CBCS SCHEME

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

BCS402

Fourth Semester B.E./B.Tech. Degree Examination, June/July 2024
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.

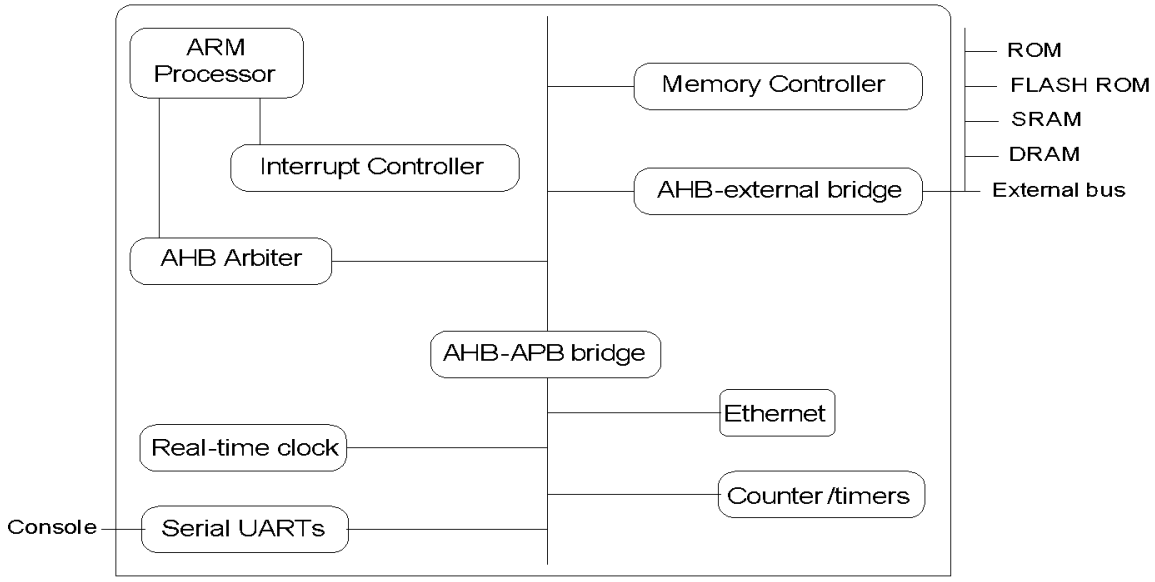
			M	L	C
		Module – 1			
Q.1	a.	Explain the architecture of an arm embedded device with a neat diagram.	10	L2	CO1
	b.	How are monitor and control internal operations performed in ARM core? Explain in brief.	10	L2	CO1
		OR			
Q.2	a.	Explain memory management in ARM core. Compare cache and tightly coupled memory.	10	L2	CO1
	b.	Explain mechanism applied by ARM core to handle exception, interrupts using different vector table.	10	L2	CO1
		Module – 2			
Q.3	a.	Examine data processing instructions requirement in the manipulation of data register? Explain in brief data processing instructions.	10	L2	CO2
	b.	Explain with examples the following 32-bit instruction of ARN processor i) CMN ii) MLA iii) MRS iv) BIC v) LDR.	10	L2	CO2
		OR			
Q.4	a.	Explain the following with example : i) Stock operation ii) Swap instructions.	10	L2	CO2
	b.	Explain Branch instructions in ARM with suitable example. Demonstrate Branch instruct usage flow of execution with an example program.	10	L2	CO2
		Module – 3			
Q.5	a.	How registers are allocated to optimize the program? Develop an assembly level program to find the sum of first to integer numbers.	10	L2	CO3
	b.	How complier handles a "for loop" with variable number of iterations N and loop controlling with an example.	10	L2	CO3
		OR			
Q.6	a.	Explain the following terms with an appropriate example : i) Pointer Aliasing ii) Portability issues.	10	L2	CO3
	b.	How function calling is efficiently used by ARM through APCS with an example program.	10	L2	CO3
		Module – 4			
Q.7	a.	Explain ARM processors exception and modes with a neat diagram.	10	L2	CO4
	b.	Explain exception priorities and link register offset.	10	L2	CO4
		OR			
Q.8	a.	List ARM firmware suite features. Explain firmware execution flow and Red Hat Boot.	10	L2	CO4
	b.	Explain IRQ and Fir exception, also to enable and disable IRQ and FIQ interrupts.	10	L2	CO4
		Module – 5			
Q.9	a.	Explain basic architecture of cache memory.	10	L2	CO5
	b.	Explain process involved in main memory mapping to a cache memory.	10	L2	CO5
		OR			
Q.10	a.	Explain with diagram set associative cache. How are efficiency is measured?	10	L2	CO5
	b.	Briefly explain cache line replacement policies with an example.	10	L2	CO5

