Veszprémi Egyetem, Müszaki Informatika Szak

Digitális rendszerek és Számitógép architektúrák I.

Tavaszi félév, heti (5+1 óra)

Előadás jegyzet

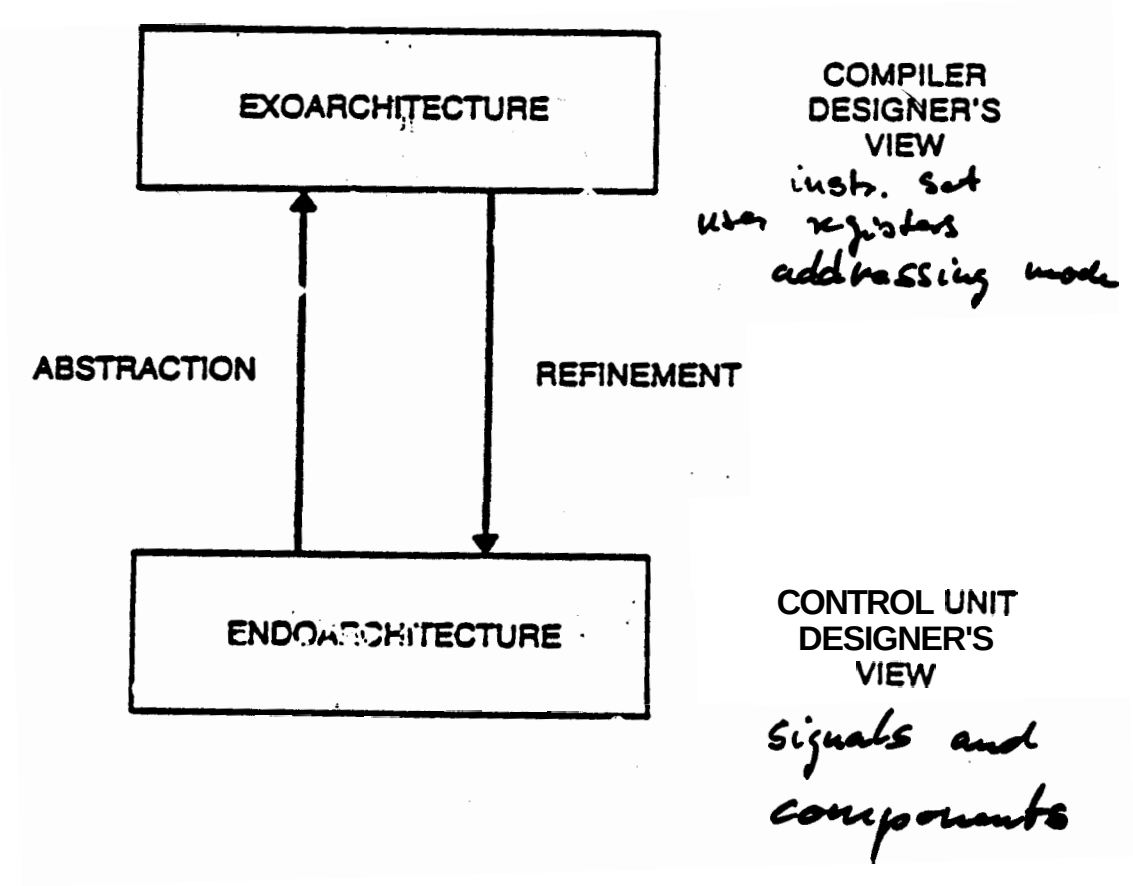
Összeállitotta:
Katona Attila
dr. Szolgay Péter

Veszprém, 1996.

