Computer Organization and ARM Microcontrollers (21EC52)

VTU Solution- Dec23/Jan24

CBCS SCHEME

USN | | | | | | | | | | 21EC52

Fifth Semester B.E. Degree Examination, Dec.2023/Jan.2024
Computer Organization and Arm Microcontrollers

Time: 3 hrs. Max. Marks: 100

Note: Answer any FIVE full questions, choosing ONE full question from each module.

Module-1

1 a. With a neat diagram, discuss the operational concepts in a computer highlighting the role of PC, MAR, MDR and IR. **(10 Marks)**
b. Explain system software functions in computer. **(06 Marks)**
c. Explain computer basic performance equation. **(04 Marks)**

OR

2 a. Explain operation of DMA with neat diagram. **(10 Marks)**
b. With a neat diagram, discuss implementation of interrupt priority using individual request and acknowledge lines. **(06 Marks)**
c. Illustrate with a neat diagram, a computer using different interface standards. **(04 Marks)**

Module-2

3 a. With a neat diagram, explain the internal organization of 16 × 8 memory chip. **(10 Marks)**
b. State and explain the types of read only memory and memory hierarchy. **(10 Marks)**

OR

4 a. With a neat diagram, explain the three bus organization of a datapath. **(10 Marks)**
b. Explain basic idea of pipelining and 4-stage pipeline structure. **(10 Marks)**

Module-3

5 a. With a neat diagram, explain the four main hardware components of an ARM based embedded device. **(08 Marks)**
b. Discuss ARM design philosophy. **(06 Marks)**
c. Explain the factors that make ARM instruction set suitable for embedded applications. **(06 Marks)**

OR

6 a. Explain ARM core data flow model with a neat diagram. **(08 Marks)**
b. Explain the different processor modes provided by ARM7. **(06 Marks)**
c. Discuss with a neat diagram:
i) Von Neumann architecture with cache
ii) Harvard architecture with TCM. **(06 Marks)**

Module-4

7 a. Explain with neat diagram, barrel shifter operation in ARM processor. **(08 Marks)**
b. Explain with an example the concept of semaphore using swap instruction. **(06 Marks)**
c. Develop an assembly language program to multiply two 16-bit numbers. **(06 Marks)**

OR

8 a. Explain the following with example:
i) MSR ii) MVN iii) TST iv) BIC. **(08 Marks)**

b. Explain with an example forward and backward branch. **(06 Marks)**

c. Develop an assembly language program to find GCD of two numbers using conditional execution. **(06 Marks)**

Module-5

9 a. Discuss with an example code density in thumb instruction set over ARM. **(08 Marks)**

b. Explain ARM-thumb interworking. **(06 Marks)**

c. Explain with example thumb stack operations. **(06 Marks)**

OR

10 a. Explain with an example the effect of using 'char' and 'short' as local variable types in ARM processor. **(08 Marks)**

b. List the C compiler data type mapping for an ARM target with their implementation. **(05 Marks)**

c. With an example, compare the efficiencies of signed int and unsigned int with an example. **(07 Marks)**

* * * * *

1. (a) With a neat diagram explain the operational concepts in a computer highlighting the role of PC, MAR, MDR, IR.

BASIC OPERATIONAL CONCEPTS

- An Instruction consists of 2 parts, 1) Operation code (Opcode) and 2) Operands.

OPCODE	*OPERANDS*

- The data/operands are stored in memory.
- The individual instruction are brought from the memory to the processor.
- Then, the processor performs the specified operation.
- Let us see a typical instruction

  ADD LOCA, R0
- This instruction is an addition operation. The following are the steps to execute the instruction:

  Step 1: Fetch the instruction from main-memory into the processor.
  Step 2: Fetch the operand at location LOCA from main-memory into the processor.
  Step 3: Add the memory operand (i.e. fetched contents of LOCA) to the contents of register R0.
  Step 4: Store the result (sum) in R0.
- The same instruction can be realized using 2 instructions as:

  Load LOCA, R1
  Add R1, R0
- The following are the steps to execute the instruction:

  Step 1: Fetch the instruction from main-memory into the processor.
  Step 2: Fetch the operand at location LOCA from main-memory into the register R1.
  Step 3: Add the content of Register R1 and the contents of register R0.
  Step 4: Store the result (sum) in R0.

MAIN PARTS OF PROCESSOR

- The **processor** contains ALU, control-circuitry and many registers.
- The processor contains 'n' general-purpose registers $\mathbf{R_0}$ through $\mathbf{R_{n-1}}$.
- The **IR** holds the instruction that is currently being executed.
- The **control-unit** generates the timing-signals that determine when a given action is to take place.
- The **PC** contains the memory-address of the next-instruction to be fetched & executed.
- During the execution of an instruction, the contents of PC are updated to point to next instruction.
- The **MAR** holds the address of the memory-location to be accessed.
- The **MDR** contains the data to be written into or read out of the addressed location.
- MAR and MDR facilitates the communication with memory.

  (IR → Instruction-Register, PC → Program Counter)
  (MAR → Memory Address Register, MDR→ Memory Data Register)

STEPS TO EXECUTE AN INSTRUCTION

1) The address of first instruction (to be executed) gets loaded into PC.
2) The contents of PC (i.e. address) are transferred to the MAR & control-unit issues Read signal to memory.
3) After certain amount of elapsed time, the first instruction is read out of memory and placed into MDR.
4) Next, the contents of MDR are transferred to IR. At this point, the instruction can be decoded & executed.
5) To fetch an operand, it's address is placed into MAR & control-unit issues Read signal. As a result, the operand is transferred from memory into MDR, and then it is transferred from MDR to ALU.
6) Likewise required number of operands is fetched into processor.
7) Finally, ALU performs the desired operation.
8) If the result of this operation is to be stored in the memory, then the result is sent to the MDR.

9) The address of the location where the result is to be stored is sent to the MAR and a Write cycle is initiated.
10) At some point during execution, contents of PC are incremented to point to next instruction in the program.

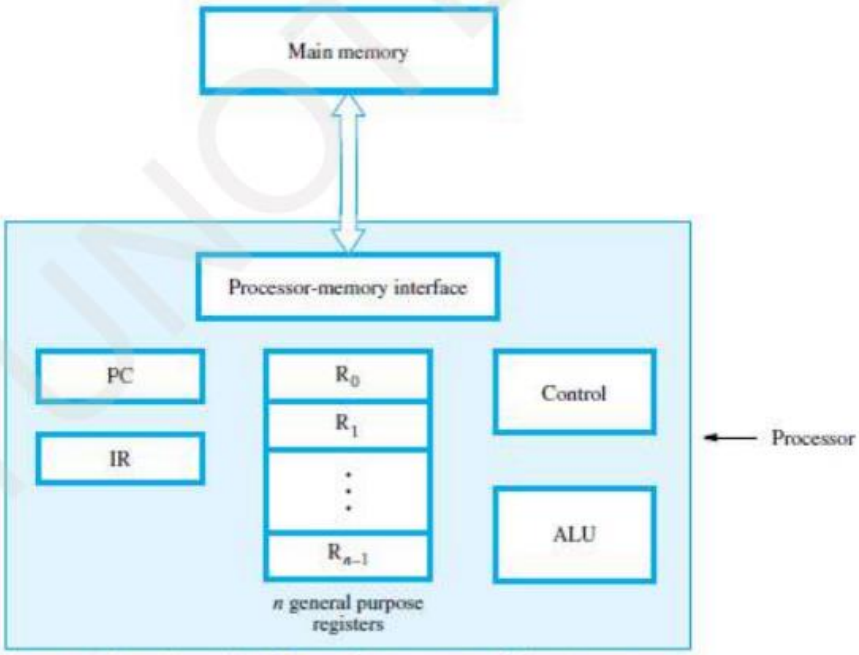


Figure 1.2 Connection between the processor and the main memory.

1(b) Explain system software functions in computer.

- System software is a collection of programs that are executed as needed to perform functions such as

  - Receiving and interpreting user commands
  - Managing the storage and retrieval of files in secondary storage devices
  - Running standard application programs such as word processors, spreadsheets, or games, with data supplied by the user
  - Controlling I/O units to receive input information and produce output results
  - Translating programs from source form prepared by the user into object form consisting of machine instructions
  - Linking and running user-written application programs with existing standard library routines, such as numerical computation packages

- Application programs are usually written in a high-level programming language, such as C, C++, Java, or Fortran, in which the programmer specifies mathematical or text-processing operations.
- A system software program called a *compiler* translates the high-level language program into a suitable machine language program.
- A large program or a collection of routines, that is used to control the sharing of and interaction among various computer units is called operating system (OS).
- The OS routines perform the tasks required to assign computer resources to individual application programs.
- These task include assigning memory and magnetic disk space to program and data files, moving data between memory and disk units, and handling I/O operations.

1© Explain Computer basic performance equation.

Basic Performance Equation

- T – processor time required to execute a program that has been prepared in high-level language
- N – number of actual machine language instructions needed to complete the execution (note: loop)
- S – average number of basic steps needed to execute one machine instruction. Each step completes in one clock cycle
- R – clock rate
- Note: these are not independent to each other

$$T = \frac{N \times S}{R}$$

How to improve T?

Performance Measurement

SPEC - System Performance Evaluation Corporation

$$\text{SPEC rating} = \frac{\text{Running time on the reference computer}}{\text{Running time on the computer under test}}$$

$$\text{SPEC rating} = \left(\prod_{i=1}^{n} \text{SPEC}_i \right)^{\frac{1}{n}}$$

2 (a) Explain the operation of DMA with neat diagram.

