1. Fundamentals of Computer Design

1.1 Introduction

The concept of stored program computers appeared in 1945 when John von Neumann drafted the first version of EDVAC (Electronic Discrete Variable Computer). Those ideas have since been the milestones of computers:

- an input device through which data and instructions can be entered
- storage in which data can be read/written; instructions are like data, they reside in the same memory
- an arithmetic unit to process data
- a control unit which fetches instructions, decode and execute them
- output devices for the user to access the results.

The improvements in computer technology have been tremendous since the first machines appeared. A personal computer that can be bought today with a few thousand dollars, has more performance (in terms of, say, floating point multiplications per second), more main memory and more disk capacity than a machine that cost millions in the 50s-60s.

Four lines of evolution have emerged from the first computers (definitions are very loose and in many case the borders between different classes are blurring):

1. **Mainframes**: large computers that can support very many users while delivering great computing power. It is mainly in mainframes where most of the innovations (both in architecture and in organization) have been made.

2. **Minicomputers**: have adopted many of the mainframe techniques, yet being designed to sell for less, satisfying the computing needs for smaller groups of users. It is the minicomputer group that improved at the fastest pace (since 1965 when DEC introduced the first minicomputer, PDP-8), mainly due to the evolution of integrated circuits technology (the first IC appeared in 1958).

3. **Supercomputers**: designed for scientific applications, they are the most expensive computers (over one million dollars), processing is usually done in batch mode, for reasons of performance.

4. **Microcomputers**: have appeared in the microprocessor era (the first microprocessor, Intel 4004, was introduced in 1971). The term *micro* refers only to physical dimensions, not to computing performance. A typical microcomputer (either a PC or a workstation) nicely fits on a desk. Microcomputers are a direct product of technological advances: faster CPUs, semiconductor memories, etc. Over the time many of the concepts previously used in mainframes and minicomputers have become common place in microcomputers.

For many years the evolution of computers was concerned with the problem of *object code compatibility*. A new architecture had to be, at least partly, compatible with older ones. Older programs (“the dusty deck”) had to run without changes on the new machines. A dramatic example is the IBM-PC architecture, launched in 1981, it proved so successful that further developments had to conform with the first release, despite the flaws which became apparent in a couple of years thereafter.

The assembly language is no longer the language in which new applications are written, although the most sensitive parts continue to be written in assembly language, and this is due to advances in languages and compiler technology.

The obsolescence of assembly language programming, as well as the creation of portable operating systems (like UNIX), have reduced the risks of introducing new architectures. New families of computers are emerging, many of them hybrids of “classical” families: graphical supercomputers, multiprocessors, MPP (Massively Parallel Processors), mini-supercomputers, etc.

1.2 Performance Definition

What is the meaning of saying that a computer is faster than another one? It depends upon the position you have: if you are a simple user (end user) then you say a computer is faster when it runs your program in less time, and you think at the time it takes from the moment you launch your program until you get the results, this the so called wall-clock time. On the other hand, if you are system's manager, then you say a computer is faster when it completes more jobs per time unit.

As a user you are interested in reducing the **response time** (also called the **execution time** or **latency**). The computer manager is more interested in increasing the **throughput** (also called **bandwidth**), the number of jobs done in a certain amount of time.

Response time, execution time and throughput are usually connected to tasks and whole computational events. Latency and bandwidth are mostly used when discussing about memory performance.

Example 1.1 EFFECT OF SYSTEM ENHANCEMENTS ON RESPONSE TIME, THROUGHPUT:

The following system enhancements are considered:
a) faster CPU
b) separate processors for different tasks (as in an airline reservation system or in a credit card processing system)
Do these enhancements improve response-time, throughput or both?

Answer:
A faster CPU decreases the response time and, in the mean time, increases the throughput
a) both the response-time and throughput are increased.

b) several tasks can be processed at the same time, but no one gets done faster; hence only the throughput is improved.

In many cases it is not possible to describe the performance, either response-time or throughput, in terms of constant values but in terms of some *statistical* distribution. This is especially true for I/O operations. One

can compute the best-case access time for a hard disk as well as the worst-case access time: what happens in real life is that you have a disk request and the completion time (response-time) which depends not only upon the hardware characteristics of the disk (best/worst case access time), but also upon some other facts, like what is the disk doing at the moment you are issuing the request for service, and how long the queue of waiting tasks is.

