USN

Internal Assessment Test 2 – MAY 2025

Sub:	MICROCONTROLLERS					Sub Code:	BCS402	Branch:	CSE
Date:	26/05/25	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 Arm c compiler data types. **ANS:** Compilers *use* the datatype mappings which is shown in below table- C compiler datatype mappings.	[2]	CO2	L2

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

1. b)	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;
	}
	
	Ans:
	The compiler output generated for the same code are as follows—
		[8]	CO4	L2

```
        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
```

The loop is now three instructions longer which are
ADD r3,r2,r1,LSL #1
MOV r0,r0,LSL #16
MOV r0,r0,ASR #16

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.

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.

How to overcome the drawback

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.

However, 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.

2 a)	Briefly explain steps to enable IRQ and FIQ mode in ARM processor. **Ans:** IRQ and FIQ exceptions only occur when a specific interrupt mask is cleared in the *cpsr*. An IRQ or FIQ exception causes the processor hardware to go through a standard procedure (provided the interrupts are not masked): 1. The processor changes to a specific interrupt request mode, which reflects the interrupt being raised. 2. The previous mode's *cpsr* is saved into the *spsr* of the new interrupt request mode. 3. The *pc* is saved in the *lr* of the new interrupt request mode. 4. *Interrupt/s are disabled*—either the IRQ or both IRQ and FIQ exceptions are disabled in the *cpsr*. This immediately stops another interrupt request of the same type being raised. 5. The processor branches to a specific entry in the vector table. Following Table shows how IRQ and FIQ interrupts are enabled. The procedure uses three ARM instructions. The first instruction MRS copies the contents of the *cpsr* into register *r1*. The second instruction clears the IRQ or FIQ mask bit. The third instruction then copies the updated contents in register *r1* back into the *cpsr*, enabling the interrupt request. The postfix _c identifies that the bit field being updated is the control field bit [7:0] of the *cpsr*. The interrupt request is either enabled or disabled only once the MSR instruction has completed the execution stage of the pipeline. Interrupts can still be raised or masked prior to the MSR completing this stage.	[6]	CO3	L3

Enabling an interrupt.

cpsr value	IRQ	FIQ
Pre	*nzcvqj**I**Ft_SVC*	*nzcvqjI**F**t_SVC*

Code	`enable_irq`
	`MRS r1, cpsr`
	`BIC r1, r1, #0x80`
	`MSR cpsr_c, r1`
		`enable_fiq`
		`MRS r1, cpsr`
		`BIC r1, r1, #0x40`
		`MSR cpsr_c, r1`

Post	*nzcvqj**i**Ft_SVC*	*nzcvqjI**f**t_SVC*

Disabling an interrupt.

cpsr	IRQ	**FIQ**
Pre	*nzcvqj**i**ft_SVC*	*nzcvqji**f**t_SVC*

Code	`disable_irq`
	`MRS r1, cpsr`
	`ORR r1, r1, #0x80`
	`MSR cpsr_c, r1`
		`disable_fiq`
		`MRS r1, cpsr`
		`ORR r1, r1, #0x40`
		`MSR cpsr_c, r1`

Post	*nzcvqj**I**ft_SVC*	*nzcvqji**F**t_SVC*

2.b) Explain full descending Stack with proper example. [4] CO2 L2

Ans:
The ARM architecture uses the load-store multiple instructions to carry out stack operations. The *pop* operation (removing data from a stack) uses a load multiple instruction; similarly, the *push* operation (placing data onto the stack) uses a store multiple instruction. When using a stack we have to decide whether the stack will grow up or down in memory. A stack is either *ascending* (A) or *descending* (D). Ascending stacks grow towards higher memory addresses; in contrast, descending stacks grow towards lower memory addresses. When we 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 we 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).

ARM has specified an ARM-Thumb Procedure Call Standard (ATPCS) that defines how routines are called and how registers are allocated. In the ATPCS, stacks are defined as being full descending stacks. Thus, the LDMFD and STMFD instructions provide the pop and push functions, respectively.