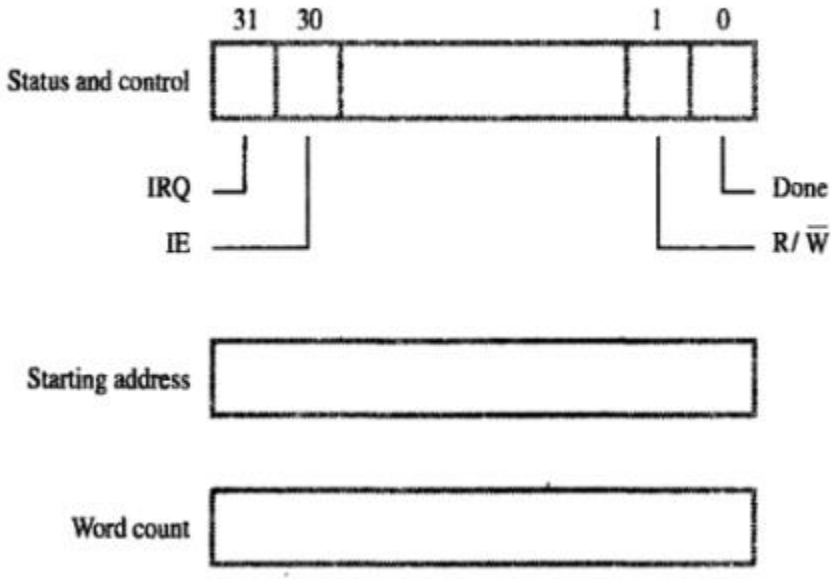
DMA

- Think about the overhead in both polling and interrupting mechanisms when a large block of data need to be transferred between the processor and the I/O device.
- A special control unit may be provided to allow transfer of a block of data directly between an external device and the main memory, without continuous intervention by the processor – direct memory access (DMA).
- The DMA controller provides the memory address and all the bus signals needed for data transfer, increment the memory address for successive words, and keep track of the number of transfers.

DMA Procedure

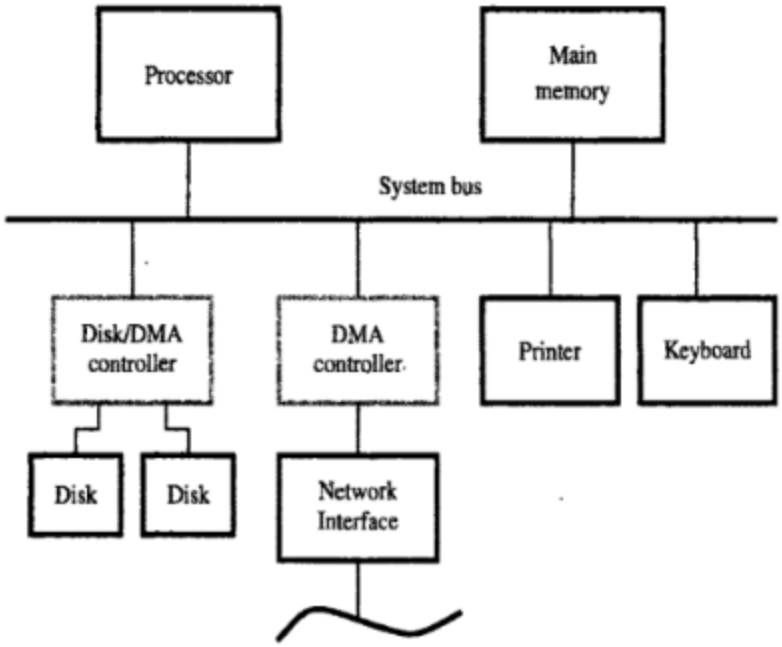
- Processor sends the starting address, the number of data, and the direction of transfer to DMA controller.
- Processor suspends the application program requesting DMA, starts DMA transfer, and starts another program.
- After the DMA transfer is done, DMA controller sends an interrupt signal to the processor.
- The processor puts the suspended program in the Runnable state.

DMA Register



Registers in a DMA interface.

System

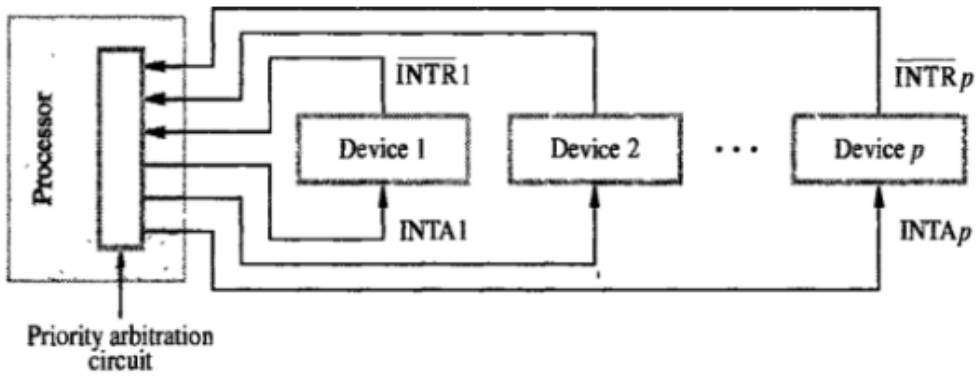


Use of DMA controllers in a computer system.

2. (b) With the neat diagram discuss implementation of interrupt priority using individual request and acknowledge lines.

Interrupt Nesting

- Simple solution: only accept one interrupt at a time, then disable all others.
- Problem: some interrupts cannot be held too long.
- Priority structure



Implementation of interrupt priority using individual interrupt-request and acknowledge lines.

2 © Illustrate with a neat diagram, a computer with different interface standards.

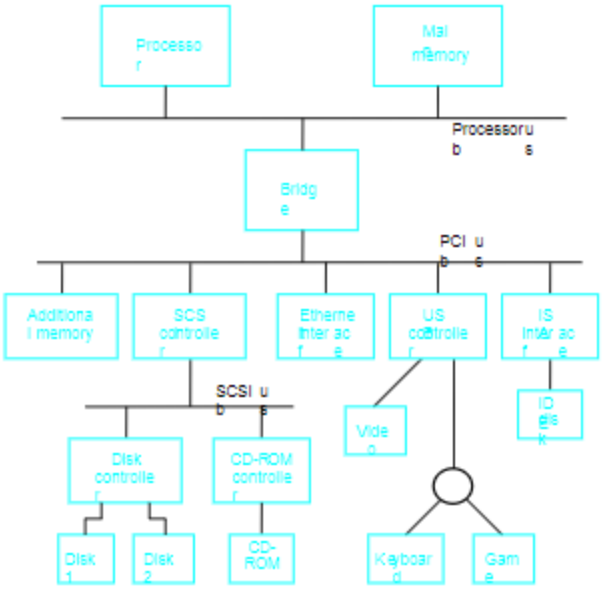


Figure 4.38. An example of a computer system using different interface standards.

The previous sections point out that there are several alternative designs for the bus of a computer. This variety means that I/O devices fitted with an interface circuit suitable for one computer may not be usable with other computers. A different interface may have to be designed for every combination of I/O device and computer, resulting in many different interfaces. The most practical solution is to develop standard interface signals and protocols.

It is helpful at this point to understand how a computer system is put together. A typical personal computer, for example, includes a printed circuit board called the motherboard. This board houses the processor chip, the main memory, and some I/O interfaces. It also has a few connectors into which additional interfaces can be plugged.

The processor bus is the bus defined by the signals on the processor chip itself. Devices that require a very high speed connection to the processor, such as the main memory, may be connected directly to this bus. For electrical reasons, only a few devices can be connected in this manner. The motherboard usually provides another bus that can support more devices. The two buses are interconnected by a circuit, which we will call a *bridge,* that translates the signals and protocols of one bus into those of the other. Devices connected to the expansion bus appear to the processor as if they were connected directly to the processor's own bus. The only difference is that the bridge circuit introduces a small delay in data transfers between the processor and those devices.

It is not possible to define a uniform standard for the processor bus. The structure of this bus is closely tied to the architecture of the processor. It is also dependent on the electrical characteristics of the processor chip, such as its clock speed. The expansion bus is not subject to these limitations, and therefore it can use a standardized signaling scheme. A number of standards have been developed. Some have evolved by default, when a particular design became commercially successful. For example, IBM

developed a bus they called ISA (Industry Standard Architecture) for their personal computer known at the time as PC AT. The popularity of that computer led to other manufacturers producing ISA-compatible interfaces for their I/O devices, thus making ISA into a de facto standard.

Some standards have been developed through industrial cooperative efforts, even among competing companies driven by their common self-interest in having compatible products. In some cases, organizations such as the IEEE (Institute of Electrical and Electronics Engineers), ANSI (American National Standards Institute), or international bodies such as ISO (International Standards Organization) have blessed these standards and given them an official status.

In this section, we present three widely used bus standards, PCI (Peripheral Component Interconnect), SCSI (Small Computer System Interface), and USB (Universal Serial Bus). The way these standards are used in a typical computer system is illustrated in Figure 4.38. The PCI standard defines an expansion bus on the motherboard. SCSI and USB are used for connecting additional devices, both inside and outside the

3. (a) With a neat diagram explain the internal organization of 16*8 memory chip.

