Branch Instructions

In textbook, 4 are given.
among which 1st 2 are for 32-bit ARM7
& next two are for 16-bit Thumb instructions
For our syllabus only 1st two.

1. B : Branch : pc = label
2. BL : Branch with link : pc = label (pc = program counter)
   lr = address of the next instruction after BL.
   (lr ↗ link register)

Load and store Instructions

~~1. LDR~~

* LDR is used to read complete 32-bit data into a register. (Load into register from memory)
  Store a register value into memory.
  Def^n of word depends on the architecture of the computer.
  In a 16-bit microprocessor, word = 16 bit
  In a 32-bit architecture, word = 32-bit.
  At a time how much it can read/write is word.

* Note:

1. LDR : Load word (32-bit) into a register
   ~~(word)~~
2. LDRH : Load half-word (16-bit) into a register
3. LDRB : Load a byte (8-bit) into a register
4. STR : store 32-bit
5. STRH : store 16-bit
6. STRB : store 8-bit

In case of Unsigned

If I store 32-bit data in a 32-bit wide register then all bits will be having data.
But less than 32-bit data if I store, then remaining bits of 32-bit wide register are padded with zero's.

In case of Signed

If less than 32-bit data we store in 32-bit wide register then, remaining bits will be padded with:

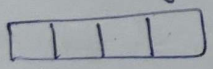
- zero's if it's a positive number &
- one's if it's a negative number.

Program:- (on load & store instructions.)

In the program, we are suppose to load, the starting address of FBLOCK (1st memory block) & SBLOCK (2nd memory block) into R0 & R1.

(open memory windows)

* In order to store 32-bit data, you require 4 ~~blocks~~ memory locations in 1 block.



* index it by 4, if you're reading complete 32 bit data
" " " 2, " " " only 16 " "
" " " 1 only 8 " "

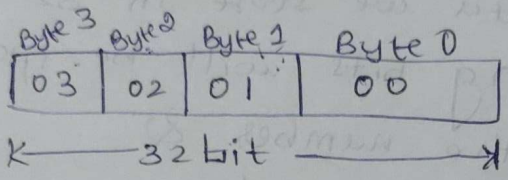
eg- LDR R3, [R0], #4
STR R3, [R1], #4
} we're loading complete 32-bit data & storing complete 32-bit data.

* DCW 0 means initialize memory location with zero's bcz I want to store some width data in it. There should n't be any garbage value.

* LDRH ... #2
  STRH ... #2
  } for 16-bit → eg:- LOOP LDRH R3, [R0], #2
  STRH R3, [R4], #2
  FBLOCK DCW 0x1234, 0x9ABC, 0x4000, 0xC000
  SBLOCK DCW 0

* LDRB ... #1
  STRB ... #1
  } for 8-bit → eg:- LOOP LDRB R3, [R0], #1
  STRB R3, [R1], #1
  FBLOCK DCB 0x12, 0x9A, 0x43, 0xC8
  SBLOCK DCB 0 (16-bit)

* **LDRSH:** load signed half word → fetch it with sign
  LDRSB: load signed byte → (32-bit)



PROGRAM:-

```
        AREA    LDRpgm, CODE, READONLY
        ENTRY
        MOV     R4, #4
        LDR     R0, =FBLOCK
        LDR     R1, =SBLOCK
Loop    LDR     R3, [R0], #4 ; we're loading complete 32-bit data into the register from memory
        STR     R3, [R1], #4 ; we're storing register value into memory
        SUBS    R4, #1 ; Subtract 1 from R4, updating flags
        BNE     LOOP ; Branch to "loop" if the result is not zero (ie. Z flag is not set)
FBLOCK  DCD     0x12345678, 0x9ABCDEF0, 0x00004000, 0x00000C000
L       B       L
        AREA    MYDATA, DATA, READWRITE
SBLOCK  DCD     0 ; I've written this down bcz I wan't SBLOCK to be Read write memory. I wrote FBLOCK above only bcz, FBLOCK, can't just a Read only memory, as I don't write anything in FBLOCK & I'm just using FBLOCK to read data
        END
```

we index it by 4 as we are reading complete 32-bit data.

FBLOCK is an array of four 32-bit words, each initialized with a hexadecimal value.

Subtract 1 from the value in register R4. The result is not stored in R4 but the flags are updated based on this subtraction.

Read only memory

Define constant Doubleword
It is a directive used to allocate space in memory & initialize it with values.

Read write memory

When R4=0, Z flag is set to 1 & is compared with 1

If Z==1 then it comes out of loop.

Scanned with ACE Scanner

Extra page for tracing of key points

Note that you use

DCD when it is (32-bit data) → eg:- 0x12345678

DCW " " " (16-bit data) → eg:- 0x1234

and DCB " " " (8-bit data) → eg:- 0x12

DCD →

Points:

- LDR & STR instructions can load and store data on a boundary alignment that is the same as the datatype size being loaded or stored. 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. This example shows a load from a memory address contained in register r1, followed by a store back to the same address in memory.

```
; load register r0 with the contents of the memory address pointed
; to by register r1.
    LDR   r0, [r1]          ; = LDR r0, [r1, #0]
; ← It's just a comment
; store the contents of register r0 to the memory address
; pointed to by register r1.
    STR   r0, [r1]          ; = STR r0, [r1, #0]
```

Here, Register r1 is called the base address register.

Swap instruction

Exchanging the content of memory with the register content.

1st two operands are registers, next operand is memory location

SWP r0, r1, [r2]

These two are registers; [r2] is memory location

8 digit represents 32-bit number.

Note :-

The swap instruction is a special case of load/store instruction. It swaps the content of memory with the contents of register.

This instruction is an atomic operation ie. it reads & writes a location in the same bus operation preventing any other instruction from reading or writing to that location until it completes.

(> means in OS, I'll lock that memory location until one process completes writing data to it. multiple processes can read simultaneously but can't write simultaneously

Swap can't be interrupted by any other instruction or any other bus access. We say that the system holds the bus until the transaction is complete.

eg:- PRE:

mem32[0x9000] = 0x12345678

r_0 = 0x00000000

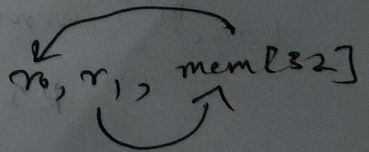
r_1 = 0x11112222

POST:

mem32[0x9000] = 0x11112222

r_0 = 0x12345678

r_1 = 0x11112222



SWP Rd, Rm, [Rn]

SWP r0, r1, [r2]

[r2] = r0

r1 → [r2]

~~temporary register~~ which I'm using is a memory ~~R4~~

content present in address
↳ which is present in R2
is ~~loaded~~ copied to R4

container temp or tmp

Block 1 (Read only) | Block 2 (Read write)

0x12345678

R2 is pointing to Read only memory

R3 is pointing to Read write memory

I'm loading the content of memory location whose address is in R2.

Through R0 I'm copying from 1 memory location to other.

R0 = 0x12345678

Store the content of R0 into memory location whose address is in R3

Block 2

0x12345678

Now Clear R0 with zeroes.

R0 = 0x00000000

Next, Load any immediate value into R1.

R1 = 0x8000000F

SWP R0, R1, [R3]

R0 becomes my destination register which will hold earlier content of R3 & R1 content is copied into R3.

