USN

CMRIT
CELEBRATING 25 YEARS
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

Internal Assessment Test 2 – JULY
2024- SCHEME

Sub:	**MICROCONTROLLERS**				Sub Code:	BCS402	Branch:	CSE
Date:	10/07/24	Duration:	90 mins	Max Marks: 50	Sem / Sec:	IV Sem A/B/C		OBE

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1. a)	Explain how Registers are allocated to optimize the program.	[6]	CO3	L2

4 Marks:

The compiler attempts to allocate a processor register to each local variable you 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 (in a similar way virtual memory is swapped out to disk). Spilled variables are slow to access compared to variables allocated to registers.

To implement a function efficiently

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

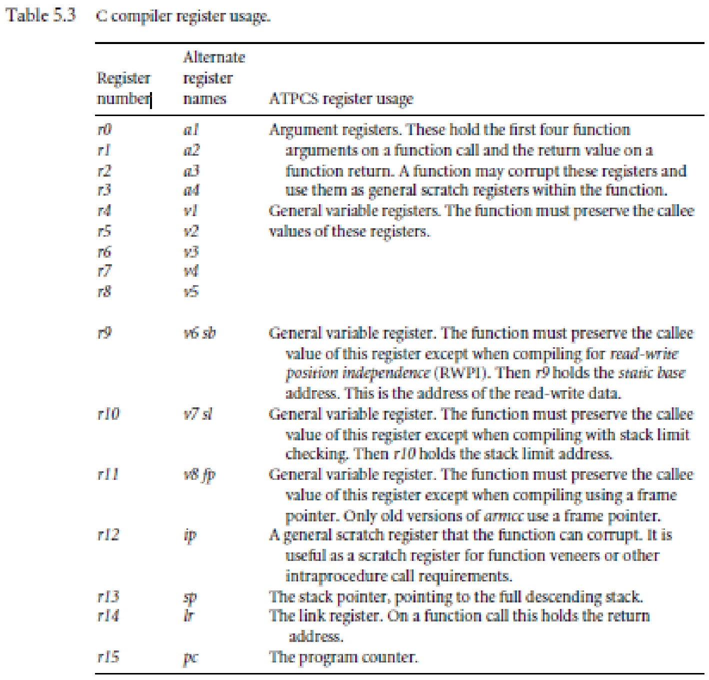


Table 5.3 C compiler register usage.

Register number	Alternate register names	ATPCS register usage
r0	*a1*	Argument registers. These hold the first four function arguments on a function call and the return value on a function return. A function may corrupt these registers and use them as general scratch registers within the function.
r1	*a2*	
r2	*a3*	
r3	*a4*	
r4	*v1*	General variable registers. The function must preserve the callee values of these registers.
r5	*v2*	
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.

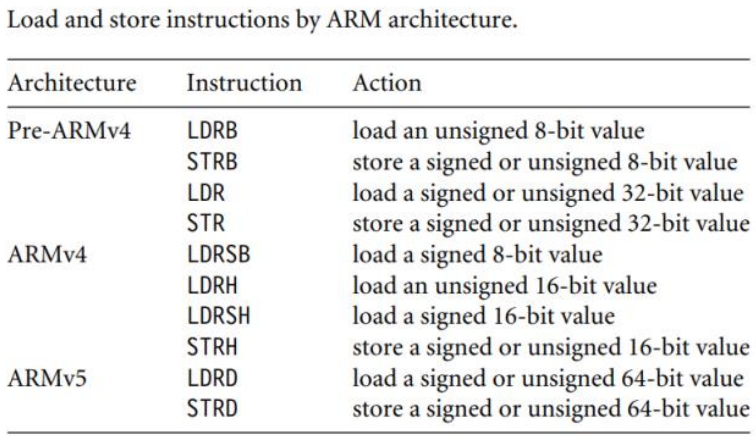
The register keyword in C hints that a compiler should allocate the given variable to a register. However, different compilers treat this keyword in different ways, and different architectures have a different number of available registers (for example, Thumb and ARM). Therefore we recommend that you avoid using register and rely on the compiler's normal register allocation routine.

