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:

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

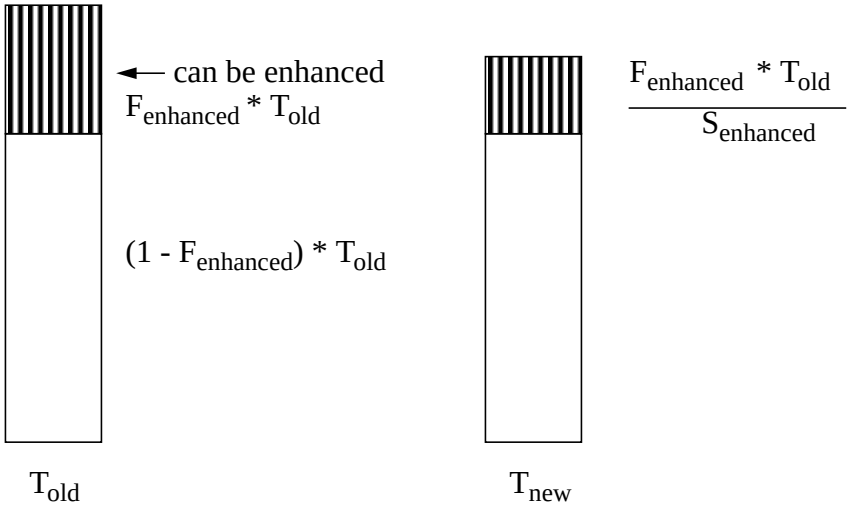
where T_{old} represent the execution time without the enhancement, and T_{new} is the execution time with the enhancement. In terms of performance the speedup can be defined as:

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

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.

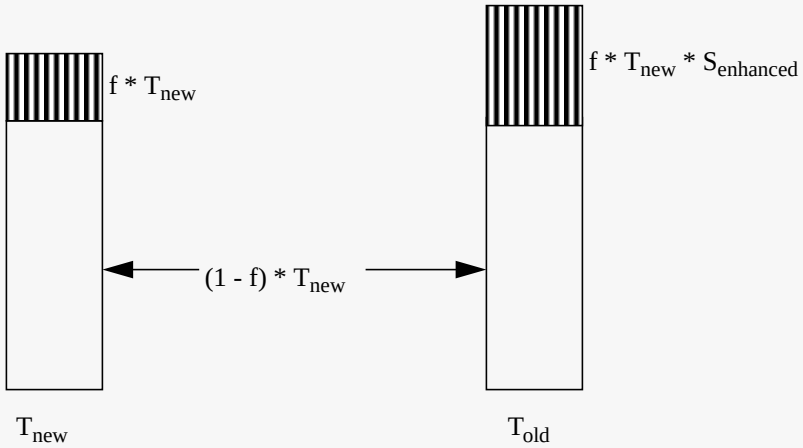
It is important to stress that, in using the Amdahl's law you have to use the fraction of time that can use the enhancement; it is a mistake to use relation (1) with a value of $F_{enhanced}$ that was measured after the enhancement has been in use.

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**.

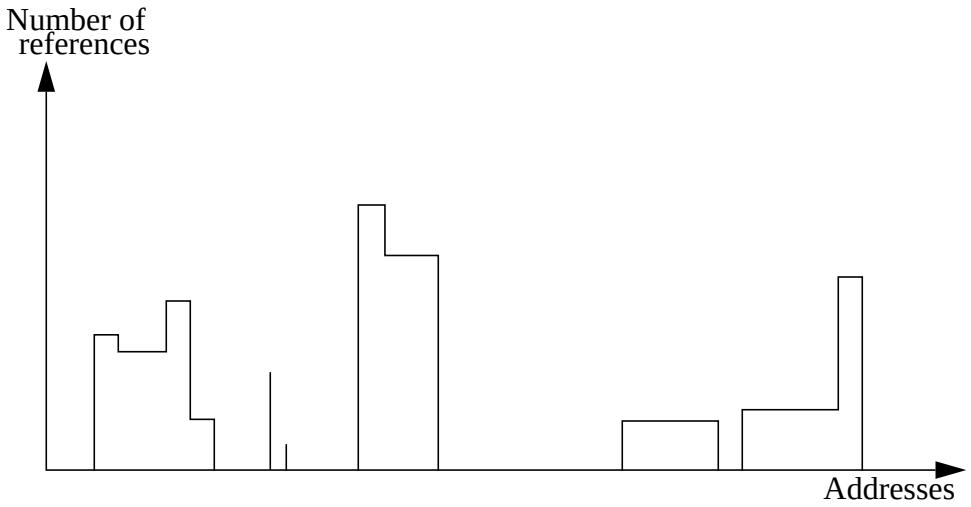


FIGURE 1.1 In a normal program, the number of references is not uniformly distributed over the whole address space.

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?

2. Basic Organization of a Computer

2.1 The block diagram

Most of the computers available today on the market are the so called **von Neumann computers**, simply because their main building parts, CPU or processor, memory, and I/O are interconnected the way von Neumann suggested. Figure 2.1 presents the basic building blocks of today's computers; even though there are many variations, and the level at which these blocks can be found is different, sometimes at the system level, other times at board level or even at chip level, their meaning is yet the same.

- **CPU** is the core of the computer: all *computation* is done here and the whole
- system is *controlled* by the CPU
- the program and the data for the program are stored in the **memory**
- **I/O** provide means of entering the program and data into the system. It also allows the user to get the results of the computation.

2.2 Computation and control in CPU

The computation part of the CPU, called the datapath, consists of the following units:

- **ALU** (Arithmetic and Logic Unit) which performs arithmetic and logic operations;
- **registers** to hold variables or intermediary results of computation, as well as special purpose registers;
- the **interconnections** between them (internal buses).

The datapath contains most of the CPU's **state**; this is the information the programmer has to save when the program is suspended; restoring this information makes the computation look like nothing had happened.

The state includes the user visible general purpose registers, as well as the **Program Counter** (PC: it contains the address of the next instruction to be executed), the **Interrupt Address Register** (IAR: contains the address of the instruction being suspended), and a **Program Status Register** (PSR: this usually holds the status flags for the machine, like condition codes, masks for interrupts, etc.).

With a few exceptions (like PC or IAR) there is no rule to indicate if some special signification register must be kept in the general purpose area (also called the **register-file**), or in a specially dedicated register. Should the stack-pointer or the frame-pointer, for instance, have special registers with dedicated hardware to help them perform the functions they are meant to, or they can simply reside in the register-file?

On one hand a structure without “special features” is “cleaner”, in the sense that it is easier to design and debug; on the other hand there are strong reasons to have special purpose registers, and the most important is *efficiency.* The PC, for example, is a special register, because it has a special function which could be otherwise impossible to perform: its content has to be incremented in each instruction with some value; special hardware helps optimizing this function; as a matter of fact, in many designs, the program counter is closer to a counter than to a simple D-type register.

Specialized hardware also means that some functions in the machine may execute in a parallel fashion, thus increasing the efficiency: using the same example, the program counter can be incremented while some register(s) in the register-file are read/written, and maybe a memory access is in progress.

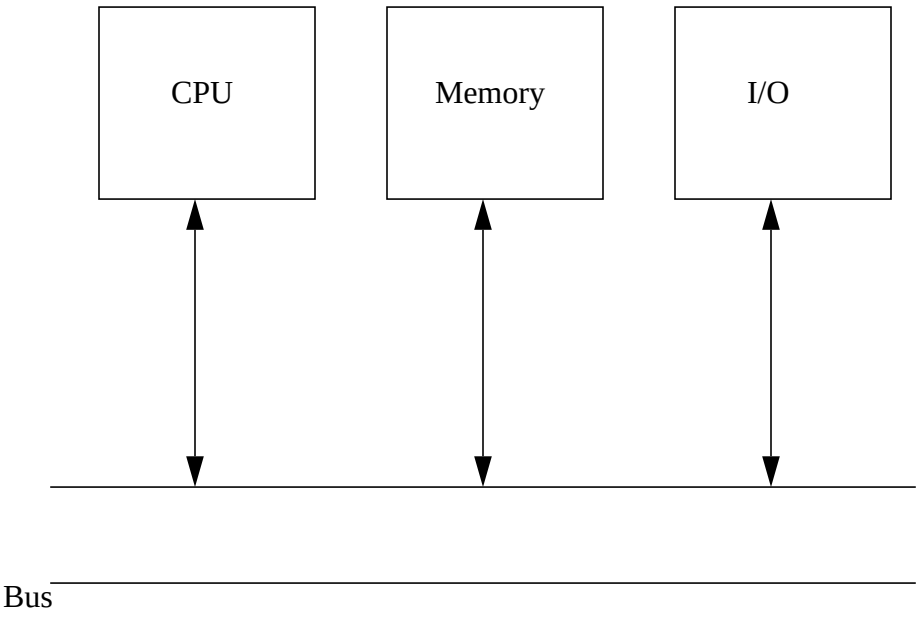


FIGURE 2.1 The building blocks of a computer.

It is also to be mentioned that some special registers can be accessed only by specialized instructions (in the case of PC only by jumps, call/return, branches, with all their variants), thus providing superior protection against accidental alteration, as compared with a general purpose register.

It can be long argued about what functions the ALU should perform, and there are at least two aspects to be considered:

- **encoding**: the operation to be performed in the CPU is somewhere encoded in the instruction, using a number of bits; with n bits one can specify 2n different binary configurations, i.e. that many ALU operations. If n is too small then it will be impossible to accommodate the minimum number of functions the ALU should perform; if the designer is too greedy then fewer bits will remain available to encode other information in the instruction (as for instance, where are the operands to be used, etc.), not to mention the explosive increase in the ALU's complexity. For the time being, three or four bits seem to be enough as control lines for the ALU.

- **functionality**: which is the best set of operations to implement, while keeping the design at reasonable dimensions, and, in the mean time without impairing the programmer's ability to implement any function from the basic set of functions we provide.

Example 2.1 IMPLEMENTATION OF OPERATIONS:

Assume that the instruction set has instructions with the following formats:

```
operation  destination, operand1, operand2
or
operation   destination, operand
```

where operation specifies what is to be performed with the operands operand1 and operand2, or with operand, and destination is the place where the result is to be stored. Suppose also that the only logic instructions are AND, OR, NOT. Show how to implement the XOR operation; the operands are in registers r1 and r2.

Answer: Use the relation:

A xor B = ($\overline{A}$ and B) or (A and $\overline{B}$)

The following sequence of code implements the XOR:

```
xor:  NOT   r3, r1           # the complement of A in r3
      AND   r3, r2, r3       # the first and
      NOT   r2, r2           # the complement of B in r2
      AND   r2, r1, r2       # the second and
      OR    r3, r2, r3       # final result in r3
```

Now let's consider another example in which the logic operations available are different from those in example 2.1.

Example 2.2 IMPLEMENTATION OF OPERATIONS:

Suppose you have the same instruction formats as in example 2.1, but the only available logic instructions are AND, OR, XOR. Implement the NOT operation; the operand is in register r1 and the content of register r0 is always zero.

Answer: Use the fact that:

A xor 1 = $\overline{A}$

The following sequence of code implements the NOT operation:

```
not:  SUB   r2, r0, 1        # make all 1's in r2
      XOR   r2, r2, r1       # final result in r2
```

The above sequence of code assumes that subtracting one from zero (integer substraction) yields a all ones result; this is true for unsigned and two's complement representation integers.

Example 2.2 points out that the common case has to be consider when choosing an instruction set; the efforts in design will probably go towards optimizing the common case. Certainly the designer could consider implementing both the NOT and XOR operations in the instruction set (i.e. to have corresponding instructions): same questions emerge again, are there enough opcodes to implement a new operation, and what is the hardware price we have to pay for it? Usually more hardware means a lower clock cycle and the specter of offsetting the overall performance.

It is now the time to discuss about interconnections inside the CPU and to sketch a CPU. Basically the question is *how many internal buses should the CPU have?*

If space/low-price are a must then a single internal bus may be considered as the one in figure 2.2. This approach has however a big drawback: little flexibility in choosing the instruction set; most operations have as an operand the content of the accumulator, and this is also the place where the result goes. Due to its simplicity (simple also means cheap!), this was the solution adopted by the first CPUs.

When we say simple we mean both hardware simplicity and software simplicity: because one operand is always in the accumulator, and the accumulator is also the destination, the instruction encoding is very simple: the instruction must only specify what is the operation to be performed and which is its second operand. Could the designers have encoded more than this in the first 8-bit integrated CPUs (the Intel 8080 or the Zilog Z80, the most popular 8-bit microprocessors, both appeared in the 70s)?

