

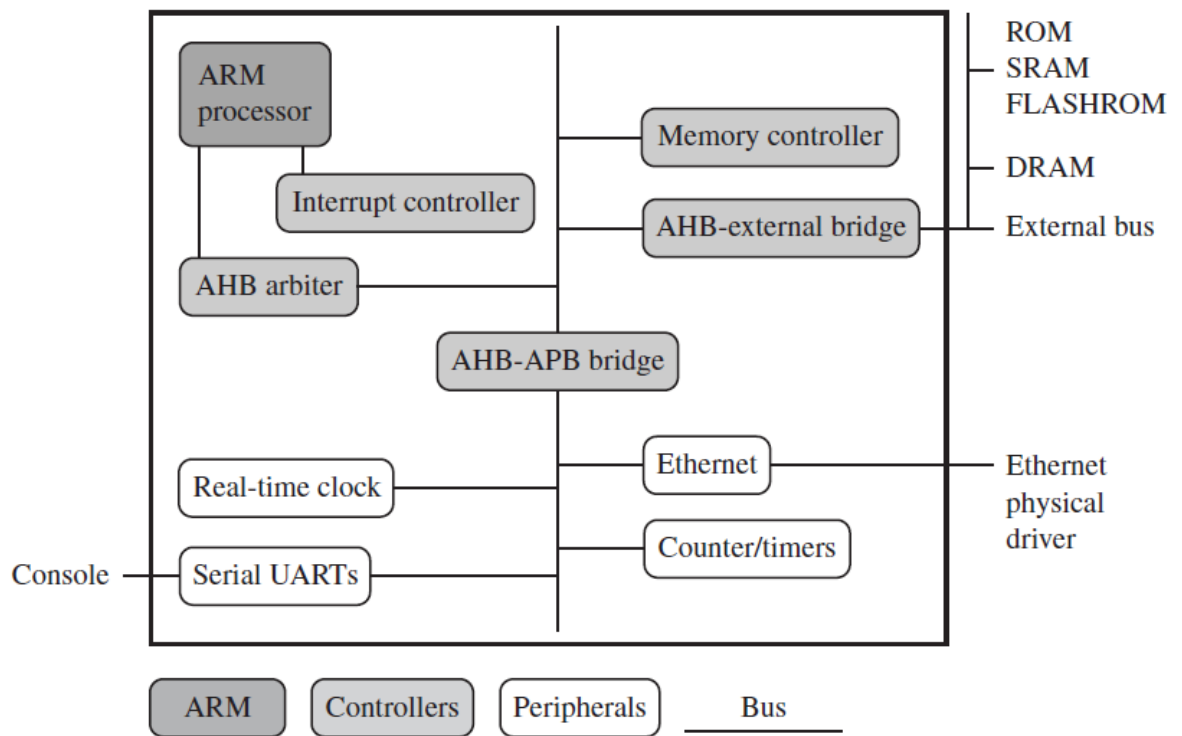
Internal Assessment Test – I

Sub:	Microcontrollers & Embedded Systems						Code:	BCO601	
Date:	24/ 03/ 2023	Duration:	90 mins	Max Marks:	50	Sem:	6 th	Branch:	CSE(AIML)
Answer Any FIVE FULL Questions									
								OBE	
								Marks	
								CO	RBT
1.	Differentiate between RISC and CISC. Explain the Architecture of an ARM Embedded device with the help of a neat diagram						[04] [06]	CO1	L3
2.	Explain the 5-stage pipelining used in ARM. Which ARM core implements 5-stage pipelining.						[10]	CO1	L3
3.	What are the various processor modes in ARM? Detail any four modes.						[10]	CO1	L3
4	Draw and detail the complete ARM Register set. Explain Banked Registers in ARM?						[07] [03]	CO1	L3
5	Explain the use of Barrel Shifters in ARM Processor with suitable examples? Write an ALP to find the sum of first 10 interger numbers.						[05] [05]	CO2	L3
6	Explain CMN, CMP, TEQ, TST, SWP instructions with suitable examples.						[10]	CO2	L3
7	Explain the different Logical Instructions in ARM Processor with an example.						[10]	CO2	L3

SOLUTION:

