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

* * * * *

Microcontrollers_BCSE402
VTU SCheme

Q. 1.a

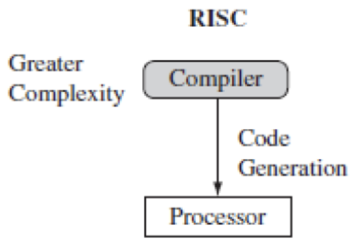
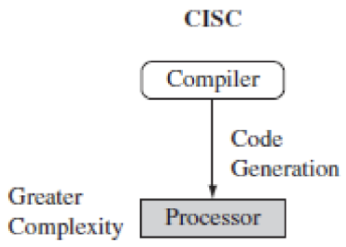
The RISC philosophy is implemented with four major design rules:

a. **Instructions:** RISC has a reduced number of instruction classes. These classes provide simple operations so that each is executed in a single cycle. Each instruction is a fixed length to allow the pipeline to fetch future instructions before decoding the current instruction.
b. **Pipeline:** The processing of instructions is broken down into smaller units that can be executed in parallel by pipelines.
c. **Register:** RISC machines have a large general-purpose register set. Any register can contain either data or an address.
d. **Load-store architecture:** The processor operates on the data held in registers. Separate load and store instructions transfer data between the register bank and external memory.

These design rules allow a RISC processor to be simpler, and thus the core can operate at higher clock speed.

Q. 1.b

<u>Difference between RISC and CISC</u>

RISC	**CISC**
 RISC Greater Complexity Compiler Code Generation Processor	 CISC Compiler Code Generation Greater Complexity Processor
Emphasizes on compiler complexity	Emphasizes on processor complexity
Simple but powerful instructions	Instructions are more complicated
Executes instruction in single cycle	Takes many cycle to execute
Instructions are of fixed length	Instructions are of variable length
Have large set of general purpose registers	Have limited set of general purpose registers
Any register can contain either data or an address	Dedicated registers for specific purpose
Separate load and store instructions transfer data between the register and external memory.	MOV instructions can be used to transfer between register and memory.

Q. 1.c

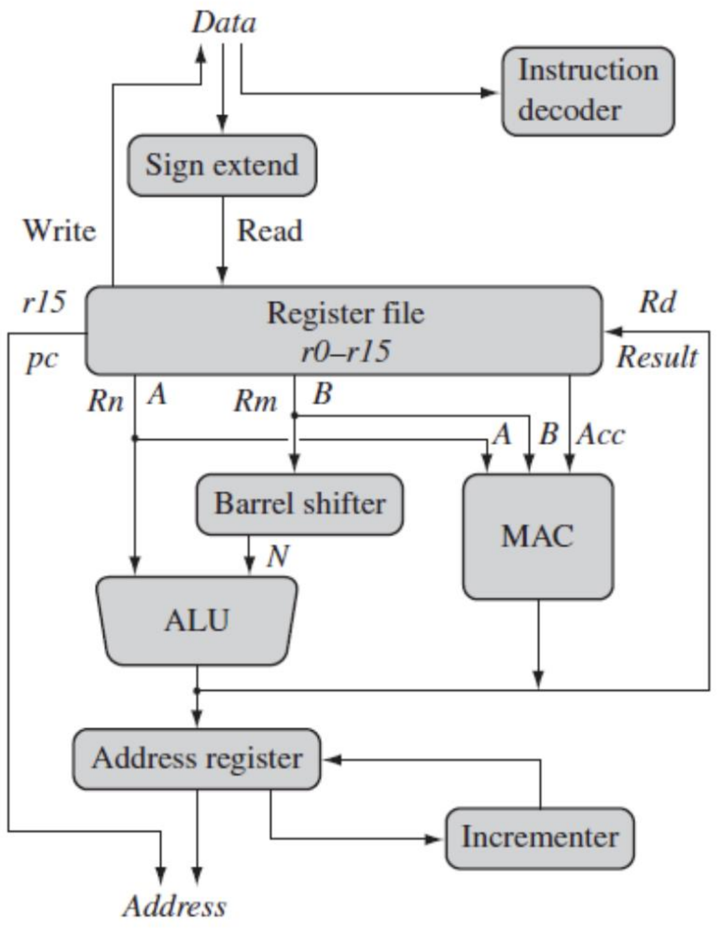


Figure: ARM Core dataflow Model

A programmer can think of an ARM core as functional units connected by data buses, as shown in the following Figure.
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 instruction to execute or a data item.
Figure shows a Von Neumann implementation of the ARM—data items and instructions share the same bus. (In contrast, Harvard implementations of the ARM use two different buses).
The instruction decoder translates instructions before they are executed. Each instruction executed belongs to a particular instruction set.
The ARM processor, like all RISC processors, uses load-store architecture—means it has two instruction types for transferring data in and out of the processor:
-load instructions copy data from memory to registers in the core
-store instructions copy data from registers to memory
There are no data processing instructions that directly manipulate data in memory. Thus, data

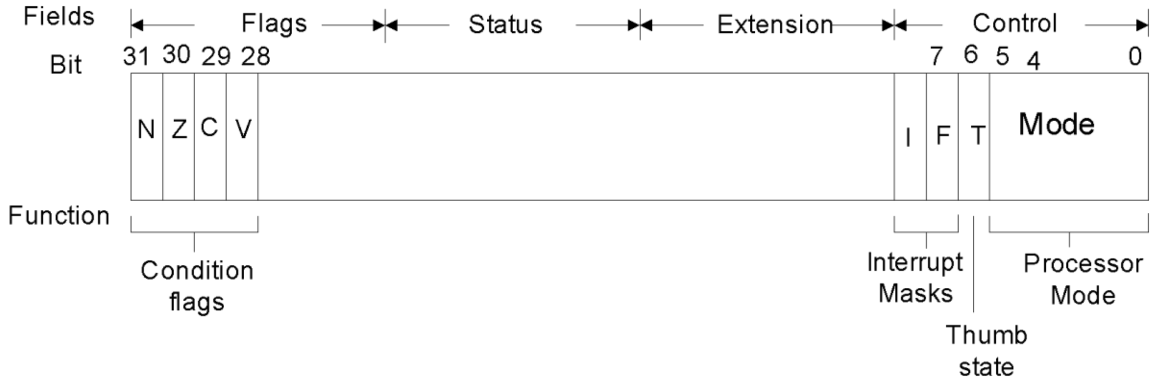
processing is carried out in registers.
Data items are placed in the register file—a storage bank made up of 32-bit registers.
Since the ARM core is a 32-bit processor, most instructions treat the registers as holding signed or unsigned 32-bit values. The sign extend hardware converts signed 8-bit and 16-bit numbers to 32-bit values as they are read from memory and placed in a register.
ARM instructions typically have two source registers, Rn and Rm, and a single result or destination register, Rd. Source operands are read from the register file using the internal buses
A and B, respectively.
The ALU (arithmetic logic unit) or MAC (multiply-accumulate unit) takes the register values Rn and Rm from the A and B buses and computes a result. Data processing instructions write the result in Rd directly to the register file.
Load and store instructions use the ALU to generate an address to be held in the address register and broadcast on the Address bus.
One important feature of the ARM is that register Rm alternatively can be preprocessed in the barrel shifter before it enters the ALU. Together the barrel shifter and ALU can calculate a wide range of expressions and addresses.
After passing through the functional units, the result in Rd is written back to the register file using the Result bus.
For load and store instructions the Incrementer updates the address register before the core reads or writes the next register value from or to the next sequential memory location.
The processor continues executing instructions until an exception or interrupt changes the normal execution flow.

Q. 2.a

<u>CPSR (Current Program Status Register)</u>

Q6. Explain the various fields in current program status register (CPSR) with neat diagram.

Answer: Figure below shows the basic layout of a generic program status register.



- The cpsr is divided into four fields, each 8 bits wide: flags, status, extension and control.
- In current designs the extension and status fields are reserved for future use.
- The control field contains the processor mode, state and interrupts mask bits.

- The flag field contains the condition flags.
- The following table gives the bit patterns that represent each of the processor modes in the cpsr.

Mode	Mode[4:0]
Abort	10111
Fast interrupt request	10001
Interrupt request	10010
Supervisor	10011
System	11111
Undefined	11011
User	10000

- When cpsr bit 5, T=1, then the processor is in Thumb state. When T=0, the processor is in ARM state.
- The cpsr has two interrupt mask bits, 7 and 6 (I and F) which control the masking Interrupt request (IRQ) and Fast Interrupt Request (FIR).
- Condition flags are updated by comparisons and the result of ALU operations that specify the S instruction suffix.
- For example, if SUBS subtract instruction results in a register value of zero, then the Z flag in the cpsr is set.
- The following table shows the conditional flags:

Flag	Flag Name	Set when
N	Negative	Bit 31 of the result is a binary 1
Z	Zero	The result is zero, frequently used to indicate equality
C	Carry	The result causes an unsigned carry
V	Overflow	The result causes a signed overflow

Q.2. b

- Pipeline is the mechanism to speed up execution by fetching the next instruction while other instruction are being decoded and executed.
- Figure 1 shows the ARM7 three-stage pipeline.

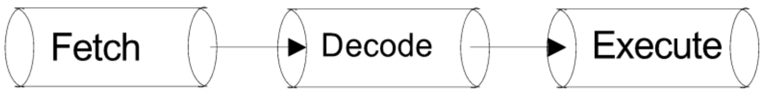
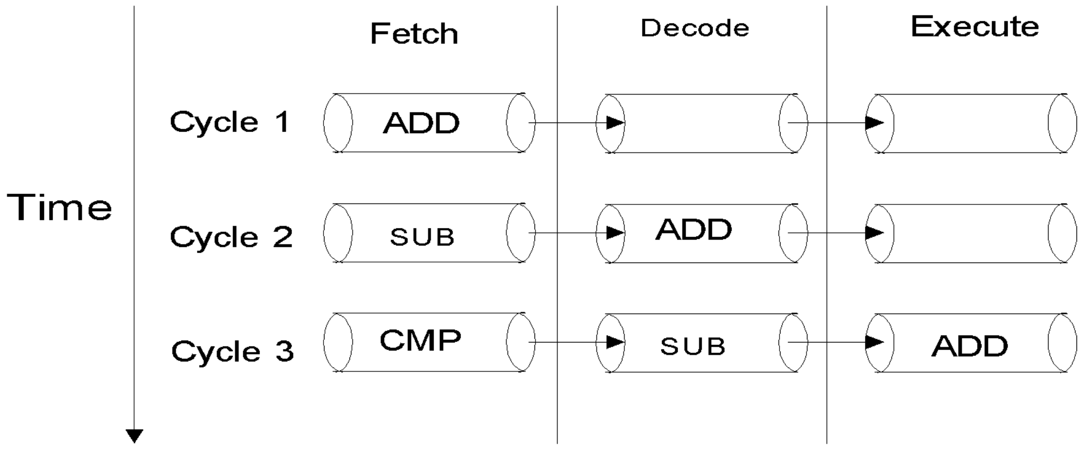


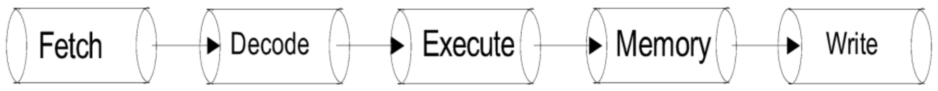
Figure 1: ARM7 Three-stage pipeline

- Fetch loads an instruction from memory.
- Decode identifies the instruction to be executed.
- Execute processes the instruction and writes the result back to a register.
- Figure 2 illustrates the pipeline using a simple example. It shows a sequence of three instructions being fetched, decoded and executed by the processor.
- Each instruction takes a single cycle to complete after the pipeline is filled.
    - In the first cycle, the core fetches the ADD instruction from the memory.
    - In the second cycle, the core fetches the SUB instruction and decode the ADD instruction.