Comparing Performance

Suppose we have to compare two machines A and B. The phrase *A is n% faster than* B means:

$$\frac{\text{Execution time of B}}{\text{Execution time of A}} = 1 + \frac{n}{100}$$

Because performance is reciprocal to execution time, the above formula can be written as:

$$\frac{\text{Performance A}}{\text{Performance B}} = 1 + \frac{n}{100}$$

Example 1.2 COMPARISON OF EXECUTION TIMES:

If machine A runs a program in 5 seconds, and machine B runs the same program in 6 seconds, how can the execution times be compared?

Answer:
Machine A is faster than machine B by n% can be written as:

$$\frac{\text{Execution\_time\_B}}{\text{Execution\_time\_A}} = 1 + \frac{n}{100}$$

$$n = \frac{\text{Execution\_time\_B} - \text{Execution\_time\_A}}{\text{Execution\_time\_A}} * 100$$

$$n = \frac{6-5}{6} \times 100 = 16.7\,\%$$

Therefore machine A is by 16.7% faster than machine B. We can also say that the performance of the machine A is by 16.7% better than the performance of the machine B.

CPU Performance

What is the time the CPU of your machine is spending in running a program? Assuming that your CPU is driven by a constant rate clock generator (and this is sure the case), we have:

CPU_{time} = Clock_cycles_for_the_program * T_{ck}

where T_{ck} is the clock cycle time.

The above formula computes the time CPU spends running a program, not the elapsed time: it does not make sense to compute the elapsed time as a function of T_{ck}, mainly because the elapsed time also includes the I/O time, and the response time of I/O devices is not a function of T_{ck}.

If we know the number of instructions that are *execute*d since the program starts until the very end, lets call this the **Instruction Count** (IC), then we can compute the average number of **clock cycles per instruction** (CPI) as follows:

$$CPI = \frac{\text{Clock\_cycles\_per\_program}}{IC}$$

The CPU_{time} can then be expressed as:

$$CPU_{time} = IC * CPI * T_{ck}$$

The scope of a designer is to lower the CPU_{time}, and here are the parameters that can be modified to achieve this:

- IC: the instruction count depends on the instruction set architecture and the compiler technology
- CPI: depends upon machine organization and instruction set architecture. RISC tries to reduce the CPI
- T_{ck}: hardware technology and machine organization. RISC machines have lower T_{ck} due to simpler instructions.

Unfortunately the above parameters are not independent of each other so that changing one of them usually affects the others.

Whenever a designer considers an improvement to the machine (i.e. you

want a lower CPU_{time}) you must thoroughly check how the change affects other parts of the system. For example you may consider a change in organization such that CPI will be lowered, however this may increase T_{ck} thus offsetting the improvement in CPI.

A final remark: **CPI has to be measured** and not simply calculated from the system's specification. This is because CPI strongly depends of the memory hierarchy organization: a program running on the system without cache will certainly have a larger CPI than the same program running on the same machine but with a cache.

1.3 What Drives the Work of a Computer Designer

Designing a computer is a challenging task. It involves software (at least at the level of designing the instruction set), and hardware as well at all levels: functional organization, logic design, implementation. Implementation itself deals with designing/specifying ICs, packaging, noise, power, cooling etc.

It would be a terrible mistake to disregard one aspect or other of computer design, rather the computer designer has to design an optimal machine across all mentioned levels. You can not find a minimum unless you are familiar with a wide range of technologies, from compiler and operating system design to logic design and packaging.

Architecture is the art and science of building. Vitruvius, in the 1st century AD, said that architecture was a building that incorporated *utilitas, firmitas* and *venustas*, in English terms commodity, firmness and delight. This definition recognizes that architecture embraces functional, technological and aesthetic aspects.

Thus a computer architect has to specify the performance requirements of various parts of a computer system, to define the interconnections between them, and to keep it harmoniously balanced. The computer architect's job is more than designing the Instruction Set, as it has been understood for many years. The more an architect is exposed to all aspects of computer design, the more efficient she will be.

- the **instruction set architecture** refers to what the programmer sees as the machine's instruction set. The instruction set is the boundary between the hardware and the software, and most of the decisions concerning the instruction set affect the hardware, and the converse is also true, many hardware decisions may beneficially/adversely affect the instruction set.

- the **implementation** of a machine refers to the logical and physical design techniques used to implement an instance of the architecture. It is possible to have different implementations for some architecture, in the same way there are different possibilities to build a house using the same plans, but other materials and techniques. The implementation has two aspects:
    - the **organization** refers to logical aspects of an implementation. In other words it refers to the high level aspects of the design: CPU design, memory system, bus structure(s) etc.
    - the **hardware** refers to the specifics of an implementation. Detailed logic design and packaging are included here.

1.3.1 Qualitative Aspects of Design

Functional requirements

The table below presents some of the functional requirements a computer designer must bear in mind when designing a new system.