* * * * *

Microcontroller VTU solution

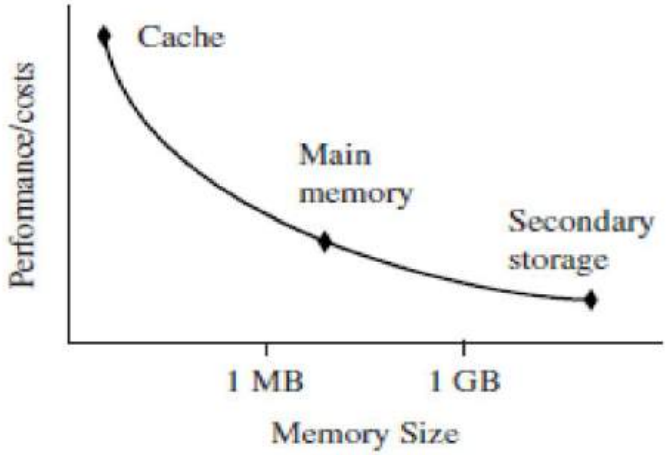
1.a

Figure shown below shows a typical embedded device based on ARM core. Each box represents a feature or function.



- ARM processor based embedded system hardware can be separated into the following four main hardware components:
  - **The ARM processor:** The ARM processor controls the embedded device. Different versions of the ARM processor are available to suit the desired operating characteristics.
  - **Controllers:** Controllers coordinate important blocks of the system. Two commonly found controllers are memory controller and interrupt controller.
  - **Peripherals:** The peripherals provide all the input-output capability external to the chip and responsible for the uniqueness of the embedded device.
  - **Bus:** A bus is used to communicate between different parts of the device.
- **ARM Bus Technology**
  - Embedded devices use an on-chip bus that is internal to the chip and that allows different peripheral devices to be interconnected with an ARM core.
  - There are two different classes of devices attached to the bus.
    - The ARM processor core is a **bus master**—a logical device capable of initiating a data transfer with another device across the same bus.
    - Peripherals tend to be **bus slaves**—logical devices capable only of responding to a transfer request from a bus master device.
- **AMBA Bus Protocol**
  - The Advanced Microcontroller Bus Architecture (AMBA) was introduced in 1996 and has been widely adopted as the on-chip bus architecture used for ARM processors.
  - The first AMBA buses introduced were the ARM System Bus (ASB) and the ARM Peripheral Bus (APB).
  - Later ARM introduced another bus design, called the ARM High Performance Bus (AHB).
  - AHB provides higher data throughput than ASB because it is based on a centralized multiplexed bus scheme rather than the ASB bidirectional bus design.
- **MEMORY**
  - An embedded system has to have some form of memory to store and execute code.
  - Figure below shows the memory trade-offs: the fastest memory cache is physically located nearer the ARM processor core and the slowest secondary memory is set further away.

- Generally the closer memory is to the processor core, the more it costs and the smaller its capacity.



- **PERIPHERALS**
  - Embedded systems that interact with the outside world need some form of peripheral device.
  - Controllers are specialized peripherals that implement higher levels of functionality within the embedded system.
  - **Memory controller:** Memory controllers connect different types of memory to the processor bus.
  - **Interrupt controller:** An interrupt controller provides a programmable governing policy that allows software to determine which peripheral or device can interrupt the processor at any specific time.

1.b

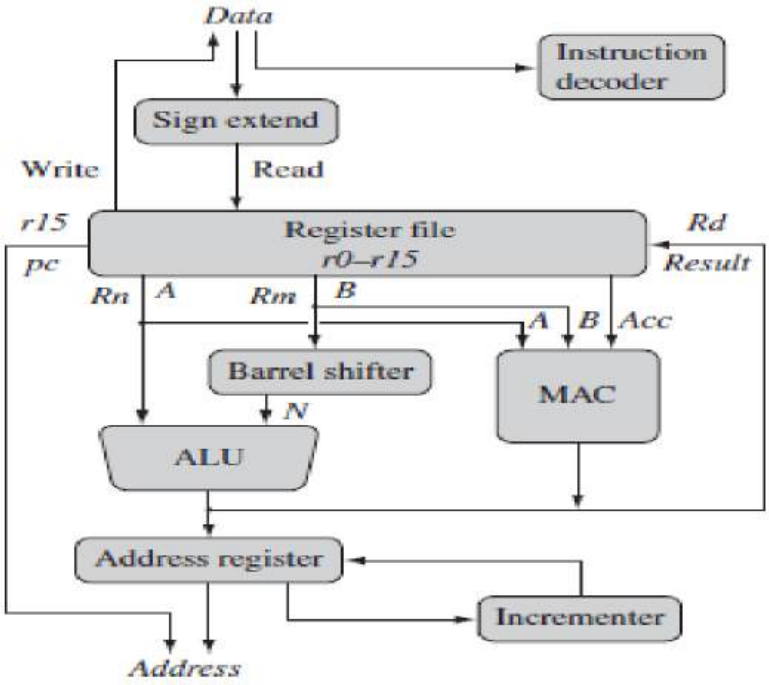


Figure1: ARM core dataflow model

- An ARM core as functional units connected by data buses, as shown in Figure1, where, the arrows represent the flow of data, the lines represent the buses, and the boxes represent either an operation unit or a storage area.
- The instruction decoder translates instructions before they are executed.