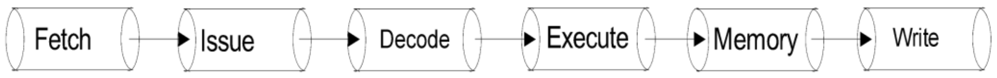
- o In the third cycle, the core fetches CMP instruction from the memory, decode the SUB instruction and execute the ADD instruction.
- o The ADD instruction is executed, the SUB instruction is decoded, and the CMP instruction is fetched. This procedure is called **filling the pipeline.**



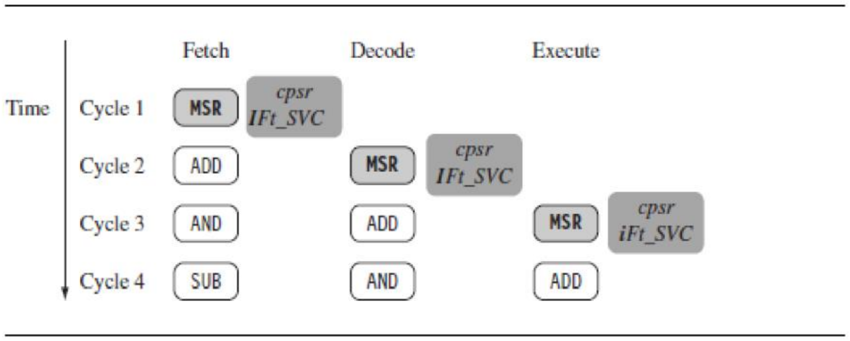
- The pipeline design for each ARM family differs. For example, the ARM9 core increases the pipeline length to five stages as shown in the figure below.



- The ARM10 increases the pipeline length still further by adding a sixth stage as shown in the figure below.



- As the pipeline length increases the amount of work done at each stage is reduced, which allows the processor to attain a higher operating frequency. This in turn increases the performance.
- **Pipeline Executing Characteristics**
  a. The ARM pipeline has not processed an instruction until it passes completely through the execute stage. For example, an ARM7 pipeline (with three stages) has executed an instruction only when the fourth instruction is fetched. Figure below shows an instruction sequence on an ARM7 pipeline.



In the execute stage, the pc always points to the address of the instruction plus 8 bytes. In other words, the pc always points to the address of the instruction being executed plus two instructions ahead as shown in figure 2 below

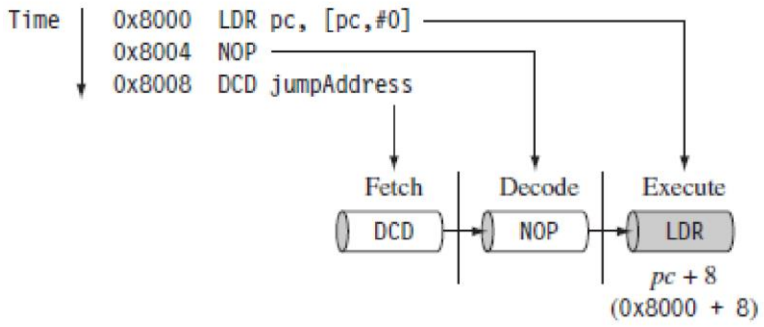


Figure 2: Example: pc = address + 8

The execution of a branch instruction or branching by the direct modification of the pc causes the ARM core to flush its pipeline.

a. ARM10 uses branch prediction, which reduces the effect of a pipeline flush by predicting possible branches and loading the new branch address prior to the execution of the instruction.
b. An instruction in the execute stage will complete even though an interrupt has been raised.

Q. 2. C

There are three core extensions wrap around ARM processor: cache and tightly coupled memory, memory management and the coprocessor interface.

1. Cache and tightly coupled memory: The cache is a block of fast memory placed between main memory and the core. With a cache the processor core can run for the majority of the time without having to wait for data from slow external memory.

- o ARM has two forms of cache. The first found attached to the Von Neumann-style cores. It combines both data and instruction into a single unified cache as shown in the figure 1 below.

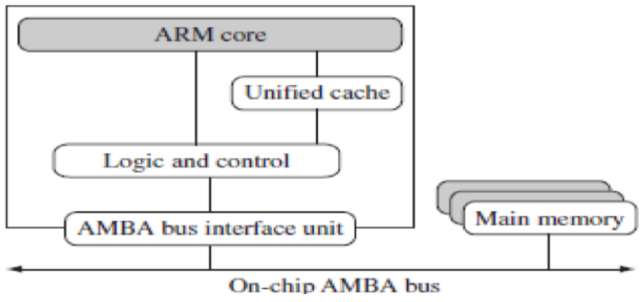


Figure 1: A simplified Von Neumann architecture with cache.

- The second form, attached to the Harvard-style cores, has separate cache for data and instruction as shown figure 2

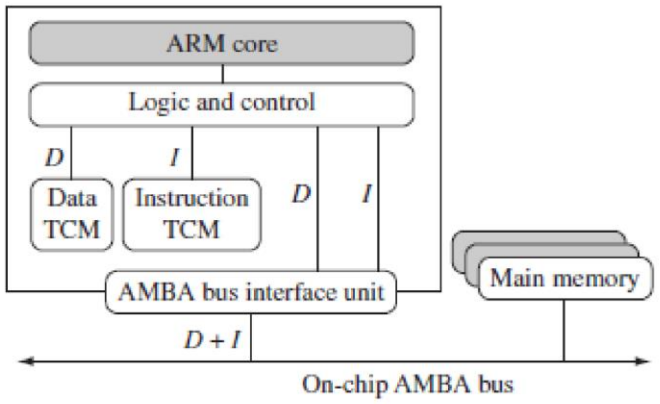


Figure 2: A simplified Harvard architecture with TCMs.

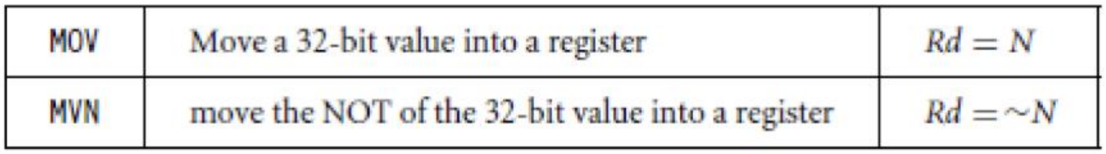
- A cache provides an overall increase in performance but will not give predictable execution.
- But for real-time systems it is paramount that code execution is *deterministic.*
- This is achieved using a form of memory called *tightly coupled memory (TCM).*
- TCM is fast SRAM located close to the core and guarantees the clock cycles required to fetch instructions or data.

Q.3. a

Data Processing Instructions

- The data processing instructions manipulate data within registers. They are move instructions, arithmetic instructions, logical instructions, compare instructions and multiply instructions.
- Most data processing instructions can process one of their operands using the barrel shifter.
- If S is suffixed on a data processing instruction, then it updates the flags in the cpsr.

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

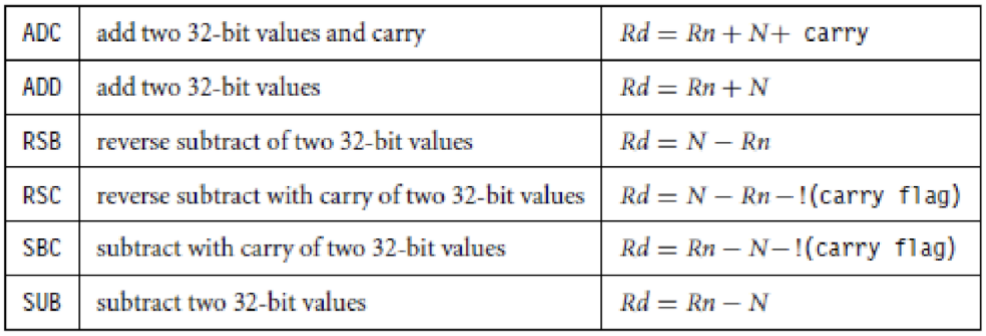


`MOV`	Move a 32-bit value into a register	$Rd = N$
`MVN`	move the NOT of the 32-bit value into a register	$Rd = \sim N$

ARITHMETIC INSTRUCTIONS:

- The arithmetic instructions implement addition and subtraction of 32-bit signed and unsigned values.

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

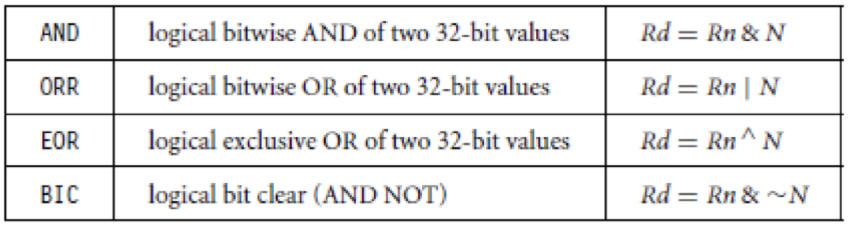


`ADC`	add two 32-bit values and carry	$Rd = Rn + N +$ `carry`
`ADD`	add two 32-bit values	$Rd = Rn + N$
`RSB`	reverse subtract of two 32-bit values	$Rd = N - Rn$
`RSC`	reverse subtract with carry of two 32-bit values	$Rd = N - Rn -$ `!(carry flag)`
`SBC`	subtract with carry of two 32-bit values	$Rd = Rn - N -$ `!(carry flag)`
`SUB`	subtract two 32-bit values	$Rd = Rn - N$

LOGICAL INSTRUCTIONS:

- Logical instructions perform bitwise operations on the two source registers.

