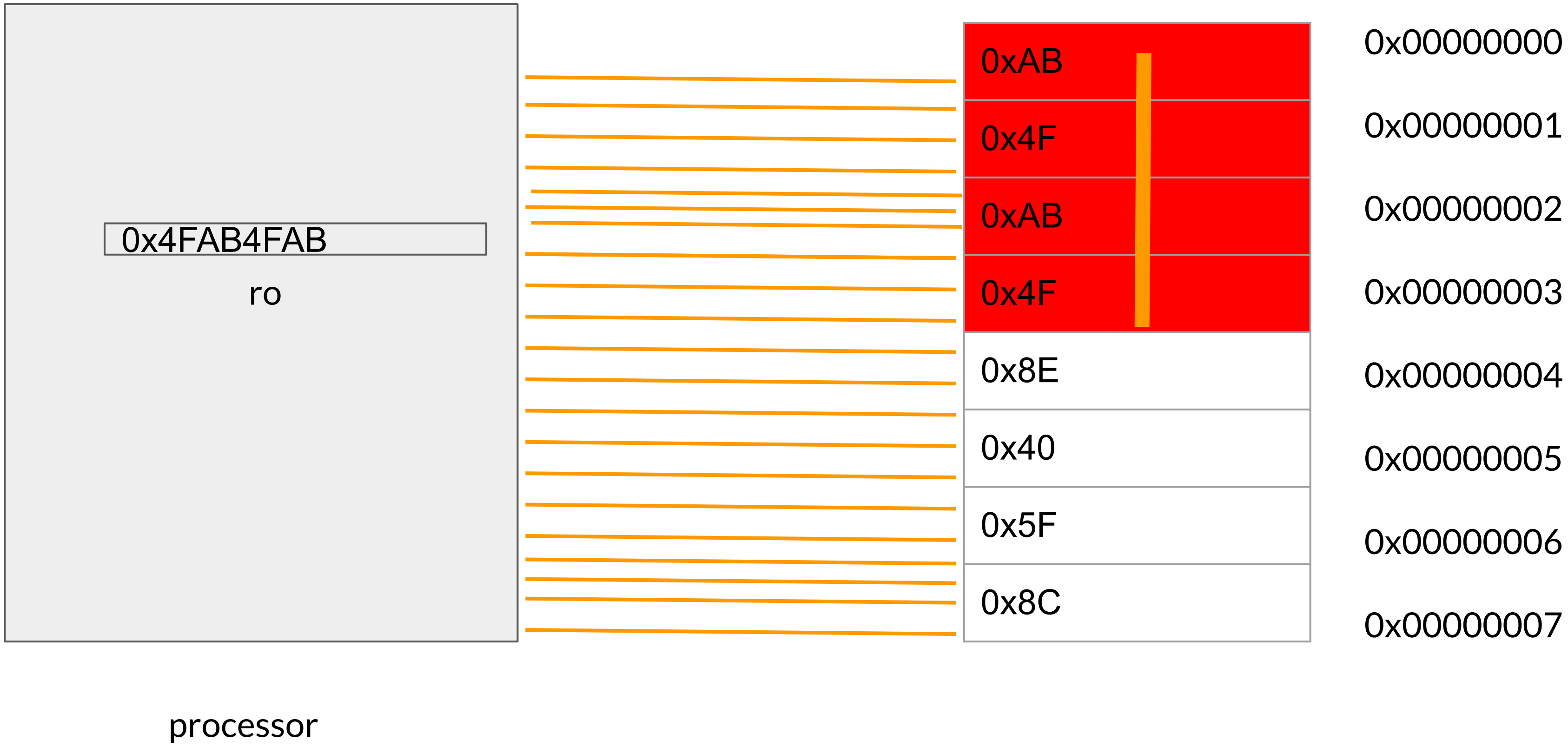
Q1



?Q1. What are the differences between the following three instructions

```
LDR R0,[R1]          LDRH R0,[R1]          LDRB R0,[R1]
```

LDR R0 , [R1]