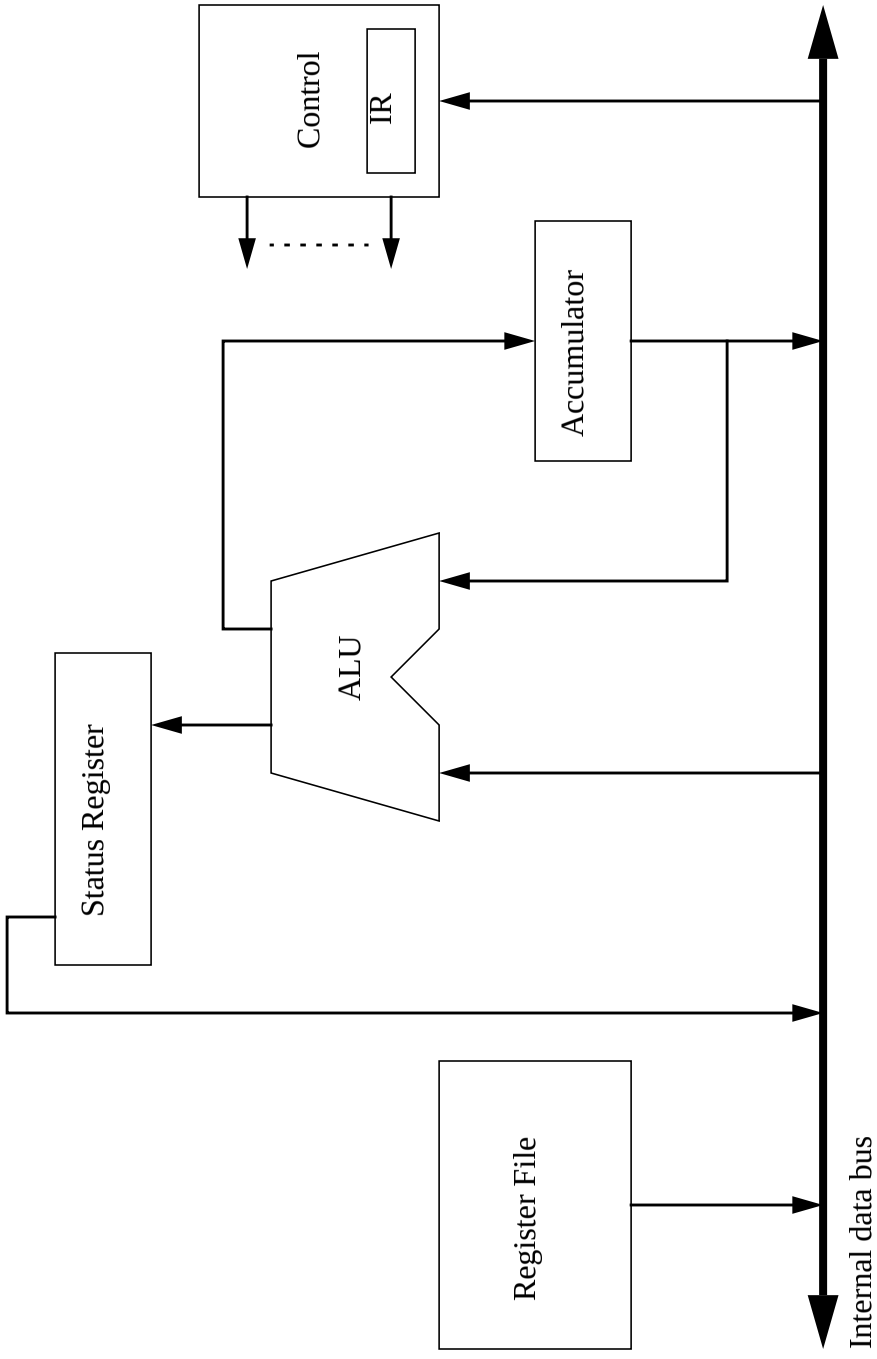


FIGURE 2.2 A possible organization of a CPU, using a single internal data bus. IR is the Instruction Register.

As the technology allowed to move to wider data paths (16, 32, 64 and even larger in the future), it has become also possible to specify more complex instruction formats: more explicit operands, more registers, larger offsets, etc. It is the moment to observe that newer CPU generations are faster due to:

- **faster clock rate** (lower T_{ck}); while the technology features decreased more transistors fit on the same surface and they may operate at higher speed;

- **lower IC**: it takes fewer instructions to perform an integer instruction on 32-bit integers, if the datapath is 32-bit wide as compared with an 8-bit datapath;

- **lower CPI**: with a more involved hardware it is possible to make large transfers (read/store from/to memory in a single clock cycle, instead of several ones as it was the case with narrower datapath CPUs.

Figure 2.3 presents a typical modern CPU, connected to memory. The CPU uses three buses (Op1, Op2 and Dest). The two operands are placed on the two buses, Op1, and Op2, an operation is performed, and the result gets on the Dest bus to be stored in any register connected to the bus.

- **MAR** is the Memory Address Register which holds the memory address during an instruction fetch on a load/store operation;

- **MDR** is the Memory Data Register, used to hold the data to be written into the memory during a store or to temporarily hold the data during a load;

- **temp** is a temporary register used for internal manipulation of data.

Figure 2.3 also assumes that the only way from a register to another is through ALU, therefore ALU must be able to, as one of its functions, pass one operand from input to output.

2.3 Instruction cycle

Obviously there are at least two steps in the cycle of an instruction: fetch (i.e. the instruction is brought into CPU, more precisely into IR) and execute.

At a closer look several substeps can be seen:

1. Instruction fetch step:
MAR ◄—— PC
IR ◄—— M [MAR]
The content of PC is transferred into MAR; then the instruction at address MAR is brought into IR.

2. Instruction decode / register fetch step:

Decoding the instruction is the step when the control decides what should be done next; if the instruction has a fixed fields format, then the contents of registers specified in the instruction can be read into A and B at the same time with the decoding.

It is also in this phase that PC has to be updated; how much is to be added to the PC in order to get the new PC (i.e. the address of next instruction to be executed)? Various factors are to be considered, like instruction width, byte/word addressable memory, memory alignment.

3. Execution

In the case of an arithmetic logic operation whose operands are in registers the operation is performed.

If the instruction is a load/store, then the address has to be computed and only then the operation can be performed.

In the case of a branch/jump operation the target address has to be computed and, for a conditional branch/jump, PC may be updated or not depending on the flag (condition) being tested.

The execution step could be further divided into specific substeps for each instruction or class of instructions.

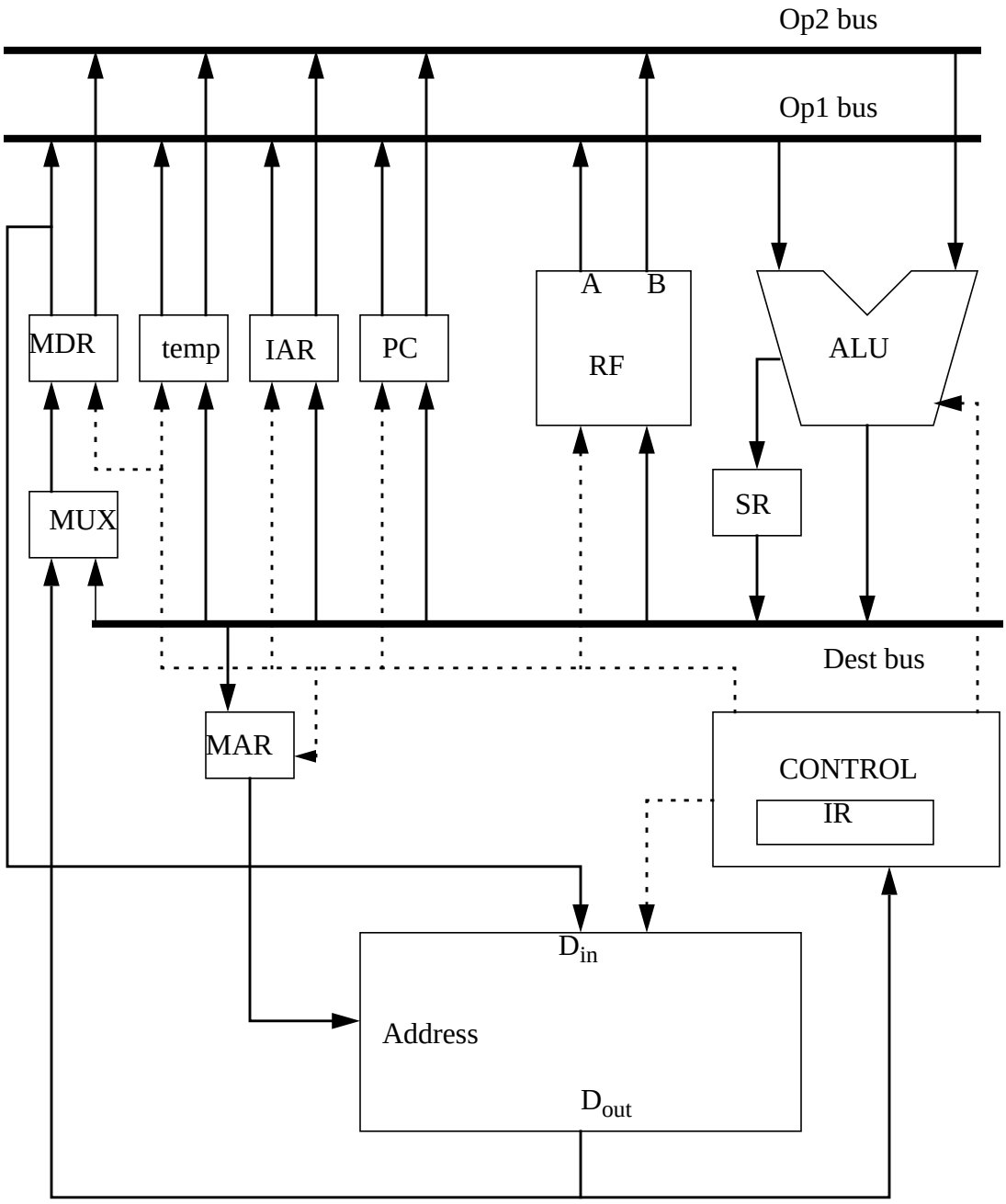


FIGURE 2.3 A typical CPU organization; it is represented in connection with the memory.

Exercises

2.1 Give some arguments for having the stack pointer as a special purpose register, and indicate which is the hardware required to manage it.

2.2 The block diagram in Figure 2.3 represents the Program Counter as a normal register, i.e. without dedicated hardware around it. How can it be incremented, and how long does it take to do so? In which case would PC require dedicated hardware?

3. Instruction set design

Clearly the design of a new machine is not a smooth process; the designer of the architecture must be aware of the possible hardware limitations when setting up the instruction set, while the hardware designers must be aware of the consequences their decisions have over the software.

It is not seldom that some architectural features cannot be implemented (at a reasonable price, in a reasonable time, using a reasonable surface of silicon, or cannot be implemented at all!); in these cases the architecture has to be redefined. Very often small changes in the instruction set can greatly simplify the hardware; the converse is also true, the process of designing the hardware often suggests improvements in the instruction set.

The topic of the following sections is the design of the instruction set: what should be included in the instruction set (what is a must for the machine), and what can be left as an option, how do instructions look like and what is the relation between hardware and the instruction set are some of the ideas to be discusses.

The coming section tries to answer the question: should we have a rich set of instructions (CISC) or a simple one (RISC)?

3.1 RISC / CISC, where is the difference?

For many years the memory in a computer was very expensive; it was only

after the introduction of semiconductor memories that prices began to fall dramatically. As long as the memory is expensive low end computers cannot afford to have a lot; there is a premium in a system with little memory to reduce the size of programs. This was the case in the 60s and 70s.

Lot of effort was invested in making the instructions smaller (tighter encoding) and in reducing the number of instructions in a program. It was observed that certain sequences of instructions occur frequently in programs like the following:

```
loop: ..................
      ..................
      DEC   r1    # decrement r1 (in r1 is the loop counter)
      BGEZ  loop  # branch if r1 >= 0
```

“Well let's introduce a new instruction that has to do the job of both” said the designers and this happened: the programmers was offered a new instruction which decrements a register and branches if the register is greater of equal to zero.

The same phenomenon happened in many other cases, like instruction that save/restore multiple registers in the case of procedure calls/returns, string manipulating instructions etc.

