CBCS SCHEME

USN | | | | | | | | | | |

BCS402

Fourth Semester B.E./B.Tech. Degree Examination, June/July 2025
Microcontrollers

Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

		Module – 1	M	L	C
Q.1	a.	Explain the major design rules to implement the RISC design philosophy.	08	L2	CO1
	b.	Differentiate between RISC and CISC processors.	04	L2	CO1
	c.	Explain ARM core data flow model, with neat diagram.	08	L2	CO1
		OR			
Q.2	a.	With the help of bit layout diagram, explain Current Program Status Register (CPSR) of ARM.	08	L2	CO1
	b.	With an example, explain the pipeline in ARM.	05	L2	CO1
	c.	Discuss the following with diagrams: (i) Von-Neuman architecture with cache (ii) Harvard architecture with TCM	07	L2	CO1
		Module – 2			
Q.3	a.	Explain the different data processing instructions in ARM.	08	L2	CO2
	b.	Explain the different branch instructions of ARM.	04	L2	CO2
	c.	Explain the following ARM instructions: (i) MOV r_1, r_2 (ii) ADDS r_1, r_2, r_4 (iii) BIC r_3, r_2, r_5 (iv) CMP r_3, r_4 (v) UMLAL r_1, r_2, r_3, r_4	08	L2	CO2
		OR			
Q.4	a.	Explain the different load store instructions in ARM.	08	L2	CO2
	b.	With an example, explain full descending stack operations.	07	L2	CO2
	c.	Develop an ALP to find the sum of first 10 integer numbers.	05	L3	CO2
		Module – 3			
Q.5	a.	List out basic C data types used in ARM. Develop a C program to obtain checksums of a data packet containing 64 words and write the compiler output for the above function.	08	L2	CO3
	b.	Explain the C looping structures in ARM.	08	L2	CO3
	c.	Explain pointer aliasing in ARM.	04	L2	CO2

		OR			
Q.6	**a.**	With an example, explain function calls in ARM.	**08**	**L2**	**CO3**
	b.	Explain register allocation in ARM.	**07**	**L2**	**CO3**
	c.	Write a brief note on portability issues when porting C code to ARM.	**05**	**L2**	**CO3**
		Module – 4			
Q.7	**a.**	Explain the ARM processor exceptions and modes, vector table and exception priorities.	**10**	**L2**	**CO4**
	b.	Explain the interrupts in ARM.	**10**	**L2**	**CO4**
		OR			
Q.8	**a.**	Explain the ARM firmware suite and red hat redboot.	**10**	**L2**	**CO4**
	b.	Explain the sandstone directory layout and sandstone code structure.	**10**	**L2**	**CO4**
		Module – 5			
Q.9	**a.**	Explain the basic architecture of a cache memory and basic operation of a cache controller.	**10**	**L2**	**CO5**
	b.	With a neat diagram, explain a 4 KB, four way set associative cache.	**10**	**L2**	**CO5**
		OR			
Q.10	**a.**	Explain the write buffers and measuring cache efficiency.	**08**	**L2**	**CO5**
	b.	Explain the cache policy.	**12**	**L2**	**CO5**

* * * * *

Fourth Semester B.E Degree Examination
June/July -2025

Time: 3hrs

Marks: 100

MICROCONTROLLERS

Module-1

Q.1.a) Explain the major design rules to implement the RISC design philosophy. — 08M.

Ans:- The RISC philosophy is implemented with four major design rules.

1. Instructions – RISC processors have a reduced no. of instruction classes. These classes provide simple operations that can execute in a single cycle. The compiler or programmer synthesizes complicated operations (for example, a divide operation) by combining several simple instructions. Each instruction is having fixed length to allow the pipline to fetch future instructions before decoding the current instruction.
   * In contrast, in CISC, instructions are often of variable size & take many cycles to execute.
2. Pipelines :— the processing of instructions is broken down into smaller units that can be executed in parallel by pipelines. Ideally the pipeline advances by one step on each cycle for maximum throughput. Instructions can be decoded in one pipeline stage.
   * There is no need for an instruction to be executed

by a mini-program called microcode as on CISC processors.

3. Registers – RISC machines have a large general-purpose register set. Any register can contain either data or an address. Registers act as the fast local memory store for all data processing operations.
   * In contrast CISC processors have dedicated registers for specific purposes.

4. Load-Store architecture :- The processor operates on data held in registers. Separate load & store instructions transfer data betn the register bank & external memory. Memory accesses are costly, so separating memory accesses from data processing provides an advantage because you can use data items held in the register bank multiple times without needing multiple memory accesses.
   * In contrast, CISC processors are more complex & operate at lower clock frequencies.

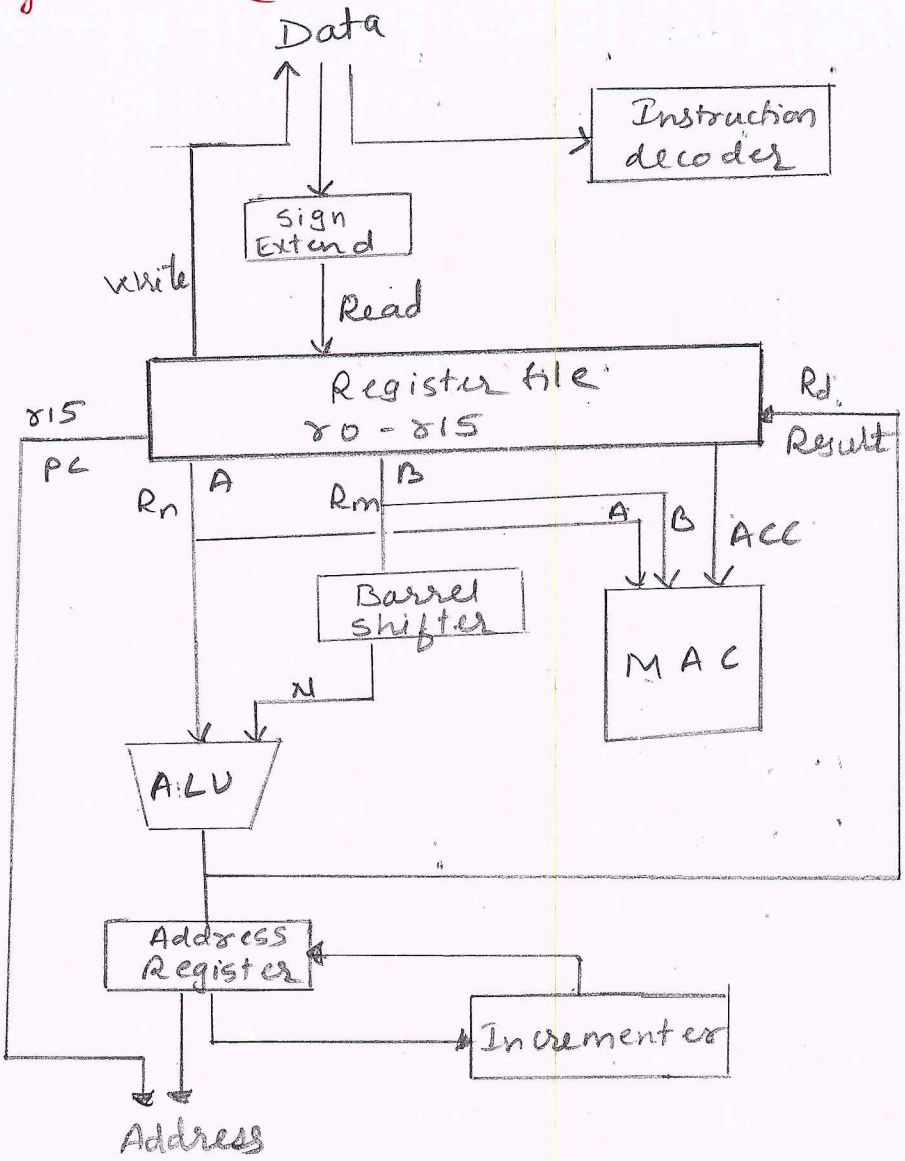
b) Differentiate between RISC & CISC processor.

Ans

RISC	CISC
1) Simple inst^n taking single clk	1) complex inst^n taking multiple clk.
2) Reduced inst^n executed by hlw	2) complex inst^n, executed by processor
3) fixed inst^n & few addressing mode	3) many inst^n & addressing mode.
4) Memory reference is embedded in ~~many~~ ~~inst^n~~ LOAD/STORE	4) memory reference is embedded in ~~LOAD/STORE~~ many inst^n.

c) Explain ARM core data flow model, with neat diagram. (8M) (2)

Ans:



A programmer can think of an ARM core as functional units connected by data buses. The arrows represent the flow of data, the lines represent the buses and the boxes represent either an operation unit or a storage area.

→ Data enters the processor core through the data bus. The data may be an instn to execute or a data item.

→ The instn decoder translates instns before they are executed. Each instn executed belongs to a particular instn set.

→ The ARM processor, like all RISC processors, uses load-store architecture - means it has two instn types for transferring data in & out of the processor

*) Load inst^n copy data from memory to registers. in the core

*) Store inst^n copy data from registers to memory

-> There are no data processing inst^n that directly manipulate data in memory, Thus data processing is carried out in registers.

-> Data items are placed in the register file - a storage bank made up of 32 - bit registers.

-> ARM inst^n have 2 source registers, Rn & Rm & a single result or destination register, Rd.
source operands are read from the register file using the internal buses A & B respectively.

-> The ALU or MAC takes the register values Rn & Rm from A & B buses & computes a result.
Data processing inst^n write the result in Rd directly to the register file.

-> Load & store inst^n uses the ALU to generate an address to be held in the address register & broadcast on the Address bus.

-> After passing through the functional units, the result R_d is written back to the register file using the result bus.

