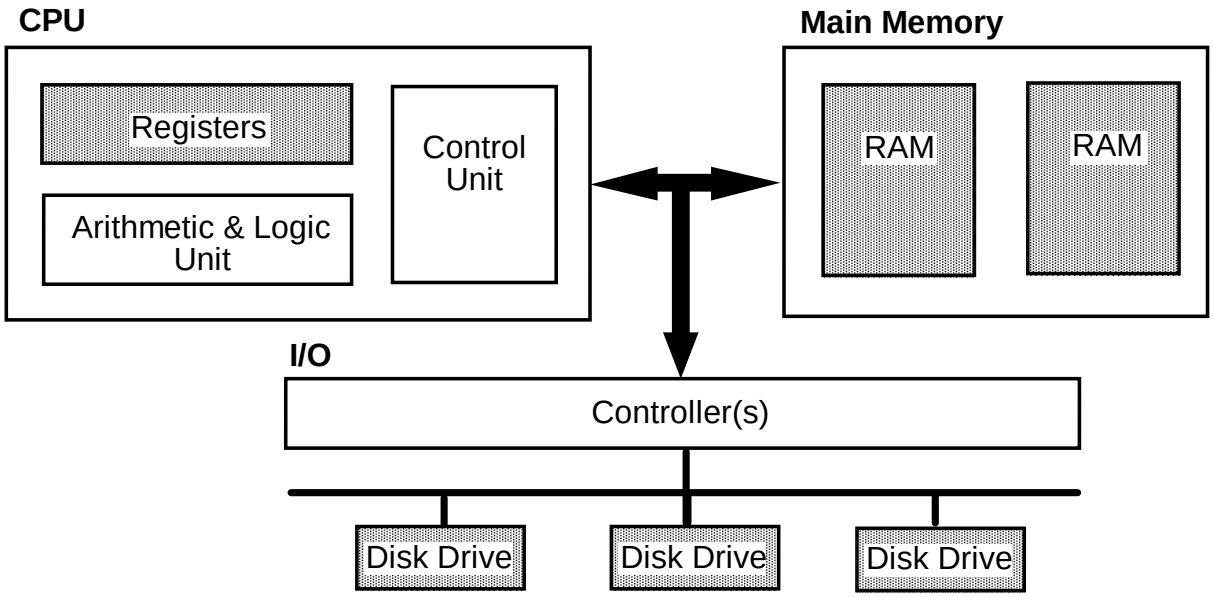
Memory Organisation

CPU Organisation and Operation

Introduction to Assembly Programming

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 quickly[1] 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 Registers[2] are available for general[3] 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

---

1 e.g. less than a nanosecond (10^{-9} sec)

2 Occasionally called Working Registers

3 Used for performing calculations, moving and manipulating data etc.

the 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 next[4] fastest memory within a computer and is much larger in size. Typical main memory capacities for different kinds of computers are: PC 512MB[5], fileserver 2GB, database server 8GB. Computer architectures also impose an architectural constraint on the maximum allowable RAM. This constraint is normally equal to $\mathbf{2^{WordSize}}$ memory locations.

RAM[6] (Random[7] 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 location[8] in RAM can still be "read" or "written" very quickly[9]. 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 onwards[10]. Like registers, the contents of RAM are lost if the power is turned off.

---

4 Actually many computers systems also include Cache memory, which is faster than Main memory, but slower than register memory. We will ignore Cache memories in this course.

5 $1K = 2^{10} = 1024$, $1M = 2^{20}$, $1G = 2^{30}$, 'B' will be used for Bytes, and 'b' or 'bit' for bits, cf. 1MB and 1Mbit

6 There are many types of RAM technologies.

7 Random is a Misnomer. Direct Access Memory would have been a better term.

8 Typically a byte multiple.

9 e.g. less than 10 nanoseconds (10×10^{-9} sec)