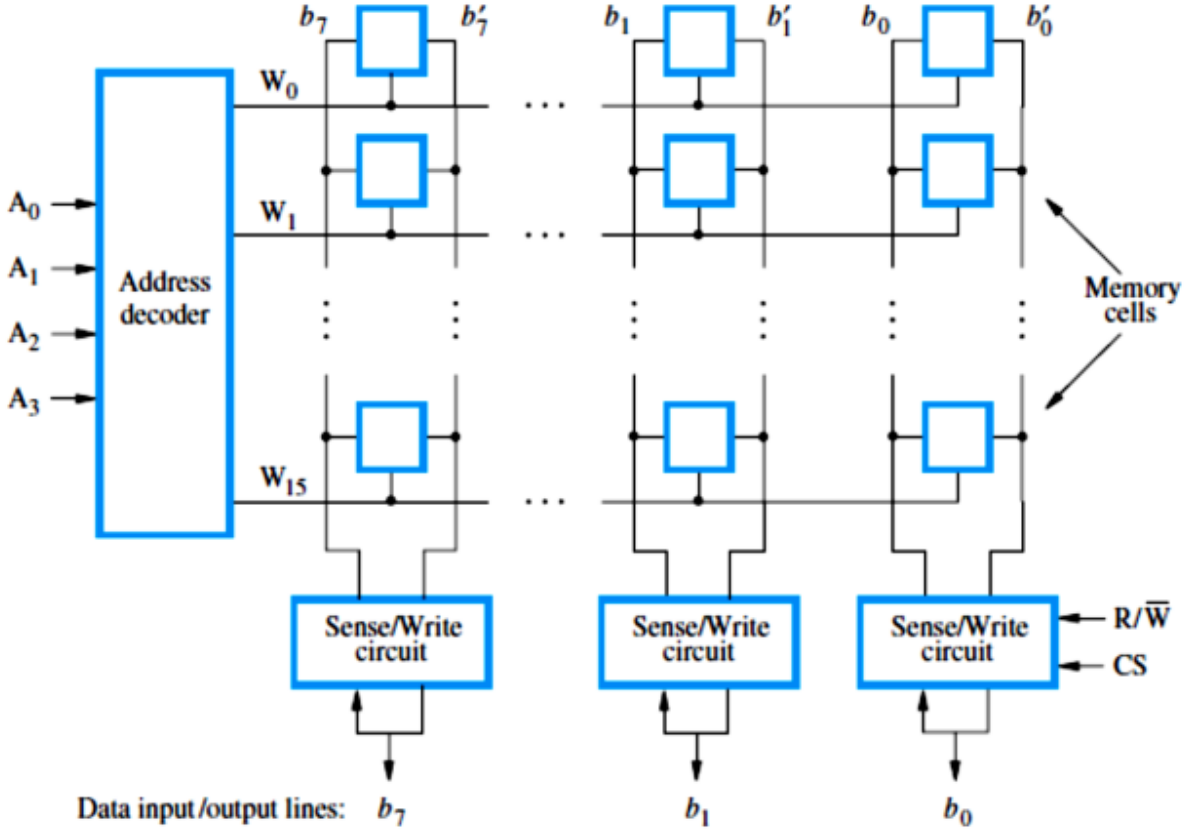


Figure 5.2 Organization of bit cells in a memory chip.

Semiconductor RAM Memories

- Semiconductor memories are available in a wide range of speeds.
- Their cycle times range from 100 ns to less than 10 ns.
- When first introduced in the late 1960s, they were much more expensive than the magnetic-core memories they replaced.
- Because of rapid advances in VLSI (Very Large Scale Integration) technology, the cost of semiconductor memories has dropped dramatically.
- As a result, they are now used almost exclusively in implementing memories.

Internal Organization of Memory Chips

- Each memory cell can hold one bit of information.
- Memory cells are organized in the form of an array.
- One row is one memory word.
- All cells of a row are connected to a common line, known as the *word line*.
- Word line is connected to the address decoder.
- The cells in each column are connected to a Sense/Write circuit by two *bit lines.*
- The Sense/Write circuits are connected to the data input/output lines of the chip.

- Figure 5.2 is an example of a very small memory circuit consisting of 16 words of 8 bits each.
- This is referred to as a 16 × 8 organization.
- The data input and the data output of each Sense/Write circuit are connected to a single bidirectional data line that can be connected to the data lines of a computer.
- Two control lines, $R/\overline{W}$ and CS, are provided.
  - The $R/\overline{W}$ ($Read/\overline{Write}$) input specifies the required operation, and the CS (Chip Select) input selects a given chip in a multichip memory system.

3 (b) State and explain the types of read only memory and memory hierarchy.

Memory Hierarchy

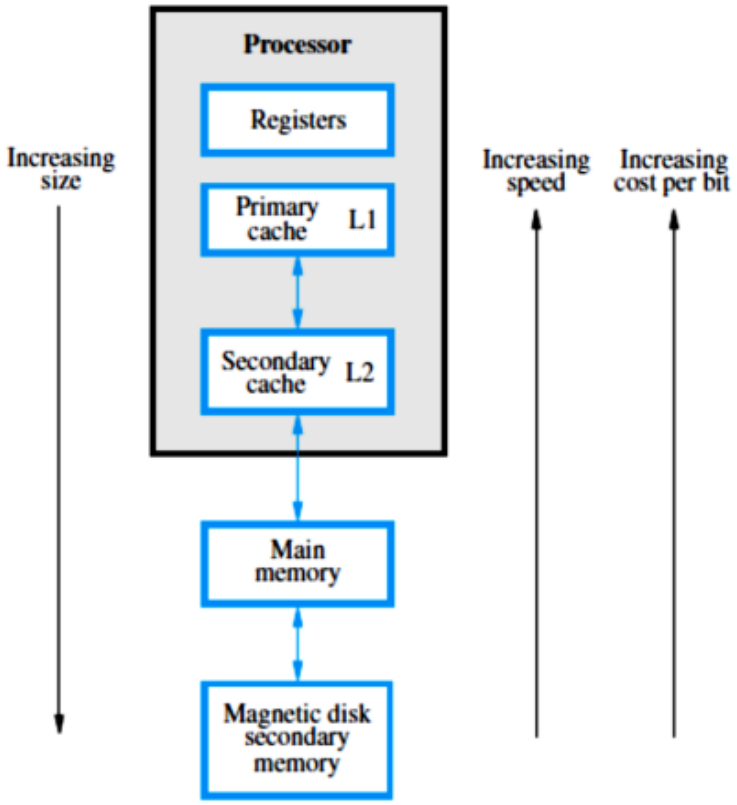


Figure 5.13 Memory hierarchy.

5.3 Read-Only Memories

Both SRAM and DRAM chips are volatile, which means that they lose the stored information if power is turned off. There are many applications that need memory devices which retain the stored information if power is turned off. For example, in a typical computer a hard disk drive is used to store a large amount of information,

including the operating system software. When a computer is turned on, the operating system software has to be loaded from the disk into the memory. This requires execution of a program that "boots" the operating system. Since the boot program is quite large, most of it is stored on the disk. The processor must execute some instructions that load the boot program into the memory. If the entire memory consisted of only volatile memory chips, the processor would have no means of accessing these instructions. A practical solution is to provide a small amount of nonvolatile memory that holds the instructions whose execution results in loading the boot program from the disk.

Nonvolatile memory is used extensively in embedded systems, which are presented in Chapter 9. Such systems typically do not use disk storage devices. Their programs are stored in nonvolatile semiconductor memory devices.

Different types of nonvolatile memory have been developed. Generally, the contents of such memory can be read as if they were SRAM or DRAM memories. But, a special writing process is needed to place the information into this memory. Since its normal operation involves only reading of stored data, a memory of this type is called *read-only memory* (ROM).

5.3.1 ROM

Figure 5.12 shows a possible configuration for a ROM cell. A logic value 0 is stored in the cell if the transistor is connected to ground at point P; otherwise, a 1 is stored. The bit line is connected through a resistor to the power supply. To read the state of the cell, the word line is activated. Thus, the transistor switch is closed and the voltage on the bit line drops to near zero if there is a connection between the transistor and ground. If there is no connection to ground, the bit line remains at the high voltage, indicating a 1. A sense circuit at the end of the bit line generates the proper output value. Data are written into a ROM when it is manufactured.

Some ROM designs allow the data to be loaded by the user, thus providing a *programmable ROM* (PROM). Programmability is achieved by inserting a fuse at point *P* in Figure 5.12. Before it is programmed, the memory contains all 0s. The user can insert 1s at the required locations by burning out the fuses at these locations using high-current pulses. Of course, this process is irreversible.

PROMs provide flexibility and convenience not available with ROMs. The latter are economically attractive for storing fixed programs and data when high volumes of ROMs are produced. However, the cost of preparing the masks needed for storing a particular information pattern in ROMs makes them very expensive when only a small number are required. In this case, PROMs provide a faster and considerably less expensive approach because they can be programmed directly by the user.

5.3.3 EPROM

Another type of ROM chip allows the stored data to be erased and new data to be loaded. Such an erasable, reprogrammable ROM is usually called an *EPROM*. It provides considerable flexibility during the development phase of digital systems. Since EPROMs are capable of retaining stored information for a long time, they can be used in place of ROMs while software is being developed. In this way, memory changes and updates can be easily made.

