MODULE-3

C Compilers and Optimization: Basic C Data Types, C Looping Structures, Register Allocation, Function
Calls, Pointer Aliasing, Portability Issues.
Textbook 1: Chapter 5.1 to 5.7 and 5.13
RBT: L1, L2, L3

3.1 Overview of C Compilers and Optimization:

Optimizing code takes time and reduces source code readability.therefore it's only worth optimizing functions that are frequently executed and important for performance.C compilers have to translate our C function literally into assembler so that it works for all possible inputs.

To write efficient C code, we must be aware of areas where the C compiler has to be conservative, the limits of the processor architecture the C compiler is mapping to, and the limits of a specific C compiler.

Most common c compilers are

●   armcc from ARM Developer Suite version 1.1 (ADS1.1).

●   arm-elf-gcc version 2.95.2. This is the ARM target for the GNU C compiler, gcc,

3.2 Basic C Data Types

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

Table 3.1 Load and store instructions by ARM architecture.

In Table 3.1 loads/store 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.This means that type casting of a loaded value to an int type does not cost extra instructions.

ARMv4 architecture and above support signed 8-bit and 16-bit loads and stores directly, through new instructions. Since these instructions are a later addition, they do not support all the addressing modes as the pre-ARMv4 instructions.

Compilers **armcc** and **gcc** use the datatype mappings for an ARM target.

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 3.2 C compiler datatype mappings.

3.2.1 Local Variable Types

ARMv4-based processors can efficiently load and store 8-, 16-, and 32-bit data.But, most ARM data processing operations are 32-bit only. For this reason, we should use a 32-bit datatype, **int** or **long**, for local variables wherever possible and avoid using char and short as local variable types. If we are doing modulo arithmetic operations(mod n=n) then we can use **char** type.

Example:The following code checksums a data packet containing 64 words. It shows why we 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;
}
```

Case1: The compiler output for this function is as given below

```
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 output for the same function by declaring I as **unsigned int** is as given below

```
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 = i+1
        CMP r1,#0x40             ; compare i, 64
        ADD r0,r3,r0             ; sum += r3
        BCC checksum_v2_loop    ; if (i<64) loop
        MOV pc, r14                ; return sum
```

In the first case, the compiler inserts an extra AND instruction to reduce i to the range 0 to 255 before the comparison with 64. This instruction disappears in the second case.

Case2: if the checksum is 16 bit value, then i is declared as **unsigned int**

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

```
}
return sum;
}
```

The expression sum + data[i] is an integer and so can only be assigned to a short using an (implicit or explicit) narrowing cast, sum = (short)(sum + data[i]);

With armcc this code will produce a warning if you enable implicit narrowing cast warnings.

```
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]
        LDR r3,[r2,r1,LSL #2]     ; r3 = data[i]
        ADD r1,r1,#1            ; r1 = i+1
        CMP r1,#0x40           ; compare i, 64
        ADD r0,r3,r0            ; r0 = sum + r3
        MOV r0,r0,LSL #16
        MOV r0,r0,ASR #16      ; sum = (short)r0
        BCC checksum_v3_loop     ; if (i<64) goto loop
        MOV pc, r14                  ; return sum
```

The loop is now three instructions longer than the loop for example checksum_v2 earlier! There are two reasons for the extra instructions:

● The LDRH instruction does not allow for a shifted address offset as the LDR instruction did in checksum_v2. Therefore the first ADD in the loop calculates the address of item i in the array. The LDRH loads from an address with no offset. LDRH has fewer addressing modes than LDR as it was a later addition to the ARM instruction set.

● The cast reducing total + array[i] to a short requires two MOV instructions. The compiler shifts left by 16 and then right by 16 to implement a 16-bit sign extend.The shift right is a sign-extending shift so it replicates the sign bit to fill the upper 16 bits.

We can avoid the second problem by using an int type variable to hold the partial sum.We only reduce the sum to a short type at the function exit.

The first problem is a new issue. We can solve it by accessing the array by incrementing the pointer data rather than using an index as in data[i]. This is

efficient regardless of array type size or element size. All ARM load and store instructions have a post increment addressing mode.