Another motivation that led to the development of complex instruction sets was the insufficient development of compilers; software was dominated by assembly language in so much that programmers found very pleasant and efficient to have powerful instructions at hand. In this trend many instructions became somehow like procedures, having parameters as the procedures may have, and saving memory in the same way procedures do.

The landmark for **CISC** architectures (Complex Instruction Set Computers) is the VAX family; introduced in 1977, the VAX architecture has more than 200 instructions, some 200 addressing modes and instructions with up to six operands. The instruction set is so powerful that a C program has almost the same number of assembly language instructions as the C source.

The main problem with CISC is that, due to different complexities, instructions require very different number of clock cycles to complete thus making very difficult efficient implementations (like pipelining); long running instructions render the interrupt handling difficult, while uneven sizes of instructions make the instruction decoding inefficient.

Due to the large number of instructions in the instruction set, the control part of a CISC machine is usually **microprogrammed**: the implication is a

lower clock rate (a higher T_{ck}) than the one a hardwired implementation would allow.

As the compiler technology developed, people realized how difficult is to figure out what is the best instruction (sequence of instructions) to be generated by the compiler: simply said there are too many combinations of instructions to do the same job, when using a complex instruction set.
To summarize, remember that the CPU performance is given by:

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

where CPU_{time} is the time spent by a CPU to run a program (the effective time), IC is the instruction count, CPI is the average number of clock cycles per instruction, and T_{ck} is the clock cycle (assumed to be constant).

Basically the CISC approach to reducing the CPU_{time} was to reduce the IC as much as possible. As a side effect the CPI has increased (instructions are complex and require many clock cycles to execute), as well as T_{ck} due to microprogramming. Very few new CISC have been introduced in the 80s and many argue that CISC survive only due to compatibility to older successful computers: a prominent example is the Intel's 80x86 family of microprocessors.

In contrast with CISC architectures, the **RISC** (Reduced Instruction Set Computers) ones make an attempt to reduce CPU_{time} by decreasing CPI and T_{ck}. Instructions are simple and only a few basic addressing modes are provided. Simple instructions mean low CPI as well as the possibility of easier hardwired implementation (thus lowering the clock cycle T_{ck}).

However the major point is that simple instructions allow pipelined implementations (all instruction execute in the same number of clock cycles) which dramatically decreases the CPI (the ideal CPI in a pipelined implementation is 1); moreover pipelining also permits higher clock rates.

The disadvantage is an increase in the number of instructions (IC); the same program may have as much as twice the number of instructions (assembly language) as compared with the same program that uses a CISC assembly language.

In a CISC machine the CPI can be in the order of 10, while for a RISC pipelined machine the CPI is some where between 1 and 2: roughly speaking there is an order of magnitude difference between the two approaches.

Example 3.1 RELATIVE PERFORMANCE:

The same program is compiled for a CISC machine (like a VAX) and for a pipelined RISC (like a SPARC based computer). The following data is available:

$IC_{CISC} = 500000$

$IC_{RISC} = 1100000$

$CPI_{CISC} = 6.8$

$CPI_{RISC} = 1.4$

$T_{ck\ CISC} = 25$ ns ($25*10^{-9}$ s)

$T_{ck\ RISC} = 30$ ns

What is the relative performance of the two machines?

Answer:

$$\frac{CPU_{timeCISC}}{CPU_{timeRISC}} = \frac{5 * 10^5 * 6.8 * 25}{11 * 10^5 * 1.4 * 30} = \frac{850}{462} = 1.83$$

The RISC machine is by 83% faster than the CISC one.

It is important to observe that we have only compared the CPU times: however this does not say everything about the two machines; the overall performance is affected by the memory and the I/O. For the user it is the elapsed time which counts not the CPU_{time}.

Practically, at this moment, all major semiconductor producers are offering at least one RISC architecture.

3.2 How many addresses?

After we have introduced, in the previous sections, some general ideas about the hardware-software interaction, we shall be discussing, in more detail, about the instruction set.

Consider the following statement in a high level language (C for instance):
a = a + b + a * c;

For all familiar with a programming language is clear what the meaning of this statement is: take the value of a and multiply it with c, then add a and c to the above result; the result is assigned to the variable a.

We know what are the rules of precedence and associativity; these rules are

also included in the high level languages (in the compiler more precisely). It is however obvious that we cannot expect the hardware to directly use these rules and to “understand” the whole sentence at once.

For sake of simplicity we require operations to be carried out in small steps; the result will be obtained after a **sequence of simple steps**. Incidentally this eliminates the need for the machine to know about the grouping rules.

Example 3.2 OPERATION SEQUENCE:

What is the sequence of operations (single statement per operation) that evaluates the statement:

```
a = a + b + a * c;
```

Answer:

```
t = a * c;
a = a + b;
a = a + t;
```

It is important to note that, in order to understand what the above sequence does, one has to know:

- how sequencing works: execute the first instruction, then the second and so on;
- what every operation does.

As an aside, the above sequence points out why our computers are said to be **sequential**.
In the example above t stands for temporary, an intermediate variable that holds the value of the first operation; in this case we could not assign the value of multiplication a * c to a because a is also needed for the second operation.

Which is the operation performed in the first statement of the answer in example 3.2? The spontaneous and yet not very correct answer is multiplication; it is not simple multiplication because the statement specify not only the **operation** to be performed but also **where** the result is to be stored (this is a computer statement, not a a simple mathematical equality). The proper name is therefore **multiply_and_store**, while for the second statement the proper name would be **add_and_store**.

Multiplication and addition are **binary** operations; multiply_and_store and add_and _store are **ternary** operations.

The operands which specify the values for the binary operation (which is a part of the ternary operation) are called **source operands**. The operand that specifies where the result is to be stored is called the **destination operand**.

Operands may be constants like in the following example:

a = b * 4

where we have to multiply the value designed by the operand b with 4 (the other operand) and to assign the result to the operand a. However, in most cases we deal with generic names: we don't know what the value is but where it is stored: we know its **address**. This is the reason we discuss about:

- 3-address machines;
- 2-address machines;
- 1-address machines;
- 0-address machines.

From now on, when we say we have an *n-address machine* we understand that *the maximum number of operands is n*.

Throughout this lecture we shall use the convention that the destination is the first operand in the instruction. This a commonly used convention though not generally accepted. It is consistent with the assignment statements in high level languages. The other used convention, listing the destination after the source operands, is coherent with our verbal description of operations.

3.2.1 Three-address machines

In a 3-address machine all three operands are explicit in each instruction. The general format of an instruction is:

```
operation dest, op1, op2
```

where:

- operation is the name of the operation to be performed;
- dest is the destination operand, the place where the result will be stored;
- op1 and op2 are the two source operands.

Thus the meaning of:

```
ADD r2, r1, r0
```

is to add the value stored in register r1, with the value stored in register r0, and put the result in the register r2.

Let’s see what addresses can specify and what are the implications for the hardware. In the example above all operands were held in registers and you may wonder why we discuss about addresses in this case: the reason is that registers can be seen as a special part of the memory, very fast and very close to the CPU; for historical reasons they have special names instead of being designated with some addresses.

Suppose you are designing an instruction set for a machine with 32 registers; then you need five bits to specify a register and, because the three operands may be different, 15 bits to specify the three registers that hold the three operands.

An instruction must also have a field to specify the operation to be performed, the opcode, probably a few bits depending of the number of instructions you want to have in the instruction set. We end up with an instruction that is 20 to 22 bits wide only for reasons of specifying the operands and the opcode. we shall see there are also other things to be included in a instruction, like an offset, a displacement, or an immediate value, with the possibility of having an even larger instruction size.

Should the machine be a 24 bit machine (24 is the first multiple of eight after 20-22) or not, and the answer is not necessarily:

- if we choose to have a 24 bit (or more) datapath then there is a definite advantage: the instruction is fetched at once, using a single memory access (assuming also that the data bus between the CPU and the memory has the same size as the datapath);

- if we settle for a cheap implementation, an 8 bit machine for instance, then the CPU must perform three memory accesses to memory only to get the whole instruction; then it can execute it. It is easy to see that in this case the performance both because number are processes in 8 bit chunks and because the instruction fetch takes so many clock cycles.

When all instructions (again we refer to the most common arithmetic/logic operations) specify only register operands then we say we have a **register-register** machine or a **load-store** machine; the load-store names comes from the fact that operands have to be loaded in registers from memory, and the result is stored in memory after the operation is performed.

In the following we'll take a different approach, we consider that all operands are in memory.

Example 3.3 UNDERSTANDING INSTRUCTIONS:

What is the meaning of the following instruction?

```
ADD   x, y, z
```

Answer:
Add the value of variable y (this is somewhere in the memory, we don't have to worry about, its address will be known after the translation is done), to the value of variable z (also in memory), and then store the result in the memory location corresponding the variable x.

Example 3.3 shows another way to process operands, when they all reside in the memory. If **all** instructions specify only memory operands then we say we have a **memory-memory** machine. We will next explore the implications for hardware of a memory-memory architecture.
In the case of a memory-memory machine the CPU knows it has to get two operands from memory before executing the operation, and has to store the result in memory. There are several ways to specify the address of an operand: this is the topic of addressing modes; for the following example we will assume a very simple way to specify the address of an operand: the absolute address of every operand is given in the instruction (this addressing mode is called *absolute*).

Example 3.4 MEMORY ACCESSES:

Addresses in an 8 bit machine are 16 bit wide. How many memory accesses are necessary to execute an instruction? Assume that the machine is memory-memory and operands are specified using absolute addresses.

Answer:
The instruction must specify three addresses, which means 3*2 = 6 bytes, plus an opcode (this is the first to be read by the CPU) which, for an 8 bit machine, will probably fit in one byte. Therefore the CPU has to read:

1 + 6 = 7 bytes

only to get the whole instruction. The source operands have to be read from memory and the result has to be stored into memory: this means 3 memory accesses. The instruction takes:

7 + 3 = 10

memory accesses to complete.

So far a register-register machine seems to be faster than a memory-memory machine, mainly because it takes less time to fetch the instruction

and there are fewer accesses to the memory. The picture however is not complete: to work within registers operands must be brought in from the memory, and write out to the memory, which means extra instructions. On the other hand intermediate results can be kept in registers thus sparing memory accesses. We'll resume this discussion after introducing the addressing modes.

3.2.2 Two-address machines

We start with an example.

Example 3.5 STATEMENT IMLPEMENTATION:

Implement the high level statement:

```
a = a + b + a * c
```

on a 3-address machine; assume that variables a, b, c are in registers r1, r2, and r3 respectively.

Answer:

```
MUL r4, r1, r3    # use r4 as a temporary
ADD r1, r1, r2    # a + b in r1
ADD r1, r1, r4    # final result in r1
```

Note that if the values of b or c are no longer necessary, one of the registers r2 or r3 could be used as a temporary.

In this example two out of three instructions have only **two distinct addresses**; one of the operands is both a source operand and the destination. This situation occurs rather frequently such that you may think it is worth defining a **2-address machine**, as one in which instructions have only two addresses.

The general format of instructions is:
`operation dest, op`
where:

- operation is the name of the operation to be performed
- dest designates the name of one source operand and the name of the destination
- op is the name of the second source operand

Thus the meaning of an instruction like:

```
ADD r1, r2
```

is to add the values stored in the registers r1 and r2, and to store the result

in r1.

There is an advantage in having two-address instructions as compared with three-address instructions, namely that instruction are shorter, which is important when preserving memory is at big price; moreover shorter instructions might be fetched faster (in the case instructions are wider than the datapath and multiple accesses to memory are required). There is a drawback however with two-address instructions: one of the source operands is destroyed; as a result extra moves are sometimes necessary to preserve the operands that will be needed later.

Example 3.6 STATEMENT IMLPEMENTATION:

Show how to implement the high level statement

```
a = a + b + a * c
```

on a 3-address machine and then on a 2-address machine. Both machines are 8 bit register-register machines with 32 general purpose registers and a 16 bit addresses. The values of variables a, b, and c are stored in r1, r2 and r3 respectively. In any case calculate the number of clock cycles necessary if every memory access takes two clock cycles and the execution phase of an instruction takes one clock cycle.

Answer:
For the 3-address machine:

```
MUL r4, r1, r3    # 3 * 2 + 1 clock cycles
ADD r1, r1, r2    # 3 * 2 + 1
ADD r1, r1, r4    # 3 * 2 + 1
```

This sequence requires 21 clock cycles to complete; each instruction has a fetch phase that takes three (3 bytes/instruction) times two clock cycles (2 clock cycles per memory access), plus an execution phase which is one clock cycle long.For the 2-address machine:

```
MOV r4, r1        # 2 * 2 + 1 clock cycles
MUL r4, r3        # 2 * 2 + 1
ADD r1, r2        # 2 * 2 + 1
ADD r1, r4        # 2 * 2 + 1
```

The sequence requires 20 clock cycles to complete; it is slightly faster than the implementation of the same statement on the 3-address machine.
The two address machine requires 10 bits (5 + 5) to encode the two operands and the example assumes an instruction is 16 bit wide.