An EPROM cell has a structure similar to the ROM cell in Figure 5.12. In an EPROM cell, however, the connection to ground is always made at point *P* and a special transistor is used, which has the ability to function either as a normal transistor or as a disabled transistor that is always turned off. This transistor can be programmed to behave as a permanently open switch, by injecting charge into it that becomes trapped inside. Thus, an EPROM cell can be used to construct a memory in the same way as the previously discussed ROM cell.

The important advantage of EPROM chips is that their contents can be erased and reprogrammed. Erasure requires dissipating the charges trapped in the transistors of memory cells; this can be done by exposing the chip to ultraviolet light. For this reason, EPROM chips are mounted in packages that have transparent windows.

4 (a) With a neat diagram explain the three bus organization of a data path.

Multiple-Bus Organization

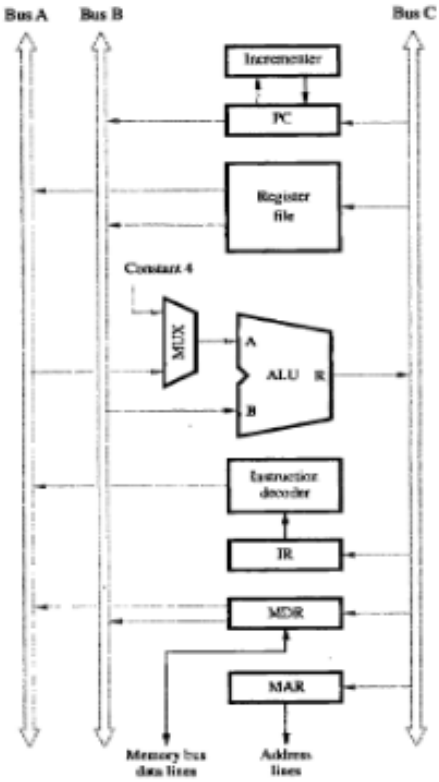


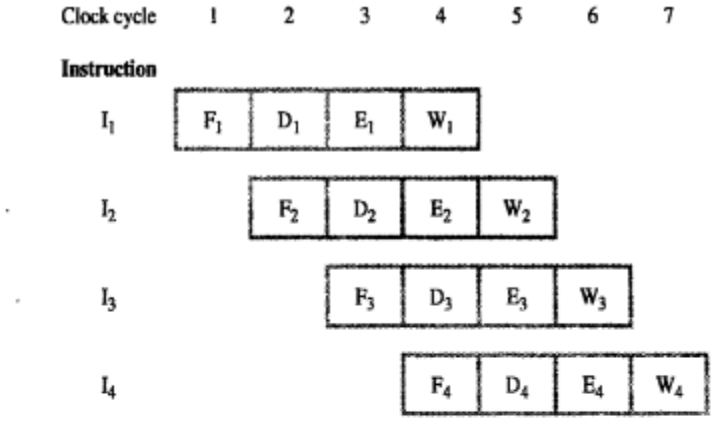
Figure 7.8 Three-bus organization of the datapath.

- Allow the contents of two different registers to be accessed simultaneously and have their contents placed on buses A and B.
- Allow the data on bus C to be loaded into a third register during the same clock cycle.
- Incrementer unit.
- ALU simply passes one of its two input operands unmodified to bus C
- control signal: R=A or R=B

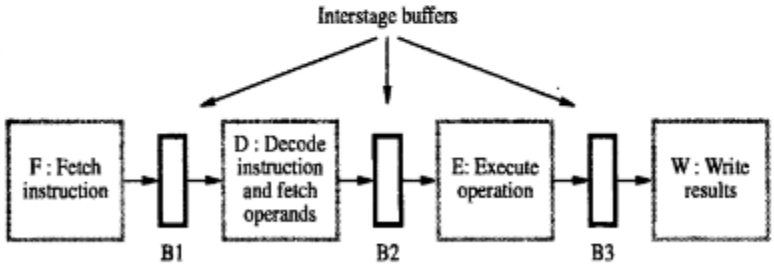
- General purpose registers are combined into a single block called registers.
- 3 ports,2 output ports –access two different registers and have their contents on buses A and B
- Third port allows data on bus c during same clock cycle.
- Bus A & B are used to transfer the source operands to A & B inputs of the ALU.
- ALU operation is performed.
- The result is transferred to the destination over the bus C.
- ALU may simply pass one of its 2 input operands unmodified to bus C.
- The ALU control signals for such an operation R=A or R=B.
- Incrementer unit is used to increment the PC by 4.

- Using the incrementer eliminates the need to add the constant value 4 to the PC using the main ALU.
- The source for the constant 4 at the ALU input multiplexer can be used to increment other address such as loadmultiple & storemultiple

4 (b) Explain the basic idea of pipelining and 4 stage pipeline structure.



(a) Instruction execution divided into four steps



(b) Hardware organization

Figure 8.2 A 4-stage pipeline.

8.1 BASIC CONCEPTS

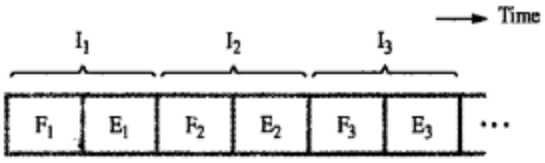
The speed of execution of programs is influenced by many factors. One way to improve performance is to use faster circuit technology to build the processor and the main memory. Another possibility is to arrange the hardware so that more than one operation can be performed at the same time. In this way, the number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed.

We have encountered concurrent activities several times before. Chapter 1 introduced the concept of multiprogramming and explained how it is possible for I/O transfers and computational activities to proceed simultaneously. DMA devices make this possible because they can perform I/O transfers independently once these transfers are initiated by the processor.

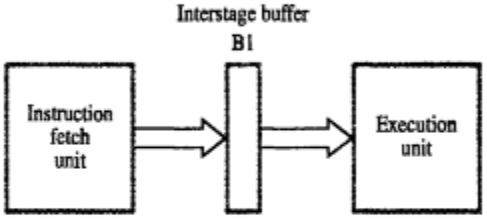
Pipelining is a particularly effective way of organizing concurrent activity in a computer system. The basic idea is very simple. It is frequently encountered in manufacturing plants, where pipelining is commonly known as an assembly-line operation. Readers are undoubtedly familiar with the assembly line used in car manufacturing. The first station in an assembly line may prepare the chassis of a car, the next station adds the body, the next one installs the engine, and so on. While one group of workers is installing the engine on one car, another group is fitting a car body on the chassis of another car, and yet another group is preparing a new chassis for a third car. It may take days to complete work on a given car, but it is possible to have a new car rolling off the end of the assembly line every few minutes.

Consider how the idea of pipelining can be used in a computer. The processor executes a program by fetching and executing instructions, one after the other. Let F_i and E_i refer to the fetch and execute steps for instruction I_i. Execution of a program consists of a sequence of fetch and execute steps, as shown in Figure 8.1*a*.

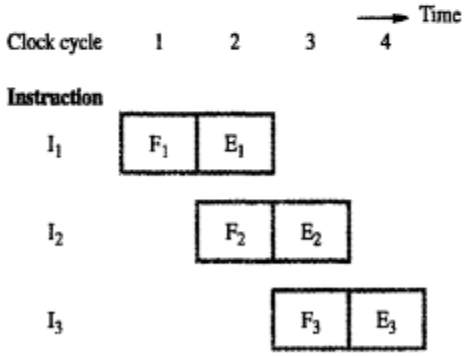
Now consider a computer that has two separate hardware units, one for fetching instructions and another for executing them, as shown in Figure 8.1*b*. The instruction fetched by the fetch unit is deposited in an intermediate storage buffer, B1. This buffer is needed to enable the execution unit to execute the instruction while the fetch unit is fetching the next instruction. The results of execution are deposited in the destination location specified by the instruction. For the purposes of this discussion, we assume that both the source and the destination of the data operated on by the instructions are inside the block labeled "Execution unit."



