CMR INSTITUTE OF TECHNOLOGY

USN

Internal Assessment Test - II

Sub:	MICROCONTROLLERS						Code:	BCS402
Date:	10-07-2024	Duration:	90 mins	Max Marks:	50	Sem:	4th C	Branch:	CS(DS)

Answer Any FIVE FULL Questions

		Marks	OBE	
			CO	RBT
1	a. Explain Co-Processor instructions of ARM Processor.	6	CO2	L2
	b. Discuss SWAP instruction with an example.	4	C03	L2
2	Explain code optimization, profiling and cycle counting.	10	CO2	L2
3	Write a note on Instruction scheduling with respect to ARM processor.	10	CO3	L1
4	Discuss how registers are allocated to optimize the program.	10	CO3	L2

CCI **CI** **HOD**

CMR INSTITUTE OF TECHNOLOGY

USN

Internal Assessment Test - II

Sub:	MICROCONTROLLERS						Code:	BCS402
Date:	10-07-2024	Duration:	90 mins	Max Marks:	50	Sem:	4th C	Branch:	CS(DS)

Answer Any FIVE FULL Questions

		Marks	OBE	
			CO	RBT
1	a. Explain Co-Processor instructions of ARM Processor.	6	CO2	L2
	b. Discuss SWAP instruction with an example.	4	C03	L2
2	Explain code optimization, profiling and cycle counting.	10	CO2	L2
3	Write a note on Instruction scheduling with respect to ARM processor.	10	CO3	L1
4	Discuss how registers are allocated to optimize the program.	10	CO3	L2

CCI **CI** **HOD**

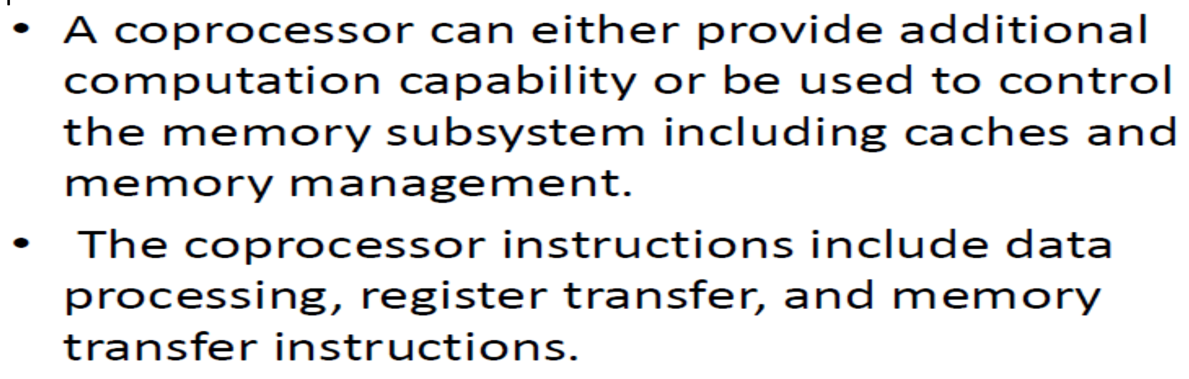
5	If r0=0x00000000, r1=0x00090000 mem32[0x00090000]=0x01010101 and mem32[00090004]=0x02020202 Find the content of the register r0, r1 after the following instructions are executed in isolation. (i) LDR r0, [r1, #4] (ii) LDR r0, [r1, #4]! (iii) LDR r0,[r1],#4	10	CO2	L3
6	Write ALP program for ARM7 to find number of zeroes and number of ones in a 32-bit number.	10	CO2	L3
7	Explain Portability issues.	10	CO3	L2

5	If r0=0x00000000, r1=0x00090000 mem32[0x00090000]=0x01010101 and mem32[00090004]=0x02020202 Find the content of the register r0, r1 after the following instructions are executed in isolation. (i) LDR r0, [r1, #4] (ii) LDR r0, [r1, #4]! (iii) LDR r0,[r1],#4	10	CO2	L3
6	Write ALP program for ARM7 to find number of zeroes and number of ones in a 32-bit number.	10	CO2	L3
7	Explain Portability issues.	10	CO3	L2