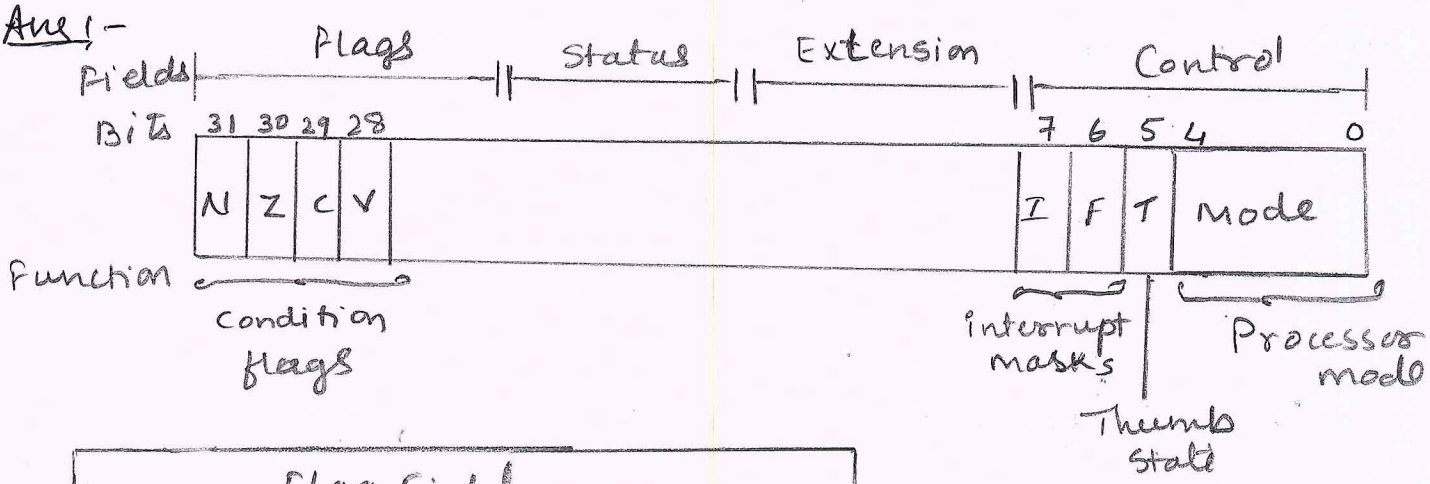
-> For load & store inst^n the incrementer updates the address register before the core reads or write the next register value from or to the next sequential memory location.

-> The processor continues executing inst^n until an exception or interrupt changes the normal execution flow.

OR

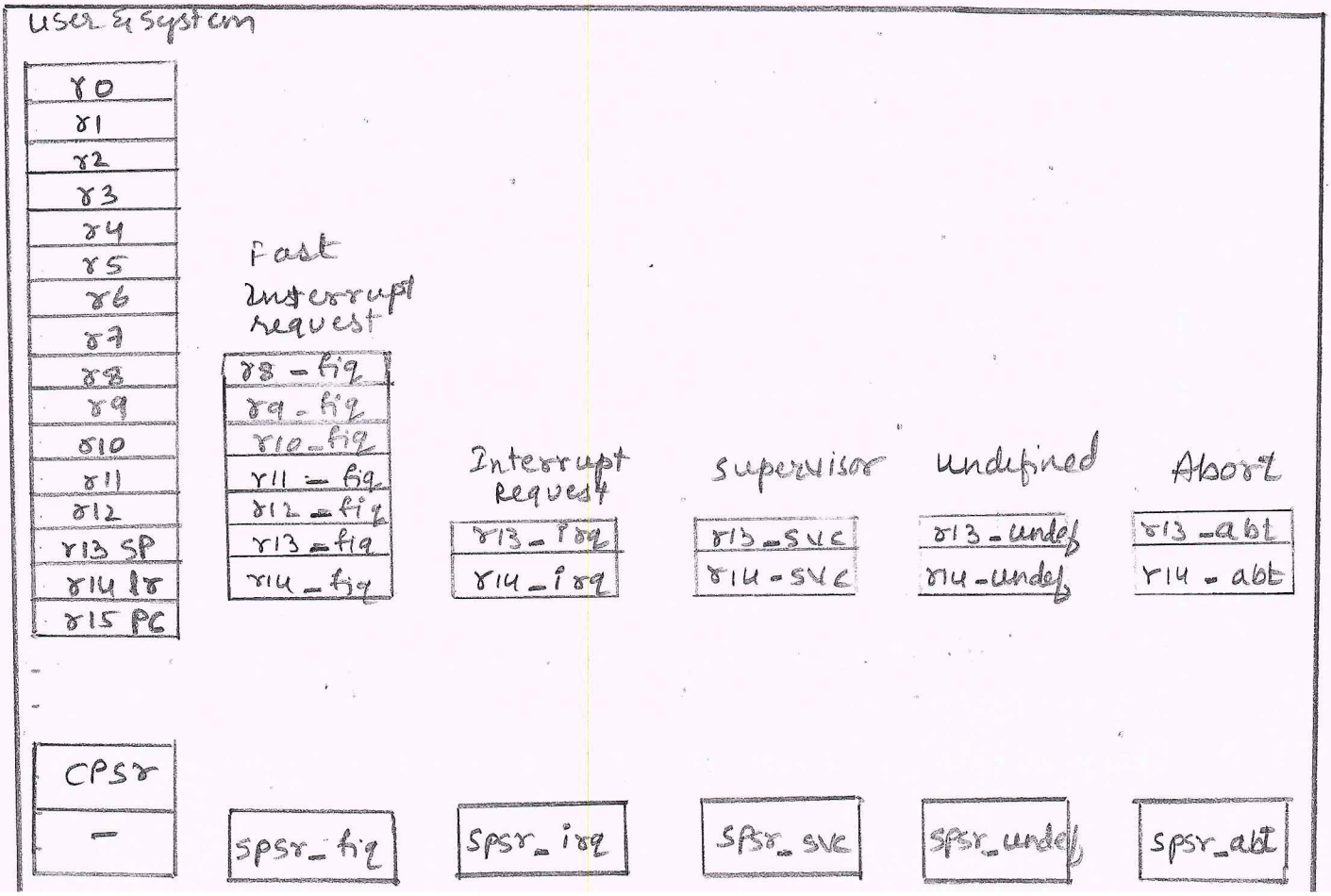
Q.2.a) With the help of bit layout diagm, explain Current Program Status Register (CPSR) of ARm. (08 M)

Ans:-



Flag Field	
N	Negative Result from ALU
Z	Zero result from ALU
C	ALU operation Caused Carry
V	ALU operation Overflowed
Q	ALU operation Saturated
J	Java Byte Code Execution

Control Bits	
I	1: disables IRQ
F	1: disables FIQ
T	1: Thumb, 0: ARm



The ARM core uses the CPSR to monitor & control internal operations. The CPSR is a dedicated 32-bit register & resides in the register file. The CPSR is divided into 4 fields each 8 bits wide: flags, status, extension & control. In current designs the extension & status fields are reserved for future use.

-> The control field contains the processor mode, state & interrupts mask bits

-> The flags field contains the cond^n flags.

Processor Modes:-

-> The processor mode determines which registers are active & the access rights to the CPSR register itself.

-> There are 7 processor modes in total;

*) Six privileged modes (abort, fast interrupt request, interrupt request, supervisor, system & undefined)

*) The processor enters abort mode when there is a failed attempt to access memory.

*) Fast interrupt request & interrupt request modes correspond to the 2 interrupt levels available on the ARM processor.

*) Supervisor mode is the mode that the processor is in the after reset & is generally the mode that an OS kernel operates in

*) System mode is a special version of user mode that allows full read - write access to the CPSR.

*) Undefined mode is used when the processor encounters an inst^n that is undefined or not supported by the implementation.

*) User mode is used for programs & applications

Banked Registers :- There are 37 registers in register file.

-> Of these, 20 registers are hidden from a pgm at different times.

-> They are available only when the processor is in a particular mode.

Ex :- abort mode has banked registers r13_abt, r14_abt & spsr_abt

-> A banked register maps one-to-one onto a user mode register.

-> Banked registers of a particular mode are denoted by an underline character post-fixed to the mode mnemonic or _mode.

b) With an example, explain the pipeline in ARM. (05M)

Ans :- *) A pipeline is the mechanism in a RISC processor, which is used to execute instn

*) Pipelines speeds up execution by fetching the next instn while other instn are being decoded & executed.

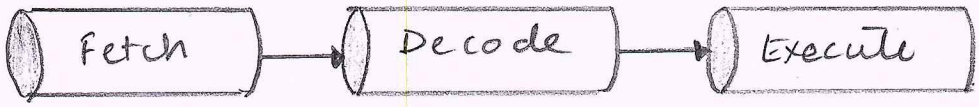


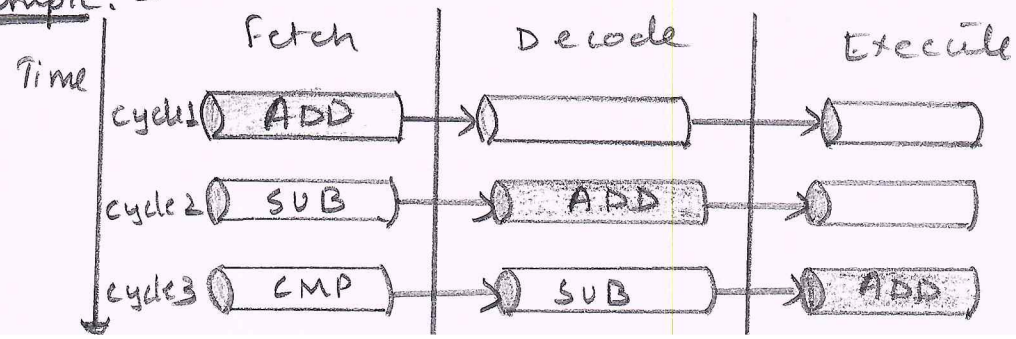
Fig :- ARM7 3-stage pipeline

*) Fetch loads an instn from memory

*) Decode identifies the instn to be executed

*) Execute processes the instn & write the result back to a register.