Q-1	Differentiate between RISC and CISC. Explain the Architecture of an ARM Embedded device with the help of a neat diagram			
Sol:	CISC	RISC		
	CISC: stands for Complex Instruction Set Computing	RISC: stands for Reduced Instruction Set Computing		
	Generally used for General purpose Applications like Laptops, Desktops, etc	Generally used for Specific purpose-oriented application like Projectors, Ovens, Remotes, etc		
	A large number of a instructions are present in the architecture.	Very few instructions are present. The number of instructions is generally less than 100.		
	CISC based processors have more complex hardware architectures and relatively simpler compilers	RISC based Processors are supported with complex Compilers to support operations		

		with reduced instruction set architectures (simpler processor hardware).	
	Some instructions with long execution times. These include instructions that copy an entire block from one part of memory to another and others that copy multiple registers to and from memory.	No instruction with a long execution time due to a very simple instruction set. Some early RISC machines did not even have an integer multiply instruction, requiring compilers to implement multiplication as a sequence of additions.	
	Variable-length encodings of the instructions.	Fixed-length encodings of the instructions are used.	
	Multiple formats are supported for specifying operands. A memory operand specifier can have many different combinations of displacement, base, and index register.	Simple addressing formats are supported. Only base and displacement addressing is allowed.	
	CISC supports array.	RISC does not support an array.	
	Arithmetic and logical operations can be applied to both memory and register operands.	Arithmetic and logical operations only use register operands. Memory referencing is only allowed by loading and storing instructions, i.e. reading from memory into a register and writing from a register to memory respectively.	
	Multiple formats are supported for specifying operands. A memory operand specifier can have many different combinations of displacement, base, and index register.	Simple addressing formats are supported. Only base and displacement addressing is allowed.	
	CISC supports array.	RISC does not support an array.	
	The stack is being used for procedure arguments and returns addresses.	Registers are being used for procedure arguments and return addresses. Memory references can be avoided by some procedures.	
	Pipelining implementation becomes complex due to variable length instructions and variable cycle instruction execution	Pipelining implementation is relatively easy due to fixed length Instruction Code and fixed cycle instruction execution	
	Architecture of an ARM Embedded device with the help of a neat diagram:		



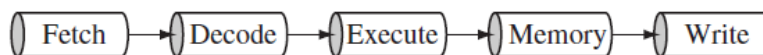
An example of an ARM-based embedded device, a microcontroller.

Above figure shows a typical embedded device based on an ARM core. Each box represents a feature or function. The lines connecting the boxes are the buses carrying data. We can separate the device into four main hardware components:

- The *ARM processor* controls the embedded device. Different versions of the ARM processor are available to suit the desired operating characteristics. An ARM processor comprises a core (the execution engine that processes instructions and manipulates data) plus the surrounding components that interface it with a bus. These components can include memory management and caches.
- *Controllers* coordinate important functional blocks of the system. Two commonly found controllers are interrupt and memory controllers.
- The *peripherals* provide all the input-output capability external to the chip and are responsible for the uniqueness of the embedded device.
- A *bus* is used to communicate between different parts of the device.

Q-2 Explain the 5-stage pipelining used in ARM. Which ARM core implements 5-stage pipelining.

Sol:



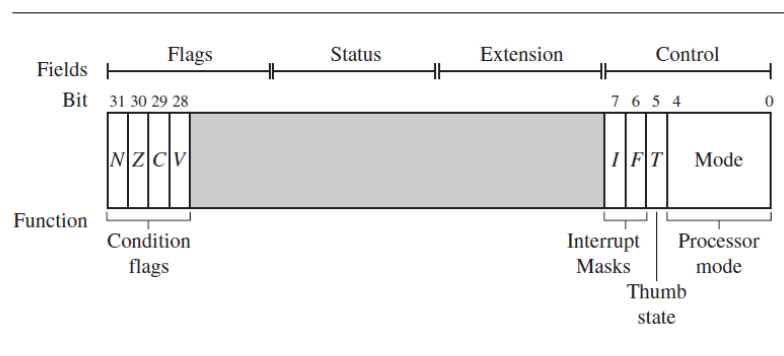
ARM9 five-stage pipeline.

