

Computer Memory

Computer Memory can be classified basically into two types:

1. Primary Memory
2. Secondary Memory

Primary Memory:

It is also called as the Main Memory or Internal Memory. It stores the data for processing and the Program instructions. It is also the Immediate Access Memory. It is temporary memory and of limited capacity.

The standard base capacity for Primary memory starts from 640 KB, 1MB, 2MB, 8MB, 128MB, 256MB, 1GB, 2GB, 4GB etc (in multiples of 2). Depending on the needs of the user.

Functions of Primary Memory:

1. It holds the OS instructions while the computer is booting.
2. It temporarily holds the input instructions from the input devices while the data is being input and processed.
3. It stores the results temporarily until it is transferred to the respective output devices.

Types of Primary Memory

1. RAM – Random Access Memory
 - a. Static RAM
 - b. Dynamic RAM
2. ROM – Read Only Memory
 - a. PROM – Programmable Read Only Memory
 - b. EPROM – Erasable Programmable Read Only Memory
 - c. EEPROM – Electronic Erasable Programmable Read Only Memory

1. RAM – Random Access Memory

It is a read/write memory. Data can be stored by addressing one RAM cell. The data and programming instructions fed via the input device are stored in the RAM temporarily until that program is used. RAM may be updated.

There are 2 types of RAM

- a. Static RAM – It retains the stored information until the computer is working.
- b. Dynamic RAM – It retains stored information only until the program is working and loses the information once the program execution is completed.

2. ROM – Read Only Memory

It is a permanent memory. The instructions can only be read by the computer. The instructions related to system operations are stored here. These instructions are written by the manufacturer and cannot be edited by the user. When the system is turned ON, ROM instructions are instantly executed and used in operation of all I/O devices.

There are 3 types of ROM

a. **PROM** - Programmable Read Only Memory

It is a non-volatile memory. Instructions can be written once by the programmer and then subsequently read.

b. **EPROM** – Erasable Programmable Read Only Memory

It is an improvement over the PROM chips. Instructions can be rewritten by the programmer using special techniques. The instructions are erased using UV light and rewritten. To change the instructions the chip has to be removed from the machine and then put back after the changes have been made.

c. **EEPROM** – Electronic Erasable Programmable Read Only Memory

It is advancement over EPROM, and the chip need not be taken out from the machine. Instead the programming is done using software. These chips are used in Point-of-sale (POS) terminals to record price related information and can be updated as and when needed. However they are expensive compared to the normal ROM chips.

Secondary Memory:

Secondary Memory is memory which stores the data and programming instructions permanently in the computer system for future use. It could be placed within the computer or connected externally. Secondary memory is also called Backing Storage Devices or External Storage Devices. The storage capacity of Secondary Memory is huge and it is permanent in nature. It can also be regularly updated. Data from the Secondary Memory is transferred to the Primary Memory before execution. It is held there temporarily until the execution process is completed and then transferred to the Secondary Memory.

Types of Secondary Memory

1. Magnetic Tape
2. Magnetic Disk [Floppy Disk, Hard Disk etc]
3. Magnetic Drum
4. MICR [Magnetic Ink Character Reader]
5. CD-ROMs
6. Pen Drives

The Role of Memory

The term "**memory**" applies to any electronic component capable of temporarily storing data. There are two main categories of memories:

- **internal memory** that temporarily memorises data while programs are running. Internal memory uses microconductors, i.e. fast specialized electronic circuits. Internal memory corresponds to what we call random access memory (RAM).
- **auxiliary memory** (also called *physical memory* or *external memory*) that stores information over the long term, including after the computer is turned off. Auxiliary memory corresponds to magnetic storage devices such as the hard drive, optical storage devices such as CD-ROMs and DVD-ROMs, as well as read-only memories.

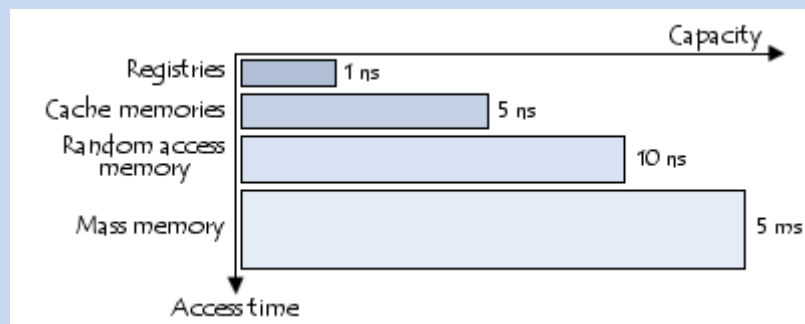
Technical Characteristics

The main characteristics of a memory are:

- **Capacity**, representing the global volume of information (in bits) that the memory can store
- **Access time**, corresponding to the time interval between the read/write request and the availability of the data
- **Cycle time**, representing the minimum time interval between two successive accesses
- **Throughput**, which defines the volume of information exchanged per unit of time, expressed in bits per second
- **Non-volatility**, which characterises the ability of a memory to store data when it is not being supplied with electricity

The ideal memory has a large capacity with restricted access time and cycle time, a high throughput and is non-volatile.

However, fast memories are also the most expensive. This is why memories that use different technologies are used in a computer, interfaced with each other and organized hierarchically.



The fastest memories are located in small numbers close to the processor. Auxiliary memories, which are not as fast, are used to store information permanently.

Types of Memories

Random Access Memory

Random access memory, generally called **RAM** is the system's main memory, i.e. it is a space that allows you to temporarily store data when a program is running.

Unlike data storage on an auxiliary memory such as a hard drive, RAM is volatile, meaning that it only stores data as long as it supplied with electricity. Thus, each time the computer is turned off, all the data in the memory are irremediably erased.

Read-Only Memory

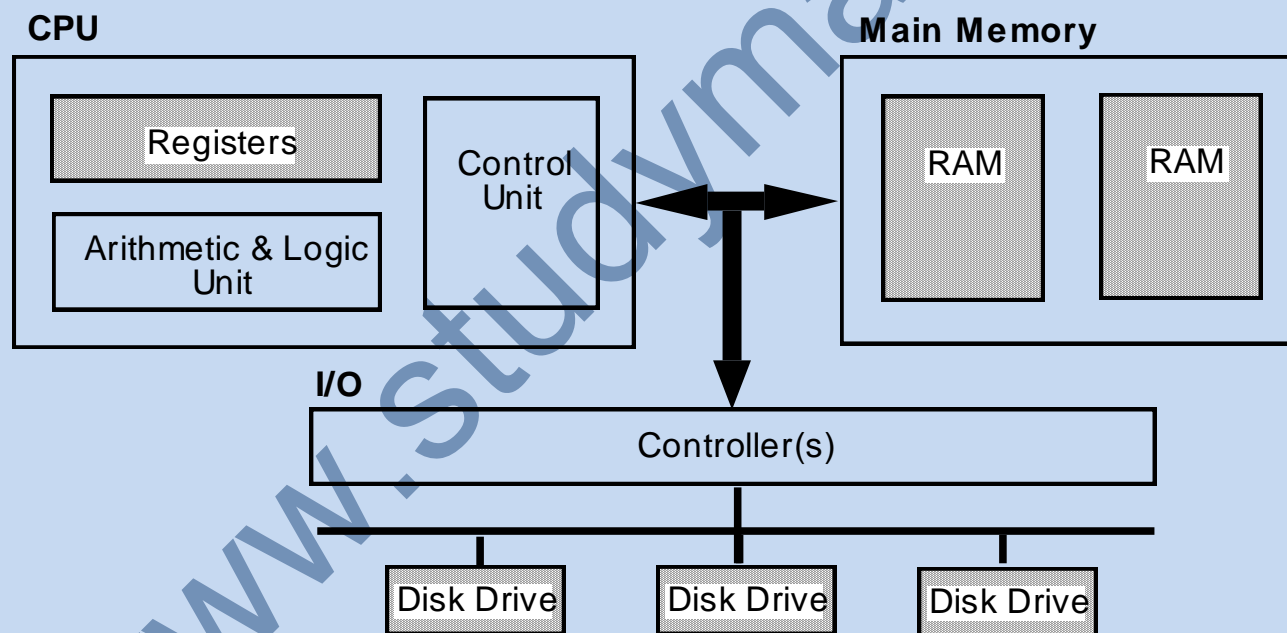
Read-only memory, called **ROM**, is a type of memory that allows you to keep the information contained on it even when the memory is no longer receiving electricity. Basically, this type of memory only has read-only access. However, it is possible to save information in some types of *ROM* memory.