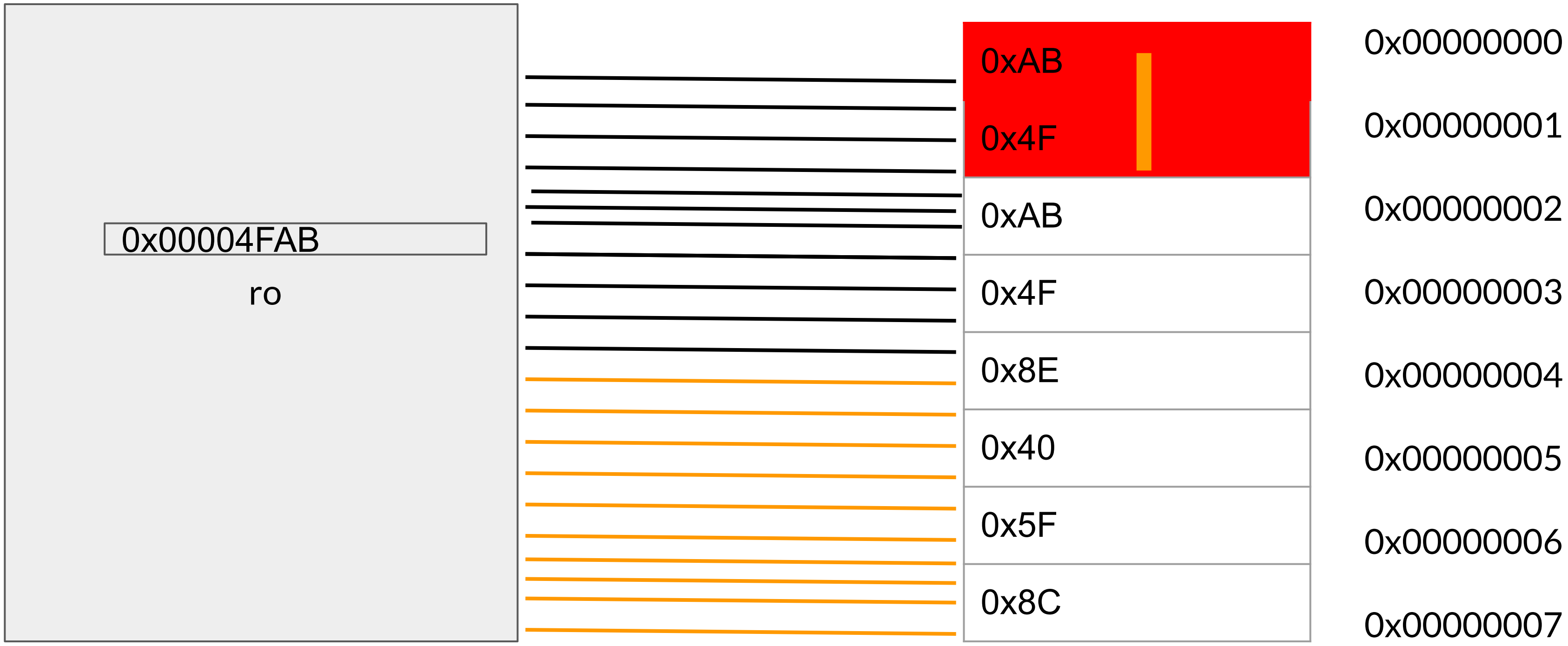
0x4FAB4FAB

ro

processor

0xAB	0x00000000
0x4F	0x00000001
0xAB	0x00000002
0x4F	0x00000003
0x8E	0x00000004
0x40	0x00000005
0x5F	0x00000006
0x8C	0x00000007

LDRH R0 , [R1]