The above example has introduced a new two operands instruction:

```
MOV dest, op
```

which transfers (moves) the value stored at address op to address dest.

Even if the MOV instruction is typical two operands instruction there is no reason to believe it only belongs to 2-address machines: a 3-address machine is one in which instructions specify up to 3 operands, not a machine in which all instructions have precisely 3 operands.

In this section we discussed about a register-register, 2-address machine. Nothing can stop us thinking about a 2-address, memory-memory machine, or even about a register-memory one.

3.2.3 One address machine (Accumulator machines)

In a 1-address machine the accumulator is implicitly both a source operand and the destination of the operation. The instruction has only to specify the second source operand. The format of an instruction is:

```
operation op
```

where:

- operation is the name of the operation to be performed
- op is a source or a destination operand. Example of source or destination operand is the accumulator (in the case of a store op denotes the destination).

Thus the meaning of:

```
ADD a
```

is to add the value of variable a to the content of the accumulator, and to leave the result in the accumulator.The accumulator is a register which has a special position in hardware and in software. Instructions are very simple and the hardware is also very simple. Figure 2.2 is an exemplification of an accumulator machine.

Example 3.7 STATEMENT IMPLEMENTATION:

Show how to implement the statement

```
a = a + b + a * c
```

using an accumulator machine.

Answer:

```
LOAD a      # bring the value of a in accumulator
MUL  c      # a * c
ADD  b      # a * c + b
ADD  a      # a * c + b + a
STO  a      # store the final result in memory
```

Due to its simplicity, only one operand has to be explicitly specified, accumulator machines present compact instruction sets. The problem is that the accumulator is the only temporary storage: memory traffic is the highest for accumulator machines compared with other approaches.

3.2.4 Zero-address machines (stack machines)

How is it possible to have a machine without explicit operands instructions? This is possible if we know where the operands are, and where is the result to be stored. In other words all operands are implicit.

A stack is a memory (sometimes called LIFO = Last In First Out) defined by two operations PUSH and POP: PUSH moves a new item from the memory into the stack (you don't have to care where), while POP gets the last item that was pushed into the stack. The formats of operations on a stack machine are:

```
operation
PUSH op
POP  op
```

where:

- operation indicates the name of the operation to be performed operation always acts on the value(s) at top of the stack
- op is the address in the main memory where the value to be pushed/popped is located.

A stack machine has two memories: an unstructured one, we call it the main memory, where instructions and data are stored, and a structured one,

the **stack** where access is allowed only through predefined operations (PUSH/POP). It is worth mentioning here that a stack has usually a very fast part where the top of the stack is located, and which is usually inside CPU, while the large body of the stack is outside the CPU (possibly extensible to disk) and, hence, slower. One may envision the fast part of the stack (inside CPU) as the registers of the machine; as a matter of fact, a stack machine does not have explicit registers.

Example 3.8 STATEMENT IMLPEMENTATION:

Show how to implement the statement

```
a = a + b + a * c
```

using a stack machine.

Answer:

```
PUSH a      # push the value of a;
PUSH c      # push the value of c
MUL         # multiply the two values on top of the stack
PUSH b
ADD
PUSH a
ADD
POP  a      # store the result back in memory at the address
              where a is located.
```

Whenever an operation is performed, the source operands are popped from the stack, the operation is performed, and the result is pushed into the stack.

Example 3.9 STATEMENT IMLPEMENTATION:

Show the content of the stack while implementing the statement:

```
a = a + b + a * c
```

ANSWER:

PUSH a	PUSH c	MUL	PUSH b	ADD	PUSH a	ADD	POP a
	c		b		a		
a	a	a*c	a*c	b+a*c	b+a*c	a+b+a*c	

The stack machine has the most compact encoded instructions possible.

To conclude this section let's see how fast is an expression evaluated on a stack machine as compared with other machines.

Example 3.10 CALCULATION OF CLOCK CYCLES:

Compute the number of clock cycles necessary to evaluate the statement:

```
a = a + b + a * c;
```

on an 8 bit stack machine, with 16 bit addresses. Assume that every memory access takes two clock cycles and that the execution phase of any instruction take only one clock cycle; assume also that addresses are absolute.

Answer:

```
PUSH a      # 2 + 2 * 2 + 2 clock cycles
PUSH c      # 2 + 2 * 2 + 2
MUL         # 2 + 1
PUSH b      # 2 + 2 * 2 + 2
ADD         # 2+ 1
PUSH a      # 2 + 2 * 2 + 2
ADD         # 2 + 1
POP  a      # 2 + 2 * 2 + 2
```

The above sequence of code completes in 49 clock cycles. Every PUSH or POP takes 2 clock cycles to read the opcode (one byte), plus 2 * 2 clock cycles to read the address (2 bytes) of the operand, plus 2 clock cycles to read/store the operand.

It would be unfair to directly compare the performance of the stack machine in example 3.10 with the performance of machine in the previous examples as long as the stack machine has to bring the operands from memory while in the other cases it was assumed that the operands are in registers. The stack has the disadvantage that, by definition, it cannot be randomly accessed thus making difficult to generate *efficient* code; note however that the easiest way to generate code in a compiler is for a stack machine.

3.3 Register or memory?

In classifying architectures we used the number of operands the most common arithmetic instructions specify. As we saw the 3-address and 2-address machines may have operands in registers, memory or both. A

question may be naturally asked: which is the best way to design the instruction set? This is to say: should all instructions be register-register or maybe should they all be memory-memory; what happens if we mix registers and memory in specifying operands?

Example 3.11 CLOCK CYCLES AND MEMORY TRAFFIC:

We have two 32 bit machines, a register-register and a memory-memory one. Addresses are 32 bit wide and the register-register machine has 32 general purpose registers. A memory access takes two clock cycles and an arithmetic operation executes in two clock cycles (the execution phase). Addresses of operands are absolute. Show how to implement the statement:
a = b * c + a * d;
and compare the number of clock cycles, and the memory traffic for the two machines. Variables a, b, c, d reside in memory and no one may be destroyed but a. Assume also that instructions are 32 bit wide or multiples of 32 bit.

Answer:

		Words per instruction	Memory accesses	Clock cycles
LOAD	r1, b	2	3	6
LOAD	r2, c	2	3	6
MUL	r3, r1, r2	1	1	3
LOAD	r1, a	2	3	6
LOAD	r2, d	2	3	6
MUL	r1, r1, r2	1	1	3
ADD	r1, r1, r3	1	1	3
STORE	a, r1	2	3	6
Total		13	18	39

For the memory-memory machine:

		Words per instruction	Memory accesses	Clock cycles
MUL	temp, b, c	4	4+3	15
MUL	a, a, d	4	4+3	15
ADD	a, a, temp	4	4+3	15
Total		12	21	45

For the memory-memory machine every instruction require 7 memory accesses: one to get the opcode, plus 3 to get the three addresses of the operands, plus other 3 to access the operands. The code for the memory-memory machine is shorter than the code for the register-register machine, 12 words as compared with 13, but the memory traffic is higher, as well as

the execution time.

Even though in the above example we considered two different machines, it is quite common to have hardware that supports both models. This is the case with very successful machines like IBM-360 or DEC VAX; moreover these machines also support the register-memory model.

The most valuable advantage of registers is their use in computing expression values and in storing variables. When variables are allocated to registers, the memory traffic is lower, the speed is higher because registers are faster than memory and code length decreases (since a register can be named with fewer bits than a memory location). If the number of registers is sufficient, then local variables will be loaded into registers when the program enters a new scope; the available registers will be used as temporaries, i.e. they will be used for expression evaluation.

How many registers should a machine have? If their number is too small then the compiler will reserve all for expression evaluation and variables will be kept in memory thus decreasing the effectiveness of a register-register machine. A too large number of registers, on the other hand, may mean wasted resources that could be used otherwise for other purposes.

Most microprocessors from the first generation (8 bit like Intel 8080, or Motorola 6800) had 8 general purpose registers. A very successful 16 bit microprocessor, Intel 8086, had 14 registers though most of them were reserved for special purposes. Almost all 32 bit microprocessors on the market today have 32 integer registers, and many have special registers for floating point operations.

32 registers seem to be sufficient even for a 64 bit architecture; at least this is the point of view of the team who designed the ALPHA chip.

Let's now summarize some of the basic concepts we have introduced so far. The basic criteria of differentiating architectures was the number of operands an instruction can have. Operands may be named explicitly or implicitly:

- **stack architectures**: all operands are implicitly on the stack
- **accumulator architectures**: one source operand and the destination are implicitly the accumulator; the second source operand has to be explicitly named
- **general purpose register (GPR) architectures** have only

explicit operands, either registers or memory locations. GPR architectures dominate the market at this moment, basically for two reasons: first registers are faster than memory and second, compilers use more efficient registers than other forms of temporary storage like accumulators or stacks. The main characteristics that divide GPRs are:

- th**e number of operands** a typical arithmetic operation has we discussed about 2-address machines (one of the operands is both source and destination), and about three address-machines;

- number of operands that may be **memory addresses** in ALU operation. This number may vary from none to three.

Using the two parameters we have to differentiate GPR architectures, there are seven possible combinations in the table below:

Number of operands	Number of memory addresses	Examples
2	0	IBM
	1	PDP 10, Motorola 68000, IBM-360
	2	PDP 11, IBM-360, National 32x32, VAX
3	0	SPARC, MIPS, HP-PA, Intel 860, Motorola 88000
	1	IBM-360
	2	-
	3	VAX

Please observe that IBM-360 appears in several positions in the table, due to the fact that the architecture supports multiple formats. The same is true for the VAX architecture.

Three combinations classify most of the existing machines:

- **register-register** machines which are also called load-store. They are defined by 2-0 or 3-0 (operands-number of memory addresses); in other words all operands are located in registers

- **register-memory** machines: defined by 2-1 (operands-number of memory addresses); one of the operands may be a memory address;

- **memory-memory** machines: 2-2 or 3-3; all operands are memory addresses.

Here are some of the advantages and disadvantages of the above alternatives:

register-register machines

Advantages:
- simple instruction encoding (fixed length)
- simple compiler code generation
- instruction may execute in the same number of clock cycles (for a carefully designed instruction set)
- lower memory traffic as compared with other architectures.

Disadvantages:
- higher instruction count than in architectures with memory operands
- the fixed format wastes space in the case of simple instructions that do not specify any register (NOP, RETURN etc.)

register-memory machines

Advantages:
- more compact instructions than in register-register machines
- in many cases loads are spared (when an operand is in register and the other one is in memory)

Disadvantages:
- operands are not equivalent: in an arithmetic or logic operation one of the source operands is destroyed (this complicates the code generation in the compiler).

memory-memory machines

Advantages:
- compact code
- easy code generation

Disadvantages:
- different length instructions
- different running times for instructions
- high memory traffic.

3.4 Problems in instruction set design

The instruction set is the collection of operations that define how data is transformed/moved in the machine. An architecture has an unique set of operations and addressing modes that form the **instruction set**.

The main problems the designer must solve in setting up an instruction set for an architecture are:

- which are the operations the instruction set will implement;
- relationship between operations and addressing modes;
- relationship between operations and data representation.

3.4.1 Operations in the instruction set

We obviously need **arithmetic and logic operations** as the purpose of most applications is to perform mathematical computation. We also need **data transfer** operations as data has to be moved from memory to CPU, from CPU to memory, between registers or between memory locations.

The instruction set must provide some instructions for the **control flow** of programs; we need, in other words, instructions that allow us to construct loops or to skip sections of code if some condition happens, to call subroutines and to return from them.

Arithmetic and logic instructions

Integer operations:

- add and subtract are a must and they are provided by all instruction sets. At this point it must be very clear how integers are represented in the machine: unsigned or signed and, in the latter case what signed representation is used? The most frequent used representation for signed integers is the two's complement. Some architectures provide separate instructions for unsigned integers and for signed integers.
- multiply and divide: the early machines had no hardware support for these operations. Almost all new integrated CPUs, introduced since the early 80s provide hardware for multiplication and division. Again the integer representation must be very clear because it affects the hardware design. Integer multiplication and division are complicated by the results they generate;

multiplication of two n bit integers may result in a 2*n bit number. If all registers are n bit wide, then some provisions must be made in hardware to accommodate the result.