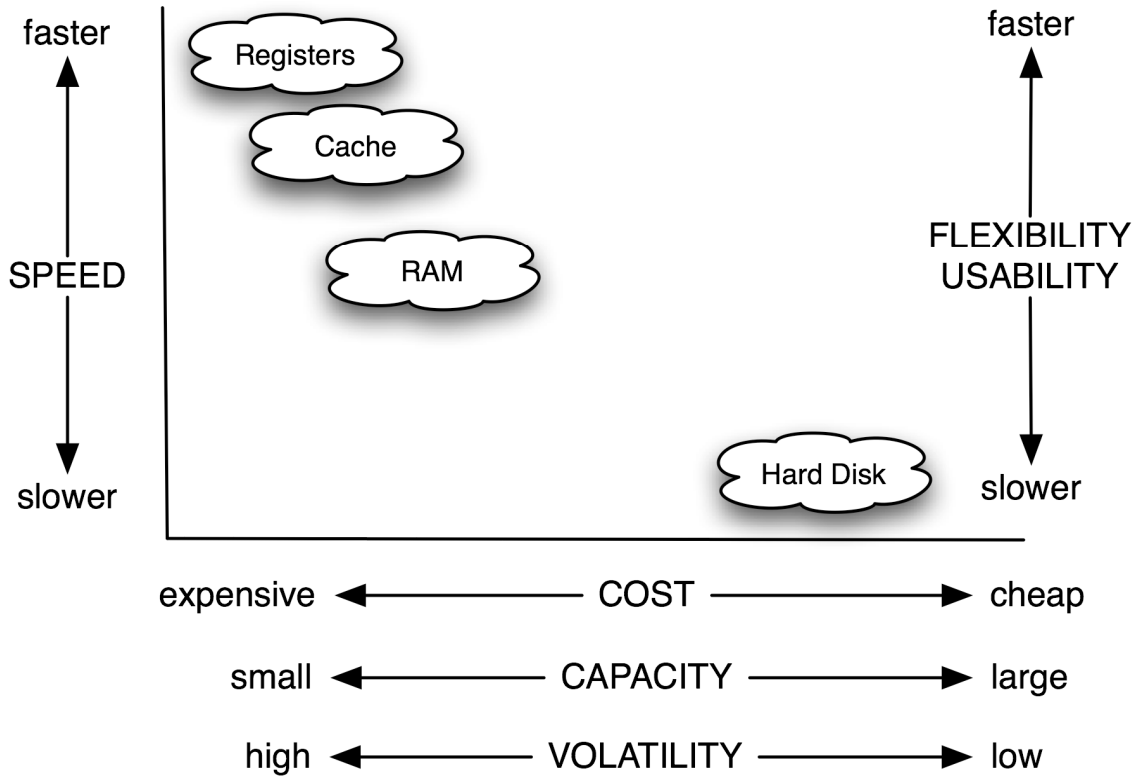
10 Some RAM locations (typically those with the lowest & highest addresses) may cause side-effects, e.g. cause data to be transferred to/from external devices

Disk Memory

Disk memory[11] 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 ($5x10^{29}$). 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 cable[12]. 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 x 8 ~ 1.3GB/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 x 8 x 2 ~ 3.2GB/sec over an 8-byte (64-bit) data bus.

ROM, PROM, EPROM, EEPROM, Flash

In addition to RAM, they 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.

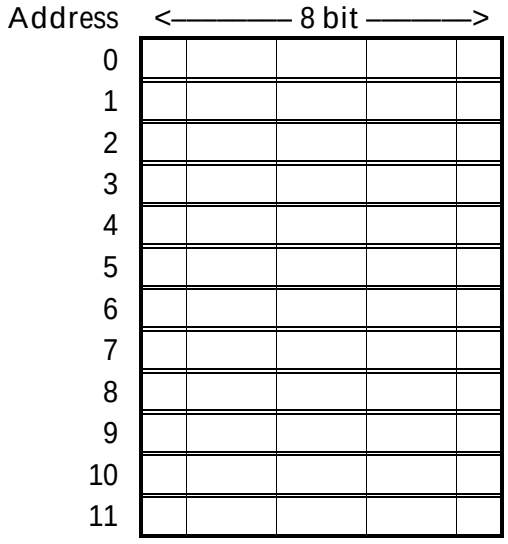
EPROM (Erasable PROM). With EPROM we can erase (using strong ultra-violet light) the contents of the chip and rewrite it with new contents, typically several thousand times. It is commonly used to store the "boot" program of a computer, known as the firmware. PCs call this firmware, the BIOS (Basic I/O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

