1

Introduction to Computer Architecture

Welcome to the exciting world of **computer architecture**. Computer architecture is the study of computers. We shall study the basic design principles of computers in this book including the basic technologies, algorithms, design methodologies and future trends.

The field of computer architecture is a very fast moving field, and every couple of years there are a plethora of new inventions. Fifty years ago, the existence of computers was almost unknown to the common man. Computers were visible only in large financial institutions or in top universities. However, today billions of people all over the world have access to some form of computing device. They use it actively, and have found a place for it in their daily activities. Such kind of an epic transformation in the use, and ubiquity of computers has made the field of computer architecture extremely interesting.

In this chapter, we shall present an overview of computer architecture from an academic standpoint, and explain the major principles behind today's computers. We shall observe that there are two perspectives in computer architecture. We can look at computer architecture from the point of view of software applications. This point of view is sometimes referred to as *architecture* in literature. It is very important for students of computer architecture to study computer architecture from the viewpoint of a software designer because they need to know about the expectations of software writers from hardware. Secondly, it is also important for software writers to know about computer architecture because they can tailor their software appropriately to make it more efficient. In the case of system software such as operating systems and device drivers, it is absolutely essential to know the details of the architecture because the design of such kind of software is very strongly interlinked with low level hardware details.

The other perspective is the point of view of hardware designers. Given the software interface, they need to design hardware that is compatible with it and also implement algorithms that make the system efficient in terms of performance and power. This perspective is also referred to as *organisation* in literature.

Definition 1

Architecture *The view of a computer presented to software designers.*

Organisation *The actual implementation of a computer in hardware.*

Computer architecture is a beautiful amalgam of software concepts and hardware concepts. We design hardware to make software run efficiently. Concomitantly, we also design software keeping in mind the interface and constraints presented by hardware. Both the perspectives run hand in hand. Let us start out by looking at the generic definition of a *computer*.

1.1 What is a Computer?

Let us now answer the following questions.

Question 1
What is a computer?
What it can do, and what it cannot do?
How do we make it do intelligent things?

Let us start out with some basic definitions. The first question that we need to answer is – What is a computer? Well to answer this question, we just need to look all around us. We are surrounded by computers. Nowadays, computers are embedded in almost any kind of device such as mobile phones, tablets, mp3 players, televisions, dvd players, and obviously desktops and laptops. What is common between all of these devices? Well, each one of them has a computer that performs a specific task. For example, the computer in a mp3 player can play a song, and the computer in a dvd player can play a movie. It is absolutely not necessary that the mp3 player and dvd player contain different types of computers. In fact, the odds are high that both the devices contain the same type of computer. However, each computer is programmed differently, and processes different kinds of **information**. An mp3 player processes music files, and a dvd player processes video files. One can play a song, while the other can play a video.

Using these insights, let us formally define a computer in Definition 2.

Definition 2
A computer is a general purpose device that can be programmed to process information, and yield meaningful results.

Note that there are three important parts to the definition as shown in Figure 1.1 – the computer, information store, and the program. The computer takes as an input a program, and in response performs a set of operations on the information store. At the end it yields meaningful results. A typical program contains a set of instructions that tell the computer regarding the operations that need to be performed on the information store. The *information store* typically contains numbers and pieces of text that the program can use. Let us consider an example.

Example 1
Here is a snippet of a simple C program.

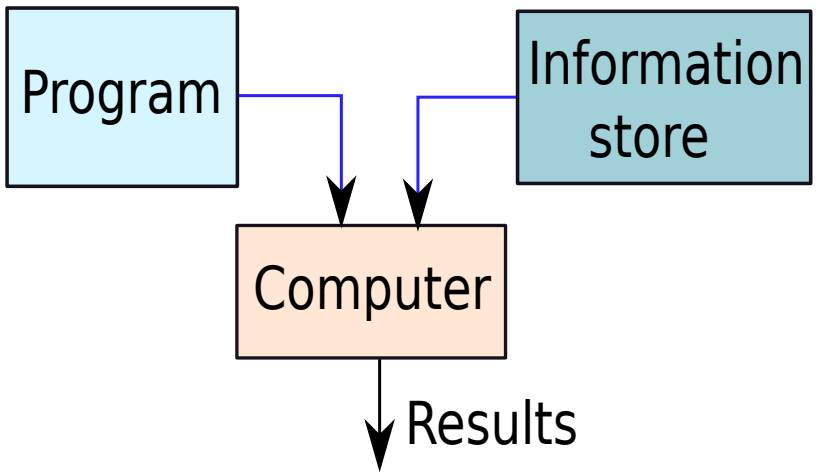


Figure 1.1: A basic computer

```
1: a = 4;
2: b = 6;
3: c = a + b;
4: print c
```

A computer will produce the output - 10. This C program contains four statements. Here, each statement can conceptually be treated as an instruction. Each statement instructs the computer to do something. Statements 1 and 2 instruct the computer to assign the variables a and b, the values 4 and 6 respectively. Statement 3 instructs the computer to add a and b, and assign the result to variable c. Finally, statement 4 instructs the computer to print the value of c (output of the program).

Given the fact that we have defined a computer as a sophisticated device that follows the instructions in a program to produce an output, let us see how it can be built. Modern day computers are made of silicon based transistors and copper wires to connect them. However, it is absolutely not necessary that computers need to be built out of silicon and copper. Researchers are now looking at building computers with electrons (quantum computers), photons(optical computers), and even DNA. If we think about it, our own brains are extremely powerful computers themselves. They are always in the process of converting thoughts(program) into action(output).

1.2 Structure of a Typical Desktop Computer

Let us now open the lid of a desktop computer, and see what is inside (shown in Figure 1.2). There are three main parts of a typical desktop computer – CPU (Central Processing Unit), Main Memory, and Hard Disk. The CPU is also referred to as the *processor* or simply *machine* in common parlance. We will use the terms interchangeably in this book. The CPU is the main part of the computer that takes a program as input, and executes it. It is the brain of the computer. The main memory is used to store data that a program might need during its execution (information store). For example, let us say that we want to recognise all the faces in an image. Then the image will be stored in main memory. There is some limited storage on the

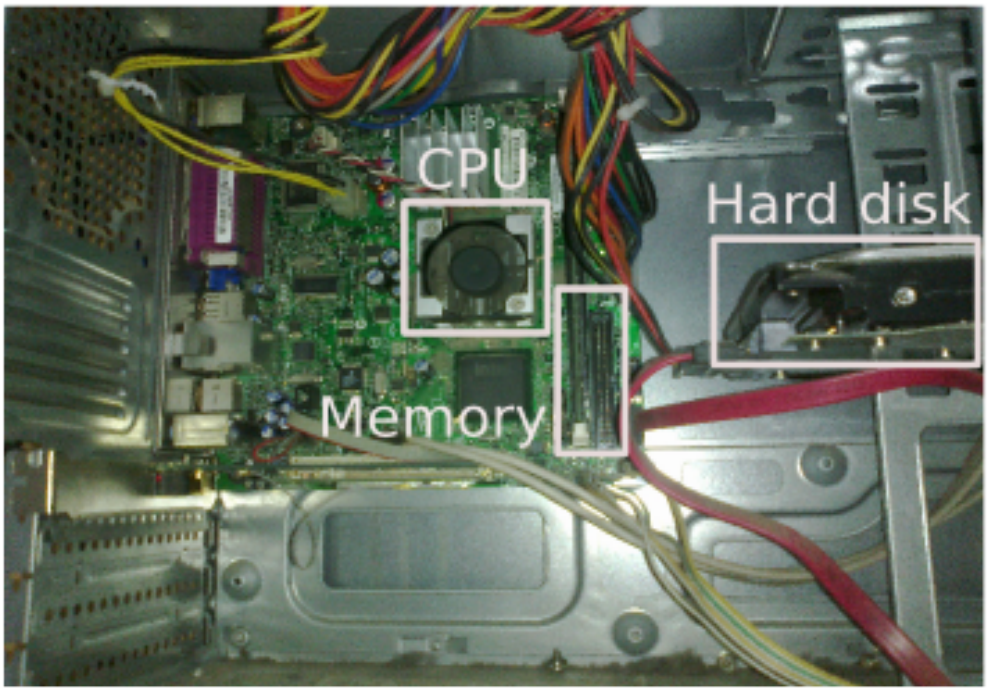


Figure 1.2:

processor itself. However, we shall discuss this aspect later. When we turn off the power, the processor and main memory lose all their data. However, the hard disk represents permanent storage. We do not expect to lose our data when we shut down the system. This is because all our programs, data, photos, videos, and documents are safely backed up in the hard disk.

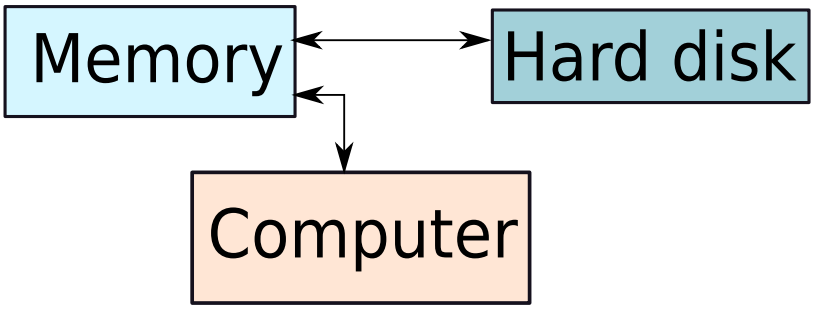


Figure 1.3: Block diagram of a simple computer

Figure 1.3 shows a simplistic block diagram of the three components. Along with these main components, there are a host of peripheral components that are connected to the computer. For example, the keyboard and mouse are connected to a computer. They take inputs from the user and communicate them to programs running on the processor. Similarly, to show the output of a program, the processor typically sends the output data to a monitor that can graphically display the result. It is also possible to print the result using a printer. Lastly the computer can be connected to other computers through the network. A revised block diagram with all the peripherals is shown in Figure 1.4.