Case3: The checksum_v4 code fixes all the problems we have discussed in this section. It uses **int** type local variables to avoid unnecessary casts. It increments the pointer data instead of using an index offset data[i]

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

The compiler produces the following output. Three instructions have been removed from the inside loop, saving three cycles per loop compared to checksum_v3

```
checksum_v4
        MOV r2,r0                    ; r2 = data
        MOV r1,#0             ;i=0
checksum_v4_loop
        LDRSH 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) goto loop
        MOV r0,r2,LSL #16
        MOV r0,r0,ASR #16        ; sum = (short)sum
        MOV pc, r14                  ; return sum
```

3.2.2 Function Argument Types

Converting local variables from types **char** or **short** to type **int** increases performance and reduces code size. The same holds for function arguments. If the function arguments are **short** type either the caller or the callee must perform the cast to a short type. And these **char** or **short** type function arguments and return values introduce extra casts.This

increase code size and decrease performance. Therefore It is more efficient to use the int type for function arguments and return values, even if you are only passing an 8-bit value.

3.2.3 Signed versus Unsigned Types

If our code uses addition, subtraction, and multiplication, then there is no performance difference between signed and unsigned operations , but there is a difference when it is division operation.
Example:
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
        MOV pc,r14   ; return r0
The compiler adds one to the sum before shifting by right if the sum is negative.In other words it replaces x/2 by the statement:
(x<0) ? ((x+1) >> 1): (x >> 1)

It must do this because x is signed.. In C on an ARM target, a divide by two is not a right shift if x is negative.
For example, −3 >> 1 =−2 but −3/2 = −1.
Division rounds towards zero, but arithmetic right shift rounds towards −∞.
It is more efficient to use unsigned types for divisions. The compiler converts unsigned power of two divisions directly to right shifts. For general divisions, the divide routine in the C library is faster for unsigned types.

Summary:
The Efficient Use of C Types
- For local variables held in registers,we shouldn't use a **char** or **short** type unless 8-bit or 16-bit modular arithmetic is necessary. Use the signed or **unsigned int** types instead. **Unsigned** types are faster when we use divisions.

- For array entries and global variables held in main memory, use the type with the smallest size possible to hold the required data. This saves memory footprint. The ARMv4 architecture is efficient at loading and storing all data widths provided we traverse arrays by incrementing the array pointer. Avoid using offsets from the base of the array with short type arrays, as LDRH does not support this.
- Use explicit casts when reading array entries or global variables into local variables, or writing local variables out to array entries. The casts make it clear that for fast operation we are taking a narrow width type stored in memory and expanding it to a wider type in the registers. Switch on implicit narrowing cast warnings in the compiler to detect implicit casts.
- Avoid implicit or explicit narrowing casts in expressions because they usually cost extra cycles. Casts on loads or stores are usually free because the load or store instruction performs the type casting.
- Avoid **char** and **short** types for function arguments or return values. Use the **int** type even if the range of the parameter is smaller. This prevents the compiler performing unnecessary casts.

3.3 C Looping Structures

In this section we will learn the most efficient ways to code **for** and **while** loops on the ARM. This section includes, loops with a fixed number of iterations, loops with a variable number of iterations and loop unrolling.

3.3.1 Loops with a Fixed Number of Iterations

This concept is explained with 64-word checksum routine.
The below code explains how the compiler treats a loop with incrementing count ie., 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 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,#1         ; 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
```

It takes three instructions to implement the for loop structure:
● An ADD to increment i
● A compare to check if i is less than 64
● A conditional branch to continue the loop if i < 64

This is not efficient. On the ARM, a loop should only use two instructions:
● A subtract to decrement the loop counter, which also sets the condition code flags on the result
● A conditional branch instruction

3.3.2 Loops Using a Variable Number of Iterations

In this case a **do-while** loop gives better performance and code density than a **for** loop.

Example: We pass in a variable N giving the Variable Number of Iterations as an argument and count down N until N=0 ,no need of extra loop counter i.

```
 int checksum_v8(int *data, unsigned int N)
{
int sum=0;
do
{
sum += *(data++);
} while (--N!=0);
return sum;
}
```

The compiler output is now