Flash Memory

Flash memory is a compromise between RAM-type memories and ROM memories. Flash memory possesses the non-volatility of ROM memories while providing both read and write access. However, the access times of flash memories are longer than the access times of RAM.

Main Memory (RAM) Organisation

Computers employ many different types of memory (semi-conductor, magnetic disks and tapes, DVDs etc.) to hold data and programs. Each type has its own characteristics and uses. We will look at the way that Main Memory (RAM) is organised and very briefly at the characteristics of Register Memory and Disk Memory. Let's locate these 3 types of memory in an abstract computer:



Register Memory

Registers are memories located within the Central Processing Unit (CPU). They are few in number (there are rarely more than 64 registers) and also small in size, typically a register is less than 64 bits; 32-bit and more recently 64-bit are common in desktops.

The contents of a register can be “read” or “written” very quickly¹ however, often an order of magnitude faster than main memory and several orders of magnitude faster than disk memory.

Different kinds of register are found within the CPU. General Purpose Registers² are available for general³ use by the programmer. Unless the context implies otherwise we’ll use the term “register” to refer to a General Purpose Register within the CPU. Most modern CPU’s have between 16 and 64 general purpose registers. Special Purpose Registers have specific uses and are either non-programmable and internal to the CPU or accessed with special instructions by the programmer. Examples of such registers that we will encounter later in the course include: the Program Counter register (PC), the Instruction Register (IR), the ALU Input & Output registers, the Condition Code (Status/Flags) register, the Stack Pointer register (SP). The size (the number of bits in the register) of these registers varies according to register type. The Word Size of an architecture is often (but not always!) defined by the size of the general purpose registers.

In contrast to main memory and disk memory, registers are referenced directly by specific instructions or by encoding a register number within a computer instruction. At the programming (assembly) language level of the CPU, registers are normally specified with special identifiers (e.g. R0, R1, R7, SP, PC)

As a final point, the contents of a register are lost if power to the CPU is turned off, so registers are unsuitable for holding long-term information or information that is needed for retention after a power-shutdown or failure. Registers are however, the fastest memories, and if exploited can result in programs that execute very quickly.

Main Memory (RAM)

If we were to sum all the bits of all registers within CPU, the total amount of memory probably would not exceed 5,000 bits. Most computational tasks undertaken by a computer require a lot more memory. Main memory

is the next⁴ fastest memory within a computer and is much larger in size. Typical main memory capacities for different kinds of computers are: PC 512MB⁵, fileserver 2GB, database server 8GB. Computer architectures also impose an architectural constraint on the maximum allowable RAM. This constraint is normally equal to 2^{WordSize} memory locations.

RAM⁶ (Random⁷ Access Memory) is the most common form of Main Memory. RAM is normally located on the motherboard and so is typically less than 12 inches from the CPU. ROM (Read Only Memory) is like RAM except that its contents cannot be overwritten and its contents are not lost if power is turned off (ROM is non-volatile).

Although slower than register memory, the contents of any location⁸ in RAM can still be “read” or “written” very quickly⁹. The time to read or write is referred to as the **access time** and is constant for all RAM locations.

In contrast to register memory, RAM is used to hold both program code (instructions) and data (numbers, strings etc). Programs are “loaded” into RAM from a disk prior to execution by the CPU.

Locations in RAM are identified by an **addressing scheme** *e.g.* numbering the bytes in RAM from 0 onwards¹⁰. Like registers, the contents of RAM are lost if the power is turned off.

Disk Memory

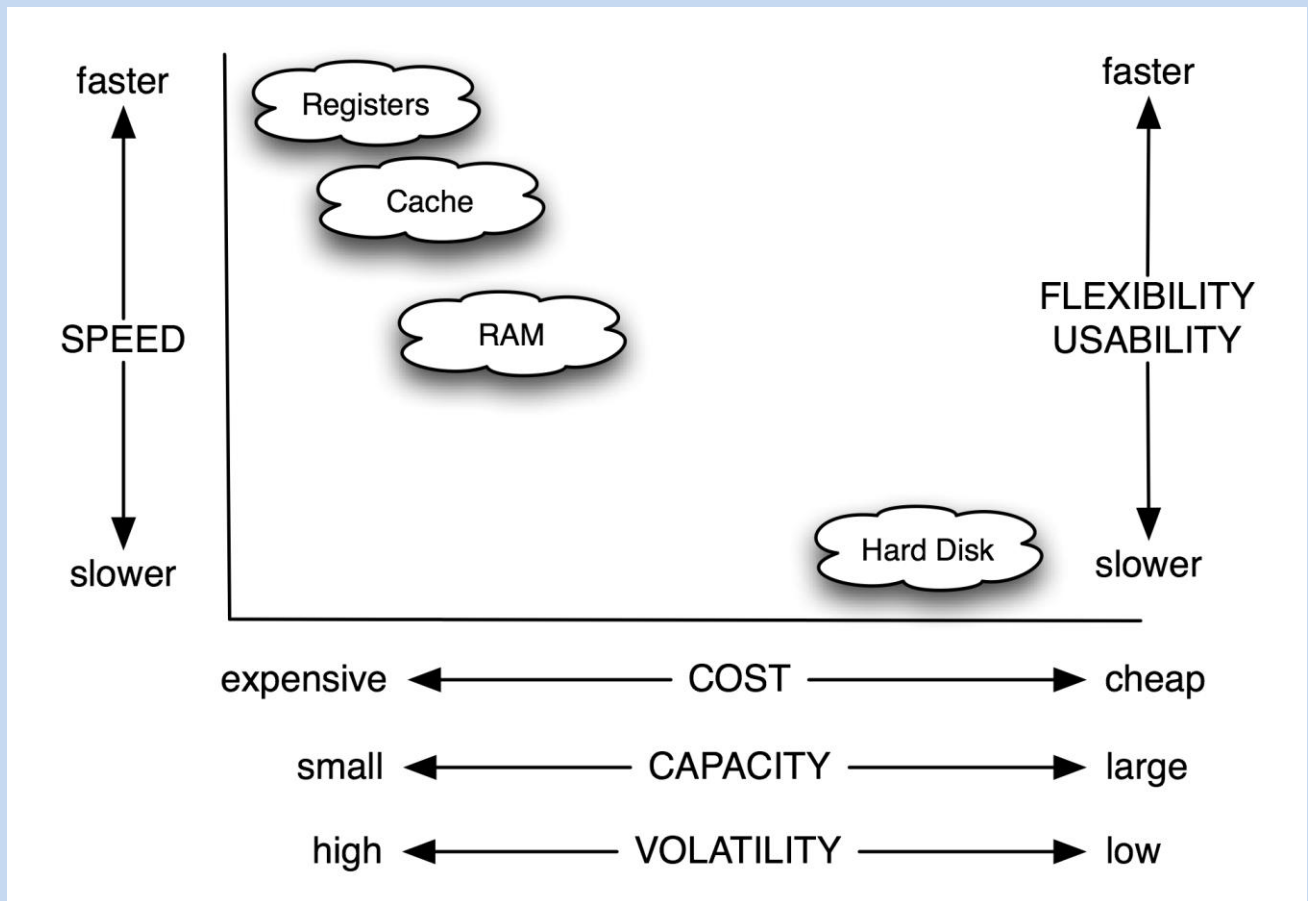
Disk memory¹¹ is used to hold programs and data over the longer term. The **contents of a disk are NOT lost if the power is turned off**. Typical hard disk capacities range from 40GB to over 500 GB (5×10^{29}). Disks are much slower than register and main memory, the access-time (known as the seek-time) to data on disk is typically between 2 and 4 milli-seconds, although disk drives can transfer thousands of bytes in one go achieving transfer rates from 25MB/s to 500MB/s.

