21CS43

MICROCONTROLLER & EMBEDDED SYSTEM

Chapter-01

Module-02

Introduction To The ARM Instruction set

C compilers & optimization

Topic-01

Data Processing Instruction

Mnemonic	operation	Action
AND	Logical AND	Rd := Rn AND shifter_operand
EOR	Logical Exclusive OR	Rd := Rn EOR shifter_operand
SUB	Subtract	Rd := Rn shifter_operand

RSB	Reverse Subtract	Rd := Subtract_operand Rn.
ADD	Add	Rd := Rn+ shifter_oper-and
ADC	Add with carry	Rd := Rn+ shifter_oper-and + Carry Flag.
SBC	Subtract with carry	Rd := Rn - shifter_operand - NOT (~~Flag~~ (carry flag.
RSC	Reverse subtract with carry	Rd := shifter operand Rn - NOT (Carry Flag)
TST	Test	Update flags after Rn ~~&~~ AND shifter_operand
TEQ	Test Equivalence	update flags after Rn EOR shifter_operand.
CMP	Compare	Update flags after Rn - Shifter_Operand
CMN	Compare Negated	Update flags after Rn + shifter_operand.

ORR	Logical (inclusive) OR	RD := Rn OR shifter_operand
MOV	Move	Rd := shifter operand (no first operand)
BIC	Bit clear (Logical NAND)	Rd := Rn AND NOT (shifter_operand)
MVN	Move Not	Rd := NOT shifter_operand (no first operand)
ASR	Arithmatic Shift right	Rd = Rm >> immediate, C flag = Rm [immediate -1] Rd = Rd >> Rs, C flag = Rd [Rs-1]
LSL	Logical Shift Left	Rd = Rm << immediate C flag = Rm [32-immediate] Rd = Rd << Rs, C flag = Rd [32-Rs]

LSR	Logical shift right	Rd = Rm >> immediate
		C flag = Rd [immediate − 1]
		Rd = Rd >> Rs,
		C flag = Rd [Rs − 1]

ROR	Rotate right a 32 bit value	Rd = Rd RIGHT-ROTATE Rs, C flag = Ra [Rs − 1]

* Most data - processing Instructions take two source operands, through Move & Move Not take only one.

* The Compare & Test Instructions only Update the Condition flags.

* Of the Two source operands, one is always a Register.

* The ~~Other~~ other is called a shift_operand & is either an immediate Value ⓞ a Register.

AND, ORR, EOR, BIC - Logical Instructions

Mnemonic	Operation	Action
AND	Logical AND	Rd := Rn AND shifter_operand
ORR	Logical (inclusive) OR	RD := Rn OR shifter_Operand
EOR	Logical Exclusive OR	Rd := Rn EOR shifter_operand
BIC	Bit clear (Logical NAND)	Rd := Rn AND NOT (shifter_operand).

* The Instruction performs a Bitwise Logical Operation of the value of Register <Rn> with the value of <shifter_operands> and stores the Result in the destination Register <Rd>
* The condition code flags are optionally updated, Based on the Result

Example: AND r0, r1, r2

Logical AND operation

ADC, ADD, RSB, RSC, SBC, SUB, NEG, - ARITHMATIC Instruction

Mnemonic	operation	Action
ADC	Add with Carry	Rd := Rn + shifter_operand + carry flag.
ADD	Add	Rd := Rn + shifter_operand
RSB	Reverse Subtract	Rd := shifter_operand - Rn
RSC	Reverse Subtract with carry	Rd := shifter_operand - Rn - NOT (Carry flag).
SBC	Subtract with carry	Rd := Rn - shifter_operand - NOT (Carry flag)
SUB	Subtract	Rd := Rn - shifter_operand
NEG	Negate a 32-bit value	

* The Arithmatic Instructions Implement addition & subtraction of 32-bit signed & unsigned values.

Example:- If r_1 = ob1111 , r_2 = ob0101 after

BIC r_0, r_1, r_2

Solution:- 1111 AND 0101 = 1010

∴ Value in r0 after Execution of given instruction is : r_0 = ob1010

CMN, CMP, TEQ, TST - Compare & Test Instruction

Mnemonic	Operation	Action
CMN	Compare Negated	Update flags shifter + shifter_operand
CMP	Compare	Update flags after Rn - shifter_operand
TEQ	Test Equivalence	Update flags after Rn EOR shifter_operand.
TST	Test	Update flags after Rn AND shifter_ Operand.