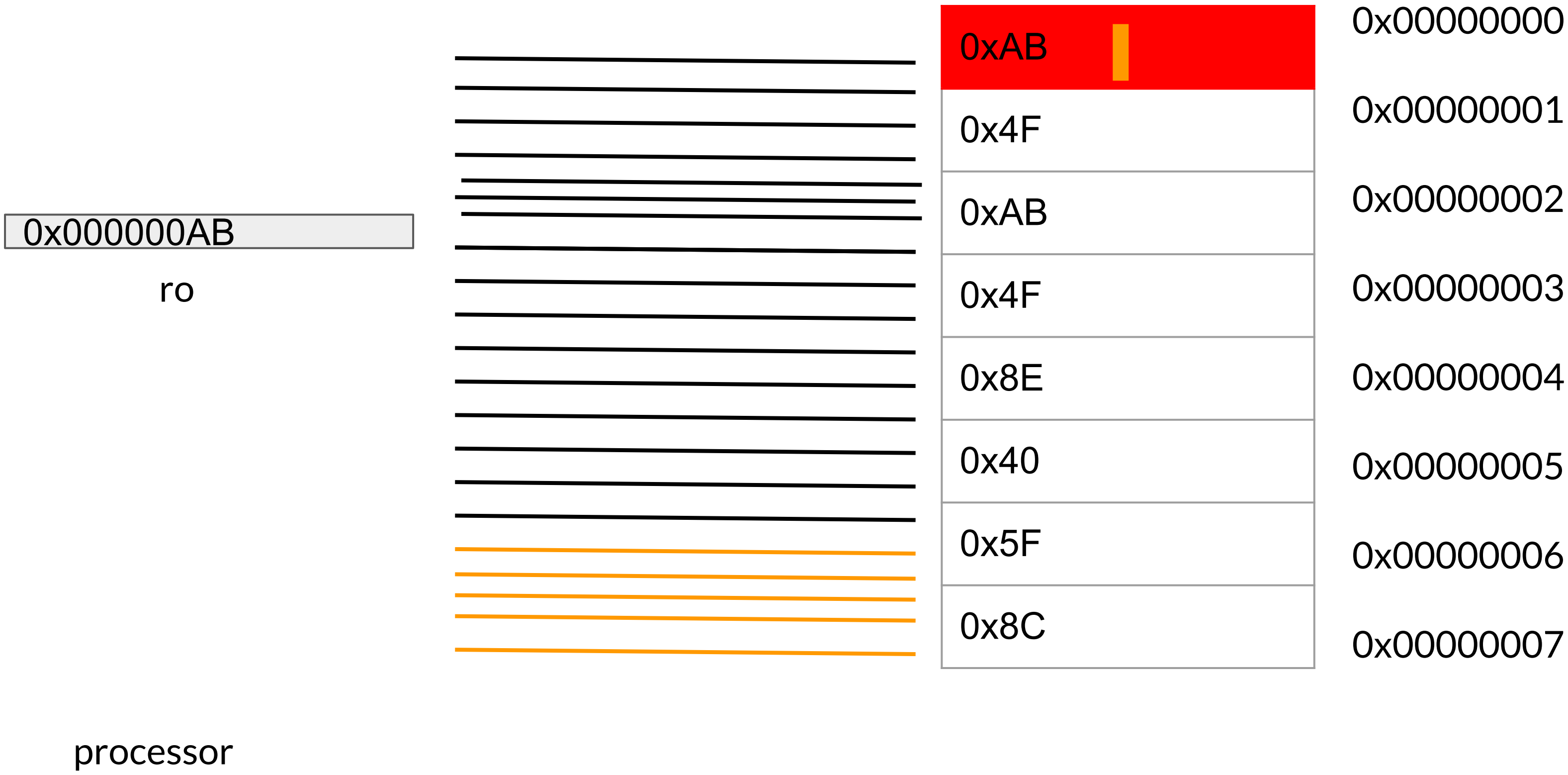
0x00004FAB

ro

0xAB	0x00000000
0x4F	0x00000001
0xAB	0x00000002
0x4F	0x00000003
0x8E	0x00000004
0x40	0x00000005
0x5F	0x00000006
0x8C	0x00000007

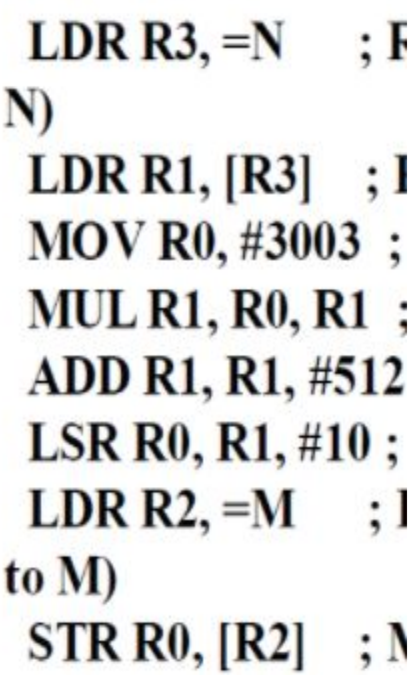
processor

LDRB R0 , [R1]



0xAB	0x00000000
0x4F	0x00000001
0xAB	0x00000002
0x4F	0x00000003
0x8E	0x00000004
0x40	0x00000005
0x5F	0x00000006
0x8C	0x00000007

Q2



```
 LDR R3, =N      ; F
N)
 LDR R1, [R3]    ; I
 MOV R0, #3003  ;
 MUL R1, R0, R1  ;
 ADD R1, R1, #512
 LSR R0, R1, #10 ;
 LDR R2, =M      ; I
to M)
 STR R0, [R2]    ; M
```