Disks can be housed internally within a computer “box” or externally in an enclosure connected by a fast USB or firewire cable¹². Disk locations are identified by special disk addressing schemes (e.g. track and sector numbers).

Summary of Characteristics

¹¹ Some authors refer to disk memory as disk storage.

¹² For details about how disks and other storage devices work, check out Tanenbaum or Stallings.



SRAM, DRAM, SDRAM, DDR SDRAM

There are many kinds of RAM and new ones are invented all the time. One of aims is to make RAM access as fast as possible in order to keep up with the increasing speed of CPUs.

SRAM (Static RAM) is the fastest form of RAM but also the most expensive. Due to its cost it is not used as main memory but rather for cache memory. Each bit requires a 6-transistor circuit.

DRAM (Dynamic RAM) is not as fast as SRAM but is cheaper and is used for main memory. Each bit uses a single capacitor and single transistor circuit. Since capacitors lose their charge, DRAM needs to be refreshed every few milliseconds. The memory system does this transparently. There are many implementations of DRAM, two well-known ones are SDRAM and DDR SDRAM.

SDRAM (Synchronous DRAM) is a form of DRAM that is synchronised with the clock of the CPU's system bus, sometimes called the front-side bus (FSB). As an example, if the system bus operates at 167Mhz over an 8-byte (64-bit) data bus, then an SDRAM module could transfer $167 \times 8 \sim 1.3\text{GB/sec}$.

DDR SDRAM (Double-Data Rate DRAM) is an optimisation of SDRAM that allows data to be transferred on both the rising edge and falling edge of a clock signal. Effectively doubling the amount of data that can be transferred in a period of time. For example a PC-3200 DDR-SDRAM module operating at 200Mhz can transfer $200 \times 8 \times 2 \sim 3.2\text{GB/sec}$ over an 8-byte (64-bit) data bus.

ROM, PROM, EPROM, EEPROM, Flash

In addition to RAM, there are also a range of other semi-conductor memories that retain their contents when the power supply is switched off.

ROM (Read Only Memory) is a form of semi-conductor that can be written to once, typically in bulk at a factory. ROM was used to store the "boot" or start-up program (so called firmware) that a computer executes when powered on, although it has now fallen out-of-favour to more flexible

memories that support occasional writes. ROM is still used in systems with fixed functionalities, e.g. controllers in cars, household appliances etc.

PROM (Programmable ROM) is like ROM but allows end-users to write their own programs and data. It requires a special PROM writing equipment. Note: users can only write-once to PROM.

EPROM (Erasable PROM). With EPROM we can erase (using strong ultra-violet light) the contents of the chip and rewrite it with new contents, typically several thousand times. It is commonly used to store the “boot” program of a computer, known as the firmware. PCs call this firmware, the BIOS (Basic I/O System). Other systems use Open Firmware. Intel-based Macs use EFI (Extensible Firmware Interface).

EEPROM (Electrically Erasable PROM). As the name implies the contents of EEPROMs are erased electrically. EEPROMSs are also limited to the number of erase-writes that can be performed (e.g, 100,000) but support updates (erase-writes) to individual bytes whereas EPROM updates the whole memory and only supports around 10,000 erase-write cycles.

FLASH memory is a cheaper form of EEPROM where updates (erase-writes) can only be performed on blocks of memory, not on individual bytes. Flash memories are found in USB sticks, flash cards and typically range in size from 32M to 2GB. The number of erase/write cycles to a block is typically several hundred thousand before the block can no longer be written.

Address <———— 8 bit ————>

0						
1						
2						
3						
4						
5						
6						
7						
8						
9						
10						
11						

Address <----- 12 bit ----->

[illegible]

[illegible]

Address <----- 16 bit ----->

[illegible]

Byte Addressing

Main-memories generally store and recall rows, which are multi-byte in length (e.g. 16-bit word = 2 bytes, 32-bit word = 4 bytes). Many architectures, however, make main memory **byte-addressable** rather than **word addressable**. In such architectures the CPU and/or the main memory hardware is capable of reading/writing any individual byte. Here is an example of a main memory with 16-bit memory locations¹⁴. Note how the memory locations (rows) have even addresses.

Word Address	16 bit = 2 bytes															
0																
2																
4																
6																
8																
10																
12																
14																
16																
18																
20																

Byte Ordering

The ordering of bytes within a **multi-byte** data item defines the endianness of the architecture.

In **BIG-ENDIAN** systems the most significant byte of a multi-byte data item always has the lowest address, while the least significant byte has the highest address.

In **LITTLE-ENDIAN** systems, the least significant byte of a multi-byte data

item always has the lowest address, while the most significant byte has the highest address.

In the following example, table cells represent bytes, and the cell numbers indicate the address of that byte in main memory. Note: by convention we draw the bytes within a memory word left-to-right for big-endian systems, and right-to-left for little-endian systems.

Word Address	Big-Endian				Word Address	Little-Endian			
s									
0	0	1	2	3	0	3	2	1	0
4	4	5	6	7	4	7	6	5	4
8	8	9	10	11	8	11	10	9	8
12	12	13	14	15	12	15	14	13	12
	MSB —————>				LSB	MSB —————>			
—> LSB									

Note: an N-character ASCII string value is not treated as one large multi-byte value, but rather as N byte values, i.e. the first character of the string always has the lowest address, the last character has the highest address. This is true for both big-endian and little-endian. An N-character Unicode string would be treated as N two-byte value and each two-byte value would require suitable byte-ordering.

Example: Show the contents of memory at word address 24 if that word holds the number given by 122E 5F01H in both the big-endian and the little-endian schemes?

