# UNIT -3
# Efficient C Programming

# LOCAL VARIABLE TYPES

- ARMv4-based processors can efficiently load and store 8-, 16-, and 32-bit data.
- most ARM data processing operations are 32-bit only.
- For this reason, you should use a 32-bit datatype, int or long for local variables wherever possible.
- Avoid using char and short as local variable types, even if you are manipulating an 8- or 16-bit value.

# Example

- 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.

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)
{
char i;
int sum=0;
for (i=0; i<64; i++)
{
Sum +=data[i];
}
Return sum;
}
```

- At first sight it looks as though declaring i as a char is efficient.
- You may be thinking that a char uses less register space or less space on the ARM stack than an int.
- On the ARM,both these assumptions are wrong
- All ARM registers are 32-bit and all stack entries are at least 32-bit.
- Furthermore, to implement the i++ exactly, the compiler must account for the case when i = 255.
- Any attempt to increment 255 should produce the answer 0

- Consider the compiler output for this function.
- We've added labels and comments to make the assembly clear.

```
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
```

- The compiler inserts an extra AND instruction to reduce I to the range 0 to 255 before the comparison with 64

Now compare this to the compiler output where instead we declare I as an unsigned int.

```
checksum_v2
        MOV r2,r0     ;  r2=data
        MOV r0,#0     ;  sum=0
        MOV r1,#0     ;  i=0

checksum_v2_loop
        LDR r3,[r2,r1,LSL#2]          ;  r3=data[i]
        ADD r1,r1,#1                  ;  r1++
        CMP r1,#0x40                  ;  comparei,64
        ADD r0,r3,r0                  ;  sum+=r3
        BCC    checksum_v2_loop       ;  if(I <64) go to loop
        MOV PC,r14                    ;   return sum
```

- The AND instruction disappears in the second case

Suppose data packet contains 16bit values and we need 16bit checksum the code typically looks like

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

Consider the compiler output for this function.

checksum_v3

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

    checksum_v3_loop
                ADD r3,r2,r1,LSL #1         ; r3=&data[i]
                LDRH r3,[r3,#0]              ; r3=data[i]
                ADD r1,r1,#1                 ; r1=i+1
                CMP r1,#0x40                 ; compare i,64
                ADD r0,r3,r0                 ; sum+=r3
                MOV r0,r0,LSL #16
                MOV r0,r0,ASR #16           ; sum= (short) r0
                BCC checksum_v3_loop     ;  if(i<=64)loop
                MOV pc,r14                  ; return sum
```

To fix this problem the C code can be written as

```
Short checksum_v4(short *data)
{
unsigned int i;
int sum=0;
for (i=0; i<64; i++)
{
Sum  +=*(data++);
}
Return short(sum);
}
```

The compiler output for this function.

```
checksum_v4
                MOV r2,#0       ;  sum=0
                MOV r1,#0       ;   i=0

    checksum_v4_loop
                LDRH r3,[r0],#2             ; r3=*(data++)
                ADD r1,r1,#1                ; r1=i+1
                CMP r1,#0x40                ; compare i,64
                ADD r2,r3,r2                ; sum+=r3
                BCC checksum_v4_loop     ;  if(i<=64)loop
                MOV r0,r0,LSL #16
                MOV r0,r0,ASR #16           ; r0= (short) sum
                MOV pc,r14                  ; return sum
```