- compare instructions: compare two integers and set either a condition code (a bit in a special register called the Status Register), or a register (in the new architectures); EQ, NE, GT, LT, GE, LE may be considered for inclusion in the instruction set. Set operations are used in conjunction with branches, so they must be considered together at the design time.

- others: like the remainder of a division, or increment/decrement register, add with carry, subtract with borrow etc.; most of these instructions can be found in the instruction sets of earlier architectures (8 and 16 bit).

Floating point operations:

While the IEEE 754 standard is currently adopted by all new designs, there are other formats still in use, like those in the IBM 360/370 or in the DEC's machines.

- add, subtract, multiply, divide: are provided sometimes in an additional instruction set; most of the new architectures have special registers to hold floating point numbers.

- compare instructions: compare two floating point numbers and set either a condition code or a register: EQ, NE, GT, LT, GE, LE may be considered.

- convert instructions: convert numbers from integer to float and vice-versa, conversion between single precision and double precision representations might be considered.

Logical instructions

It is not necessary that all possible logical operations are provided, but care must be taken that the instruction set provides the minimum necessary such that the operations that are not implemented can be simulated.

Decimal operations

Decimal add, subtraction, multiply, divide are useful for machines running business applications (usually written in COBOL); they are sometimes primitives like in IBM-360 or VAX, but in most cases they are simulated

using other instructions.

String operations

String move, string compare, string search, may be primitive operations as in IBM-360 or VAX, or can be simulated using simpler instructions.

Shifts

Shift left, right, arithmetic/logic are useful in manipulating bit strings, in implementing multiplication /division when not provided by hardware; all new architectures provide shifts in the instruction set.

- arithmetic shift: whenever the binary configuration in a register is shifted towards the least significant bit (shift right) the most significant positions are filled with the value of the sign bit (the most significant bit in register before shifting);

- logical shift: positions emptied by shift are filled with zeros.

Example 3.12 LOGICAL AND ARITHMETIC SHIFTS:

The binary configuration in a register is:
11010101
Show the register's configuration after the register is shifted one and two positions, logical and arithmetic.

Answer:

1 1 0 1 0 1 0 1	The initial configuration
1 1 1 0 1 0 1 0	arithmetic right one bit
1 1 1 1 0 1 0 1	arithmetic right two bits
0 1 1 0 1 0 1 0	logical right one bit
0 0 1 1 0 1 0 1	logical right two bits
1 0 1 0 1 0 1 **0**	left one bit
0 1 0 1 0 1 **0 0**	left two bits

Note that it does not make sense to discuss about left arithmetic shifts.

Data transfer instructions

As long as the CPUs were 8 bit wide the problem of transfers was simple: loads and stores had to move *byte*s from memory to CPU, from CPU to memory or between registers. With a 32 bit architecture we must consider the following transfer possibilities:

- **byte**s (8 bits);
- **half-word**s (16 bits);
- **word**s (32 bits);
- **double word**s (64 bits).

While allowing all transfers provides maximum flexibility, it also poses some hardware complications that may slow down the clock rate (this will be discussed again in chapter 4).

If transfers of items narrower than a word (for the today 32 bit architectures) are included in the instruction set, then two possibilities are to be considered for **loads:**

- **unsigned:** data is loaded in a register at the least significant positions while the most significant positions are filled with zeros;
- **signed:** data is loaded into a register at the least significant positions while the most significant positions in the register are filed with the sign of the data being loaded (i.e. the MSB of data).

Example 3.13 EFFECT OF INSTRUCTIONS ON REGISTERS:

Show the content of the register r1 after executing the instructions:

```
LB    r1,   0xa5        # load bite (signed)
and
LBU   r1,   0xa5        # load byte unsigned
```

All registers are 16 bit wide.

Answer:

MSB LSB

1111 1111 1010 0101 after executing `LB r1, 0xa5`

0000 0000 1010 0101 after executing `LBU r1, 0xa5`

Moves

Allow for transfer of data between registers; if the architecture provides floating point registers, then the instruction set should include also the possibility to transfer between the two sets of registers (integers & floats).

Control flow instructions

In the followings we shall use the names:

- **branch:** when the change in control is conditional
- **jump:** when the change in control is unconditional (like a goto).

The four categories of control flow instructions are:

- **branches**
- **jumps**
- **subroutine calls**
- **subroutine returns**

The destination address of a branch, jump or call can always be specified in the instruction; this can be done because the *target* (i.e. the instruction where control must be transferred) is known at compile time. In the case of a return the target is not known at the compile time; a subroutine can be called from different points in a program.

Specifying the destination

There are two basic ways to specify the target:

- use the **absolute address** of the target. This has two main drawbacks, first it requires the full address of the target to be provided in the instruction which is impossible for fixed length instructions, and second, programs using absolute addresses are difficult to relocate.

- use **PC-relative** computation of the target. This is the most used way to specify the destination in control-flow; a displacement is added to the program counter (PC) to get the target address. The displacement is usually represented as a two's complement integer, thus allowing to specify targets that are after the current instruction (positive displacement), or before the present instruction (negative displacement). The main advantages in using the PC-relative way to compute the target address are:

    - **fewer bits** necessary to specify the target. Targets are

usually not very far away from the present instruction, so that it is not worth specifying the target as an absolute address;

- **position independence:** because addressing is PC-relative, the code can run independent of where it is loaded in memory. The linking and loading of programs is thus simplified.

What happens with returns

For returns the target address is not known at the compile time; the target must be somehow specified dynamically so that it can change at run time:

- use the **run time stack** to hold the return address: the address of the next address to be executed after a call instruction (at the return from the subroutine) is pushed into the run time stack when the subroutine is called; at the end of the subroutine the address that was saved in the stack is popped into the PC;

- use a **register** that contains the target address (of course the register must be wide enough to can accommodate an address); most of the new architectures use this mechanism for the returns; a register from the set of general purpose registers is dedicated to this purpose.

A major concern in the design of an instruction set is the size of the displacement; if the size is too small then we must very often resort to absolute addresses which mean larger instructions. Too much space for displacement may be detrimental for other useful functions.

Exercises

3.1 What is the value of variables x and y after the following operations are performed:

```
x = 3 - 2 - 1;
y = 12/6/3;
```

If you are unsure about the result, then write a small program in the language you like, and see which are the rules for the associativity.

3.2 Addresses in a 32 bit machine are 32 bit wide. How many memory accesses are necessary to execute an instruction? Assume that the machine is memory-memory and operands are specified using absolute addresses.

3.3 What was the need for the early microprocessors (8 bit) to provide instructions like add with carry or subtract with borrow? Modern 32 bit CPUs do not provide such instructions.

Unit 8 : Microprocessor Architecture

Lesson 1 : Microcomputer Structure

1.1. Learning Objectives

On completion of this lesson you will be able to :

- draw the block diagram of a simple computer
- understand the function of different units of a microcomputer
- learn the basic operation of microcomputer bus system.

1.2. Digital Computer

A digital computer is a multipurpose, programmable machine.

A digital computer is a multipurpose, programmable machine that reads binary instructions from its memory, accepts binary data as input and processes data according to those instructions, and provides results as output.

1.3. Basic Computer System Organization

Basic computer system organization.

Every computer contains five essential parts or units. They are

i. the arithmetic logic unit (ALU)
ii. the control unit
iii. the memory unit
iv. the input unit
v. the output unit.

1.3.1. The Arithmetic and Logic Unit (ALU)

The arithmetic and logic unit (ALU) is that part of the computer that actually performs arithmetic and logical operations on data.

The arithmetic and logic unit (ALU) is that part of the computer that actually performs arithmetic and logical operations on data. All other elements of the computer system - control unit, register, memory, I/O - are there mainly to bring data into the ALU to process and then to take the results back out.

An arithmetic and logic unit and, indeed, all electronic components in the computer are based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations. Data are presented to the ALU in registers. These registers are temporary storage locations within the CPU that are connected by signal paths of the ALU.

The ALU is the area of the computer in which arithmetic and logic operations are performed on data. The type of operation that is to be performed is determined by signals from the control unit. The data that are to be operated on by the ALU can come from either the memory unit or the input unit. Results of operations performed in the ALU can be transferred to either the memory unit for storage or the output unit.

1.3.2. Control Unit

Control unit is like the conductor of an orchestra, who is responsible for keeping each of the orchestra members in proper synchronization. This unit contains logic and timing circuits that generate the proper signals necessary to execute each instruction in a program.

The control unit fetches an instruction from memory by sending an address and read command.

The control unit fetches an instruction from memory by sending an address and read command. The instruction word stored at the memory location is then transferred to the control unit. This instruction word, which is in some form of binary code, is then decoded by logic circuitry in the control unit to determine which instruction is being called for. The control unit uses this information to send the proper signals to the rest of the units in order to execute the specified operation.

This sequence of fetching an instruction code and then executing the indicated operation is repeated over and over by the control unit.

1.3.3. Memory Unit

The memory section usually consists of a mixture of RAM and ROM.

The memory section usually consists of a mixture of RAM and ROM. It may also have magnetic floppy disks, magnetic hard disk or laser optical disks, magnetic hard disk or laser optical disks. Memory has two purposes. The first purpose is to store the instructions (Program) that the computer is to perform. The second purpose of the memory is to store the data that are to be operated on by the program. The memory stores these instruction and data as groups of binary digits (words).

Bus

Bus

Two different-size arrows are used; the larger arrows represent data or information that actually consists of a relatively large number of parallel lines, and the smaller arrows represent control signals that are normally only one or a few lines. The various arrows are also numbered to allow easy reference to them in the following descriptions. These components are common communication path called a bus.

Address Bus

Address bus is a unidirectional bus.

This is a unidirectional bus, because information flows over it in only one direction, from the CPU to the memory or I/O elements. The CPU alone can place logic levels on the lines of the address bus, thereby generating 2^{16} = 65,536 different possible addresses. Each of these addresses corresponds to one memory location or one I/O element.

When the CPU wants to communicate (read or write) with a certain memory location or I/O device, it places the appropriate 16-bit address code on its 16 address pin outputs, A_0 through A_{15}, and onto the address bus. These address bits are then decoded to select the desired memory location or I/O device.

Data Bus

Data bus is a bi-directional bus.

This is a bi-directional bus, because data can flow to or from the CPU. The CPU's eight data pins, D_0 through D_7,can be either inputs or outputs, depending on whether the CPU is performing a read or a write operation. During data bus by the memory or I/O element. During a write operation the CPU's data pins act as outputs and place data on the data bus, which are then sent to the selected memory or I/O element. in all cases, the transmitted data words are 8bits wide because the CPU handles 8-bit data words, making this an 8-bit μC.

Control Bus

Control Bus

This is the set of signals that is used to synchronize the activities of the separate μC elements. Some of these control signals, such as RD and WR are sent by the CPU to the other elements to tell them what type of operation is currently in progress. The I/O elements can send control signals to the CPU. An example is the rest input (RES) of the CPU which, when driven LOW, causes the CPU to reset to a particular starting stare.

Memory Unit

Operation of the memory is controlled by the control unit which signals for either a read or a write operation. A given location in memory is accessed by the control unit that provides the appropriate address code. Information can be written into the memory from the ALU or the input unit, again under control of the control unit. Information can be read from memory into the ALU or into the output unit.

1.3.4. Input Unit

The input unit consists of all of the devices used to take information and data that are external to the computer and put them into the memory unit or the ALU. The control unit determines where the input information is sent. The input unit is used to enter the program and data into the memory unit or into the ALU from an external device during the execution of a program. Some of the common input devices are keyboards, Joysticks, mouse, OCR, OMR etc.

1.3.5. Output Unit

Input Unit
Output Unit
Interfacing

The output unit consists of the devices used to transfer data and information from the computer to the "outside world." The output devices are directed by the control unit and can receive data from memory or the ALU. Examples of common output devices are printers, disk or tape units, video monitors, and digital-to-analog converters (DACs).

1.3.6. Interfacing

The devices that make up the input and output units are called peripherals because they are external to the rest of the computer. The most important aspect of peripherals involves interfacing. Computer interfacing is specifically defined as transmitting digital information between a computer and its peripherals in a compatible and synchronized way.

Many input/output devices are not directly compatible with the computer because of differences in such characteristics as operating speed, data format (e.g., BCD, ASCII, binary), data transmission mode (e.g., serial, parallel), and logic signal level. Such I/O devices require special interface circuits which allow them to communicate with the control, memory, and ALU portions of the computer system. A common example is the video display terminal (VDT), which can operate both as an input and an output device. The VDT transmits and receives data serially (one bit at a time) while most computers handle data in parallel form. Thus, a VDT requires interface circuitry in order to send data to or receive data from a computer.