- The ARM processor, like all RISC processors, uses a load - store architecture.
- **Load instructions** copy data from memory to registers, and conversely the **store instructions** copy data from registers to memory.
- There are no data processing instructions that directly manipulate data in memory.
- ARM instructions typically have two source registers, Rn and Rm, and a single 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.
- 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.

2.a

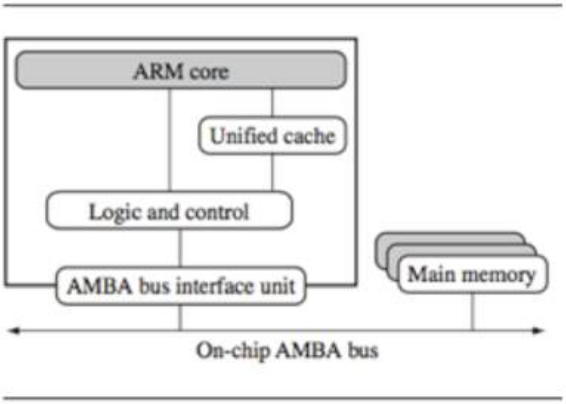
Memory management:

- Embedded systems often use multiple memory devices. It is usually necessary to have a method to help organize these devices and protect the system from applications trying to make appropriate accesses to hardware.
- This is achieved with the assistance of memory management hardware.
- ARM cores have three different types of memory management hardware- no extensions provide no protection, a memory protection unit (MPU) providing limited protection and a memory management unit (MMU) providing full protection.
    - **Nonprotected memory** is fixed and provides very little flexibility. It normally used for small, simple embedded systems that require no protection from rogue applications.
    - **Memory protection unit (MPU)** employs a simple system that uses a limited number of memory regions. These regions are controlled with a set of special coprocessor registers, and each region is defined with specific access permission but don't have a complex memory map.
    - **Memory management unit (MMU)**are the most comprehensive memory management hardware available on the ARM. The MMU uses a set of translation tables to provide fine-grained control over memory.
        - These tables are stored in main memory and provide virtual to physical address map as well as access permission. MMU designed for more sophisticated system that supports multitasking.

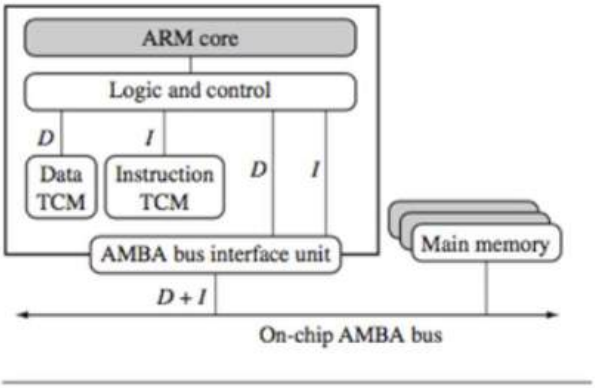
A. Cache and Tightly Coupled Memory

The cache is a block of fast memory placed between main memory and 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.

ARM has **two forms of cache**. The first is found attached to the Von Neumann–style cores. It combines both data and instruction into a single unified cache, as shown in Figure. For simplicity, we have called the glue logic that connects the memory system to the AMBA bus *logic and control.*



A simplified Von Neumann architecture with cache.

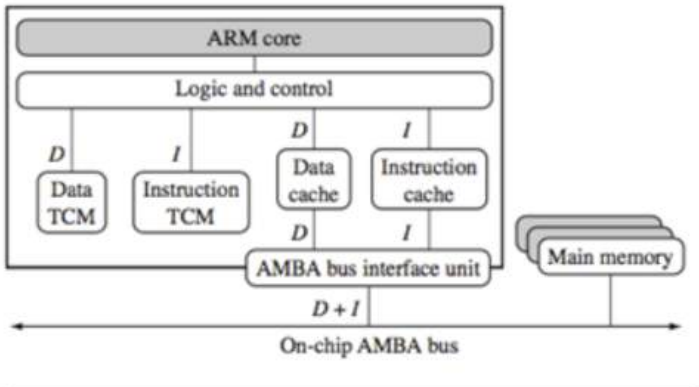


A simplified Harvard architecture with TCMs.