Functional requirements	Required features
Application area	
General purpose	Balanced performance
Scientific	Efficient floating point arithmetic
Commercial	Support for Cobol, data bases and transaction processing
Special purpose	High performance for specific tasks: DSP, functional programming, etc.
Software compatibility	
Object code	Frozen architecture; programs move easily from one machine to another without any investment.
High level language	Designer has maximum freedom; substantial effort in software (compilers) is needed
Operating system requirements	
Size of address space	Too low an address space may limit applications
Memory management	Flat, paged, segmented etc.
Protection	Page protection v. segment protection
Context switch	required to interrupt and restart programs
Interrupts	Hardware support, software support
Standards	
Buses	VME, SCSI, IPI etc.
Floating point	IEEE 754, IBM, DEC
Operating system	UNIX, DOS, Windows NT, OS/2, proprietary
Networks	Ethernet, FDDI, etc.
Programming languages	The choice will influence the instruction set

Balancing software and hardware

This is really a difficult task. You have chosen some functional requirements that must be met, and now have to **optimize** your design. To discuss about an optimum you have to choose some criteria to quantize the design, such that different versions can be compared. The most common metrics (criteria) are cost and performance although there are places where other requirements are important and must be taken into account: reliability and fault tolerance is of paramount importance in military, transaction processing, medical equipment, nuclear equipment, space and avionics, etc.

Sometimes certain hardware support is a must, you probably won't try to enter the scientific market without strong floating point hardware; ofcourse the floating point arithmetic can be implemented in software, but you can not compete with other vendors in this way. Other times it is not clear at all if certain functional requirements must be implemented in hardware (where it is presumed to run very fast), or in software, where the major advantages are easy design and debugging, simple upgradability, and low cost of errors.

Design today for the tomorrow's markets

Because the design of a new system may take from months to years, the architect must be aware of the rapidly improving implementation technologies. Here are some of the major hardware trends:

- **Integrated circuit technology**: transistor count on a chip increases by about 25% per year, thus doubling every three years. Device speed increases at almost the same pace.

- **Semiconductor RAM**: density increases by some 60% per year, thus quadrupling every three years; the cycle time has decreased very slow in the last decade, only by 33%.

- **Disk technology**: density increases by about 25% per year, doubling every three years. The access time has decreased only by one third in ten years.

A new design must support, not only the circuits that are available now, at the design time, which will become obsolete eventually, but also the circuits that will be available when the product gets into the market.

The designer must also be aware of the software trends:

- the **amount of memory** an average program requires has grown by a factor of 1.5 to 2 per year. In this rhythm the 32 bit address

space of the processors dominating the market today, will soon be exhausted. As a matter of fact, the most recently appeared designs, as DEC's Alpha, have shifted to larger address spaces: the virtual address of the Alpha architecture is 64 bit and various implementations must provide at least 43 bits of address.

- increased **dependency on compiler technology**: the compiler is now the major interface between the programmer and the machine. While in the pioneering era of computers a compiler had to deal with ill designed instruction sets, now the architectures move toward supporting efficient compiling and ease of writing compilers.

1.3.2 Quantitative aspects

Make the common case fast

This is a very simple to enounce principle: whenever you have to make a tradeoff **favor the frequent case over the infrequent one**.A common example is related to multiply/divide in a CPU: in most programs the multiplications (integer of float), by far exceed the number of divisions; it is therefore no wonder that many CPUs have hardware support for multiplication (at least for integers) while division is emulated in software. It is true that in such a situation the division is slow, but if it occurs rarely then the overall performance is improved by optimizing the common case (the multiplication).

This simple principle applies not only to hardware, but to software decisions as well: if you find that some addressing mode, for instance, is heavily used as compared with others than you may try to optimize your design such that this particular addressing mode will run faster, hence increasing the performance of the machine.

A common concern is, however to detect the common case and to compute the performance gain when this case is optimized. This is easy when you have to design a system for a dedicated application: its behavior can be observed and the possible optimizations can be applied. On the other hand, when you design a general purpose machine, you must very cautiously consider possible improvements for what seems to be the common case, if this slows down other parts of the machine; the system will run fine for applications that fit the common case, but results will be poor in other cases.