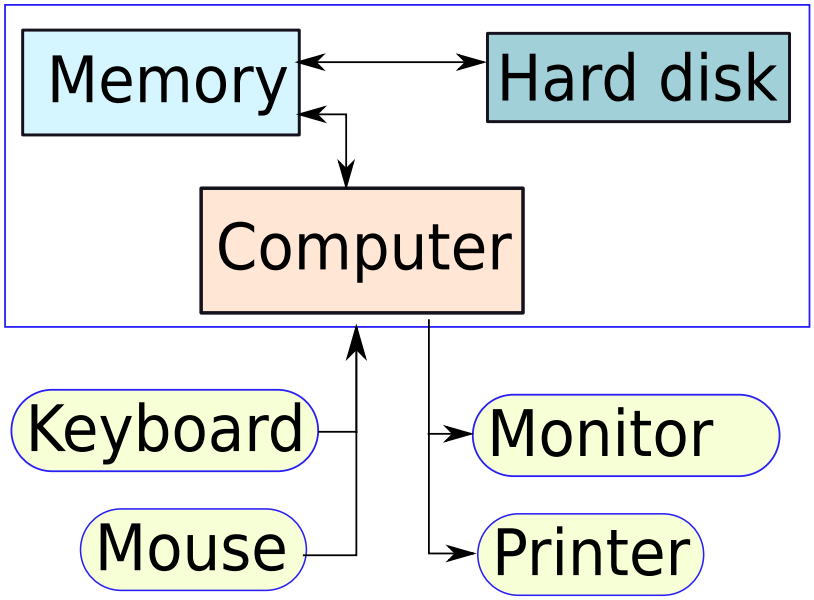


Figure 1.4: Block diagram of a simple computer with peripherals

In this book, we will mainly study the processor. The processor has the central responsibility of executing programs, communicating with the main memory, hard disk, and peripherals. It is the only active unit in our entire system. The others are passive and only respond to requests. They do not have any computational capability of their own.

1.3 Computers are Dumb Machines

Irrespective of the underlying technology, a fundamental concept that we need to understand is that a computer is fundamentally a *dumb* machine. Unlike our brains, it is not endowed with abstract thought, reason, and conscience. At least at the moment, computers cannot take very sophisticated decisions on their own. All they can do is execute a program. Nonetheless, the reason computers are so powerful is because they are extremely good at executing programs. They can execute billions of basic instructions per second. This makes them dumb yet very fast. A comparison of the computer with the human brain is shown in Table 1.1.

Feature	Computer	Our Brilliant Brain
Intelligence	Dumb	Intelligent
Speed of basic calculations	Ultra-fast	Slow
Can get tired	Never	After some time
Can get bored	Never	Almost always

Table 1.1: Computer vs the brain

If we combine the processing power of computers, with intelligent programs written by the human brain, we have the exquisite variety of software available today. Everything from operating systems to word processors to computer games is written this way.

The basic question that we need to answer is :

Question 2
How, do we make a dumb machine do intelligent things?

Computers are these tireless machines that can keep on doing calculations very quickly without ever complaining about the monotonicity of the work. As compared to computers, our brains are creative, tire easily, and do not like to do the same thing over and over again. To combine the best of both worlds, our brains need to produce computer programs that specify the set of tasks that need to be performed in great detail. A computer can then process the program, and produce the desired output by following each instruction in the program.

Hence, we can conclude that we should use the creative genius of our brains to write programs. Each program needs to contain a set of basic instructions that a computer can process. Henceforth, a computer can produce the desired output. An *instruction* is defined as a basic command that can be given to a computer.

1.4 The Language of Instructions

We observe that to communicate with a computer, we need to speak its language. This language consists of a set of basic instructions that the computer can understand. The computer is not smart enough to process instructions such as, "calculate the distance between New Delhi and the North Pole". However, it can do simple things like adding two numbers. This holds for people as well. For example, if a person understands only Spanish, then there is no point speaking to her in Russian. It is the responsibility of the person who desires to communicate to arrange for a translator. Likewise, it is necessary to convert high level thoughts and concepts to basic instructions that are machine understandable.

Programmers typically write programs in a high level language such as C or Java™. These languages contain complex constructs such as structures, unions, switch-case statements, classes and inheritance. These concepts are too complicated for a computer to handle. Hence, it is necessary to pass a C or C++ program through a dedicated program called a *compiler* that can convert it into a sequence of basic instructions. A compiler effectively removes the burden of creating machine (computer) readable code from the programmer. The programmer can concentrate only on the high level logic. Figure 1.5 shows the flow of actions. The first step is to write a program in a high level language (C or C++). Subsequently, the second step involves compiling it. The compiler takes the high level program as input, and produces a program containing machine instructions. This program is typically called an *executable* or *binary*. Note, that the compiler itself is a program consisting of basic machine instructions.

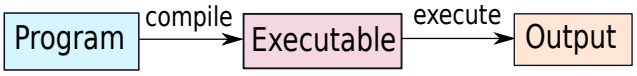


Figure 1.5: Write-compile-execute

Let us now come to the semantics of instructions themselves. The same way that any language has a finite number of words, the number of basic instructions/rudimentary commands that a processor can support have to be finite. This set of instructions is typically called the **instruction set**. Some examples

of basic instructions are: add, subtract, multiply, logical or, and logical not. Note that each instruction needs to work on a set of variables and constants, and finally save the result in a variable. These variables are not programmer defined variables; they are internal locations within the computer. We define the term instruction set architecture as:

Definition 3
The semantics of all the instructions supported by a processor is known as the instruction set architecture (ISA). This includes the semantics of the instructions themselves, along with their operands, and interfaces with peripheral devices.

The instruction set architecture is the way that software perceives hardware. We can think of it as the list of basic functions that the hardware exports to the external world. It is the, "language of the computer". For example, Intel and AMD CPUs use the x86 instruction set, IBM processors use the PowerPC® instruction set, HP processors use the PA-RISC instruction set, and the ARM processors use the ARM® instruction set (or variants of it such as Thumb-1 and Thumb-2). It is thus not possible to run a binary compiled for an Intel system on an ARM based system. The instruction sets are not compatible. However, in most cases it is possible to reuse the C program. To run a C program on a certain architecture, we need to procure a compiler for that specific architecture, and then appropriately compile the C program.

1.5 Instruction Set Design

Let us now begin the difficult process of designing an instruction set for a processor. We can think of an instruction set as a legal contract between software and hardware. Both sides need to implement their side of the contract. The software part needs to ensure that all the programs that users write can be successfully and efficiently translated to basic instructions. Likewise, hardware needs to ensure that all the instructions in the instruction set are efficiently implementable. On both sides we need to make reasonable assumptions. An ISA needs to have some necessary properties and some desirable properties for efficiency. Let us first look at a property, which is absolutely necessary.

1.5.1 Complete - The ISA should be able to Implement all User Programs

This is an absolutely necessary requirement. We want an ISA to be able to represent all programs that users are going to write for it. For example, if we have an ISA with just an ADD instruction, then we will not be able to subtract two numbers. To implement loops, the ISA should have some method to re-execute the same piece of code over and over again. Without this support *for* and *while* loops in C programs will not work. Note that for general purpose processors, we are looking at all possible programs. However, a lot of processors for embedded devices have limited functionality. For example, a simple processor that does string processing does not require support for floating point numbers (numbers with a decimal point). We need to note that different processors are designed to do different things, and hence their ISAs can be different. However, the bottom line is that any ISA should be *complete* in the sense that it should be able to express all the programs in machine code that a user intends to write for it.

Let us now explore the desirable properties of an instruction set.

1.5.2 Concise – Limited Size of the Instruction Set

We should ideally not have a lot of instructions. We shall see in Chapter 8 that it takes a fairly non-trivial amount of hardware to implement an instruction. Implementing a lot of instructions will unnecessarily increase the number of transistors in the processor and increase its complexity. Consequently, most instruction sets have somewhere between 64 to 1000 instructions. For example, the MIPS instruction set contains 64 instructions, whereas the Intel x86 instruction set has roughly a 1000 instructions as of 2012. Note that 1000 is considered a fairly large number for the number of instructions in an ISA.

1.5.3 Generic – Instructions should Capture the Common Case

Most of the common instructions in programs are simple arithmetic instructions such as add, subtract, multiply, divide. The most common logical instructions are logical and, or, exclusive-or, and not. Hence, it makes sense to dedicate an instruction to each of these common operations.

It is not a good idea to have instructions that implement a very rarely used computation. For example, it might not make sense to implement an instruction that computes $sin^{-1}(x)$. It is possible to provide dedicated library functions that compute $sin^{-1}(x)$ using existing mathematical techniques such as Taylor series expansion. Since this function is rarely used by most programs, they will not be adversely affected if this function takes a relatively long time to execute.