By contrast, the second form, attached to the Harvard-style cores, has separate caches for data and instruction.

A cache provides an overall increase in performance but at the expense of predictable execution. But for real-time systems it is paramount that code execution is *deterministic*—the time taken for loading and storing instructions 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 required to fetch instructions or data—critical for real-time algorithms requiring deterministic behavior. TCMs appear as memory in the address map and can be accessed as fast memory. An example of a processor with TCMs is shown in Figure.

By combining both technologies, ARM processors can have both improved performance and predictable real-time response. Figure shows an example core with a combination of caches and TCMs.



A simplified Harvard architecture with caches and TCMs.

2.b

When an exception or interrupt occurs, the processor sets the *pc* to a specific memory address. The address is within a special address range called the *vector table*. The entries in the vector table are instructions that branch to specific routines designed to handle a particular exception or interrupt.

The memory map address 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 address in memory (starting at the offset 0xffff0000).

When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the exception vector table (see Table 2.6). Each vector table entry contains a form of branch instruction pointing to the start of a specific routine:

■ *Reset vector* is the location of the first instruction executed by the processor when power is applied. This instruction branches to the initialization code.

■ *Undefined instruction vector* is used when the processor cannot decode an instruction.

■ *Software interrupt vector* is called when you execute a SWI instruction. The SWI instruction is frequently used as the mechanism to invoke an operating system routine.

■ *Prefetch abort vector* occurs when the processor attempts to fetch an instruction from an address without the correct access permissions. The actual abort occurs in the decode stage.

■ *Data abort vector* is similar to a prefetch abort but is raised when an instruction attempts to access data memory without the correct access permissions.

■ *Interrupt request vector* is used by external hardware to interrupt the normal execution flow of the processor. It can only be raised if IRQs are not masked in the *cpsr*.

■ *Fast interrupt request vector* is similar to the interrupt request but is reserved for hardware requiring faster response times. It can only be raised if FIQs are not masked in the *cpsr*.

The vector table.

Exception/interrupt	Shorthand	Address	High address
Reset	RESET	0x00000000	0xffff0000
Undefined instruction	UNDEF	0x00000004	0xffff0004
Software interrupt	SWI	0x00000008	0xffff0008
Prefetch abort	PABT	0x0000000c	0xffff000c
Data abort	DABT	0x00000010	0xffff0010
Reserved	—	0x00000014	0xffff0014
Interrupt request	IRQ	0x00000018	0xffff0018
Fast interrupt request	FIQ	0x0000001c	0xffff001c

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.

MOVE INSTRUCTIONS:

- It copies N into a destination register Rd, where N is a register or immediate value. This instruction is useful for setting initial values and transferring data between registers.

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	$Rd = \sim N$

- In the example shown below, the MOV instruction takes the contents of register r5 and copies them into register r7.

```
PRE     r5 = 5
        r7 = 8
        MOV     r7, r5     ; let r7 = r5
POST    r5 = 5
        r7 = 5
```

3.b

i)

- **CMN – Compare with Negated Value**

  `CMN R0, R1  ; R0 + R1`

- It performs R0 + R1, but does not stores result of it.
- It is just affecting flag for comparison, here 'S' is not required to change the status.

ii)

– MLA{<cond>}{S} Rd,Rm,Rs,Rn

MLA r4,r3,r2,r1; $r4:=((r3*r2)+r1)_{[31:0]}$

iii)

Syntax: MRS{<cond>} Rd,<cpsr|spsr>

MRS	copy program status register to a general-purpose register	$Rd = psr$

iv)

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

BIC	logical bit clear (AND NOT)	$Rd = Rn \& \sim N$

- **BIC – AND with Compliment**

  `BIC R0, R1, R2  ;R0 ← R1 AND R2`

- It will load R0 with logical AND operation of R1 and $\overline{R2}$.
- It will not update flag, to get result with updated flag use BICS.

v)

- For word and unsigned byte accesses, offset can be:
  - An unsigned 12-bit immediate value (i.e. 0 - 4095 bytes)
    ```
    LDR r0, [r1, #8]
    ```
  - A register, optionally shifted by an immediate value
    ```
    LDR r0, [r1, r2]
    LDR r0, [r1, r2, LSL#2]
    ```
- This can be either added or subtracted from the base register:
    ```
    LDR r0, [r1, #-8]
    LDR r0, [r1, -r2, LSL#2]
    ```
- For halfword and signed halfword / byte, offset can be:
  - An unsigned 8 bit immediate value (i.e. 0 - 255 bytes)
  - A register (unshifted)
- Choice of *pre-indexed* or *post-indexed* addressing
- Choice of whether to update the base pointer (pre-indexed only)
    ```
    LDR r0, [r1, #-8]!
    ```