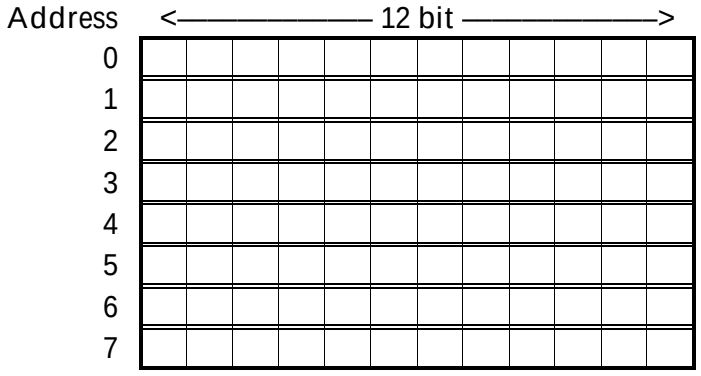
EEPROM (Electrically Erasable PROM). As the name implies the contents of EEPROMs are erased electrically. EEPROMSs are also limited to the number of erase-writes that can be performed (e.g, 100,000) but support updates (erase-writes) to individual bytes whereas EPROM updates the whole memory and only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

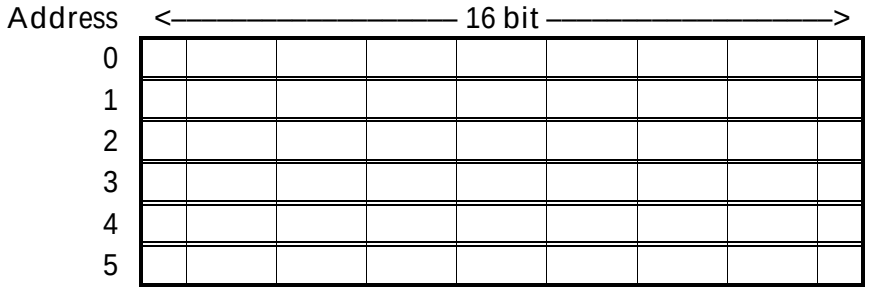
FLASH memory is a cheaper form of EEPROM where updates (erase-writes) can only be performed on blocks of memory, not on individual bytes. Flash memories are found in USB sticks, flash cards and typically range in size from 32M to 2GB. The number of erase/write cycles to a block is typically several hundred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

Main Memory Organisation

Main memory can be considered to be organised as a matrix of bits. Each row represents a memory location, typically this is equal to the word size of the architecture, although it can be a word multiple (e.g. 2xWordsize) or a partial word (e.g. half the wordsize**). For simplicity we will assume that data within main memory can only be read or written a single row (memory location) at a time.** For a 96-bit memory we could organise the memory as 12x8 bits, or 8x12 bits or, 6x16 bits, or even as 96x1 bits or 1x96 bits. Each row also has a natural number called its **address**[13] which is used for selecting the row:



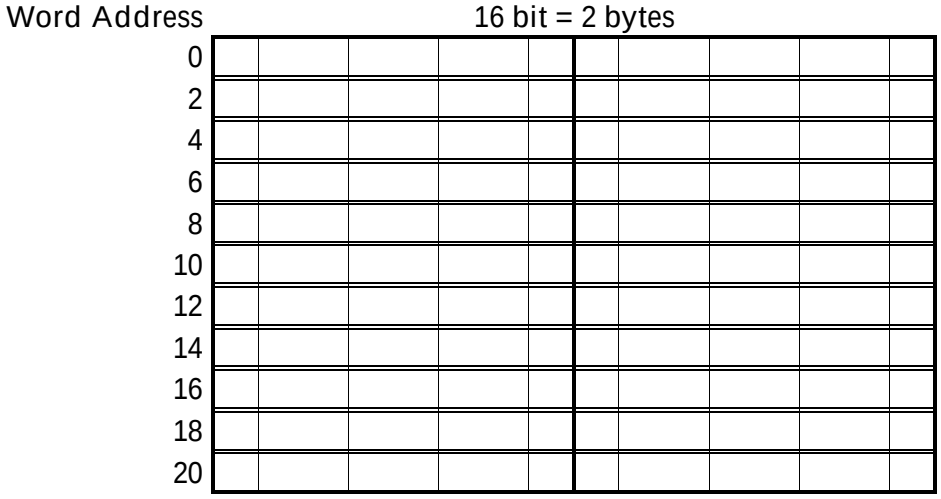




[13] The concept of an address is very important to properly understanding how CPUs work.

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 locations[14]. Note how the memory locations (rows) have even addresses.



Byte Ordering

The ordering of bytes within a **multi-byte** data item defines the endian-ness 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.