R1 is overridden on to the memory location

Store cannot be used with ROM. So I'm using another memory location Block2

Programs:- (exchanging the content of memory with the register contents)

```
        AREA    swapMC, CODE, READONLY
        ENTRY
        LDR     R2, =BLOCK1
        LDR     R3, =BLOCK2
        LDR     R4, [R2]
        STR     R4, [R3]
        MOV     R0, #0X00000000
        MOV     R1, #0X10000002 ; Immediate value I've taken
        SWP     R0, R1, [R3]
L       B       L
BLOCK1  DCD     0X12345678 ; Declaring the data
        AREA    MYDATA, DATA, READWRITE
BLOCK2  DCD     0 ; I've written this down bcz
        END
```

ROM is only read only memory. But I want Block2 to be Read write memory.

BLOCK2 (Read only) → Read write

R2 is temporary register

After you write SWP R0, R3, [R3]; in R0 now 0x12345678

In R3 = 0x8000000F is present at last!! Finish.

R0, R1, [R3]

R3 ☺ content is placed in R0

R1 ☺ content is placed in R3

This program is about swapping 32-bit data among memory & register.

Program:-

```
        AREA  swapMC, CODE, READONLY
              ENTRY
              LDR  R2, = BLOCK1
              LDR  R3, = BLOCK2
              LDR  R4, [R2]
              STR  R4, [R3]
              MOV  R0, #00
              MOV  R1, #0X8000000F
              SWP  R0, R1, [R3]
L             B    L
BLOCK1  DCD  0X12345678
        AREA  MYDATA, DATA, READWRITE
BLOCK2  DCD  0
        END
```

SWPB (Swap Byte)

Comple 8-bit data is swapped. 1 byte = 8 bit

Program (about swapping 8-bit data b/w memory & register.)

Register	Value
R0	0x00000012
R1	0x00000005
R2	0x00000020
R3	0x40000000
R4	0x00000012

```
        AREA    swapMC, CODE, READONLY
        ENTRY
        LDR     R2, =BLOCK1
        LDR     R3, =BLOCK2
        LDRB    R4, [R2] ; loading 8-bit
                           data ; 1 byte
                                = 8 bit
        STRB    R4, [R3] ; storing 8-bit data
        MOV     R0, #00
        mov     R1, #05 ; Immediate value
                          I've taken
        SWPB    R0, R1, [R3]
L       B       L
BLOCK1  DCB     0X12 ; Declaring the data
```

It is also 8 bit but it is hexadecimal

```
        AREA    MYDATA, DATA, READWRITE
BLOCK2  DCB     0 ; I've written this down bcz ROM
                    is only read only memory.
        END
```

Zero bez If'll tell the Assembler to reserve memory locations of type DCB (or anything which you specify there) & initialize them with zeroes

This line DCB 0

BLOCK2 DCB 0 (> This line will tell the Assembler to reselve memory locations of type DCB (you can write whichever you want acc. to your program) & initialize them with zeroes (you can initialize with any number).

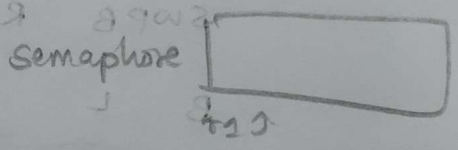
You can write zero or one or anything. It is written so that garbage value present earlier is removed & you can store fresh value in it.

9. Develop an assembly language program to implement semaphores & mutual exclusion in operating system.

⇒ This example shows a simple dataguard that can be used to protect data from being written by another task. The swp instruction "holds the bus" until the transaction is complete.

```
spin
        MOV     r1, =semaphore
        MOV     r2, #1
        SWP     r3, r2, [r1]      ; hold the bus until complete
        CMP     r3, #1
        BEQ     spin
```

Tracing:-

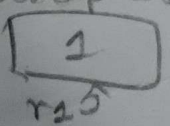


← this is a memory block

r2 = [1] ← register

r3, r2, [r1]

r3 = [1] maybe



So,

CMP r3, #1
Compare r3 with 1

Yes.

Then BEQ spin

to BEQ

Branch if equal to.

If r3 == 1 then spin here only!!

Loading constants

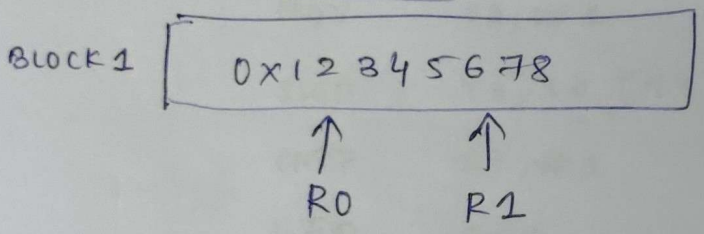
LDR Rd = constant value

ADR Rd = memory address

LDR is used to hold (or fetch) the starting address of the memory location as well as to hold/fetch a constant value in the memory location.

ADR is used to hold only memory address.

Our program:



Both are pointing to same memory location
ie are holding starting address of BLOCK1

Now copy content of memory location pointed by R0 into R2 & that of R1 into R3.

So, R2 & R3 will have the same content

Program:

Question :- Write content of R0, R1, R2 & R3 (R0, R1 will have same memory address; R2 & R3 will have same content)

```
        AREA  MS, CODE, READONLY
        ENTRY
        LDR   R0,=BLOCK1
        ADR   R1, BLOCK1
        LDR   R2, [R0]
        LDR   R3, [R1]
BLOCK1  DCD   0X12345678
L       B     L
        END
```

* We've 7 modes.
* you know what there is user mode & supervisor mode.
* Only if you are in supervisor mode, you can modify the content of CPSR
  ↳ Current program status register.
* MRS Rd, CPSR → Syntax of MRS
  0X000000D3 → value of user mode with no interrupts
  ↳ అలాంటి program start అయినప్పుడు CPSR లో ఈ value వుంటుంది.
  In our program it is loaded into R1

```
        AREA  MG, CODE, READONLY
        ENTRY
        MRS   R1, CPSR
L       B     L
        END
```

Program:-

R0 = 0X C0000003

1100 0000 0000 0000 0000 0000 0000 0011

R_1 = 0XE0000001

1110 0000 0000 0000 0000 0000 0000 0001

Rotate Right the content of R0 1 time

MOVS R2, R0, ROR #1

1100 0000 0000 0000 0000 0000 0000 0011 → C = 1

110 0000 0000 0000 0000 0000 0000 0001

MSB bit is one

So, N = 1

indicates MSB bit

So, N = 1
Z = 0
C = 1
V = 0

Last bit shifted out of Barrel shifter is one. here C = 1

Whatever we've rotated is available in our destination register (R_2)

1110 0000 0000 0000 0000 0000 0000 0001

↳ E 0 0 0 0 0 0 1

So R_2 = E0000001

CMP R2, R1

(you are comparing both numbers.

Both are equal.

So Z=1 If they are equal then Z=1

C=1 bcz it is retaining the last bit shifted out of the barrel shifter.

CMP R2, R1

means R2 − R1

R2 = E0000001

R1 = E0000001

On Subtracting, result=zero

As result is zero zero flag is set to 1.