Big Endian					Little Endian				
	MS	—————		LS		MS	—————		LS
	B	—>		B		B	—>		B
	24	25	26	27		27	26	25	24
Word 24	12	2E	5F	01	Word 24	12	2E	5F	01

Example: Show the contents of main memory from word address 24 if those words hold the text JIM SMITH.

Big Endian					Little Endian				
	+0	+1	+2	+3		+3	+2	+1	+0
Word 24	J	I	M		Word 24		M	I	J
Word 28	S	M	I	T	Word 28	T	I	M	S
Word 32	H	?	?	?	Word 32	?	?	?	H

The bytes labelled with ? are unknown. They could hold important data, or they could be don't care bytes – the interpretation is left up to the programmer.

Unfortunately computer systems¹⁵, in use today are split between those that are big-endian, and those that are little-endian¹⁶. This leads to problems when a big-endian computer wants to transfer data to a little-endian computer. Some architectures, for example the PowerPC and ARM, allow the endian-ness of the architecture to be changed programmatically.

Word Alignment

Although main-memories are generally organised as byte-addressed rows of words and accessed a row at a time, some architectures, allow the CPU to access any word-sized bit-group regardless of its byte address. We say that accesses that begin on a memory word boundary are **aligned accesses** while accesses that do not begin on word boundaries are **unaligned accesses**.

Address Memory (16-bit)
word

0	MSB	LSB
2		
4		MSB
6	LSB	

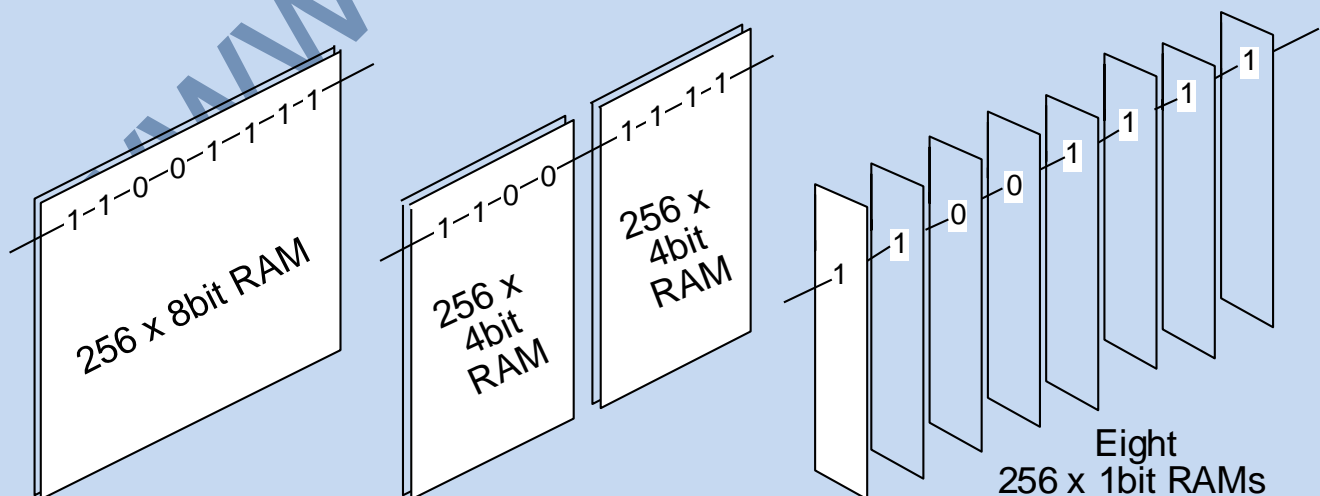
Word starting at Address 0 is Aligned

Word starting at Address 5 is Unaligned

Reading an unaligned word from RAM requires (i) reading of adjacent words, (ii) selecting the required bytes from each word and (iii) concatenating those bytes together => SLOW. Writing an unaligned word is more complex and slower¹⁷. For this reason some architectures prohibit unaligned word accesses. e.g. on the 68000 architecture, words must not be accessed starting from an odd-address (e.g. 1, 3, 5, 7 etc), on the SPARC architecture, 64-bit data items must have a byte address that is a multiple of 8.

Memory Modules, Memory Chips

So far, we have looked at the logical organisation of main memory. Physically RAM comes on small memory modules (little green printed circuit-boards about the size of a finger). A typical memory module holds 512MB to 2GB. The computer's motherboard will have slots to hold 2, 4 maybe 8 memory modules. Each memory module is itself comprised of several memory chips. For example here are 3 ways of forming a 256x8 bit memory module.

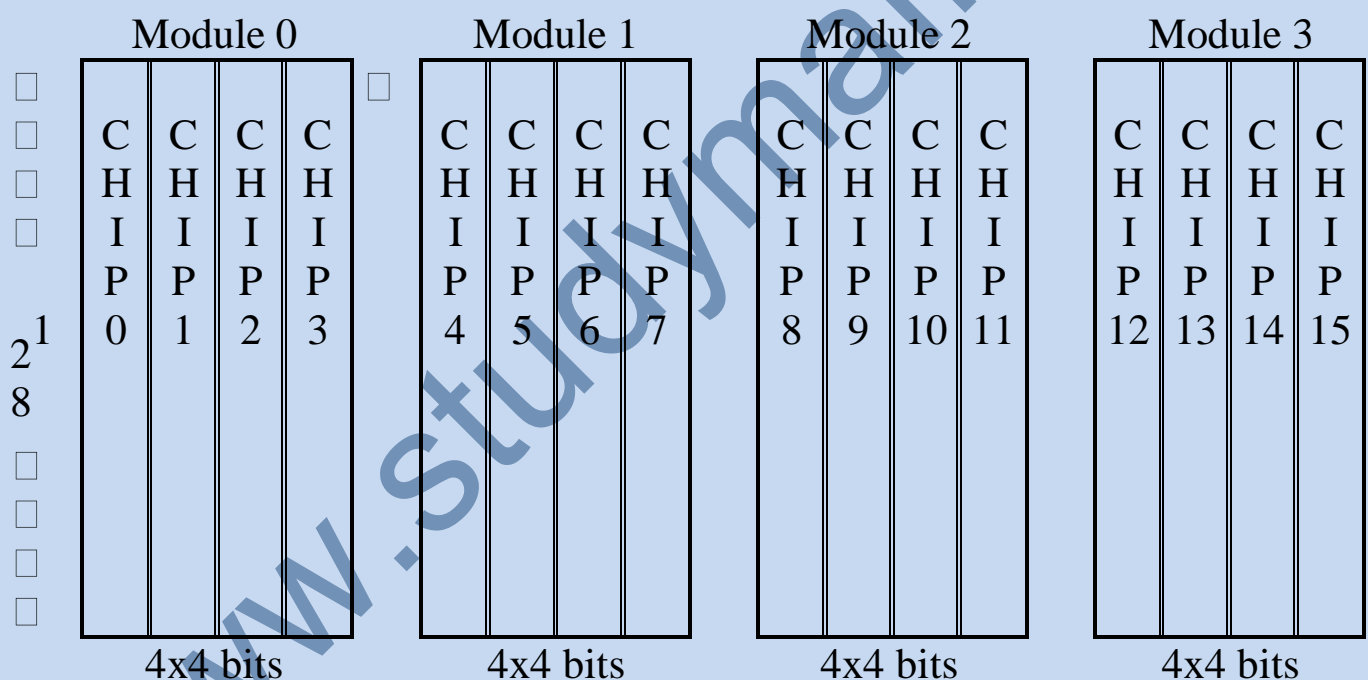


In the first case, main memory is built with a single memory chip. In the second, we use two memory chips, one gives us the most significant 4 bits, the other, the least significant 4 bits. In the third we use 8 memory chips, each chip gives us 1 bit - to read an 8 bit memory word, we would have to access all 8 memory chips simultaneously and concatenate the bits.