Big-Endian

Word Address				
0	0	1	2	3
4	4	5	6	7
8	8	9	10	11
12	12	13	14	15

MSB ——————-> LSB

Little-Endian

Word Address				
0	3	2	1	0
4	7	6	5	4
8	11	10	9	8
12	15	14	13	12

MSB <—————— LSB

[14] To avoid confusion we will use the term **memory word** for a word-sized memory location.

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

	MSB ——————>			LSB
	24	25	26	27
Word 24	12	2E	5F	01

Little Endian

	MSB ——————>			LSB
	27	26	25	24
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

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

Little Endian

	+3	+2	+1	+0
Word 24		M	I	J
Word 28	T	I	M	S
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 systems[15], in use today are split between those that are big-endian, and those that are little-endian[16]. 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**.

15 The interested student might want to read the paper, "On Holy Wars and a Plea for Peace", D. Cohen, IEEE Computer, Vol 14, Pages 48-54, October 1981.

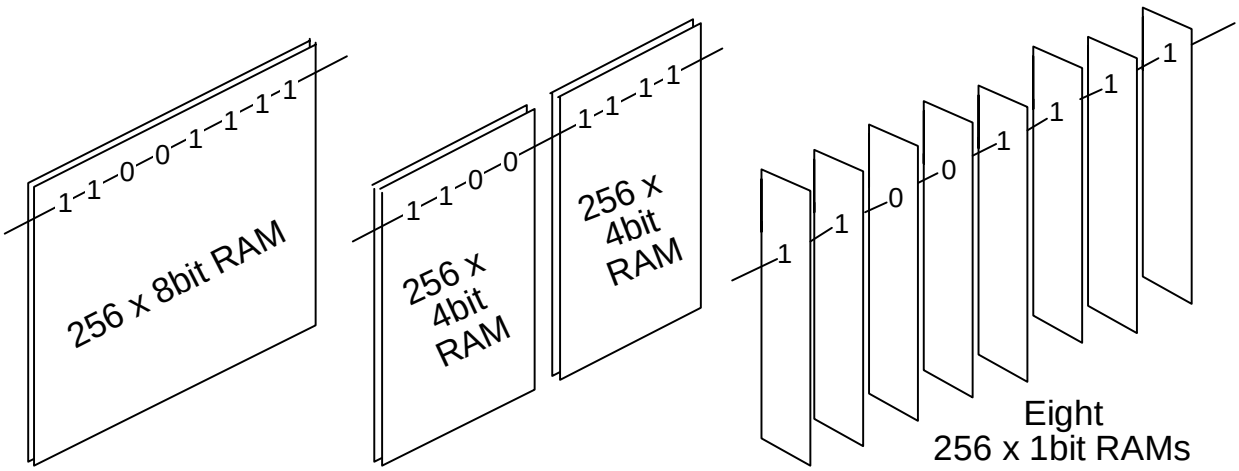
16 The Motorola 68000 architecture is big-endian, while the Intel Pentium architecture is little-endian.

Address	Memory (16-bit) word		
0	**MSB**	**LSB**	Word starting at Address 0 is Aligned
2			
4		**MSB**	Word starting at Address 5 is Unaligned
6	**LSB**		

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 slower[17]. 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.

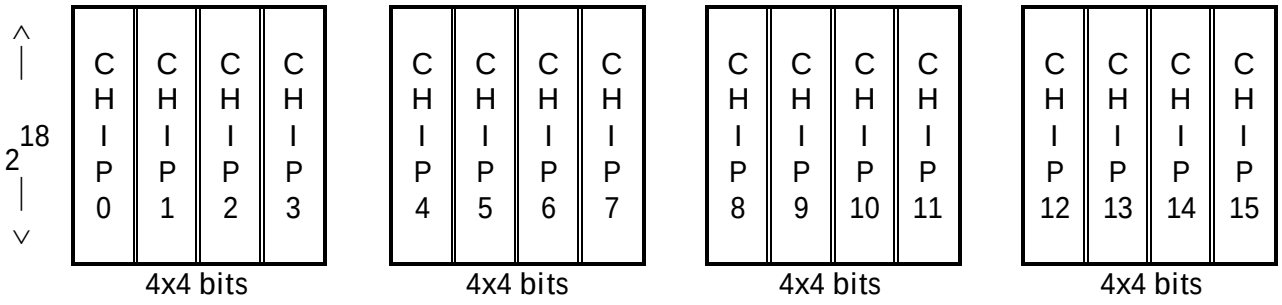
On PCs, memory modules are known as DIMMs (dual inline memory modules) and support 64-bit transfers. The previously generation of modules were called SIMMs (single inline memory modules) and supported 32-bit data transfers.