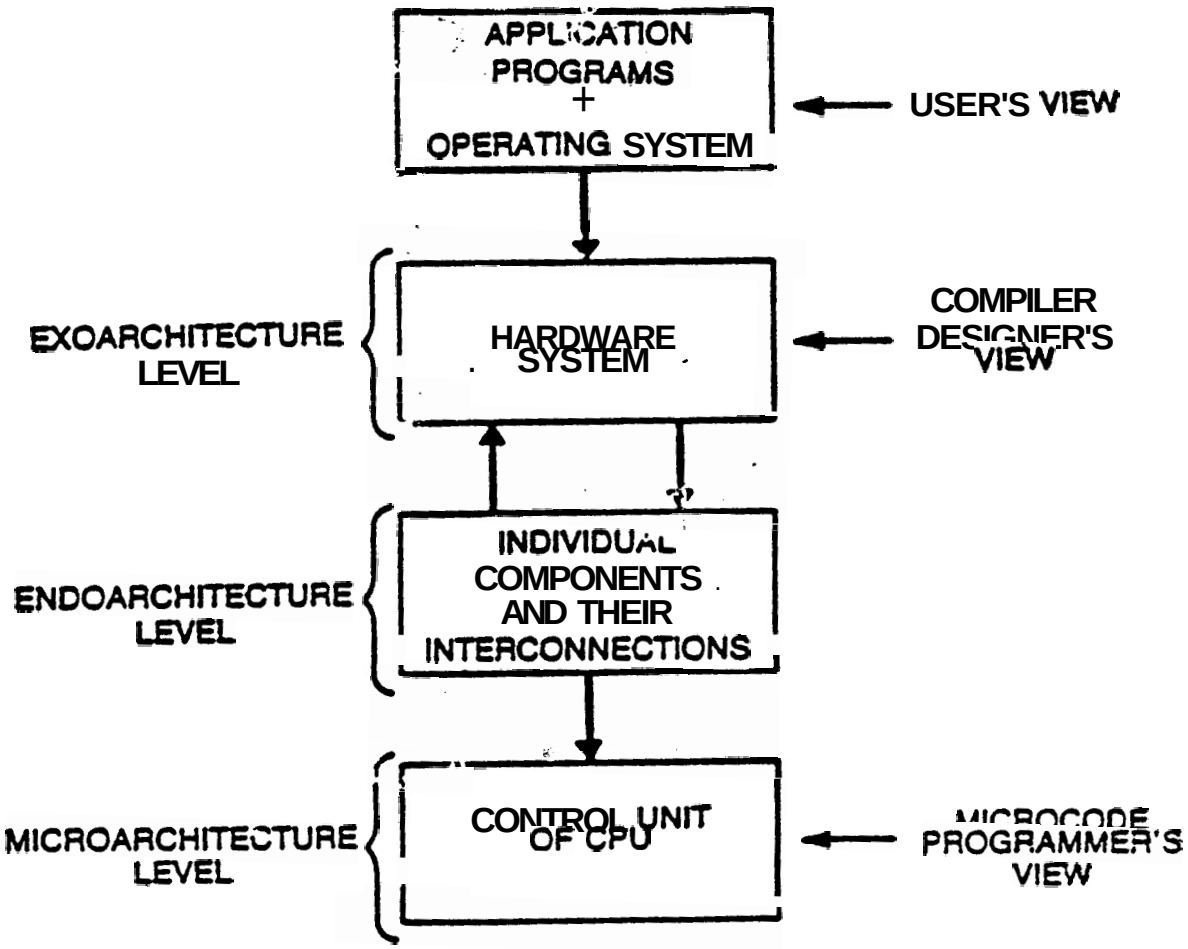
DEFINITION OF COMPUTER ARCHITECTURE

- *Computer architecture* is an abstraction of the hardware; it is concerned with the structure and behavior *of* the hardware without regard to a particular system.
- Architectural attributes include the functional components as well as the internal form (i.e., the structure) of these functional components.

EXOARCHITECTURE VS. ENDOARCHITECTURE



DEFINITION OF COMPUTER ARCHITECTURE (cont.)



Computer Design and Architecture

L. HOWARD POLLARD
University of New Mexico

Prentice-Hall International, Inc.
Englewood Cliffs, N.J.

1

Introduction

Computers have become a common fixture in today's society, prevalent not only as a computational tool for scientific use, but also providing the control mechanism for a wide variety of systems. Computers and computer-like devices are found in appliances, automobiles, supermarket checkout stands, telephones, radios, and televisions, where they are used to control the function of the device. They are also found in information systems, such as office computer systems, inventory control systems, and library reference systems. Computers form the basis for engineering workstations, personal computers, and intelligent communication systems. These applications have become reasonable since the cost and size of computer devices have decreased. Not only have computers proliferated into many different areas, but the power of large "mainframe" computers has continued to increase. These large machines provide the tools necessary for research and applied technology in science, engineering, medicine, and many other areas. New machines with more capabilities will permit development of new techniques for analysis and research in many areas.

The field of computer design encompasses a wide variety of disciplines, each of which forms a necessary part of the whole system. The electrical nature of the systems is understood through applications of principles of electrical engineering. Included in this area is the semiconductor technology used to create the logic, memory, processors, and other devices of a computer system. Different technologies provide devices with different speeds, different power, and different capabilities, and the task left to the system architect and computer designer is to use these devices in reasonable ways to meet the design objectives of the system.