	<div><div>Instruction address</div><div>$pc$$pc-4$$pc-8$$pc-12$$pc-16$</div></div> <div><div>Action</div><div><div>Fetch</div><div>Decode</div><div>ALU</div><div>LS1</div><div>LS2</div></div></div>	
	<div>ARM9TDMI pipeline executing in ARM state.</div> <div>The ARM9TDMI processor performs five operations in parallel:<ul style="list-style-type: none">■ <i>Fetch</i>: Fetch from memory the instruction at address pc. The instruction is loaded into the core and then processes down the core pipeline.■ <i>Decode</i>: Decode the instruction that was fetched in the previous cycle. The processor also reads the input operands from the register bank if they are not available via one of the forwarding paths.■ <i>ALU</i>: Executes the instruction that was decoded in the previous cycle. Note this instruction was originally fetched from address $pc - 8$ (ARM state) or $pc - 4$ (Thumb state). Normally this involves calculating the answer for a data processing operation, or the address for a load, store, or branch operation. Some instructions may spend several cycles in this stage. For example, multiply and register-controlled shift operations take several ALU cycles.■ <i>LS1</i>: Load or store the data specified by a load or store instruction. If the instruction is not a load or store, then this stage has no effect.■ <i>LS2</i>: Extract and zero- or sign-extend the data loaded by a byte or halfword load instruction. If the instruction is not a load of an 8-bit byte or 16-bit halfword item, then this stage has no effect.</div>	
Q-3	What are the various processor modes in ARM? Detail any four modes.	
Sol:	<div>Processor Modes:<p>The processor mode determines which registers are active and the access rights to the cpsr register itself. Each processor mode is either privileged or nonprivileged: A privileged mode allows full read-write access to the cpsr. Conversely, a nonprivileged mode only allows read access to the control field in the cpsr but still allows read-write access to the condition flags.</p><p>There are seven processor modes in total: six privileged modes (abort, fast interrupt request, interrupt request, supervisor, system, and undefined) and one nonprivileged mode (user).</p><div><div>a)</div><div>The processor enters abort mode when there is a failed attempt to access memory.</div></div><div><div>b)</div><div>Fast interrupt request and interrupt request modes correspond to the two interrupt levels available on the ARM processor.</div></div><div><div>c)</div><div>Supervisor mode is the mode that the processor is in after reset and is generally the mode that an operating system kernel operates in.</div></div><div><div>d)</div><div>System mode is a special version of user mode that allows full read-write access to the cpsr.</div></div><div><div>e)</div><div>Undefined mode is used when the processor encounters an instruction that is undefined or not supported by the implementation.</div></div><div><div>f)</div><div>User mode is used for programs and applications.</div></div></div>	

Processor mode.

Mode	Abbreviation	Privileged	Mode[4:0]
<i>Abort</i>	abt	yes	10111
<i>Fast interrupt request</i>	fiq	yes	10001
<i>Interrupt request</i>	irq	yes	10010
<i>Supervisor</i>	svc	yes	10011
<i>System</i>	sys	yes	11111
<i>Undefined</i>	und	yes	11011
<i>User</i>	usr	no	10000

An important feature to note is that the *cpsr* is not copied into the *spsr* when a mode change is forced due to a program writing directly to the *cpsr*. The saving of the *cpsr* only occurs when an exception or interrupt is raised.



A generic program status register (*psr*).

Figure above shows that the current active processor mode occupies the five least significant bits of the *cpsr*. When power is applied to the core, it starts in *supervisor* mode, which is privileged. Starting in a privileged mode is useful since initialization code can use full access to the *cpsr* to set up the stacks for each of the other modes.

Table presented above lists the various modes and the associated binary patterns. The last column of the table gives the bit patterns that represent each of the processor modes in the *cpsr*.

Q-4 Draw and detail the complete ARM Register set.
Explain Banked Registers in ARM?