Example :-



-> The fig shows a sequence of 3 inst^n being fetched, decoded & executed by the processor.

*) The three inst^n are placed into the pipeline sequentially

*) In the 1st cycle, the core fetches the ADD inst^n from memory

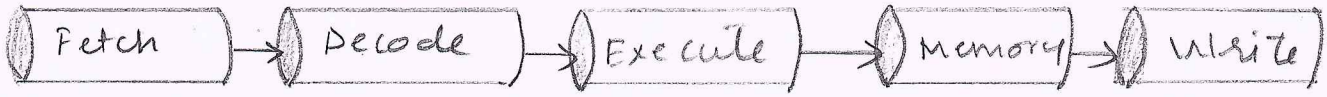
*) In the 2nd cycle, the core fetches the SUB inst^n & decodes the ADD inst^n

*) In the 3rd cycle, both SUB & ADD inst^n are moved along the pipeline. The ADD inst^n is executed, the SUB inst^n is decoded, & the CMP inst^n is fetched.

-> This procedures is called filling the pipeline

-> The pipeline allows the core to execute an inst^n every ~~et~~ cycle.

-> Then ARM9 added a memory & write back stage



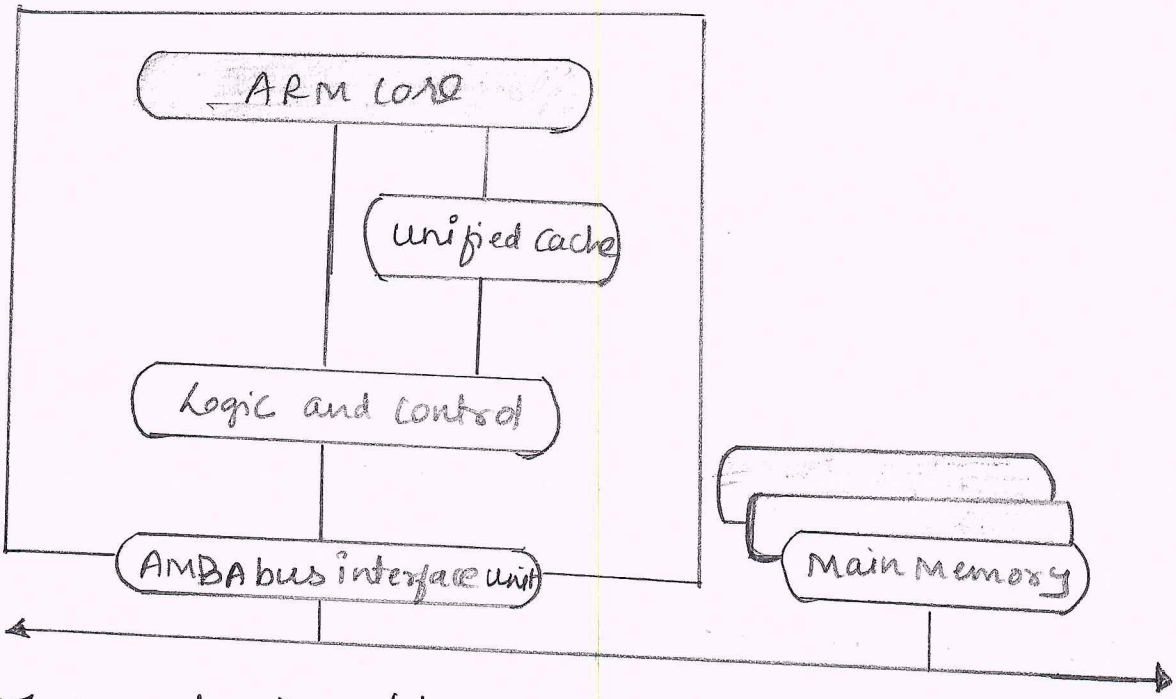
-> Then ~~ARM9~~ ARM10 increases the pipeline length still further by adding a Sixth Stage.

c) Discuss the following with diagrams: (07 M)

i) Von-Neuman architecture with cache

ii) Harvard architecture with TCM.

Ans:- i) Von-Neuman architecture with cache

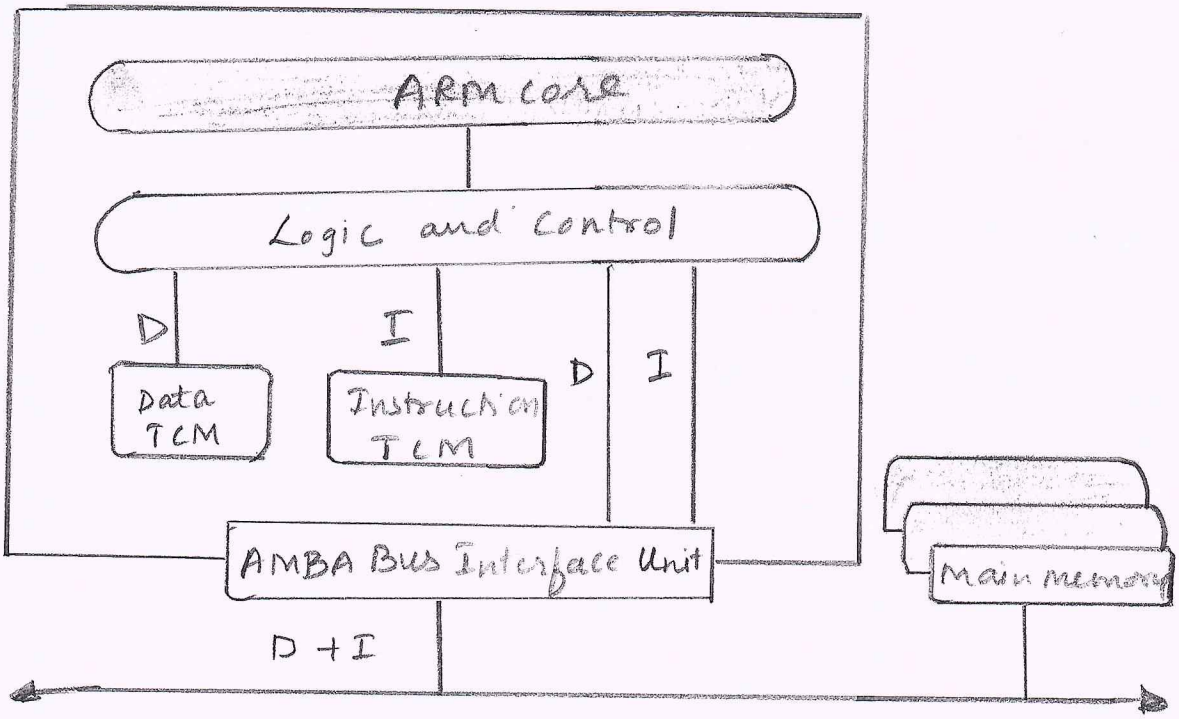


*) The cache is a block of fast memory placed both main memory & the core. It allows for more efficient fetches from some memory types. With a cache the processor core can run for the majority of the time without having to wait for data from slow external memory.

*) Most ARM-based embedded sysm uses a single level cache internal to the processor.

*) ARM has two forms of cache. The first is found attached to the Von Neumann-style cores. It combines both data & instn into a single unified cache as shown in the above fig.

ii) Harvard architecture with TCM's.



*) The second form, attached to the Harvard-style cores, has separate caches for data and instn as shown in the above fig.

*) A cache provides an overall increase in performance but at the expense of predictable execution. But the real-time sym require the code execution to be deterministic - the time taken for loading & storing instn or data must be predictable.

*) This is achieved using a form of memory called tightly coupled memory (TCM). TCM is fast SRAM located close to the core and guarantees the clock cycles reqd to fetch instn or data.

*) TCMs appear as memory in the address map & can be accessed as fast memory.

Module – 2

Q.3. a) Explain the different data processing instruction in ARM. (08M)

Soln:- The data processing instn manipulate data within registers. They are

i) Move instruction
ii) Arithmetic instn
iii) Logical instn
iv) comparison instn
v) Multiply instn.

I] Move instructions:-

Move is the simplest ARM instn. It copies N into a distination register R_d, where N is a register or immediate value. This instn is useful for setting initial values & transferring data betn registers.

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

1) MOV → Move a 32-bit value into a register → $R_d = N$
2) MVN → move the NOT of the 32-bit value into a register → $R_d = \sim N$

Ex:- i) Pre r5 = 5, r7=8
MOV r7, r5
O/P → r5=5, r7=5

ii) Pre, r5=5, r7=8
MVN r7, r5 ; r7 = ~r5
O/P → r5=5, r7=10

II] Arithmetic instructions:

It implements the addition and subtraction of 32-bit signed & unsigned values.

Syntax:- instruction{<cond>}{S} R_d, R_n, N

1)	ADC	→ add 2, 32-bit values & carry	→ $R_d = R_N + N + Carry$
2)	ADD	→ add 2, 32-bit values	→ $R_d = R_n + N$
3)	RSB	→ reverse sub of 2, 32-bit values	→ $R_d = N - R_n$
4)	RSC	→ reverse sub with carry of 2, 32-bits	→ $R_d = N - R_n - !(Carry\ flag)$

5) SBC → Sub with carry of 2, 32-bits → $R_d = R_n - N - !(carry\ flag)$

6) SUB → Sub 2, 32-bits values → $R_d = R_n - N$.

where N → is the result of shift operation.

Ex:- i) Pre $r_0 = 0x00000000$
$r_1 = 0x00000002$
$r_2 = 0x00000001$

SUB r_0, r_1, r_2 ; $r_0 = r_1 - r_2$

∴ o/p $r_0 = 0x00000001$

ii) Pre $r_0 = 0x00000000$
$r_1 = 0x00000011$
$r_2 = 0x00000022$

ADD R_0, R_1, R_2 ; $R_0 = R_1 + R_2$

o/p $R_0 = 0x00000033$.