1.5.4 Simple – Instructions should be Simple

Let us assume that we have a lot of programs that add a sequence of numbers. To design a processor especially tailored towards such programs, we have several options with regards to the add instruction. We can implement an instruction that adds two numbers, or we can also implement an instruction that can take a list of operands, and produce the sum of the list. There is clearly a difference in complexity here, and we cannot say which implementation is faster. The former approach requires the compiler to generate more instructions; however, each add operation executes quickly. The latter approach generates a fewer number of instructions; but, each instruction takes longer to execute. The former type of ISA is called a *Reduced Instruction Set*, and the latter ISA type is called a *Complex Instruction Set*. Let us give two important definitions here.

Definition 4
A reduced instruction set computer (RISC) implements simple instructions that have a simple and regular structure. The number of instructions is typically a small number (64 to 128). Examples: ARM, IBM PowerPC, HP PA-RISC

Definition 5
A complex instruction set computer (CISC) implements complex instructions that are highly irregular, take multiple operands, and implement complex functionalities. Secondly, the number of instructions is large (typically 500+). Examples: Intel x86, VAX

The RISC vs CISC debate used to be a very contentious issue till the late nineties. However, since then designers, programmers, and processor vendors have been tilting towards the RISC design style. The

consensus seems to be go for a small number of relatively simple instructions that have a regular structure and format. It is important to note that this point is still debatable as CISC instructions are sometimes preferable for certain types of applications. Modern processors typically use a hybrid approach where they have simple, as well as some complicated instructions. However, under the hood CISC instructions are translated into RISC instructions. Hence, we believe that the scale tilts slightly more towards RISC instructions. We shall thus consider it a desirable property to have *simple* instructions.

Important Point 1
An ISA needs to be complete, concise, generic, and simple. It is necessary to be complete, whereas the rest of the properties are desirable (and sometimes debatable).

Way Point 1
We have currently considered the following concepts.

- *Computers are dumb yet ultra-fast machines.*
- *Instructions are basic rudimentary commands used to communicate with the processor. A computer can execute billions of instructions per second.*
- *The compiler transforms a user program written in a high level language such as C to a program consisting of basic machine instructions.*
- *The instruction set architecture(ISA) refers to the semantics of all the instructions supported by a processor.*
- *The instruction set needs to be complete. It is desirable if it is also concise, generic, and simple.*

Let us subsequently look at the conditions that ensure the completeness of an ISA. We will then try to create a concise, simple, and generic ISA in Chapter 3.

1.6 How to Ensure that an ISA is Complete?

This is a very interesting, difficult, and theoretically profound question. The problem of finding if a given ISA is complete for a given set of programs, is a fairly difficult problem, and is beyond the scope of the book. The general case is far more interesting. We need to answer the question:

Question 3
Given an ISA, can it represent all possible programs?

We will need to take recourse to theoretical computer science to answer this question. Casual readers can skip Sections 1.6.1 to 1.6.6 without any loss in continuity. They can directly proceed to Section 1.6.7, where we summarise the main results.

1.6.1 Towards a Universal ISA*

Let us try to answer Question 3. Assume that we are given an ISA that contains the basic instructions add, and multiply. Can we use this ISA to run all possible programs? The answer is no, because we cannot subtract two numbers using the basic instructions that we have. If we add the subtract instruction to our repertoire of instructions, can we compute the square root of a number? Even if we can, is it guaranteed that we can do all types of computations? To answer such vexing questions we need to first define a *universal machine*.

Definition 6
A machine that can execute any program is known as a universal *machine.*

It is a machine that can execute all programs. We can treat each basic action of this machine as an instruction. Thus the set of actions of a universal machine is its ISA, and this ISA is complete. Consequently, when we say that an ISA is complete, it is the same as saying that we can build a universal machine exclusively based on the given ISA. Hence, we can solve the problem of completeness of an ISA by solving the problem of designing universal machines. They are dual problems. It is easier to reason in terms of universal machines. Hence, let us delve into this problem.

Computer scientists started pondering at the design of universal machines at the beginning of the 20^{th} century. They wanted to know what is computable, and what is not, and the power of different classes of machines. Secondly, what is the form of a theoretical machine that can compute the results of all possible programs? These fundamental results in computer science form the basis of today's modern computer architectures.

Alan Turing was the first to propose a universal machine that was extremely simple and powerful. This machine is aptly named after him, and is known as the *Turing* machine. This is merely a theoretical entity, and is typically used as a mathematical reasoning tool. It is possible to create a hardware implementation of a Turing machine. However, this would be extremely inefficient, and require a disproportionate amount of resources. Nonetheless, Turing machines form the basis of today's computers and modern ISAs are derived from the basic actions of a Turing machine. Hence, it is very essential for us to study its design. Note that we provide a very cursory treatment in this book. Interested readers are requested to take a look at the seminal text on the theory of computation by Hopcroft, Motwani and Ulmann [Hopcroft et al., 2006].

1.6.2 Turing Machine*

The general structure of a Turing machine is shown in Figure 1.6. A Turing machine contains an infinite tape that is an array of cells. Each cell can contain a symbol from a finite alphabet. There is a special symbol \$ that works as a special marker. A dedicated tape head points to a cell in the infinite tape. There is a small piece of storage to save the current state among a finite set of states. This storage element is called a *state register*.

The operation of the Turing machine is very simple. In each step, the tape head reads the symbol in the current cell, its current state from the state register, and looks up a table that contains the set of actions for each combination of symbol and state. This dedicated table is called a *transition function table* or *action table*. Each entry in this table specifies three things – whether to move the tape head one step to the left or right, the next state, and the symbol that should be written in the current cell. Thus, in each step, the tape head can overwrite the value of the cell, change its state in the state register and move to a new cell. The only constraint is that the new cell needs to be to the immediate left or right of the current cell. Formally, its format is $(state, symbol) \rightarrow (\{L, R\}, new\_state, new\_symbol)$. L stands for left, and R stands for right.

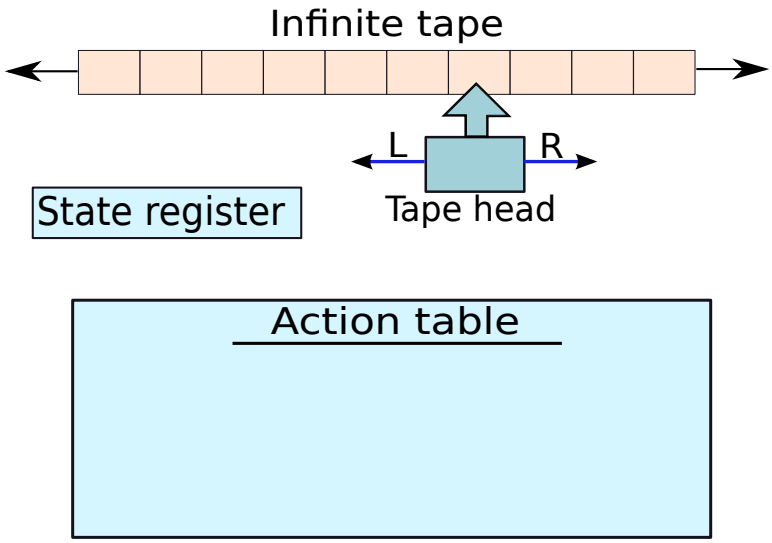
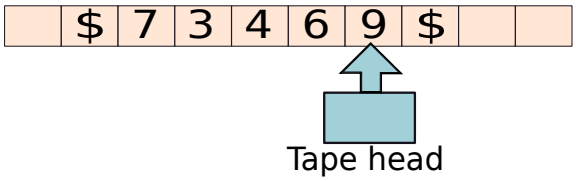


Figure 1.6: A Turing machine

This seemingly abstract and obscure computing device is actually very powerful. Let us explain with examples. See Examples 2, 3, and 4. In all the cases, we assume that the input is demarcated by the special marker symbol \$.

Example 2
Design a Turing machine to increment a number by 1.



Answer: *Each cell contains a single digit. The number is demarcated at both ends by the special marker \$. Lastly, the tape head points to the unit's digit.*

We first define four states (S_0, S_1)*:* pre-exit *and* exit*. The computation is over when the Turing machine reaches the* exit *state. The states* S_0 *and* S_1 *represent the value of the carry, 0 or 1, respectively. The state register is initialised to* S_1 *since we are incrementing the number by 1. In other words, we can assume that the starting value of the carry digit is equal to 1.*

At each step, the tape head reads the current digit, d, and the value of the carry, c, from the state register. For each combination of d, and c, the action table contains the next state (new value of carry), and the result digit. The tape head always moves to the left. For example, if $(d, c) = (9, 1)$*, then we are effectively adding (9 + 1). The next state is equal to* S_1 *(output carry), the Turing machine writes 0 in the current cell, and the tape head moves to the cell on the left.*

The only special case arises when the computation is ending. When the tape head encounters the \$ symbol, then it looks at the value of the carry. If it is equal to 0, then it leaves the value untouched and

moves to the exit *state. If it is equal to 1, then it moves to the* pre-exit *state, writes 1 to the cell, and moves to the left. Subsequently, it writes \$ to the cell under the tape head, and then moves to the* exit *state.*

Example 3
Design a Turing machine to find out if a string is of the form $aaa \ldots abb \ldots bb$.