(a) Sequential execution



(b) Hardware organization

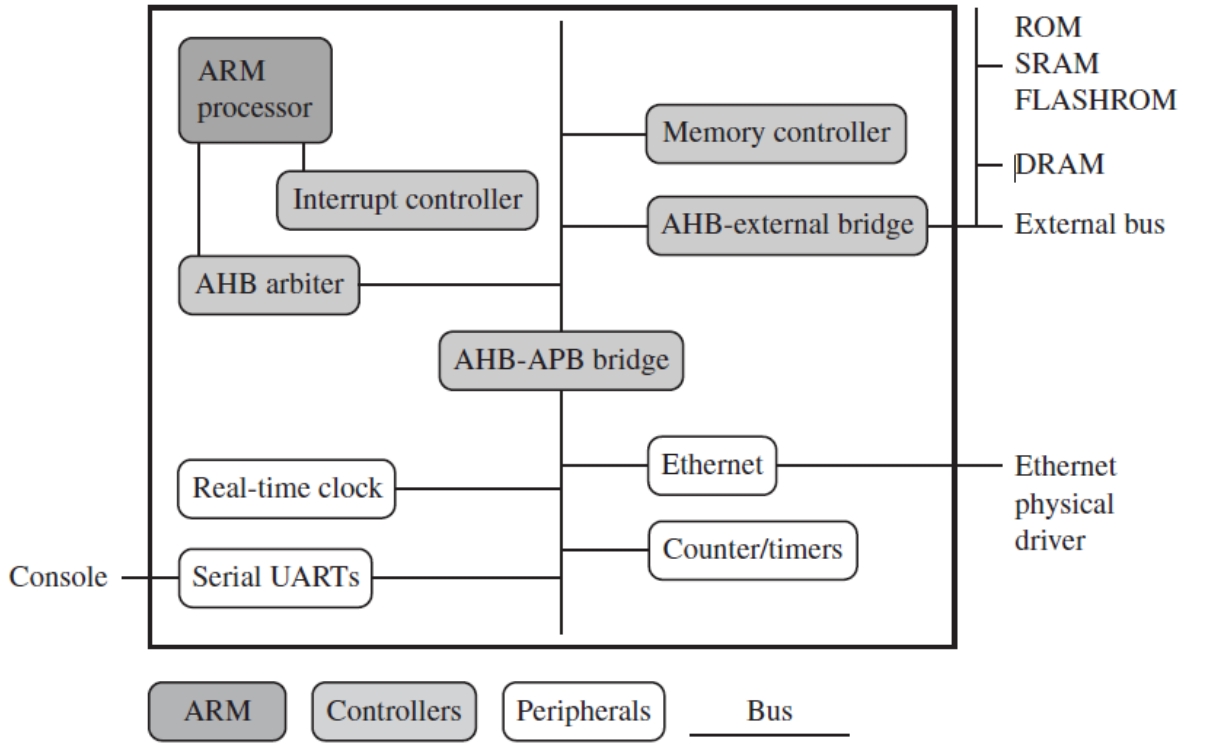


(c) Pipelined execution

Figure 8.1 Basic idea of instruction pipelining.

Q.5: [a] With a neat diagram explain the four main hardware components of an ARM based embedded device.

Sol:



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

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

■ The *ARM processor* controls the embedded device. Different versions of the ARM processor are available to suit the desired operating characteristics. An ARM processor comprises a core (the execution engine that processes instructions and manipulates data) plus the surrounding components that interface it with a bus. These components can include memory management and caches.

■ *Controllers* coordinate important functional blocks of the system. Two commonly found controllers are interrupt and memory controllers.

■ The *peripherals* provide all the input-output capability external to the chip and are responsible for the uniqueness of the embedded device.

■ A *bus* is used to communicate between different parts of the device.

Q.5 [b]:

Discuss ARM Design Philosophy

Sol:

The ARM Design Philosophy

There are a number of physical features that have driven the ARM processor design. First, portable embedded systems require some form of battery power. The ARM processor has been specifically designed to be small to reduce power consumption and extend battery operation—essential for applications such as mobile phones and personal digital assistants (PDAs).

High code density is another major requirement since embedded systems have limited memory due to cost and/or physical size restrictions. High code density is useful for applications that have limited on-board memory, such as mobile phones and mass storage devices.

In addition, embedded systems are price sensitive and use slow and low-cost memory devices. For high-volume applications like digital cameras, every cent has to be accounted for in the design. The ability to use low-cost memory devices produces substantial savings.

Another important requirement is to reduce the area of the die taken up by the embedded processor. For a single-chip solution, the smaller the area used by the embedded processor, the more available space for specialized peripherals. This in turn reduces the cost of the design and manufacturing since fewer discrete chips are required for the end product.

ARMhas incorporated hardware debug technology within the processor so that software engineers can view what is happening while the processor is executing code. With greater visibility, software engineers can resolve issues faster, which has a direct effect on the time to market and reduces overall development costs.

The ARM core is not a pure RISC architecture because of the constraints of its primary application—the embedded system. In some sense, the strength of the ARM core is that it does not take the RISC concept too far. In today's systems the key is not raw processor speed but total effective system performance and power consumption.

Q5. [c]:

Explain the factors that make ARM instruction set suitable for embedded Applications.

Sol:

Instruction Set for Embedded Systems:

The ARM instruction set differs from the pure RISC definition in several ways that make the ARM instruction set suitable for embedded applications:

■ *Variable cycle execution for certain instructions*—Not every ARM instruction executes in a single cycle. For example, load-store-multiple instructions vary in the number of execution cycles depending upon the number of registers being transferred. The transfer can occur on sequential memory addresses, which increases performance since sequential memory accesses are often faster than random accesses. Code density is also mproved since multiple register transfers are common operations at the start and end of functions.

■ *Inline barrel shifter leading to more complex instructions*—The inline barrel shifter is a hardware component that preprocesses one of the input registers before it is used by an instruction. This expands the capability of many instructions to improve core performance and code density.

■ *Thumb 16-bit instruction set*—ARM enhanced the processor core by adding a second 16-bit instruction set called Thumb that permits the ARM core to execute either 16- or 32-bit instructions. The 16-bit instructions improve code density by about 30% over 32-bit fixed-length instructions.

■ *Conditional execution*—An instruction is only executed when a specific condition has been satisfied. This feature improves performance and code density by reducing branch instructions.

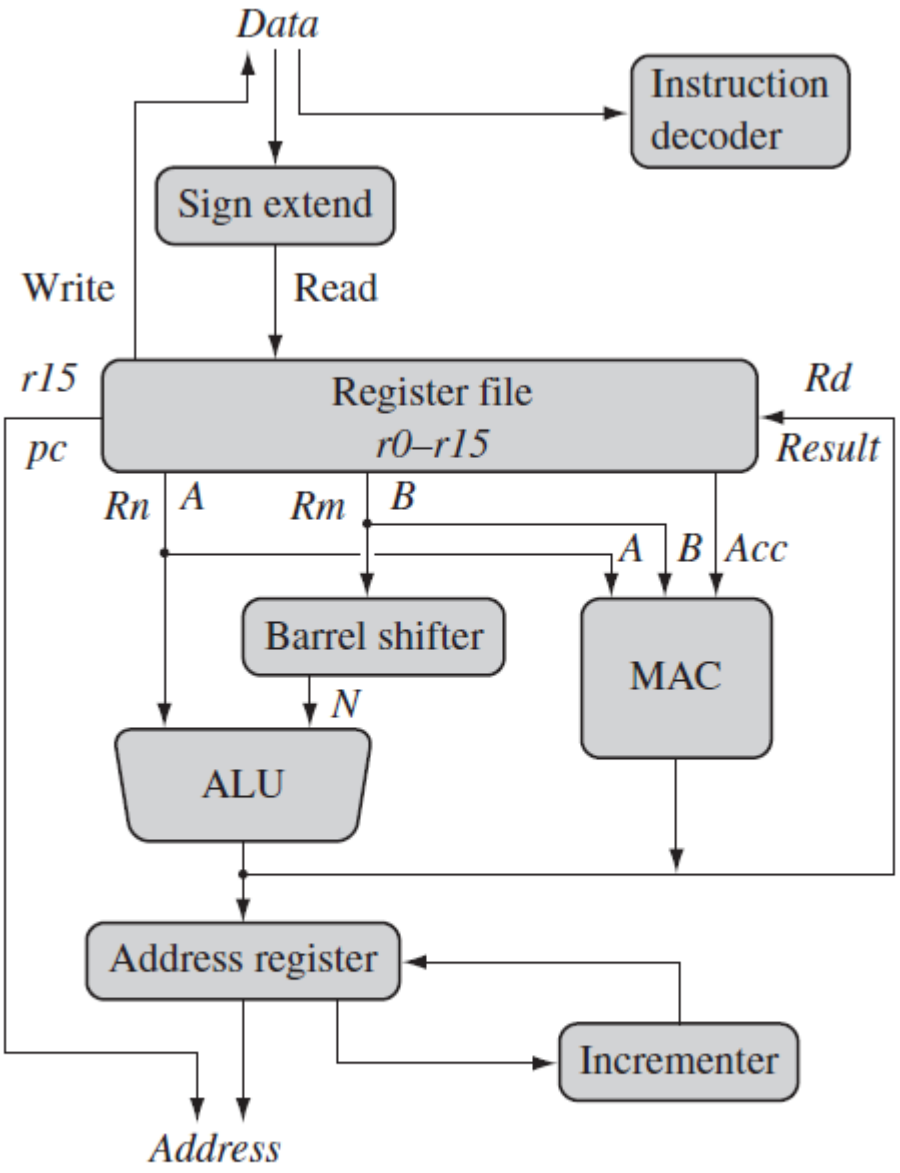
■ *Enhanced instructions*—The enhanced digital signal processor (DSP) instructions were added to the standard ARM instruction set to support fast 16×16-bit multiplier operations and saturation. These instructions allow a faster-performing ARM processor in some cases to replace the traditional combinations of a processor plus a DSP.

These additional features have made the ARM processor one of the most commonly used 32-bit embedded processor cores.

Q6. [a]

Explain ARM core data flow model with a neat diagram

Sol:



ARM core dataflow model.

A programmer can think of an ARM core as functional units connected by data buses, as shown in Figure above, 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 figure shows not only the flow of data but also the abstract components that make up an ARM core.

Data enters the processor core through the *Data* bus. The data may be an instruction to execute or a data item. Figure 2.1 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 a *load-store architecture*. This 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, and conversely the 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 solely 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.