Program :-

Register	Value
R0	0xC0000003
R1	0xE0000001
R2	0xE0000001

loadingconstants.s

```
1     AREA loadingconstants, CODE, READONLY
2     ENTRY
3     MOV R0, #0XC0000003 ;R0=?
4     MOV R1,#0XE0000001  ; R1 =?
5     MOVS R2,R0, ROR #1 ;R2=? R0=? STATUS of NZCV=?
6     CMP R2,R1 ; R2=? R1=? status of NZCV=?
7  L  B L
8     END
```

CPSR

N	0
Z	1
C	1
V	0

Program of Termwork 4

```
1   AREA MCTW4, CODE, READONLY
2       MOV  R6, #6    ; COUNT
3       LDR  R1, =VALUE  ; ARRAY
4       LDR  R8, =RESULT ; RESULT MEMORY
5
6       LDR  R2,[R1],#4 ; 1st number into R2
7   LOOP LDR R4,[R1],#4 ; 2nd number into R4
8       CMP  R2,R4
9       BHI  LOOP1 ; IF R2>R4
10      MOV  R2,R4 ; COPY R4 INTO R2
11  LOOP1 SUBS R6,R6,#1  ; R6 = R6-1
12      BNE  LOOP
13      STR  R2,[R8] ; COPY LARGEST NUMBER PRESENT IN R2 INTO
                                                        R8
14
15  VALUE DCD 0x11111111, 0x44444444, 0x22222222, 0x33333333,
          0x55555555, 0x77777777, 0x66666666
16  L       B  L
17      AREA MYDATA, DATA, READWRITE
18  RESULT  DCD 0
19      END
```

Tw4

Registers

Register	Value
R0	0x00000000
R1	0x00000044
R2	0x77777777
R3	0x00000000
R4	0x77777777
R5	0x00000000
R6	~~0xFFFFFFFB~~ → 0xFFFFFFFB
R7	0x00000000
R8	0x40000000

R14(LR) → 0x00000044 & R15 (PC) → 0x00000010

Memory 2

Address: 0x40000000

0x40000000: 77 77 77 77

Memory 1

Address: 0x0000002C

0x0000002C: 11 11 11 11 44 44 44 44 22 22 22 22
0x00000038: 33 33 33 33 55 55 55 55 77 77 77 77
0x00000044: 66 66 66 66 FE FF FF EA 2C 00 00 00
0x00000050: 00 00 0040 00 00 0000 0000 0000

Practice Program :-

PRACTICE.s

```
        AREA PRACTICE, CODE, READONLY
        ENTRY
        MOV  R0, # 0XF0000007
        MOV  R1, # 0XE0000008
        AND  R2, R0, R1
        ORR  R3, R0, R1
        EOR  R4, R0, R1
        TEQ  R4, #00
L       B    L
        END
```

Register	Value
R0	0xF0000007
R1	0xE0000008
R2	0xE0000000
R3	0xF000000F
R4	0x1000000F

TW 4

Q. Develop an assembly language program to find the largest number from an array and store the result in RAM location. → In exam they may ask smallest.

Explaination:

* Given n elements, you compare 2 at a time. So no. of comparisons = n-1.
counter register → till (n-1)

* CMP R_2, R_4 means $R_2 - R_4$ subtraction

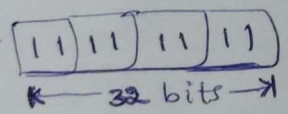
* BHI if first no > second no. then proceed with subtraction operation. else interchange 1st & 2nd no.

* Assembly director DCD → to 32-bit data we are using.

0x11111111	0x22222222	0x33333333	0x44444444	0x55555555	0x66666666	0x77777777

R1 → will hold address

* SUBS $R_6, R_6, \#1$ means I will decrement R_6 by 1 & store it again in R_6 (indicating that 1 comparison got over)

* 

* At last R2 is holding largest number then paste this content in the memory location whose address is stored in R8.

* BLS for finding smallest numbers & BHI for largest numbers. You just change this to BLS & run your program. You'll get 11111111 → is smallest number.

TW 6 (on Sorting)

tion:-

gram:-

```
        AREA MCTW6,CODE,READONLY
        ENTRY
        MOV R8,#4 ; Counter register for array
        LDR R1,=CVALUE ; R1 points to the array present in CODE region
        LDR R2,=DVALUE ; R2 points to the array present in DATA region
LOOPO   LDR R3,[R1],#4 ; Fetch a number from first array
        STR R3,[R2],#4 ; paste it in 2nd array
        SUBS R8,R8,#1 ; Decrement counter register
        CMP R8,#0 ; Compare R8 with zero
        BNE LOOPO ; Once R8 becomes zero it goes to LOOPO.
                  If R8!=0, you continue data transfer
                  Once R8=0, come out of loop.
START   MOV R7,#0 ;Swap flag is cleared. ~~If there are~~
                  ; If there are n-elements, then number of comparisons = n-1. Let me take one register (say R5) to hold the no. of comparisons.
        MOV R5,#3 ; Comparison COUNT.
        LDR R2,=DVALUE
LOOP2   LDR R1,[R2],#4 ; LDR R1, content of R2 (means) ; 1st number in R1
        LDR R3,[R2] ; 2nd number in R3
        CMP R1,R3 ; R1-R3 will happen, then make use of branch instruction
        BLT LOOP1 ; BLT means Branch if Lesser Than. If you don't want to exchange (i.e., swap) then branch to LOOP1.

;else if you want to swap the two numbers then perform the below 4 steps. In these 4, the 1st two steps actually perform swapping.
```

```
        STR  R1, [R2], #-4
        STR  R3, [R2]
        MOV  R7, #1 ; SWAP = 1 ; flag to indicate that swapping has occured.
        ADD  R2, #4 ; Restore Pointer
LOOP1   SUBS R5, R5, #1
        CMP  R5, #0
        BNE  LOOP2
        CMP  R7, #0
        BNE  START
        B    L
L
CVALUE  DCD  0X44444444, 0X11111111, 0X33333333, 0X22222222
        AREA MYDATA, DATA, READONLY
DVALUE  DCD  0
        END
```

Output

Registers

Register	Value
R0	0X00000000
R1	0X33333333
R2	0X4000000C
R3	0X44444444
R4	0X00000000
R5	0X00000000
R6	0X00000000
R7	0X00000000
R15(PC)	0X00000054

Memory1

Address: 0X00000064

0X00000064: 44 44 44 44 11 11 11 11 33 33 33 33 22 22 22 22 64 00 00 00 00 00 40

Address: 0X40000000

0X40000000: 11 11 11 11 22 22 22 22 33 33 33 33 44 44 44 44

Efficient C Programming

8051 microcontroller was of 8 bit. So we used char datatype as far as possible.

But in Efficient C programming,

1. avoid char (except modular arithmetic), while declaring local variables.
   (If you want wrap-around to occur. If you require modulo arithmetic of the form 255+1 = 0, then use char type.)
2. use int & long for local variables.

1. char	8-bit	(unsigned)	0 to 255 means the variable can store only +ve values.
2. short	16-bit	(signed)	
3. int	32-bit	(signed)	
4. long	32-bit	(signed)	
5. long long	64-bit	(signed)	

* Keep in mind that by default char is unsigned. ie not signed

We'll look at many eg. to show how the compiler translates C source to ARM assembler. Once you have a feel for this translation process, you can distinguish fast C code from slow C code.

* To keep our examples concrete, we have tested them using the following specific C compilers:

- armcc for ARM Developer suite version 1.1 (ADS1.1). You can license this compiler, or a later version, directly from ARM.
- arm-elf-gcc version 2.95.2. This is the ARM target for the GNU C compiler, gcc and is freely available.

BASIC C DATA TYPES

Let's start by looking at how ARM compilers handle the basic C data types. We'll see that some of these types are more efficient to use for local variables than others. There are also differences b/w the addressing modes available when loading & storing data of each type.

ARM processors have 32-bit registers & 32-bit data processing operations. The ARM architecture is a RISC load/store architecture.