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

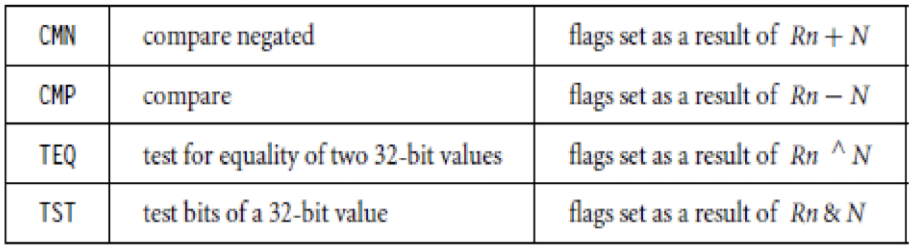


`AND`	logical bitwise AND of two 32-bit values	$Rd = Rn \& N$
`ORR`	logical bitwise OR of two 32-bit values	$Rd = Rn \mid N$
`EOR`	logical exclusive OR of two 32-bit values	$Rd = Rn \wedge N$
`BIC`	logical bit clear (AND NOT)	$Rd = Rn \& \sim N$

<u>COMPARISON INSTRUCTIONS:</u>

- The comparison instructions 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 the bits have been set, the information can be used to change program flow by using conditional execution.

Syntax: <instruction> {<cond>} Rn, N

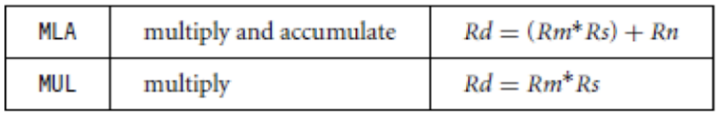


CMN	compare negated	flags set as a result of $Rn + N$
CMP	compare	flags set as a result of $Rn - N$
TEQ	test for equality of two 32-bit values	flags set as a result of $Rn \wedge N$
TST	test bits of a 32-bit value	flags set as a result of $Rn \& N$

<u>MULTIPLY INSTRUCTIONS:</u>

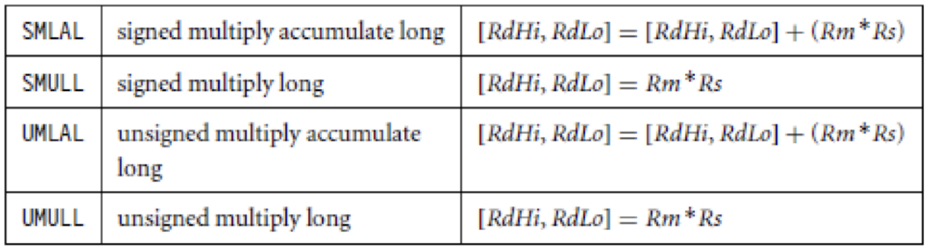
- The multiply instructions multiply the contents of a pair of registers and depending upon the instruction, accumulate the results in another register.
- The long multiplies accumulate onto a pair of registers representing a 64-bit value.

Syntax: MLA {<cond>} {S} Rd, Rm, Rs, Rn
MUL {<cond>} {S} Rd, Rm, Rs



MLA	multiply and accumulate	$Rd = (Rm^*Rs) + Rn$
MUL	multiply	$Rd = Rm^*Rs$

Syntax: <instruction> {<cond>} {S} RdLo, RdHi, Rm, Rs



SMLAL	signed multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm^*Rs)$
SMULL	signed multiply long	$[RdHi, RdLo] = Rm^*Rs$
UMLAL	unsigned multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm^*Rs)$
UMULL	unsigned multiply long	$[RdHi, RdLo] = Rm^*Rs$

Q. 3.b

- A branch instruction changes the flow of execution or is used to call a routine.
- This type of instruction allows programs to have subroutines, if-then-else structures, and loops.
- The change of execution flow forces the program counter (pc) to point to a new address.

```
Syntax: B{<cond>} label
        BL{<cond>} label
        BX{<cond>} Rm
        BLX{<cond>} label | Rm
```

B	branch	*pc* = *label*
BL	branch with link	*pc* = *label* *lr* = address of the next instruction after the BL
BX	branch exchange	*pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1
BLX	branch exchange with link	*pc* = *label*, *T* = 1 *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 *lr* = address of the next instruction after the BLX

- T refers to the Thumb bit in the cpsr.

- When instruction set T, the ARM switches to Thumb state.
- The example shown below is a forward branch. The forward branch skips three instructions.

```
        B       forward
        ADD     r1, r2, #4
        ADD     r0, r6, #2
        ADD     r3, r7, #4
forward
        SUB     r1, r2, #4
```

- The branch with link (BL) instruction changes the execution flow in addition overwrites the link register lr with a return address. The example shows below a fragment of code that branches to a subroutine using the BL instruction.

```
        BL      subroutine      ; branch to subroutine
        CMP     r1, #5          ; compare r1 with 5
        MOVEQ   r1, #0          ; if (r1==5) then r1 = 0
        :
subroutine
        <subroutine code>
        MOV     pc, lr          ; return by moving pc = lr
```

- **The branch exchange (BX)** instruction uses an absolute address stored in register Rm. It is primarily used to branch to and from Thumb code. The T bit in the cpsr is updated by the least significant bit of the branch register.
- Similarly, **branch exchange with link (BLX)** instruction updates the T bit of the cpsr with the least significant bit and additionally sets the link register with the return address.

Q.3.c

`mov r1, r2`
→ Copy the value of `r2` into `r1`.

`add r1, r2, r4`
→ Add `r2` and `r4`, store result in `r1`.

`bic r3, r2, r5`
→ Clear bits in `r2` where `r5` has 1s, store in `r3`.
(`r3 = r2 & (~r5)`)

`cmp r3, r4`
→ Compare `r3` and `r4` by subtracting (sets flags, no result stored).

`UMLAL r1, r2, r3, r4`
→ Multiply `r3 × r4`, add to 64-bit value in `r2:r1`, store result in `r2:r1`.

Q.4.a

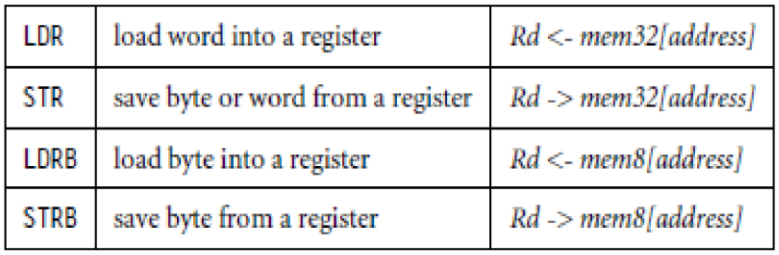
<u>LOAD-STORE INSTRUCTIONS (Memory Access Instructions)</u>

- Load-store instructions transfer data between memory and processor registers. There are three types of load-store instructions: single-register transfer, multiple-register transfer, and swap.

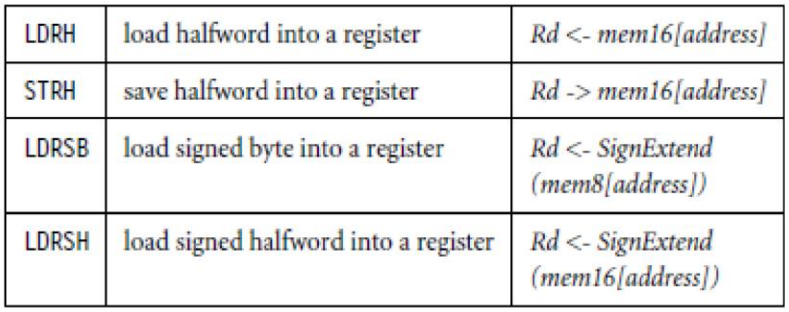
a) Single-Register Transfer

- These instructions are used for moving a single data item in and out of a register.
- Here are the various load-store single-register transfer instructions.

Syntax: <LDR|STR>{<cond>}{B} Rd, addressing[1]
LDR{<cond>}SB|H|SH Rd, addressing[2]
STR{<cond>}H Rd, addressing[2]



LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*



LDRH	load halfword into a register	*Rd <- mem16[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*
LDRSB	load signed byte into a register	*Rd <- SignExtend (mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend (mem16[address])*

Example:

1. LDR r0, [r1]
   - o This instruction loads a word from the address stored in register r1 and places it into register r0.

2. STR r0, [r1]
   - This instruction goes the other way by storing the contents of register r0 to the address contained in register r1.

A **full descending stack** is a **stack that grows downward (decreasing memory addresses)**, and the **stack pointer (SP)** always points to the **last used (full) location** in the stack.
Full → SP points to a full (occupied) location.
Descending → Stack grows to **lower addresses** (top moves down).
STMFD SP!, {r0}   ; Store r0 in stack, Full Descending (STMFD)
`STMFD` = Store Multiple Full Descending

`SP!` = Update SP after storing (post-decrement)

`{r0}` = Register to push
Decrement SP → `SP = SP - 4`

Store `r0` at new `SP` address

When popping (restoring) a value:

asm
CopyEdit
```
LDMFD SP!, {r0}   ; Load r0 from stack, Full Descending (LDMFD)
```

- `LDMFD` = Load Multiple Full Descending
- `SP!` = Update SP after loading (post-increment)
- `{r0}` = Register to pop into

This does:

1. Load `r0` from address at `SP`
2. Increment SP → `SP = SP + 4`

Example:
STMFD SP!, {r0, r1, r2} ; Push r0, r1, r2 (r2 at lowest addr)
LDMFD SP!, {r0, r1, r2} ; Pop r0, r1, r2
Assume initial SP = 0x1000

Address	**Value**
0x0FF4	r2 value
0x0FF8	r1 value
0x0FFC	r0 value
0x1000	(old SP)

New SP = 0x0FF4 (after pushing 3 registers)

Q.4.c

```
        AREA SUM10, CODE, READONLY
        EXPORT __main
__main
ENTRY
        MOV R1, #0X01
        MOV R2, #0

LOOP            ADD R2, R2, R1
                ADD R1, R1, #1
                CMP R1, #0X0B
                BNE LOOP
                LDR R0, = Result
                STRB R2, [R0]
STOP    B       STOP

        AREA data2, DATA, READWRITE

Result  DCB 0x0
        END
```

```
output:
R1=01
        R2=00
        R2=37
```

Result is 55 but in the hexadecimal it is 37 so it will gives output as 37

Q.5.a

On most 32-bit ARM systems, the basic C data types and their typical sizes are:

Data Type	**Size (Bytes)**	**Description**
`char`	**1**	**8-bit character**
`short`	**2**	**16-bit integer**
`int`	**4**	**32-bit signed integer**
`unsigned int`	**4**	**32-bit unsigned integer**
`long`	**4**	**32-bit signed integer (same as `int`)**
`float`	**4**	**32-bit IEEE-754 floating point**
`double`	**8**	**64-bit IEEE-754 floating point**
`void*`	**4**	**32-bit address pointer**

C Program to Compute Checksum of 64-Word Packet

```
#include <stdio.h>
#include <stdint.h>
```

```
#define PACKET_SIZE 64

// Function to compute checksum
uint32_t compute_checksum(uint32_t packet[PACKET_SIZE]) {
    uint32_t checksum = 0;
    for (int i = 0; i < PACKET_SIZE; i++) {
        checksum += packet[i];
    }
    return checksum;
}

int main() {
    // Sample data packet
    uint32_t data_packet[PACKET_SIZE];
    for (int i = 0; i < PACKET_SIZE; i++) {
        data_packet[i] = i + 1;  // Fill with values 1 to 64
    }

    uint32_t result = compute_checksum(data_packet);
    printf("Checksum: %u\n", result);

    return 0;
}
```

Sample Compiler Output

```
compute_checksum:
    PUSH    {r4, lr}
    MOV     r2, #0          ; checksum = 0
    MOV     r3, #0          ; i = 0
.L2:
    CMP     r3, #64
    BGE     .L3
    LDR     r1, [r0, r3, LSL #2]   ; load packet[i]
    ADD     r2, r2, r1             ; checksum += packet[i]
    ADD     r3, r3, #1             ; i++
    B       .L2
.L3:
    MOV     r0, r2
    POP     {r4, pc}
```

Q. 5.b

1. `for` Loop

✅Best when the number of iterations is known in advance.

c
CopyEdit
```
for (initialization; condition; increment) {
    // Loop body
}
```
```
for (int i = 0; i < 5; i++) {
   printf("%d\n", i);
}
```

2. `while` Loop

✅Best when the number of iterations is not known, but depends on a condition.

c
CopyEdit
```
while (condition) {
    // Loop body
}
```
```
int i = 0;
while (i < 5) {
   printf("%d\n", i);
   i++;
}
```

3. `do...while` Loop

✅Best when the loop must run at least once, even if the condition is false initially.

c
CopyEdit
```
do {
    // Loop body
} while (condition);
```
```
int i = 0;
do {
    printf("%d\n", i);
    i++;
} while (i < 5);
```

The termination condition

```
SUBS r1,r1,#1 ; compare i with 1, i=i-1
BGT loop ; if (i+1>1) goto loop


SUB r1,r1,#1 ; i--
CMP r1,#0 ; compare i with 0
BGT loop ; if (i>0) goto loop
```

i!=0 for signed or unsigned loop counters. It saves one instruction over the condition i>0 for signed i.

Loops with a Fixed Number of Iterations

```
int checksum_v5(int *data)
{
unsigned int i;
int sum=0;
for (i=0; i<64; i++)
{
sum += *(data++);
}
return sum;
}
```

Loops Using a Variable Number of Iterations

```
int checksum_v7(int *data, unsigned int N)
{
int sum=0;
for (; N!=0; N--)
{
sum += *(data++);
}
return sum;
}
```

Q.5.c

Pointer Aliasing

- ❖ Two pointers are said to alias when they point to the same address.
- ❖ If you write to one pointer, it will affect the value you read from the other pointer.
- ❖ In a function, the compiler often doesn’t know which pointers can alias and which pointers can’t. The compiler must be very pessimistic and assume that any write to a pointer may affect the value read from any other pointer, which can significantly reduce code efficiency.

```
void timers_v1(int *timer1, int *timer2, int *step)
{
*timer1 += *step;
*timer2 += *step;
}
```

```
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
```

Usually a compiler optimization called common subexpression elimination would kick in so that *step was only evaluated once, and the value reused for the second occurrence.

However, the compiler can’t use this optimization here. The pointers timer1 and step might alias one another.

In other words, the compiler cannot be sure that the write to timer1 doesn’t affect the read from step.

In this case the second value of *step is different from the first and has the value *timer1.

This forces the compiler to insert an extra load instruction.

Structure

```
typedef struct {int step;} State;
typedef struct {int timer1, timer2;} Timers;
void timers_v2(State *state, Timers *timers)
{
timers->timer1 += state->step;
timers->timer2 += state->step;
}
```

Q.6.a

Function Calls

ARM Procedure Call Standard (APCS): how to pass function arguments and return values in ARM registers.

ARM-Thumb Procedure Call Standard (ATPCS):covers ARM and Thumb interworking as well.

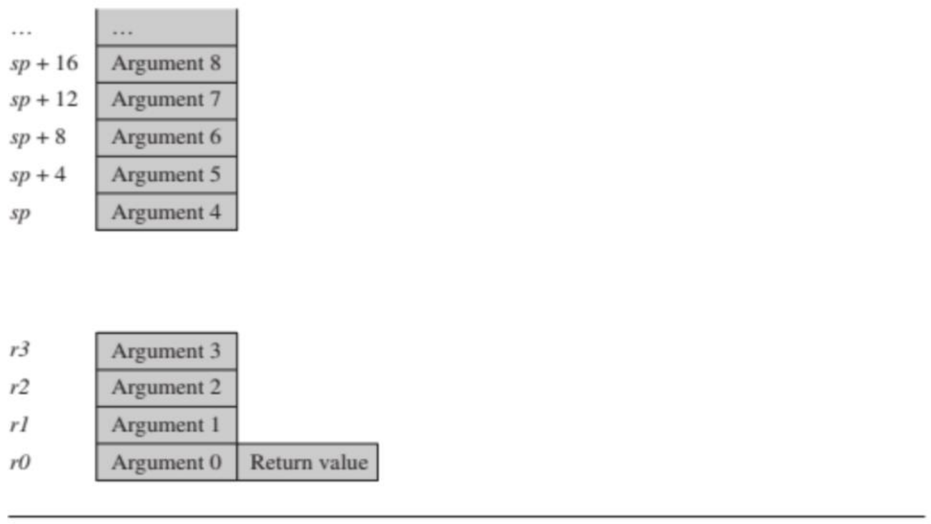
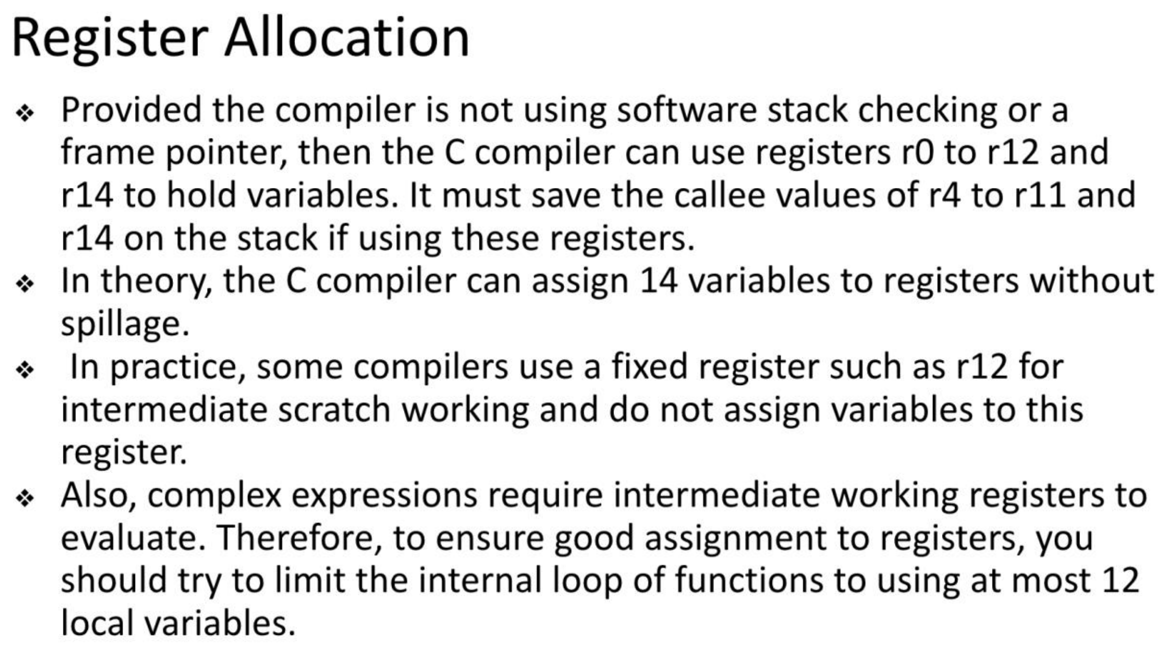


Figure 5.1 ATPCS argument passing.

Four-register rule

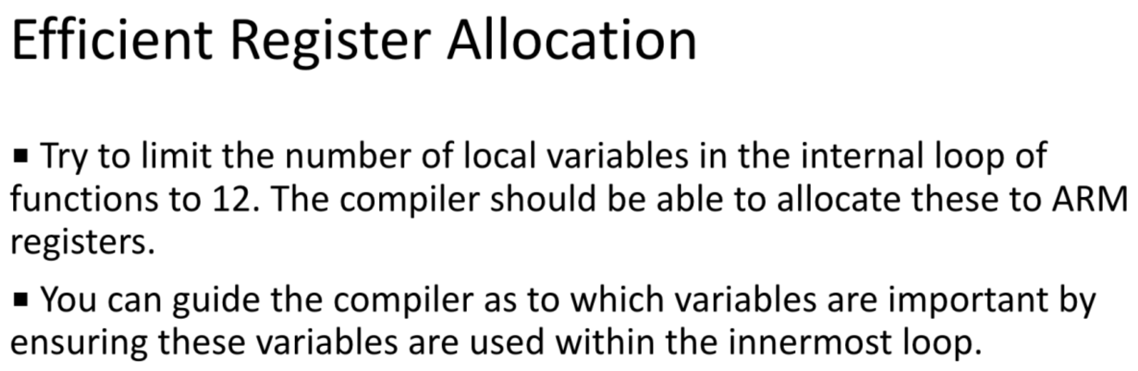
- ❖ Functions with four or fewer arguments are far more efficient to call than functions with five or more arguments.
- ❖ For functions with four or fewer arguments, the compiler can pass all the arguments in registers.
- ❖ For functions with more arguments, both the caller and callee must access the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

Q.6.b



Register Allocation

- ❖ 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.



Efficient Register Allocation

■ Try to limit the number of local variables in the internal loop of functions to 12. The compiler should be able to allocate these to ARM registers.

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

Q.6.c

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 inlined 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 a 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.

Q.7a

- ## What is an exception?