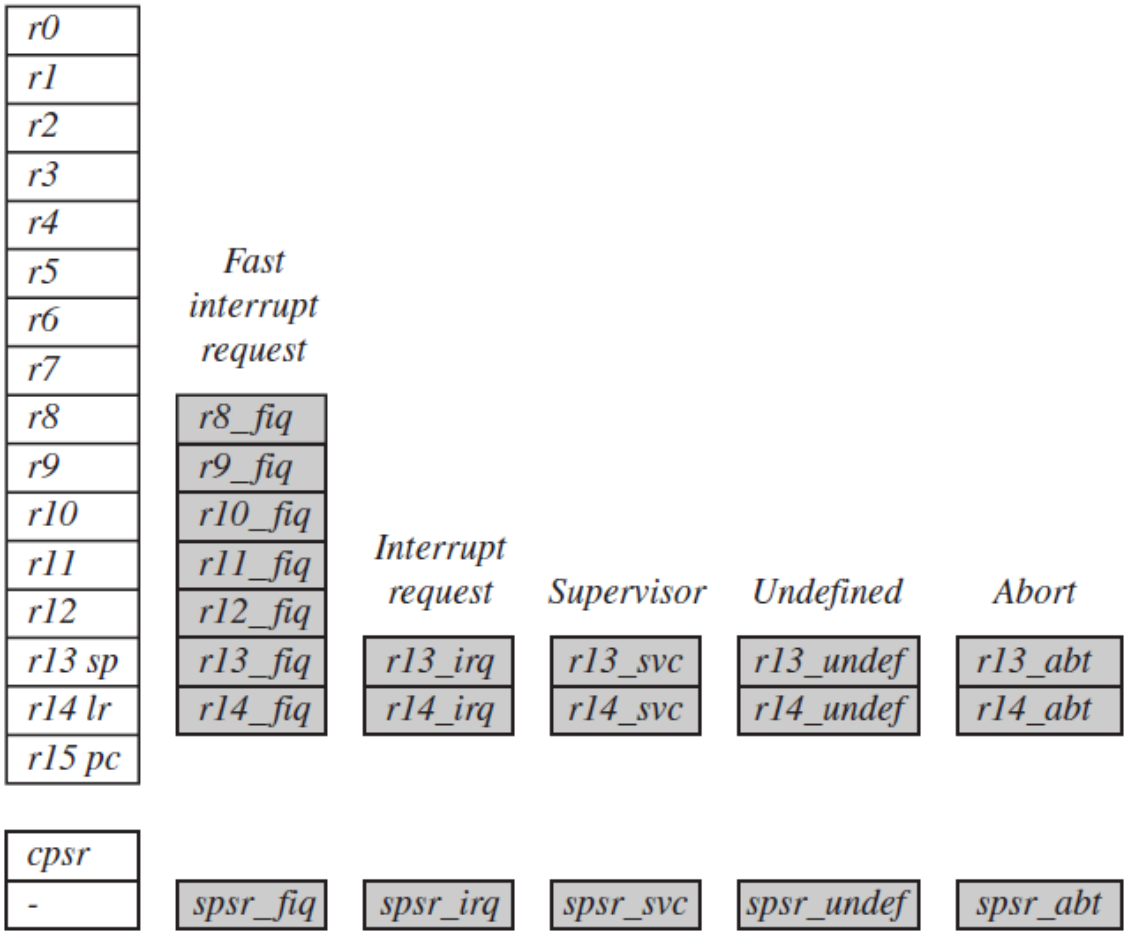
After passing through the functional units, the result in *Rd* is written back to the register file using the *Result* bus. For load and store instructions the incrementer updates the address register before the core reads or writes the next register value from or to the next sequential memory location. The processor continues executing instructions until an exception or interrupt changes the normal execution flow.

Q6.[a]:

Explain the different processor modes provided by ARM-7

Sol:

Processor Modes: (Complete Register Set with Processor Modes is shown in figure below)



Complete ARM register set.

The processor mode determines which registers are active and the access rights to the *cpsr* register itself. Each processor mode is either privileged or nonprivileged: A privileged mode allows full read-write access to the *cpsr*. Conversely, a nonprivileged mode only allows read access to the control field in the *cpsr* but still allows read-write access to the condition flags.

There are seven processor modes in total: six privileged modes (*abort*, *fast interrupt request*, *interrupt request*, *supervisor*, *system*, and *undefined*) and one nonprivileged mode (*user*).

The processor enters *abort* mode when there is a failed attempt to access memory. *Fast interrupt request* and *interrupt request* modes correspond to the two interrupt levels available on the ARM processor. *Supervisor* mode is the mode that the processor is in after reset and is generally the mode that an operating system kernel operates in. *System* mode is a special version of *user* mode that allows full read-write access to the *cpsr*. *Undefined* mode is used when the processor encounters an instruction that is undefined or not supported by the implementation. *User* mode is used for programs and applications.

Figure shows all 37 registers in the register file. Of those, 20 registers are hidden from a program at different times. These registers are called *banked registers* and are identified by the shading in the diagram. They are available only when the processor is in a particular mode; for example, *abort* mode has banked registers *r13_abt*, *r14_abt* and *spsr_abt*. Banked registers of a particular mode are denoted by an underline character post-fixed to the mode mnemonic or *_mode*.

Every processor mode except *user* mode can change mode by writing directly to the mode bits of the *cpsr.* All processor modes except *system* mode have a set of associated banked registers that are a subset of the main 16 registers. A banked register maps one-to one onto a *user* mode register. If you change processor mode, a banked register from the new mode will replace an existing register.

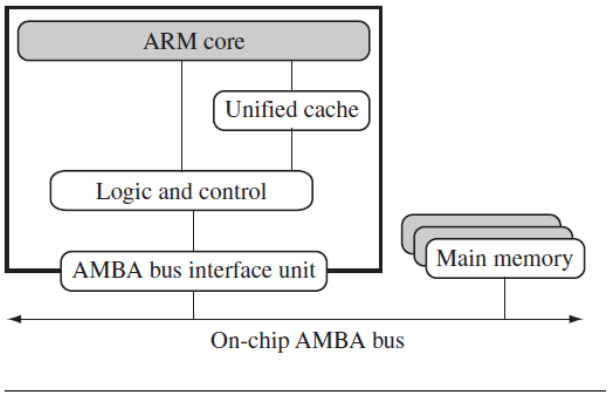
Q6. [b]:

1. Discuss with a neat diagram Von-Neumann architecture with cache
2. Hardware architecture with TCM

Sol:

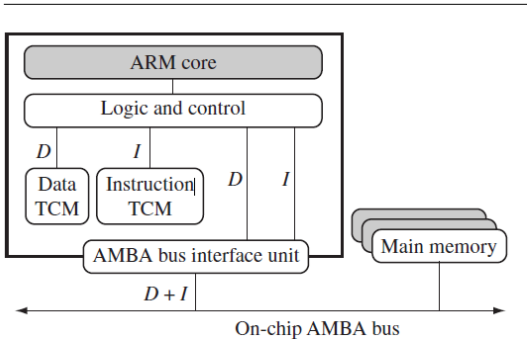
The cache is a block of fast memory placed between main memory and the core. It allows for more efficient fetches from some memory types. With a cache the processor core can run for the majority of the time without having to wait for data from slow external memory.

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(A) below . 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.

(FIGURE -A)



A simplified Harvard architecture with TCMs.

FIGURE- B

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-B

Q-7 [a]:

Explain with neat diagram barrel shifter operation in ARM Processor.

Sol:

Barrel Shifter

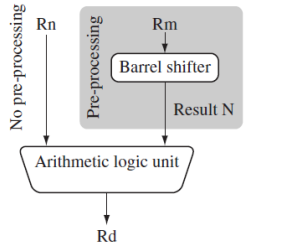
In Example 3.1 we showed a MOV instruction where *N* is a simple register. But *N* can be more than just a register or immediate value; it can also be a register *Rm* that has been preprocessed by the barrel shifter prior to being used by a data processing instruction.

Data processing instructions are processed within the arithmetic logic unit (ALU).

A unique and powerful feature of the ARM processor is the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU. This shift increases the power and flexibility of many data processing operations.

There are data processing instructions that do not use the barrel shift, for example, the MUL (multiply), CLZ (count leading zeros), and QADD (signed saturated 32-bit add) instructions.

Pre-processing or shift occurs within the cycle time of the instruction. This is particularly useful for loading constants into a register and achieving fast multiplies or division by a power of 2.



Barrel shifter and ALU.

Barrel shifter and ALU.

To illustrate the barrel shifter we will take the example in Figure and add a shift operation to the move instruction example. Register *Rn* enters the ALU without any preprocessing of registers. Figure shows the data flow between the ALU and the barrel shifter.

Example

We apply a logical shift left (LSL) to register *Rm* before moving it to the destination register. This is the same as applying the standard C language shift operator ≪ to the register. The MOV instruction copies the shift operator result *N* into register *Rd*. *N* represents the result of the LSL operation described in Table below.

PRE r5 = 5

r7 = 8

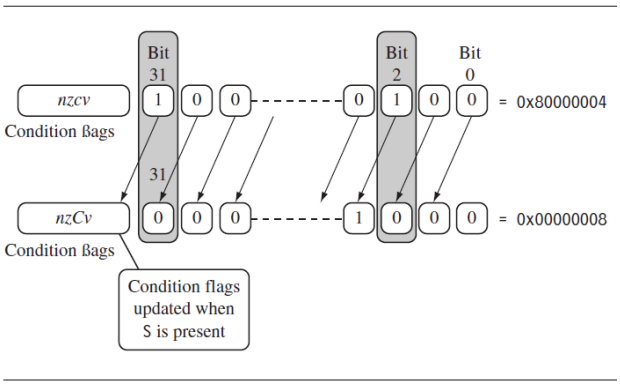
MOV r7, r5, LSL #2 ; let r7 = r5*4 = (r5 << 2)

POST r5 = 5

r7 = **20**

The example multiplies register *r5* by four and then places the result into register *r7*. The five different shift operations that you can use within the barrel shifter are summarized in Table below.