Central Processing Unit (CPU)

Central Processing Unit.

In the ALU and control units are combined into one unit called the central processing unit (CPU). This is commonly done to separate the actual "brains" of the computer from the other units. In a microcomputer the CPU is usually implemented on a single chip silicon wafer.

1.4. Exercise

1.4.1. Multiple choice questions

a) VDT stands for

i) Video processing unit
ii) Video display terminal
iii) Input output terminal
iv) all of the above.

b) A digital computer is

i) a programmable machine
ii) a digital machine
iii) a digital calculator
iv) none of the above.

1.4.2. Questions for short answers

a) What is the CPU?
b) What is meant by interfacing in a computer system?
c) What basic operations occur repeated by in computer?

1.4.3. Analytical questions

a) Name the five basic units of a computer and describe the major functions of each.
b) Describe about the computer interfacing.

Lesson 2 : Microprocessor Architecture

2.1. Learning Objectives

On completion of this lesson you will be able to :

- define the address bus, the data bus, and the control bus, and explain their functions in reference to the 8085/8080A microprocessor.
- list the registers in the 8085/8080A microprocessor, and explain their functions.

The microprocessor is usually a single IC that contains all of the circuitry of the control and arithmetic-logic units-in other words, the CPU.

It is important to understand the difference between the microcomputer (μC) and the microprocessor (μP). A microcomputer contains several elements, the most important of which is the microprocessor. The microprocessor is usually a single IC that contains all of the circuitry of the control and arithmetic-logic units-in other words, the CPU. It is common to refer to the microprocessor as the MPU (microprocessor unit), since it is the CPU (central processing unit) of the microcomputer.

The microprocessor unit of the computer consists of various registers to store data, the arithmetic logic unit (ALU) to perform arithmetic and logical operations, instruction decoders, counters, and control lines. The CPU reads instructions from the memory and performs the tasks specified. The CPU communicates with input/ output devices to accept or to send data. The input and output devices are known also as peripherals. The CPU is the primary and central player in communicating with various devices such as memory, input, and output; however, the timing of the communication process is controlled by the group of circuits called the control unit.

2.2. Microprocessor

A microcomputer is a computer similar to any other computer, except that the CPU functions of the microcomputer are performed by the microprocessor.

The microprocessor is one component of the microcomputer. A microcomputer is a computer similar to any other computer, except that the CPU functions of the microcomputer are performed by the microprocessor. Similarly, the term peripheral is used for input/ output devices; however, occasionally memory is also included in this term. The various components of the microcomputer and their function are described in the following paragraphs.

The MPU (microprocessor unit) is the heart of every microcomputer. It performs a number of functions, including.

The MPU microprocessor unit.

1. Providing timing and control signals for all elements of the μC
2. Fetching instructions and data from memory
3. Transferring data to and from memory and I/O devices

4. Decoding instructions
5. Performing arithmetic and logic operations called for by instructions
6. Responding to I/O generated control signals such as RESET and INTERRUPT.

The MPU contains all of the logic circuitry for performing these functions, but its internal logic is generally not externally accessible. Instead, we can control what happens inside the MPU by the program of instructions that we put in memory for the MPU to execute. This is what makes the MPU so versatile and flexible - when we want to change its operation, we simply change the programs stored in RAM (software) or ROM (firmware) rather than rewire the electronics (hardware).

Very-large-scale integration (VLSI)

The microprocessor is a semiconductor device consisting of electronic logic circuits manufactured by using either a large-scale (LSI) or very-large-scale integration (VLSI) technique. The microprocessor is capable of performing computing functions and making decisions to change the sequence of program execution. In large computers, the CPU performs these computing functions and it is implemented on one or more circuit boards. The microprocessor is in may ways similar to the CPU; however, the microprocessor includes all the logic circuitry (including the control unit) on one chip. For clarity, the microprocessor can be divided into three segments, arithmetic/logic unit (ALU), register unit, and control unit.

- ***Arithmetic and Logic Unit :*** In this area of the microprocessor, computing functions are performed on data. The CPU performs arithmetic operations such as addition and subtraction, and logic operations such as AND, OR, and exclusive OR. Results are stored either in register or in memory or sent to output devices.
- ***Register Unit :*** This area of the microprocessor consists of various registers. The register are used primarily to store data temporarily during the executing of a program. Some of the registers are accessible to the user through instructions.
- ***Control Unit :*** The control unit provides the necessary timing and control signals to all the operations in the microcomputer. It controls the flow of data between the microprocessor and peripherals (including memory).

Input

The input section transfers data and instructions in binary from the outside world to the microprocessor.

The input section transfers data and instructions in binary from the outside world to the microprocessor. It includes devices such as keyboards, teletypes, and analog-to-digital converters. Typically, a microcomputer includes a keyboard as an input device. The key board has sixteen data keys (o to 9 and A to F) and some additional function keys to perform operations such as storing data and executing programs.

Output

The output section transfers data from the microprocessor to output devices Output.

The output section transfers data from the microprocessor to output devices such as light emitting diodes (LEDs), cathode-ray-tubes (CRTs), printers, magnetic tape, or another computer. Typically, single-board computers include LEDs and seven-segment LEDs as output devices.

Memory

Memory stores binary information such as instructions and data, and provides that information to the microprocessor whenever necessary.

Memory stores binary information such as instructions and data, and provides that information to the microprocessor whenever necessary. To execute programs, the microprocessor reads instructions and data from memory and performs the computing operations in its ALU section. Result are either transferred to the output section for display or stored in memory for later use. The memory block has two sections : Read - Only Memory (ROM) and Read / Write Memory (R/WM), popularly known as Random Access Memory (RAM).

The ROM is used to store programs that do not need alterations. The monitor program of a single - board microcomputer is generally stored in the ROM. Program stored in the ROM can only be read; they cannot be altered.

The Read / Write memory (R/WM) is also known as user memory. It is used to store user programs and data. The information stored in this memory can be read and altered easily.

System Bus

The system bus is a communication path between the microprocessor and the peripherals.

The system bus is a communication path between the microprocessor and the peripherals; it is nothing but a group of wires that carries bits. The microcomputer bus is in many ways similar to a one-track, express subway, the microcomputer bus carries bits between the microprocessor and only one peripheral at a time. The same bus is time - shared to communicate with various peripherals, with the timing provided by the control section of the microprocessor.

The Bus System

The μC has three buses which carry all the information and signals involved in the system operation.

The μC has three buses which carry all the information and signals involved in the system operation. These buses connect the microprocessor (CPU) to each of the memory and I/O elements so that and information can flow between the CPU and any of these other elements. The buses involved in all the data transfers have functions that are described as follows.

2.3. How does the Microcomputer Work?

At first program and data are entered in the R/W memory. The program includes binary instructions to add given data and to display the answer at the monitor. When the microcomputer is given a command to execute the program, it reads and executes one instruction at a time and finally sends the result to the display.

This process of program execution can best be described by comparing it to the process of assembling a radio kit. The instructions for assembling the radio are printed on a sheet of paper in sequence. One reads the firs instruction, then picks up the necessary components of the radio and performs the task. The sequence the instructions are stored sequentially in the memory. The microprocessor fetches the first instruction form its memory sheet, decodes it, and executes that instruction. The sequence of fetch, decode, and execute is continued until the microprocessor comes across the instruction, Stop. During the entire process, the microprocessor uses the system bus to fetch the binary instructions and data from the memory. It uses registers from the register section to store data temporarily, and it performs the computing function in the ALU section. Finally it sends out the result in binary, using the same bus lines, to the monitor.

The microprocessor fetches the first instruction from its memory sheet, decodes it, and executes that instruction.

This lesson describes the internal architecture and various operations of the microprocessor in the context of the 8085/8080A. It also expands on topics such as memory and I/O .

2.4. Microprocessor Architecture and its Operations

The microprocessor is a programmable logic device, designed with registers, flip-flops. The microprocessor has a set of instructions designed internally, to manipulate data and communicate with peripherals. This process of data manipulation and communication is determined by the logic design of the microprocessor, called the architecture.

The microprocessor is a programmable logic device, designed with registers, flip-flops.

The microprocessor can be programmed to perform functions on given data by selecting necessary instructions from its set. These instructions are given to the microprocessor by writing them into its memory. Writing (Or entering) instruction and data is done through an input device such as a keyboard. The microprocessor reads or transfers each instruction one at a time, matches it with its instruction set, and performs the data manipulation indicated by the instruction. The result can be stored in memory or scent to such output devices as LEDs or a CRT terminal. In addition, the microprocessor can respond to external signals. Its can be interrupted, reset, or asked to wait to synchronize with

All the various functions performed by the microprocessor can be classified in three general categories.

slower peripherals. All the various functions performed by the microprocessor can be classified in three general categories :

- Microprocessor-initiated operations
- Internal data operations
- Peripheral (or externally) initiated operations.

To perform these functions, the microprocessor requires a group of logic circuits and a set of signals called control signals. However, early processors do not have the necessary circuitry on one chip; the complete units are made up more than one chip. Therefore, the term Micro Processing Unit (MPU) is defined here as a group of devices that can perform these functions with the necessary set of control signals. This term is similar to the term Central Processing Unit (CPU). However, later microprocessors include most of the necessary circuitry to perform these operations on a single chip. Therefore, the terms MPU and microprocessor often are used synonymously.

The microprocessor functions are explained here in relation to the 8085 or 8080A MPU.

2.5. Microprocessor-Initiated Operations and 8085/8080A Bus Organization

The MPU performs primarily four operations.

The MPU performs primarily four operations.

1. Memory Read: Reads data from memory.
2. Memory Write: Writes data into memory.
3. I/O read: Accepts data from input devices.
4. I/O Write: Sends data to output devices.

All these operations are part of the communication process between the MPU and peripheral devices (including memory). To communicate with a peripheral (or a memory location), the MPU needs to perform the following steps:

Step 1 : Identify the peripheral or the memory location (with its address).
Step 2 : Transfer data
Step 3 : Provide timing or synchronization signals.

The 8085/8080A MPU performs these functions using three sets of communication lines called buses : The address bus, the data bus, and the control bus.

2.5.1. Internal Data Operations and the 8085/8080A Registers

Internal data operations and the 8085/8080A registers.

The internal architecture of the 8085/8080A microprocessor determines how and what operations can be performed with the data. These operations are

1. Store 8-bits data.
2. Perform arithmetic and logical operations.
3. Test for conditions.
4. Sequence the execution of instructions.
5. Store data temporarily during execution in the defined R/W memory locations called the stack.

To perform these operations, the microprocessor requires registers, an arithmetic logic unit (ALU) and control logic, and internal buses (path for information flow).

Registers

Registers

The 8085/8080A has six general - purpose registers to perform the first operation listed above, that is, to store 8-bit data during a program execution. These registers are identified as B, C, E, H, and L. They can be combined as register pairs - BC, DE, and HL - to perform some 16-bit operation.

These registers are programmable, meaning that a programmer can use them to load or transfer data from the registers by using instructions. For example, the instruction MOV B, C transfers the data from register C to register B. Conceptually, the registers can be viewed as memory locations, except they are built inside the microprocessor and identified by specific names. Some microprocessors do not have these types of registers; instead, they use memory space as their registers.

Accumulator

Accumulator
Flags

The accumulator is an 8-bit register that is part of the arithmetic logic unit (ALU). This register is used to store 8-bit data and to perform arithmetic and logical operations. The result of an operation is stored in the accumulator. The accumulator is also identified as register A.

Flags

The ALU includes five flip-flops that are set or reset according to data conditions in the accumulator and other registers. The microprocessor uses them to perform the third operation; namely testing for data conditions.

For example, after an addition of two numbers, if the sum in the accumulator is larger than eight bits, the flip-flop that is used to indicate a carry, carry the Carry flag (CY), is set to one. When an arithmetic operation results in zero, the flip-flop called the Zero flag (Z) is set to one. The 8085/8080A has five flags to indicate five different types of data conditions. They are called Zero (Z), Carry (CY), Sign (S), Parity (P), and Auxiliary Carry (AC) flags. The most commonly used flags are Zero and Carry; the others will be explained as necessary.

These flags have critical importance in the decision-making process of the microprocessor. The conditions (set or reset) of the flags are tested through software instructions. For example, the instruction JC (Jump On Carry) is implemented to change the sequence of a program when the CY flag is set. The implemented to flags cannot be emphasized enough; they will be discussed again in applications of conditional jump instructions.