* The Compare & Test instructions are Used to Compare & Test a Register with 32-bit Value Respectively.

Example

CMP r0, r1 ; r0 - r1 & updates flags according to Result

MOV, MVN - Move Instruction

Mnemonic	Operation	Action
MOV	Move	Rd := shifter_operand (no first operand).
MVN	Move Not	Rd:= NOT shifter_operand (no first operand).

* The MOV (MOVE) instruction Copies the Values of <shifter_ operands> to the destination Register <Rd>.
* The MVN <Move negative> instruction copies the Logical One's Complement of the

value of <shifter_operand> to the destination Register <Rd>.

Ex:- MOV r2, r3 ; r2 ← r3

ASR, LSL, LSR and ROR instructions.

* Thumb deviates from the ARM style in that the Barrel shift operations ASR, LSL, LSR & ROR are separate Instructions

ASR - Arithmatic Shift Register

LSL - Logical Shift Left

LSR - Logical Shift Right

ROR - Rotate Right

Ex:- LSL r2, r4

Topic-02

Branch Instruction

* All ARM processors support a Branch Instruction that allows a conditional Branch forwards / Backwards up to 32 MB.

* The Branch with Link (BL) instruction preserves the Address of the instruction after the Branch in the LR(R14), this allows to perform subroutine call.

B. BL, Branch, and Branch and Link.

Syntax:-

```
B {<cond>} <target_address>

BL {<cond>} <target_address>
```

Examples :

```
B Label      ; branch Unconditionally to
                                   Label.
BCC Label    ; Branch to Label if carry flag
                     is clear.
```

BEQ label : Branch to label if zero flag is set.

MOV PC, #0 : R15 = 0, Branch to location zero.

BL func : subroutine call to function func.

MOV PC, LR : R15 = R14 return to instruct-ion after the BL.

MOV LR, PC : store the address of the instruction
: after the next one into R14 ready to Return

LDR PC, =func ; load a 32-bit value into the program
; counter.

Syntax: BLX < target_address>

* The BLX (Branch with Link & Exchange) instruction is used to call a Thumb subrout-ine from the ARM instruction set ~~on~~ at

an address specified in the instruction.

Ex:- BLX T_func ; Thumb subroutine call to func.

BX - Branch & Exchange Instruction

Syntax:- BX {<cond>} <Rm>

* BX (Branch & Exchange) Instruction Branches to an address held in a Register Rm with an optional switch to Thumb Execution.

Example:

BX Fun ; Branch to Target ARM instruction (or) to Target Thumb instruction.

Topic-03

Software Interrupt Instructions

The ARM instruction set provide two types of Instruction whose Main purpose is to cause a processor Exception to occur:

* The Software Interrupt (SWI) instruction is Used to cause a SWI Exception to occur.

* This is the Main Mechanism in the ARM instruction set by which User mode code can Make calls to previleged operating System code.

* The Breakpoint (BKPT) instruction is Used for Software Breakpoints in ARM Architecture versions 5 & Above.

* It's default Behaviour is to cause a prefetch abort Exception to occur.

Syntax:

SWI {<cond>} <immed 24>

<immed_24> Is a 24-bit immediate value that is put into bits [23:0] of the Instruction.

This value is ignored by the ARM processor, But Can be Used by an

Operating System SWI Exception handler to determine what Operating System Service is Being Requested.

BKPT<immediate>

<immediate> Is a 16-bit immediate value the top 12 bits of which are placed in Bits [19:8] of the Instruction. and the Bottom 4 bits of which are placed in Bits [3:0] of the instruction.

This value is ignored By the ARM hardware But can be Used by a debugger to store Additional Information about the Break-point.

Topic-04

Program Status Register Instructions

There are Two instructions for Moving the Contents of a program Status Register to or from a general-purpose Register.

* Both the CPSR & SPSR can be Accessed.