4.a
i)

Stack Operation

- A stack is an example of a data structure, Stacks typically used for temporary storage of data.
- Operations on Stack :

  Push: Place cards on the top of the stack

  Pop: Remove cards from the top of the stack

Stack Operation

- To implement a stack data structure we need
  - An area of memory to store the data items
  - A Stack Pointer (SP) register to point to the top of the stack
  - stack growth convention
  - Some well defined operations: initialize, push, pop

Stack Operation

- An **Ascending stack** grows upwards. It starts from a low memory address and, as items are pushed onto it, progresses to higher memory addresses.
- A **Descending stack** grows downwards. It starts from a high memory address, and as items are pushed onto it, progresses to lower memory addresses.
- In an **Empty stack**, the stack pointers points to the next free (empty) location on the stack, i.e. the place where the next item to be pushed onto the stack will be stored.
- In a **Full stack**, the stack pointer points to the topmost item in the stack, i.e. the location of the last item to be pushed onto the stack.

Stack in ARM

- ARM does not support any hardware stack.
- Software stack can be implemented using the LDM and STM family of instructions.
- The load and store multiple instructions can update the base register.
- For stack operations, the base register is usually the stack pointer, sp.
- This means that we can use these instructions to implement push and pop operations for any number of registers in a single instruction.

ii)

SWAP INSTRUCTION contd...

- **Syntax:**

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

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

SWAP INSTRUCTION contd...

- **Example1:** ***Load a word from memory into register R0 and overwrite the memory with register R1.***

  PRE

  mem32[0x9000] = 0x12345678

  r0 = 0x00000000

  r1 = 0x11112222

  r2 = 0x00009000

  SWP r0, r1, [r2]

  POST

  mem32[0x9000] = **0x11112222**

  r0 = 0x12345678

  r1 = 0x11112222

  r2 = 0x00009000

4.b

Control Flow Instructions

- These instructions change the order of instruction execution or to jump from one memory location to other.
- Types of conditional flow instructions:
  - Unconditional branch
  - Conditional branch
  - Branch and Link

Branching Instructions

- In ARM there are four important branch instructions available,
  - **B** **Branch**
  - **BL** **Branch with link**
  - **BX** **Branch Exchange**
  - **BLX** **Branch Exchange with link**

Branching Instructions - B and BL

- Instruction Format:
  - Branch: **B{<cond>} Label**
  - Branch with Link: **BL{<cond>} subroutine_label**
- **B** – Branch

  PC = <address> or <label>
- **BL** – Branch with Link

  R14 = address of next instruction, PC = <address>
- Thus to return from a linked branch

  MOV r15,r14 or

  MOV pc,lr

- **B Instruction :**

```
Back : MOV r0,r1
       ADD r0,r1,r2
       SUB r3,r0,r1
       B Back
```

- **BL Instruction**

```
       CMP r2,#10
       BL Next
       ADDEQ r2,r3,r4
Next:  AND r4,r5,r6
       MOV PC,lr
```

Branch Examples

- Unconditional jump

```
    B    LABEL
         ...
LABEL    ...
```

- Conditional jump

```
         MOV r0,#10
Loop     ...
         SUBS r0,#1
         BNE Loop
```

- Call a subroutine

```
         BL  SUB
         ...
SUB      ...
         MOV PC,r14
```

Call a Conditional subroutine

```
CMP r0,#5
BLLT SUB1     ;if r0<5,
              ;call sub1
BLGE SUB2     ;else call sub2
```

5.a

- The compiler attempts to allocate a processor register to each local variable you use in a C function.
- It will try to use the same register for different local variables if the use of the variables do not overlap. When there are more local variables than available registers, the compiler stores the excess variables on the processor stack. These variables are called spilled
- or swapped out variables since they are written out to memory (in a similar way virtual memory is swapped out to disk).
- Spilled variables are slow to access compared to variables allocated to registers.