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

- ## Examples

•Resetting ARM core

•Failure of fetching instructions

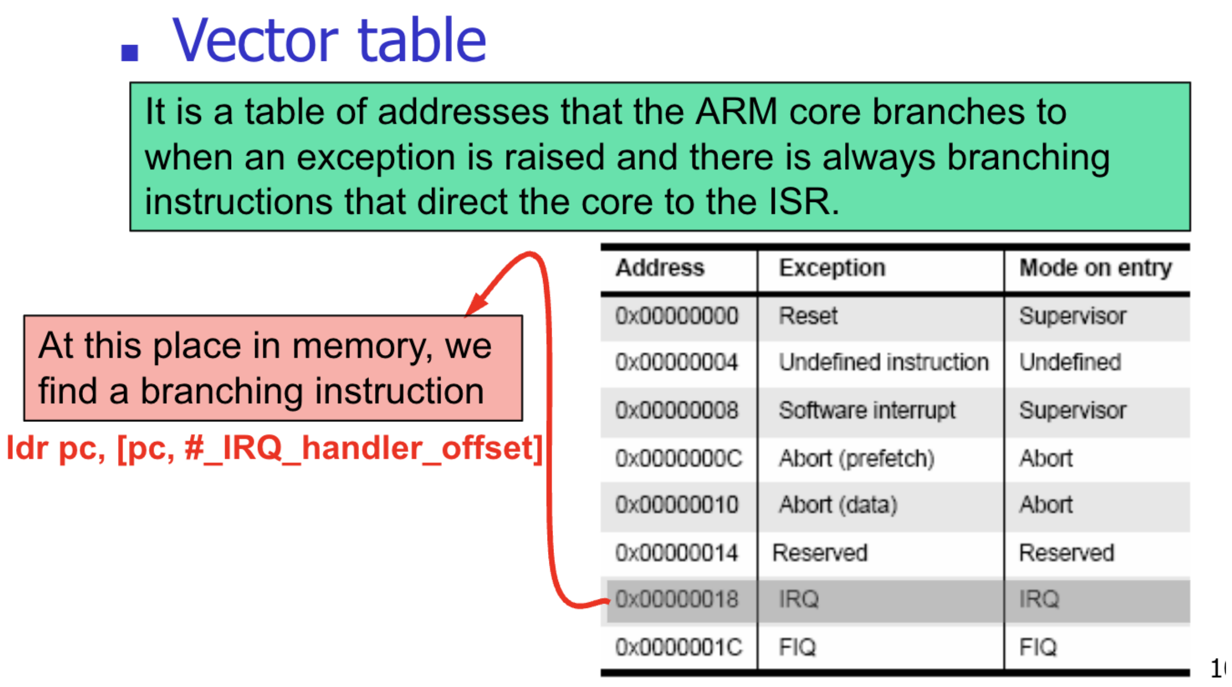
•HWI

•SWI

- ## Exceptions and modes

Each exception causes the ARM core to enter a specific mode.

Exception	**Mode**	**Purpose**
Fast Interrupt Request	FIQ	Fast interrupt handling
Interrupt Request	IRQ	Normal interrupt handling
SWI and RESET	SVC	Protected mode for OS
Pre-fetch or data abort	ABT	Memory protection handling
Undefined Instruction	UND	SW emulation of HW coprocessors



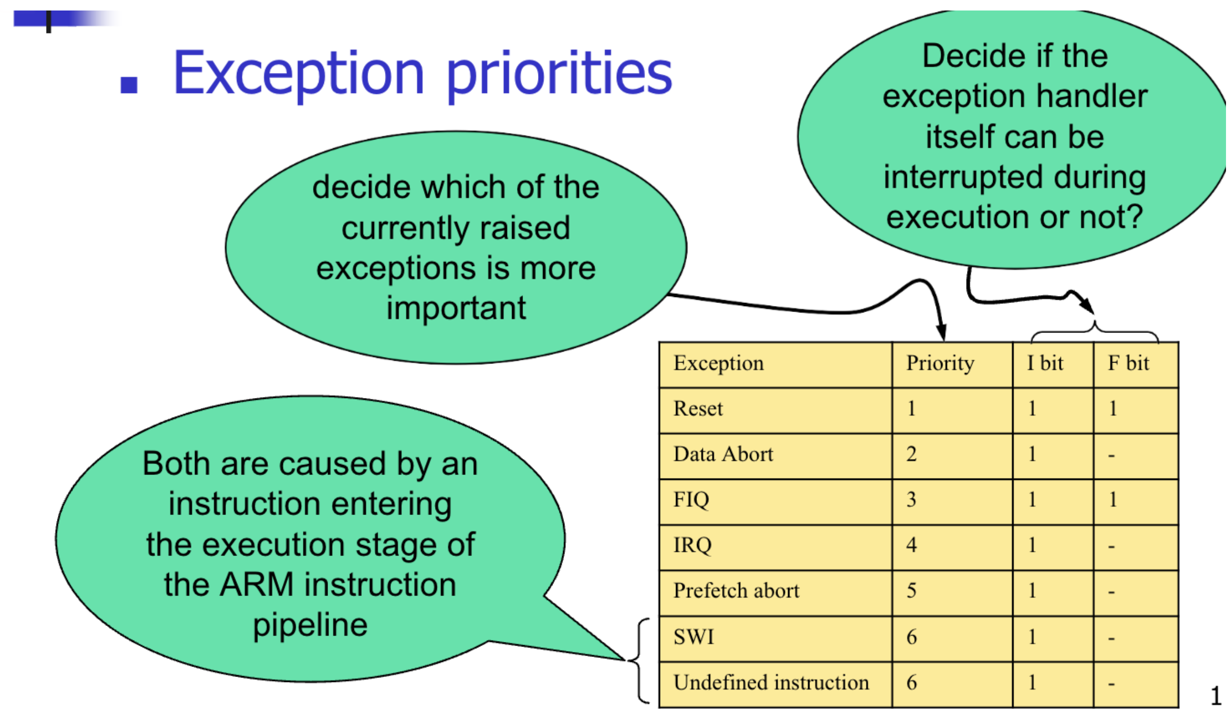
Vector table

It is a table of addresses that the ARM core branches to when an exception is raised and there is always branching instructions that direct the core to the ISR.

At this place in memory, we find a branching instruction

ldr pc, [pc, #_IRQ_handler_offset]

Address	Exception	Mode on entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	Reserved	Reserved
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ



Exception priorities

decide which of the currently raised exceptions is more important

Decide if the exception handler itself can be interrupted during execution or not?

Both are caused by an instruction entering the execution stage of the ARM instruction pipeline

Exception	Priority	I bit	F bit
Reset	1	1	1
Data Abort	2	1	-
FIQ	3	1	1
IRQ	4	1	-
Prefetch abort	5	1	-
SWI	6	1	-
Undefined instruction	6	1	-

Q.7.b

Assigning interrupts

It is up to the system designer who can decide which HW peripheral can produce which interrupt.

But system designers have adopted a standard design for assigning interrupts:

- SWI are used to call privileged OS routines.
- IRQ are assigned to general purpose interrupts like periodic timers.
- FIQ is reserved for one single interrupt source that requires fast response time.

Interrupt latency

It is the interval of time from an external interrupt signal being raised to the first fetch of an instruction of the ISR of the raised interrupt signal.

System architects try to achieve two main goals:

- To handle multiple interrupts simultaneously.
- To minimize the interrupt latency.

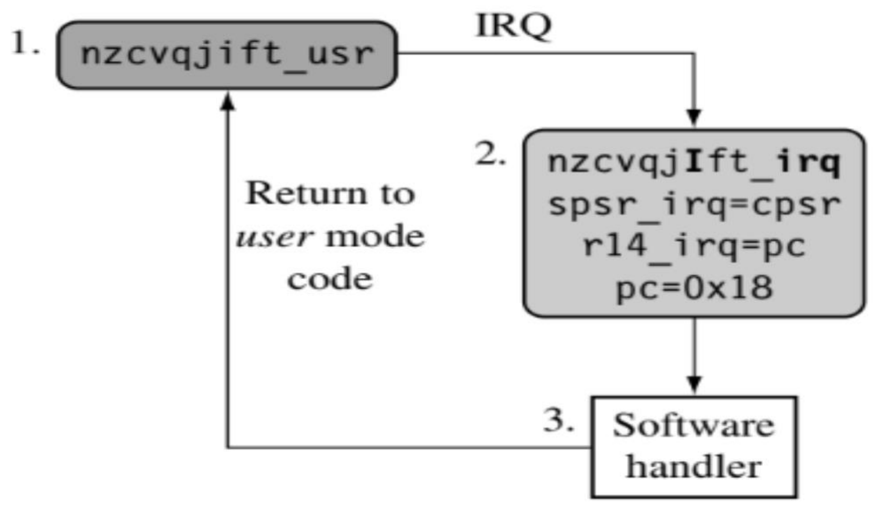
And this can be done by 2 methods:

- allow nested interrupt handling
- give priorities to different interrupt sources

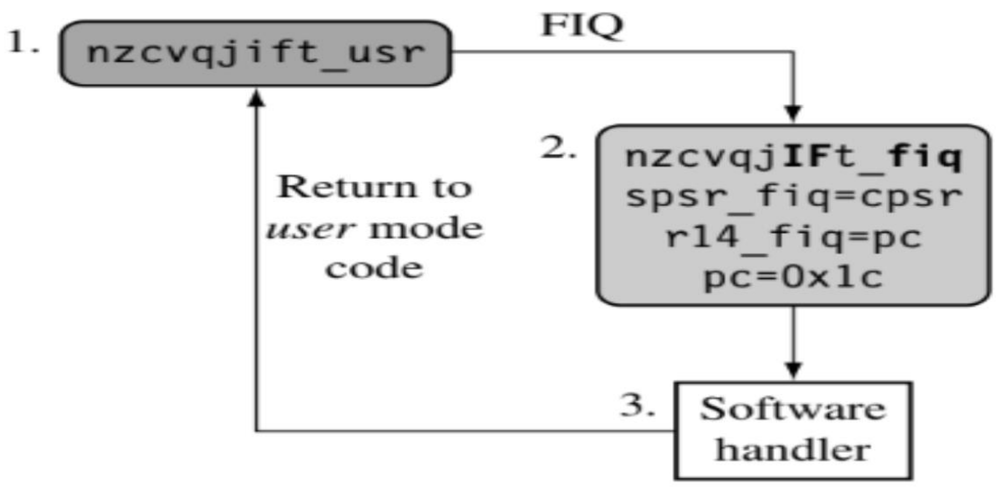
IRQ and FIQ Exceptions

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.

Interrupt Request (IRQ).



Fast Interrupt Request (FIQ).