III] Logical Instructions :- It performs bit wise logical operations on the two source registers.

Syntax:- <instruction>{<cond>}{S} R_d, R_n, N

AND	Logical bitwise AND of 2, 32-bit values	$R_d = R_n \& N$
ORR	Logical bitwise OR of 2, 32-bit values	$R_d = R_n \mid N$
EOR	Logical Exclusive-OR of 2, 32-bit values	$R_d = R_n \wedge N$
BIC	Logical bit clear (AND NOT)	$R_d = R_n \& \sim N$

Ex:- AND R_0, R_1, R_2

Let $R_0 = 0x00000000$
$R_1 = 0x00000011$
$R_2 = 0x00000033$

R_1 = 0000 0000 0000 0000 0000 0000 0001 0001
R_2 = 0000 0000 0000 0000 0000 0000 0011 0011

0000 0000 0000 0000 0000 0000 0001 0001

∴ R_0 = 0x0000 0011

iv) Comparison Instruction :- These are used to compare or test a register with a 32-bit value. They update the CPSR flag bits according to the result, but do not affect other registers. After bits have been set, the information can then be used to change program flow by using conditional execution. Here not need to apply the 'S' suffix for comparision instn to update the flag. S is nothing but it updates the flags in CPSR

Syntax :- <instructions>{<cond>} R_n, N.

CMN	Compare Negate	flags set as Result of R_n + N
CMP	compare	flags set as Result of R_n − N
TEQ	test for equality of 2, 32-bits	flags set as result of R_n ^ N
TST	test bits of a 32-bits	flags set as reseelt of R_n & N

Ex :- TEQ R_0, #0x80008000 ; sets Z=1, if R_0 contains 0x80008000

Let R_0 = 0x00000033 ; R_0 is X-OR with immediate data of 0x80008000

R_0 = 0000 0000 0000 0000 0000 0000 0011 0011
Data = 1000 0000 0000 0000 1000 0000 0000 0000

1000 0000 0000 0000 1000 0000 0011 0011
8 0 0 0 8 0 3 3

∴ Since result is not zero, Z=0, N=1.

R_0 = 0x80008033.

I] Multiply Instruction :- This instn multiplies the content of a pair of registers, accumulate the result in with another register.

Syntax :- MLA{<conds>}{S} R_d, R_m, R_s R_n.

MUL{<conds>}{S} R_d, R_m, R_s.

i) MLA → multiply & accumulate → $R_d = (R_m * R_s) + R_n$

ii) MUL → Multiply. → $R_d = R_m * R_s$

Ex :- i) MUL R_2, R_1, R_0 ; $R_2 = R_1 \times R_0$

Let R_0 = 0x00000002

R_1 = 0x00000004

R_2 = 0x00000000

$\therefore R_2 = R_1 \times R_0$

$= (0x00000002) \times (0x00000004)$

R_2 = 0x00000008 //

ii) MLA R_3, R_2, R_1, R_0 ; $R_3 = (R_1 \times R_2) + R_0$

Let R_0 = 0x00000002

R_1 = 0x00000004

R_2 = 0x00000002

R_3 = 0x00000000

$\therefore R_3 = [(0x00000004) \times (0x00000002)] + (0x00000002)$

R_3 = 0x00000000A

i.e (4x2) + 2

8 + 2 = 10

b) Explain the different branch instruction of ARM. (04M)

Soln :- A branch inst^n changes the flow of execution or is used to call a routine. This type of inst^n allows programs to have subroutines, if-then-else and loops.

The ARM use instruction set includes four different branch instruction.

1) Branch (B)
2) Branch with link (BL)
3) Branch Exchange (BX)
4) Branch Exchange with link (BLX)

Syntax :-

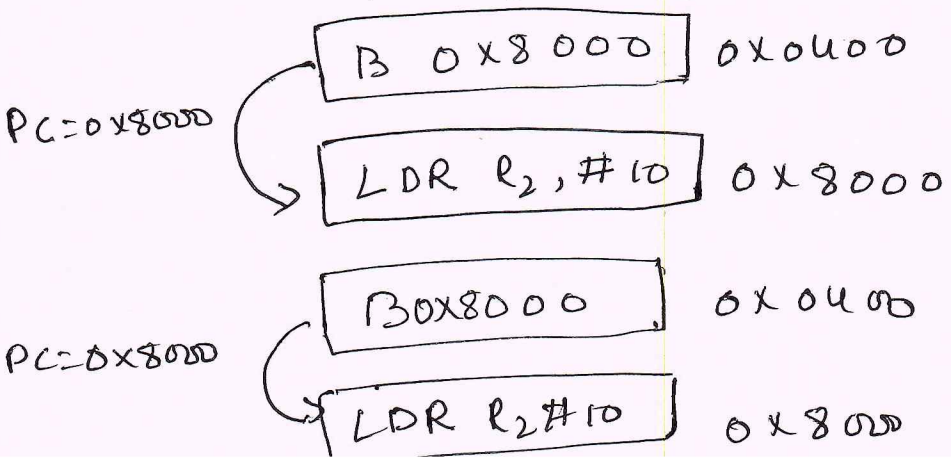
B {<Conds>} label

BL {< Conds>} label

BX {< Conds>} R_m

BLX {< Conds>} label R_m

i) Branch (B) :- The basic branch inst^n allows a jump forward or backwards of upto 32 MB. A modified version of the branch inst^n, the branch link allows the same jump but stores the current PC address plus four bytes in the link register (LR)



$\therefore R_{14} = 0x04004$

The branch instruction will jump you to a destination address. The Branch link inst^{n} jumps to the destination & stores a return address in R_{14}

The branch inst^{n} have 2 other variants called "branch exchange" & "branch exchange with link". These 2 inst^{n} perform the same branch operation but also swap inst^{n} operation from ARM to THUMB & vive-versa

2) Branch with Link (BL):- The Branch with link (BL) inst^{n} is similar to the Branch (B) inst^{n}, but overwrites the link register (LR) with a return address. It perform a subroutine call.

c) Explain the following ARM instruction (08 M)

i) MOV r_1, r_2 ii) ADDS r_1, r_2, r_4

iii) BIC r_3, r_2, r_5 iv) CMP r_3, r_4, v) UMLAL $r_1, r_2, r_3 r_4$

Soln:- i) MOV r_1, r_2 ; $r_1 \leftarrow r_2$

This inst^{n} will move (copy) the content of operand r_2 into operand r_1 without changing the content of operand r_2.

Ex:- Let $R_1 = 0x00000000$

$R_2 = 0x00000011$

After execution of this inst^{n}

$R_1 = 0x00000011$

$R_2 = 0x00000011$

ii) ADDS r_1, r_2, r_4.

This instn adds the value of stored in 2 registers (R_2 & R_4) & stores the result in a third register (R_1) & updates the processor's condition flag based on the result.

→ ADD is the addition

→ S is the suffix that tells the processor to update the cond^n flags (N, Z, C, V) in the CPSR based on the result.

→ Logic: $R_1 = R_2 + R_4$ (and update flags)

Ex:- Let $r_2 = 0x00000005$ (5 in decimal)
$r_4 = 0x0000000A$ (10 in decimal)

ADDS r1, r2, r4

$r_2 = 0x00000005$
$r_4 = 0x0000000A$
$r_1 = 0x0000000F$

5+10 = 15 (F)

→ Because of the 'S' suffix, the processor updates the flags

1) Z (Zero): cleared (result is not zero)
2) N (Negative): cleared (result is positive)
3) C (carry): cleared (no signed overflow)
4) V (overflow): cleared (no signed overflow.

iii) BIC r_3, r_2, r_5

Soln :- BIC is a way to clear bits within a word, a sort of reverse or operand two is a 32 bit mask. If a bit is set in the mask, it will be cleared. unset mask bits, indicate bits to be left. This is useful when clearing status bit.

This will perform logical AND operation betn operand 1 and complemented value of operand 2.

Ex :- BIC r_3, r_2, r_5

Let $r_3 = 0x00\ 00\ 00\ 00$

$r_2 = 0x0000000F$

$r_5 = 0x00000005$

Now R_2 & $\sim R_5$

$r_2 =$ 0000 0000 0000 0000 0000 0000 0000 1111

$\sim r_5 =$ 1111 1111 1111 1111 1111 1111 1111 1010

0000 0000 0000 0000 0000 0000 0000 1010

0 0 0 0 0 0 0 A

$\therefore r_3 = 0x0000000A$

iv) CMP r_3, r_4

This compare instn performs subtraction, but does not store the result. The flags are always updated. CMP always allow you to compare the contents of a register with another register or an immediate value, updating the status flags to allow conditional execution to take place.

CMP r_3, r_4 ; Z=1 if $R_0 = R_1$, N=0 if $R_0 > R_1$

v) UMLAL r_1, r_2, r_3, r_4

Soln :- UMLAL -> unsigned multiply accumulate long

UMLAL r_1, r_2, r_3, r_4 ", $(r_3 * 4) + [r_2, r_1)$.

Let $r_3 = 0x00000002$

$r_4 = 0x00000005$

$r_1 = 0x00000003$

$r_2 = 0x00000000$

Execution:

1) multiples $r_3 \times r_4 = 0x00000002 \times 0x00000005$
$= 0x0000000A$

2) Accumulate : Add initial $[r_2, r_1)$ which is 0+3 to the product (10)

3) Result : 10 + 3 = 13 (0x0000000D

$\therefore r_1 = 0x0000000D$

$r_2 = 0x00000000$

Q4. a) Explain the different load store Instⁿ in ARM (8M)

Soln :- *) Load - Store instⁿ transfer data betⁿ memory & processor registers. There are 3 types of Load - store instⁿ

1) Single - register transfer
2) Multiple - register transfer
3) SWAP.

i) Single - register transfer

*) These instⁿ are used for moving a single data item in & out of a register.