To implement a function efficiently, you need to

- minimize the number of spilled variables
- ensure that the most important and frequently accessed variables are stored in registers

Register number	Alternate register names	ATPCS register usage
r0	*a1*	Argument registers. These hold the first four function arguments on a function call and the return value on a function return. A function may corrupt these registers and use them as general scratch registers within the function.
r1	*a2*	
r2	*a3*	
r3	*a4*	
r4	*v1*	General variable registers. The function must preserve the callee values of these registers.
r5	*v2*	
r6	*v3*	
r7	*v4*	
r8	*v5*	
r9	*v6 sb*	General variable register. The function must preserve the callee value of this register except when compiling for *read-write position independence* (RWPI). Then *r9* holds the *static base* address. This is the address of the read-write data.
r10	*v7 sl*	General variable register. The function must preserve the callee value of this register except when compiling with stack limit checking. Then *r10* holds the stack limit address.
r11	*v8 fp*	General variable register. The function must preserve the callee value of this register except when compiling using a frame pointer. Only old versions of *armcc* use a frame pointer.
r12	*ip*	A general scratch register that the function can corrupt. It is useful as a scratch register for function veneers or other intraprocedure call requirements.
r13	*sp*	The stack pointer, pointing to the full descending stack.
r14	*lr*	The link register. On a function call this holds the return address.
r15	*pc*	The program counter.

```
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

5.b

Loop unrolling

only unroll loops that are important for the overall performance of the application. Otherwise unrolling will increase the code size with little performance benefit. Unrolling may even reduce performance by evicting more important code from the cache.

- What if the number of loop iterations is not a multiple of the unroll amount? For example,

what if N is not a multiple of four in checksum_v9?

try to arrange it so that array sizes are multiples of your unroll amount. If this isn't possible, then you must add extra code to take care of the leftover cases.

This increases the code size a little but keeps the performance high.

```
for (i=N&3; i!=0; i--)
{
sum += *(data++);
}
return sum;
}
```

The second for loop handles the remaining cases when N is not a multiple of four. Note

that both N/4 and N&3 can be zero, so we can't use do-while loops.

- Use loops that count down to zero. Then the compiler does not need to allocate a register to hold the termination value, and the comparison with zero is free.

- Use unsigned loop counters by default and the continuation condition i!=0 rather than i>0. This will ensure that the loop overhead is only two instructions.

6.a

i) 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.

Note that the compiler loads from step twice.

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

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

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

In this case the second value of *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;
}
```

ii)

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 $i \geq 0$, 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.

6.b

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.

Calling Functions Efficiently

- Try to restrict functions to four arguments. This will make them more efficient to call. Use structures to group related arguments and pass structure pointers instead of multiple arguments.
- Define small functions in the same source file and before the functions that call them.

The compiler can then optimize the function call or inline the small function.

- Critical functions can be inlined using the __inline keyword.

```
typedef struct {
  char *Q_start;          /* Queue buffer start address */
  char *Q_end;            /* Queue buffer end address */
  char *Q_ptr;            /* Current queue pointer position */
} Queue;

void queue_bytes_v2(Queue *queue, char *data, unsigned int N)
{
  char *Q_ptr = queue->Q_ptr;
  char *Q_end = queue->Q_end;

  do
  {
    *(Q_ptr++) = *(data++);

    if (Q_ptr == Q_end)
    {
      Q_ptr = queue->Q_start;
    }
  } while (--N);
  queue->Q_ptr = Q_ptr;
}
```

7.a

Modes of operation

•ARM processor has 7 modes of operation.

•Switching between modes can be done manually through modifying the mode bits in the CPSR register.

•Most application programs execute in user mode

•Non user modes (called privileged modes) are entered to serve interrupts or exceptions

•The system mode is special mode for accessing protected resources. It don't use registers used by exception hanlders, so it can't be corrupted by any exception handler error!!!

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.

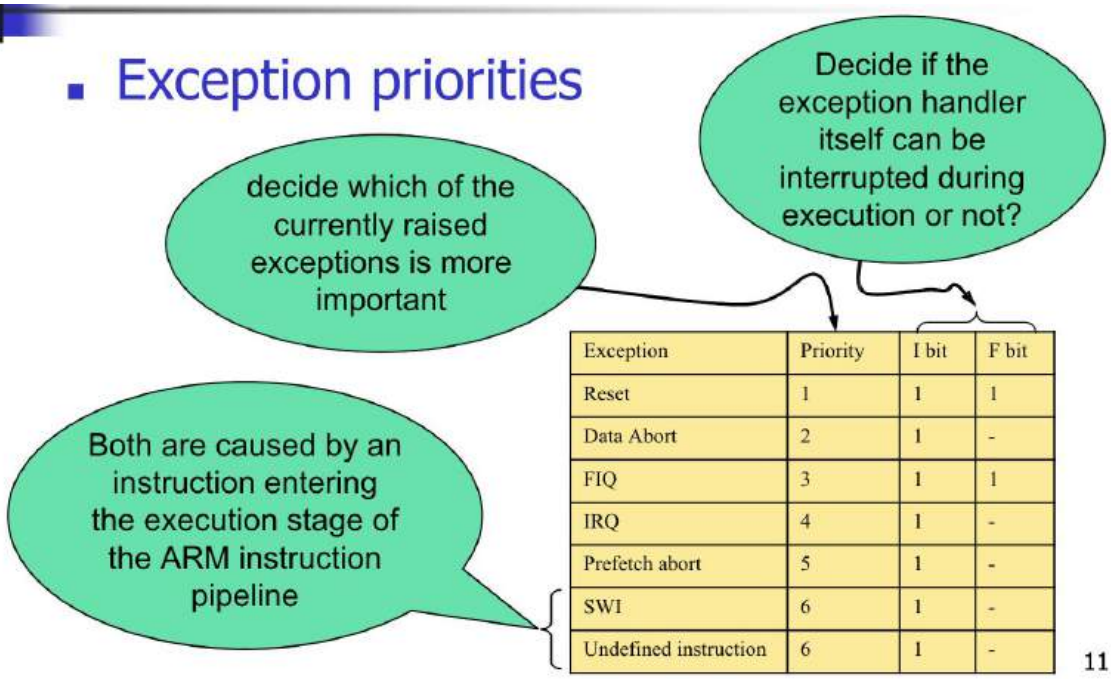
At this place in memory, we find a branching instruction