Answer: *Let us define two states* (S_a, S_b)*, and two special states –* exit *and* error*. If the state becomes equal to* exit *or* error*, then the computation stops. The Turing machine can start scanning the input from right to left as Example 2. It starts in state* S_b*. The action table is as follows:*

$(S_b, b) \rightarrow (L, S_b, b)$
$(S_b, a) \rightarrow (L, S_a, a)$
$(S_b, \$) \rightarrow (L, error, \$)$
$(S_a, b) \rightarrow (L, error, b)$
$(S_a, a) \rightarrow (L, S_a, a)$
$(S_a, \$) \rightarrow (L, exit, \$)$

Example 4
Design a Turing machine to find out if a string of characters is a palindrome. A palindrome is a word that reads the same forward and backwards. Example: civic, rotator, rotor. Furthermore, assume that each character is either 'a' or 'b'.

Answer: *Let us assume that the Turing machine starts at the rightmost character in the* begin *state. Let us consider the case when the symbol under the tape head is a in the* begin *state. The machine enters the state* L_a*(move left, starting symbol is a) and replaces a with \$. Now it needs to see if the leftmost character is a. Hence, the tape head moves towards the left until it encounters \$. It then enters the* $Rcheck_a$ *state. It moves one cell to the right and checks if the symbol is equal to a. If it is a, then the string might be a palindrome. Otherwise, it is definitely not a palindrome and the procedure can terminate by entering the* error *state. The tape head again rewinds by moving all the way to the right and starts at the cell, which is to the immediate left of the starting cell in the previous round. The same algorithm is performed iteratively till either an error is encountered or all the symbols are replaced with \$.*

If the starting symbol was b, the procedure would have been exactly the same albeit with a different set of states – L_b *and* $Rcheck_b$*. The action table is shown below.*

$(begin, \$) \rightarrow (L, exit, \$)$
$(begin, a) \rightarrow (L, L_a, \$)$
$(L_a, a) \rightarrow (L, L_a, a)$
$(L_a, b) \rightarrow (L, L_a, b)$
$(L_a, \$) \rightarrow (R, Rcheck_a, \$)$
$(Rcheck_a, a) \rightarrow (R, Rmove, \$)$
$(Rcheck_a, b) \rightarrow (R, error, \$)$
$(Rmove, a) \rightarrow (R, Rmove, a)$
$(Rmove, b) \rightarrow (R, Rmove, b)$
$(Rmove, \$) \rightarrow (L, begin, \$)$

$(begin, b) \rightarrow (L, L_b, \$)$
$(L_b, a) \rightarrow (L, L_b, a)$
$(L_b, b) \rightarrow (L, L_b, b)$
$(L_b, \$) \rightarrow (R, Rcheck_b, \$)$
$(Rcheck_b, a) \rightarrow (R, error, \$)$
$(Rcheck_b, b) \rightarrow (R, Rmove, \$)$

In these examples we have considered three simple problems and designed Turing machines from them. We can immediately conclude that designing Turing machines for even simple problems is difficult, and cryptic. The action table can contain a lot of states, and quickly blow out of size. However, the baseline is that it is possible to solve complex problems with this simple device. It is in fact possible to solve all kinds of problems such as weather modelling, financial calculations, and solving differential equations with this machine!

Definition 7
Church-Turing thesis: Any real-world computation can be translated into an equivalent computation involving a Turing machine. (source: Wolfram Mathworld)

This observation is captured by the **Church-Turing thesis**, which basically says that all functions that are computable by any physical computing device are computable by a Turing machine. In lay man's terms, any program that can be computed by deterministic algorithms on any computer known to man, is also computable by a Turing machine.

This thesis has held its ground for the last half century. Researchers have up till now not been able to find a machine that is more powerful than a Turing machine. This means that there is no program that can be computed by another machine, and not by a Turing machine. There are some programs that might take forever to compute on a Turing machine. However, they would also take infinite time on all other computing machines. We can extend the Turing machine in all possible ways. We can consider multiple tapes, multiple tape heads, or multiple tracks in each tape. It can be shown that each of these machines is as powerful as a simple Turing machine.

1.6.3 Universal Turing Machine*

The Turing machine described in the Section 1.6.2 is not a universal machine. This is because it contains an action table, which is specific to the function being computed by the machine. A true universal machine will have the same action table, symbols, and also the same set of states for every function. We can make a universal Turing machine, if we can design a Turing machine that can simulate another Turing machine. This Turing machine will be generic and will not be specific to the function that is being computed.

Let the Turing machine that is being simulated be called $\mathcal{M}$, and the universal Turing machine be called $\mathcal{U}$. Let us first create a generic format for the action table of $\mathcal{M}$, and save it in a designated location on the

tape of $\mathcal{U}$. This *simulated action table* contains a list of actions, and each action requires the five parameters – old state, old symbol, direction(left or right), new state, new symbol. We can use a common set of basic symbols that can be the 10 decimal digits (0-9). If a function requires more symbols then we can consider one symbol to be contained in a set of contiguous cells demarcated by special delimiters. Let such a symbol be called a *simulated symbol*. Likewise, the state in the simulated action table can also be encoded as a decimal number. For the direction, we can use 0 for left, and 1 for right. Thus a single action table entry might look something like (@1334@34@0@1335@10@). Here the '@' symbol is the delimiter. This entry is saying that we are moving from state 1334 to 1335 if symbol 34 is encountered. We move left (0), and write a value of 10. Thus, we have found a way of encoding the action table, set of symbols, and states of a Turing machine designed to compute a certain function.

Similarly, we can designate an area of the tape to contain the state register of $\mathcal{M}$. We call this the simulated state register. Let the tape of $\mathcal{M}$ be given a dedicated space in the tape of $\mathcal{U}$, and let us call this space the *work area*.

The organisation is shown in Figure 1.7.

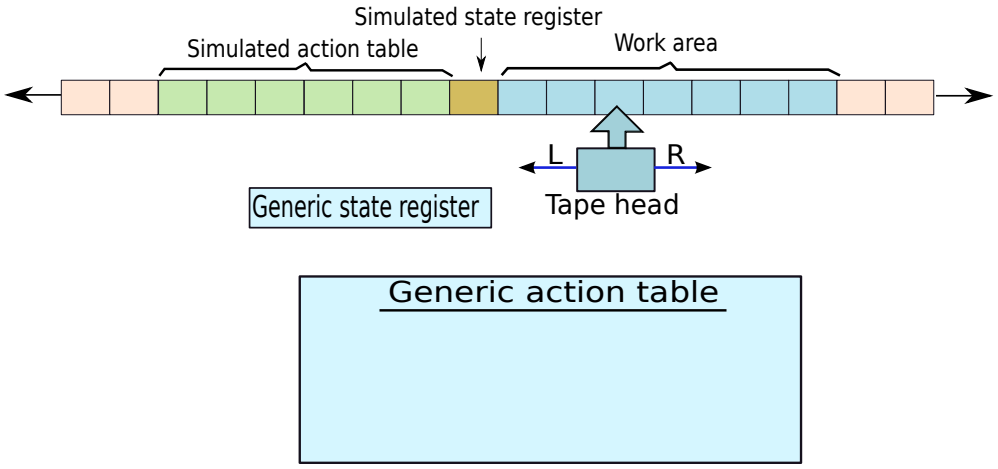


Figure 1.7: Layout of a universal Turing machine

The tape is thus divided into three parts. The first part contains the simulated action table, the second part contains the simulated state register, and the last part contains the work area that contains a set of simulated symbols.

The universal Turing machine($\mathcal{U}$) has a very simple action table and set of states. The idea is to find the right entry in the simulated action table that matches the value in the simulated state register and simulated symbol under the tape head. Then the universal Turing machine needs to carry out the corresponding action by moving to a new simulated state, and overwriting the simulated symbol in the work area if required.

The devil is in the details. For doing every basic action, $\mathcal{U}$ needs to do tens of tape head movements. The details are given in Hopcroft, Motwani, and Ulmann [Hopcroft et al., 2006]. However, the conclusion is that we can construct a universal Turing machine.

Important Point 2
It is possible to construct a universal Turing machine that can simulate any other Turing machine.

Turing Completeness

Since the 1950s, researchers have devised many more types of hypothetical machines with their own sets of states and rules. Each of these machines have been proven to be at most as powerful as the Turing machine. There is a generic name for all machines and computing systems that are as expressive and powerful as a Turing machine. Such systems are said to be *Turing complete*. Any universal machine and ISA is thus Turing complete.

Definition 8
Any computing system that is equivalent to a Turing machine is said to be Turing complete.

We thus need to prove that an ISA is complete or universal if it is Turing complete.