2 Marks
Efficient Register Allocation:

Try to 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
You can guide the compiler as to which variables are important by ensuring these variables are used within the innermost loop.

1. b) Interpret the load/store instruction classes available in ARM architecture .Explain the actions performed by these instructions. [4] CO3 L2

Load and store instructions - 2 M
Action for each - 2 M



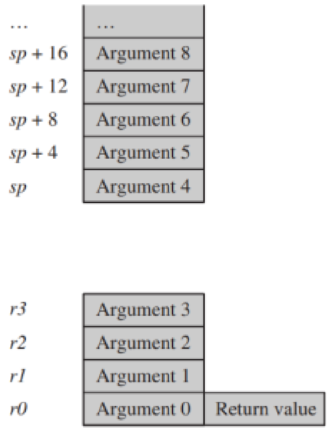
Load and store instructions by ARM architecture.

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

2 a) Explain four- register rule used for passing function arguments in ARM-Thumb Procedure Call Standard (ATPCS) with a proper example. [6] CO3 L2

Explanation - 4 M
Digram- 2 M
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, ascending in memory . Function return integer values are passed in r0.

This description covers only integer or pointer arguments. Two-word arguments such as long long or double are passed in a pair of consecutive argument registers and returned in r0, r1. The compiler may pass structures in registers or by reference according to command line compiler options.



The first point to note about the procedure call standard is the four-register rule. Functions with four or fewer arguments are far more efficient to call than functions with five or more arguments. For functions with four or fewer arguments, the compiler can pass all the arguments in registers. For functions with more arguments, both the caller and callee must access the stack for some arguments.Note that for C++ the first argument to an object method is the this pointer. This argument is implicit and additional to the explicit arguments.If your C function needs more than four arguments, or your C++ method has more than three explicit arguments, then it is almost always more efficient to use structures. Group related arguments into structures, and pass a structure pointer rather than multiple arguments. Which arguments are related will depend on the structure of your software.

2.b)	Describe loop unrolling.How can you overcome the loop-overhead using loop unrolling.
	Loop unrolling- 2M
	How to overcome overhead - 1M
	Example - 1M
	Each loop iteration costs two instructions in addition to the body of the loop: a subtract to decrement the loop count and a conditional branch.
	We call these instructions the 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.
	You can save some of these cycles by unrolling a loop—repeating the loop body several times, and reducing the number of loop iterations by the same proportion.
	Example:
	The following code unrolls our packet checksum loop by four times. We assume that the number of words in the packet N is a multiple of four.
	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;
	}
		[4]	CO3	L2

3 a)	Identify the issues encountered when porting C code to ARM. Any 5 issues with explanation- 5 M The char type The int type. Unaligned data pointers Endian assumptions Function prototyping Use of bit-fields Use of enumerations Inline assembly The volatile keyword.	[5]	CO3	L3

3. b)	Define Pointer aliasing.Analyze the concept of pointer-aliasing by using the code given below.
	void timers_v1(int *timer1, int *timer2, int *step)
	{
	*timer1 += *step;
	*timer2 += *step;
	}
	Definition: 1 M
	Analysis of the code - 4 M
	Two pointers are said to alias when they point to the same address. If you write to one pointer, it will affect the value you read from the other pointer. In a function, the compiler often doesn’t know which pointers can alias and which pointers can’t. The compiler must be very pessimistic and assume that any write to a pointer may affect the value read from any other pointer, which can significantly reduce code efficiency
	The above c code compiles to
		[5]	CO3	L4

	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 Note that the compiler loads from step twice. Usually a compiler optimization called common subexpression elimination would kick in so that *step was only evaluated once, and the value reused for the second occurrence. However, the compiler can’t use this optimization here. The pointers timer1 and step might alias one another. In other words, the compiler cannot be sure that the write to timer1 doesn’t affect the read from step. In this case the second value of *step is different from the first and has the value *timer1. This forces the compiler to insert an extra load instruction			

4 a) [08] CO3 L3

Consider the following C code to calculate the Checksum of a data packet containing 64 words.Illustrate the compiler output generated for the same code shown below.Summarize the drawbacks of the compiler output.