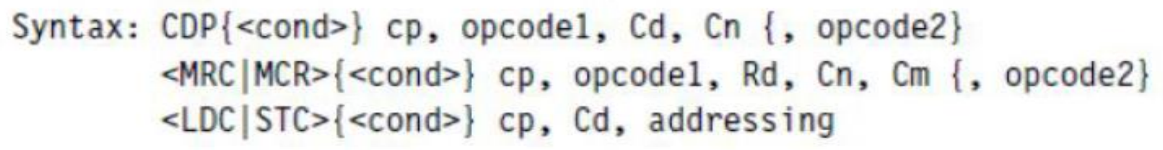
MICROCONTROLLERS (BCS402) IAT II Solution

1. a

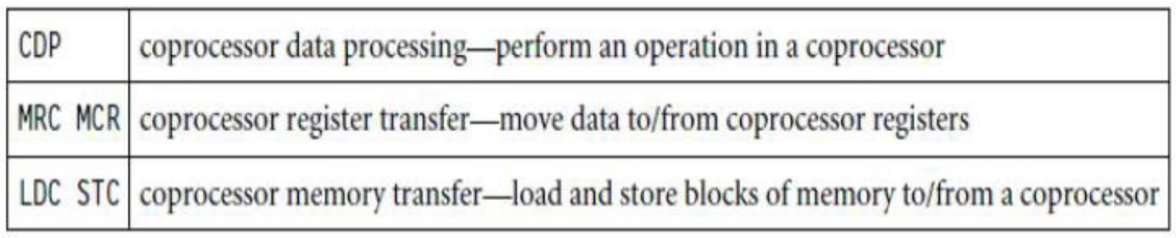
Coprocessor Instructions:



- A coprocessor can either provide additional computation capability or be used to control the memory subsystem including caches and memory management.
- The coprocessor instructions include data processing, register transfer, and memory transfer instructions.

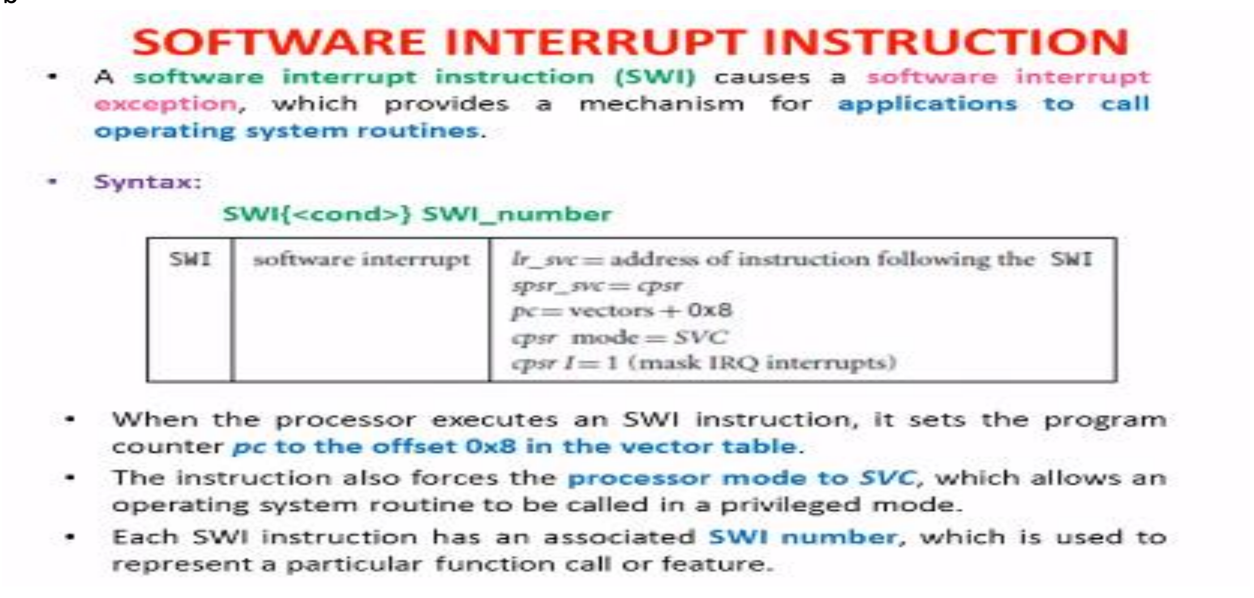


```
Syntax: CDP{<cond>} cp, opcode1, Cd, Cn {, opcode2}
        <MRC|MCR>{<cond>} cp, opcode1, Rd, Cn, Cm {, opcode2}
        <LDC|STC>{<cond>} cp, Cd, addressing
```