```
checksum_v8
        MOV r2,#0              ; sum = 0
checksum_v8_loop
        LDR r3,[r0],#4         ; r3 = *(data++)
        SUBS r1,r1,#1          ; N-- and set flags
        ADD r2,r3,r2           ; sum += r3
        BNE checksum_v8_loop              ; if (N!=0) goto loop
        MOV r0,r2              ; r0 = sum
        MOV pc,r14                    ; return r0
```

3.3.3 Loop Unrolling

While implementing the loop , there are some additional instructions in addition to body of the loop :: a subtract to decrement the loop count and a conditional branch.These instructions are called loop overhead.

On ARM7 or ARM9 processors the subtract takes one cycle and the branch three cycles, giving an overhead of four cycles per loop.

We can save some of these cycles by unrolling a loop, means,repeating the loop body several times, and reducing the number of loop iterations by the same proportion.

There are two thing we we need to ask when unrolling a loop:

■ How many times should we unroll the loop?

Only unroll loops that are important for the overall

performance of the application. Otherwise unrolling will increase the code size with little performance benefit. Unrolling may even reduce performance by evicting more important

code from the cache.

■ What if the number of loop iterations is not a multiple of the unroll amount?

We can try to arrange it so that array sizes are multiples of our unroll

amount. If this isn't possible, then we must add extra code to take care of the leftover cases.This increases the code size a little but keeps the performance high.

Summary:

For Writing Loops Efficiently we need to consider the below points

- Use loops that count down to zero. Then the compiler does not need to allocate a register to hold the termination value, and the comparison with zero is free.
- Use unsigned loop counters by default and the continuation condition i!=0 rather than i>0. This will ensure that the loop overhead is only two instructions.
- Use do-while loops rather than for loops when you know the loop will iterate at least once. This saves the compiler checking to see if the loop count is zero.
- Unroll important loops to reduce the loop overhead. Do not overunroll. If the loop overhead is small as a proportion of the total, then unrolling will increase code size and hurt the performance of the cache.
- Try to arrange that the number of elements in arrays are multiples of four or eight. You can then unroll loops easily by two, four, or eight times without worrying about the leftover array elements.

3.4 Register Allocation

The compiler attempts to allocate a processor register to each local variable we use in a C function.It will try to use the same register for different local variables if the use of the variables do not overlap. When there are more local variables than available registers, the compiler stores the excess variables on the processor stack. These variables are called **spilled or swapped out variables** since they are written out to memory .

Spilled variables are slow to access compared to variables allocated to registers.

To implement a function efficiently, we need to

- minimize the number of spilled variables
- ensure that the most important and frequently accessed variables are stored in registers.

Table 3.3 shows the standard register names and usage when following the ARM-Thumb procedure call standard (ATPCS), which is used in code generated by C compilers.

Table 3.3   C compiler register usage.

Register number	Alternate register names	ATPCS register usage
r0	a1	Argument registers. These hold the first four function
r1	a2	arguments on a function call and the return value on a
r2	a3	function return. A function may corrupt these registers and
r3	a4	use them as general scratch registers within the function.
r4	v1	General variable registers. The function must preserve the callee
r5	v2	values of these registers.
r6	v3	
r7	v4	
r8	v5	
r9	v6 sb	General variable register. The function must preserve the callee value of this register except when compiling for *read-write position independence* (RWPI). Then *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 limit checking. Then *r10* holds the stack limit address.
r11	v8 fp	General variable register. The function must preserve the callee value of this register except when compiling using a frame pointer. Only old versions of *armcc* use a frame pointer.
r12	ip	A general scratch register that the function can corrupt. It is useful as a scratch register for function veneers or other intraprocedure call requirements.
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.

In theory, the C compiler can assign 14 variables to registers without spillage. In practice, some compilers use a fixed register such as r12 for intermediate scratch working and do not assign variables to this register. Also, complex expressions require intermediate working registers to evaluate. Therefore, we should try to limit the internal loop of functions to using at most 12 local variables.

If the compiler does need to swap out variables, then it chooses which variables to swap out based on frequency of use.

Summary:

Efficient Register Allocation

- limit the number of local variables in the internal loop of functions to 12. The compiler should be able to allocate these to ARM registers.
- we can guide the compiler as to which variables are important by ensuring these variables are used within the innermost loop.

3.5 Function Calls

The ARM Procedure Call Standard (APCS) defines how to pass function arguments and return values in ARM registers.

Four Register Rule:

The first four integer arguments are passed in the first four ARM registers: r0, r1, r2, and r3. Subsequent integer arguments are placed on the full descending stack.Function return integer values are passed in r0.

Two-word arguments such as long long or double are passed in a pair of consecutive argument registers and returned in r0, r1.

Fig 3.1 ATPCS argument passing.
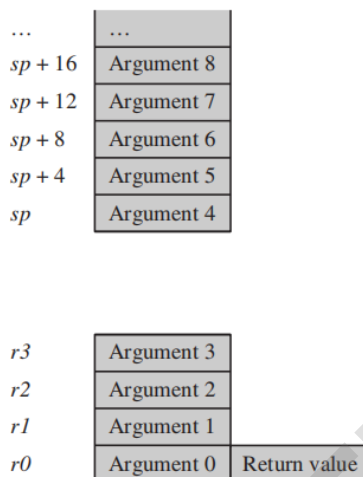
If our C function needs more than four arguments,it is always more efficient,if we group related arguments into structures, and pass a structure pointer rather than using multiple arguments.

Example: Insert N bytes(from Array data into a queue)

Case1: 5 Arguments