Program Counter (PC)

Program Counter

This 160bit register deals with the fourth operation, sequencing the execution of instructions. This register is a memory pointer. Memory locations have 16-bit address, and that is why this is a 16-bit register.

The microprocessor uses this register to sequence the execution of instructions. The function of the program counter is to point to the memory address from which the next byte is to be fetched. When a byte (machine code) is being fetched the program counter is incremented by one to point to the next memory location.

Stack Pointer (SP)

Stack Pointer

The stack pointer is also a 16-bit register used as a memory pointer; initially, it will be called the stack pointer register to emphasize that it is a register. It pointers of a memory location in R/W memory, called the stack. The beginning of the stack is defined by loading a 16-bit address in the stack pointer (register).*

2.6. Exercise

2.6.1. Multiple choice questions

a) Which one of the following is not the function of memory unit?

i) Stores data awaiting to process
ii) To store instruction
iii) Perform arithmetic operation
iv) Stores processed data.

b) The central processing unit which is the ‘brains’ of the computer is the combination of

i) the ALU and control unit
ii) the input unit and ALU
iii) the control unit and bus structure
iv) the ALU and bus structure.

c) A CPU with 16 address lines can address

i) 16,384 memory locations
ii) 32,768 memory locations
iii) 64,000 memory locations
iv) 65,536 memory locations.

2.6.2. Questions for short answers

a) What is meant by interfacing in a computer system?
b) What is the function of central processing unit?

2.6.3. Analytical question

a) Draw the typical structure of a micro computer and show its bus organization

Unit 9 : Fundamentals of Parallel Processing

Lesson 1 : Types of Parallel Processing

1.1. Learning Objectives

On completion of this lesson you will be able to :

- classify different types of parallel processing techniques.

Traditionally, the computer has been viewed as a sequential machine. Most computer programming languages require the programmer to specify algorithms as sequences of instructions. CPUs execute programs by executing machine instructions in a sequence and one at a time. Each instruction is executed in a sequence of operation (fetch instruction, fetch operands, perform operation, store results).

CPUs execute programs by executing machine instructions in a sequence and one at a time.

This view of the computer has never been entirely true. At the micro-operation level, multiple control signals are generated at the same time. Instruction pipelining, at least to the extent of overlapping fetch and execute operations, has been around for a long time. Both of these are examples of performing functions in parallel.

When a computer application requires a very large amount of computation that must be completed in a reasonable amount of time, it becomes necessary to use machines with correspondingly large computing capacity. Such machines are called supercomputers. Typical applications that require supercomputers include weather forecasting; simulation of large, complex, physical systems; and computer-aided design (CAD) using high-resolution graphics.

As a general quantitative measure, a supercomputer should have the capability to execute at least 100 million instructions per second.

A supercomputer execute at least 100 million instruction per second.

In the development of powerful computers, two basic approaches can be followed. The first possibility is to build the system around a high-performance single processor, and to use the fastest circuit technology available. Architectural features such as multiple functional units for arithmetic operations, pipelining, large caches, and separate buses for instructions and data can be used to achieve very high throughput. As the design is based on a single processor, the system is relatively easy to use because conventional software techniques can be applied.

The second approach is to configure a system that contains a large number of conventional processors.

The second approach is to configure a system that contains a large number of conventional processors. The individual processors do not have to be complex, high-performance units. They can be standard microprocessors. The system derives its high performance form the fact that many computations can proceed in parallel.

1.2. Classification of Parallel Structures

Flynn classification of parallel processors.

A general classification of forms of parallel processing has been proposed by Flynn. Machines are categorized according to the way that data and instructions are related. According to the Flynn classification there are four basic types :

- Single instruction stream - single data stream (SISD)
- Single instruction stream - multiple data steam (SIMD)
- Multiple instruction stream - single data steam (MISD)
- Multiple instruction stream - multiple data steam (MIMD).

1.2.1. Single Instruction Stream - Single Data Stream (SISD)

A single-processor computer system is called a single instruction stream, single data stream (SISD) system.

In this classification, a single-processor computer system is called a single instruction stream, single data stream (SISD) system. A program executed by the processor constitutes the single instruction stream, the sequence of data items that it operates on constitutes the single data stream. This computer design is the basis for the traditional general purpose computer and is called a von Neumann computer. A von Neumann machine fetches an instruction from memory, decodes the instruction, and then executes it. When the action on an instruction is complete, the next instruction follows the same fetch-decode-execute sequence until a stop occurs, as shown in Fig. 9.1.

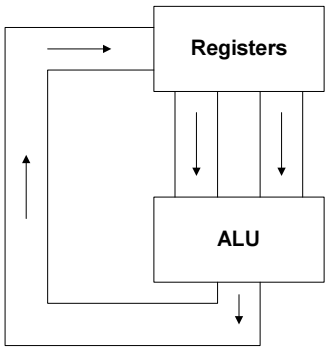


Fig. 9.1 : A simple von Neumann computer.

1.2.2. Single Instruction Stream - Multiple Data Stream (SIMD)

This scheme, in which all processors execute the same program, is called a single Instruction stream.

A single stream of instructions is broadcasted to a number of processors. Each processor operator on its own data. This scheme, in which all processors execute the same program, is called a single Instruction stream, multiple data stream (SIMD) system. This type includes machines supporting array parallelism.

1.2.3. Multiple Instruction Stream: Multiple Data Stream (MIMD)

This scheme involves a number of independent processors are called multiple instruction stream.

This scheme involves a number of independent processors, each executing a different program and accessing its own sequence of data items. Such machines are called multiple instruction stream multiple data stream (MIMD) system. This type covers the multiprocessor systems supporting process parallelism.

1.2.4. Multiple Instruction Stream Single : Data Stream (MISD)

In such a system, a common data structure is manipulated by separate processors and each executes a different program. This form of computation does not arise often in practice. Hence, no system have been built which fit this classification.

As computer technology has evolved, and as the cost of computer hardware has dropped, computer designers have sought more and more opportunities for parallelism, usually to improve performance and, in some cases, to improve reliability.

In the subsequent lessons we shall examines multiprocessing, one of the earliest and still the most common example of parallel organization. Typically, multiprocessing involves the use of multiple CPUs sharing a common memory. Next, we will look at hardware organizational approaches to vector computation. These approaches optimize the ALU for processing vectors or arrays of floating-point numbers. They have been used to implement the class of systems know as supercomputers. Finally, we will look at the more general area referred to as parallel processor organization.

1.3. Exercise

1.3.1. Multiple choice questions

a) A Von Neumann computer uses which one of the following?

i) SISD
ii) SIMD
iii) MISD
iv) MIMD.

1.3.2. Questions for short answer

a) Classify different types of parallel processing.

1.3.3. Analytical question

a) Write down the classification of parallel structures.

Lesson 2 : Pipelined Vector Processors

2.1. Learning Objectives

On completion of this lesson you will be able to :

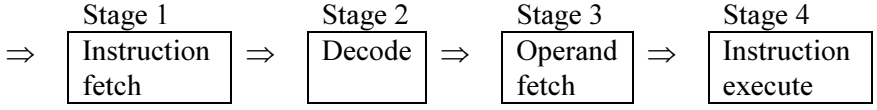
- learn about pipelined vector processors.

Many of the early attempts at exploiting parallelism in computer architecture were based on the use of pipelining. The ATLAS computer, designed and built at the University of Manchester, UK. In the late 1950s, was one of the earliest computers to make use of pipelining.

Pipelining is a well-known technique for using several pieces of hardware concurrently.

Pipelining is a well-known technique for using several pieces of hardware concurrently. Thus the system performance and overall system throughput capability of a system increases by increasing the number of instructions that may be completed per second. The intention behind this technique is to subdivide a computational task into several sub-tasks, each one handled by a separate hardware stage. A series of such hardware stages is called a pipeline. All stages operate simultaneously with independent inputs. A computational task advances from one stage to the next, and comes closer to completion as the end of the pipeline is approached the pipeline accepts new inputs before previously accepted inputs have been completely processed and output from it. When one sub-task result leaves a stage, that stage can accept new results from the previous stage. Once the pipline is full, the output rate will match the input rate. The processing of any instruction can be subdivided into the following four sub-tasks: instruction fetch, instruction decode, operand fetch and instruction execute. Suppose instruction # 1 is fetched. While it is decoded, instruction # 2 is fetched. While instruction # 1 fetches operand, instruction # 2 fetches operand, instruction # 3 is decoded and instruction # 4 is fetched. This process continues until all the instructions are executed. Fig. 9.2 shows the process of pipelining.

A true pipeline implementation establishes a series of four hardware stages, in which each stage operates concurrently with the others on one of these sub-tasks of several instructions.



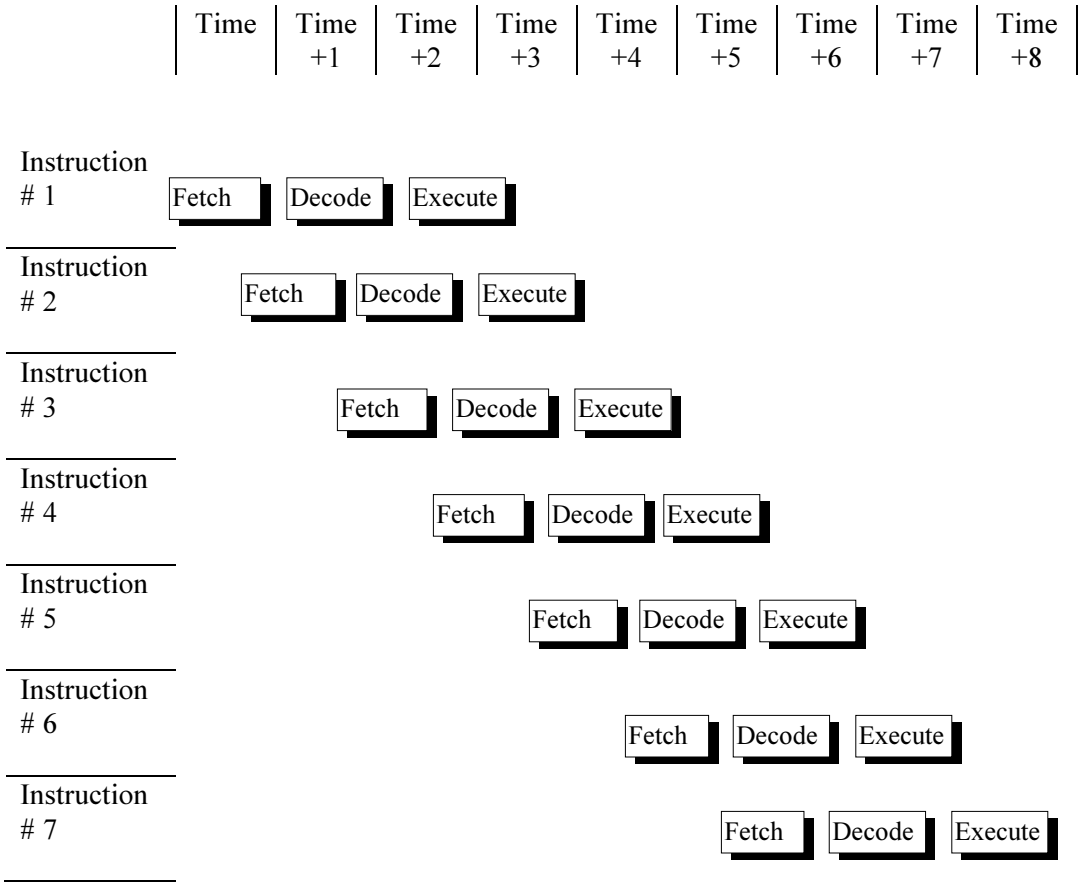


Fig. 9.2 : Process of pipelining.

Process of pipelining.

Vector pipelines are simply the extension of the idea of pipelining to vector instructions with vector operands. They form a predictable, regular, sequence of scalar operations on the components of the vectors. As a simple example, suppose that we wish to form the vector sum $c \leftrightarrow a + b$ and that the addition of two scalars can be divided into three stages, each requiring one clock to execute then.

- on the first clock cycle.
  - a_1 and b_1 are in stage 1 of the addition
- on the second clock cycle
  - a_1 and b_1 are in stage 2 of the addition and
  - a_2 and b_2 are in stage 1
- on the third clock cycle
  - a_1 and b_1 are in stage 3 of the addition
  - a_2 and b_2 are in stage 2 and
  - a_3 and b_3 are in stage 1.