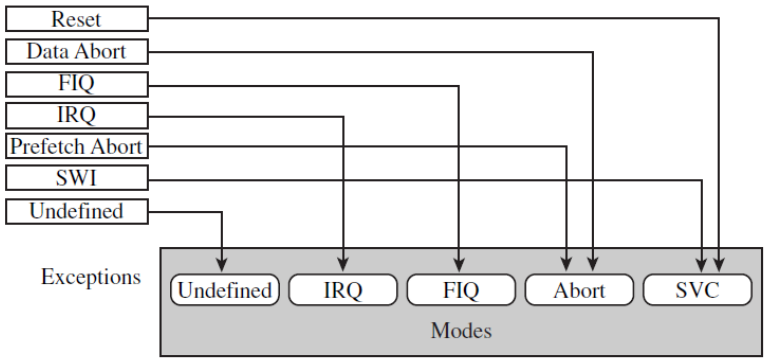
The STMFD instruction pushes registers onto the stack, updating the *sp*. Figure 3.7 shows a push onto a full descending stack. We can see that when the stack grows the stack pointer points to the last full entry in the stack.
Example--
PRE
r1 = 0x00000002
r4 = 0x00000003
sp = 0x00080014
STMFD sp!, {r1,r4}

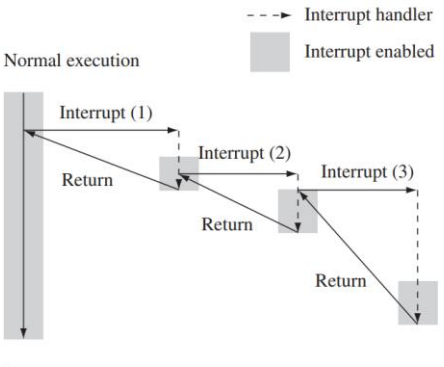
STMFD instruction—full stack push operation

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

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

POST
r1 = 0x00000002

	r4 = 0x00000003 sp = 0x0008000c			
3 a)	With a neat diagram explain ARM processor exceptions and modes. **Ans:** Each exception causes the core to enter a specific mode. In addition, any of the ARM processor modes can be entered manually by changing the *cpsr*. *User* and *system* mode are the only two modes that are not entered by a corresponding exception. 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 ARM processor exceptions and associated modes. **Figure shows a simplified view of exceptions and associated modes.**  • sets the *cpsr* to the exception mode • sets *pc* to the address of the exception handler	[6]	CO4	L2
3. b)	What is interrupt latency and how software handler can minimize the interrupt latency. **Ans:** The interval of time from an external interrupt request signal being raised to the first fetch of an instruction of a specific interrupt service routine (ISR). Interrupt latency depends on a combination of hardware and software. System architects must balance the system design to handle multiple simultaneous interrupt sources and minimize interrupt latency. **The first method is to use a nested interrupt handler** **Nested interrupt handler**– This allows further interrupts to occur even when currently servicing an existing interrupt This is achieved by reenabling the interrupts as soon as the interrupt source has been serviced (so it won't generate more interrupts) but before the interrupt handling is complete. Once a nested interrupt has been serviced, then control is relinquished to the original interrupt service routine.	[4]	CO4	L1



gure 9.3 A three-level nested interrupt.

The second method involves prioritization-- In this case program the interrupt controller

	to ignore interrupts of the same or lower priority than the interrupt we are handling, so only a higher-priority task can interrupt wer handler. The processor spends time in the lower-priority interrupts until a higher-priority interrupt occurs. Therefore higher-priority interrupts have a lower average interrupt latency than the lower-priority interrupts, which reduces latency by speeding up the completion time on the critical time-sensitive interrupts.			

4 a)	Write a C program for ARM micro controller to sort the numbers in ascending order using bubble sort
	
	Ans:
	#include<lpc21xx.h>
	int main(void)
	{
	unsigned long int temp, arr[4]= {0x00000001, 0x00000002, 0x00000004, 0x00000003};
	unsigned char i,j,n=4;
	for (i=0;i<n-1;i++)
	{
	for (j=0;j<n-1;j++)
	{
	if (arr[j]<arr[j+1])
	{
	temp = arr[j];
	arr[j]=arr[j+1];
	arr[j+1]= temp;
	}
	}
	}
	}
		[05]	CO3	L3

4 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;
	}
	
	Ans: Two pointers are said to *alias* when they point to the same address. If we write to one pointer, it will affect the value we read from the other pointer.
	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 = t0
	MOV pc,r14 ; return
	