In other words you must load values from memory into registers before acting on them. There are no arithmetic or logical instructions that manipulate values in memory directly.

Early versions of the ARM architecture (ARMLv1 to ARM v3) provided hardware support for loading & storing unsigned 8-bit & unsigned or signed 32-bit values.

Table 5.1 Load & store instructions by ARM architecture.

Architecture	Instruction	Action
Pre-ARMV4	LDRB	load an unsigned 8-bit value
	STRB	store a signed or unsigned 8-bit value
	LDR	load a signed or unsigned 32-bit value
	STR	store a signed or unsigned 32-bit value
ARMV4	LDRSB	load a signed 8-bit value
	LDRH	load an unsigned 16-bit value
	LDRSH	load a signed 16-bit value.
	STRH	store a signed or unsigned 16-bit value
ARMV5	LDRD	load a signed or unsigned 64-bit value.
	STRD	store a signed or unsigned 64-bit value.

These architectures we used on processors prior to the ARM7TDMI. Table 5.1 shows the load/store instruction classes available by ARM architecture.

In Table 5.1, loads that act on 8- or 16-bit values extend the value to 32 bits before writing to an ARM register. Unsigned values are zero-extended, and signed values sign-extended. (Similarly, a store of an 8-) This means that the cast of a loaded value to an int type does not cost extra instructions. Similarly, a store of an 8- or 16-bit value selects the lowest 8 or 16 bits of the register. The cast of an int to smaller type does not cost extra instructions on a store.

The ARMv4 architecture & above support signed 8-bit & 16-bit loads & stores directly, through new instructions. Since these instructions are a later addition, they do not support as many addressing modes as the pre-ARMv4 instructions. We'll see the effect of this in the eg. checksum-v3 in section 5.2.1.

Finally, ARMv5 adds instruction support for 64-bit load & stores. This is available in ARM9E & later cores.

Prior to ARMv4, ARM processors were not good at handling signed 8-bit or any 16-bit values. Therefore ARM C compilers define char to be an unsigned 8-bit value, rather than a signed 8-bit value as is typical in many other compilers.

Compilers armcc & gcc use the datatype mappings in Table 5.2 for an ARM target. The exceptional case for type char is worth nothing as it can cause problems when you're porting code from another processor architecture. A common eg. is using a char type variable i as a loop counter, with loop continuation condition $i \geq 0$. As i is unsigned for the ARM compilers the loop will never terminate. Fortunately armcc produces a warning in this situation: unsigned comparison with 0. Compilers also provide an override switch to make char signed. For eg., the command line option -fsigned-char will make char signed on gcc. The command line option -zc will have the same effect with armcc.

For the rest of this book we assume that you're using an ARMv4 processor or above. This includes ARM7TDMI & all later processors.

Table 5.2 C compiler datatype mappings.

C Data Type	Implementation
char	unsigned 8-bit byte
short	signed 16-bit halfword
int	signed 32-bit word
long	signed 32-bit word
long long	signed 64-bit double word.

To see the effect of local variable types, let's consider a simple eg. We'll look in detail at a checksum function that sums the values in a data packet. Most communication protocols (such as TCP/IP) have a checksum or cyclic redundancy check (CRC) routine to check for errors in a data packet.

The following code checksums a data packet containing 64 words. It shows why you should avoid using char for local variables.

```
int checksum_v1 (int *data)        // represents version
{                                  // (>pointer to array.
  char i;                          // array name is data.
  for (i=0; i<64; i++)             // 0 to 255 is range of char.
  {                                // I'm adding 64 numbers
    sum += data[i];
  }
  return sum;
}
```

let's see how compiler is going to produce assembler code for this.⇒ we've added labels & comments to make the assembly clear.

```
checksum_v1
    MOV  r2, r0   ; r2 = data   loading starting address of array whose name is data in register
    MOV  r0, #0   ; sum = 0     initialize sum to zero.
    MOV  r1, #0   ; i = 0  → for iterating.
checksum_v1_loop
                  → r1 is leftshifted 2 times and is stored in r2
    LDR  r3, [r2, r1, LSL #2]  ; r3 = data[i]
    ADD  r1, r1, #1            ; r1 = i+1  incrementing i   (i.e. i = i+1)
    AND  r1, r1, #0XFF         ; i = (char) r1
    CMP  r1, #0X40             ; Compare i, 64
    ADD  r0, r3, r0            ; sum += r3 → sum = sum + data[i]
    BCC  checksum_v1_loop      ; if i<64 loop
    MOV  pc, r14               ; return sum
```

extra instruction to fetch 8-bit data

you're trying to fetch index value.

Branch if carry is cleared ↑ If carry is cleared then you've to loop it

↳ means restore the content of program counter
↳ means where I should go back to my main program.

For checksum, we've to discard the carry & compute 2's compliment for that.

Import points while tracing the above program.

* shifting any data logically left is nothing but multiplication. (by 2)

Here, we are shifting 2 times.

index values 0, 3, 8, ...

1) 0000 zero 2) 0000 0001
out ← 0000 zero×2 = zero → 0000 0010 = 0x2
out ← 0000 zero×2 = zero → 0000 0100 = 0x4

data 0x 2222 2222

0x22	0x22	0x22	0x22
0x33	0x33	0x33	0x33

AND r1, r1, #0XFF r1 is holding value of i. datatype of i is char. → 8-bit wide

r1 = 0x24

00 00 00 24 → 32 bit data

8-bit data

I'm interested only in this last 8-bit

Compiler will AND it with 0XFF so that it'll extract only last 8-bit data.

#0x40 : 40 in hexadecimal is 64 in decimal.

$4 \times 16^1 + 0 \times 16^0 = 64 + 0 = 64$

* Is Assembly language program case sensitive?
→ No.

* If you declare the local variable by int then you can remove the additional instruction AND. If you make use of char to declare the local variable, you have to make use of additional instruction AND.

* our registers are of 32-bit. → So, in v1 program, we typecasted/converted 8-bit data to 32-bit & then again to 8-bit.

* when you logically shift left the content by 1, it is multiplied by 2. when you logically shift left the content by 2, it is multiplied by 4. when we are accessing 16-bit data, we post index it by 2. bcz at a time two memory locations I access each of 4 bytes.

Now, lets see version-2 of the checksum program where we remove the additional instruction AND by declaring the local variable as int instead of char.

* when I use char, by default it is unsigned. But int is signed by default. As we are using unsigned integers explicitly, we have to declare as unsigned.

```
int checksum_v2(int *data)
{
  unsigned int i;   # take index value as unsigned.
                    ↳ bcz it can have values from 0 to 64
  unsigned int sum=0;
                    ↳ index only the values. it takes.
  for(i=0; i<64; i++)
  {
    sum += data[i];
  }
  return sum;
}
```

It can add either the or -ve values. So, we didn't declare it as unsigned.

Assembly language program

checksum_v2

```
MOV   R2, R0    ; R2 = data
MOV   R0, #0    ; sum = 0
MOV   R1, #0    ; i = 0
```

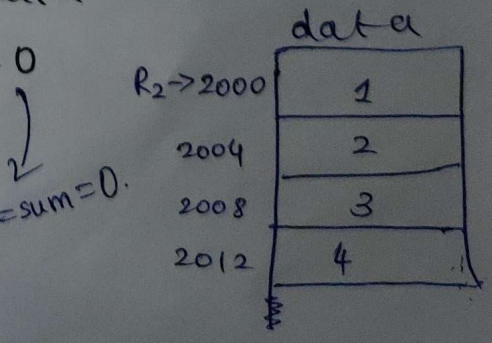
→ R2 is holding base address of array.

base address, sum & i I'm initializing.