Other areas of electrical engineering are required in computer systems. The power needed to drive the system must be transported, converted, isolated from noise, and delivered to the active devices in the system. Also, steps must be taken to ensure that the devices do not generate noise which can propagate back into the power supply system of the machine. The signals used to communicate between

devices also present a number of problems. As time allotted for information transfer decreases, the connection can no longer be treated strictly as a wire; the transmission line characteristics of the interconnect must be considered, and steps taken to reduce any noise which may arise from reflections and impedance mismatches. In short, all electrical aspects of the system are important in producing a computer system which will be both functional and reliable.

Another type of information needed to understand computer systems has traditionally been the domain of computer science. This includes a knowledge of the operating system, which provides a mechanism for control, not only of the processor, but also of the input and output (I/O) system, the memory system, and the user interface. It includes an understanding of semaphore operations, why they are needed for resource management within both operating systems and programs, and the responsibilities of the processor architecture and implementation in creating and managing semaphores. It includes a working knowledge of data structures and their applications, both in the operating systems and in application programs. It includes a knowledge of the operation and function of compilers, the languages they work with and desirable characteristics of programs and subroutines. It includes a wide variety of ideas and concepts utilized to apply the computational and control capabilities of a computer system to the solution of problems encountered in different facets of science and industry.

Application of the computational capabilities of a computer system to real problems also mandates an understanding of numerical analysis. A mathematical model of a process can be created to provide a vehicle for studying the various aspects of the problem. Conversion of the mathematical model into a computer program results in representation of information in a limited fashion. That is, the computer system can represent a limited number of values, and the user must ascertain that the limits of the representation mechanisms and the arithmetic interactions of the variables will not introduce unacceptable errors into the results.

Many of the aspects of the computer system relating to its logical structure, its speed of operation, and its interconnection mechanisms are the domain of computer engineering. This includes:

- The design of the arithmetic elements used in a system, analyzing algorithms and methods to produce the desired answers within an acceptable time.
- The creation of the set of instructions used to control the system, which define the apparent structure of the system.
- The interconnection methods for arithmetic units, memorys, registers, and other units, which define the actual structure of the system.
- The method used to define and control the data flow between the major portions of the systems.
- The techniques used to provide communication between the computer and other devices, such as disks, tape units, and terminals.

An understanding of these aspects of the computer system provide a basis on which reasonable decisions can be made — both in the design process, where a computer system is being created, and in the application process, where the use of different techniques results in improved performance for various problem areas.

This text has been written to address these computer design issues. Design techniques and architectural issues for each of the above areas will be discussed

and tradeoffs will be illustrated. However, before we consider some of the details of computer systems, let us look at the origin of the modem computer.

1.1. Early Computational Devices

The early history of computers is fascinating, and we will not attempt here to give a complete list of the machines and activities. This information is available in a variety of texts [Haye88, Stal87], as well as the series ***Annals of the History of Computing.*** Since necessity is the mother of invention, computers came into being to provide a mechanism for removing the drudgery from repeated calculations. As the techniques were explored, modified, and refined, new ideas and concepts continued to emerge. There has always been tradeoff between the available technology and the capabilities of the computing machinery (based on that technologies). As the limits of one technique were reached, other ideas were generated and explored to utilize different techniques to provide faster machines with more features. As different technologies were used, alternate solutions and new features were added to machines, and new machines (based on the new technologies) were developed. This process continues to provide new devices and new capabilities as the number of available computer systems expands.

One of the earliest calculating devices was created by the French philosopher/scientist Blaise Pascal around 1642. This device, basically a mechanical counter, was created to automate the addition and subtraction process. Numbers were expressed in decimal form on two sets of six wheels, one wheel for each digit. Thus, the unit had the capability of manipulating six-digit numbers. The digits of each number were represented by the position of the wheels, each of which had the ten numerals engraved on it. One set of wheels acted as an accumulator register, and another number was entered onto the second set of wheels. The two sets were connected by gears, and when one set of wheels was turned, the other set was incremented accordingly. The principle innovation of the system was the creation of a mechanical carry device, which automatically incremented the appropriate wheel position by one when the wheel of one lower significance rolled over another decade. To handle negative numbers or subtraction, a complements representatidn was used.

Another mechanical calculating system was created around 1671 by the German philosopher/mathematician Gottfried Leibniz. This unit incorporated the addition/subtraction capability of the Pascal device, and extended it to perform multiplication and division automatically. The multiplication operation was implemented by using chains and pulleys, deriving the appropriate information from sets of wheels which were used to identify the multiplier and multiplicand. Like the Pascal device, this one utilized mechanical mechanisms to provide the basic functions needed for computation. In fact, one could say that the Leibniz device was the first four-function calculator.