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

RAM chips = 256K x 4 bit

Module 0 Module 1 Module 2 Module 3

[17] Describe a method for doing an unaligned word write operation.



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

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 x 16) / (256K x 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 4x2=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)

CPU Organisation & Operation

The Fetch-Execute Cycle

The operation of the CPU[18] is usually described in terms of the Fetch-Execute cycle[19].

Fetch-Execute Cycle	The cycle raises many interesting questions, e.g.
Fetch the *Instruction*	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 *Program Counter*	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 *Operands*	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 numbers[20], held in memory locations designated by B and C[21] and stores the result in memory location designated by A.

$$A = B + C$$

---

18 Central Processing Unit.

19 Sometimes called the Fetch-Decode-Execute Cycle.

20 Let's assume they are held in two's complement form.

21 A, B and C are actually main memory **addresses**, i.e. natural binary numbers.

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

Assembly Instruction	Effect
LOAD R2, 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).

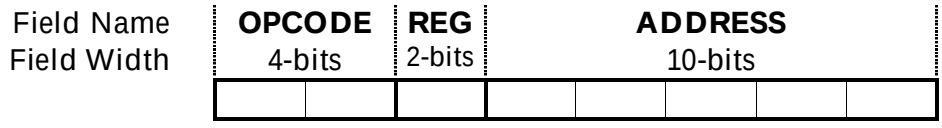
16 different instructions that the CPU can decode and execute, e.g. LOAD, STORE, ADD, SUB and so on. These different instructions constitute the **Instruction Set** of the Architecture.

The representation for integers will be two's complement.

For this architecture, the architect (us) needs to define a coding scheme[22] 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.



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 be[24]:

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

22 Most architectures actually have different instruction formats for different categories of instruction.

23 Operation Code

24 The meaning of CPU operations is defined in the Architecture's Instruction Set Manual.

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, [201H]	0001 10 10 0000 0001
ADD	R2, [202H]	0011 10 10 0000 0010
STORE	R2, [200H]	0010 10 10 0000 0000

