


CMR										
INSTITUTE OF TECHNOLOGY		USN								
Internal Test MAY 2017– II Solution										
Sub:	Microprocessors and Microcontrollers							Code:	15CS44	
Date:	09 / 05 / 2017	Duration:	90 mins	Max Marks:	50	Sem:	IV	Branch:	CSE and ISE	
Answer Any FIVE Questions										

1. Architecture of ARM based embedded system hardware and software.

Embedded systems can control many different devices, from small sensors found on a production line, to the real-time control systems used on a NASA space probe. All these devices use a combination of software and hardware components. Each component is chosen for efficiency and, if applicable, is designed for future extension and expansion.

Embedded System Hardware:

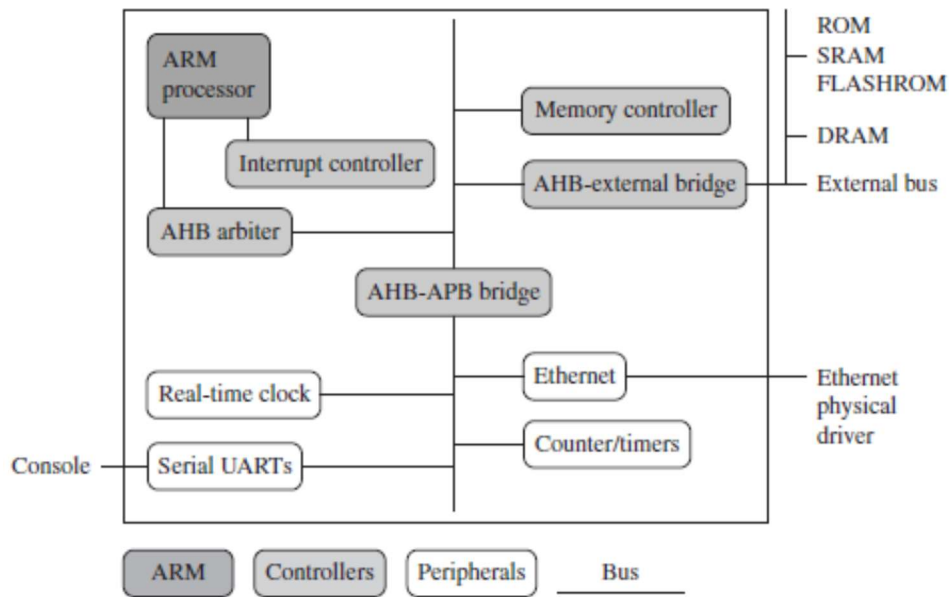


Figure 1.1 ARM based embedded device

Figure 1.1 shows a typical embedded device based on an ARM core. We can separate the device into four main hardware components:

1. The ARM processor controls the embedded device. Different versions of the ARM processor are available to suit the desired operating characteristics. An ARM processor comprises a core (the execution engine that processes instructions and manipulates data)

plus the surrounding components that interface it with a bus. These components can include memory management and caches.

2. Controllers coordinate important functional blocks of the system. Two commonly found controllers are interrupt and memory controllers.
3. The peripherals provide all the input-output capability external to the chip and are responsible for the uniqueness of the embedded device.
4. 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) 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).
- Using AMBA, peripheral designers can reuse the same design on multiple projects. A peripheral can simply be bolted on to the on-chip bus without having to redesign an interface for each different processor architecture.
- ASB is a bidirectional bus design.
- APB is used with slower peripherals.
- AHB is based on a centralized multiplexed bus scheme, thus runs at higher clock speeds and provides higher data throughput. AHB bus is used for the high performance peripherals.

Memory:

An embedded system has to have some form of memory to store and execute code. Cost, performance, and power consumption are the parameters considered while deciding upon specific memory characteristics, such as hierarchy, width, and type. Like if memory has to run twice as fast to maintain a desired bandwidth, then the memory power requirement may be higher.

- **Hierarchy:** Memory can be Cache, Main memory or Secondary memory.