R1=N

R0=3003

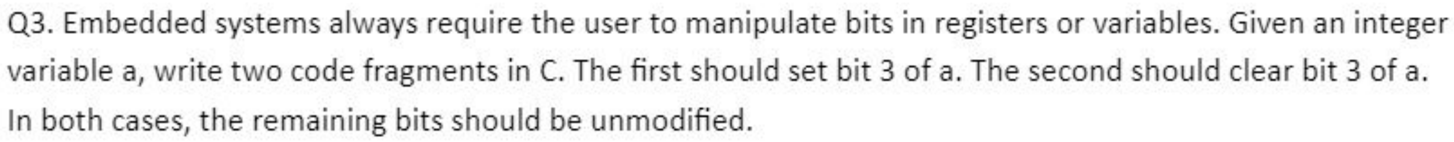
N*3003

N*3003+512

(N*3003+512)>>10   (N*3003+512)/1024

M=(N*3003+512)>>10

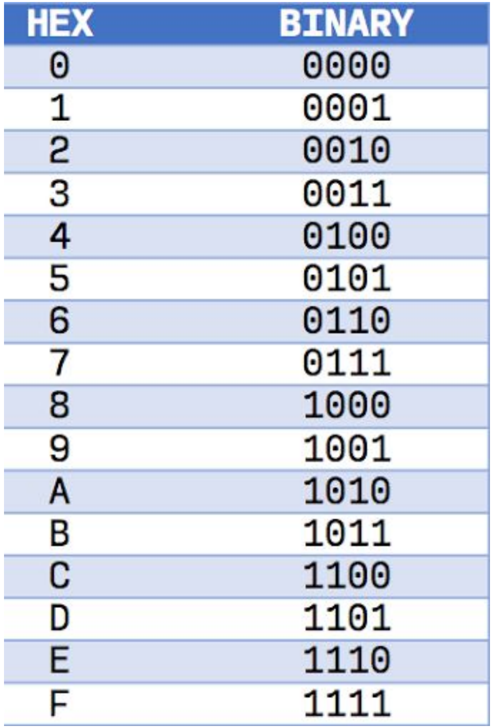
Q3



Q3. Embedded systems always require the user to manipulate bits in registers or variables. Given an integer variable a, write two code fragments in C. The first should set bit 3 of a. The second should clear bit 3 of a. In both cases, the remaining bits should be unmodified.