Memory Placement of Program and Data

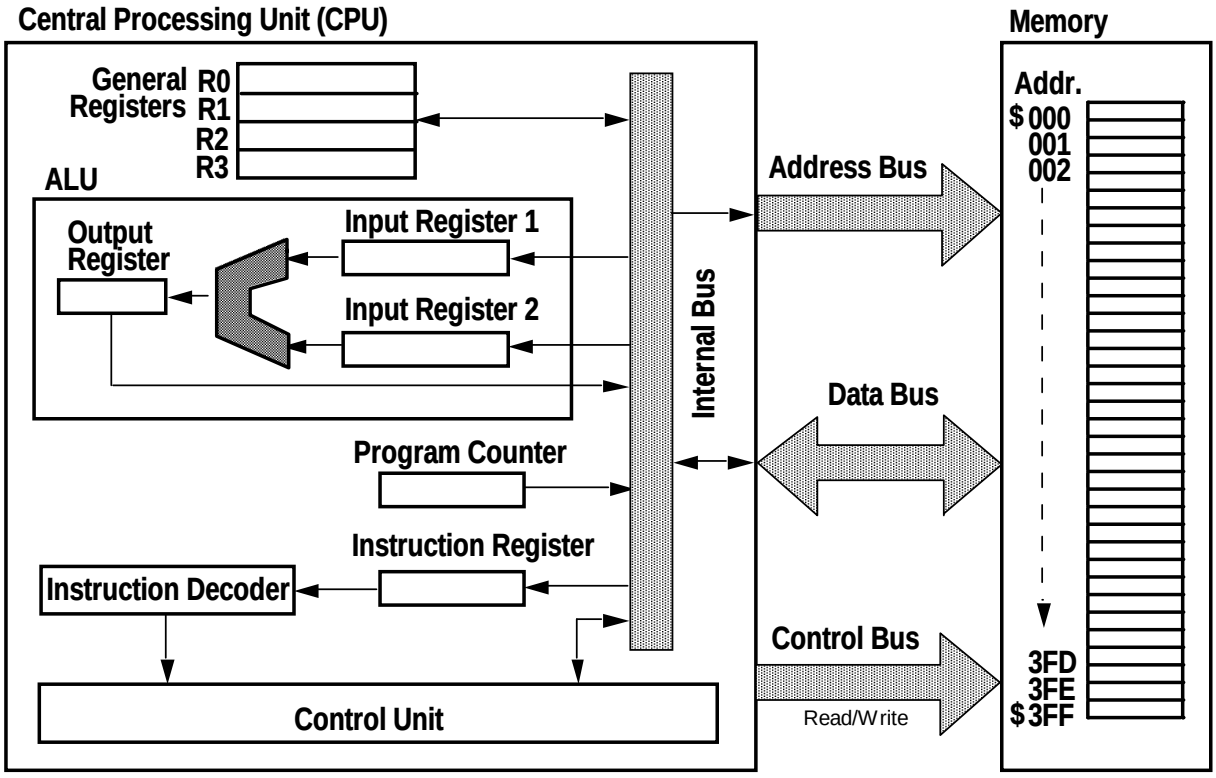
In order to execute a TOY1 program, its instructions and data needs to placed within main memory[25]. We'll place our 3-instruction program in memory starting at address 080H and we'll place the variables A, B and C at memory words 200H, 201H, and 202H respectively. Such placement results in the following 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 0	1000 8	0000 0	0001 1	10	10 A	0000 0	0001 1	LOAD R2, [201H]
0000 0	1000 8	0001 1	0011 3	10	10 A	0000 0	0010 2	ADD R2, [202H]
0000 0	1000 8	0010 2	0010 2	10	10 A	0000 0	0000 0	STORE R2, [200H]
Etc			**Etc**					**Etc**
0010 2	0000 0	0000 0	0000 0	0000 0		0000 0	0000 0	A = 0
0010 2	0000 0	0001 1	0000 0	0000 0		0000 0	1001 9	B = 9
0010 2	0000 0	0010 2	0000 0	0000 0		0000 0	0110 6	C = 6

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

[25] The Operating System software is normally responsible for undertaking this task.

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 incremented[26] 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. its 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.

---

[26] By the appropriate number of memory words.

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 bus[27] 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 program[28]. 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 until 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.

---

[27] Most control-buses are wider than a single bit, these extras bits are used to provide more sophisticated memory operations and I/O operations.

[28] This type of manipulation is not regarded as a good technique for general assembly programming.

LOAD R2, [201H]

0000	1000	0000	0001	10	10	0000	0001
0	8	0	1		A	0	1

Copy the value in memory word 201H into Register 2

Control Unit Action			**Data flows**	
FETCH INSTRUCTION[29]				
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 INSTRUCTION[35]				
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	**READ** →	0	Memory
Memory [201H] to Data bus	0009H	→	0009H	Data Bus
Data Bus to Register 2	0009H	→	0009H	Register 2

29 The micro-steps in the Fetch and Decode phases are common for all instructions.

30 This and the next 4 micro-steps initiate a fetch of the next instruction to be executed, which is to found at memory address 80H. In practice a Memory Address Register (MAR) acts as an intermediate buffer for the Address, similarly a Memory Data Register (MDR) buffers data to/from the data bus.

31 We will use 0 for a memory READ request, and 1 for a memory WRITE request.

32 For simplicity, we will assume that the PC is capable of performing the increment internally. If not, the Control Unit would have to transfer the contents of the PC to the ALU, get the ALU to perform the increment and send the results back to the PC. All this while we are waiting for the main-memory to return the word at address 80H.

33 Since TOY1's main-memory is word-addressed, and all instructions are 1 word. If main-memory was byte-addressed we would need to add 2.

34 The Instruction decoder splits the instruction into the individual instruction fields OPCODE, REG and ADDRESS for interpretation by the Control Unit.

35 The micro-steps for the execute phase actually perform the operation.

ADD R2, [202H]

0000	1000	0001
0	8	1

0011	10	1000000002
3	A	0 2

Add[36] the value in memory word 202H to Register 2

Control Unit Action		Data flows		
FETCH INSTRUCTION				
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
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

[36] Using two's complement arithmetic.

STORE R2, [200H]

0000	1000	0001
0	8	2

0010	10	1000000000	
2	A	0	0

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