1.6.4 A Modified Universal Turing Machine*

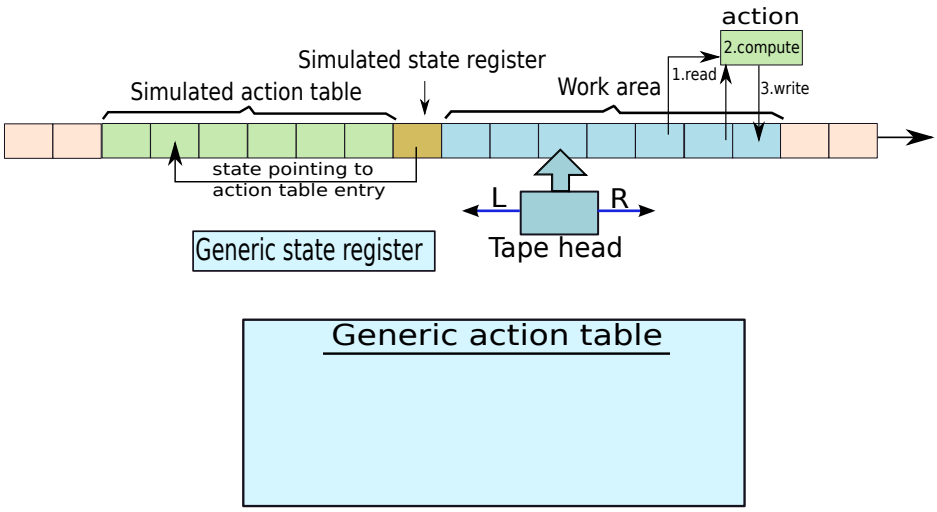


Figure 1.8: A modified universal Turing machine

Let us now consider a variant of a universal Turing machine (see Figure 1.8) that is more amenable to practical implementations. Let it have the following features. Note that such a machine has been proven to be Turing complete.

1. The tape is semi-infinite (extends to infinity in only one direction).
2. The simulated state is a pointer to an entry in the simulated action table.
3. There is one unique entry in the simulated action table for each state. While looking up the simulated action table, we do not care about the symbol under the tape head.
4. An action directs the tape head to visit a set of locations in the work area, and based on their values computes a new value using a simple arithmetical function. It writes this new value into a new location in the work area.

5. The default next state is the succeeding state in the action table.

6. An action can also arbitrarily change the state if a symbol at a certain location on the tape is less than a certain value. Changing the state means that the simulated tape head will start fetching actions from a new area in the simulated action table.

This Turing machine suggests a machine organisation of the following form. There is a large array of instructions (action table). This array of instructions is commonly referred to as the *program*. There is a state register that maintains a pointer to the current instruction in the array. We can refer to this register as the *program counter*. It is possible to change the program counter to point to a new instruction. There is a large work area, where symbols can be stored, retrieved and modified. This work area is also known as the *data* area. The instruction table (program) and the work area (data) were saved on the tape in our modified Turing machine. In a practical machine, we call this infinite tape as the *memory*. The memory is a large array of memory cells, where a memory cell contains a basic symbol. A part of the memory contains the program, and another part of it contains data.

Definition 9
The memory in our conceptual machine is a semi-infinite array of symbols. A part of it contains the program consisting of basic instructions, and the rest of it contains data. Data *refers to variables and constants that are used by the program.*

Furthermore, each instruction can read a set of locations in the memory, compute a small arithmetic function on them, and write the results back to the memory. It can also jump to any other instruction depending on values in the memory. There is a dedicated unit to compute these arithmetic functions, write to memory, and jump to other instructions. This is called the *CPU*(Central Processing Unit). Figure 1.9 shows a conceptual organisation of this machine.

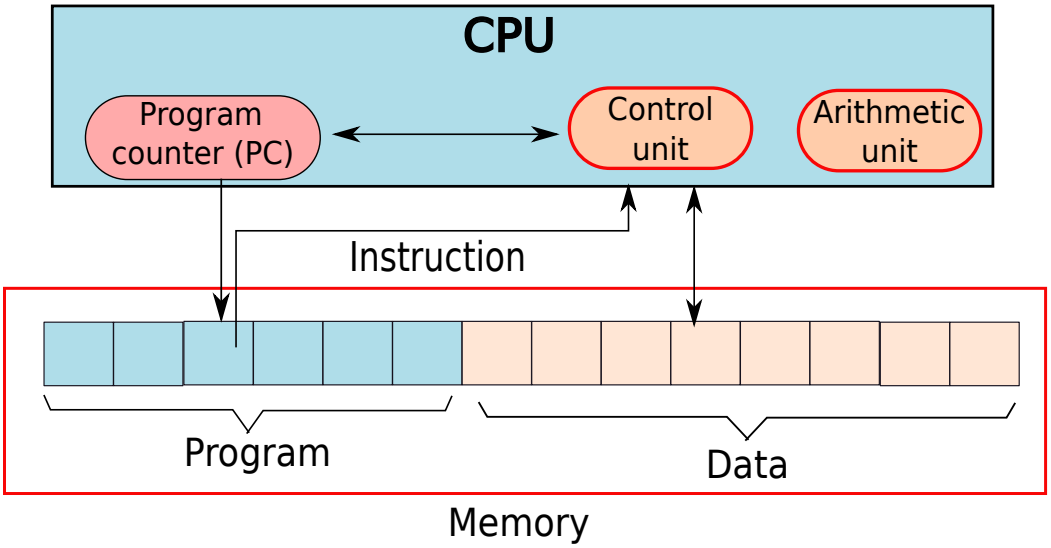


Figure 1.9: A basic instruction processing machine

Interested readers might want to prove the fact that this machine is equivalent to a Turing machine. It is not very difficult to do so. We need to note that we have captured all aspects of a Turing machine: state transition, movement of the tape head, overwriting symbols, and decisions based on the symbol under the tape head. We shall see in Section 1.7.2 that such a machine is very similar to the Von Neumann machine. Von Neumann machines form the basis of today's computers. Readers can also refer to books on computational complexity.

Important Point 3
Figure 1.9 represents a universal machine that can be practically designed.

1.6.5 Single Instruction ISA*

Let us now try to design an ISA for our modified Turing machine. We shall see that it is possible to have a complete ISA that contains just a single instruction. Let us consider an instruction that is compatible with the modified Turing machine and has been proven to be Turing complete.

```
sbn a, b, c
```

sbn stands for subtract and branch if negative. Here, a, and b are memory locations. This instruction subtracts b from a, saves the results in a, and if $a < 0$, it jumps to the instruction at location c in the instruction table. Otherwise, the control transfers to the next instruction. For example, we can use this instruction to add two numbers saved in locations a and b. Note that *exit* is a special location at the end of the program.

```
1: sbn temp, b, 2
2: sbn a, temp, exit
```

Here, we assume that the memory location *temp* already contains the value 0. The first instruction saves $-b$ in *temp*. Irrespective of the value of the result it jumps to the next instruction. Note that the identifier (*number* :) is a sequence number for the instruction. In the second instruction we compute $a = a + b = a - (-b)$. Thus, we have successfully added two numbers. We can now use this basic piece of code to add the numbers from 1 to 10. We assume that the variable *counter* is initialised to 9, *index* is initialised to 10, *one* is initialised to 1, and *sum* is initialised to 0.

```
1: sbn temp, temp, 2   // temp = 0
2: sbn temp, index, 3  // temp = -1 * index
3: sbn sum, temp, 4    // sum += index
4: sbn index, one, 5   // index -= 1
5: sbn counter, one, exit  // loop is finished, exit
6: sbn temp, temp, 7   // temp = 0
7: sbn temp, one, 1    // (0 - 1 < 0), hence goto 1
```

We observe that this small sequence of operations runs a *for* loop. The exit condition is in line 5, and the loop back happens in line 7. In each iteration it computes – $sum+ = index$.

There are many more similar single instruction ISAs that have been proven to be complete such as subtract and branch if less than equal to, reverse subtract and skip if borrow, and a computer that has generic memory move operations. The interested reader can refer to the book by Gilreath and Laplante [Gilreath and Laplante, 2003].

1.6.6 Multiple Instruction ISA*

Writing a program with just a single instruction is very difficult, and programs tend to be very long. There is no reason to be stingy with the number of instructions. We can make our life significantly easier by considering a multitude of instructions. Let us try to break up the basic *sbn* instructions into several instructions.

Arithmetic Instructions We can have a set of arithmetic instructions such as add, subtract, multiply and divide.

Move Instructions We can have move instructions that move values across different memory locations. They should allow us to also load constant values into memory locations.

Branch Instructions We require branch instructions that change the program counter to point to new instructions based on the results of computations or values stored in memory.

Keeping these basic tenets in mind, we can design many different types of complete ISAs. The point to note is that we definitely need three types of instructions – arithmetic (data processing), move (data transfer), and branch (control).

Important Point 4
In any instruction set, we need at least three types of instructions:

1. *We need arithmetic instructions to perform operations such as add, subtract, multiply, and divide. Most instruction sets also have specialised instructions in this category to perform logical operations such as logical OR and NOT.*
2. *We need data transfer instructions that can transfer values between memory locations and can load constants into memory locations.*
3. *We need branch instructions that can start executing instructions at different points in the program based on the values of instruction operands.*

1.6.7 Summary of Theoretical Results

Let us summarise the main results that we have obtained from our short discussion on theoretical computer science.

1. The problem of designing a complete ISA is the same as that of designing a *universal machine*. A universal machine can run any program. We can map each instruction in the ISA to an action in this universal machine. A universal machine is the most powerful computing machine known to man. If a universal machine cannot compute the result of a program because it never terminates (infinite loop), then all other computing machines are also guaranteed to fail for this program.

2. Universal machines have been studied extensively in theoretical computer science. One such machine is the Turing machine named after the father of computer science – Alan Turing.

3. The Turing machine is a very abstract computing device, and is not amenable to practical implementations. A practical implementation will be very slow and consume a lot of resources. However, machines equivalent to it can be much faster. Any such machine, ISA, and computing system that is equivalent to a Turing machine is said to be *Turing complete*.