CDP	coprocessor data processing—perform an operation in a coprocessor
MRC MCR	coprocessor register transfer—move data to/from coprocessor registers
LDC STC	coprocessor memory transfer—load and store blocks of memory to/from a coprocessor

b



SOFTWARE INTERRUPT INSTRUCTION

- A software interrupt instruction (SWI) causes a software interrupt exception, which provides a mechanism for applications to call operating system routines.
- Syntax:
  SWI{<cond>} SWI_number

SWI	software interrupt	*lr_svc* = address of instruction following the SWI *spsr_svc* = *cpsr* *pc* = vectors + 0x8 *cpsr* mode = *SVC* *cpsr I* = 1 (mask IRQ interrupts)

- When the processor executes an SWI instruction, it sets the program counter *pc* to the offset 0x8 in the vector table.
- The instruction also forces the processor mode to *SVC*, which allows an operating system routine to be called in a privileged mode.
- Each SWI instruction has an associated SWI number, which is used to represent a particular function call or feature.

.

2.

Compiler optimization is a list of techniques implemented in compilers to generate "better" code,

faster or smaller.
I. Local Variable Types:converting local variables from types char or short to type int increases performance and reduces code size.
Ii. Function Argument Types:Char or short type function arguments and return values increase code size and decrease performance.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.
Iii. C Looping Structures
Iv. Register Allocation
V. Funcion call
Vi. Pointer Aliasing

Profiling is a form of dynamic program analysis that measures various aspects of program execution to identify performance bottlenecks and opportunities for optimization.

Key aspects of profiling include:

- **Execution Time:** Measures how much time the program spends in each function or line of code.
- **Memory Usage:** Tracks memory allocation and deallocation to identify memory leaks or excessive memory usage.
- **Function Call Counts:** Counts how many times each function is called.
- **I/O Operations:** Measures the number and duration of input/output operations.

Profiling tools often provide a visual representation of the data, helping developers pinpoint areas where the program is consuming excessive resources.

Cycle Counting is a technique often used in embedded systems and hardware design to measure the number of clock cycles a particular operation or set of operations takes to execute.

Key aspects of cycle counting include:

- **Instruction Cycle Counting:** Measures the number of clock cycles taken by individual instructions or sequences of instructions.
- **Pipeline Analysis:** Analyzes how instructions are processed in the pipeline stages of a CPU, identifying stalls, hazards, and inefficiencies.
- **Resource Utilization:** Evaluates how efficiently the hardware resources (like CPU, memory, and I/O) are being utilized.

Cycle counting is critical in systems where performance and power efficiency are crucial, such as real-time systems, embedded systems, and high-performance computing.

3.

Instruction scheduling is a compiler optimization technique used in computer architecture to improve the performance of a program by reordering the instructions in such a way that maximizes the utilization of the CPU's execution units and minimizes stalls. The primary goals of instruction scheduling are to reduce pipeline stalls, improve instruction-level parallelism, and optimize the usage of processor resources.

key aspects of instruction scheduling

1. **Dependencies:**

- **Data Dependencies:** Instructions that depend on the result of previous instructions.
- **Control Dependencies:** Instructions that depend on the outcome of branch instructions.
- **Resource Dependencies:** Instructions that require the same computational resources.

2. Pipeline Stalls:

- Occur when the CPU pipeline cannot continue to execute the next instruction because the current instruction has not yet completed. Instruction scheduling aims to minimize these stalls

3. Types of Instruction Scheduling:

- **Static Scheduling:** Performed at compile time by the compiler. It does not change during execution.
- **Dynamic Scheduling:** Performed at runtime by the processor. It can adapt to changing conditions and resource availability

4. Register Allocation

- 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, you need to

■ minimize the number of spilled variables

■ ensure that the most important and frequently accessed variables are stored in registers

- ❖ Provided the compiler is not using software stack checking or a frame pointer, then the C compiler can use registers r0 to r12 and r14 to hold variables. It must save the callee values of r4 to r11 and r14 on the stack if using these registers.
- ❖ 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, to ensure good assignment to registers, you should try to limit the internal loop of functions to using at most 12 local variables.