P.T.O

R0 = sum = 0.

R1 = i = 0



```
checksum_v2_loop
    LDR   R3, [R2, R1, LSL #2]   ; r3 = data[i]
    ADD   R1, R1, #1             ; i = i+1
    CMP   R1, #0x40              ; compare i, 64
    ADD   R0, R3, R0             ; sum = R0 + R3   (sum = sum + R3)
                                 (R0) (sum)
    BCC   checksum_v2_loop       ; if (i<64) goto loop
    MOV   pc, r14                ; return sum
```

pc → program counter; r14 → link register

LDR R3, [R2, R1, LSL #2] → R3 is loaded with content present in this address.

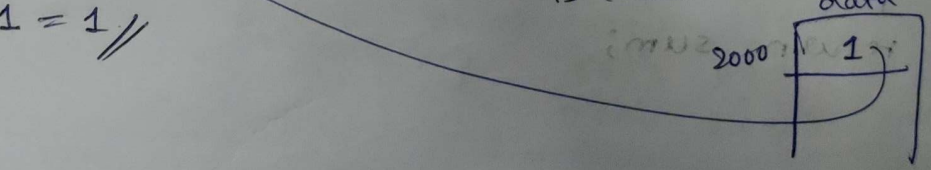
R1 = 0 LSL #2 = 0

When you left shift zero twice, the result is zero only.

R2 = 2000 + 0 = 2000

R3 = 1 (∵ LDR R3, [2000])

Content present in 2000 is loaded into R3.



Sum = 0 + 1 = 1 //

r14 only I use for pc → program counter.

link register will hold return address (after words it should return here from any subroutine)

r14 → lr → where ARM processor puts the return address whenever it calls a subroutine (ie function)

r15 → pc → contains the address of the next instruction to be fetched by the processor.

0000 0001

0000 0010 → LSL one time

0000 0100 → LSL another time

this value is 4

R3 = [2000 + 4]

= [2004] ← content present in this address.

= 2 ← content of 2004 is 2.

2000	1
2004	2

Space to write tracing neatly.

If I'm interested (→ to extract) in 16-bit data, I can use short.
The containers (ie registers) are of 32-bit.
If you are interested in extracting 32-bit or 16-bit or 8-bit data out of that, you should decide.

```
short checksum_v3(short *data)
{
  unsigned int i;
  short sum=0;
  for(i=0; i<64; i++)
  {
    sum = (short)(sum+data[i]);
  }
  return sum;
}
```

Assembly language program for the above C pgm.

```
checksum_v3
    MOV  R2,R0   ; R2=data   → initialize R2 with base address of array
    MOV  R0,#0   ; sum=0
    MOV  R1,#0   ; i=0
checksum_v3_loop
    ADD  R3,R2,R1,LSL #1 ; R3=&data[i]   → cannot perform barrel shifting operations in LDRH
    LDRH R3,[R3,#0]      ; R3=data[i]
    ADD  R1,R1,#1        ; i++
    CMP  R1,#0X40
    ADD  R0,R3,R0        ; R0 is sum.  R0=R0+R3  sum=sum+R3
    MOV  R0,R0,LSL #16
    MOV  R0,R0,ASR #16
    BCC  checksum_v3_loop
    MOV  pc,R14
```

LDRH for halfword ie 16-bit
LDRSH for signed half-word.

2000	1
2002	2
2004	3
2008	4

As it is 16-bit, it requires only 2 memory locations.

Space to write tracing of previous assembly level

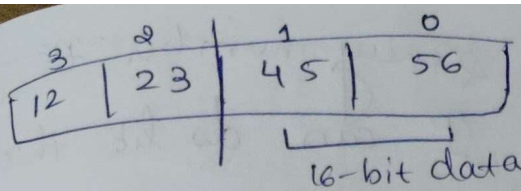
Note:-

1. The LDRH instruction doesn't allow for a shifted address of set, as the LDR instruction did in checksum_v2.

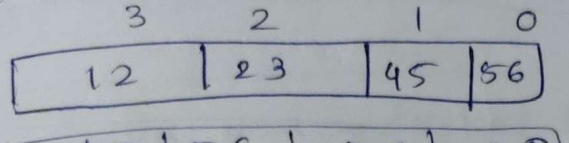
   > LDRH doesn't support shifting operations in the instruction itself. ✓

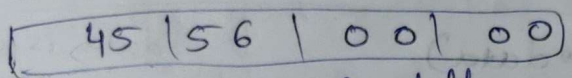
2. Whenever you are reading 16-bit data from a 32-bit number in register, you don't know whether it is signed or unsigned.

P.T.O



If I wan't to remove/vanish upper 16-bit data from the 32-bit data, I do LSL #16
↳ LSL 16 times





← It became now.

But, 45 56 00 00 is different from 45 56.

If I do LSR, my signed bit will be lost.

So, I make use of ASR so that I will retain the signed bit.

upper 16-bit will hold sign extension → FF FF

Lower 16-bit will hold actual value i.e. magnitude → 45 56

} when I do ASR #16

So, finally I got only 45 56 i.e. 16-bit data.

LDR was used however in v2 but in v3 we make use of (3 additional instructions.)

```
ADD  R3, R2, R1, LSL #1
MOV  R0, R0, LSL #16
MOV  R0, R0, ASR #16
```

} v2

Note :-

The cast reducing total + array[i] to a short (ie short data type) requires two MOV instruction. The compiler shifts left by 16 & then Right ~~write~~ by 16 to implement, a 16-bit sign extend.

Instead of converting each & every number to 16-bit in each iteration, I can do like this: Obtain 32-bit sum & from this sum, you obtain only 16-bit data. (Let's see it in version-4)

Version-4

```
short checksum_v4(short *data)                    data is array name
{
  unsigned int i;
  int sum=0;        # values can be +ve or -ve. So I've not declared it as unsigned
  for(i=0; i<64; i++)    # i here is pointer. Pointer is incremented instead of index.
  {
    sum+=*(data+1);      # Major change.
  }
  return (short)sum;     # converting the computed integer sum to short type.
}
```

Assembly language program for the above C program is:

LDRSH for signed half-word

```
checksum_v4
    MOV  R2, #0     ; sum=0
    MOV  R1, #0     ; i=0
checksum_v4_loop
    LDRSH R3, [R0], #2    ; R3=*(data++)   (memory location content)
    ADD  R1, R1, #1       ; i++
    CMP  R1, #0x40        ; compare i, 64
    ADD  R2, R3, R2       ; sum+=R3
    BCC  checksum_v4_loop ; loop if i<64
    MOV  R0, R2, LSL #16  ; It removes/masks upper 16 bit
    MOV  R0, R0, ASR #16  ; R0=(short)sum
    MOV  PC, R14          ; return R0
```

This is going to save one ADD instruction compared to previous one.

2 because return type is short

Instead of incrementing the index value, we are incrementing the pointer itself in version-4 assembly code.

* Can you notice, I'm not using R0 in MOV R2, R0 Need not load base address into R0 register. So I didn't use MOV R2, R0 as in previous case.