*) Here various Load - Store instⁿ are.

Syntax:-

<LDR|STR>{<cond>}{B} Rd , addressing

LDR{<Cond>} SB|H|SH Rd , addressing

STR{<Cond>} H Rd , addressing

LDR — Load word into a register

STR — save byte or word from a register

LDRB — Load byte into a register

STRB — Save byte from a register

LDRH — Load halfword into a register

STRH — Save halfword into a register

LDRSB — Load signed byte into a register

LDRSH — Load signed halfword into a register

Ex:-

1) LDR r0, [r1]

This instn loads a word from the address stored in register r1 & places it into register r0.

2) STR r0, [r1]

This instn goes the other way by storing the contents of register r0 to the address contained in register r1.

ii) Multiple - Register Transfer

x) Load - Store instn can transfer multiple regis -ers betn memory & the processor in a single instn The transfer occurs from a base address register Rn pointing into memory.

*) Multiple - register transfer instn are more efficient from single - register transfers for moving blocks of data around memory & saving & restoring context & stacks.

Syntax! -

<LDM|STM>{<cond>}<addressing mode> Rn{!}, <reg>{^}

LDM - Load multiple registers

STM - Save multiple registers.

Addressing mode for Load - Store multiple instn

IA - increment after

IB - increment before

DA - Decrement after

DB - Decrement before

Ex! - LDMIA r0!, {r1-r3}

Let r0 = 0x00080010

r1 = 0x00000000

r2 = 0x00000000

r3 = 0x00000000

After execution

r0 = 0x0008001C

r1 = 0x00000001

r2 = 0x00000002

r3 = 0x00000003

iii) SWAP Instruction

*) This will swap the content of memory with the contents of a register

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

SWP – swap a word betn memory & register.
SWPB – swap a byte betn memory & register.

Ex :- SWP r0, r1, [r2]

Let r0 = 0x00000000 Let memory = 0x12345678
r1 = 0x 11 11 2222
r2 = 0x 00009000

After execution r0 = 0x12345678
r1 = 0x1111 2222
r2 = 0x00009000
memory = 0x11112222

b) With an example, explain full descending stack operations. (07m)

Soln :- *) The ARM architecture uses the load-store multiple instn to carry out stack operations.
*) The pop operation (removing data from a stack) uses a load multiple instn. similarly push operation (placing data onto the stack) uses a store multiple instn.
*) When you use a full stack (F), the stack pointer sp points to an address that is the last used or full location.
*) In contrast, if you use an empty stack (F) the sp points to an address that is the first unused or empty location.
*) A stack is either ascending (A) or descending (D) Ascending stacks grow towards higher memory addresses; in contrast, descending stacks grow towards lower memory address.

*) Addressing modes for stack operation.

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

*) The LDMFD & STMFD Instn provide the pop and push functions respectively.

Ex:- With full descending.

STMFD SP! , {r1, r4}

Let r1 = 0x00000002
r4 = 0x00000003
SP = 0x00080014

After execution
r1 = 0x00000002
r4 = 0x00000003
SP = 0x0008000C

Pre:-

	Address	data
	0x80018	0x00000001
	0x80014	0x00000002
P	0x80010	Empty
	0x8000C	empty

After:-

	Address	data
	0x80018	0x00000001
	0x80014	0x00000002
SP→	0x80010	0x00000003
	0x8000C	0x00000002

c) Develop an ALP to find the sum of first 10 integer numbers. (5m)

Soln:-

Label Field	Mnemonic Field	Comments Field
AREA INTSUM, CODE, READONLY		
	ENTRY	; Mark 1st instn to execute
	MOV R1, #10	; Load 10 to register
	MOV R2, #0	; Empty R2 register to store result
LOOP	ADD R2, R2, R1	; ADD the content of R1 with the result at R2
	MUL R3, R1, R2	; Multiplication
	SUBS R1, #0X01	; Decrement R1 By 1
	BNE LOOP	; Repeat Till R1 goes to zero
HERE	B Here	
	END	

Module-3

Q5. a) List out basic C data types used in ARM. Develop a C prgm to obtain checksum of a data packet containing 64 words & write the compiler o/p for the above function. (8M)

Soln:- C data types used in ARM

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.

C-program to obtain checksum of data packed containing 64 words.

```
int checksum_V1 (int *data)
{
 char i;
 int sum=0;
 for(i=0;i<64;i++)
 {
  sum+=data[i];
 }
 return sum;
}
```

The compiler o/p for this function is:

```
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.
```

b) Explain the C-looping structures in ARM. (8M)

Soln:- The most efficient ways to code for and while loops on the ARM. There are 2 types.

i) Loops with a fixed no. of iterations
ii) Loops with a variable no. of iterations.
iii) Loop unrolling

i) Loops with a fixed No. of iterations :- It is most efficient way to write a for loop on the ARM. Let us return to our checksum example & look at the looping structure.

The compiler treats a loop with incrementing i.e count i++

```
int check sum_V5 (int *data)
{
  unsigned int i;
  int sum = 0;
  for (i=0; i<64; i++)
  {
   sum += *(data ++);
  }
   return sum;
  }
```

This compiles to

```
check sum_V5
        MOV r2, r0          ; r2 = data
        MOV r0, #0          ; sum = 0
        MOV r1, # 0         ; i = 0
check sum_V5_loop
        LDR r3, [r2], #4    ; r3 = *(ddta ++)
        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 3 instn 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
*) A sub to decrement the loop counter, which also sets the cond^n code flags on the result.
*) A conditional branch instn.

ii) Loops using a variable No. of iterations:

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

Ex:- We pass in a variable N giving the variable no. of iterations as an argument & 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 o/p is.

```
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
```

iii) Loop unrolling :- While implementing the loop, there are some additional inst^n in add^n to body of the loop :: A sub to decrement the loop count and a conditional branch. These inst^n are called loop overhead. We can save some of these cycles by * unrolling a loop, means, repeating the loop body several times, & reducing the no·of loop iterations by the same proportion.

c) Explain pointer aliasing in ARM. (04M)

Sol^n :- 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 fun^n, the compiler often does not know which pointer cause aliasing & which pointer not.

Ex :- Function Increments two timer values by a step amount;

```
Void timer_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 = to
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;
}
```

Q6. a) With an example, explain function calls in ARM. (8M)

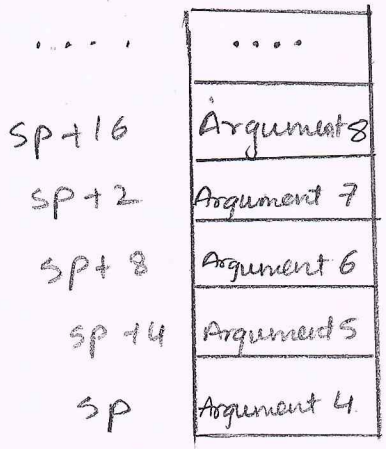
Soln:- The ARM procedure Call Standard (APCS) defines how to pass functions arguments and return values in ARM registers.

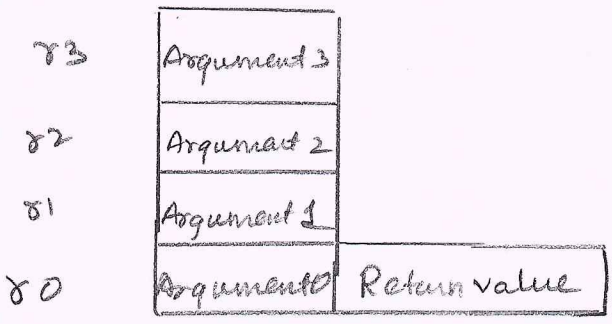
Four Register Rule:

The 1st four integer arguments are passed in the 1st four ARM registers: r0, r1, r2 & r3. Subsequent integer arguments are placed on the full descending stack. Function return integer value are passed in r0.

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

ATPCS argument Passing





If our c fun^n needs more than 4 arguments, it is always more efficient, if we group related

arguments into structures & pass on struct-ure point rather than using multiple arguments.

Ex:- Insert N bytes (from Array data into a que

Case 1: 5 Arguments

```
Char * queue_bytes_V1(
 Char * Q_start,
 Char * Q_end,
 Char * Q_ptr,
 Char * data,
 Unsigned int N)
 {
  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];
    r12,[r13,#4]
queue_V1_loop
    LDRB r14,[r3],#1
    STRB r14,[r2],#1
    CMP r2,r1
```