```
char *queue_bytes_v1(
char *Q_start,          /* Queue buffer start address */
char *Q_end,            /* Queue buffer end address */
char *Q_ptr,            /* Current queue pointer position */
char *data,             /* Data to insert into the queue */
unsigned int N)         /* Number of bytes to insert */
{
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
        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
```

Case2: using Structure

```
typedef struct {
char *Q_start;          /* Queue buffer start address */
char *Q_end;            /* Queue buffer end address */
char *Q_ptr;            /* Current queue pointer position */
} Queue;

void queue_bytes_v2(Queue *queue, char *data, unsigned int N)
{
char *Q_ptr = queue->Q_ptr;
char *Q_end = queue->Q_end;
do
{
*(Q_ptr++) = *(data++);
if (Q_ptr == Q_end)
{
```

```
Q_ptr = queue->Q_start;
}
} while (--N);
queue->Q_ptr = Q_ptr;
}
```

This compiles to
queue_bytes_v2

```
        STR r14,[r13,#-4]!                    ; save lr on the stack
        LDR r3,[r0,#8]              ; r3 = queue->Q_ptr
        LDR r14,[r0,#4]            ; r14 = queue->Q_end
queue_v2_loop
        LDRB r12,[r1],#1            ; r12 = *(data++)
        STRB r12,[r3],#1            ; *(Q_ptr++) = r12
        CMP r3,r14                      ; if (Q_ptr == Q_end)
        LDREQ r3,[r0,#0]           ; Q_ptr = queue->Q_start
        SUBS r2,r2,#1              ; --N and set flags
        BNE queue_v2_loop          ; if (N!=0) goto loop
        STR r3,[r0,#8]             ; queue->Q_ptr = r3
        LDR pc,[r13],#4            ; return
```

The queue_bytes_v2 is one instruction longer than queue_bytes_v1, but it is more efficient overall.

The second version has only three function arguments rather than

five. Each call to the function requires only three register setups. This compares with four register setups, a stack push, and a stack pull for the first version. There is a net saving of two instructions in function call overhead. It only needs to assign a single register to the Queue structure pointer, rather than three registers in the non structured case

There are other ways of reducing function call overhead
- The caller function need not preserve registers that it can see the callee doesn't corrupt. Therefore the caller function need not save all the ATPCS corruptible registers.
- If the callee function is very small, then the compiler can inline the code in the caller function. This removes the function call overhead completely.

Summary
For Calling Functions Efficiently we need to consider the below points

- Try to restrict functions to four arguments. This will make them more efficient to call. Use structures to group related arguments and pass structure pointers instead of multiple arguments.
- Define small functions in the same source file and before the functions that call them. The compiler can then optimize the function call or inline the small function.
- Critical functions can be inlined using the__inline keyword.

3.6 Pointer Aliasing

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

That means , If we write to one pointer, it will affect the value we read from the other pointer. In function, the compiler often does not know which pointer cause aliasing and which pointer not.

Example: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 = t0
        MOV pc,r14          ; return
```

If the pointer timer1 and step alias then different step value will be added to timer2. This can be avoided by creating a new local variable to hold the value of state->step so the compiler only performs a single load.

```
void timers_v3(State *state, Timers *timers)
{
int step = state->step;
timers->timer1 += step;
timers->timer2 += step;
}
```

CSE,SJCIT

Summary:

Avoiding Pointer Aliasing

- Do not rely on the compiler to eliminate common sub-expressions involving memory accesses. Instead create new local variables to hold the expression. This ensures the expression is evaluated only once.
- Avoid taking the address of local variables. The variable may be inefficient to access from then on.

5.7 Structure Arrangement

Every data type have alignment requirements (it is mandated by processor architecture, not by language). A processor will have processing word length as that of data bus size. On a 32-bit machine, the processing word size will be 4 bytes.