4. We defined a modified Turing machine that is Turing complete in Section 1.6.4. It has the structure shown in Figure 1.10. Its main parts and salient features are as follows.

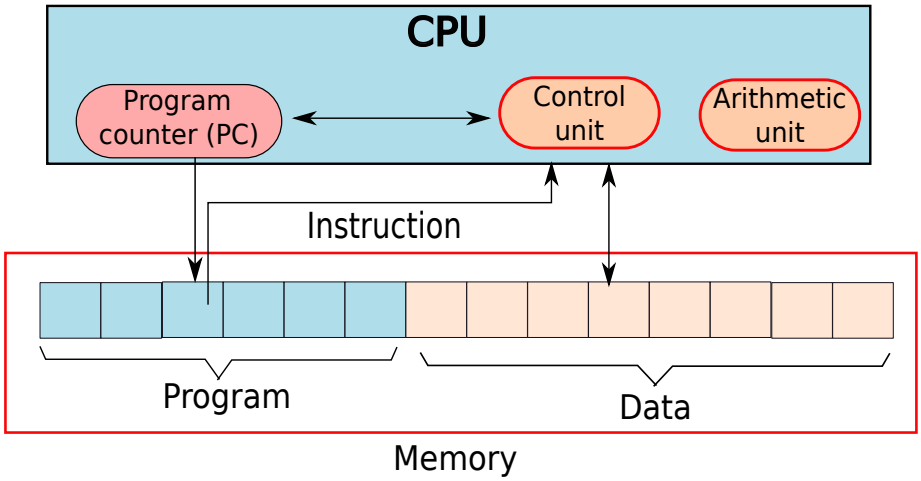


Figure 1.10: A basic processing machine

   (a) It contains a dedicated *instruction table* that contains a list of instructions.

   (b) It has a *program counter* that keeps track of the current instruction that is being executed. The program counter contains a pointer to an entry in the instruction table.

   (c) It has a semi-infinite array of storage locations that can save symbols belonging to a finite set. This array is known as the *memory*.

   (d) The memory contains the instruction table (also referred to as the *program*), and contains a data area. The *data* area saves all the variables and constants that are required by the program.

   (e) Each instruction can compute the result of a simple arithmetic function using values stored at different memory locations. It can then save the result in another memory location.

   (f) The machine starts with the first instruction in the program, and then by default, after executing an instruction, the machine fetches the next instruction in the instruction table.

   (g) It is possible for an instruction to direct the machine to fetch a new instruction from an arbitrary location in the instruction table based on the value stored in a memory location.

5. A simple one instruction ISA that is compatible with our modified Turing machine, contains the single instruction *sbn* (subtract the values of two memory locations, and branch to a new instruction if the result is negative).

6. We can have many Turing complete ISAs that contain a host of different instructions. Such ISAs will need to have the following types of instructions.

Arithmetic Instructions Add, subtract, multiply and divide. These instructions can be used to simulate logical instructions such as OR and AND.

Move Instructions Move values across memory locations, or load constants into memory.

Branch Instructions Fetch the next instruction from a new location in the instruction table, if a certain condition on the value of a memory location holds.

1.7 Design of Practical Machines

A broad picture of a practical machine has emerged from our discussion in Section 1.6.7. We have summarised the basic structure of such a machine in Figure 1.10. Let us call this machine as the *concept* machine. Ideas similar to our concept machine were beginning to circulate in the computer science community after Alan Turing published his research paper proposing the Turing machine in 1936. Several scientists got inspired by his ideas, and started pursuing efforts to design practical machines.

1.7.1 Harvard Architecture

One of the earliest efforts in this direction was the Harvard Mark-I. The Harvard architecture is very similar to our concept machine shown in Figure 1.10. Its block diagram is shown in Figure 1.11. There are separate structures for maintaining the instruction table and the memory. The former is also known as *instruction memory* because we can think of it as a specialised memory tailored to hold only instructions. The latter holds data values that programs need. Hence, it is known as the *data memory*. The engine for processing instructions is divided into two parts – control and ALU. The job of the control unit is to fetch instructions, process them, and co-ordinate their execution. ALU stands for arithmetic-logic-unit. It has specialised circuits that can compute arithmetic expressions or logical expressions (AND/OR/NOT etc.).

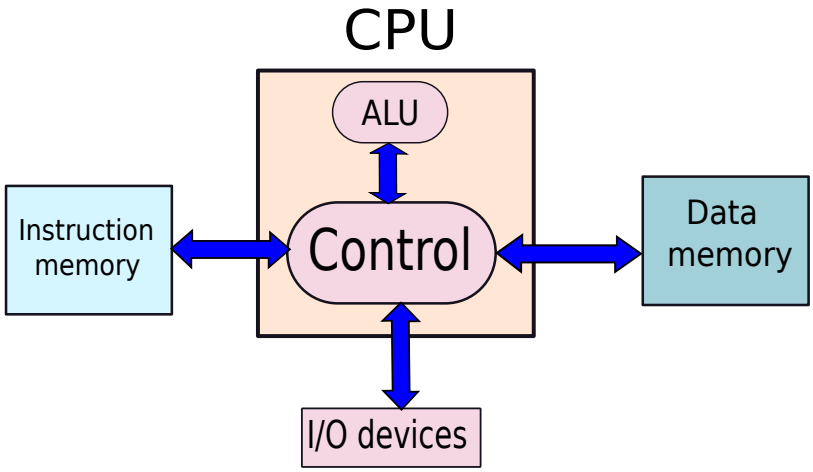


Figure 1.11: The Harvard architecture

Note that every computer needs to take inputs from the user/programmer and needs to finally communicate results back to the programmer. This can be done through a multitude of methods. Today we use a keyboard and monitor. Early computers used a set of switches and the final result was printed out on a piece of paper.

1.7.2 Von Neumann Architecture

John von Neumann proposed the Von Neumann architecture for general purpose Turing complete computers. Note that there were several other scientists such as John Mauchly and J. Presper Eckert who independently developed similar ideas. Eckert and Mauchly designed the first general purpose Turing complete computer(with one minor limitation) called ENIAC (Electronic Numerical Integrator and Calculator) based on this architecture in 1946. It was used to compute artillery firing tables for the US army's ballistic research laboratory. This computer was later succeeded by the EDVAC computer in 1949, which was also used by the US army's ballistics research laboratory.

The basic Von Neumann architecture, which is the basis of ENIAC and EDVAC is shown in Figure 1.12. This is pretty much the same as our concept machine. The instruction table is saved in memory. The processing engine that is akin to our modified Turing machine is called the CPU (central processing unit). It contains the program counter. Its job is to fetch new instructions, and execute them. It has dedicated functional units to calculate the results of arithmetic functions, load and store values in memory locations, and compute the results of branch instructions. Lastly, like the Harvard architecture, the CPU is connected to the I/O subsystem.

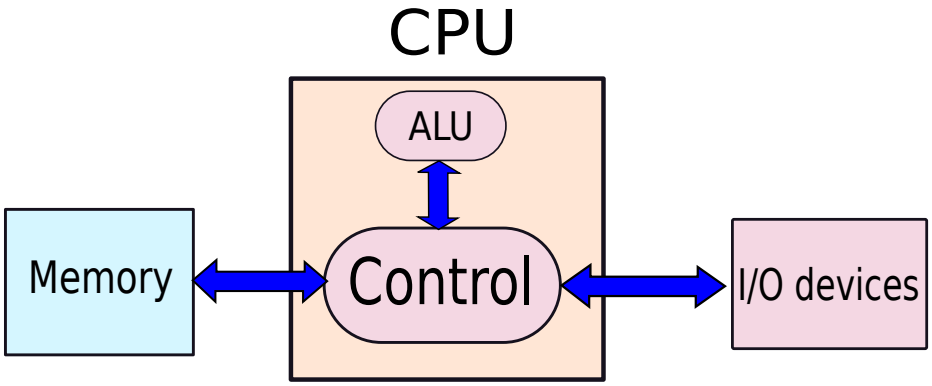


Figure 1.12: Von Neumann architecture

The path breaking innovation in this machine was that the instruction table was stored in memory. It was possible to do so by encoding every instruction with the same set of symbols that are normally stored in memory. For example, if the memory stores decimal values, then each instruction needs to be encoded into a string of decimal digits. A Von Neumann CPU needs to decode every instruction. The crux of this idea is that instructions are treated as regular data(memory values). We shall see in later chapters that this simple idea is actually a very powerful tool in designing elegant computing systems. This idea is known as the **stored program concept**.

Definition 10
Stored-program concept: A program is stored in memory and instructions are treated as regular memory values.

The stored program concept tremendously simplifies the design of a computer. Since memory data and instructions are conceptually treated the same way, we can have one unified processing system and memory system that treats instructions and data the same way. From the point of view of the CPU, the program

counter points to a generic memory location whose contents will be interpreted as that of an encoded instruction. It is easy to store, modify, and transmit programs. Programs can also dynamically change their behavior during runtime by modifying themselves and even other programs. This forms the basis of today's complex compilers that convert high level C programs into machine instructions. Furthermore, a lot of modern systems such as the Java virtual machine dynamically modify their instructions to achieve efficiency.