```
MOVEQ r2,r0
SUBS r12, r12,#1
BNE queue_VI_loop
MOV r0,r2
LDR PC, [r13],#4
```

b) Explain register allocation in ARM. (07M)

Soln :- 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 variables ~~do not~~ than available registers, the compilers stores the excess variables on the processor stack. These variables are called spilled or swapped out variables. Spilled variables are slow to access compared to variables allocated to registers.

To implement a function efficiently, we need to

*) minimize the no. of spilled variables
*) ensure that most important & frequently accessed variables are stored in registers.

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

C-compiler register usage

Register Number	Alternative Register Names	ATPCS register Usage
r0	a1	Argument registers. These hold the first four fun^n arguments on a fun^n call and the return value on a function return
r1	a2	
r2	a3	
r3	a4	
r4	V1	General variable registers. The fun^n must preserve the callee values of these registers.
r5	V2	
r6	V3	
r7	V4	
r8	V5	
r9	V_6 SB	The r9 holds the static base addr
r10	V_7 SL	The R10 holds the stack limit addr
r11	V_8 FP	The fun^n must preserve the callee value of this register except when compiling using a frame pointer
r12	ip	A general scratch register that the fun^n can corrupt, it is useful as a scratch register for fun^n veneers or other intra procedure call requirements
r13	sp	The stack pointer, pointing to the full descending stack
r14	lr	The link register. On a fun^n call this holds the return addr
r15	PC	The program Counter

c) Write a brief note on portability issues when porting C code to ARM. (05M)

Soln :- Here is a summary of the issues you may encounter when porting C-code to the ARM.

i) The 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 cond^n 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.

ii) The int type :-

Some older architectures use a 16-bit int, which may cause problems when moving to ARM's 32-bit int type although this is ~~not~~ rare now a days. Note that expressions are promoted to an int type before evaluation

iii) Unaligned data pointers :

Some processors support the loading of short & int typed values from unaligned addresses. A C-program may manipulate pointers directly. So that they become unaligned

iv) Endian assumptions :-

C-code may make assumptions about the endianness of a memory sym.

Ex :- 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 & replace them by endian-independent ones.

v) 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 func^n 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.

vii) Use of bit-fields :-

The layout of bits within a bit-field is implementation & endian dependent. If C-code assumes that bits are laid out in a certain order, then the code is not portable.

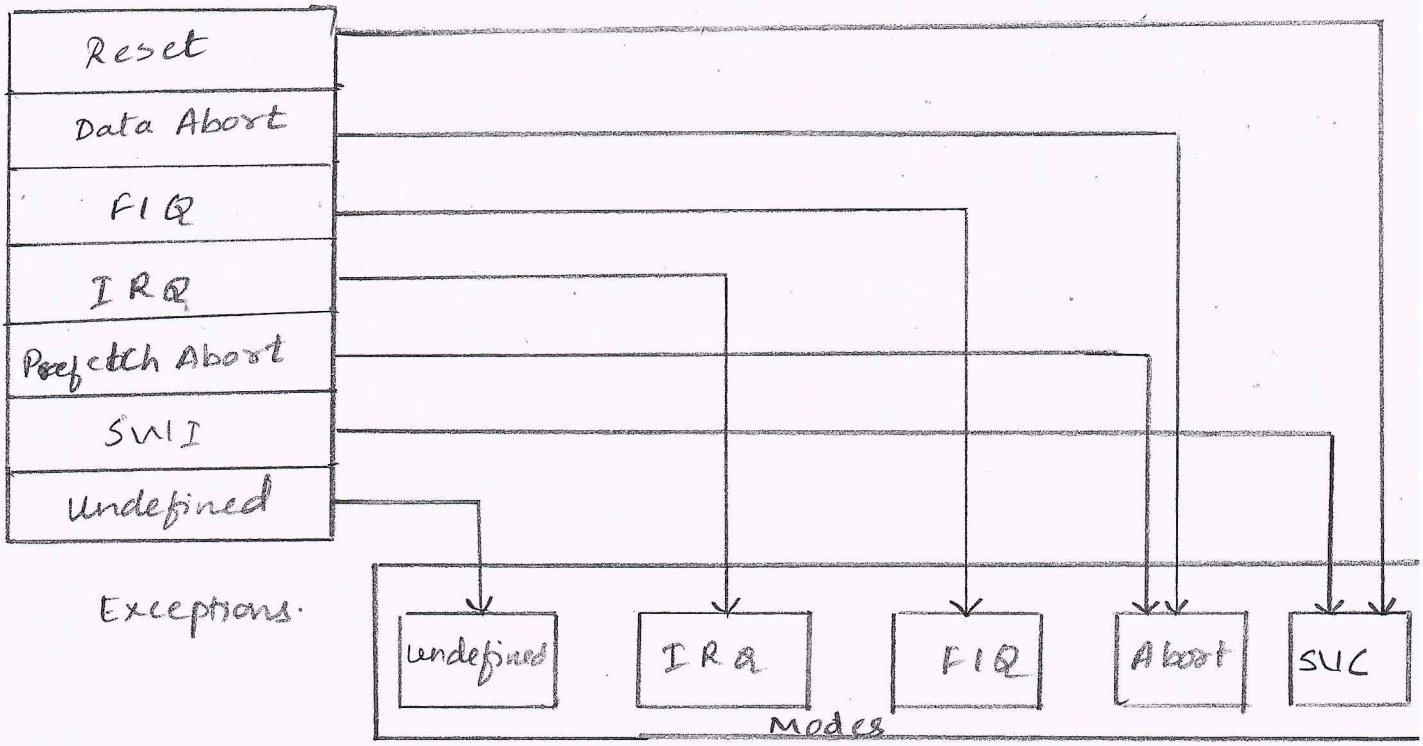
Module-4

Q7. a) Explain the ARM processor exceptions & modes, vector table and exception priorities. (10M)

Soln:- An exception is any condition that needs to halt the normal sequential execution of instn.

ARM Processor Exception & Modes:-

When an exception occures, the core enter a specific mode. The ARM processor modes can be entered manually by changing the cpsr. When an exception occurs the ARM processor always switches to ARM state.



The user and system mode are the only 2 modes that are not entered by an exception.

When an exception causes a mode change, the core automatically

a) Saves the cpsr to the spsr of the exception mode
b) saves the pc to the lr of the exception mode.

*) sets the cpsr to the exception mode
*) sets pc to the address of the exception handler.

Vector table :- The vector table is a table of addr that ARM core branches to when an exception is raised. These addr contain branch inst^n. The memory map addr 0x00000000 is reserved for the vector table, a set of 32-bit words. On some processors the vector table can be optionally located at a higher addr^n in memory (starting at the offset of 0xffff0000).

Exception	Mode	Vector table offset
Reset	SVC	+0x00
Undefined Inst^n	UND	+0x04
Software Interrupt (SWI)	SVC	+0x08
Prefetch abort	ABT	+0x0C
Data Abort	ABT	+0x10
Not assigned	—	+0x14
IRQ	IRQ	+0x18
FIQ	FIQ	+0x1C.

The branch inst^n are:

1) B <address>
2) LDR pc,[pc, #offset]
3) LDR pc,[pc, #0xff0]
5) Mov pc, #immediate.

Exception priorities :-

Exceptions can occur simultaneously, so the processor has to adopt a priority mechanism. Each exception is dealt with according to the priority level set out.

Exceptions	Priority	I bit	F bit
Reset	1	1	1
Data Abort	2	1	—
Fast Interrupt Request	3	1	1
Interrupt Request	4	1	—
Prefetch Abort	5	1	—
Software Interrupt	6	1	—
Undefined Instn	6	1	—

1) Reset :- exception is the highest priority & it occurs when power is applied to the processor.

2) Data abort occur when the memory controller or MMU indicates that an invalid memory addr has been accessed or when current code attempts to read or write to memory without the correct access permission

3) Fast Interrupt Request :- occurs when an external peripheral sets the FIQ pin to nFIQ.

4) Interrupt Request occurs when external periph-eral sets the IRQ pin to nIRQ.

5) Prefetch Abort exception :- occur when an attempt to fetch an instn results in a memory fault.

6) Software Interrupt (SWI) occurs when the SWI instn is executed & none of the other higher-priority exceptions have been flagged

7) Undefined Instn :- occurs when an instn not in the ARM or Thumb instn set reaches the execute stage of the pipeline & non of the other exceptions have been flagged.

b) Explain the interrupts in ARM. (10M)

Soln :- Interrupts :- There are 2 types of interrupts available on the ARM processor. The first type of interrupt causes an exception raised by an external peripheral - namely IRQ & FIQ. The 2nd type is a specific instn that causes an exception - the SWI instn. Both of types suspend the normal flow of a pgm.

- -> Assigning Interrupts
- -> Interrupt Latency
- -> IRQ & FIQ exceptions
- -> Basic Interrupt stack design & implementn.

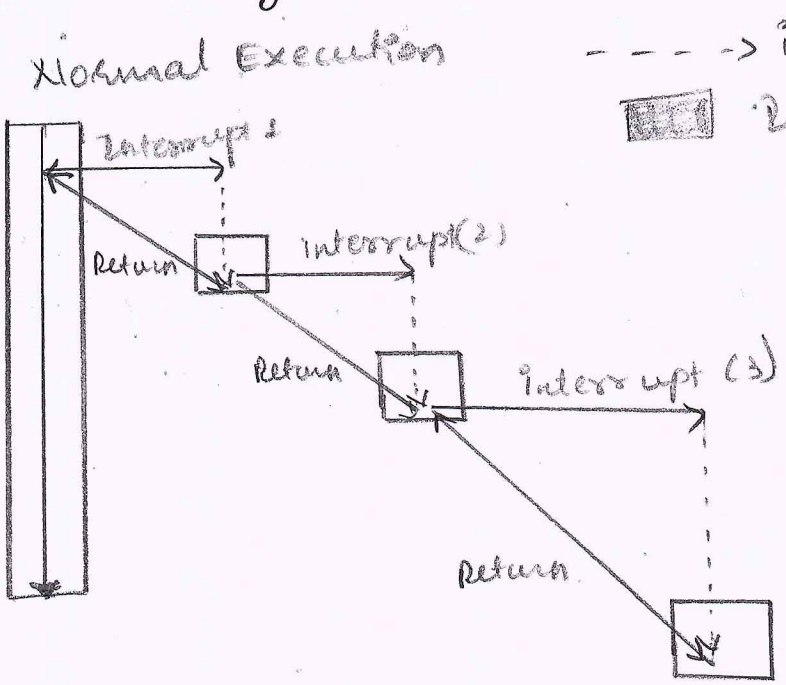
Assigning Interrupts :-

A system designer can decide which hardware peripheral can produce which interrupt request. This decision can be implemented in HW & SW and depends upon the embedded sysm being used.

An interrupt controller connects multiple external interrupts to one of the 2 ARM

interrupt requests. Sophisticated controllers can be programmed to allow an external interrupt source to cause either an IRQ or FIQ exception

Interrupt Latency :- Interrupt latency depends on a combination of h/w & s/w. sym architects must balance the sym design to handle multiple simultaneous interrupt sources & minimize interrupt latency



IRQ & FIQ Exceptions:-

They only occur when a specific interrupt mask is cleared in the CPSR. The ARM processor will continue executing the current instn in the execution stage of the pipeline before handling the interrupt.