■ 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.

5. If r0=0x00000000, r1=0x00090000 mem32[0x00090000]=0x01010101 and

mem32[00090004]=0x02020202 Find the content of the register r0, r1 after the following

instructions are executed in isolation. (i) LDR r0,[r1,#4] (ii) LDR r0,[r1,#4]! (iii) LDR r0,[r1],#4

Answer:

(i) After the execution, r0=0x02020202 and r1=0x00090000

(ii) (ii) After the execution, r0=0x02020202 and r1=0x00090004

(iii) (iii) After the execution, r0=0x01010101 and r1=0x00090004

6.

```
AREA CountOneZero, CODE, READONLY
          ENTRY
          LDR R0, =0X40000000
          LDR R1, [R0]
          MOV R2, #32
AGAIN RORS R1, #1
          BCS ONES
         ADD R3, R3, #1
         B NEXT
ONES ADD R4, R4, #1
NEXT ADD R2, R2, #-1
        CMP R2, #0
        BNE AGAIN
        ADD R0, R0, #4
        STRB R3, [R0]
        STRB R4, [R0, #1]
        MOV R0, #0X18
        LDR R1, =0X20026
        SVC #0123456
        END
```

7.

Portability Issues-char type

On the ARM, char is unsigned rather than signed as for many other processors.

A common problem concerns loops that use a char loop counter i and

the continuation condition i ≥ 0, they become infinite loops. In this situation, armcc produces a warning of unsigned comparison with zero.

You should either use a compiler option to make char signed or change loop counters to type int.

Portability Issues-int type

Some older architectures use a 16-bit int.

May cause problems when moving to ARM's 32-bit int type although this is rare nowadays.

Expressions are promoted to an int type before evaluation.

Therefore if i = -0x1000,

the expression i == 0xF000 is true on a 16-bit machine

but false on a 32- bit machine.

Portability Issues-Unaligned data pointers

Some processors support the loading of short and int typed values from unaligned addresses.

A C program may manipulate pointers directly so that they become unaligned.

for example, by casting a char * to an int *.

ARM architectures up to ARMv5TE do not support unaligned pointers.

To detect them, run the program on an ARM with an alignment checking trap.

For example, you can configure the ARM720T to data abort on an unaligned access.

Portability Issues-Endian assumptions

C code may make assumptions about the endianness of a memory

system, for example, by casting a char * to an int *.

If you configure the ARM for the same endianness the code is expecting, then there is no issue.

Otherwise, you must remove endian-dependent code sequences and replace them by endian-independent ones.

Portability Issues-Function prototyping

The armcc compiler passes arguments narrow, that is, reduced

to the range of the argument type.

If functions are not prototyped correctly, then the function may return the wrong answer.

Other compilers that pass arguments wide may give the correct answer even if the function prototype is incorrect.

Always use ANSI prototypes.

Portability Issues-Use of bit-fields

The layout of bits within a bit-field is implementation and endian

dependent. If C code assumes that bits are laid out in a certain order, then the code is not portable.

Portability Issues-Use of enumerations

Although enum is portable, different compilers allocate different

numbers of bytes to an enum.

The gcc compiler will always allocate four bytes to an enum

type. The armcc compiler will only allocate one byte if the enum takes only eight-bit values.

Therefore you can’t cross-link code and libraries between different compilers if you use enums in an API structure.

Portability Issues-Inline assembly

Using inline assembly in C code reduces portability between

architectures.

You should separate any inline assembly into small inline functions

that can easily be replaced.

It is also useful to supply reference, plain C implementations

of these functions that can be used on other architectures, where this is possible.

Portability Issues-The volatile keyword

Use the volatile keyword on the type definitions of ARM memory-mapped peripheral locations.

This keyword prevents the compiler from optimizing away the memory access.

It also ensures that the compiler generates data access of the correct type.

For example, if you define a memory location as a volatile short

type, then the compiler will access it using 16-bit load and store instructions LDRSH and STRH.