Interrupts

- Enabling and disabling Interrupt

  This is done by modifying the ***CPSR***, this is done using only 3 ARM instruction:
  MRS To read *CPSR* MSR
  To store in *CPSR* BIC
  Bit clear instruction ORR
  OR instruction

Enabling an IRQ/FIQ Interrupt:

```
MRS     r1, cpsr
BIC r1, r1, #0x80/0x40
MSR     cpsr_c, r1
```

Disabling an IRQ/FIQ Interrupt:

```
MRS     r1, cpsr
ORR     r1, r1, #0x80/0x40
MSR     cpsr_c, r1
```

Enabling FIQ and IRQ Exceptions

cpsr value	IRQ			FIQ		
Pre	*nzcvqj**I**Ft_SVC*			*nzcvqjI**F**t_SVC*		
Code	`enable_irq`			`enable_fiq`		
		`MRS`	`r1, cpsr`		`MRS`	`r1, cpsr`
		`BIC`	`r1, r1, #0x80`		`BIC`	`r1, r1, #0x40`
		`MSR`	`cpsr_c, r1`		`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`			`disable_fiq`		
		`MRS`	`r1, cpsr`		`MRS`	`r1, cpsr`
		`ORR`	`r1, r1, #0x80`		`ORR`	`r1, r1, #0x40`
		`MSR`	`cpsr_c, r1`		`MSR`	`cpsr_c, r1`
Post	*nzcvqj**I**ft_SVC*			*nzcvqji**F**t_SVC*		

- Interrupt stack

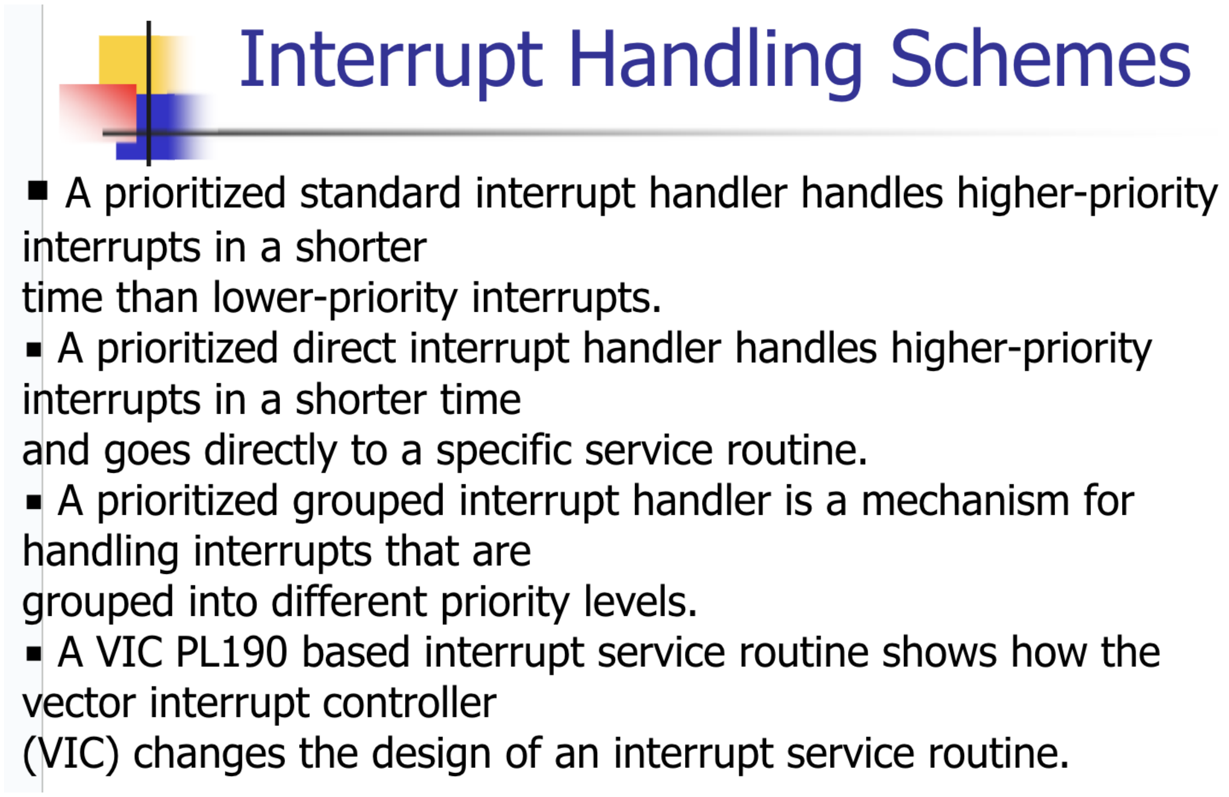
Stacks are needed extensively for context switching between different modes when interrupts are raised.

The design of the exception stack depends on two factors:
•OS Requirements.
•Target hardware.

A good stack design tries to avoid stack overflow because it cause instability in embedded systems.

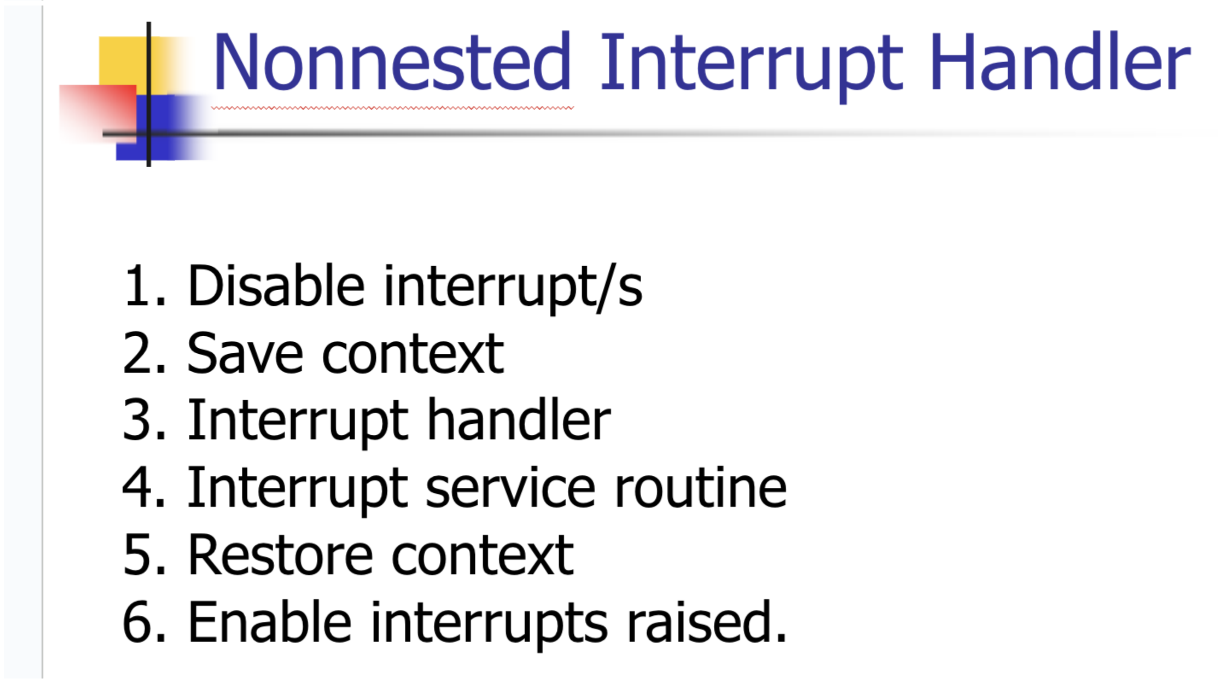
Interrupt Handling Schemes

- A nonnested interrupt handler handles and services individual interrupts sequentially. It
is the simplest interrupt handler.
- A nested interrupt handler handles multiple interrupts without a priority assignment.
- A reentrant interrupt handler handles multiple interrupts that can be prioritized.
- A prioritized simple interrupt handler handles prioritized interrupts.



Interrupt Handling Schemes

- A prioritized standard interrupt handler handles higher-priority interrupts in a shorter
time than lower-priority interrupts.
- A prioritized direct interrupt handler handles higher-priority interrupts in a shorter time
and goes directly to a specific service routine.
- A prioritized grouped interrupt handler is a mechanism for handling interrupts that are
grouped into different priority levels.
- A VIC PL190 based interrupt service routine shows how the vector interrupt controller
(VIC) changes the design of an interrupt service routine.



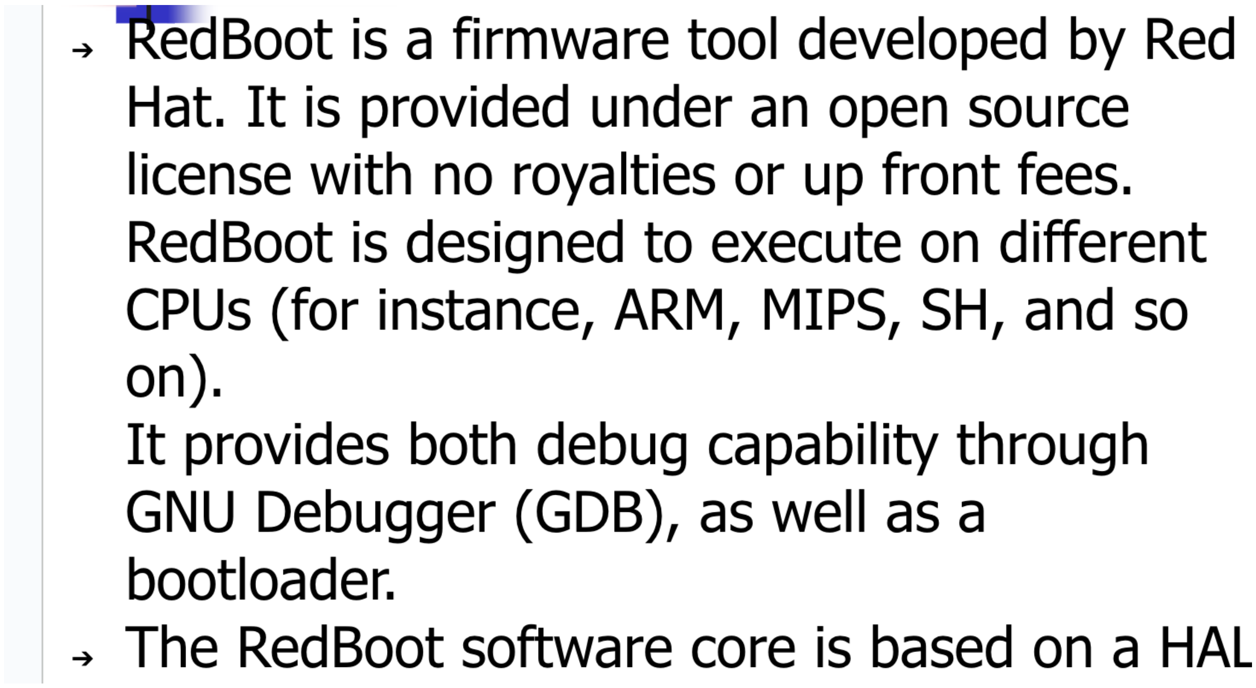
Nonnested Interrupt Handler

1. Disable interrupt/s
2. Save context
3. Interrupt handler
4. Interrupt service routine
5. Restore context
6. Enable interrupts raised.