Amdahl's Law

Suppose you enhance somehow your machine, to make it run faster: the speedup is defined as:

$$\text{speedup} = \frac{T_{old}}{T_{new}}$$

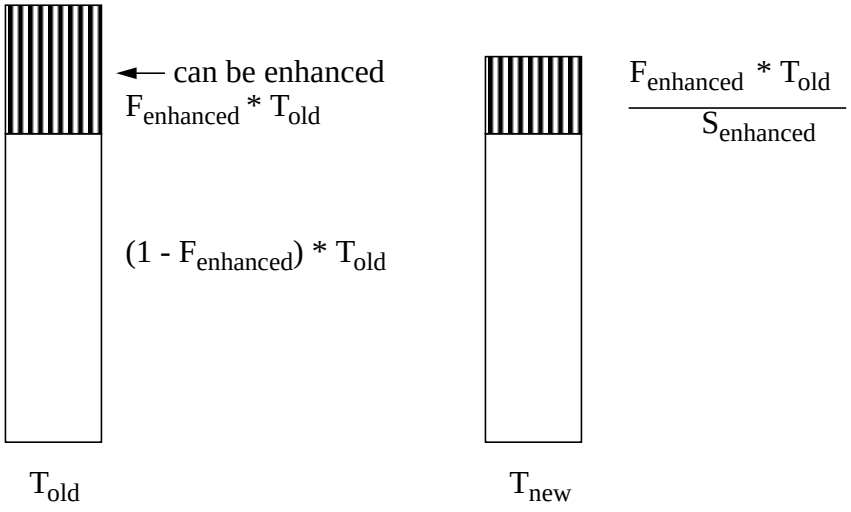
where T_{old} represent the execution time without the enhancement, and T_{new} is the execution time with the enhancement. In terms of performance the speedup can be defined as:

$$\text{speedup} = \frac{\text{performance for task using enhancement}}{\text{performance for task without enhancement}}$$

The Amdahl's law gives us a way to compute the speedup when an enhancement is used only some fraction of the time:

$$S_{overall} = \frac{1}{(1 - F_{enhanced}) + \frac{F_{enhanced}}{S_{enhanced}}} \qquad (1)$$

where $S_{overall}$ represents the overall speedup and $F_{enhanced}$ is the fraction of the time that the enhancement can be used (measured **before** the enhancement is applied).



As it can easily be seen the running time after the enhancement is applied:

$$T_{new} = (1 - F_{enhanced}) * T_{old} + \frac{F_{enhanced} * T_{old}}{S_{enhanced}}$$

from which the relation (1) can be easily derived.

Example 1.3 EFFECT OF SYSTEM ENHANCEMENTS ON OVERALL SPEEDUP:

Suppose you speed up all floating point multiplications by a factor of 10. At present floating point multiplications represent 20% of the running time of your program. Which is the overall speedup when you use the enhancement?

Answer:
$F_{enhanced} = 20\% = 0.2$
$S_{enhanced} = 10$

The result is a 22% increase in performance.

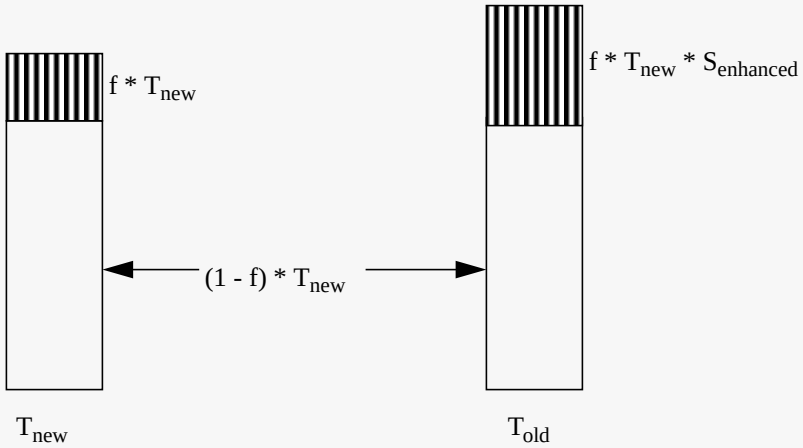
It is important to stress that, in using the Amdahl's law you have to use the fraction of time that can use the enhancement; it is a mistake to use relation (1) with a value of $F_{enhanced}$ that was measured after the enhancement has been in use.

The following example gives a relation for the overall speedup that uses the fraction of time the enhancement represents from the total, measured when the enhancement is in use.

Example 1.4 EFFECT OF SYSTEM ENHANCEMENTS ON OVERALL SPEEDUP:

Suppose you speed up all floating point operations by a factor of 10; the enhanced floating point operations represent f = 20% of the running time, with the enhancement in use. Which is the overall speedup?