Lastly, astute readers would notice that a Von Neumann machine or a Harvard machine do not have an infinite amount of memory like a Turing machine. Hence, strictly speaking they are not exactly equivalent to a Turing machine. This is true for all practical machines. They need to have finite resources. Nevertheless, the scientific community has learnt to live with this approximation.

1.7.3 Towards a Modern Machine with Registers and Stacks

Many extensions to the basic Von-Neumann machine have been proposed in literature. In fact this has been a hot field of study for the last half century. We discuss three important variants of Von Neumann machines that augment the basic model with registers, hardware stacks, and accumulators. The register based design is by far the most commonly used today. However, some aspects of stack based machines and accumulators have crept into modern register based processors also. It would be worthwhile to take a brief look at them before we move on.

Von-Neumann Machine with Registers

The term "register machine" refers to a class of machines that in the most general sense contain an unbounded number of named storage locations called registers. These registers can be accessed randomly, and all instructions use register names as their operands. The CPU accesses the registers, fetches the operands, and then processes them. However, in this section, we look at a hybrid class of machines that augment a standard Von Neumann machine with registers. A register is a storage location that can hold a symbol. These are the same set of symbols that are stored in memory. For example, they can be integers.

Let us now try to motivate the use of registers. The memory is typically a very large structure. In modern processors, the entire memory can contain billions of storage locations. Any practical implementation of a memory of this size is fairly slow in practice. There is a general rule of thumb in hardware, "large is slow, and small is fast." Consequently, to enable fast operation, every processor has a small set of registers that can be quickly accessed. The number of registers is typically between 8 and 64. Most of the operands in arithmetic and branch operations are present in these registers. Since programs tend to use a small set of variables repeatedly at any point of time, using registers saves many memory accesses. However, it sometimes becomes necessary to bring in memory locations into registers or writeback values in registers to memory locations. In those cases, we use dedicated *load* and *store* instructions that transfer values between memory and registers. Most programs have a majority of pure register instructions. The number of load and store instructions are typically about a third of the total number of executed instructions.

Let us give an example. Assume that we want to add the cubes of the numbers in the memory locations b and c, and we want to save the result in the memory location a. A machine with registers would need the following instructions. Assume that $r1$, $r2$, and $r3$ are the names of registers. Here, we are not using any specific ISA (the explanation is generic and conceptual).

```
1: r1 = mem[b]   // load b
2: r2 = mem[c]   // load c
3: r3 = r1 * r1  // compute b^2
4: r4 = r1 * r3  // compute b^3
5: r5 = r2 * r2  // compute c^2
```

```
6: r6 = r2 * r5  // compute c^3
7: r7 = r4 + r6  // compute b^3 + c^3
4: mem[a] = r7   // save the result
```

Here, *mem* is an array representing memory. We need to first load the values into registers, then perform arithmetic computations, and then save the result back in memory. We can see in this example that we are saving on memory accesses by using registers. If we increase the complexity of the computations, we will save on even more memory accesses. Thus, our execution with registers will get even faster. The resulting processor organisation is shown in Figure 1.13.

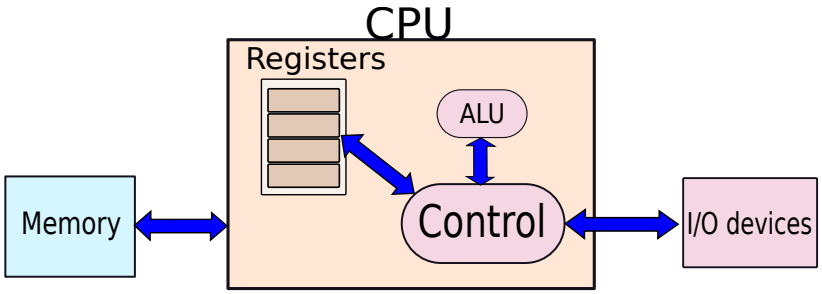


Figure 1.13: Von Neumann machine with registers

Von-Neumann Machine with a Hardware Stack

A stack is a standard data structure that obeys the semantics – last in, first out. Readers are requested to lookup a book on data structures such as [Lafore, 2002] for more information. A stack based machine has a stack implemented in hardware.

First, it is necessary to insert values from the memory into the stack. After that arithmetic functions operate on the top k elements of the stack. These values get replaced by the result of the computation. For example, if the stack contains the values 1 and 2 at the top. They get removed and replaced by 3. Note that here arithmetic operations do not require any operands. If an add operation takes two operands, then they do not need to be explicitly specified. The operands are implicitly specified as the top two elements in the stack. Likewise, the location of the result also does not need to be specified. It needs to be inserted at the top of the stack. Even though, generating instructions for such a machine is difficult and flexibility is an issue, the instructions can be very compact. Most instructions other than load and store do not require any operands. We can thus produce very dense machine code. Systems in which the size of the program is an issue can use a stack based organisation. They are also easy to verify since they are relatively simpler systems.

A stack supports two operations – *push* and *pop*. *Push* pushes an element to the top of the stack. *Pop* removes an element from the top of the stack. Let us now try to compute $w = x + y/z - u * v$ using a stack based Von Neumann machine, we have:

```
1: push u     // load u
2: push v     // load v
3: multiply   // u*v
4: push z     // load y
5: push y     // load z
6: divide     // y/z
```

```
7: subtract   // y/z - u*v
8: push x     // load x
9: add        // x + y/z - u*v
10: pop w     // store result in w
```

It is clearly visible that scheduling a computation to work on a stack is difficult. There will be many redundant loads and stores. Nonetheless, for machines that are meant to evaluate long mathematical expressions, and machines for which program size is an issue, typically opt for stacks. There are few practical implementations of stack based machines such as Burroughs Large Systems, UCSD Pascal, and HP 3000 (classic). The Java language assumes a hypothetical stack based machine during the process of compilation. Since a stack based machine is simple, Java programs can virtually run on any hardware platform. When we run a compiled Java program, then the Java Virtual Machine(JVM) dynamically converts the Java program into another program that can run on a machine with registers.

Accumulator based Machines

Accumulator based machines use a single register called an *accumulator*. Each instruction takes a single memory location as an input operand. For example, an *add* operation adds the value in the accumulator to the value in the memory address and then stores the result back in the accumulator. Early machines in the fifties that could not accommodate a register file used to have accumulators. Accumulators were able to reduce the number of memory accesses and speed up the program.

Some aspects of accumulators have crept into the Intel x86 set of processors that are the most commonly used processors for desktops and laptops as of 2012. For multiplication and division of large numbers, these processors use the register *eax* as an accumulator. For other generic instructions, any register can be specified as an accumulator.

1.8 The Road Ahead

We have outlined the structure of a modern machine in Section 1.7.3, which broadly follows a Von Neumann architecture, and is augmented with registers. Now, we need to proceed to build it. As mentioned at the outset, computer architecture is a beautiful amalgam of software and hardware. Software engineers tell us **what to build?** Hardware designers tell us **how to build?**

Let us thus first take care of the requirements of software engineers. Refer to the roadmap of chapters in Figure 1.14. The first part of the book will introduce computer architecture from the point of view of system software designers and application developers. Subsequently, we shall move on to designing processors, and lastly, we shall look at building a full systems of processors, memory elements, and I/O cum storage devices.

1.8.1 Representing Information

In modern computers, it is not possible to store numbers or pieces of text directly. Today's computers are made of transistors. A transistor can be visualised as a basic switch that has two states – on and off. If the switch is on, then it represents 1, otherwise it represents 0. Every single entity inclusive of numbers, text, instructions, programs, and complex software needs to be represented using a sequence of 0s and 1s. We have only two basic symbols that we can use namely 0 and 1. A variable/value that can either be 0 or 1, is known as a *bit*. Most computers typically store and process a set of 8 bits together. A set of 8 bits is known as a *byte*. Typically, a sequence of 4 bytes is known as a *word*.

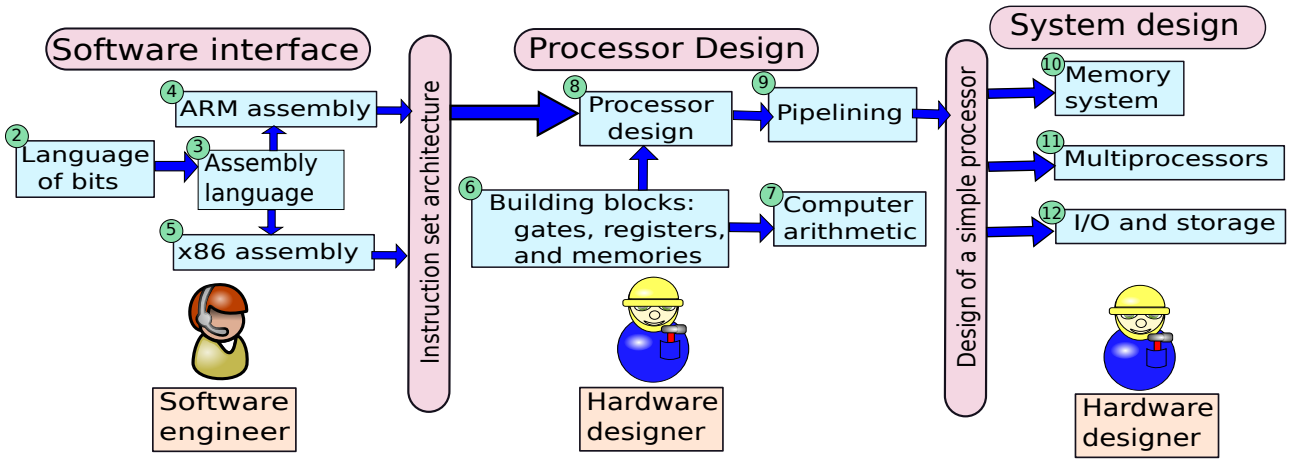


Figure 1.14: Roadmap of chapters

Definition 11

bit *A value that can either be 0 or 1.*

byte *A sequence of 8 bits.*

word *A sequence of 4 bytes.*

We can thus visualise all the internal storage structures of a computer such as the memory or the set of registers as a large array of switches as shown in Figure 1.15. In Chapter 2, we shall study the language of bits. We shall see that using bits it is possible to express logical concepts, arithmetic concepts (integer and real numbers), and pieces of text.