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. The cache is placed between main memory and the core. It is used to speed up data transfer between the processor and main memory. The main memory is large and is generally stored in separate chips. Load and store instructions access the main memory unless the values have been stored in the cache for fast access. Secondary storage is the largest and slowest form of memory. Hard disk drives and CD-ROM drives are examples of secondary storage. Many small embedded systems do not require the performance benefits of a cache.

- Width: The memory width is the number of bits the memory returns on each access—typically 8, 16, 32, or 64 bits. The memory width has a direct effect on the overall performance and cost ratio. If you have an un-cached system using 32-bit ARM instructions and 16-bit-wide memory chips, then the processor will have to make two memory fetches per instruction. Each fetch requires two 16-bit loads. This obviously has the effect of reducing system performance, but the benefit is that 16-bit memory is less expensive. In contrast, if the core executes 16-bit Thumb instructions, it will achieve better performance with a 16-bit memory. The higher performance is a result of the core making only a single fetch to memory to load an instruction. Hence, using Thumb instructions with 16-bit-wide memory devices provides both improved performance and reduced cost.
- Types: RAM or ROM
 - Read only memory (ROM) is the least flexible of all memory types because it contains an image that is permanently set at production time and cannot be reprogrammed. ROMs are used in high-volume devices that require no updates or corrections. Many devices also use a ROM to hold boot code.
 - Random Access memory (RAM)- SRAM, DRAM or SDRAM

Peripherals

Embedded systems that interact with the outside world need some form of peripheral device. A peripheral device performs input and output functions for the chip by connecting to other devices that are off-chip.

All ARM peripherals are memory mapped—the programming interface is a set of memory-addressed registers. The address of these registers is an offset from a specific peripheral base address.

Controllers are specialized peripherals that implement higher levels of functionality within an embedded system. Two important types of controllers are memory controllers and interrupt controllers. Memory controllers connect different types of memory to the processor bus. On power-on a memory controller is configured in hardware to allow certain memory devices to be active. These memory devices allow the initialization code to be executed. Some memory devices must be set up by software.

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 by setting the appropriate bits in the interrupt controller registers. There are two types of interrupt controller available for the ARM processor: the standard interrupt controller and the vector interrupt controller (VIC). The standard interrupt controller sends an interrupt signal to the processor core when an external device requests servicing. It can be programmed to ignore or mask an individual device or set of devices. The VIC is more powerful than the standard interrupt controller because it prioritizes interrupts and simplifies the determination of which device caused the interrupt.

Embedded System Software

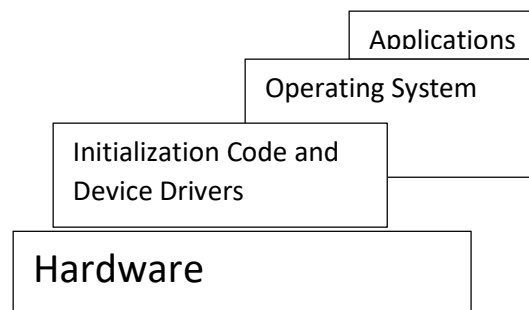


Figure 1.2 Embedded system software

An embedded system needs software to drive it. Figure 1.2 shows four typical software components required to control an embedded device. Each software component in the stack uses a higher level of abstraction to separate the code from the hardware device. The software components can run from ROM or RAM. ROM code that is fixed on the device (for example, the initialization code) is called firmware.

- The **initialization code** is the first code executed on the board and is specific to a particular target. It sets up the minimum parts of the board before handing control over to the operating system. Initialization code (or boot code) takes the processor from the reset state to a state where the operating system can run. It usually configures the memory controller and processor caches and initializes some devices. In a simple system the operating system might be replaced by a simple scheduler or debug monitor. The initialization code handles a number of administrative tasks prior to handing control over to an operating system image. We can group these different tasks into three phases: initial hardware configuration, diagnostics, and booting.
- The **operating system** provides an infrastructure to control applications and manage hardware system resources. Many embedded systems do not require a full operating system but merely a simple task scheduler that is either event or poll driven. An operating system organizes the system resources: the peripherals, memory, and processing time. With an operating system controlling these

resources, they can be efficiently used by different applications running with in the operating system environment.