Answer:



$$T_{old} = (1 - f) * T_{new} + f * T_{new} * S_{enhanced}$$

$$S_{overall} = \frac{T_{old}}{T_{new}} = (1 - f) + f * S_{enhanced}$$

$$S_{overall} = (1 - 0.2) + 0.2 * 10 = 2.8$$

As expected, using the same values but in different conditions yield different results (compare with the result in example 1.3).

Example 1.5 COST AND PERFORMANCE:

Suppose you have a machine used in an I/O intensive environment; the CPU is working 75% of the time and the rest is waiting for I/O operations to complete. You may consider an improvement of the CPU by a factor of 2 (it will run twice as fast as it runs now) for a fivefold increase in cost. The present cost of the CPU is 20% of the machine's cost. Is the suggested improvement cost effective?

Answer:

$$S_{enhanced} = 2$$

$$F_{enhanced} = 75\% = 0.75$$

The overall speedup is

$$S_{overall} = \frac{1}{(1 - 0.75) + \frac{0.75}{2}} = \frac{1}{0.623} = 1.6$$

The cost of the new machine would be Cnew:

$C_{new} = (1 - 0.2) * C_{old} + 0.2 * 5 * C_{old} = 1.8 * C_{old}$

Since the price increases by a factor of 1.8 while the performance increases only by a factor of 1.6 the improvement is not worth.

Locality of reference

A largely used property of programs is the **locality of reference**. This describes the fact that the addresses generated by a normal program, tend to be clustered to small regions of the total address space, as Figure 1.1 suggests.

The **90/10 rule of thumb** says that a program spends 90% of its execution time in only 10% of the code. There are two aspects of reference locality:

- **temporal locality**: refers to the fact that recently accessed items from memory are likely to be accessed again in the near future; loops in a program are a good illustration for temporal locality;

- **spatial locality**: items that are near to each other in memory tend to be referenced near one to another in time; data structures and arrays are good illustrations for spatial locality.

It is the locality of reference that allows us to build **memory hierarchies**.

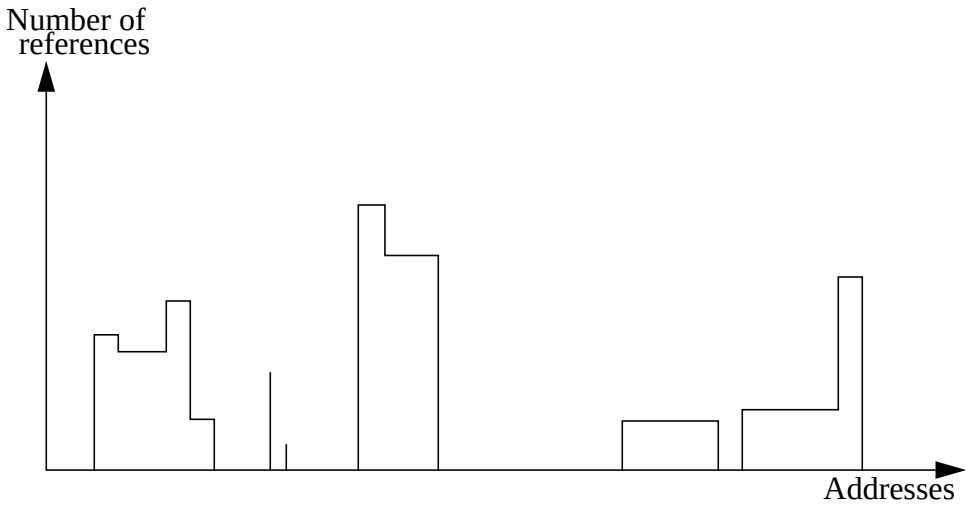


FIGURE 1.1 In a normal program, the number of references is not uniformly distributed over the whole address space.

Exercises

1.1 The 4 Mbit DRAM chip was introduced in 1990. When do you think the 64 Mbit DRAM chip will be available?

1.2 Suppose you have enhanced your machine with a floating point coprocessor; all floating point operations are faster by a factor of 10 when the coprocessor is in use:

a) what percent of the time should be spent in floating point operations such that the overall speedup is 2?

b) you know that 40% of the run-time is spent in floating point operations in the enhanced mode; you could buy new floating point hardware for a high price (10 times the price of your actual hardware for doubling the performance), or you may consider an improvement in software (compiler). How much should increase the percentage of floating point utilization, compared with the present usage, such that the increase in performance is the same as with the new hardware you could buy. Which investment is better; don't forget to state all your assumptions.

1.3 Compute the average CPI for a program running for 10 seconds (without I/O), on a machine that has a 50 MHz clock rate, if the number of instructions executed in this time is 150 millions?

1.4 Write a program which generates a uniform range of addresses; what are the problems you face in trying to write this program?