<u>Syntax:-</u>

MRS{<cond>} <Rd>, CPSR

MRS{ <cond>} <Rd>, SPSR

MSR{<cond>} CPSR_<fields>, # <immediate>

MSR{<cond>} CPSR-<fields>, <Rm>

MSR{ ~~cond~~ <cond>} SPSR_<field>, # <immediate>

MSR{<cond> } SPSR_<fields>, <Rm>

<fields> Is sequence of one or more of the following:

sets the control field mask bit(bit(6)

sets the extension field mask bit(bit (7)

sets the status field mask bit(bit (8)

sets the flags field mask bit(bit (9)

<Rm> Is the general - purpose Register to

be Transferred to the CPSR or SPSR.

Examples

* These Example assume that the ARM processor is Already in a privileged mode.

* If the ARM processor starts in User mode only the flag update has any Effect.

```
MRS  R0 CPSR          ; Read the CPSR.
BIC  R0 R0 #0XF0000000 ; clear the N
                         Z, C and V
                         bits.
MSR  CPSR_f R0        ; Update the flag
                         bit in the CPSR
                         N, Z, C, and V flags
                         now all clear.
MRS  R0 CPSR          ; Read the CPSR.
ORR  R0 R0 #0X80 ;    Set the Interrupt
                         disable bit.
MSR  CPSR-C, R0 ;     Update the Control
                         bits in the CPSR
```

interrupts (IRQ) now disabled.

~~MRS CPSR_C, R0 // Update the control bits in the CPSR~~

```
MRS R0, CPSR        ; Read the CPSR.
BIC R0, R0, #0x1F   ; Clear the mode
                      bits.
ORR R0, R0, #0x11   ; Set the mode bits
                      to FIQ mode.
MSR CPSR_C, R0      ; Update the control
                      bits in the CPSR
                      now in FIQ mode.
```

Topic-05

Coprocessor Instructions

The ARM instruction set provides three types of Instruction for communicating with coprocessors.

* The ARM processor to Initiate a coprocessor data processing Operation.

* ARM Register to be Transferred to and from Coprocessor Register.

* The ARM Processor to generate address for the coprocessor Load & Store Instructions.

Mnemonic	Operation
CDP	Coprocessor data operation
LDC	Load - Coprocessor Register.
MCR	Move to Coprocessor from ARM Register.
MRC	Move to ARM Register from Coprocessor.
STC	Store Coprocessor Register.

Table: Coprocessor Instruction

Syntax:

CDP {<cond>} <coproc>, <opcode_1>, <CRd>, <CRn>, <CRn>, <CRm>, <opcode-2>

CDP2 <coproc>, <opcode_1>, <CRd>, <CRn>, <CRm>, < opcode_2>

Suffix causes the condition field of the instruction to be set to 0b1111. This Provides additional opcode space for coprocessor designers. The Resulting instructions can only be Executed unconditionally.

<coproc> specifies the ~~same~~ name of the coprocessor & causes the Correspond-ing coprocessor number to be placed in the cp-num field of the instruction. The standard generic co-processor names are P0, P1, --- P15.

<opcode-1> specifies which coprocessor operation is to be performed.

<CRd> specifies the destination coprocessor Register for the instruction

<CRn> specifies the coprocessor Register that contains the first operand for the Instruction.

<CRm> specifies the coprocessor Registes that contains the second operand. for the Instruction

<opcode-2> specifies which coprocessor is to be performed.

Example/

CDP p5, 2, C12, C10, C3, 4 ; Coproc 5 data operation
; ~~opcod~~ opcode 1=2,
opcode 2=4
destination Register is 12

; Source registere are 10 and 3

Syntax:-

MCR {<cond>} <coproc>, <opcode-1>, <Rd>,
<CRn>, <CRm> {, <opcode-2>}

MCR2 <coproc>, <opcode1>, <Rd>, <CRn>,
<CRm>, {<opcode-2>}

<Rd> is the ARM Register whose value is transferred to the Coprocessor.

If R15 is specified for <Rd>, the result is UNPREDICTABLE

<CRn> Is the destination coprocessor Register.

<CRm> Is ~~the~~ an Additional destination or source Coprocessor Register.

Example:

```
MCR p14, 1, R7, C7, C12, 6

    ; ARM Register transfer to
    ; coproc 14, opcode 1 = 1,
    ; opcode 2 = 6
    ; ARM source Register = R7
    ; coproc dest Registers
    ; are 7 and 12
```

Syntax:

```
MRC {<cond>} <coproc>, <opcode_1>, <Rd>,
        <CRn>, <CRm>, {<opcode_2>}

MRC2 <coproc>, <opcode_1>, <Rd>, <CRn>,
        <CRm> {<opcode_2>}
```

<Rd> specifies the destination ARM Register for the Instruction.