A prolific scientist in the field of computing was Charles Babbage, an Englishman who worked on two different computing systems, as well as creating a wealth of supporting material that was ahead of its time. The first machine, called the Difference Engine, was created around 1823 to automatically generate mathematical tables. The machine was to not only calculate a number, but also directly transfer this information to the plates used for printing the tables. The only operation supported was addition, but the addition mechanism could be used repeatedly to create the desired result, using the method of finite differences to

represent or approximate the functions needed. The Difference Engine consisted of a number of mechanical registers, each of which stored a decimal number. The registers were connected in pairs by an mechanical addition mechanism that functioned much like Pascal's calculator. To form a result, initial values were entered into the registers, and the system could then be driven by a motor of some kind to produce the final result. Although some difference engines were built, the unit proposed by Babbage was never completed, partly because of mechanical difficulties and partly because Babbage became more interested in a new device, the Analytical Engine.

Whereas the Difference Engine was designed to create answers using the technique of finite differences, the Analytical Engine was designed to perform any mathematical function in an automatic fashion. The basic organization of the Analytical Engine is shown in Figure 1.1; this system bears a striking resemblance to the computers of today. The principle parts were the store, the mill, the control section, and the output section. The store was a memory unit composed of sets of counter wheels; the design called for storage of 1,000 numbers, each consisting of 50 digits. The mill, corresponding to the arithmetic/logic unit (ALU) in more modem machines, was capable of performing the four basic arithmetic operations. The output unit was intended to be either printed on paper or punched on cards. The system was controlled by two sets of punched cards: the operation cards identified the basic operation that was to be performed by the mill, and the variable cards identified the source of the operands used in the calculation, as well as the destination of the result. One of the most significant contributions of the Analytical Engine was a mechanism for altering the sequence of operations depending on the state of the machine, basically a conditional branch capability. The testable condition was the sign of a number; if the number was positive, one course of action was followed; if the number was negative, a different set of instructions was identified.

Although the system was proposed and the design was worked on for many years, only a small portion of the system was actually constructed. Had he been successful in construction of the system, Babbage estimated that the time required for an addition operation was on the order of a second, and the time required for a multiplication was on the order of a minute. It is doubtful that a mechanical computer of the size and complexity of the Analytical Engine could ever be built.

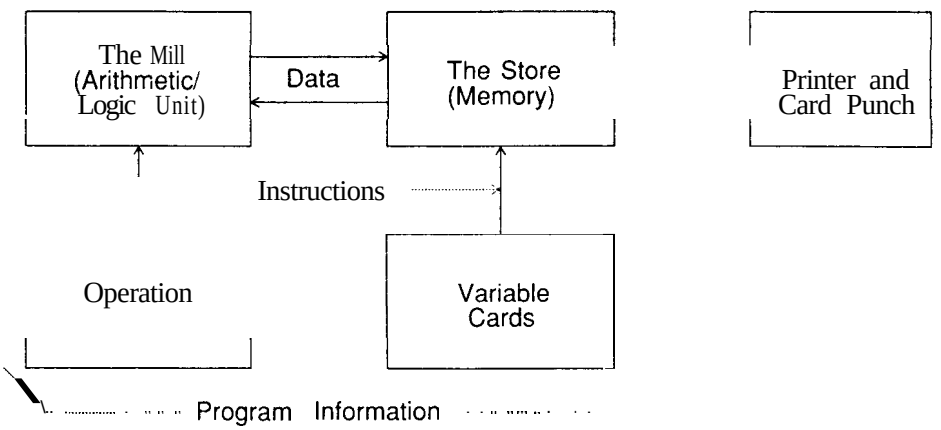


Figure 1.1. Block Diagram of Babbage's Analytical Engine.

A number of mechanical calculators were implemented in the early 1900s, and these contributed to the general idea of automating the computing process. Other mechanical devices would play a role in the advancement of computing devices. One of these was the Jacquard Loom, a device that automated the weaving of rugs by using patterns punched on cards. This device was actually operational by about 1801, and the idea of using cards for controlling machines was used by Babbage and others. Another card-oriented machine was the punched-card tabulating machine, invented by an American, Herman Hollerith. One of the first use of Hollerith's card system was processing data taken in the 1890 census of the United States. The characteristics of the population were punched on cards, entered into the system, and counted mechanically. Hollerith formed the Tabulating Machine Company in 1911, which later merged with several other companies into a venture that would become International Business Machines. Card systems were used for data entry and output in computer systems for many years.