- Operating systems may be divided into two main categories: real-time operating systems (RTOSs) and platform operating systems. RTOSs provide guaranteed response times to events. Platform operating systems require a memory management unit to manage large, non-real-time applications and tend to have secondary storage. These two categories of operating system are not mutually exclusive, there are operating systems that use an ARM core with a memory management unit and have real-time characteristics.
- The **device drivers** provide a consistent software interface to the peripherals on the hardware device.
- An **application** performs one of the tasks required for a device. There may be multiple applications running on the same device, controlled by the operating system.

2. (a) Draw and explain ARM data flow model.

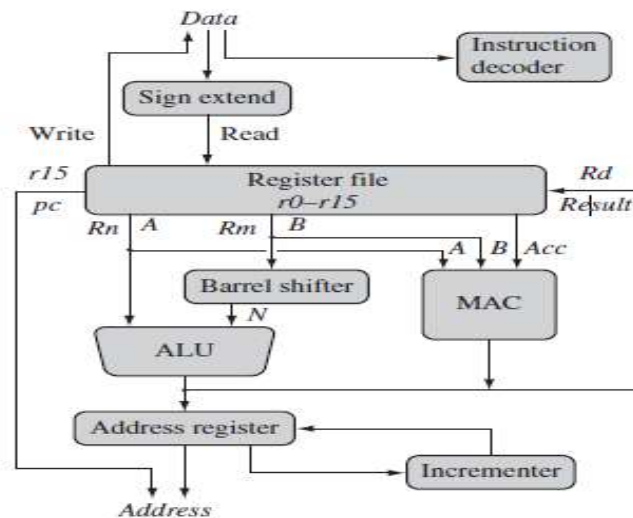


Figure 2.1 ARM core dataflow model.

- The instruction decoder translates instructions before they are executed.
- The ARM processor, like all RISC processors, uses a load-store architecture. 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.

Note: There are no data processing instructions that directly manipulate data in memory. Thus, data processing is carried out solely in registers.

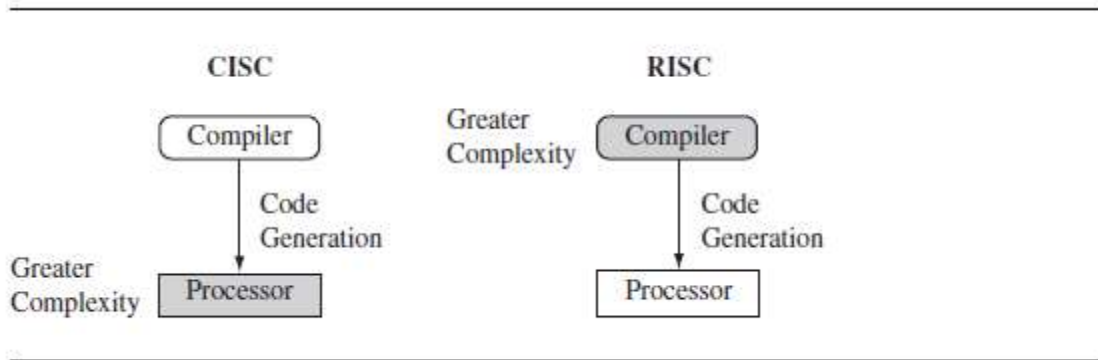
- The sign extend hardware converts signed 8-bit and 16-bit numbers to 32-bit values as they are read from memory and placed in a register.
- ARM instructions typically have two source registers, Rn and Rm, and a single result or destination register, Rd. Source operands are read from the register file using the internal buses A and B
- 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.

(b) What are the design rules for RISC design philosophy

The ARM core uses a RISC architecture. RISC is a design philosophy aimed at delivering simple but powerful instructions that execute within a single cycle at a high clock speed. The RISC philosophy concentrates on reducing the complexity of instructions performed by the hardware because it is easier to provide greater flexibility and intelligence in software rather than hardware. As a result, a RISC design places greater demands on the compiler. In contrast, the traditional complex instruction set computer (CISC) relies more on the hardware for instruction functionality, and consequently the CISC instructions are more complicated. Figure 1.1 illustrates these major differences.