If R15 ~~is~~ is specified for <Rd>, the condition code flags are Updated instead of a general-purpose Register.

<CRn> specifies the Coprocessor Register that contains the first operand for the Instruction.

<CRm> Is an Additional coprocessor source (or) destination Register.

Example:-

MRC p15, 5, R4, C0, C2, 3 ; coproc 15 transfer to ARM

; Register opcode 1=5, opcode 2 = 3 ARM destination

; Register = R4 coproc source

; register are 0 and 2.

Topic-06

Loading ~~Constants~~ Constants

* We cannot load an Arbitrary 32-bit Immediate constant into a Register in a ~~Big~~ single Instruction without performing a data load from Memory.

* we can load any 32-bit value into a Register with a data load, But there are more direct & Efficient ways to load Many ~~con~~ Commonly-Used Constants.

~~direct~~

* we can include many Commonly-Used Constants directly as operands within data processing instructions, without a separate load operation.

* The LDR Rd = Const pseudo-instruction can construct any 32-bit numeric Constant in a single Instruction.

* The LDR pseudo-instruction generates the Most Efficient single Instruction for a specific constant:

* If the constant can be constructed with a single MOV (or) MVN instruction the assembler generates the Appropriate instruction.

* If the constant cannot be constructed with a single MOV (or) MVN instruction the Assembler:

  - places the value in a literal pool
  - Generates an LDR instruction with a Program - Relative address That reads the Constant from the literal pool.

Example

LDR rn, [pc, #offset to literal pool]
 ; load Register n with one word.

; from the address [pc+ offset]

* we must Ensure that there is a literal pool within Range of the LDR instruction generated by the Assembler.

Chapter - 02

C Compilers & Optimization

Topic - 01

Basic C Data Types

* ARM processor have 32-bit Register & 32-bit data processing operations.

* Early Versions of the ARM Architecture (ARM V1 to ARM V3) provided hardware support for loading & storing Unsigned 8-bit & Unsigned (or) signed 32-bit values.

Architect-ure	Instruction	Action
Pre-ARM V4	LDR B	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.
ARM V4	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.
ARM V5	LDRD	load a signed or Unsigned 64-bit Value
		store a signed or unsigned 64-bit value.

Table:- Load & store Instruction by ARM Architecture

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.

Table:- C compiler data type mappings

1. Local variable Types

* ARMv4-Based processor can Efficiently load & store 8, 16-bit & 32bit data
* However most ARM data processing operati-on are 32 bit only.

* For this Reason you should use a 32-bit datatype. int or long for local variable wherever possible.

* Avoid using char & short as local variable types.

Example : Checksum function

```
int checksum_v1(int *data)
{
    char i;
    int sum =0;
   for(i=0; i<64; i++)
   {
    sum+= data[i];
    }
   return sum;
}
```

Consider the compiler output for this function

```
checksum_v1
        MOV   r2, r0        ; r2 = data
        MOV   r0, #0        ; sum = 0
        MOV   r1, #0        ; i = 0

checksum_v1_loop
        LDR  r3, [r2, r1, LSL, #2] ; r3 = data[i].
        ADD  r1, r1, #1             ; r1 = i+1
        AND  r1, r1, #0xff          ; i = (char) r1.
        CMP  r1, #0x40              ; compare i 64
        ADD  r0, r3, r0             ; sum += r3
        BCC     checksum_v1_loop ; if (i<64) loop
        MOV     pc, r14         ; return sum
```

2. Function Argument Types

* we saw the Local variable types that converting Local variables from types char (or) short to type int increases performance & Reduce code size.

* The same holds ~~the~~ for function arguments.

Example: Consider the following simple function, which adds two 16-bit values having the second, & Return a 16-bit sum.

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

* For armcc in ADS function arguments are passed narrow & values Returned narrow.

```
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)r0
        MO    pc, r14            ; return r0.
```

3. Signed Verses Unsigned Types

* The Above To Function Argument Types demonstrate the Advantages of Using int, rather than a char ⓞ ~~short~~ short type for Local variables & function Arguments.

* In Sub topic compares the Efficienci -es of signed int & Unsigned int.