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

7.b

Exception priorities



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	-

11

Link Register Offset

This register is used to return the ***PC*** to the appropriate place in the interrupted task since this is not always the old ***PC*** value.It is modified depending on the type of exception.

The ***PC*** has advanced beyond the instruction causing the exception. Upon exit of the prefetch abort exception handler, software must re-load the PC back one instruction from the ***PC*** saved at the time of the exception.

Exception	Returning Address
Reset	None
Data Abort	LR-8
FIQ, IRQ, prefetch Abort	LR-4
SWI, Undefined Instruction	LR

18

Entering exception handler

1. Save the address of the next instruction in the appropriate Link Register ***LR***.
2. Copy ***CPSR*** to the ***SPSR*** of new mode.
3. Change the mode by modifying bits in ***CPSR***.
4. Fetch next instruction from the vector table.

Leaving exception handler

1. Move the Link Register ***LR*** (minus an offset) to the ***PC***.
2. Copy ***SPSR*** back to ***CPSR,*** this will automatically changes the mode back to the previous one.
3. Clear the interrupt disable flags (if they were set).

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.

Stage	Features
Set up target platform	Program the hardware system registers Platform identification Diagnostics Debug interface Command line interpreter
Abstract the hardware	Hardware Abstraction Layer Device driver
Load a bootable image	Basic filing system
Relinquish control	Alter the *pc* to point into the new image

- 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.

8.b

Enabling an IRQ/FIQ Interrupt:

```
MRS    r1, cpsr
BICr1, r1, #0x80/0x40
MSR    cpsr_c, r1
```

Disabling an IRQ/FIQ Interrupt:

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

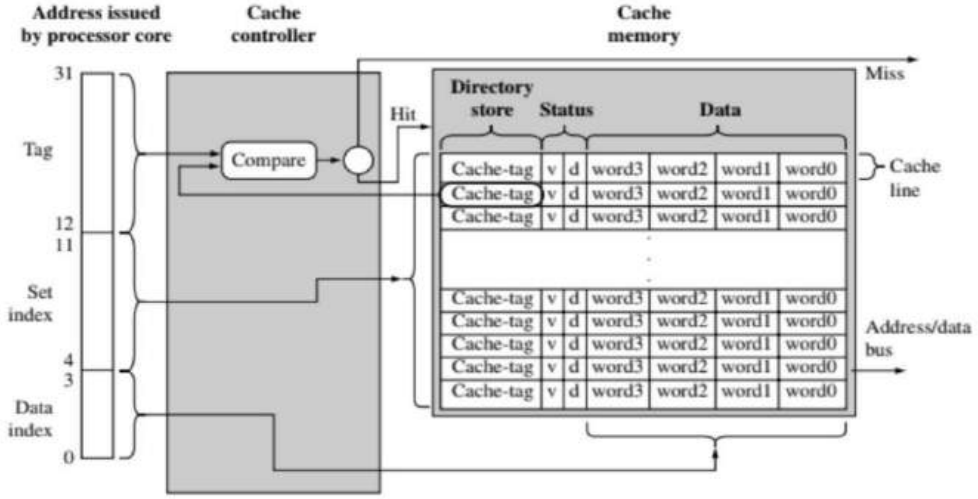
cpsr value	IRQ	FIQ
Pre	*nzcvqjIFt_SVC*	*nzcvqjIFt_SVC*

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

Post	*nzcvqjiFt_SVC*	*nzcvqjIft_SVC*

9.a

Cache Architecture



Cache Architecture

ARM uses two bus architectures in its cached cores, the Von Neumann and the Harvard.

The Von Neumann and Harvard bus architectures differ in the separation of the instruction and data paths between the core and memory.