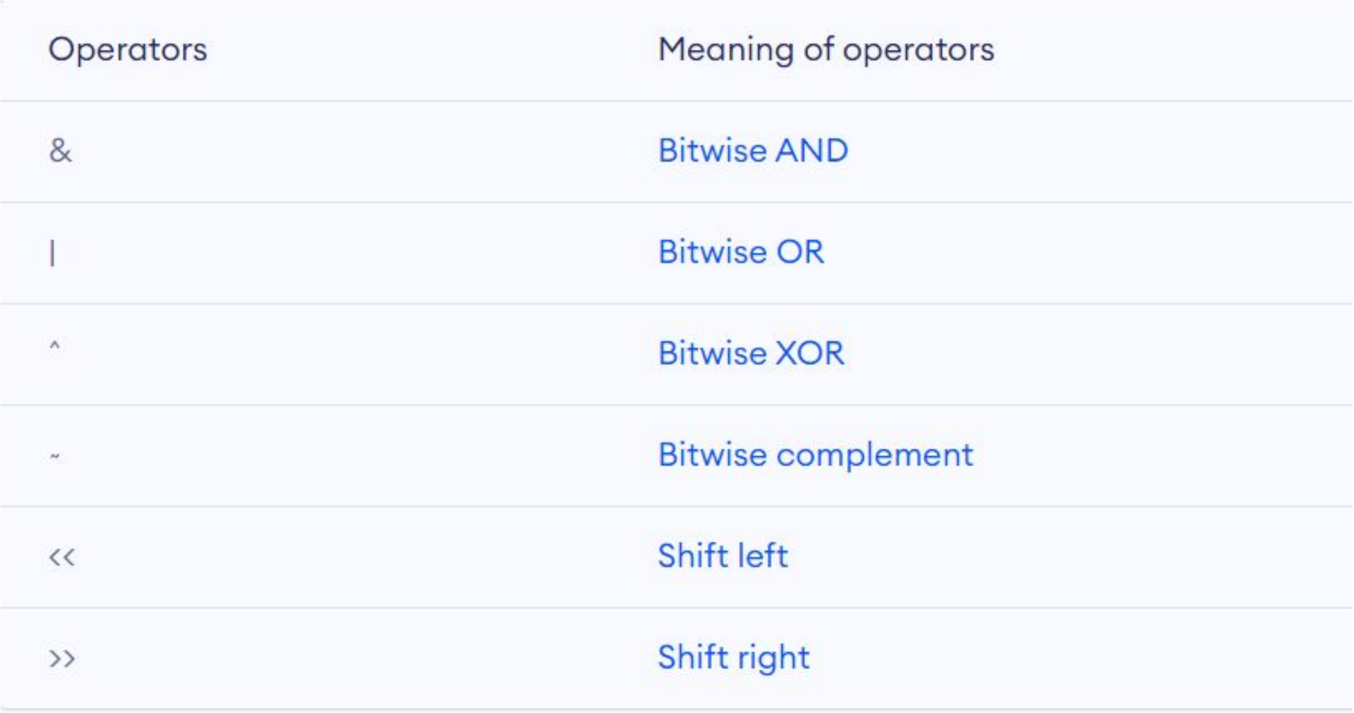
0	0	0	0	1	1	0	1	0	0	1	1	1	0	0	1	1	0	1	1

Q3



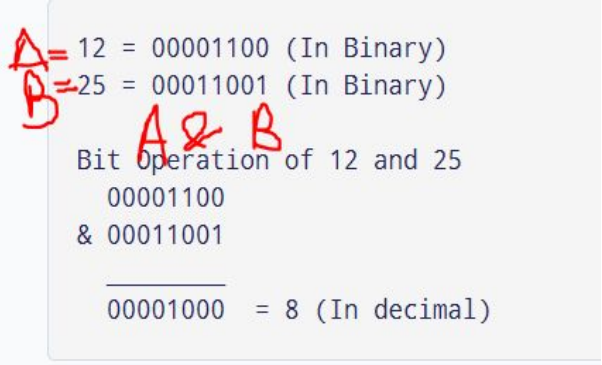
HEX	BINARY
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Bitwise operations

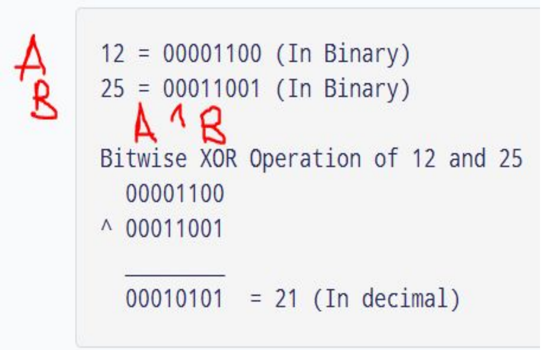


Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise complement
<<	Shift left
>>	Shift right

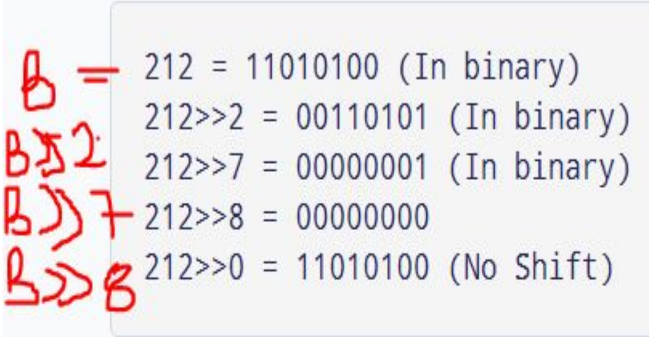
Bitwise operations



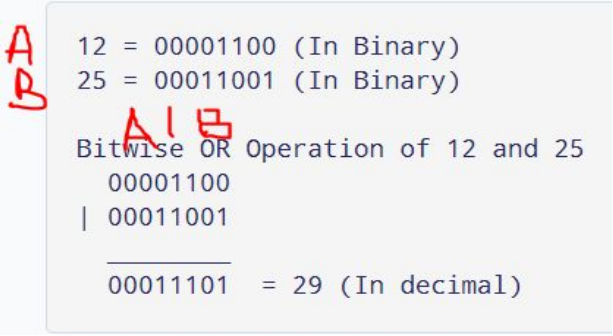
```
A= 12 = 00001100 (In Binary)
B= 25 = 00011001 (In Binary)
        A & B
 Bit Operation of 12 and 25
   00001100
 & 00011001
   ________
   00001000  = 8 (In decimal)
```



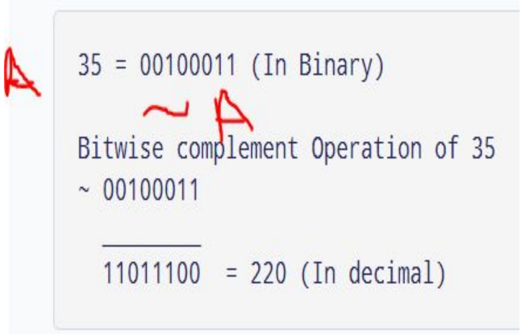
```
A  12 = 00001100 (In Binary)
B  25 = 00011001 (In Binary)
        A ^ B
   Bitwise XOR Operation of 12 and 25
     00001100
   ^ 00011001
     ________
     00010101  = 21 (In decimal)
```