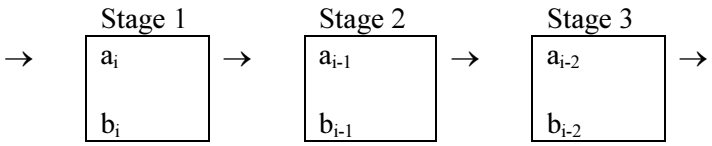


Fig. 9.3 : Status of the pipeline on the i-th clock cycle.

Status of the pipeline.

There after all these stages of the pipeline are kept busy until the final components and enter the pipe. At the end of the i-th clock cycle, illustrated in Fig. 9.3, the result ci-2 ← ai-2 + bi-2 is produced.

A vector pipeline processor is an implementation, in hardware, of the pipelining just described. Vector pipelining forms the basis of many of today's vector super computers, such as the Cray family and the Japanese (Fuji's, Hitachi and NEC) supercomputers. The Cray supercomputers have several processors and a number of pipeline units per processor and hence support both process and pipeline parallelism.

2.2. Exercise

2.2.1. Questions for short answer

a) Which are the subtasks of pipeline operation?

2.2.2. Analytical question

a) Discuss the operation of pipelined vector processor system.

Lesson 3 : Array Processor

3.1. Learning Objectives

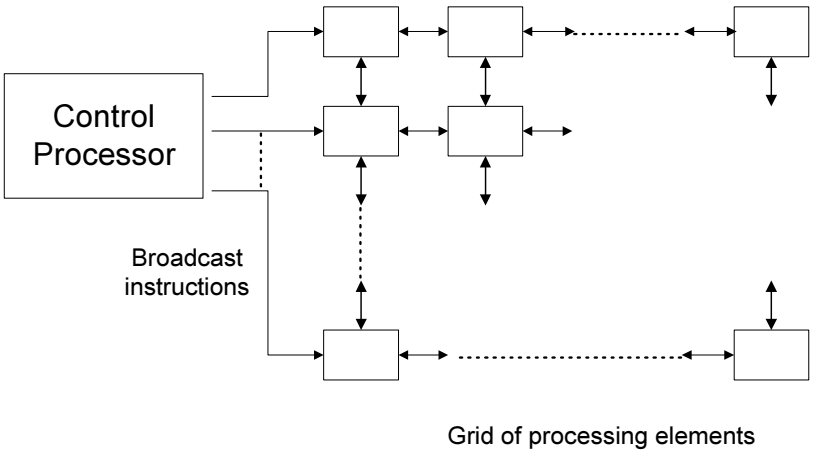
On completion of this lesson you will be able to :

- learn about array processor.

An array processor is of SIMD type processors connected together to form a grid.

One version of a parallel computer is the array processor. An array processor (or data parallel machine) is of SIMD type and consists of a number of identical and relatively elementary processors connected together to form a grid, which is often square. Each active processor (some may be idle) obeys the same instruction, issued by a single control unit but on different local data. As there is a single control unit, there is a single program which operates on all the data at once. Thus there are no difficulties with synchronization of the processors. Array processors are particularly well suited to computations involving matrix manipulations. Examples of array processors are the Active Memory Technology (AMT), Distributed Array processor (DAP). Thinking Machines, connection machine and the MasPar MP-1.

Array processor has its own memory but all the processors share the same control unit. Array processors are usually in a grid formation of rows and columns and work well for matrix applications. A two dimensional grid of processing elements executes an instruction stream broadcast from a central control processor. As each instruction is broadcast, all elements is connected to its four nearest neighbors for purposes of exchanging data. End around connections may be provided on both rows and columns.



Array processor.

Fig. 9.4 : Array processor.

The grid of processing elements can be used to generate solutions to two-dimensional problems.

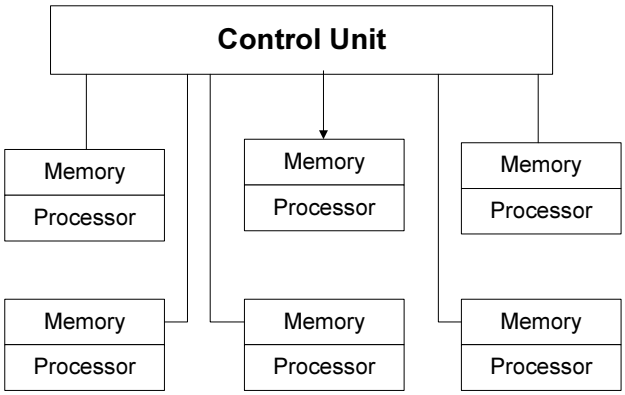


Fig. 9.5 : Example of an array processor.

Fig. 9.5 shows 2×3 array: each unit has local memory and a processor with shared control. Each processing element has a dew registers and local memory to store data. It also has a register, which is called network register. This register facilitate movement of values to and from its neighbors. The control processor can broadcast an instruction to shift the values in the network registers one step up, down, left or right. Each processing element also contains an ALU to execute arithmetic instructions broadcast by the control processor. Using these basic facilities, a sequence of instructions can be broadcast repeatedly to implement the iterative loop. Array processors are highly specialized machines. They are well suited to numerical problems that can be expressed in matrix or vector format. However, they are not very useful in speeding up general computations.

3.2. Exercise

3.2.1. Multiple choice questions

a) An array processor has its

i) own memory
ii) different memory
iii) two memory
iv) all of the above.

b) Array processors one highly specialized

i) computers
ii) machines
iii) diodes
iv) none of the above.

3.2.2. Questions for short answer

a) What is an array processor?

3.3.3. Analytical question

a) Describe the function of an array processor.

Lesson 4 : Multiprocessor Systems

4.1. Learning Objectives

On completion of this lesson you will be able to

- learn about multi-processor systems.

Multiprocessor system involves a number of processors capable of independently executing different routines in parallel.

Multiprocessor system involves a number of processors capable of independently executing different routines in parallel. This type of system supports MIMD architecture. A general MIMD configuration, usually called a multiprocessor is shown in Fig. 9.6.

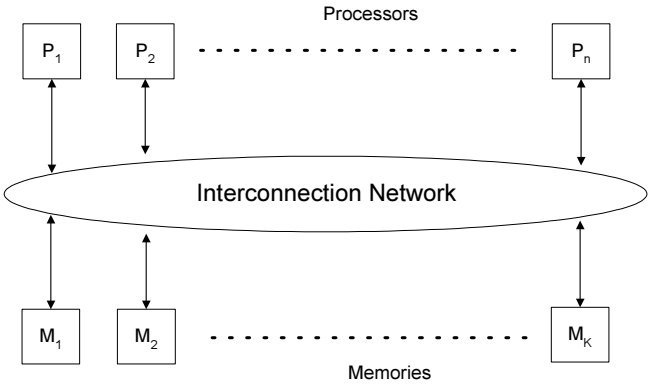


Fig. 9.6 : A general multiprocessor.

An interconnection network permits n processors to access K memories, so that any of the processors can access any of the memories.

An alternative arrangement, which allows a high computation rate to be sustained in all processors, is to attach some local memory directly to each processor.

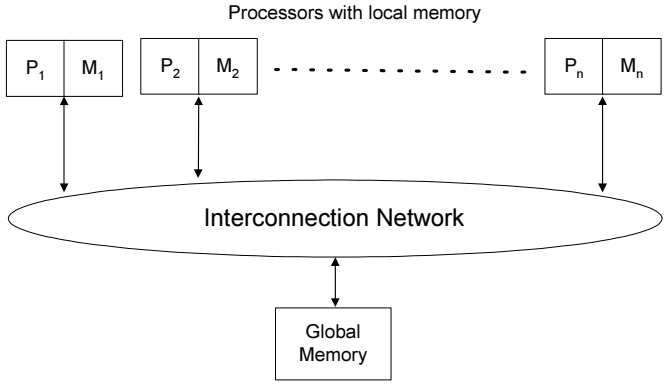


Fig. 9.7 : A global memory multiprocessor.

The way in which the local memory is used varies from one system to another. Normally, each processor accesses instructions and data form its own local memory. The interconnection network provides access from any processor to a global memory. This memory holds programs and data that can be shared among the processors, providing a means for interaction among the tasks performed by the processors. These tasks may be components of a single large application that has been divided into sub-tasks. They may also be independent tasks that require the use of some shared resources such as printers or magnetic disks. The relationship between the information stored in the local memories and that stored in the global memory.

The physical connection between processors and/or memory modules may take the form of a common bus.

The physical connection between processors and/or memory modules may take the form of a common bus. The processor themselves may have communications capabilities which allow them to be directly connected together. Alternatively, some sort of switching mechanism may be employed.

A crossbar switch is one which is able to connect a number of inputs with a number of outputs according to specified permitted combinations.

For example, a 2×2 crossbar switch has two inputs and two outputs. The switch will permit input i (i = 0 or 1) to be connected to output i, and also input i to be connected to output (i+1) mod 2.

Further, a broadcast can be achieved by permitting either input to be connected to both output simulate aneously. By increasing the number, N, of inputs and outputs the number of possible interconnections grows rapidly.

Another way of implementing an interconnection network is to construct a multistage switch in which each component is a crossbar switch. For example, four inputs can be connected to four outputs via a two-stage system in which the switches at each stage are 2×2 crossbars.

4.2. Speed-up and Efficiency

Speed-up and efficiency.

Let T_P be the execution time for a parallel program on p processors. We define the following terms;

- S_P, the speed-up ratio on p processors, is given by

$$S_p = T_0 / T_p$$

Where T_0 is the time for the fastest serial algorithm on a single processor. It should be noted that this definition of speed-up ratio compares a parallel algorithm with the fastest serial algorithm for a given problem. It measures the benefit to be gained by moving the application from a serial machine with one processor to a parallel machine with p identical processors. Even if the same algorithm is employed we normally expect the time taken by the parallel implementation executing on a single processor (T_1) t exceed the time taken by the serial implementation on the same processor (T_0) because of the overheads associated with running parallel processes.

4.3. Programming Languages

Programming Languages

The earliest machines had to be programmed in machine code, but it soon became clear that some higher level of abstraction was necessary. The development of FORTRAN (Formula Translator), ALGOL (Algorithmic language) and their derivatives. Closely reflect the von Neumann model; a FORTRAN program consists of a sequence of instructions which the program writer expects to be obeyed in sequence unless he explicitly includes a jump instruction to some other part of the code.

This has been accompanied by the development of software tools designed to assist the transfer of a sequential code to a parallel environment.

There are very few programming languages in common use that explicitly support parallelism. Rather, extensions have been made to existing languages, such as FORTRAN and Pascal (and even Basic), to provide appropriate facilities. Two languages which explicitly support concurrency, namely, Ada and occur, and also extensions to FORTRAN which are appropriate for multiprocessor systems.

In the case of array processors, languages such as CM FORTRAN and DAP FORTRAN have been developed to permit exploitation of the underlying architecture. For vector processors we can use a standard language, such as FORTRAN, and relay on a compiler to vectors the code, that is, produce object code which makes use of the hardware pipelines.

4.4. Performance Measurements

Performance Measurements

To compare the performance of various computer systems some means of measuring their speed is clearly required. We have already mentioned Mflops as a means of quantifying the speed of a floating-point unit. An

alternative performance measure frequently employed is mips (millions of instructions per second).

Manufacturers usually quote performance which refer to :

- peak speed, calculated by determining the number of instructions 9 or flops 0 that theoretically could be performed in the cycle time of the processor.
- Sustained speed, a measure of the speed likely to be achieved for a 'typical' application.

4.5. Processors and Processes

Processors and Processes

In the above discussion, we have used "multiprocessor" as a generic term for a machine which consists of a number of computing units, some form of interconnection and, possibly, a number of separate memory modules. Now, we need to differentiate between 'processors' and 'processes'. The former is a functional unit (hardware) and the latter is a combination of operations performed on data (software). On a uniprocessor we have only one process executing at any one time, but may have multiple processes time shared. On a multiprocessor system, a one-to-one correspondence exists between processors there will be P processes with one process per processor and no time sharing. On a local memory system, therefore, inter process communication will involve messages being sent along inter processor links.

4.6. Exercise

4.6.1. Multiple choice questions

a) The system performance and throughput capability system can be increased by increasing

i) clock speed
ii) number of instructions per second
iii) number of processor
iv) number of data.

b) FORTRAN stands for

i) Formula Translator
ii) Formula Transmission
iii) Function Translation
iv) none of the above.

c) A general MIMD configuration usually called

i) a multiprocessor
ii) a vector processor
iii) array processor
iv) none of the above.

4.6.2. Questions for short answers

a) How does multiprocessor system work ? What is the function of a crossbar switch?
b) Define speed up and efficiency.
c) What is multiprocessor?

4.6.3. Analytical question

a) Give a brief description on multiprocessor systems.