So, finally you understood that

- In Assembly level pgm, char is unsigned.
- Avoid using char & short data types bcz they lead to additional instructions and it'll lead to burden on the compiler & your program becomes slow.

one means left shift -> multiply by 2
right shift -> divide by 2
one means >> both right shift

Function Argument Types.

```
short add_v1 (short a, short b)
{
    return a+(b>>1);
}
```

a+(b>>1)
↓ means
a+(b/2)

you'll get the output depending on which compiler you're using.

armcc is a compiler

armcc output:

```
add_v1
    ADD  R0,R0,R1,ASR #1 ; R0 = (int)a + ((int)b >> 1)
    MOV  R0,R0,LSL #16
    MOV  R0,R0,ASR #16 ; R0 = (short)sum
    MOV  PC,R14        ; Return R0
```

R0 (R0 + (R1 right shifted once))

arm -> -32768 ↔ 32768

ASR will retain the signed bit.
So, we are using ASR & not LSR.

gcc output:

```
add_v1_gcc
MOV  R0, R0, LSL #16
MOV  R1, R1, LSL #16
MOV  R1, R1, ASR #17
ADD  R1, R1, R0, ASR #16
MOV  R1, R1, LSL #16
MOV  R0, R1, ASR #16
MOV  PC, LR
```

gcc has not assumed anything

(R0's upper 16 bit removed)

(R1's upper 16 bit are removed or lost & lower 16 bits are filled with zero & lower 16 bits are filled with zero instead of writing these two steps as MOV R1, R1, ASR #16 MOV R1, R1, ASR #1 I did to divide it by two)

to divide it by two I did the number ASR by 1 time

bcz the second number has to be divided by 2

(15 bit magnitude + 1 bit MSB) -> 16-bit

Add R0+R1 & store in R1

instead of doing separately we can do in one instruction

R0 is ASR 16 times

R0 is loaded with R1 which arithmetic shifted right by 16 times.

caller function / calling function

function argument
- narrow
- wide

callee function -> The one which is called to compute something

return value
- narrow
- wide

mostly the main() function

mostly the subroutines (or subfunctions)

caller function :-

- function argument narrow means (function main())
  It has done the necessary type casting before it is passed to the callee function. The callee function will perform only the task given to it & return the value.
- function argument wide means
  The calling function (or main()) will not do necessary type casting before passing to callee function. The callee function has to do that necessary type casting.

callee function

- narrow means all necessary data conversion is done by the callee function & then returned to the calling (or main()) function.

- wide means the callee function will return necessary data & the conversion must be done by the calling (or main()) function.

ANSI C prototype

-

Note:-

- armcc will not convert a & b to 16-bit. It'll compute sum first & the sum is converted to 16 bit → The armcc compiler uses ANSI C prototype of the function where the function arguments are passed narrow & values returned narrow.

- The gcc compiler makes no assumptions about the range of argument value. It reduces the input arguments to the range of short in both, the caller and the callee function. It also casts the return value to a short type.

2 In gcc compiler a & b will be converted to 16 bit, then compute sum & the sum is also converted to 16-bit by gcc.

Signed versus unsigned Types

```
int average_v1(int a, int b)
{
  return (a+b)/2;
}
```

→ Here arguments are of integer type & also the return type.

a & b are signed numbers

```
average_v1
ADD  R0, R0, R1          ; R0 = a+b
ADD  R0, R0, R0, LSR #31 ; if R0<0; R0++
MOV  R0, R0, ASR #1      ; R0 = R0 >>1 ; R0/2
MOV  PC, R14             ; Return R0
```

→ means → R0 gets divided by 2.

- Bcz of the constraint on hardware architecture, we don't have division operation in assembly language program.
- To perform the division operation, I'll use LSR
- I'm dealing with signed numbers.
  - ↳ a & b are signed.
  - ↳ means either +ve or -ve number.
- If you shift right anything 31 times, then msb will come to the position of LSB

eg:- 1st I'll do LSR 31 times

→ only the MSB bit will be retained in LSB position & rest all bits are lost

* If the sum is -ve, I'm adding 1 to it. else I'll keep the sum as it is.

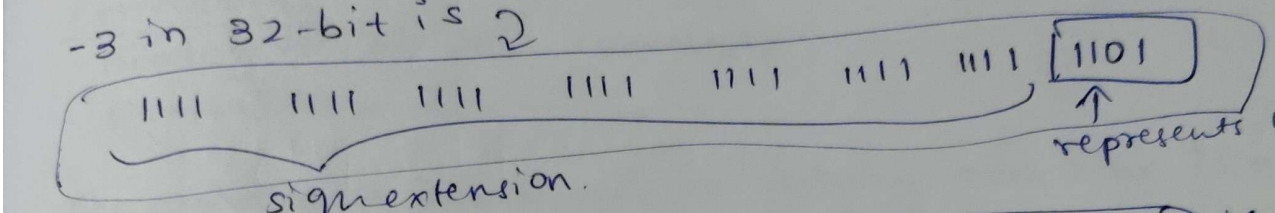
* magnitude is represented by 2's compliment & MSB represents sign in a negative numbers in a computer system.

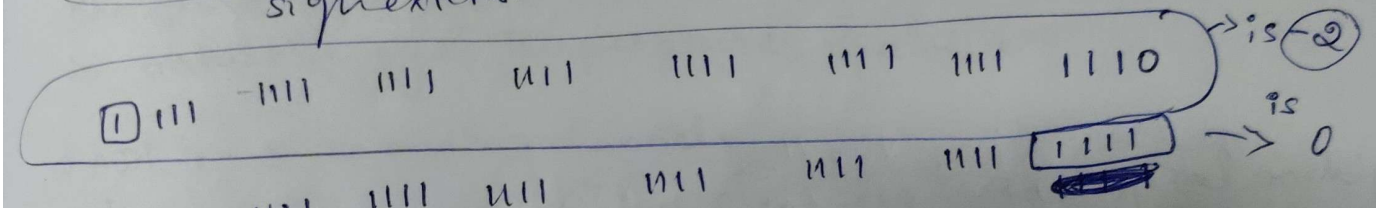
Sum = -3 $\boxed{+3 = 0011}$

$$\begin{array}{r} 0011 \\ \downarrow\ 1100 \\ +\quad\ 1 \\ \hline 1101 \end{array}$$

-3 in 4bit is 1101

-3 in 32-bit is 2





$$\begin{array}{r} 0000 \\ +\quad\ 1 \\ \hline 0001 \end{array}$$

Whenever you want to divide a -ve number by 2, check whether result (it) is <0. If so add 1 to it. If it is +ve nothing is added

Unit :-

Input devices	Output
• sensors	• Motor → eg: DC motor, stepper motor.
• switches	• LCD
	• 7-segment display • LED

Potentiometer → is basically a resistor.
Potentiometer is used to vary the intensity
DC motor " " " " " speed.

Convertors
- Analog to Digital convertors
- Digital to Analog "

The parameter which will be varying over time is analog in nature. Eg:- Temperature, Speed, etc.

ADC's → 14 analog channels to read analog & convert it to digital onchip itself.
(Analog to Digital convertors)

D2A convertor → Digital to Analog convertors.

IC → Integrated circuit of ARM7

8051 → 40 pin IC

ARM7 is advanced than 8051.

ARM7 is 64 pin IC, available in quad core.

quad core processor.

↳ on each side 16 pins. 16 × 4 = 64 pins.

each pin can have min. 1 functionality & max 4 functionality.

Single pin can work in a different mode means the pin is multiplexed in nature.

How can I select the functionality of pin as it is multiplexed in nature? We've Pin Selectors

GPIO → General Purpose Input Output.

8051 has 4 ports

P0, P1, P2, P3.

ARM7 has

2 ports

P0 & P1 each 32-bit wide

How to configure port zero pins as input ports?

& " " " " one " " output " ?