The RISC philosophy is implemented with four major design rules:

1. Instructions—RISC processors have a reduced number of instruction classes. These classes provide simple operations that can each execute in a single cycle. The compiler or programmer synthesizes complicated operations (for example, a divide operation) by combining several simple instructions. Each instruction is a fixed length to allow the pipeline to fetch future instructions before decoding the current instruction. In contrast, in CISC processors the instructions are often of variable size and take many cycles to execute.
2. Pipelines—The processing of instructions is broken down into smaller units that can be executed in parallel by pipelines. Ideally the pipeline advances by one step on each cycle for maximum throughput. Instructions can be decoded in one pipeline stage. There is no need for an instruction to be executed by a mini program called microcode as on CISC processors.
3. Registers—RISC machines have a large general-purpose register set. Any register can contain either data or an address. Registers act as the fast local memory store for all data processing operations. In contrast, CISC processors have dedicated registers for specific purposes.
4. Load-store architecture—The processor operates on data held in registers. Separate load and store instructions transfer data between the register bank and external memory. Memory accesses are costly, so separating memory accesses from data processing provides an advantage because you can use data items held in the register bank multiple times without needing multiple memory accesses. In contrast, with a CISC design the data processing operations can act on memory directly. These design rules allow a RISC processor to be simpler, and thus the core can operate at higher clock frequencies. In contrast, traditional CISC processors are more complex and operate at lower clock frequencies. Over the course of two decades, however, the distinction between RISC and CISC has blurred as CISC processors have implemented more RISC concepts.



3. (a) What is CPSR. Explain in detail

The CPSR is a dedicated 32-bit register and resides in the register file. The ARM core uses the CPSR to monitor and control internal operations. The Figure 3.1 shows the CPSR layout.

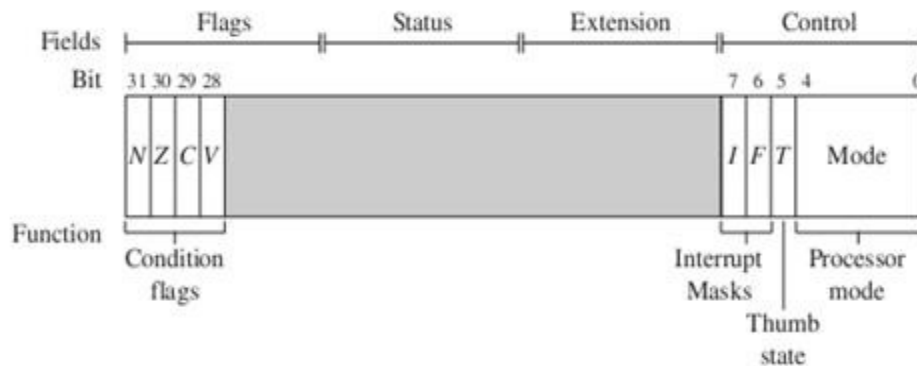


Fig 3.1 CPSR Layout

The CPSR is divided into four fields, each 8bits 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 interrupt mask bits.

Flags: The flags field contains the condition flags. Some ARM processor cores have extra bits allocated. For example, the J bit (24), which can be found in the flags field, is only available on Jazelle-enabled processors, which execute 8 bit java code.

V-oVerflow: the result causes a signed overflow

C-Carry: the result causes an unsigned carry

Z- Zero: the result is zero, frequently used to indicate equality

N- Negative: bit 31 of the result is a binary 1

Q (bit 27)-Saturation: the result causes an overflow and/or saturation

Processor Modes: The processor mode determines which registers are active and the access rights to the CPSR register itself. Each processor mode is either privileged or non-privileged: A privileged mode allows full read-write access to the CPSR. Conversely, a non-privileged mode only allows read access to the control field in the CPSR but still allows read-write access to the condition flags. There are seven processor modes in total: six privileged modes (abort, fast interrupt request, interrupt request, supervisor, system, and undefined) and one non-privileged mode (user).

State and Instruction Sets: The state of the core determines which instruction set is being executed. There are three instruction sets: ARM, Thumb, and Jazelle. The ARM instruction set is only active when the processor is in ARM state. Similarly the Thumb instruction set is only active when the processor is in Thumb state. Once in Thumb state the processor is executing purely Thumb 16-bit instructions. Jazelle executes 8-bit instructions and is a hybrid mix of software and hardware designed to speed up the execution of Java bytecodes. You cannot intermingle sequential ARM, Thumb, and Jazelle instructions.