Sol:

User and
system

<i>r0</i>					
<i>r1</i>					
<i>r2</i>					
<i>r3</i>					
<i>r4</i>					
<i>r5</i>					
<i>r6</i>					
<i>r7</i>					
<i>r8</i>	<i>r8_fiq</i>				
<i>r9</i>	<i>r9_fiq</i>				
<i>r10</i>	<i>r10_fiq</i>				
<i>r11</i>	<i>r11_fiq</i>				
<i>r12</i>	<i>r12_fiq</i>				
<i>r13 sp</i>	<i>r13_fiq</i>	<i>r13_irq</i>	<i>r13_svc</i>	<i>r13_undef</i>	<i>r13_abt</i>
<i>r14 lr</i>	<i>r14_fiq</i>	<i>r14_irq</i>	<i>r14_svc</i>	<i>r14_undef</i>	<i>r14_abt</i>
<i>r15 pc</i>					
<i>cpsr</i>					
-	<i>spsr_fiq</i>	<i>spsr_irq</i>	<i>spsr_svc</i>	<i>spsr_undef</i>	<i>spsr_abt</i>

Complete ARM register set.

Registers

General-purpose registers hold either data or an address. They are identified with the letter *r* prefixed to the register number. For example, register 4 is given the label *r4*.

Figure above shows the active registers available in *user* mode—a protected mode normally used when executing applications.

All the registers shown are 32 bits in size. There are up to 18 active registers: 16 data registers and 2 processor status registers. The data registers are visible to the programmer as *r0* to *r15*.

The ARM processor has three registers assigned to a particular task or special function: *r13*, *r14*, and *r15*. They are frequently given different labels to differentiate them from the other registers.

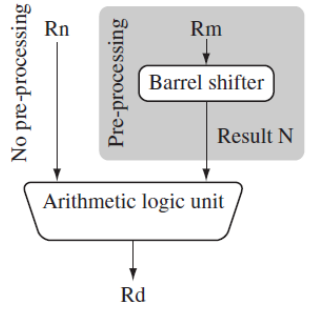
In Figure, they are the shaded registers to identify the assigned special-purpose functions:

- Register *r13* is traditionally used as the stack pointer (*sp*) and stores the head of the stack in the current processor mode.
- Register *r14* is called the link register (*lr*) and is where the core puts the return address whenever it calls a subroutine.
- Register *r15* is the program counter (*pc*) and contains the address of the next instruction to be fetched by the processor.

Depending upon the context, registers *r13* and *r14* can also be used as general-purpose registers, which can be particularly useful since these registers are banked during a processor mode change. However, it is dangerous to use *r13* as a general register when the processor is running any form of operating system because operating systems often assume that *r13* always points to a valid stack frame.

In ARM state the registers *r0* to *r13* are *orthogonal*—any instruction that you can apply to *r0* you can equally well apply to any of the other registers. However, there are instructions that treat *r14* and *r15* in a special way.

In addition to the 16 data registers, there are two program status registers: *cpsr* and *spsr* (the current and saved program status registers, respectively).

	The register file contains all the registers available to a programmer. Which registers are visible to the programmer depend upon the current mode of the processor	
Q-5	Explain the use of Barrel Shifters in ARM Processor with suitable examples? Write an ALP to find the sum of first 10 interger numbers.	
	<p>Barrel Shifter: Consider the following instruction: Syntax: <instruction>{<cond>}{S} Rd, N MOV r7, r5</p> <p>As depicted in the syntax above, N is a simple register.</p> <p>But N can be more than just a register or immediate value; it can also be a register Rm that has been preprocessed by the barrel shifter prior to being used by a data processing instruction.</p>  <p>Barrel shifter and ALU.</p> <p>Data processing instructions are processed within the arithmetic logic unit (ALU).</p> <p>A unique and powerful feature of the ARM processor is the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU. This shift increases the power and flexibility of many data processing operations.</p> <p>Pre-processing or shift occurs within the cycle time of the instruction. This is particularly useful for loading constants into a register and achieving fast multiplies or division by a power of 2.</p> <p>There are data processing instructions that do not use the barrel shift, for example, the MUL (multiply), CLZ (count leading zeros), and QADD (signed saturated 32-bit add) instructions.</p> <p>We apply a logical shift left (LSL) to register Rm before moving it to the destination register. This is the same as applying the standard C language shift operator \ll to the register. The MOV instruction copies the shift operator result N into register Rd. N represents the result of the LSL operation:</p> <p>PRE r5 = 5 r7 = 8</p> <p>MOV r7, r5, LSL #2 ; let $r7 = r5 * 4 = (r5 \ll 2)$</p> <p>POST r5 = 5 r7 = 20</p>	

The five different shift operations that you can use within the barrel shifter are summarized in Table 3.2.