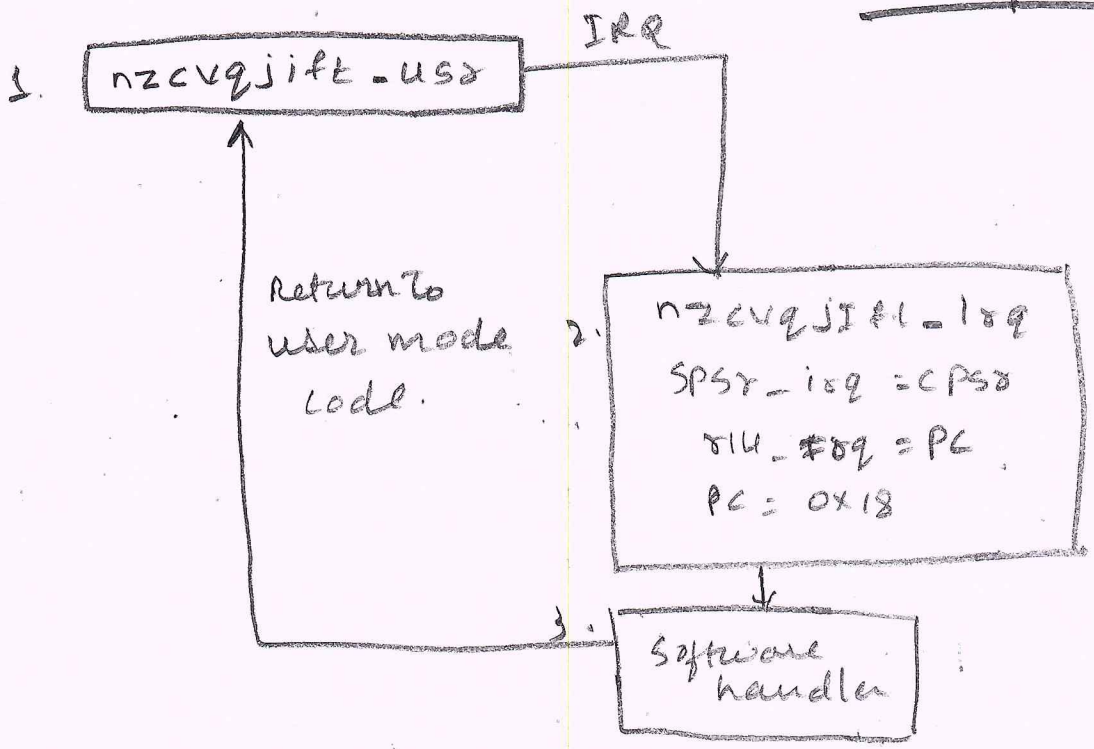
It goes through following procedure

1) The processor changes to a specific interrupt request mode, which being raised.

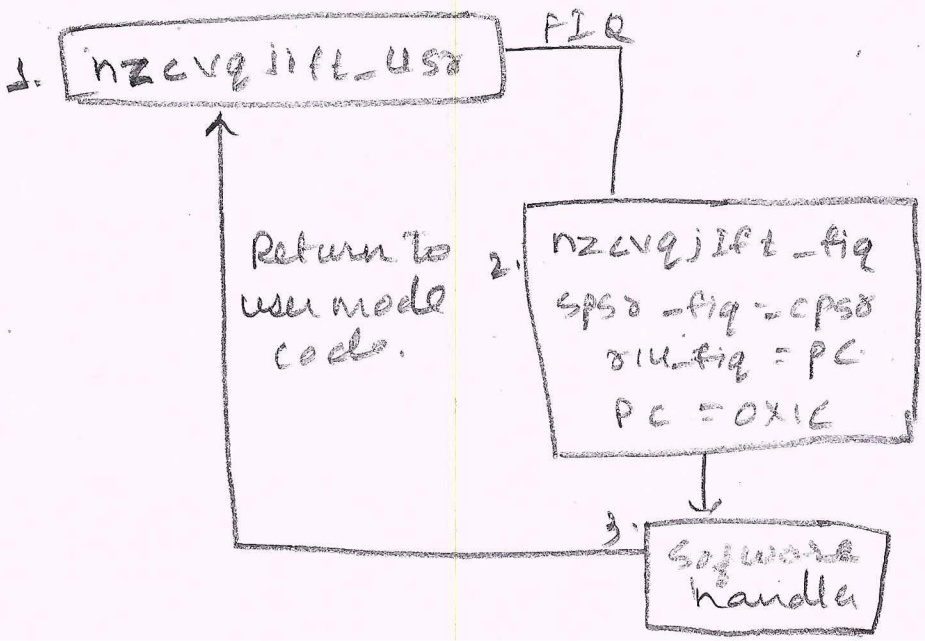
2) The previous modes cpsr is saved into the spsr of the new interrupt request mode.
3) The pc is saved in the lr of the new interrupt request mode.
4) Interrupts are disabled either IRQ or both exceptions are disabled in the CPSR.
5) The processor branches to a specific entry in the vector table.

Interrupt Request (IRQ)

Ex:-



Fast Interrupt Request (FIQ)



① 8. a) Explain the ARM firmware suite and redhat redboot : (10M)

Soln :- ARM has developed a firmware package called the ARM Firmware Suite (AFS). AFS is designed purely for ARM-based embedded sysm. It provides support for a no. of boards & processors including the Intel KScale & Strong ARM Processors. The package includes two 2 major pieces of technology a Hardware Abstraction layer called µHAL and a debug monitor called Angel.

µHAL provides a low level device driver framework that allows it to operate over different commn devices (for example, USB, Ethernet, serial). It also provides a std API. Consequently when a port takes place, the various h/w specific parts must be implemented in accordance with the various µHAL API funn.

µHAL supports these main features:

* System initialization :- setting up the target platform and processor core. Depending upon the complexity of the platform, this can either be a simple or complicated task.
* Polled serial driver :- used to provide a basic method of commn with a host.
* LED support :- allows control over the LED's for simple user feedback. This provides an application the ability to display operational status.

*) Timer Support :- allows a periodic interrupt to be set up. This is essential for pre emptive context switching operating sym that require this mechanism.

*) Interrupt controllers :- Support for different interrupt controllers.

The second technology, Angel, allows comm^n between a host debuggee & a target platform. It allows you to inspect & modify memory, download and execute images, set breakpoints, & display processor register contents. All this control is through the host debuggee. The angel debug monitor must have access to the SWI & IRQ or FIQ Vectors

Angel uses SWI inst^n to provide a set of API's that allows a program to open, read & write to a host filing system. IRQ/FIQ interrupts are used for comm^n purpose with the host debuggee

Red HAT RedBOOT :-

Red Boot is a firmware tool developed by Red Hat. It is provided under an open source license with no royalties or up front fees. Redboot is designed to execute on different CPU's for instances, ARM, etc. It provides both debug capability through GNU debugger as well as boot loader. The Red Boot S/W core is based on a HAL Redboot supports these main features.

a) Communication — Configuration is over serial or Ethernet. For serial, X-modem protocol is used for comm^n with the GDB. For ethernet, TCP is used.

*) Flash Rom memory Management :- provides a set of filing sym routines that can download, update and erase image in flash ROM. also images can be either compressed or uncompressed.

*) Full operating sym support :- supports the loading and booting of Embedded linux, Red Hat ecos & many other popular operating sym.

b) Explain the sandstone directory layout & sand stone code structure. (10M)

Soln :- Sand stone Directory Layout :-

Sandstone can be found on ~~our~~ the website. If you take a look at sandstone, you will see that the directory structure as in fig.

Tabel 1 :-

Feature	configuration
code	ARM instn only
Toolchain	ARM Developer Suite 1.2
Image size	700 bytes
Source	17KB
memory	remapped

Fig : Summary of Sandstone

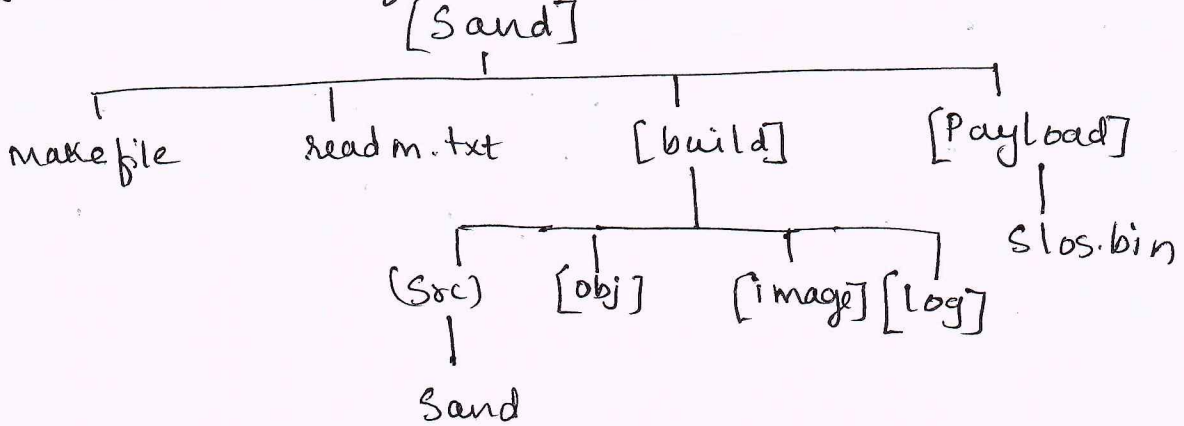


Fig : Sandstone directory layout.

The Sandstone source file Sand.s is located under the sand/build/src directory.

The object file produced by the assembler is placed under the build/obj directory. The object file is then linked & the final Sandstone image is placed under the sand/build/image directory.

This image includes both the Sandstone code & the payload. The payload image, the image that is loaded & booted by Sandstone is found under the sand/payload directory.

Sandstone Code Structure:-

Sandstone consists of a single assembly file. The file structure is broken down into a no. of steps, where each step corresponds to a stage in the execution flow of Sandstone.

Step	Description
1	Take the Reset exception
2.	Start initializing the hardware
3.	Remap memory
4.	Initialize comm^{n} hardware
5.	Boot loader - copy payload & relinquish control

Fig:- Sandstone execution flow.

Module - 5

Q9. a) Explain the basic architecture of a cache memory and basic operation of a cache controller. (10 M)

Soln :- Basic architecture of a cache memory :-

A simple cache memory is shown in fig.

It has 3 main parts
1) A directory store
2) A data section
3) Status information.

All 3 parts of the cache memory are present for each cache line. The cache must know where the information stored in a ~~th~~ cache line originates from in main memory. It uses a directory store to hold the addr identifying where the cache line was copied from main memory. The directory entry is known as a cache-tag.

A cache memory must also store the data read from the main memory.

There are also status bits in cache memory to maintain state information. Two common bits are the valid bit & dirty bit. A valid bit marks a cache line as active, meaning it contains live data originally taken from main memory & is currently available to the processor core on demand. A dirty bit defines whether or not a ~~th~~ cache line contain data that is different from the value it represents in main memory.

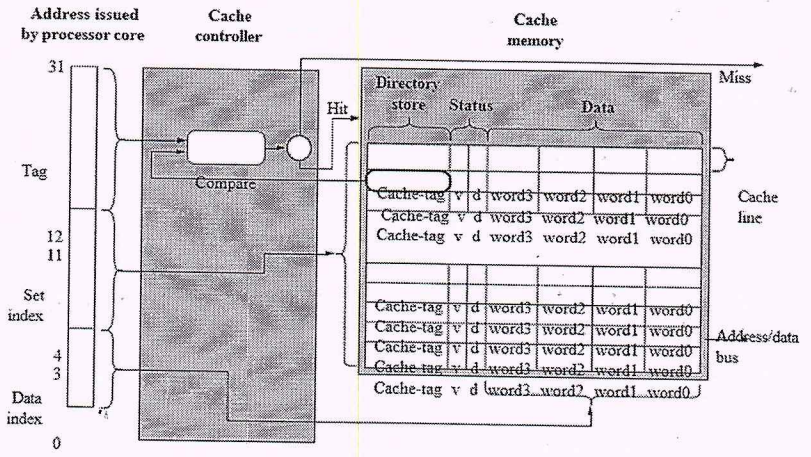


Figure 1.24 A 4 KB cache consisting of 256 cache lines of four 32-bit words.

Basic operation of a cache controller :-

- The cache controller is h/w that copies code or data from main memory to cache memory automatically.
- It performs this task automatically to conceal cache operation from the s/w it supports.
- Thus same appn software can run unaltered on sys with & without a cache.
- The cache controller intercepts read & write memory requests before passing them on to the memory controller.
- It processes a request by dividing the addr of the request into 3 fields, the tag field, the set index field & the data index field.
- First the controller uses the set index portion of the address to locate the cache line within the cache memory that might hold requested

*) This cache line contains the cache-tag & status bits, which the controller uses to determine the actual data stored there.

*) The controller then checks the valid bit to determine if the cache line is active & compares the cache-tag to the tag field of the requested addr.