The Jazelle J and Thumb T bits in the CPSR reflect the state of the processor. When both J and T bits are 0, the processor is in ARM state and executes ARM instructions. If J=1 then the core is in Jazelle state and T=1 then the core is in Thumb state.

3 (b) Define the concept of pipelining for ARM processors.

A pipeline is the mechanism a RISC processor uses to execute instructions. Using a pipeline speeds up execution by fetching the next instruction while other instructions are being decoded and executed. ARM7 has a three stage pipeline as shown in the fig3.2



Fig 3.2 ARM7 pipeline stages

Figure 3.3 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.

The three instructions are placed into the pipeline sequentially. In the first cycle the core fetches the ADD instruction from memory. In the second cycle the core fetches the SUB instruction and decodes the ADD instruction. In the third cycle, both the SUB and ADD instructions are moved along the pipeline. 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 allows the core to execute an instruction every cycle.

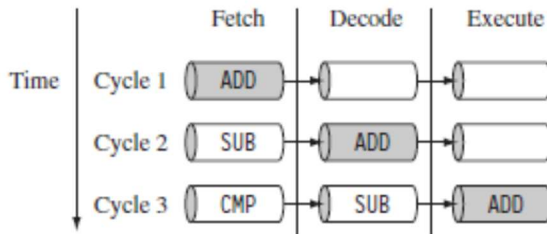


Fig 3.3 ARM7 pipeline instruction execution sequence

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.
- Increases the performance.
- The system *latency* also increases because it takes more cycles to fill the pipeline before the core can execute an instruction.
- The increased pipeline length implies data dependency between certain stages.

The pipeline design for each ARM family differs. For example, The ARM9 core increases the pipeline length to five stages, as shown in Figure 3.4. The ARM9 adds a memory and write back stage.

- Under memory stage data memory is accessed if required else the result is buffered for one clock cycle.
- Under write back stage the result is written back in to the register file including any data loaded from memory.

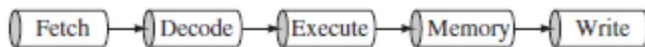


Fig 3.4 ARM9 pipeline stages

ARM10 has 6 stage pipeline with an additional issue stage. The issue stage translates Thumb instructions into their appropriate ARM counterpart, issues coprocessor instructions where appropriate, and begins to decode the instruction.

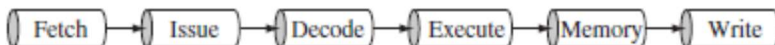


Fig 3.5 ARM10 pipeline stages

4. Write a program that scan the string “MICROPROCESSOR” and replaces all the uppercase R with lowercase r.

.MODEL SMALL

.STACK 64H

.DATA

M1 DB 'MICROPROCESSOR\$'

M2 DB 'THE NEW STRING IS:', 10, 13, '\$'

LEN EQU 14

.CODE

MOV AX,@DATA

; INITIALISE DATA SEGMENT AND EXTRA SEGMENT REG

MOV DS, AX

MOV ES, AX

LEA DI, M1 ; DI POINTING TO THE STRING

MOV AL, 'R' ; CHARACTER TO BE SEARCHED FOR IN AL REG

CLD ; CLEAR DIRECTION FLAG, DI INCREMENTS AFTER
; EXECUTION OF EACH STRING INSTRUCTION

MOV CX, LEN ; INITIALISE COUNTER

MOV DL, 'r'

; SCAN THE STRING FOR R, IF FOUND REPLACE WITH r

CHECK: REPNE SCASB

CMP CX, 0

JE DIS

MOV [DI-1], DL

JMP CHECK

; DISPLAY THE STRING

DIS: LEA DX, M2

MOV AH, 09H

INT 21H

```

LEA DX, M1

MOV AH, 09H

INT 21H

MOV AH, 4CH

IN 21H

END

```

5. Write a program to store byte FFH in 200 memory locations and also test the contents of each location to see if FFH is there. If it fails, the system should display the message “Bad Memory”.