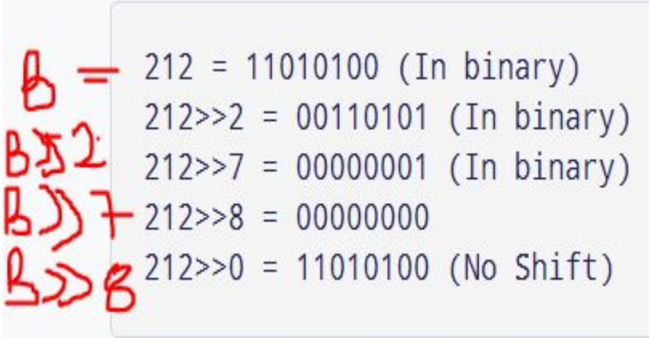
```
B =    212 = 11010100 (In binary)
B>>2   212>>2 = 00110101 (In binary)
       212>>7 = 00000001 (In binary)
B>>7   212>>8 = 00000000
B>>8   212>>0 = 11010100 (No Shift)
```



```
A  12 = 00001100 (In Binary)
B  25 = 00011001 (In Binary)
      A | B
 Bitwise OR Operation of 12 and 25
    00001100
  | 00011001
    ________
    00011101  = 29 (In decimal)
```

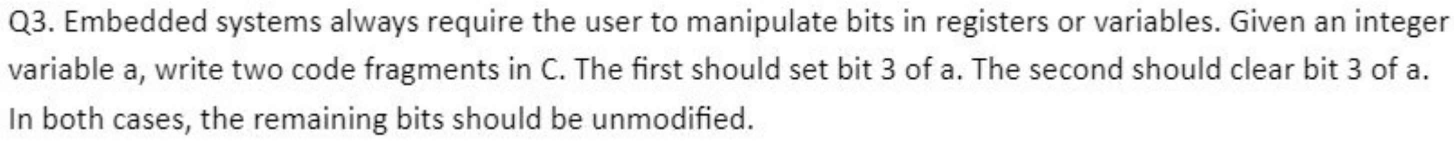


```
A  35 = 00100011 (In Binary)
        ~ A
   Bitwise complement Operation of 35
   ~ 00100011
     ________
     11011100  = 220 (In decimal)
```



```
B =    212 = 11010100 (In binary)
B>>2   212>>2 = 00110101 (In binary)
       212>>7 = 00000001 (In binary)
B>>7   212>>8 = 00000000
B>>8   212>>0 = 11010100 (No Shift)
```

Q3



Q3. Embedded systems always require the user to manipulate bits in registers or variables. Given an integer variable a, write two code fragments in C. The first should set bit 3 of a. The second should clear bit 3 of a. In both cases, the remaining bits should be unmodified.

a= a|?

a = a | 0000 0001

a= a|1

a= a | 0000 1000

a = a| (1<<3)

a|= 1<<3

a= a & 0 ✗

a=a&0000 0000

a=a&1111 0111

a= a &1

a= a& 0000 0001

a = a& (1<<3)

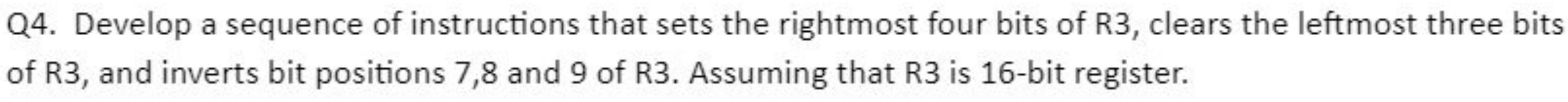
a= a &0000 1000

a= a & (~(1 <<3))

a= a& 1111 0111

a & = ~(1<<3)

Q4



Q4. Develop a sequence of instructions that sets the rightmost four bits of R3, clears the leftmost three bits of R3, and inverts bit positions 7,8 and 9 of R3. Assuming that R3 is 16-bit register.