On PCs, memory modules are known as DIMMs (dual inline memory modules) and support 64-bit transfers. The previous generation of modules were called SIMMs (single inline memory modules) and supported 32-bit data transfers.

Example: Given Main Memory = 1M x 16 bit (word addressable),

RAM chips = 256K x 4 bit



$$\text{RAM chips per memory module} = \frac{\text{Width of Memory Word}}{\text{Width of RAM Chip}} = \frac{16}{4} = 4$$

18 bits are required to address a RAM chip (since $256K = 2^{18}$ = Length of RAM Chip)

A 1Mx16 bit word-addressed memory requires 20 address bits (since $1M = 2^{20}$)

Therefore 2 bits (=20–18) are needed to select a module.

The total number of RAM Chips = $(1M \times 16) / (256K \times 4) = 16$.

Total number of Modules = Total number of RAM chips /
RamChipsPerModule = $16/4 = 4$

Interleaved Memory

When memory consists of several memory modules, some address bits will select the module, and the remaining bits will select a row within the selected module.

When the module selection bits are the least significant bits of the memory address we call the resulting memory a **low-order interleaved** memory.

When the module selection bits are the most significant bits of the memory address we call the resulting memory a **high-order interleaved** memory.

Interleaved memory can yield performance advantages **if** more than one memory module can be read/written at a time:-

- (I) for low-order interleave if we can read the same row in each module. This is good for a single multi-word access of sequential data such as program instructions, or elements in a vector,
- (ii) for high-order interleave, if different modules can be independently accessed by different units. This is good if the CPU can access rows in one module, while at the same time, the hard disk (or a second CPU) can access different rows in another module.

Example: Given that Main Memory = 1Mx8bits, RAM chips = 256K x 4bit. For this memory we would require $4 \times 2 = 8$ RAM chips. Each chip would require 18 address bits (ie. $2^{18} = 256K$) and the full 1Mx16 bit memory would requires 20 address bits (ie. $2^{20} = 1M$)

www.studymafia.org

CPU Organisation & Operation

The Fetch-Execute Cycle

The operation of the CPU¹⁸ is usually described in terms of the Fetch-Execute cycle¹⁹.

Fetch-Execute Cycle	The cycle raises many interesting questions, e.g.
Fetch the <i>Instruction</i>	What is an Instruction? Where is the Instruction? Why does it need to be fetched? Isn't it okay where it is? How does the computer keep track of instructions? Where does it put the instruction it has just fetched?
Increment the <i>Program Counter</i>	What is the Program Counter? What does the Program Counter count? Increment by how much? Where does the Program Counter point to after it is incremented?
Decode the Instruction	Why does the instruction need to be decoded? How does it get decoded?
Fetch the <i>Operands</i>	What are operands? What does it mean to fetch? Is this fetching distinct from the fetching in Step 1 above? Where are the operands? How many are there? Where do we put the operands after we fetch them?
Perform the Operation	Is this the main step? Couldn't the computer simply have done this part? What part of the CPU performs this operation?
Store the results	What results? Where from? Where to?
Repeat forever	Repeat what? Repeat from where? Is it really an infinite loop? Why? How do these steps execute any instructions at all?

In order to appreciate the operation of a computer we need to answer such questions and to consider in more detail the organisation of the CPU.

Representing Programs

Each complex task carried out by a computer needs to be broken down into a sequence of simpler tasks and a **binary machine instruction** is needed for the most primitive tasks. Consider a task that adds two numbers²⁰, held in memory locations designated by B and C²¹ and stores the result in memory location designated by A.

$$A = B + C$$

This assignment can be broken down (compiled) into a sequence of simpler tasks or **assembly instructions**, e.g:

Assembly Instruction	Effect
LOADR2, B	Copy the contents of memory location designated by B into Register 2
ADD R2, C	Add the contents of the memory location designated by C to the contents of Register 2 and put the result back into Register 2
STORE R2, A	Copy the contents of Register 2 into the memory location designated by A.

Each of these assembly instructions needs to be encoded into binary for execution by the Central Processing Unit (CPU). Let's try this encoding for a simple architecture called TOY1.

TOY1 Architecture

TOY1 is a fictitious architecture with the following characteristics:

1024 x 16-bit words of RAM maximum. RAM is word-addressable.

4 general purpose registers R0, R1, R2 and R3. Each general purpose register is 16-bits (the same size as a memory location).

16 different instructions that the CPU can decode and execute, e.g. LOAD, STORE, ADD, SUB and so on. These different instructions constitute the **Instruction Set** of the Architecture.

The representation for integers will be two's complement.

For this architecture, the architect (us) needs to define a coding scheme²² for instructions. This is termed the **Instruction Format**. Lets look at an example before we consider how we arrived at it. Here's our instruction format for TOY1:

TOY1 Instruction Format

TOY1 instructions are 16-bits (so they will fit into a main-memory word). Each instruction is divided into a number of **instruction fields** that encode a different piece of information for the CPU.

Field	OPCODE	RE	ADDRESS
Name	E	G	10-bits
Field	4-bits	2-bits	
Width		bits	
	<div><div></div><div></div><div></div><div></div></div>	<div><div></div><div></div></div>	<div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div>

The **OPCODE**²³ field identifies the CPU operation required. Since TOY1 only supports 16 instructions, these can be encoded as a 4-bit natural number. For TOY1, opcodes 1 to 4 will be²⁴:

0001 = LOAD 0010 = STORE 0011 = ADD 0100 = SUB

The **REG** field defines a General CPU Register. Arithmetic operations will use 1 register **operand** and 1 main memory **operand**, results will be written back to the register. Since TOY1 has 4 registers; these can be encoded as a 2-bit natural number:

00 = Register 0 01 = Register 1 10 = Register 2 11 = Register 3

The **ADDRESS** field defines the address of a word in RAM. Since TOY1 can have upto 1024 memory locations; a memory address can be encoded as a 10-bit natural number.

If we define addresses 200H, 201H and 202H for A, B and C, we can encode the example above as:

Assembly Instruction		Machine Instruction
LOAD	R2,	0001 10 10 0000
[201H]		0001
ADD	R2,	0011 10 10 0000
[202H]		0010
STORE	R2,	0010 10 10 0000
[200H]		0000

Memory Placement of Program and Data

In order to execute a TOY1 program, its instructions and data needs to be placed within main memory²⁵. We'll place our 3-instruction program in memory starting at address 080H and we'll place the variables A, B and C at memory words 200H, 201H, and 202H respectively. Such placement results in the following memory layout prior to program execution. For convenience, memory addresses and memory contents are also given in hex.