Solution:

Assuming that ES and DS have been assigned in the ASSUME directive, the following is from the code segment:

```

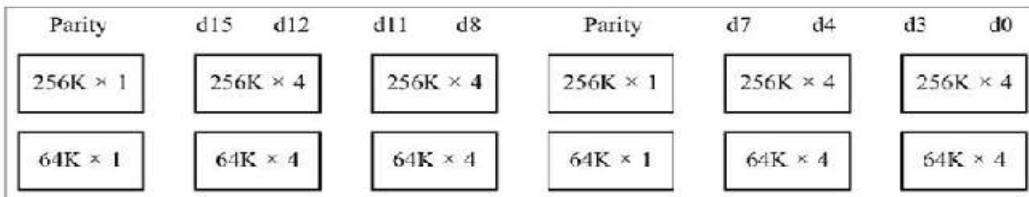
;PUT PATTERN AAAAH IN TO 50 WORD LOCATIONS
MOV AX,DTSEG ;INITIALIZE
MOV DS,AX ;DS REG
MOV ES,AX ;AND ES REG
CLD ;CLEAR DF FOR INCREMENT
MOV CX,50 ;LOAD THE COUNTER (50 WORDS)
MOV DI,OFFSET MEM_AREA ;LOAD THE POINTER FOR DESTINATION
MOV AX,0AAAAH ;LOAD THE PATTERN
REP STOSW ;REPEAT UNTIL CX=0
;BRING IN THE PATTERN AND TEST IT ONE BY ONE
MOV SI,OFFSET MEM_AREA ;LOAD THE POINTER FOR SOURCE
MOV CX,100 ;LOAD THE COUNT (COUNT 100 BYTES)
AGAIN: LODSB ;LOAD INTO AL FROM DS:SI
XOR AL,AH ;IS PATTERN THE SAME?
JNZ OVER ;IF NOT THE SAME THEN EXIT
LOOP AGAIN ;CONTINUE UNTIL CX=0
JMP EXIT ;EXIT PROGRAM
OVER: MOV AH,09 ;{ DISPLAY
MOV DX, OFFSET MESSAGE ;{ THE MESSAGE
INT 21H ;{ ROUTINE
EXIT: ...

```

6. Explain 16-bit Memory Interfacing with relevant block diagrams.

- In the design of current x86 PCs, details of CPU connection to memory and other peripherals are not visible for educational purposes.
 - 16-bit bus interfacing to memory chips is a detail now buried within a current PCs chipset.
 - Concepts from 80286 apply to any 16-bit microprocessor.

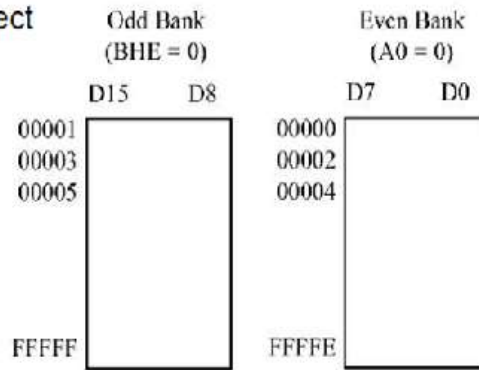
640 KB of DRAM for 16-bit buses.



- In a 16-bit CPU, memory locations 00000–FFFFF are designated as odd and even bytes.
 - To distinguish between odd & even bytes, the CPU provides a signal called **BHE** (bus high enable).
 - BHE, with A0 is used to select odd/even bytes.