PORT 0 : 32 bit, But only 29 pins available as GPIO. → In that only 1 pin is available as GPO (General Purpose Output. You can't configure it as Input. ie Port no: 31 (MSB of Port0)

PORT 1 : 32 bit, But only 16 pins available as GPIO.

Total

So, (45) pins are available for GPIO

$P_{1.0}$ - $P_{1.15}$

There are 4 functions. If you want to select one function at a time among them, then how many bits do you need?

Ask mam. →

$2^0 = 1$ Ans- 1 bit

Control bit b_1	Control bit b_0	functions
0	0	→ GPIO
0	1	→ 1st alternative
1	0	→ 2nd alt
1	1	→ 3rd alt.

$2^n \rightarrow 2^2 = 4$ combinations

$n=2$

3 pin select Registers

(> all are 32-bit wide.

used to configure port zero

PINSEL 0 → pin number 0-15

PINSEL 1 → 15-31 pin number

PINSEL 2 → 32-bit wide

whereas for port 0, I'll use two pin select registers.

to configure port one

It configures port 1 functionality i.e 0 to 15

Bcz only 16 pins are available from port 1 to the user. So, I'll use only one pin select register for port 1

PWM → Pulse Bit Modulated input

Do you know reset value? / Know about set & reset

If you want to make any of the output pin high -> then use set.

you'll use reset if you want to clear the output

---

you can't use set & reset for input pins.

The combination 00 is always for GPIO functionality.

1 -> Output
0 -> Input
} in ARM7

0 -> [redacted]
1 -> [redacted]
} in 8051 microcontrollers.

---

* PINSEL 0 used to access pins from $P_{0.0}$ to $P_{0.15}$
* PINSEL 1 " " " " " $P_{0.16}$ to $P_{0.31}$

STRUCTURE OF ARM7 'C' program

1) Include headerfile.
   headerfile used here is LPC21XX
   # include <LPC21XX.h>

LPC21XX.h → headerfile

2) Function declaration
   see that its return value is integer rather than char or short
3) Global variables
4) int main() ← Always main() should be of integer data type.
5) In hardware always blinking the LED will be first program.

Assume LED is connected to P0.0

PINSEL 0 is used as I'm using only P0.0.

We're interested only in LSB bit of port 0.

0000 → 0
0001 → 1

You've to make 0x0000 0000 only!!) Port 0 (0-15) pins are GPIO

32 bit wide register it will assign direction to individual pins

->IO direct

IODIR = 0x0000 0000 → Value 0 is dedicated for input & value 1 is dedicated for output

→ making all GPIO pins available

= 0xFFFF FFFF → all one's means 8 F's

→ means 1111 1111

means GPIO pins won't be available.

IODIR
↓
to define whether it is an input pin or output pin, Direction will tell it

IODIR is dedicated for port zero pins & IO1DIR is dedicated for port1.

1 indicates that, that particular pin is configured as output pin

0 → input
1 → output

→ to set the values
IOSET = FFFFFFFF

→ to clear the values
IOCLR = FFFFFFFF

IO0SET = 0x00FF0000

pin no. 16 to 23 are associated with this

pin no. 0 to 3

pin no 4-7

Embedded C Termworks

TW 1 (on LED blinking)

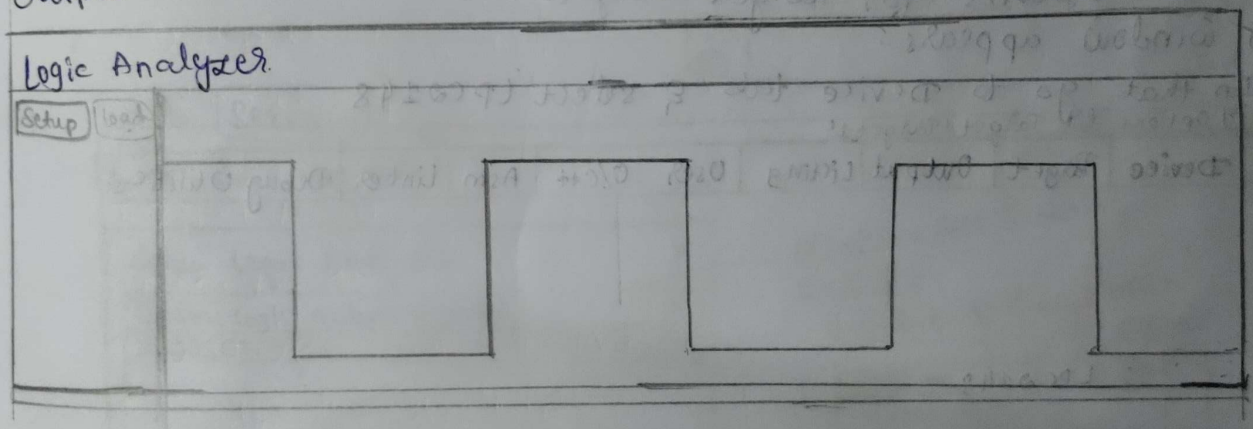
Program:-

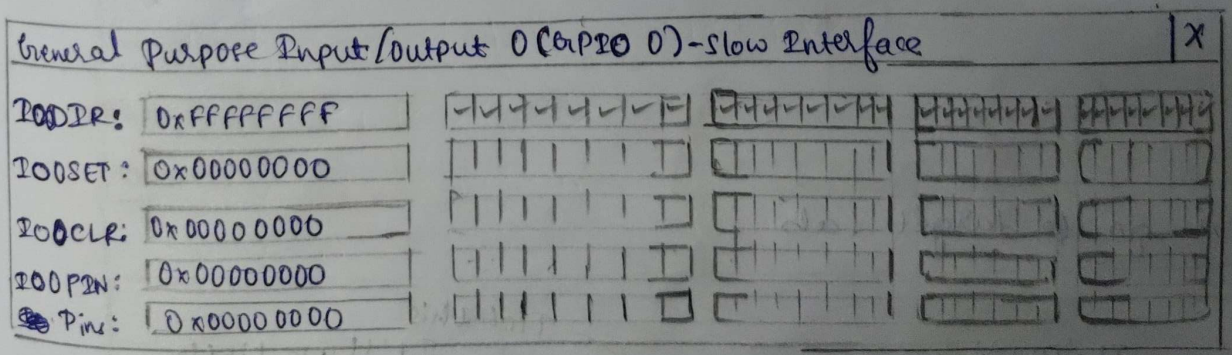
```
#include<LPC21xx.h>
unsigned int delay;
int main()
{
  PINSEL1 = 0X00000000; // configure P0.16 to P0.23 as GPIO
  IO0DIR = 0XFFFFFFFF; // configure P0.16 to P0.23 as OUTPUT pins
  while(1)
  {
    IO0SET = 0X00FF0000; // SET PINS 16-23 of PORT 0
    for(delay=0; delay<10000; delay++);
    IO0CLR = 0X00FF0000; //CLEAR PINS 16-23 OF PORT 0
    for(delay=0; delay<10000; delay++);
  }
}
```

pin no. 16-23 (8 pins)