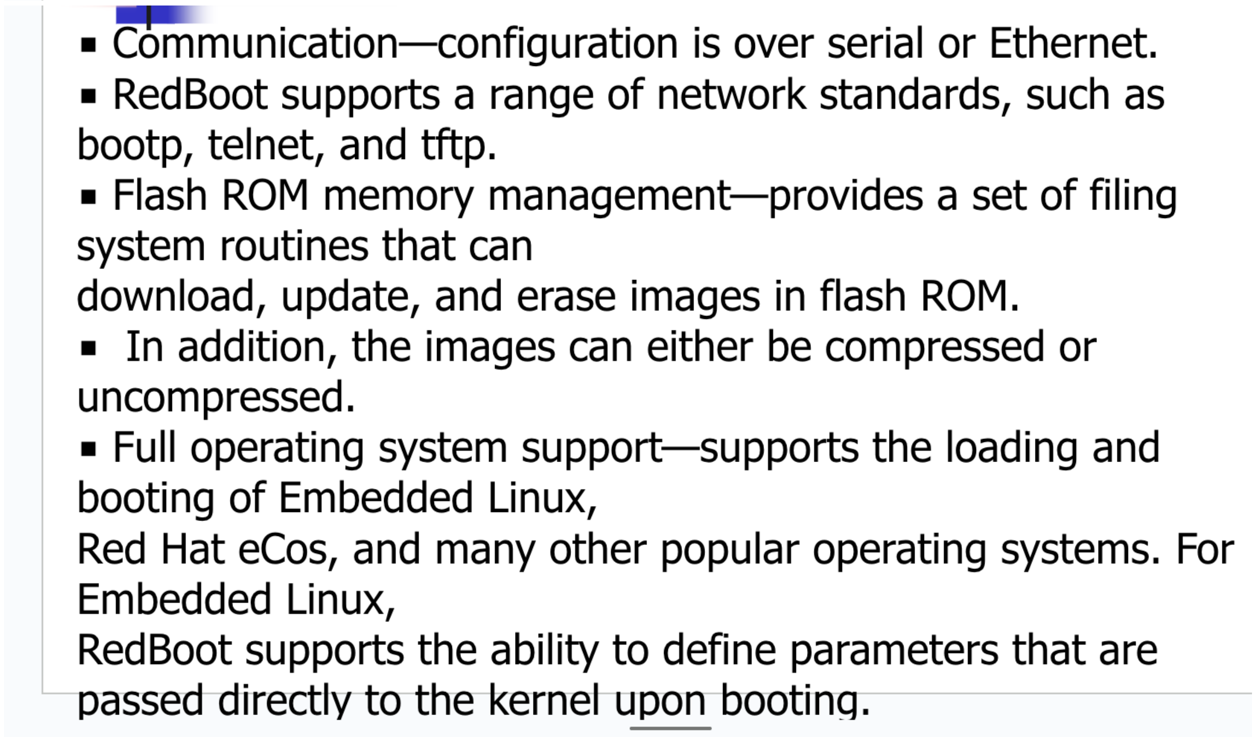
Q.8.a

- The firmware is the deeply embedded, low-level software that provides an interface between the hardware and the application/operating system level software.
- It resides in the ROM and executes when power is applied to the embedded hardware system.
- Firmware can remain active after system initialization and supports basic system operations.
- The choice of which firmware to use for a particular ARM-based system depends upon the specific application, which can range from loading and executing a sophisticated operating system to simply relinquishing control to a small microkernel.

- The bootloader is a small application that installs the operating system or application onto a hardware target.
- The bootloader only exists up to the point that the operating system or application is executing, and it is commonly incorporated into the firmware.



- RedBoot 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 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



▪ Communication—configuration is over serial or Ethernet.
▪ 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.

Q.8.b

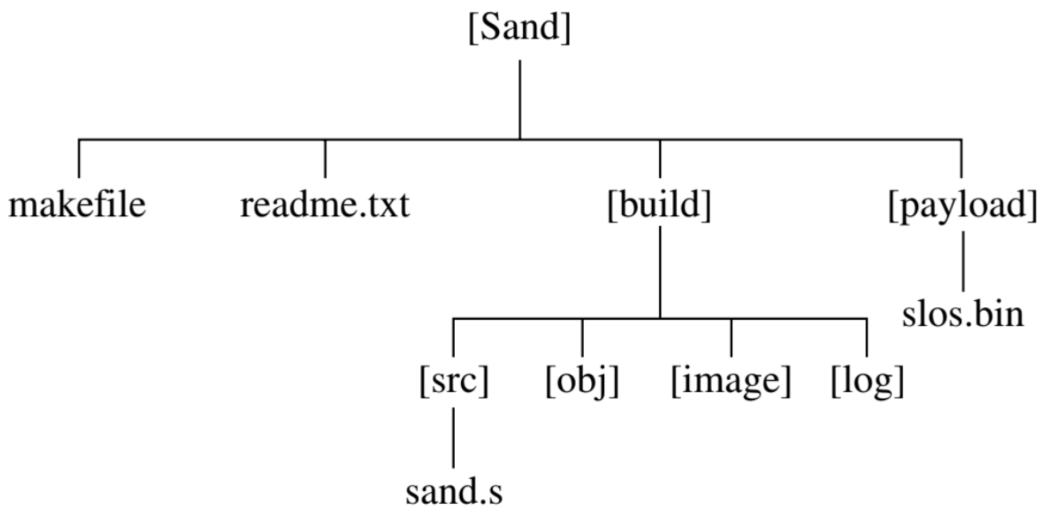
Sandstone Directory Layout

Sandstone can be found on our Website.
The structure follows a standard style.

Summary of Sandstone.

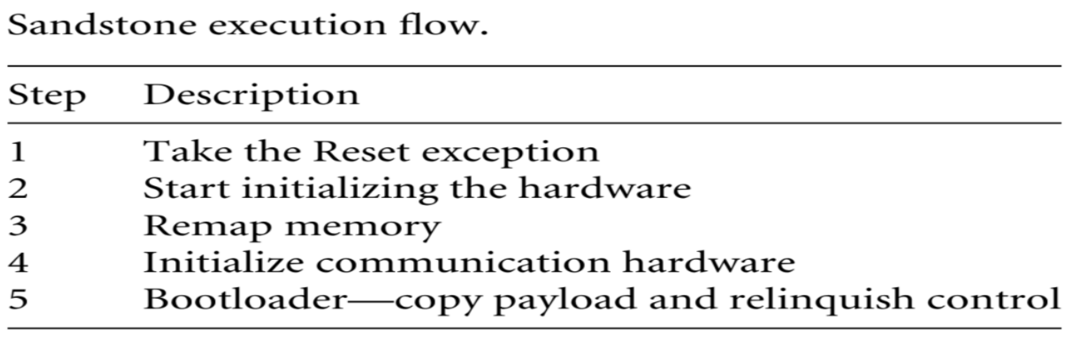
Feature	Configuration
Code	ARM instructions only
Tool chain	ARM Developer Suite 1.2
Image size	700 bytes
Source	17 KB
Memory	remapped

Standstone directory layout



Sandstone Code Structure

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



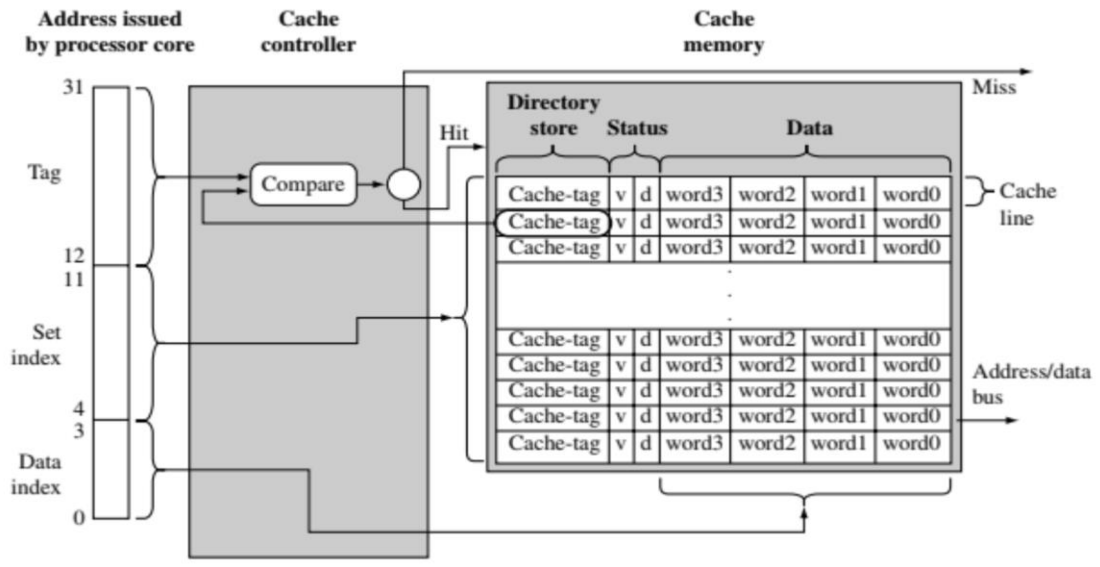
Sandstone execution flow.

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

Q.9.a

Q.9.a

Cache Architecture



Cache Controller

- ➔ The cache controller is hardware that copies code or data from main memory to cache memory automatically.
- ➔ It performs this task automatically to conceal cache operation from the software it supports.
- ➔ The same application software can run unaltered on systems with and without a cache.
- ➔ The cache controller intercepts read and write memory requests before passing them on to the memory controller. It processes a request by dividing the address of the request into three fields, the tag field, the set index field, and the data index field.
- ➔ The controller uses the set index portion of the address to locate the cache line within the cache memory that might hold the requested code or data. This cache line contains the cache-tag and 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, and compares the cache-tag to the tag field of the requested address.
- ➔ If both the status check and 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 and provides the requested code or data to the processor.
- ➔ The copying of a cache line from main memory to cache memory is known as a cache line fill.
- ➔ On a cache hit, the controller supplies the code or data directly from cache memory to the processor.
- ➔ To do this it moves to the next step, which is to use the data index field of the address request to select the actual code or data in the cache line and provide it to the processor.