This chapter is a prerequisite for the next chapter on **assembly language**. Assembly language is a textual representation of an ISA. It is specific to the ISA. Since an instruction is a sequence of 0s and 1s, it is very difficult to study it in its bare form. Assembly language gives us a good handle to study the semantics of instructions in an ISA. Chapter 3 introduces the general concepts of assembly language and serves as a common introduction to the next two chapters that delve into the details of two very popular real world ISAs – ARM and x86. We introduce a simple ISA called *SimpleRisc* in Chapter 3. Subsequently, in Chapter 4 we introduce the ARM ISA, and in Chapter 5 we briefly cover the x86 ISA. Note that it is not necessary to read both these chapters. After reading the introductory chapter on assembly language and obtaining an understanding of the *SimpleRisc* assembly language, the interested reader can read just one chapter to deepen her knowledge about a real world ISA. At this point, the reader should have a good knowledge of what needs to be built.

1.8.2 Processing Information

In this part, we shall actually build a basic computer. Chapter 6 will start out with the basic building blocks of a processor – logic gates, registers, and memories. Readers who have already taken a digital design

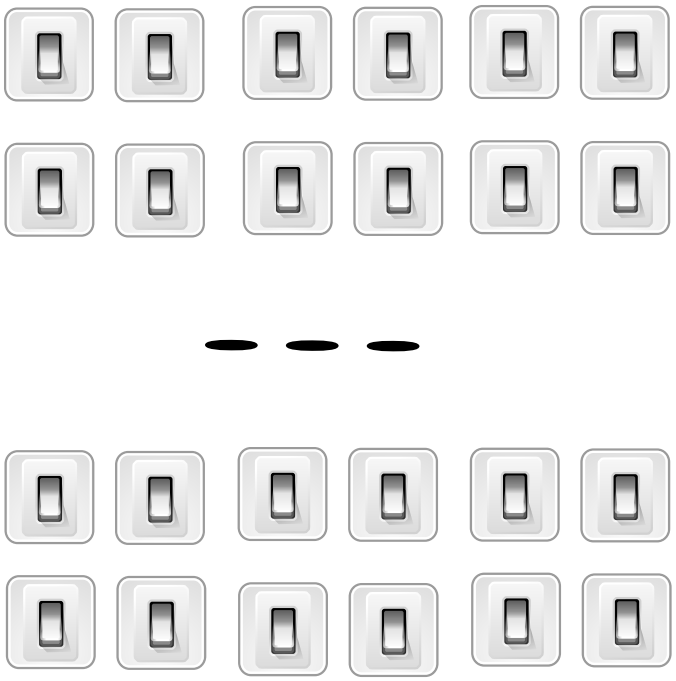


Figure 1.15: Memory – a large array of switches

course can skip this chapter. Chapter 7 deals with computer arithmetic. It introduces detailed algorithms for addition, subtraction, multiplication, and division for both integers as well as real numbers. Most computers today perform very heavy numerical computations. Hence, it is necessary to obtain a firm understanding of how numerical operations are actually implemented, and get an idea of the tradeoffs of different designs.

After these two chapters, we would be ready to actually design a simple processor in Chapter 8. We shall assemble a simple processor part by part, and then look at two broad design styles – hardwired, and microprogrammed. Modern processors are able to process many instructions simultaneously, and have complex logic for taking the dependences across instructions into account. The most popular technique in this area is known as *pipelining*. We shall discuss pipelining in detail in Chapter 9.

1.8.3 Processing More Information

By this point, we would have gotten a fair understanding of how simple processors are designed. We shall proceed to optimise the design, add extra components, and make a full system that can support all the programs that users typically want to run. We shall describe three subsystems –

Memory System We shall see in Chapter 10 that it is necessary to build a fast and efficient memory system, because it is a prime driver of performance. To build a fast memory system, we need to introduce many new structures and algorithms.

Multiprocessors Nowadays, vendors are incorporating multiple processors on a single chip. The future belongs to multiprocessors. The field of multiprocessors is very extensive and typically forms the core of an advanced architecture course. In this book, we shall provide a short overview of multiprocessors in Chapter 11.

I/O and Storage In Chapter 12, we shall look at methods to interface with different I/O devices, especially storage devices such as the hard disk. The hard disk saves all our programs and data when the computer is powered off, and it also plays a crucial role in supplying data to our programs during their operations. Hence, it is necessary to study the structure of the hard disk, and optimise it for performance and reliability.

1.9 Summary and Further Reading

1.9.1 Summary

Summary 1

1. *A computer is a dumb device as compared to the human brain. However, it can perform routine, simple and monotonic tasks, very quickly.*
2. *A computer is defined as a device that can be programmed to process information.*
3. *A* program *consists of basic instructions that need to be executed by a computer.*
4. *The semantics of all the instructions supported by a computer is known as the instruction set architecture (ISA).*
5. *Ideally, an ISA should be complete, concise, simple, and generic.*
6. *An ISA is complete, if it is equivalent to an universal Turing machine.*
7. *A practical implementation of any complete ISA requires:*
   (a) *A memory to hold instructions and data.*
   (b) *A CPU to process instructions and perform arithmetic and logical operations.*
   (c) *A set of I/O devices for communicating with the programmer.*
8. *Harvard and Von Neumann architectures are practical implementations of complete ISAs, and are also the basis of modern computers.*
9. *Modern processors typically have a set of registers, which are a set of named storage locations. They allow the processor to access data quickly by avoiding time consuming memory accesses.*
10. *Some early processors also had a stack to evaluate arithmetic expressions, and had accumulators to store operands and results.*

1.9.2 Further Reading

The field of computer architecture is a very exciting and fast moving field. The reader can refer to the books by Jan Bergstra [Bergstra and Middelburg, 2012] and Gilreath [Gilreath and Laplante, 2003] to learn more about the theory of instruction set completeness and classes of instructions. The book on formal languages by by Hopcroft, Motwani, and Ulmann [Hopcroft et al., 2006] provides a good introduction to Turing machines

and theoretical computer science in general. To get a historical perspective, readers can refer to the original reports written by Alan Turing [Carpenter and Doran, 1986] and John von Neumann [von Neumann, 1945].

Exercises

Processor and Instruction Set

Ex. 1 — Find out the model and make of at least 5 processors in devices around you. The devices can include desktops, laptops, cell phones, and tablet PCs.

Ex. 2 — Make a list of peripheral I/O devices for computers. Keyboards are mice are common devices. Search for uncommon devices. (HINT: joysticks, game controllers, fax machines)

Ex. 3 — What are the four properties of an instruction set?

Ex. 4 — Design an instruction set for a simple processor that needs to perform the following operations:

1.Add two registers

2.Subtract two registers

Ex. 5 — Design an instruction set for a simple processor that needs to perform the following operations:

1.Add two registers

2.Save a register to memory

3.Load a register from memory

4.Divide a value in a register by two

Ex. 6 — Design an instruction set to perform the basic arithmetic operations – add, subtract, multiply, and divide. Assume that all the instructions can have just one operand.

* **Ex. 7** — Consider the *sbn* instruction that subtracts the second operand from the first operand, and branches to the instruction specified by the label (third operand), if the result is negative. Write a small program using only the *sbn* instruction to compute the factorial of a positive number.

* **Ex. 8** — Write a small program using only the *sbn* instruction to test if a number is prime.

Theoretical Aspects of an ISA*

Ex. 9 — Explain the design of a modified Turing machine.

Ex. 10 — Prove that the *sbn* instruction is Turing complete.

Ex. 11 — Prove that a machine with memory load, store, branch, and subtract instructions is Turing complete.

** **Ex. 12** — Find out other models of universal machines from the internet and compare them with Turing Machines.

Practical Machine Models

Ex. 13 — What is the difference between the Harvard architecture and Von Neumann architecture?

Ex. 14 — What is a register machine?

Ex. 15 — What is a stack machine?

Ex. 16 — Write a program to compute $\mathbf{a} + \mathbf{b} + \mathbf{c} - \mathbf{d}$ on a stack machine.

Ex. 17 — Write a program to compute $\mathbf{a} + \mathbf{b} + (\mathbf{c} - \mathbf{d}) * \mathbf{3}$ on a stack machine.

Ex. 18 — Write a program to compute $(\mathbf{a} + \mathbf{b}/\mathbf{c}) * (\mathbf{c} - \mathbf{d}) + \mathbf{e}$ on a stack machine.

** **Ex. 19** — Try to search the internet, and find answers to the following questions.

1.When is having a separate instruction memory more beneficial?

2.When is having a combined instruction and data memory more beneficial?