:

```
ORR R3, R3, #0x000F
AND R3, R3, #0x1FFF
EOR R3, R3, #0x0380
```

R3= 1101 0010 1100 1000

R3&= 0001 1111 1111 1111

R3 | = 0000 0000 0000 1111 = 1101 0010 1100 1111

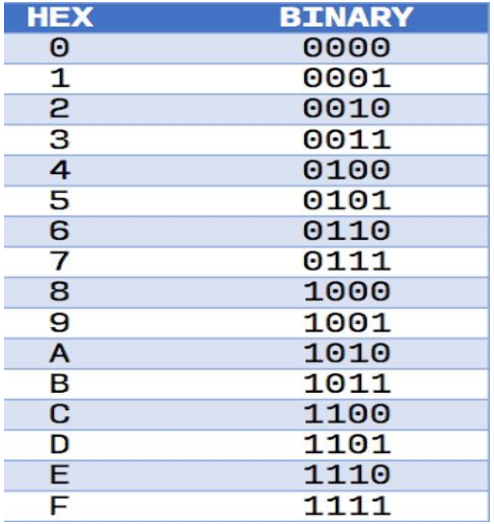
0X000F

0^1 =1

1^1= 0

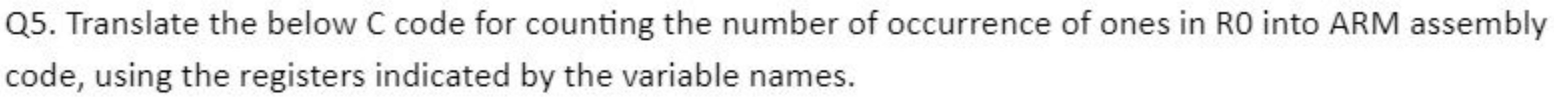
R3 ^= 0000 0011 1000 0000 = 1101 0001 0100 1000

0X0380

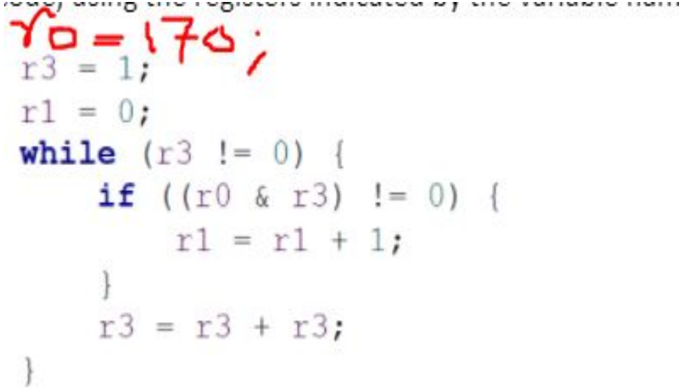


HEX	BINARY
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

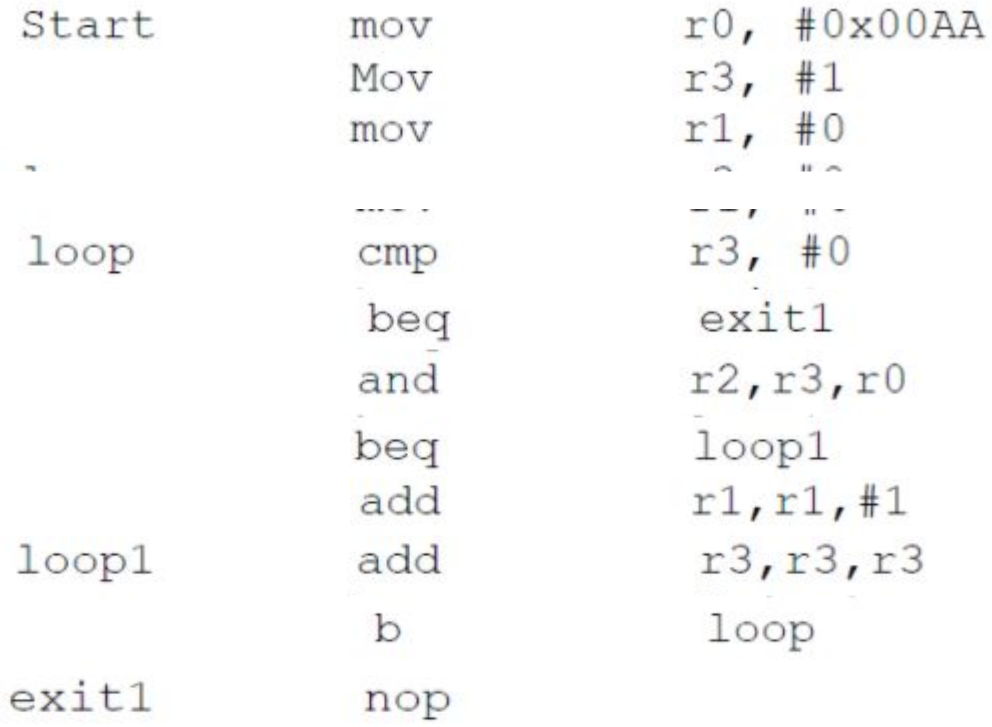
Q5



Q5. Translate the below C code for counting the number of occurrence of ones in R0 into ARM assembly code, using the registers indicated by the variable names.



```
r0 = 170;
r3 = 1;
r1 = 0;
while (r3 != 0) {
    if ((r0 & r3) != 0) {
        r1 = r1 + 1;
    }
    r3 = r3 + r3;
}
```