Table 10-7: Distinguishing Between Odd and Even Bytes

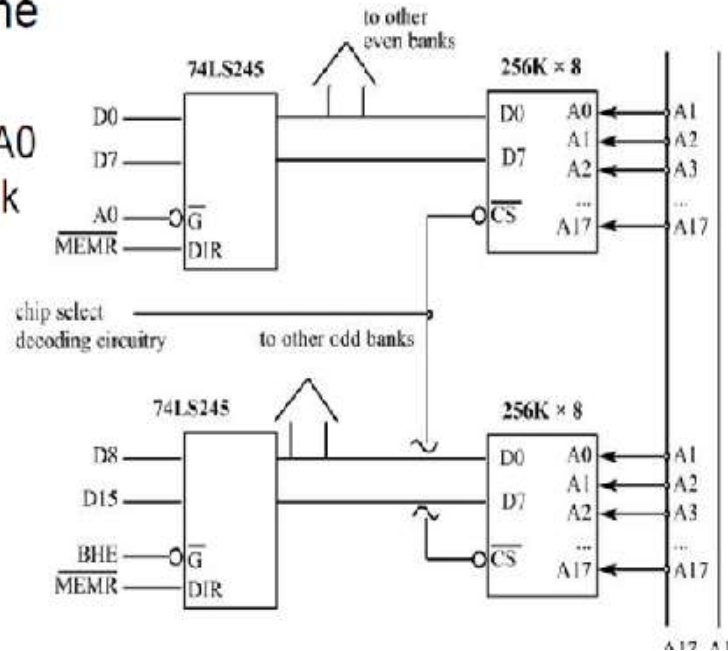
BHE	A0		
0	0	Even word	D0–D15
0	1	Odd byte	D8–D15
1	0	Even byte	D0–D7
1	1	None	



- Connection for the 16-bit data bus.

- Note the use of A0 and BHE as bank selectors.

- Also use of the 74LS245 chip as a data bus buffer.



7. Write an assembly language program to program 8255 to read data from port A and send it to port B. Read data from port C lower and send it to port C upper. (Explain how to find out the required control word). Port address is from 0E880H to 0E883H.

CONTROL WORD: 10010001 = 91H

.MODEL SMALL

.STACK 100

.DATA

PA EQU 0E880H

PB EQU 0E881H

PC EQU 0E882H

CT EQU 0E883H

.CODE

MOV AX, @DATA

MOV DS, AX

```
MOV DX, CT
MOV AL, 91H
OUT DX, AL
MOV DX, PA
IN AL, DX
MOV DX, PB
OUT DX, AL
MOV DX, PC
IN AL, DX
AND AL, 0FH
MOV BL, 04
ROL AL, BL
OUT DX, AL
MOV AH, 4CH
INT 21H
END
```

8. (a) *Exceptions, Interrupts and Vector Table*

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). Operating systems such as Linux and Microsoft's embedded products can take advantage of this feature.

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:

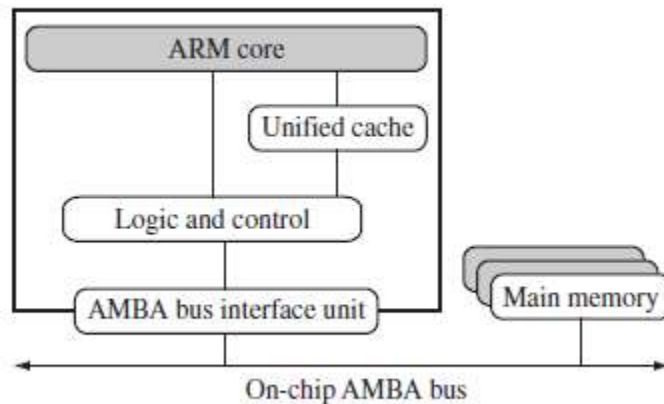
- *Reset vector* is the location of the first instruction executed by the processor when power is applied. This instruction branches to the initialization code.
- *Undefined instruction vector* is used when the processor cannot decode an instruction.
- *Software interrupt vector* is called when you execute 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*.