* If your code uses addition, subtraction & multiplication, then there is no performance difference between signed & unsigned operations.

* However there is a difference when it comes to division.

Example: consider the following short example that averages two integers.

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

This compiles to

```
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
        MO     PC, r14                 ; return r0.
```

Topic-02

C Looping structures

* The Most efficient ways to Code for & while loops on the ARM.

* we start by looking at ~~loopings~~ loops with a fixed number of Iterations & then move on to loops with a variables number of iterations.

1. Loops with a fixed Number of Iterations

* what is the Most Efficient way to write a for loop on the ARM?

* Let's Return to our checksum Example & look at the looping structure.

* This shows how the compiler treats a loop with ~~increasee~~ incrementing count i++.

```
int checksum_v5(int *data)
{
  Unsigned int i;
   int    sum=0;
  for( i=0; i<64; i++)
   {
      sum += *(data++);
   }
   return sum;
}
```

This compiles to

```
checksum_v5

     ~~MOV~~
     MOV      r2, r0        ; r2 =data
     MOV      r0, #0        ; sum =0
     MOV      r1, #0        ; i=0
```

checksum-v5_loop

```
    LDR   r3, [r2], #4              ; r3 = *(data++)

    ADD   r1, r1, #                 ; i++

    CMP   r1, #0x40                 ; compare i, 64

    ADD   r0, r3, r0                ; sum+=r3

    BCC   checksum_v5_loop          ; if (i<64)) goto
                                      loop.

    MOV   PC, r14                   ; return sum.
```

2. <u>Loops Using A Variable Number of Iterations</u>

* Now suppose we want our checksum routine to handle packets of Arbitrary size.

* We pass in a variable N giving the number of words in the data ~~packet~~ packet.

The checksum_v7 example shows how the compiler handlers a for loop with a variable number of iterations N.

```
int checksum_v7(int *data, unsigned int N).
{
    int sum=0;
    for(; N!=0; N--)
    {
        sum+=*(data++);
    }
    return sum;
}
```

This compiles to

```
checksum_v7
        MOV  r2, #0                ; sum =0
        CMP  r1, #0                ; compare N, 0
        BEQ  checksum_v7 end       ; if (N==0)
                                     goto end.
```

```
checksum_v7_loop
        LDR     r3, [r0], #4          ; r3 = *(data++)
        SUBS    r1, r1, #1            ; N-- and set
                                        flags
        ADD     r2, r3, r2            ; sum+=r3
        BNE     checksum_v7_loop      ; If (N!=0) goto
                                        loop.
checksum_v7_end.
        MOV     r0, r2
                                      ; r0 = sum
        MOV     pc, r14               ; return r0.
```

3. Loop UnRoolling

* That each loop iteration costs two instruction in addition to the Body of the loop : a subtract to decrement the loop count & a Conditional Branch

* we call these instructions the loop overhead.

* On ARM7 (or) ARM9 processors the subtract takes on cycle & the Branch three cycles, giving an overhead of four cycles ~~for~~ per cycle.

Example : following code Unrolls our packet checksum loop by four times.

```
int checksum_v9(int *data, unsigned
                                   int N).
{
    int sum=0;
     do
     {
        sum += *(data);
        sum += *(data);
        sum += *(data);
        sum += *(data);
         N -= 4;
} while (N!=0);
  return sum;
}
```

This compiler to

```
Checksum_v9
        MOV   r2, #0          ; sum =0

checksum_v9_loop
        LDR   r3, [r0], #4    ; r3= *(data++);
        SUBS  r1, r1, #4      ; N--4 & set flags.
        ADD   r2, r3, r2      ; sum += r3
        LDR   r3, [r0], #4    ; r3 = *(data++);
        ADD   r2, r3, r2      ; ~~[illegible]~~ sum+=r3.
        LDR   r3, [r0], #4    ; r3 = *(data++);
        ADD   r2, r3, r2      ; sum+=r3.
        BNE   checksum_v9_loop ; if (N!=0) goto loop
```

```
MOV    r0, r2      ; r0 = sum.
MOV    pc, r14     ; return r0.
```

Topic-03

Register Allocation

* The compiler attempts to allocate a processor Register to each local variable you use in a C function.

To Implement a function Efficiently, you need to

1. Minimize the number of spelled variables.
2. Ensure that the Most Important & frequently Accessed variables are stored in Registers.