	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
		[05]	CO3	L4

	*step is different from the first and has the value *timer1.This forces the compiler to insert an extra load instruction.			
5. a)	Describe the features of Red Hat Red Boot firmware tool. **Ans:** RedBoot is a firmware tool developed by Red Hat. It is provided under an open source license with no royalties or upfront fees. RedBoot is designed to execute on different CPUs (for instance, ARM, MIPS, SH, and so on). It provides both debug capability through GNU Debugger (GDB), as well as a bootloader. The RedBoot software core is based on a HAL. **RedBoot supports these main features:** *Communication*—configuration is over serial or Ethernet. For serial, X-Modem protocol is used to communicate with the GNU Debugger (GDB). For Ethernet, TCP is used to communicate with GDB. RedBoot supports a range of network standards, such as *bootp*, *telnet*, and *tftp*. *Flash ROM memory management*—provides a set of filing system routines that can download, update, and erase images in flash ROM. In addition, the images can either be compressed or uncompressed. ■ *Full operating system support*—supports the loading and booting of Embedded Linux, Red Hat eCos, and many other popular operating systems. For Embedded Linux, RedBoot supports the ability to define parameters that are passed directly to the kernel upon booting	[06]	CO4	L2
5. b)	Illustrate the steps in the execution flow of sandstone code structure. **Ans**: Sandstone consists of a single assembly file. The file structure is broken down into a number of steps, where each step corresponds to a stage in the execution flow of Sandstone Sandstone execution flow. Step — Description 1 — Take the Reset exception 2 — Start initializing the hardware 3 — Remap memory 4 — Initialize communication hardware 5 — Bootloader—copy payload and relinquish control **Step 1: Take the Reset Exception** Execution begins with a Reset exception. Only the reset vector entry is required in the default vector table. It is the very first instruction executed. reset vector is used to move the execution flow to the second stage. **Step 2: Start Initializing the Hardware** The primary phase in initializing hardware is setting up system registers. These registers have to be set up before accessing the hardware. For example, the ARM Evaluator-7T has a seven-segment display, which we have chosen to be used as a feedback tool to indicate that the firmware is active. Before we can set up the segment display, we have to position the base address of the system registers to a known location **Step 3: Remap Memory** One of the major activities of hardware initialization is to set up the memory environment. Sandstone is designed to initialize SRAM and remap memory. This process occurs fairly early on in the initialization of the system. **Step 4: Initialize Communication Hardware** Communication initialization involves configuring a serial port and outputting a standard banner. The banner is used to show that the firmware is fully functional and memory has been successfully remapped. N.B—if mark is 10 write the the code for each step.	[04]	CO4	L2
6.a)	Explain the swap instruction with an example code. **Ans:** The swap instruction is a special case of a load-store instruction. It swaps the contents of memory with the contents of a register. This instruction is an *atomi operation*—it reads and writes a location in the same bus operation, preventing any other instruction from reading or writing to that location until it completes.	[04]	CO2	L2

Syntax: SWP{B}{<cond>} Rd,Rm,[Rn]

SWP	swap a word between memory and a register	*tmp = mem32[Rn]* *mem32[Rn] = Rm* *Rd = tmp*
SWPB	swap a byte between memory and a register	*tmp = mem8[Rn]* *mem8[Rn] = Rm* *Rd = tmp*

Example—

```
PRE     mem32[0x9000] = 0x12345678
        r0 = 0x00000000
        r1 = 0x11112222
        r2 = 0x00009000

        SWP    r0, r1, [r2]

POST    mem32[0x9000] = 0x11112222
        r0 = 0x12345678
        r1 = 0x11112222
        r2 = 0x00009000
```

6. b) [06] CO2 L2

Given:
PRE:
r1 = 0x00000002,
r4 = 0x00000003,
Show the stack content and register contents after execution of following instructions.

STMFD sp! , {r1, r4} sp = 0x00080014.

STMED sp! , {r1, r4} sp = 0x00080010.

Ans
This instruction:

1. Decrements sp by 4 * number of registers → 2 registers ⇒ 8 bytes.
2. Stores r1 and r4 at the new sp (in order: r1 first, then r4).
3. Updates the stack pointer (due to the !).

i) New SP = 0x00080014 - 8 = 0x0008000C

Memory content after execution:

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

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

Registers after execution:
sp = 0x0008000C
r1 = 0x00000002
r4 = 0x00000003
ii) Now, **Initial sp = 0x00080010** (as given before this instruction).
Again:

- 2 registers → 8 bytes to be stored
- New SP = 0x00080010 - 8 = **0x00080008**

- Values stored in order: r1, r4

Memory content after execution:

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*

Registers after execution:

sp = 0x00080008

r1 = 0x00000002

r4 = 0x00000003

Faculty Signature CCI Signature HOD Signature