To clear them we are putting 1's

Output:-

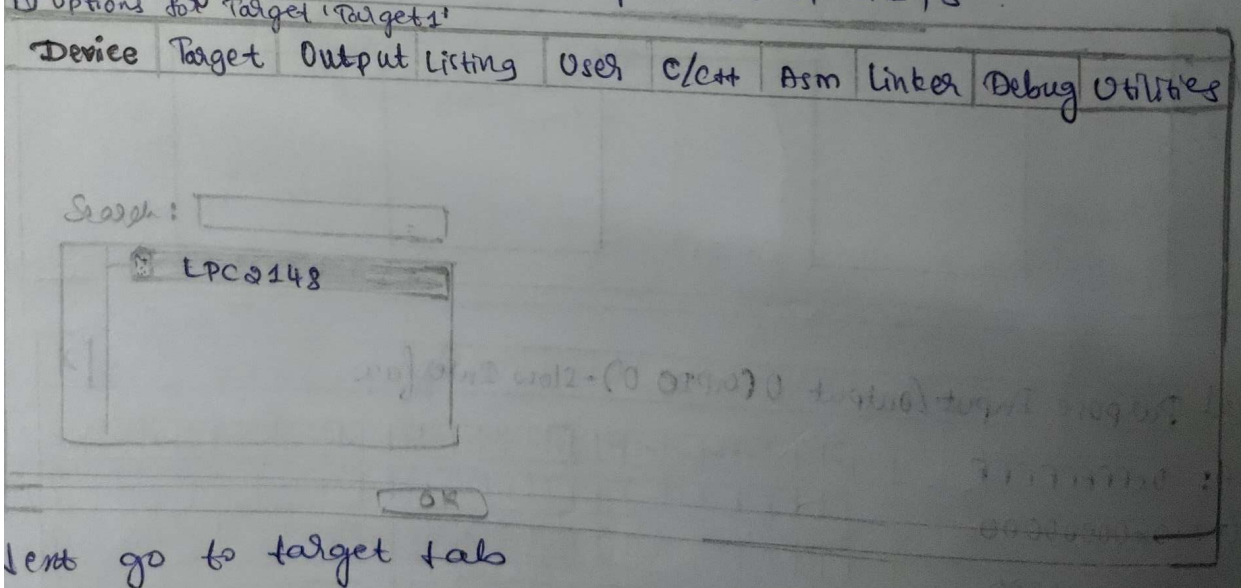




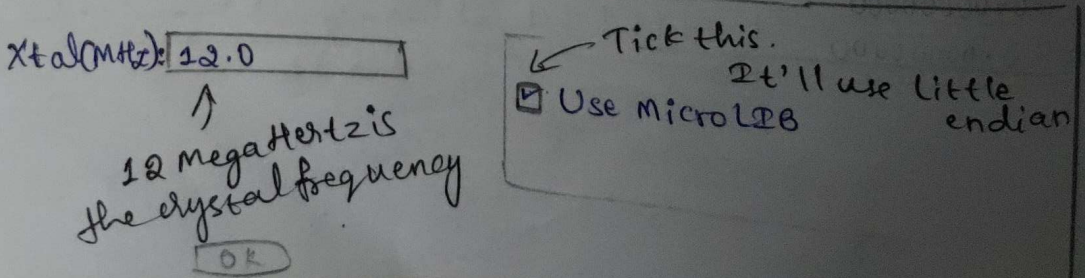
eps to do this in Keil:- → Keil microvision 4 ie Keil µV4

Next Steps after you finish writing the program:-

ight click on Target1
oose : Options for Target 'Target1'...
window appears.
In that go to Device tab & select LPC2148



Next go to target tab

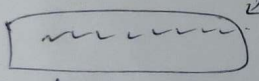
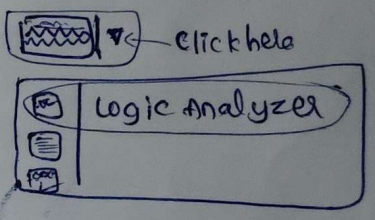


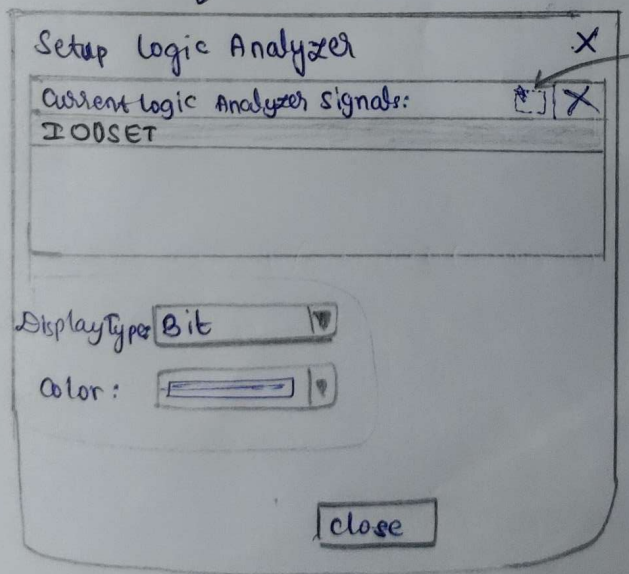
6. Next go to Output tab & Check
   ☑ Create hex file
7. Next go to Linker tab & Check
   ☑ Use Memory Layout from Target Dialog
   This will allow compiler to use on chip ROM of ARM7
8. Click on OK.

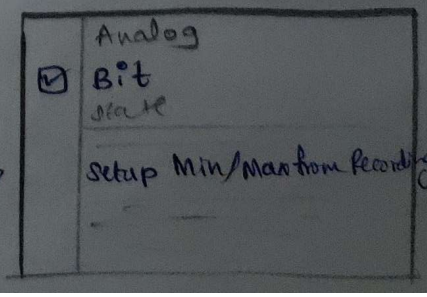
Next steps :—

1. Right click on Source group
2. Add existing files....
3. Select your file & then [Add] & [Close] you click.

Next steps :—

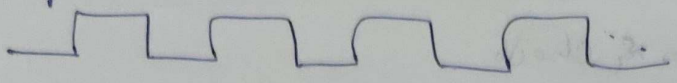
1. Translate, Build, Start-Stop-Debug Session.
2. Go to Peripherals
3. Select GPIO Slow Interface & then Port0
   A small window appears. Keep it as it is. It is to see blinking of LEDs.
   
   like this.
4. Go to Debug & Click on Run. (you can click on Run now or at the end. Both are OK.)
   ↑ Just to see Waves & blinking ticks.
5. Click on Analysis window icon
   

6. Go to [Setup...]
   A window appears.
   

   click on this & write IOOSET. select it & then close. This window will go after clicking close.
7. Now, Right click. Some options appear.
   Select Bit.
   Select Setup Min/Max from Recording
   

As you

8. As you have clicked on Run, you can see movement of waveform



So, like this when you finish checking blinking of LED on Logic Analyzer window, now how to check blinking on board, let us see...

* Counter program of Embedded C

Program:-

```
#include<LPC21xx.h>
void delay(void);
unsigned int count;
int main()
{
  unsigned int comp=0; // compliment
  PINSEL1 = 0x00000000; // Configure Port0(16-31) as GPIO pins
  IO0DIR = 0xFFFFFFFF; // Configure P0.16-P0.31 as OUTPUT
  while(1){
    for(count = 0; count <= 0xFF; count++){
      comp = (~count); // Just to ensure that after 255, 0 should
                       come.
      comp = comp & 0x000000FF ; // to fetch lower 8-bit data
      // Now I'm supposed to send it on I/O pin
      IO0PIN = (comp<<16); // you can see the status of COUNT
                           from this pin. IO0PIN because
                           I'm using PORT 0.
                                  -> IO zero pin
                                  -> Port zero
      delay();
    }
  }
}
void delay(void)
{
  unsigned int i;
  for(i=0; i<6500000; i++);
}
```