Table 3.2 Barrel shifter operations.

Mnemonic	Description	Shift	Result	Shift amount y
LSL	logical shift left	$x\text{LSL } y$	$x \ll y$	#0–31 or R_s
LSR	logical shift right	$x\text{LSR } y$	(unsigned) $x \gg y$	#1–32 or R_s
ASR	arithmetic right shift	$x\text{ASR } y$	(signed) $x \gg y$	#1–32 or R_s
ROR	rotate right	$x\text{ROR } y$	$((\text{unsigned})x \gg y) (x \ll (32 - y))$	#1–31 or R_s
RRX	rotate right extended	$x\text{RRX}$	$(c \text{ flag} \ll 31) ((\text{unsigned})x \gg 1)$	none

Note: x represents the register being shifted and y represents the shift amount.

Q-6 Explain CMN, CMP, TEQ, TST, SWP instructions with suitable examples.

Sol:

Comparison Instructions:

Syntax: <instruction> {<cond>} {S} Rd, N

N: a register or immediate value

1) CMP : compare

CMP r0, r1; compute (r0 - r1) and set NZCV

Example

PRE: CPSR = nzcqvqiFt_USER, r0 = 4, r9 = 4

CMP r0, r9

POST: CPSR = nZcvqiFt_USER

2) CMN : negated compare

CMN r0, r1; compute (r0 + r1) and set NZCV

Example

PRE: CPSR = nzcqvqiFt_USER, r0 = 4, r9 = 4

CMN r0, r9

POST: CPSR = nzcqvqiFt_USER

3) TST : bit-wise AND test

TST r0, r1; compute (r0 AND r1) and set NZCV

Example

PRE: CPSR = nzcqvqiFt_USER, r0 = 4, r9 = 4

TST r0, r9

POST: CPSR = nzcqvqiFt_USER

4) TEQ : bit-wise exclusive-or test

TEQ r0, r1; compute (r0 EOR r1) and set NZCV

Example

PRE: CPSR = nzcqvqiFt_USER, r0 = 4, r9 = 4

TEQ r0, r9

POST: CPSR = nZcvqiFt_USER

5) SWP: SWAP Instruction

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

$tmp = mem32[Rn]$

$Mem32[Rn] = Rm$

$Rd = tmp$

SWP: swap a word between memory and a register

	<p>SWPB: swap a byte between memory and a register</p> <p>Example PRE: Mem32[0x9000] = 0x12345678 r0 = 0x00000000 r1 = 0x11112222 r2 = 0x00009000</p> <p>SWP r0, r1, [r2]</p> <p>POST: mem32[0x9000] = 0x11112222 r0 = 0x12345678 r1 = 0x11112222 r2 = 0x00009000</p>	
Q-7	Explain the different Logical Instructions in ARM Processor with an example.	
Sol:	<p>Logical Operations Syntax: <instruction> {<cond>} {S} Rd, RN, N N: a register or immediate value</p> <p>1) AND : Bit-wise and Example: PRE: r1 = 0b1111, r2 = 0b0101 AND r0, r1, r2 ; r0 = r1 AND r2 POST: r0=0b0101</p> <p>2) ORR : Bit-wise OR Example: PRE: r1 = 0b1111, r2 = 0b0101 ORR r0, r1, r2 ; r0 = r1 OR r2 POST: r0=0b1111</p> <p>3) EOR : Bit-wise Exclusive-OR Example: PRE: r1 = 0b1111, r2 = 0b0101 EOR r0, r1, r2 ; r0 = r1 Ex-OR r2 POST: r0=0b1010</p> <p>4) BIC : bit clear BIC r0, r1, r2; r0 = r1 & Not(r2) Example: PRE: r1 = 0b1111, r2 = 0b0101 BIC r0, r1, r2 ; r0 = r1 AND (NOT(r2)) POST: r0=0b1010</p>	