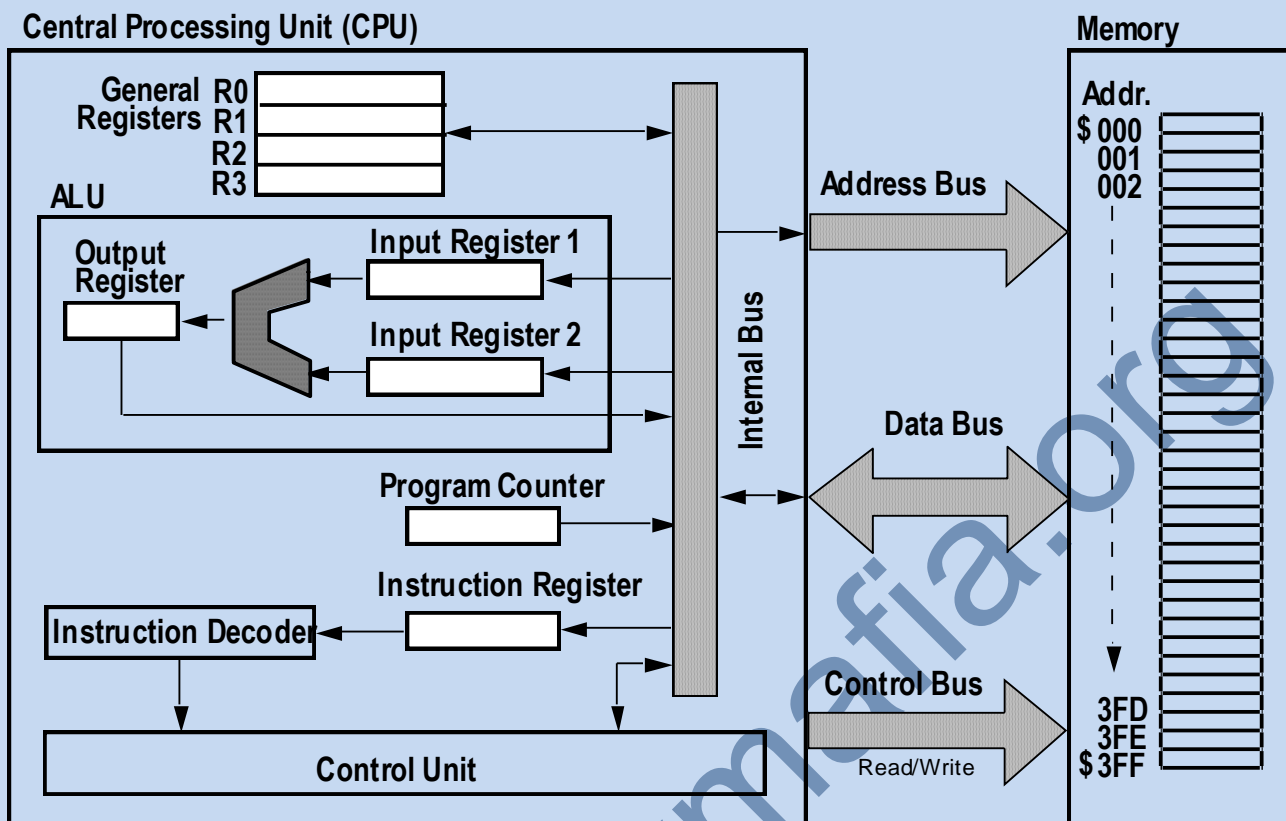
Memory Address in binary & hex	Machine Instruction OP Reg Address	Assembly Instruction
0000 1000 0000 0 8 0	0001 10 10 0000 0001 1 A 0 1	LOAD R2, [201H]
0000 1000 0001 0 8 1	0011 10 10 0000 0010 3 A 0 2	ADD R2, [202H]
0000 1000 0010 0 8 2	0010 10 10 0000 0000 2 A 0 0	STORE R2, [200H]
Etc	Etc	Etc

0010 00000000	0000 0000 0000 0000	A = 0
2 0 0	0 0 0 0	
0010 00000001	0000 0000 0000 1001	B = 9
2 0 1	0 0 0 9	
0010 00000010	0000 0000 0000 0110	C = 6
2 0 2	0 0 0 6	

Of course, the big question is “How is such a program executed by the TOY1 CPU?”

www.studymafia.org

CPU Organisation



The **Program Counter (PC)** is a special register that holds the **address** of the next instruction to be fetched from Memory (for TOY1, the PC is 10-bits wide). The PC is incremented²⁶ to "point to" the next instruction while an instruction is being fetched from main memory.

The **Instruction Register (IR)** is a special register that holds each instruction after it is fetched from main memory. For TOY1, the IR is 16-bits since instructions are 16-bit wide.

The **Instruction Decoder** is a CPU component that decodes and interprets the contents of the Instruction Register, i.e. it splits whole instruction into fields for the Control Unit to interpret. The Instruction decoder is often considered to be a part of the Control Unit.

The **Control Unit** is the CPU component that co-ordinates all activity within the CPU. It has connections to all parts of the CPU, and includes a sophisticated timing circuit.

The **Arithmetic & Logic Unit (ALU)** is the CPU component that carries out arithmetic and logical operations e.g. addition, comparison, boolean AND/OR/NOT.

The **ALU Input Registers 1 & 2** are special registers that hold the input operands for the ALU.

The **ALU Output Register** is a special register that holds the result of an ALU operation. On completion of an ALU operation, the result is copied from the ALU Output register to its final destination, e.g. to a CPU register, or main-memory, or to an I/O device.

The **General Registers R0, R1, R2, R3** are available for the programmer to use in his/her programs. Typically the programmer tries to maximise the use of these registers in order to speed program execution. For TOY1, the general registers are the same size as memory locations, i.e. 16-bits.

The **Buses** serve as communication highways for passing information within the CPU (CPU internal bus) and between the CPU and the main memory (the **address bus**, the **data bus**, and the **control bus**). The address bus is used to send addresses from the CPU to the main memory; these addresses indicate the memory location the CPU wishes to read or write. Unlike the address bus, the data bus is bi-directional; for writing, the data bus is used to send a word from the CPU to main-memory; for reading, the data bus is used to send a word from main-memory to the CPU. For TOY1, the Control bus²⁷ is used to indicate whether the CPU wishes to read from a memory location or write to a memory location. For simplicity we've omitted two special registers, the **Memory Address Register (MAR)** and the **Memory Data Register (MDR)**. These registers lie at the boundary of the CPU and Address bus and Data bus respectively and serve to buffer data to/from the buses.

Buses can normally transfer more than 1-bit at a time. For the TOY1, the address bus is 10-bits (the size of an address), the data bus is 16-bits (size of a memory location), and the control bus is 1-bit (to indicate a memory read operation or a memory write operation).

Interlude: the Von Neumann Machine Model

Most computers conform to the von Neumann's machine model, named after the Hungarian-American mathematician John von Neumann (1903-57).

In von Neumann's model, a computer has **3 subsystems** (i) a CPU, (ii) a main memory, and (iii) an I/O system. The main memory holds the program as well as data and the computer is allowed to manipulate its own program²⁸. In the von-Neumann model, instructions are executed **sequentially** (one at a time).

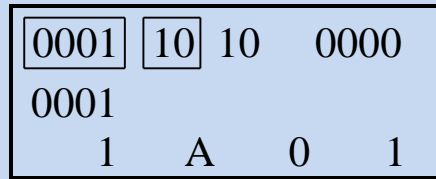
In the von-Neumann model a **single path** exists between the control unit and main-memory, this leads to the so-called "**von Neumann bottleneck**" since memory fetches are the slowest part of an instruction they become the bottleneck in any computation.