Figure below illustrates a logical shift left by one. For example, the contents of bit 0 are shifted to bit 1. Bit 0 is cleared. The *C* flag is updated with the last bit shifted out of the register. This is bit (32−*y*) of the original value, where *y* is the shift amount. When *y* is greater than one, then a shift by *y* positions is the same as a shift by one position executed *y* times.



Logical shift left by one.

Q7. [b]:

Explain with an example the concept of Semaphore using Swap Instruction

Sol:

A semaphore is a variable that controls access to a shared resource in an operating system. Semaphores can be used to prevent multiple threads or processes from accessing a shared resource at the same time

Use of SWP and SWPB semaphore instructions:

The ARM instruction set includes two semaphore instructions, Swap (SWP) and Swap Byte (SWPB), that are provided for process synchronization. Both instructions generate a load access and a store access to

the same memory location, such that no other access to that location is permitted between the load access and the store access. This enables a memory semaphore to be loaded and altered without interruption. These semaphore instructions do not provide a compare and conditional write facility. If this is required, it must be done explicitly.

The ARM architecture has always had the SWP instruction for implementing semaphores to ensure consistency in such environments. As the SoC has become more complex, however, certain aspects of SWP cause a performance bottleneck in some instances. Recall that SWP is basically a "blocking" primitive that locks the external bus of the processor and uses most of its bandwidth just to wait for a resource to be released. In this sense the SWP instruction is considered "pessimistic"—no computation can continue until SWP returns with the freed resource.

Swap Instruction:

The swap instruction is a special case of a load-store instruction. It swaps the contents of memory with the contents of a register. This instruction is an *atomic operation*—it reads and writes a location in the same bus operation, preventing any other instruction from reading or writing to that location until it completes.

```
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 cannot be interrupted by any other instruction or any other bus access. We say the system "holds the bus" until the transaction is complete.

The swap instruction loads a word from memory into register *r0* and overwrites 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
```

This instruction is particularly useful when implementing semaphores and mutual exclusion in an operating system. You can see from the syntax that this instruction can also have a byte size qualifier B, so this instruction allows for both a word and a byte swap.

This example shows a simple data guard that can be used to protect data from being written by another task. The SWP instruction "holds the bus" until the transaction is complete.

```
spin
        MOV    r1, =semaphore
        MOV    r2, #1
        SWP    r3, r2, [r1] ; hold the bus until complete
        CMP    r3, #1
        BEQ    spin
```

The address pointed to by the semaphore either contains the value 0 or 1. When the semaphore equals 1, then the service in question is being used by another process. The routine will continue to loop around until the service is released by the other process—in other words, when the semaphore address location contains the value 0.

Q7. [c]: Develop an assembly language program to multiply two 16 bit nos.

Sol:

Program to multiply two 16 bit numbers

R1 = 0x1234 = 0b0001 0010 0011 0100 LDR r1, =0x1234

R2 = 0x2345 = 0b0010 0011 0100 0101 LDR r2, =0x2345

R3 = R1 * R2 MUL r3, r1, r2

0x1234 * 0x2345

5B04

48D0x

369Cxx

2468xxx

2820404

```
    AREA MULTIPLY, CODE, READONLY
ENTRY
START
    LDR r1, =0x1234
    LDR r2, =0x2345
    MUL r3, r1, r2
    NOP
    NOP
    END
```

8. Explain the following with example:

i) MSR ii) MVN iii) TST iv) BIC

- MSR
  - Transter the contents of a *register* into the *cpsr* or *spsr*

- Syntax
  - MRS{<cond>} Rd, <cpsr | spsr>
  - MSR{<cond>} <cpsr | spsr>_<fields>, Rm
  - MSR{<cond>} <cpsr | spsr>_<fields>, #immediate
- Field: any combination of
  - Flags: [24:31]
  - Status: [16:23]
  - eXtension[8:15]
  - Control[0:7]

- MVN : move (negated)
  - MVN r0, r1; r0 = NOT(r1)=~ (r1)

- TST : bit-wise AND test
  - TST r0, r1; compute (r0 AND r1)and set NZCV

- TST : bit-wise AND test
  - TST r0, r1; compute (r0 AND r1)and set NZCV

- TEQ : bit-wise exclusive-or test
  - TEQ r0, r1; compute (r0 EOR r1)and set NZCV

- BIC : bit clear
  - BIC r0, r1, r2; r0 = r1 & Not(r2)

- PRE: r1 = 0b1111, r2 = 0b0101
- BIC r0, r1, r2 ; r0 = r1 AND (NOT(r2))
- POST: r0=0b1010

8.b Explain with an example forward and backward branch.

Branch Instructions (Cont.)

- Syntax
  - B{<cond>} lable
  - BL{<cond>} lable
- B : branch
  - B label; pc (program counter) = label
  - Used to change execution flow
- BL : branch and link
  - BL label; pc = label, lr = address of the next address after the BL
  - Similar to the B instruction but can be used for subroutine call
    - Overwrite the link register (lr) with a return address

- ***Example 5***

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

8c. Develop an assembly language program to find GCD of two numbers using conditional execution.

```
        ; Greatest Common Divisor Algorithm
gcd
        CMP    r1, r2
        BEQ    complete
        BLT    lessthan
        SUB    r1, r1, r2
        B      gcd

lessthan
        SUB    r2, r2, r1
        B      gcd

complete
...
```

9a. Discuss with an example code density in thumb instruction set over ARM.

This chapter introduces the Thumb instruction set. Thumb encodes a subset of the 32-bit ARM instructions into a 16-bit instruction set space. Since Thumb has higher performance than ARM on a processor with a 16-bit data bus, but lower performance than ARM on a 32-bit data bus, use Thumb for memory-constrained systems.

Thumb has higher *code density*—the space taken up in memory by an executable program—than ARM. For memory-constrained embedded systems, for example, mobile phones and PDAs, code density is very important. Cost pressures also limit memory size, width, and speed.

On average, a Thumb implementation of the same code takes up around 30% less memory than the equivalent ARM implementation. As an example, Figure 4.1 shows the same divide code routine implemented in ARM and Thumb assembly code. Even though the Thumb implementation uses more instructions, the overall memory footprint is reduced. Code density was the main driving force for the Thumb instruction set. Because it was also designed as a compiler target, rather than for hand-written assembly code, we recommend that you write Thumb-targeted code in a high-level language like C or C++.

Each Thumb instruction is related to a 32-bit ARM instruction. Figure 4.2 shows a simple Thumb `ADD` instruction being decoded into an equivalent ARM `ADD` instruction.

Table 4.1 provides a complete list of Thumb instructions available in the THUMBv2 architecture used in the ARMv5TE architecture. Only the branch relative instruction can be conditionally executed. The limited space available in 16 bits causes the barrel shift operations `ASR`, `LSL`, `LSR`, and `ROR` to be separate instructions in the Thumb ISA.

ARM code

```
ARMDivide
; IN:  r0(value),r1(divisor)
; OUT: r2(MODulus),r3(DIVide)

        MOV     r3,#0
loop
        SUBS    r0,r0,r1
        ADDGE   r3,r3,#1
        BGE     loop
        ADD     r2,r0,r1
```

$5 \times 4 = 20$ bytes

Thumb code

```
ThumbDivide
; IN:  r0(value),r1(divisor)
; OUT: r2(MODulus),r3(DIVide)

        MOV     r3,#0
loop
        ADD     r3,#1
        SUB     r0,r1
        BGE     loop
        SUB     r3,#1
        ADD     r2,r0,r1
```

$6 \times 2 = 12$ bytes

Figure 4.1 Code density.

9 b Explain ARM thumb interworking.

ARM-Thumb Interworking

ARM-Thumb interworking is the name given to the method of linking ARM and Thumb code together for both assembly and C/C++. It handles the transition between the two states. Extra code, called a *veneer*, is sometimes needed to carry out the transition. ATPCS defines the ARM and Thumb procedure call standards.

To call a Thumb routine from an ARM routine, the core has to change state. This state change is shown in the *T* bit of the *cpsr*. The BX and BLX branch instructions cause a switch between ARM and Thumb state while branching to a routine. The BX *lr* instruction returns from a routine, also with a state switch if necessary.

The BLX instruction was introduced in ARMv5T. On ARMv4T cores the linker uses a veneer to switch state on a subroutine call. Instead of calling the routine directly, the linker calls the veneer, which switches to Thumb state using the BX instruction.

There are two versions of the BX or BLX instructions: an ARM instruction and a Thumb equivalent. The ARM BX instruction enters Thumb state only if bit 0 of the address in *Rn* is set to binary 1; otherwise it enters ARM state. The Thumb BX instruction does the same.

```
Syntax: BX     Rm
        BLX    Rm | label
```

BX	Thumb version branch exchange	*pc* = *Rn* & 0xfffffffe *T* = *Rn*[0]
BLX	Thumb version of the branch exchange with link	*lr* = (instruction address after the BLX) + 1 *pc* = *label*, *T* = 0 *pc* = *Rm* & 0xfffffffe, *T* = *Rm*[0]

Unlike the ARM version, the Thumb BX instruction cannot be conditionally executed.