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

Compiler output: 5 M

```
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 ; i++
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
```

Drawback of the output: 3 M

The loop now has extra three instructions.
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. (See Table 5.1.)

■ 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.

4 b) [02] CO3 L1

List the C compiler datatype mappings for ARM.

Mapping c data types - 2M

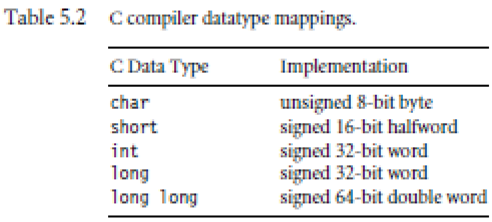


Table 5.2 C compiler datatype mappings.

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

5. a) [10] CO3 L3

Write a C program for ARM to find the factorial of a number.Also develop an Assembly Language Program for the same.

Factorial C-Program -5 M

```
#include<lpc21xx.h>
int main(void) {
        unsigned long n=5, fact=1;
        unsigned char i;
        if (n==0) {
                fact=1;
        }
        else if(n>0) {
                for (i=1;i<=n;i++)
                {
                        fact=fact*i;
                }
        }
}
```

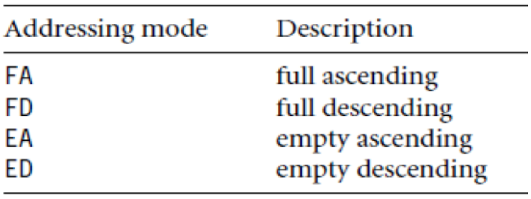
Factorial ALP - 5 M

```
AREA FACTORIAL, CODE, READONLY
ENTRY
                MOV  R0, #5
                MOV  R1, #1
                MOV  R2 #1
LOOP
               MUL R1,R2,R1
               ADD R2,R2,#1
               CMP R2,R0
               BLE LOOP
                        STOP  B STOP

        END
```

6.a) Explain full descending Stack with proper example. [4] CO2 L2

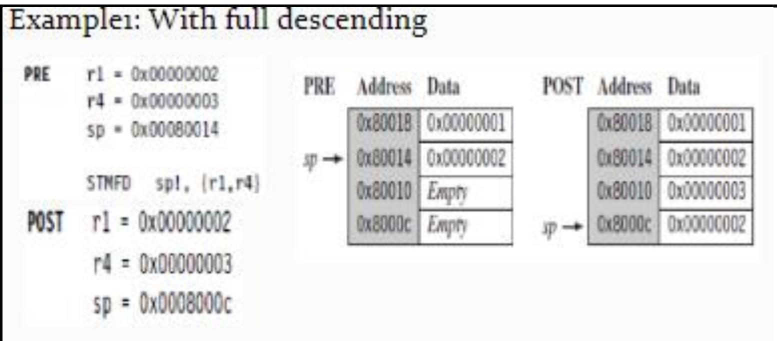
A full descending stacks grow towards lower memory addresses.
When you use a full stack (F), the stack pointer sp points to an address that is the last used or full location (i.e., sp points to the last item on the stack). In contrast, if you use an empty stack (E) the sp points to an address that is the first unused or empty location (i.e., it points after the last item on the stack).

Addressing modes for stack operation



Addressing mode	Description
FA	full ascending
FD	full descending
EA	empty ascending
ED	empty descending

Example:



Example1: With full descending

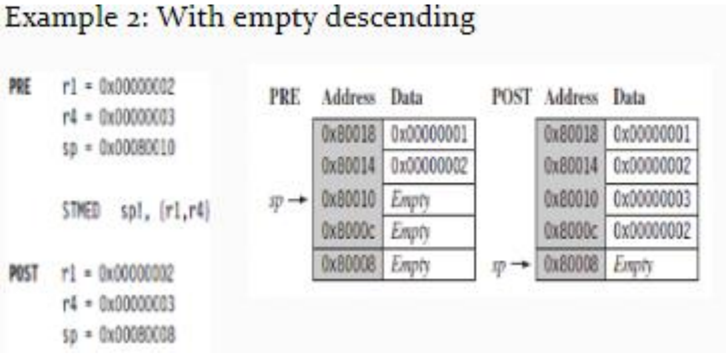
```
PRE   r1 = 0x00000002
      r4 = 0x00000003
      sp = 0x00080014

      STMFD  sp!, {r1,r4}

POST  r1 = 0x00000002
      r4 = 0x00000003
      sp = 0x0008000c
```