Instruction Execution (Fetch-Execute-Cycle Micro-steps)

In order to execute our 3-instruction program, the control unit has to issue and coordinate a series of micro-instructions. These micro-instructions form the fetch-execute cycle. For our example we will assume that the Program Counter register (PC) already holds the address of the first instruction, namely 080H.

LOAD R2, [201H]

0000 1000 0000
0 8 0



Copy the value in memory word 201H into Register 2

Control Unit Action

Data flows

FETCH

INSTRUCTION²⁹

PC to Address Bus ³⁰	080H	→	080H	Address Bus
0 to Control Bus ³¹	0	→	0	Control Bus
Address Bus to Memory	080H	→	080H	Memory
Control Bus to Memory	0	READ	0	Memory
Increment PC ³²	080	INC	081H	PC becomes PC+1 ³³
Memory [080H] to Data Bus	1A01H	→	1A01H	Data Bus
Data Bus to Instruction Register	1A01H	→	1A01H	Instruction Register

DECODE

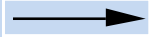



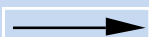
INSTRUCTION

IR to Instruction Decoder	1A01H	→	1A01H	Instruction Decoder
Instruction Decoder to Control Unit ³⁴	1, 2, 201H	→	1, 2, 201H	Control Unit

EXECUTE

INSTRUCTION³⁵

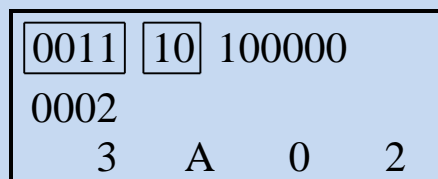
Control Unit to Address Bus	201H	→	201H	Address Bus
-----------------------------	------	---	------	-------------

0 to Control Bus	0		0	Control Bus
Address Bus to Memory	201H		201H	Memory
Control Bus to Memory	0		0	Memory
Memory [201H] to Data bus	0009H		0009H	Data Bus
Data Bus to Register 2	0009H		0009H	Register 2

www.studymafia.org

ADD R2, [202H]

0000 1000 0001
0 8 1



Add³⁶ the value in memory word 202H to Register 2

Control Unit Action

FETCH INSTRUCTION

Data flows

PC to Address Bus	081H	→	081H	Address Bus
0 to Control Bus	0	→	0	Control Bus
Address Bus to Memory	081H	→	081H	Memory
Control Bus to Memory	0	READ	0	Memory
Increment PC	081H	INC	082H	PC becomes PC+1
Memory [081H] to Data Bus	3A02H	→	3A02H	Data Bus
Data Bus to Instruction Register	3A02H	→	3A02H	Instruction Register

DECODE INSTRUCTION

IR to Instruction Decoder	3A02H	→	3A02H	Instruction Decoder
Instruction Decoder to Control Unit	3, 2, 202H	→	3, 2, 202H	Control Unit

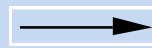
EXECUTE INSTRUCTION

Register 2 to ALU Input Reg 1	0009	→	0009	ALU Input Reg 1
Control Unit to Address Bus	202H	→	202H	Address Bus
0 to Control Bus	0	→	0	Control Bus
Address Bus to Memory	202H	→	202H	Memory
Control Bus to Memory	0	READ	0	Memory
Memory [202H] to Data Bus	0006H	→	0006H	Data Bus

bus

Data Bus to ALU Input
Reg 2

0006H



0006H

ALU Input Reg
2

Control Unit to ALU

ADD

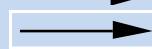


000FH

Output Register

ALU Output Reg to
Register 2

000F



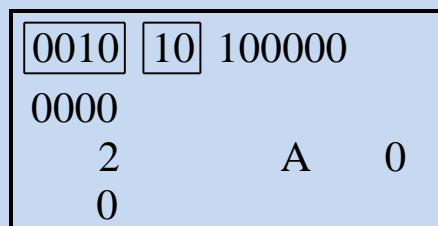
000FH

Register 2

www.studymafia.org

STORE R2, [200H]

0000 1000 0001
0 8 2



Copy the value in Register 2 into memory word 202H

Control Unit Action

Data flows

FETCH INSTRUCTION

PC to Address Bus	082H	→	082H	Address Bus
0 to Control Bus	0	→	0	Control Bus
Address Bus to Memory	082H	→	082H	Memory
Control Bus to Memory	0	READ	0	Memory
Increment PC	082H	INC	083H	PC becomes PC+1
Memory [082] to Data Bus	2A00H	→	2A00H	Data Bus
Data Bus to Instruction Register	2A00H	→	2A00H	Instruction Register

DECODE INSTRUCTION

IR to Instruction Decoder	2A00	→	2A00	Instruction Decoder
Instruction Decoder to Control Unit	2, 2, 200H	→	2, 2, 200H	Control Unit

EXECUTE INSTRUCTION

Register 2 to Data Bus	000FH	→	000FH	Data Bus
Control Unit to Address Bus	200H	→	200H	Address Bus
1 to Control Bus	1	→	1	Control Bus
Data Bus to Memory	000FH	→	000FH	Memory
Address Bus to Memory	200H	→	200H	Memory
Control Bus to Memory	1	WRITE	1	Memory

TOY1 Programming

How is computer such as TOY1 programmed? We'll consider this question with some examples. Let's first define a basic **Instruction Set** for the TOY1 architecture³⁷:

OP Code	Assembler Format	Action
0000	STOP	Stop Program Execution
0001	LOAD Rn, [addr]	Rn = Memory [addr]
0010	STORE Rn, [addr]	Memory [addr] = Rn
0011	ADD Rn, [addr]	Rn = Rn + Memory [addr]
0100	SUB Rn, [addr]	Rn = Rn - Memory [addr]
0101	GOTO addr	PC = addr
0110	IFZER Rn, addr	IF Rn = 0 THEN PC = addr
0111	IFNEG Rn, addr	IF Rn < 0 THEN PC = addr

Example 1: Multiplication

Given these instructions let's write a TOY1 assembly program, which will perform the following assignment:

$$A = B * C$$

where A, B and C denote integers placed at memory words 100H, 101H and 102H respectively. The first point to observe with this example is that a multiply operation is not available in the TOY1 instruction set!

Therefore we need to consider if we can use other instructions to carry out the multiplication. The obvious solution is to use repeated addition:

$$B * C = \sum_{N=1}^C B$$

Example: $12 * 3 = 12 + 12 + 12$

$$12 * 1 = 12$$

$$12 * 0 = 0$$

www.studymafia.org

Let's first write the multiplication algorithm in Pseudo Code

```
; Given: A, B, C
; Pre: C >= 0    Why do we have this pre-condition?
; Post: A = B * C

sum = 0    ; sum will accumulate the answer
n = C      ; n will indicate how many additions remain
loop
    exit when n <= 0
    sum = sum + B
    n = n - 1
end loop
A = sum
```

Let's try translating (compiling) this Pseudo Code to TOY1 instructions. Since we have 4 general registers, it is worthwhile allocating frequently used variables to them as this will lead to faster execution. Let's allocate Register 1 to hold 'sum', and Register 2 to hold 'n'.

```
sum = 0
```

The first assignment **sum=0** yields our first problem. How do we get zero (or any constant) into a Register?