*) If both the status check & comparison succeed, it is a cache hit.

*) If either the status check or comparison fails, it is a cache miss.

*) On a cache miss, the controller copies an entire cache line from main memory to cache memory & provide the requested code or data to the proc^r. This copying of cache line from main memory to cache memory is called cache line fill.

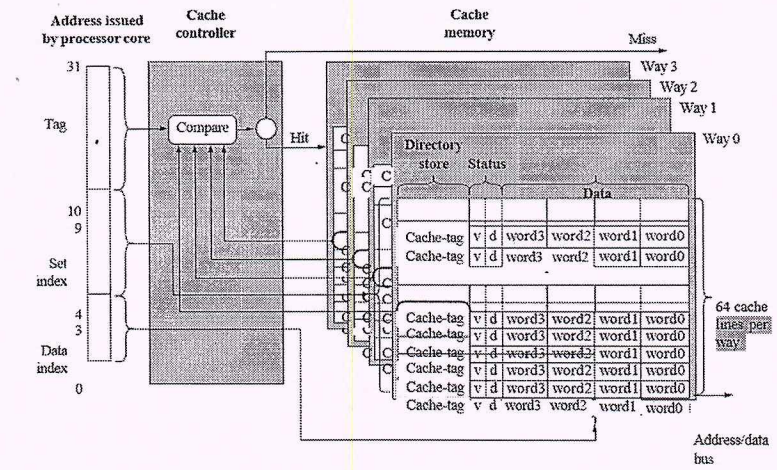
*) On a cache hit, the controller supplies the code or data directly from cache memory to the processor.

b) With a neat diagram, explain a 4KB, four way set associative cache. (10M)

Soln :- *) This structural design feature is a change that divides the cache memory into smaller equal units called way.

*) Fig shows 4 KB cache, however, the set index now addr more than one cache line — it points to one cache line in each way.

*) Instead of one way of 256 lines, the cache has 4 ways of 64 lines.



A 4 KB, four-way set associative cache. The cache has 256 total cache lines, which are separated into four ways, each containing 64 cache lines. The cache line contains four words.

*) The 4 cache line with the same set index are said to be in the same set, which is the origin of the name "set index".

*) The set of cache lines pointed to by the set index are set associative.

*) A data or code block from main memory can be allocated to any of the 4 ways in a set without affecting progm behaviour,

*) Two sequential blocks from main memory can be stored as cache lines in the same way or two different ways.

*) The size of the area of main memory that maps to cache is now 1KB instead of 4B.

*) This means that the likelihood of mapping cache line is 1/0th less likely to be evicted.

*) A routine B, & the data array would establish unique places in the 4 available locations in a set.
*) This assumes that the size of each routine & the data are less than the new smaller 1KB area that maps from main memory.

Q.10. a) Explain the write buffers & measuring cache efficiency. (08 m)

Soln:- Write Buffers:-

*) A write Buffer is a very small, fast FIFO memory buffer that temporarily holds data that the processor would normally write to main memory.
*) In a sym without a write buffer, the processor writes directly to main memory.
*) In a sym with a write buffer, data is written at high speed to the FIFO & then emptied to slower main memory.
*) The write buffer reduces the processor time taken to write small blocks of sequential data to main memory.
*) A write buffer also improves cache performance the improvement occurs during cache line evictions.
*) Data written to the write buffer is not available for reading until it has exited the write buffer to main memory.

* The same holds true for an evicted cache line: it too cannot be read while it is in the write buffer.
* This is one of the reasons that the FIFO depth of a write buffer is usually quite small, only a few cache lines deep
* Some write buffers are not strictly FIFO buffers.
* Coalescing is also known as write merging, write collapsing or write combining

Measuring Cache Efficiency:-

There are 2 terms used to characterize the cache efficiency of a pgm: The cache hit rate and the cache miss rate.

* The hit rate is the no. of cache hits divided by the total no. of memory requests over a given time interval. The value is expressed as a %.

$$\text{hit rate} = \frac{\text{cache hits}}{\text{memory requests}} \times 100$$

* The miss rate is similar in pgm's. The total cache misses divided by the total no. of memory requests expressed as a percentage over a time interval.
* The hit rate & miss rate can measure reads, writes or both, which means that the terms can be used to describe performance information in several ways.

*) Two other terms used in cache performance measurement are the hit time - the time it takes to access a memory location in the cache & the miss penalty - the time it takes to load a cache line from the main memory into cache.

b) Explain the cache Policy. (12M)

Soln:- Cache Policy:-

There are 3 policies that determine the operation of a cache

i) The write policy → which determines where data is stored during processor write operation

ii) The replacement policy → where this policy selects the cache line in a set that is used for the next line fill during a cache miss.

iii) The allocation policy → determines when the cache controller allocates a cache line.

i) Write Policy:-

*) When the processor core writes to memory, the cache controller has two alternatives for its write policy.

*) The controller can write to both cache & main memory, updating the values in both locations, this approach is known as write through.

*) The cache controller can write to cache memory & not update the main memory, this is called writeback or copy back.

ii) Cache line Replacement Policy:-

*) On a cache miss, the cache controller must select a cache line from the available set in cache memory to store the new information from main memory.

*) The cache line selected for replacement is known as a victim.

*) If victim contains valid, dirty data, the controller must write the dirty data from the cache memory to main memory before it copies new data into the victima cache line.

*) This process of selecting & replacing a victim cache line is known as eviction.

*) The strategy implemented in a cache contro to select the next victim called its replacement policy.

*) The replacement policy selects a cache line from the available associative member set, i.e it selects the way to use in the next cache line replacement.

iii) Allocation Policy on a Cache Miss

*) There are two strategies ARM caches may use to allocate a cache line after a the occurrence of a cache miss.

*) The first strategy is called read-allocate

*) The second strategy is called read-write-allocate

*) A read allocate on cache miss policy allocates a cache line only during a read from main memory.

*) If the victim cache line contains a valid data, then it is written to main memory before the cache line is filled with new data.

*) A read-write allocate on cache miss policy allocates a cache line for either a read or write to memory.

*) Any Load or store operation made to main memory, which is not in cache memory, allocates a cache line.

*) On memory reads the controller uses a read - allocate policy.

*) On a write, the controller also allocates a cache line.

*) If the victim cache line contains valid data then it is first written back to main memory before the cache controller fills the victim cache line with new data from main memory

*) If the cache line is not valid, it simply does a cache line fill.

*) After the cache line is filled from main memory, the controller write the data to the corresponding data location within the cache line.

*) The cached core also updates main memory if it is a write-through cache.

Prepared By:

(Nancy. P)
KLS VDIT, Haliyal

HOD [KLS VDIT, Haliyal]

DEAN ACADEMICS