Q.9.b

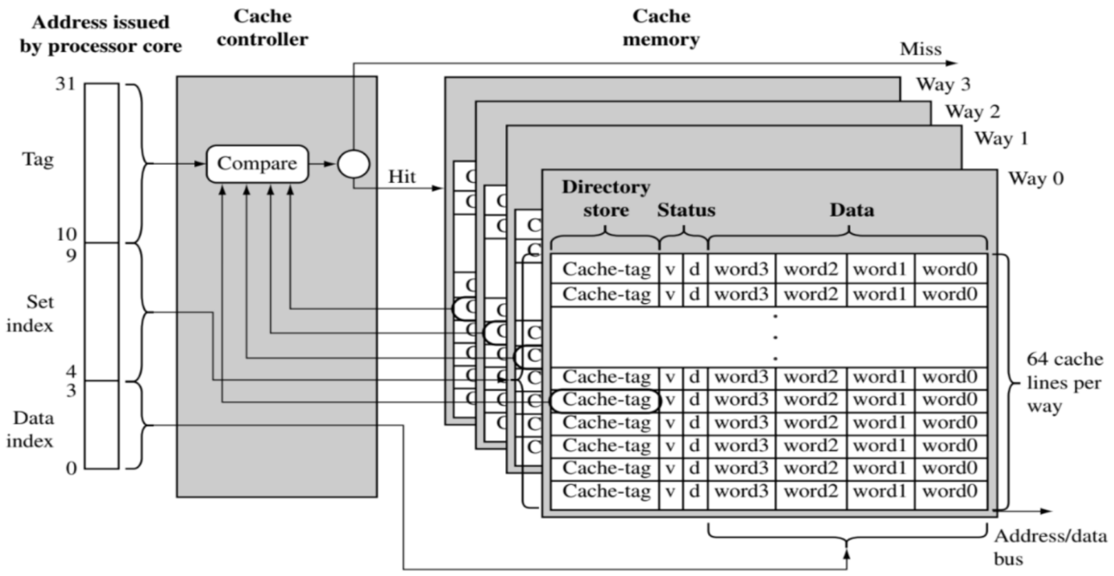


Figure 12.7 A 4 KB, four-way set associative cache. The cache has 256 total cache lines, which are separated into four ways, each containing 64 cache lines. The cache line contains four words.

Set Associativity

- ❖ A data or code block from main memory can be allocated to any of the four ways in a set without affecting program behavior; in other words the storing of data in cache lines within a set does not affect program execution.
- ❖ Two sequential blocks from main memory can be stored as cache lines in the same way or two different ways.
- ❖ The important thing to note is that the data or code blocks from a specific location in main memory can be stored in any cache line that is a member of a set.
- ❖ The placement of values within a set is exclusive to prevent the same code or data block from simultaneously occupying two cache lines in a set.

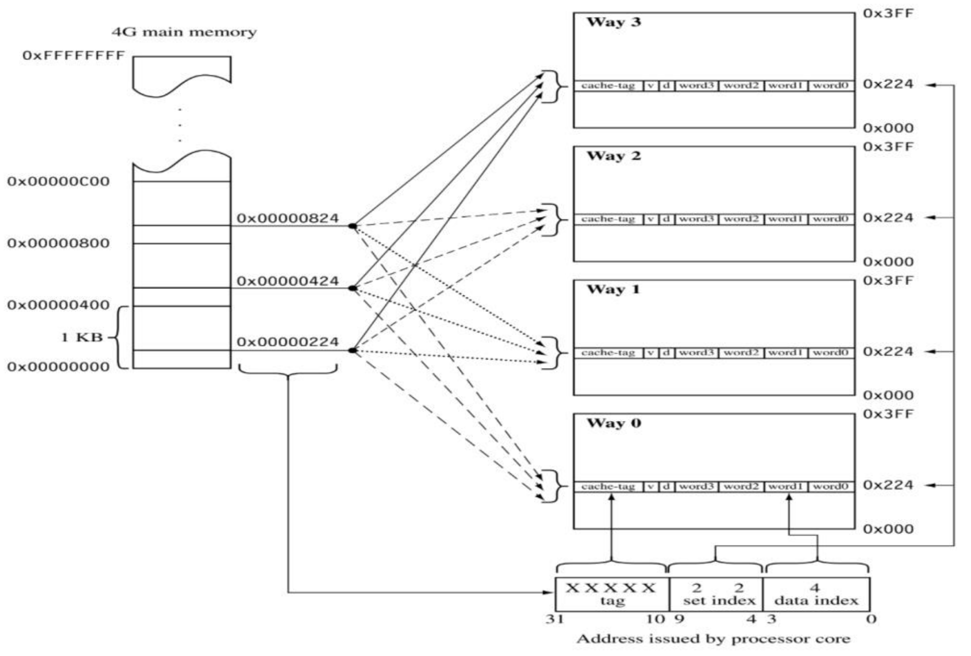


Figure 12.8 Main memory mapping to a four-way set associative cache.

- The bit field for the tag is now two bits larger, and the set index bit field is two bits smaller.
- This means four million main memory addresses now map to one set of four cache lines, instead of one million addresses mapping to one location.
- The size of the area of main memory that maps to cache is now 1 KB instead of 4 KB.
- This means that the likelihood of mapping cache line data blocks to the same set is now four times higher. This is offset by the fact that a cache line is one fourth less likely to be evicted.
- The incidence of thrashing would quickly settle down as routine A, routine B, and the data array would establish unique places in the four available locations in a set.

Q.10.a

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 system without a write buffer, the processor writes directly to main memory.
- In a system with a write buffer, data is written at high speed to the FIFO and then emptied to slower main memory.
- The write buffer reduces the processor time taken to write small blocks of sequential data to main memory.
- The FIFO memory of the write buffer is at the same level in the memory hierarchy as the L1 cache.

- The efficiency of the write buffer depends on the ratio of main memory writes to the number of instructions executed.
- Over a given time interval, if the number of writes to main memory is low or sufficiently spaced between other processing instructions, the write buffer will rarely fill.
- If the write buffer does not fill, the running program continues to execute out of cache memory using registers for processing, cache memory for reads and writes, and the write buffer for holding evicted cache lines while they drain to main memory.

- The efficiency of the write buffer depends on the ratio of main memory writes to the number of instructions executed.
- Over a given time interval, if the number of writes to main memory is low or sufficiently spaced between other processing instructions, the write buffer will rarely fill.
- If the write buffer does not fill, the running program continues to execute out of cache memory using registers for processing, cache memory for reads and writes, and the write buffer for holding evicted cache lines while they drain to main memory.

- ➢ A write buffer also improves cache performance.
- ➢ The improvement occurs during cache line evictions.
- ➢ If the cache controller evicts a dirty cache line, it writes the cache line to the write buffer instead of main memory.
- ➢ The new cache line data will be available sooner, and the processor can continue operating from cache memory.
- ➢ 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. The ARM10 family, supports coalescing—the merging of write operations into a single cache line.
- ➢ The write buffer will merge the new value into an existing cache line in the write buffer if they represent the same data block in main memory. Coalescing is also known as write merging, write collapsing, or write combining.

Measuring Cache Efficiency

There are two terms used the cache hit rate and the cache miss rate.

The hit rate is the number of cache hits divided by the total number of memory requests over a given time interval.

The value is expressed as a percentage:

$$\textit{hit rate} = \left(\frac{\textit{cache hits}}{\textit{memory requests}}\right) \times 100$$

- ★ The hit rate and 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 main memory into cache.

Q.10.b

Cache Policy

- There are three policies that determine the operation of a cache: the write policy, the replacement policy, and the allocation policy.
- The cache write policy determines where data is stored during processor write operations.
- The replacement policy selects the cache line in a set that is used for the next line fill during a cache miss.
- The allocation policy determines when the cache controller allocates a cache line.

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 the cache and main memory, updating the values in both locations; this approach is known as writethrough.
- The cache controller can write to cache memory and not update main memory, this is known as writeback or copyback.

Writethrough

- When the cache controller uses a writethrough policy, it writes to both cache and main memory when there is a cache hit on write, ensuring that the cache and main memory stay coherent at all times.
- Under this policy, the cache controller performs a write to main memory for each write to cache memory.
- Because of the write to main memory,a writethrough policy is slower than a writeback policy.

Writeback

- When a cache controller uses a writeback policy, it writes to valid cache data memory and not to main memory.
- Valid cache lines and main memory may contain different data.
- The cache line holds the most recent data, and main memory contains older data, which has not been updated.

- Caches configured as writeback caches must use one or more of the dirty bits in the cache line status information block.
- When a cache controller in writeback writes a value to cache memory, it sets the dirty bit true.
- If the core accesses the cache line at a later time, it knows by the state of the dirty bit that the cache line contains data not in main memory.
- If the cache controller evicts a dirty cache line, it is automatically written out to main memory.
- The controller does this to prevent the loss of vital information held in cache memory and not in main memory.

- One performance advantage a writeback cache has over a writethrough cache is in the frequent use of temporary local variables by a subroutine.
- These variables are transient in nature and never really need to be written to main memory.
- An example of one of these transient variables is a local variable that overflows onto a cached stack because there are not enough registers in the register file to hold the variable.

Cache Line Replacement Policies

- 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 the 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 victim cache line.
- The process of selecting and replacing a victim cache line is known a eviction.

- The strategy implemented in a cache controller to select the next victim is called its **replacement policy**.
- The replacement policy selects a cache line from the available associative member set; that is, it selects the way to use in the next cache line replacement.
- To summarize the overall process, the set index selects the set of cache lines available in the ways, and the replacement policy selects the specific cache line from the set to replace.

Replacement Policies

- ➢ ARM cached cores support two replacement policies, either pseudorandom or round-robin.
- ➢ Most ARM cores support both policies
- ➢ The round-robin replacement policy has greater predictability, which is desirable in an embedded system.
- ➢ A round-robin replacement policy is subject to large changes in performance given small changes in memory access.

Round-robin or cyclic replacement:

- ➢ Simply selects the next cache line in a set to replace.
- ➢ The selection algorithm uses a sequential, incrementing victim counter that increments each time the cache controller allocates a cache line.
- ➢ When the victim counter reaches a maximum value, it is reset to a defined base value.

Pseudorandom replacement policy:

- ➢ Randomly selects the next cache line in a set to replace.
- ➢ The selection algorithm uses a nonsequential incrementing victim counter.
- ➢ In a pseudorandom replacement algorithm the controller increments the victim counter by randomly selecting an increment value and adding this value to the victim counter.
- ➢ When the victim counter reaches a maximum value, it is reset to a defined base value.