The only instruction that we can use to set a register is **LOAD Rn, addr**. Therefore we must reserve a memory word and pre-set it to zero before program execution begins. Let's place zero in memory word 200H. Now to perform **sum = 0** we have:

```
LOAD    R1, [200H]    ; sum = 0
```

Let's place instructions starting at memory word 80H:

Address	Assembler	Instruction	Comment
80H	LOADR1,	[200H]	; sum = 0
200H	0		; holds zero

$n = C$

The next statement is $n = C$. This is easy to translate:

81H LOADR2, [102H] ; $n = C$

www.studymafia.org

exit when $n \leq 0$

What does the **loop exit when $n \leq 0$** statement mean in TOY1 terms?

Lets consider a simpler example first: **loop exit when $n = 0$** . On the TOY1 this statement has a simpler translation, namely:

Loop	Address	Instruction
exit when $n = 0$	L0	IFZER R2, Ly
<i>instructions</i>	L1	<i>instructions</i>
end loop
	Lx	GOTO L0
	Ly	

Note: GOTO alters the Program Counter register thereby causing an **unconditional branch** in the order of program execution. IFZER alters the Program Counter only if the contents of the specified Register are zero. To handle **exit when $n \leq 0$** we need to skip to the end of the **loop** if R2 is zero or if R2 is negative:

Loop	Address	Instruction
exit when $n \leq 0$	L0	IFZER R2,
0	Ly	
<i>instructions</i>	L1	IFNEG R2, Ly
end loop	L2	<i>instructions</i>

	Lx	GOTO L0
	Ly	

For our example we now have the following assembly program:

Address	Assembler Instruction	Comment
80H	LOADR1, [200H]	; sum = 0
81H	LOADR2,	; n = C

	[102H]	
82H	IFZER R2, Ly	; exit when $n \leq 0$
83H	IFNEG R2, Ly	; we will define Ly when we can
.....
Lx	GOTO 82H	; end loop
Ly	
.....		
100H	A	; holds A
101H	B	; holds B
102H	C	; holds C
.....		
200H	0	; holds 0

sum = sum + B
n = n - 1

Let's continue with: **sum = sum + B.** This is easy, namely

84H ADDR1, [101H] ; sum = sum + B

For **n = n - 1** we will assume that location 201H is pre-set to the constant 1.

85H SUB R2, [201H] ; n = n-1

201H 1 ; Holds the value 1

end loop

A = sum

Adding STORE R1, 100H for **A = sum** and a STOP instruction we arrive at the final program:

Add	Assembler	Comment	Machine
r	Instruct.		Instruction
80H	LOAD R1, [200H]	; sum = 0	0001 0110 0000 0000
81H	LOAD R2, [102H]	; n = C	0001 1001 0000 0010
82H	IFZER R2, 87H	; exit when n = 0	0110 1000 1000 0111
83H	IFNEG R2, 87H	; exit when n < 0	0111 1000 1000 0111
84H	ADD R1, [101H]	; sum = sum + B	0011 0101 0000 0001
85H	SUB R2, [201H]	; n = n - 1	0100 1010 0000 0001
86H	GOTO 82H	; end loop	0101 0000 1000 0010
87H	STORE R1, [100H]	; A = sum	0010 0101 0000 0000
88H	STOP	; End of program	0000 0000 0000

...	...		0000
100	A	; Holds A	initial value of A
H			
101	B	; Holds B	initial value of B
H			
102	C	; Holds C	initial value of C
H			
...	...		
200	0	; Holds Zero	0000 0000 0000
H			0000
201	1	; Holds One	0000 0000 0000
H			0001

Multiplication (An improvement)

The multiply program will work correctly but can be improved. Consider $3 * 1000$ if C is greater than B then it will be faster to compute $1000 * 3$. How can we adapt our program to handle this case? Consider and work through the following solution:

	Addr	Instruction
sum = 0	80H	LOAD R1, [200H] ; sum=0
if B <= C then	81H	LOAD R0, [102H] ; if
big=C, n=B		C<B
else C < B	82H	SUB R0, [101H] ; then
big=B, n=C		ELSE
end if	83H	IFNEG R0, 88H
loop exit when n	84H	LOAD R0, [102H] ; then
<= 0	85H	STORE R0, [202H] ; big
sum = sum +		= C
big	86H	LOAD R2, [101H] ; n = B
n = n - 1	87H	GOTO 8BH
end loop	88H	LOAD R0, [101H] ; else
A = sum	89H	STORE R0, [202H] ;
		big=B
	8AH	LOAD R2, [102H] ;
		n=C
	8BH	etc ; loop....
	...	
	202H	... ; Holds
	big	

Example 2: Vector Sum

Write a sequence of TOY1 instructions (and constants) to sum 100 integers stored consecutively starting at memory word 200H. The sum is to be left in Register 0.

Again, lets first write the Pseudo Code for the problem:

```
sum = 0
n = 100
addr = 200H
```

```
loop  
  exit when n <= 0  
  sum = sum + RAM [addr]  
  addr = addr + 1  
  n = n - 1  
end loop
```

Looking at this code, we find that the main "difficulty" is how to perform

sum = sum + RAM [addr]

There doesn't appear to be any way of accessing memory words based on a "Variable". We need therefore to extend TOY1 to include an indirect addressing capability³⁹.

www.studymafia.org

Indirect Addressing Instructions for TOY1

OP Code	Assembler Format	Action
1001	LOAD Rn, [Rm]	Rn = Memory [Rm] ⁴⁰
1010	STORE Rn, [Rm]	Memory [Rm] = Rn
1011	ADD Rn, [Rm]	Rn = Rn + Memory [Rm]
1100	SUB Rn, [Rm]	Rn = Rn – Memory [Rm]

A second Instruction Format is also needed for these instructions⁴¹. We will use the following:

OPCODE	REGn	REGm	Unused ⁴²
4-bits	2-bits	2-bits	8-bits

Example: Given this format the TOY1 instruction ADD R1, [R2] would be coded as

1011 01 10 0000 0000 in binary or B600H in hexadecimal.

Vector Sum Example Contd.

The vector sum example is now straightforward. This program will be placed at 0FH onwards, and the registers allocated as follows: R0 for 'sum', R1 for 'n', R2 for 'addr'

sum = 0	0	0	; Holds 0
n = 100	1	1	; Holds 1
addr = 200H	2	100	; Holds 100
loop	3	200H	; Holds 200H
exit when n <= 0	...		
sum = sum + RAM	0FH	LOAD R0, [0]	; sum = 0
[addr]	10H	LOAD R1, [2]	; n = 100
addr = addr + 1	11H	LOAD R2, [3]	; addr = 200H

$n = n - 1$ end loop ; Result in Register R0	12H IFZER R1, 18H ; exit when n<=0 13H IFNEG R1, 18H 14H ADD R0, [R2] ; sum = sum+... 15H ADD R2, [1] ; addr = addr + 1 16H SUB R1, [1] ; n = n - 1 17H GOTO 12H ; end loop 18H STOP
---	--

REFERENCES

1. www.google.com
2. www.wikipedia.org
3. www.studymafia.org
4. www.pptplanet.com