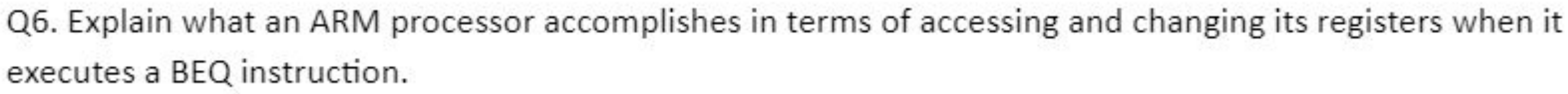


```
Start   mov   r0, #0x00AA
        Mov   r3, #1
        mov   r1, #0
loop    cmp   r3, #0
        beq   exit1
        and   r2,r3,r0
        beq   loop1
        add   r1,r1,#1
loop1   add   r3,r3,r3
        b     loop
exit1   nop
```

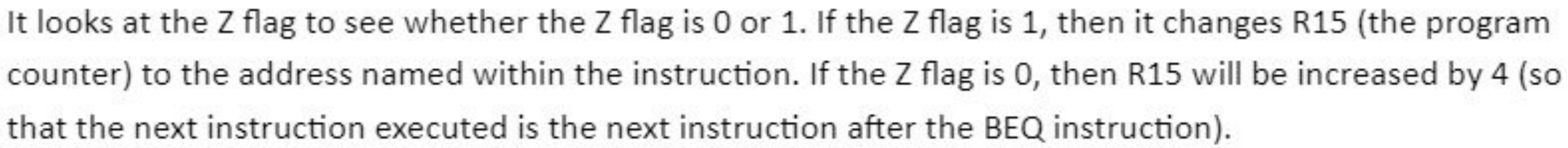
algorithm

```
x = 170;
x= 10101011;
Counter =0;
loop(until mask equal zero)

{

     value = x anding with mask    x & 0000 0010=000000010
    Check (value)   increase counter :  not increase counter
     x= 1+1=2 (to shift the mask)

}
```

Q6

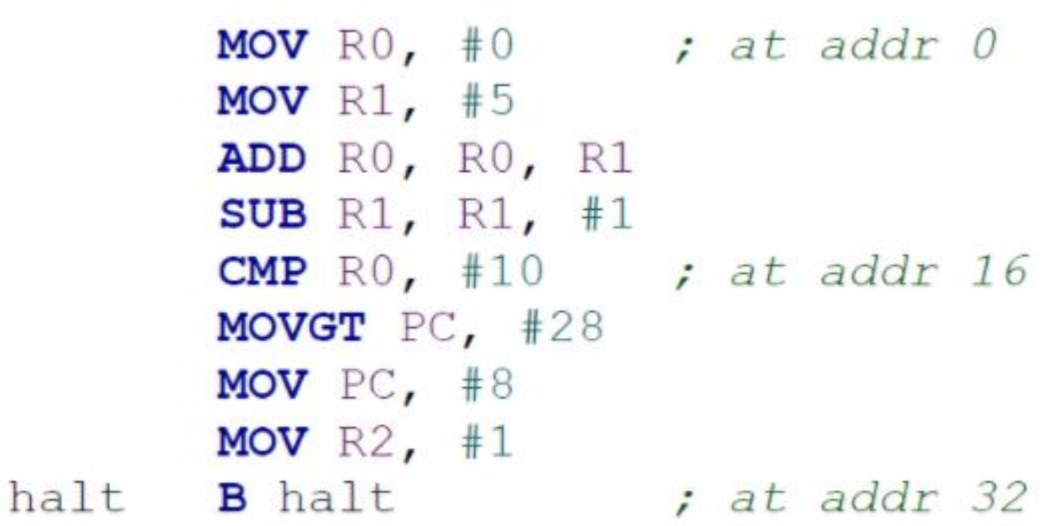


Q6. Explain what an ARM processor accomplishes in terms of accessing and changing its registers when it executes a BEQ instruction.



It looks at the Z flag to see whether the Z flag is 0 or 1. If the Z flag is 1, then it changes R15 (the program counter) to the address named within the instruction. If the Z flag is 0, then R15 will be increased by 4 (so that the next instruction executed is the next instruction after the BEQ instruction).

Q6



```
        MOV R0, #0          ; at addr 0
        MOV R1, #5
        ADD R0, R0, R1
        SUB R1, R1, #1
        CMP R0, #10         ; at addr 16
        MOVGT PC, #28
        MOV PC, #8
        MOV R2, #1
halt    B halt              ; at addr 32
```

PC	0	4	8	12	16	20	24	8	12	16	20	24	8	12	16	20	28	32	
RO	0		5					9					12						
R1		5		4					3					2					
R2																	1		