PRE

	Address	Data
	0x80018	0x00000001
sp →	0x80014	0x00000002
	0x80010	Empty
	0x8000c	Empty

POST

	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
sp →	0x8000c	0x00000002



Example 2: With empty descending

```
PRE   r1 = 0x00000002
      r4 = 0x00000003
      sp = 0x00080010

      STMED  sp!, {r1,r4}

POST  r1 = 0x00000002
      r4 = 0x00000003
      sp = 0x00080008
```

PRE

	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
sp →	0x80010	Empty
	0x8000c	Empty
	0x80008	Empty

POST

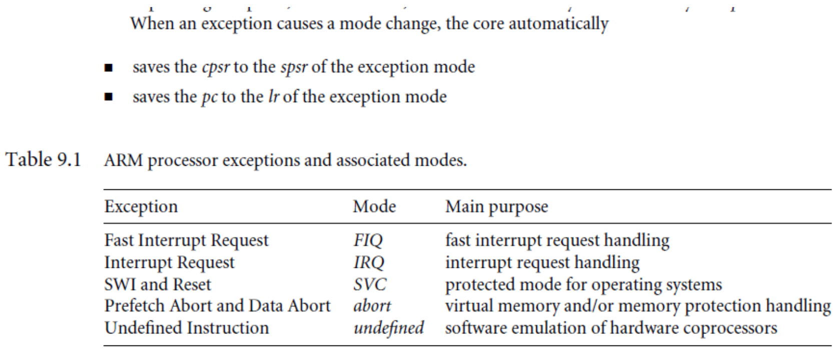
	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
	0x8000c	0x00000002
sp →	0x80008	Empty

6. b) With a neat diagram explain ARM processor exceptions and modes. [6] CO4 L2

An exception is any condition that needs to halt normal execution of the instructions

Examples
•Resetting ARM core
•Failure of fetching instructions
•HWI
•SWI

Exception - 1M
Modes - 5 M

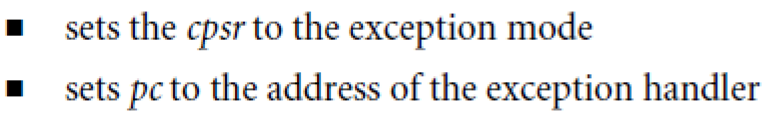


When an exception causes a mode change, the core automatically

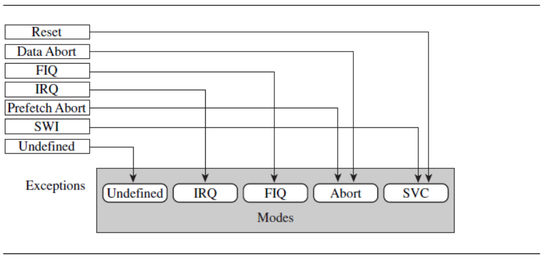
- saves the *cpsr* to the *spsr* of the exception mode
- saves the *pc* to the *lr* of the exception mode

Table 9.1 ARM processor exceptions and associated modes.

Exception	Mode	Main purpose
Fast Interrupt Request	*FIQ*	fast interrupt request handling
Interrupt Request	*IRQ*	interrupt request handling
SWI and Reset	*SVC*	protected mode for operating systems
Prefetch Abort and Data Abort	*abort*	virtual memory and/or memory protection handling
Undefined Instruction	*undefined*	software emulation of hardware coprocessors



- sets the *cpsr* to the exception mode
- sets *pc* to the address of the exception handler

	 Reset Data Abort FIQ IRQ Prefetch Abort SWI Undefined Exceptions Undefined IRQ FIQ Abort SVC Modes Figure 9.1 Exceptions and associated modes.			

Faculty Signature CCI Signature HOD Signature