Fig 3.2 Data Alignment in Memory
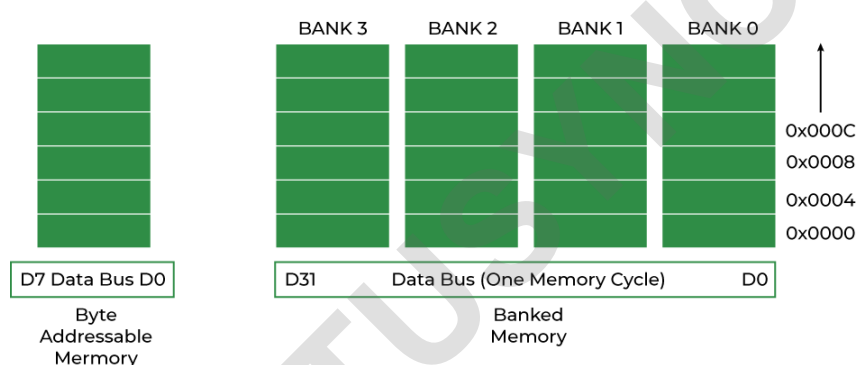
If an integer of 4 bytes is allocated on X address (X is a multiple of 4), the processor needs only one memory cycle to read the entire integer. Whereas, if the integer is allocated at an address other than a multiple of 4, it spans across two rows of the banks as shown in the below figure 3.3. Such an integer requires two memory read cycles to fetch the data.

Layout of misaligned data (0X01ABCDEF)
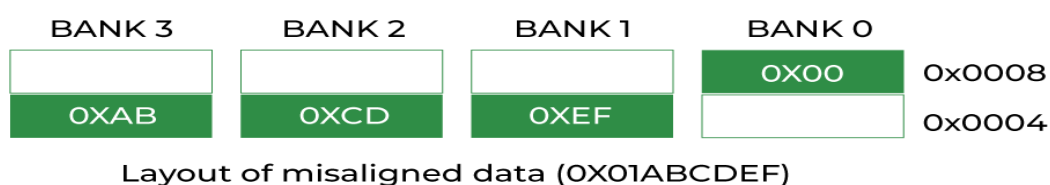
Load and store instructions are only guaranteed to load and store values with address aligned to the size of the access width.

Table 3.4 summarizes these restrictions.

CSE,SJCIT

Transfer size	Instruction	Byte address
1 byte	LDRB, LDRSB, STRB	any byte address alignment
2 bytes	LDRH, LDRSH, STRH	multiple of 2 bytes
4 bytes	LDR, STR	multiple of 4 bytes
8 bytes	LDRD, STRD	multiple of 8 bytes

Therefore ARM compilers will automatically align the start address of a structure to a multiple of the largest access width used within the structure (usually four or eight bytes) and align entries within structures to their access width by inserting padding.

Example:
struct {
char a;
int b;
char c;
short d;
}
For a little-endian memory system the compiler will lay this out adding padding to ensure that the next object is aligned to the size of that object:

Address	+3	+2	+1	+0
+0	pad	pad	pad	a
+4	b[31,24]	b[23,16]	b[15,8]	b[7,0]
+8	d[15,8]	d[7,0]	pad	c

To improve the memory usage, you should reorder the elements
struct {
char a;
char c;
short d;
int b;
}
This reduces the structure size from 12 bytes to 8 bytes, with the following new layout:

Address	+3	+2	+1	+0
+0	d[15,8]	d[7,0]	c	a
+4	b[31,24]	b[23,16]	b[15,8]	b[7,0]

The following rules generate a structure with the elements packed for maximum efficiency:

CSE,SJCIT

- Place all 8-bit elements at the start of the structure.
- Place all 16-bit elements next, then 32-bit, then 64-bit.
- Place all arrays and larger elements at the end of the structure.
- If the structure is too big for a single instruction to access all the elements, then group the elements into substructures. The compiler can maintain pointers to the individual substructures.

Summary

For Efficient Structure Arrangement we need to consider the below points

- Lay structures out in order of increasing element size. Start the structure with the smallest elements and finish with the largest.
- Avoid very large structures. Instead use a hierarchy of smaller structures.
- For portability, manually add padding (that would appear implicitly) into API structures so that the layout of the structure does not depend on the compiler.
- Beware of using enum types in API structures. The size of an enum type is compiler dependent.