A different cache design is used to support the two architectures.

Von Neumann architecture

In processor cores using the Von Neumann architecture, there is a single cache used for instruction and data.

This type of cache is known as a unified cache.

A unified cache memory contains both instruction and data values.

Harvard architecture

The Harvard architecture has separate instruction and data buses to improve overall system performance, but supporting the two buses requires two caches.

In processor cores using the Harvard architecture, there are two caches: an instruction cache (I-cache) and a data cache (D-cache).

This type of cache is known as a split cache.

In a split cache, instructions are stored in the instruction cache and data values are stored in the data cache.

Cache Architecture

The two main elements of a cache are the cache controller and the cache memory.

The cache memory is a dedicated memory array accessed in units called cache lines.

The cache controller uses different portions of the address issued by the processor during a memory request to select parts of cache memory.

Cache Architecture

- ➔ It has three main parts:a directory store, a data section, and statu information.
- ➔ All three parts of the cache memory are present for each cache line.
- ➔ The cache must know where the information stored in a cache line originates from in main memory. It uses a directory store to hold the address 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 main memory. This information is held in the **data section.**
- ➔ The size of a cache is defined as the actual code or data the cache can store from main memory. Not included in the cache size is the cache memory required to support cache-tags or status bits.
- ➔ Status bit

Status bits

- ➔ Two common status bits are the valid bit and dirty bit.
- ➔ A valid bit marks a cache line as active, meaning it contains live data originally taken from main memory and is currently available to the processor core on demand.
- ➔ A dirty bit defines whether or not a cache line contains data that is different from the value it represents in main memory.

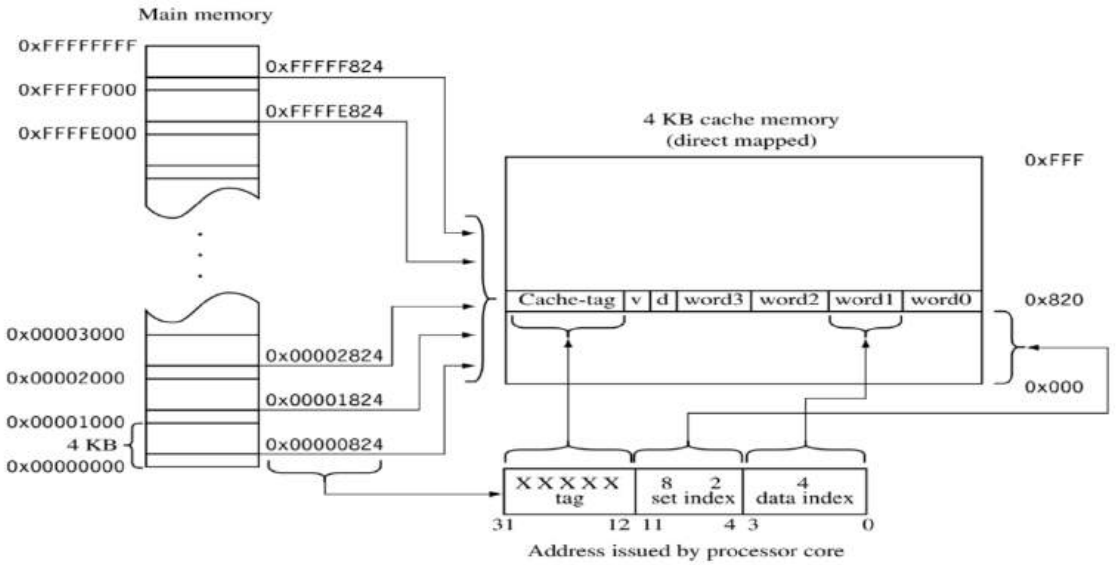
Cache Controller

- 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.

9.b

The Relationship between Cache and Main Memory



- Portions of main memory are temporarily stored in cache memory.
- The simplest form of cache, known as a direct-mapped cache.
- In a direct-mapped cache each addressed location in main memory maps to a single location in cache memory. Since main memory is much larger than cache memory, there are many addresses in main memory that map to the same single location in cache memory.
- A direct-mapped cache is a simple solution, but there is a design cost inherent in having a single location available to store a value from main memory.
- Direct-mapped caches are subject to high levels of thrashing—a software battle for the same location in cache memory.
- The result of thrashing is the repeated loading and eviction of a cache line.

10.a

Set Associativity

- Some caches include an additional design feature to reduce the frequency of thrashing.
- This structural design feature is a change that divides the cache memory into smaller equal units, called ways.
- The set index now addresses more than one cache line—it points to one cache line in each way.
- Instead of one way of 256 lines, the cache has four ways of 64 lines.
- The four cache lines 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 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.

- 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.

10.b

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.