Register Number	Alternate Register names	ATPCS Register Usage
r0 r1 r2 r3	a1 a2 a3 a4	Argument Registers. These hold the first four Function arguments on a function call & the Return value on a function Return. A function may Corrupt these Register & Use them as general Scraatch Registere within the function.
r4 r5 r6 r7 r8	v1 v2 v3 v4 v5	General variable Registers. The Function must preseve the callee values of these Registers.
r9	v6 sb	General variable Register. The Function must preresve the callee value of this Register Except when compiling for read-write position-independence (RWPI). The r9 holds the static Base address. this is the address of the read-write data.
r10	v7 sl	General variable Register. The function must preserve The Callee value of this Register Except

		when compiling with stack. ~~Using a frame pointer. only old versions of armcc Use a frame pointer~~
r11	v8 fp	General variable register. The function must preserve the calle value of this register Except when compiling Using a frame pointer. only old versions of armcc Use a frame pointer.
r12	ip	A general scaratch Register that the function can corrupt. It is useful as a scratch Register ~~that~~ ~~the~~ for function veners or other intraprocedure call Requirem-ents.
r13	sp	The stack pointer. pointing to the full descending stack.
r14	lr	The link Register. on a function call this holds the return address.
r15	pc	The program counter.

Table: C compiler Register Usage

Topic-04

Function calls

* The ARM Procedure Call Standard (APCS) defines how to pass function arguments & Return values in ARM Registers.

* The More Recent ARM-Thumb procedure Call Standard (ATPCS) covers ARM & Thumb interworking as well.

* The ~~four~~ first four Integer arguments are passed in the first four ARM registers r0, r1, r2, & r3.

* Function Return Integer values are passed in r0.

* This description covers only integer (or) pointer arguments.

* The first point to Note about The procedure Call Standard is the four-Register Rule.

Example:- ATPCS Argument passing

```
char *queue_bytes_v1(
        char *Q_start,
        char *Q_end,
        char *Q_ptr,
        char *data,
        Unsigned int N)
{
    do
    {
        *(Q_ptr++) = *(data++);
        if (Q ptr == Q_end)
        {
            Q_ptr = Q_start;
        }
    }while(--N);
    return Q_ptr;
}
```

This compiles to

```
queue-bytes_v1
    STR   r14, [r13, #-4]   ; save lr on the
                              stack
    LDR   r12, [r13, #4]    ; r12 = N
queue_v1_loop
    LDRB  r14, [r3], #1     ; r14 = *(data++)
    STRB  r14, [r2], #1     ; *(Q_ptr++) = r14
    CMP   r2, r1            ; if (Q_ptr = = Q_end)
    MOVEQ r2, r0            ; Q_ptr = Q_start;
    SUBS  r12, r12, #1      ; --N and set flags.
    BNE   queue_v1_loop     ; if (N != 0) goto loop.
    MOV   r0, r2            ; r0 = Q_Ptr.
    LDR   pc, [r13], #4     ; return r0.
```

Compare this with a more structured approach using three function arguments.

Topic - 05

Pointer Aliasing

* Two pointers are said to alias when they point to the same Address.

* If you write to one pointer it will affect the value your read from the other pointer.

* In a function the compiler often doesn't know which pointers can alias & which pointers can't.

* The compiler must be very pessimi-stic & Assume that any write to a pointer may affect the value read from any other pointer, which can significantly Reduce code Efficiency.

Example:- The following function increments two timer values by a step amount.

```
void timers_v1(int *timer1, int *timer2, int
                    *step)
{
    *timer1 += *step;
    *timer2 += *step;
}
```

This compiles to

timers_v1

```
    LDR    r3, [r0, #0]    ; r3 = *timer1
    LDR    r12, [r2, #0]   ; r12 = *step
    ADD    r3, r3, r12     ; r3 += r12
    STR    r3, [r0, #0]    ; *timer1 = r3
    LDR    r0, [r1, #0]    ; r0 = *timer2
    LDR    r2, [r2, #0]    ; r2 = *step
    ADD    r0, r0, r2      ; r0 += r2
    STR    r0, [r1, #0]    ; *timer2 = r0
    MOV    pc, r14         ; return.
```

Note:- That the Compiler loads from step twice.