9c. Explain with example thumb stack operations.

4.7 Stack Instructions

The Thumb stack operations are different from the equivalent ARM instructions because they use the more traditional POP and PUSH concept.

```
Syntax: POP {low_register_list{, pc}}
        PUSH {low_register_list{, lr}}
```

POP	pop registers from the stacks	Rd*N <- mem32[sp+4*N], sp=sp+4*N
PUSH	push registers on to the stack	Rd*N -> mem32[sp+4*N], sp=sp-4*N

The interesting point to note is that there is no stack pointer in the instruction. This is because the stack pointer is fixed as register *r13* in Thumb operations and *sp* is automatically updated. The list of registers is limited to the low registers *r0* to *r7*.

The PUSH register list also can include the link register *lr*; similarly the POP register list can include the *pc*. This provides support for subroutine entry and exit, as shown in Example 4.7.

The stack instructions only support full descending stack operations.

EXAMPLE 4.7 In this example we use the POP and PUSH instructions. The subroutine ThumbRoutine is called using a branch with link (BL) instruction.

```
        ; Call subroutine
        BL      ThumbRoutine
        ; continue

ThumbRoutine
        PUSH    {r1, lr}      ; enter subroutine
        MOV     r0, #2
        POP     {r1, pc}      ; return from subroutine
```

10a. Explain with an example the effect of using “char” and “short” as local variable types in ARM processors.

5.2.1 Local Variable Types

ARMv4-based processors can efficiently load and store 8-, 16-, and 32-bit data. However, most ARM data processing operations are 32-bit only. For this reason, you should use a 32-bit datatype, `int` or `long`, for local variables wherever possible. Avoid using `char` and `short` as local variable types, even if you are manipulating an 8- or 16-bit value. The one exception is when you want wrap-around to occur. If you require modulo arithmetic of the form $255 + 1 = 0$, then use the `char` type.

To see the effect of local variable types, let's consider a simple example. We'll look in detail at a checksum function that sums the values in a data packet. Most communication protocols (such as TCP/IP) have a checksum or cyclic redundancy check (CRC) routine to check for errors in a data packet.

The following code checksums a data packet containing 64 words. It shows why you should avoid using `char` for local variables.

```
int checksum_v1(int *data)
{
  char i;
  int sum=0;

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

At first sight it looks as though declaring `i` as a `char` is efficient. You may be thinking that a `char` uses less register space or less space on the ARM stack than an `int`. On the ARM, both these assumptions are wrong. All ARM registers are 32-bit and all stack entries are at least 32-bit. Furthermore, to implement the `i++` exactly, the compiler must account for the case when `i` = 255. Any attempt to increment 255 should produce the answer 0.

10.b List the C compiler data type mapping for an ARM target with their implementation.

Table 5.2 C compiler datatype mappings.

C Data Type	Implementation
`char`	unsigned 8-bit byte
`short`	signed 16-bit halfword
`int`	signed 32-bit word
`long`	signed 32-bit word
`long long`	signed 64-bit double word

10. C With an example, compare the efficiencies of signed int and unsigned int with an example.

Here is the last version of the 64-word packet checksum routine we studied in Section 5.2. This shows how the compiler treats a loop with incrementing count `i++`.

```
int checksum_v5(int *data)
{
  unsigned int i;
  int sum=0;

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

This compiles to

```
checksum_v5
        MOV     r2,r0             ; r2 = data
        MOV     r0,#0             ; sum = 0
        MOV     r1,#0             ; i = 0
checksum_v5_loop
        LDR     r3,[r2],#4        ; r3 = *(data++)
        ADD     r1,r1,#1          ; i++
        CMP     r1,#0x40          ; compare i, 64
        ADD     r0,r3,r0          ; sum += r3
        BCC     checksum_v5_loop  ; if (i<64) goto loop
        MOV     pc,r14            ; return sum
```

It takes three instructions to implement the `for` loop structure:

- An `ADD` to increment `i`
- A compare to check if `i` is less than 64
- A conditional branch to continue the loop if `i` < 64

This is not efficient. On the ARM, a loop should only use two instructions:

- A subtract to decrement the loop counter, which also sets the condition code flags on the result
- A conditional branch instruction

The key point is that the loop counter should count down to zero rather than counting up to some arbitrary limit. Then the comparison with zero is free since the result is stored

in the condition flags. Since we are no longer using `i` as an array index, there is no problem in counting down rather than up.

EXAMPLE 5.2 This example shows the improvement if we switch to a decrementing loop rather than an incrementing loop.

```
int checksum_v6(int *data)
{
  unsigned int i;
  int sum=0;

  for (i=64; i!=0; i--)
  {
    sum += *(data++);
  }
  return sum;
}
```

This compiles to

```
checksum_v6
        MOV     r2,r0             ; r2 = data
        MOV     r0,#0             ; sum = 0
        MOV     r1,#0x40          ; i = 64
checksum_v6_loop
        LDR     r3,[r2],#4        ; r3 = *(data++)
        SUBS    r1,r1,#1          ; i-- and set flags
        ADD     r0,r3,r0          ; sum += r3
        BNE     checksum_v6_loop  ; if (i!=0) goto loop
        MOV     pc,r14            ; return sum
```

The `SUBS` and `BNE` instructions implement the loop. Our checksum example now has the minimum number of four instructions per loop. This is much better than six for `checksum_v1` and eight for `checksum_v3`.

For an unsigned loop counter `i` we can use either of the loop continuation conditions `i!=0` or `i>0`. As `i` can't be negative, they are the same condition. For a signed loop counter, it is tempting to use the condition `i>0` to continue the loop. You might expect the compiler to generate the following two instructions to implement the loop:

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

In fact, the compiler will generate

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

The compiler is not being inefficient. It must be careful about the case when `i = -0x80000000` because the two sections of code generate different answers in this case. For the first piece of code the SUBS instruction compares `i` with 1 and then decrements `i`. Since `-0x80000000` < 1, the loop terminates. For the second piece of code, we decrement `i` and then compare with 0. Modulo arithmetic means that `i` now has the value `+0x7fffffff`, which is greater than zero. Thus the loop continues for many iterations.

Of course, in practice, `i` rarely takes the value `-0x80000000`. The compiler can't usually determine this, especially if the loop starts with a variable number of iterations (see Section 5.3.2).

Therefore you should use the termination condition `i!=0` for signed or unsigned loop counters. It saves one instruction over the condition `i>0` for signed `i`.

Consider the compiler output for this function. We've added labels and comments to make the assembly clear.

```
checksum_v1
        MOV     r2,r0               ; r2 = data
        MOV     r0,#0               ; sum = 0
        MOV     r1,#0               ; i = 0
checksum_v1_loop
        LDR     r3,[r2,r1,LSL #2]   ; r3 = data[i]
        ADD     r1,r1,#1            ; r1 = i+1
        AND     r1,r1,#0xff         ; i = (char)r1
        CMP     r1,#0x40            ; compare i, 64
        ADD     r0,r3,r0            ; sum += r3
        BCC     checksum_v1_loop    ; if (i<64) loop
        MOV     pc,r14              ; return sum
```

Now compare this to the compiler output where instead we declare `i` as an `unsigned int`.

```
checksum_v2
        MOV     r2,r0               ; r2 = data
        MOV     r0,#0               ; sum = 0
        MOV     r1,#0               ; i = 0
checksum_v2_loop
        LDR     r3,[r2,r1,LSL #2]   ; r3 = data[i]
        ADD     r1,r1,#1            ; r1++
        CMP     r1,#0x40            ; compare i, 64
        ADD     r0,r3,r0            ; sum += r3
        BCC     checksum_v2_loop    ; if (i<64) goto loop
        MOV     pc,r14              ; return sum
```

In the first case, the compiler inserts an extra AND instruction to reduce `i` to the range 0 to 255 before the comparison with 64. This instruction disappears in the second case.

Next, suppose the data packet contains 16-bit values and we need a 16-bit checksum. It is tempting to write the following C code:

```
short checksum_v3(short *data)
{
  unsigned int i;
  short sum=0;

  for (i=0; i<64; i++)
  {
    sum = (short)(sum + data[i]);




 }
 return sum;
}
```

You may wonder why the `for` loop body doesn't contain the code

```
sum += data[i];
```

With *armcc* this code will produce a warning if you enable implicit narrowing cast warnings using the compiler switch `-W+n`. The expression `sum+data[i]` is an integer and so can only be assigned to a `short` using an (implicit or explicit) narrowing cast. As you can see in the following assembly output, the compiler must insert extra instructions to implement the narrowing cast:

```
checksum_v3
        MOV     r2,r0             ; r2 = data
        MOV     r0,#0             ; sum = 0
        MOV     r1,#0             ; i = 0
checksum_v3_loop
        ADD     r3,r2,r1,LSL #1   ; r3 = &data[i]
        LDRH    r3,[r3,#0]        ; r3 = data[i]
        ADD     r1,r1,#1          ; i++
        CMP     r1,#0x40          ; compare i, 64
        ADD     r0,r3,r0          ; r0 = sum + r3
        MOV     r0,r0,LSL #16
        MOV     r0,r0,ASR #16     ; sum = (short)r0
        BCC     checksum_v3_loop  ; if (i<64) goto loop
        MOV     pc,r14            ; return sum
```