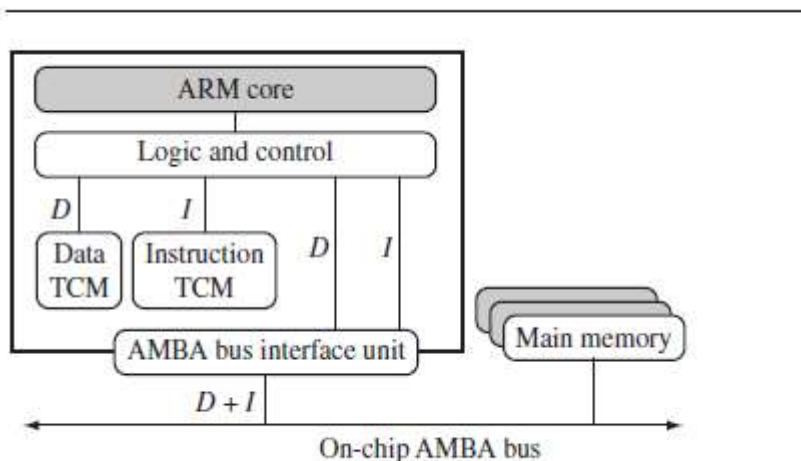
Exception/interrupt	Shorthand	Address	High address
Reset	RESET	0x00000000	0xffff0000
Undefined instruction	UNDEF	0x00000004	0xffff0004
Software interrupt	SWI	0x00000008	0xffff0008
Prefetch abort	PABT	0x0000000c	0xffff000c
Data abort	DABT	0x00000010	0xffff0010
Reserved	—	0x00000014	0xffff0014
Interrupt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

(b) Core extensions for ARM processor

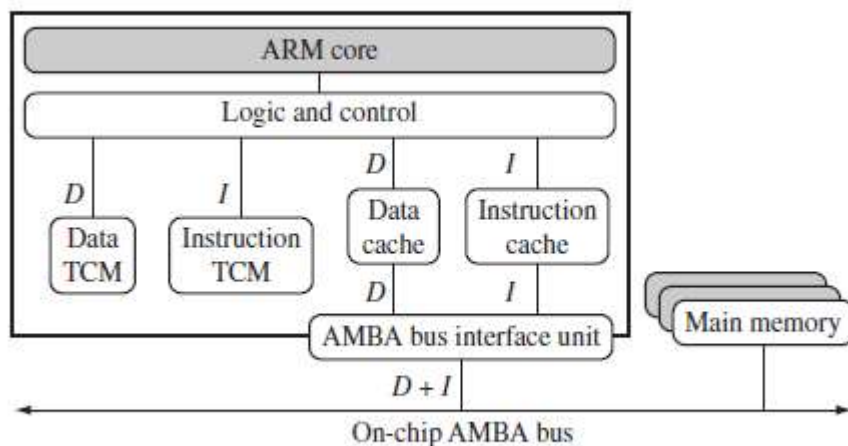
The hardware extensions covered in this section are standard components placed next to the ARM core. They improve performance, manage resources, and provide extra functionality and are designed to provide flexibility in handling particular applications. Each ARM family has different extensions available. There are three hardware extensions ARM wraps around the core: cache and tightly coupled memory, memory management, and the coprocessor interface.



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. Most ARM-based embedded systems use a single-level cache internal to the processor. Of course, many small embedded systems do not require the performance gains that a cache brings. 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 2.13. For simplicity, we have called the glue logic that connects the memory system to the AMBA bus *logic and control*.



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 2.15 shows an example core with a combination of caches and TCMs.

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

inappropriate 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 providing 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 is normally used for small, simple embedded systems that require no protection from rogue applications. *MPUs* employ 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 permissions. This type of memory management is used for systems that require memory protection but don't have a complex memory map.

- *MMUs* 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 a virtual-to-physical address map as well as access permissions. MMUs are designed for more sophisticated platform operating systems that support multitasking.

Coprocessors

Coprocessors can be attached to the ARM processor. A coprocessor extends the processing features of a core by extending the instruction set or by providing configuration registers. More than one coprocessor can be added to the ARM core via the coprocessor interface. The coprocessor can be accessed through a group of dedicated ARM instructions that provide a load-store type interface. Consider, for example, coprocessor 15: The ARM processor uses coprocessor 15 registers to control the cache, TCMs, and memory management. The coprocessor can also extend the instruction set by providing a specialized group of new instructions. For example, there are a set of specialized instructions that can be added to the standard ARM instruction set to process vector floating-point (VFP) operations. These new instructions are processed in the decode stage of the ARM pipeline. If the decode stage sees a coprocessor instruction, then it offers it to the relevant coprocessor. But if the coprocessor is not present or doesn't recognize the instruction, then the ARM takes an undefined instruction exception, which allows you to emulate the behavior of the coprocessor in software.

(c) Control Word Register of 8255