During the late 1930s a German engineering student named Konrad Zuse created several models of electromechanical computational systems. He chose as the active unit of the system a mechanical relay, and used a binary number system, rather than a decimal system, to represent the numbers. The first model, the Z1, was a primitive machine, with minimal processing capability and a memory based on mechanical relays. The third model, the Z3, was completed in 1941. This machine was also based on electromechanical relays, and is perhaps the first operational program-controlled general purpose computer. The input was through a punched tape mechanism, which utilized discarded photographic film in which instructions were represented by hole patterns punched by the programmer. Most of Zuse's machines were destroyed by the bombing of Berlin, and although Zuse later received support from IBM and Remington Rand, his efforts did not greatly influence the other computational systems that followed.

Another electromechanical system was created by Howard Aiken, a physicist and mathematics professor at Harvard University. Whereas it appears that Zuse was not aware of the work of Babbage, Aiken did know of the previous work, and followed some of the ideas presented there. The effort was initiated in 1939, and the Mark I became operational in 1944. Information was stored in wheels like the Babbage machines, but the computational system was composed of relays. It could store 72 23-digit decimal numbers, and instructions were input into the system with a punched paper tape; each instruction contained an operation and two addresses. Once operational it, could do an addition in 6 seconds and a division in 12 seconds.

Perhaps the first electronic computer system was created by John V. Atanasoff, a physicist at Iowa State College, which later became Iowa State University. Between 1937 and 1942 he and a graduate student. Clifford E. Berry, worked on a system that would perform gaussian elimination solutions for sets of equations. Their system was totally electronic in nature, and used capacitors to store information, in much the same way that dynamic RAMs (random access memories) store information on capacitors created with semiconductor technology. He also used a binary number system for information representation, and organized the functional units by separating the logic and arithmetic portion of the system from the memory portion, as well as the I/O portion. The resulting computational system performed adequately, but the punched card I/O system introduced an error about once every 10,000 operations. This portion of the system was used extensively in the operation of the unit, and so the errors were unacceptable. However,

before the source of the problem could be located and corrected, World War II interrupted work on the system. and further efforts were suspended.

1.2. The Computer Generations: Technology and Innovation

The early efforts in devising computing systems inspired the creation of other systems to perform automatic computation functions. As more experience was obtained with computing machines, and as the technology changed, different computational systems emerged. One of the ways of classifying the machines that have followed is to group them into generations, using not only the chronological position of the system, but also the characteristics and capabilities of the systems.

1.2.1 The first generation (??–1953)

The first generation includes the early machines, as well as machines created until the mid-1950s. These machines used either electromechanical elements or tubes for logic, and a variety of mechanisms for memory. Some of the systems were organized in a bit serial manner to more effective utilize the expensive hardware devices. Some of the systems operated on entire words simultaneously. to provide high speed operation. The first generation systems were primarily for scientific purposes, with business applications a low priority. For the most part, these machines were programmed at the machine level, and users of the systems were expected to provide all data and all of the required program control.

One of the first well-known computers to use electronic components was the **ENIAC** system (electronic numerical integrator and calculator). This system was built at the University of Pennsylvania by John W. Mauchly and J. Presper Eckert. Like many machines of this era, one of the principal motivations for the system was the need to generate tables automatically. Work on the **ENIAC** project began in 1943, and it was completed in 1946. The system was physically very large, with over 18,000 vacuum tubes. The electronic nature of the unit resulted in a system that was considerably faster than any previous computer system, with an addition time of approximately 3 ms. The data memory of the system consisted of 20 accumulator registers, each of which could store a 10-digit decimal number with its sign. Each digit of storage required ten flip-flops, which were organized as a ring counter: the active flip-flop indicated the value of the digit stored in that digit position. Each of the accumulators in the system combined arithmetic (addition and subtraction) logic with the storage logic. Hardware units were also provided for multiplication, division, and square root calculation. As can be seen from the **ENIAC** block diagram included in Figure 1.2, data input to the system was provided by a card reader system, and output was either printed or punched. The connections within the system were physically made with wires configured on panels; to connect one accumulator to another, the appropriate points were manually connected to one another. The programming was also accomplished manually, setting switches and establishing connections with cables between control points. In addition, constants used during the computation could be stored in the function tables and used as needed.

The **ENIAC** system was very cumbersome to program, since the program was actually determined by the physical arrangement of the cables in the system. The next step was to create a system in which the program would be stored in memory along with the data, so that the program could be altered by modifying

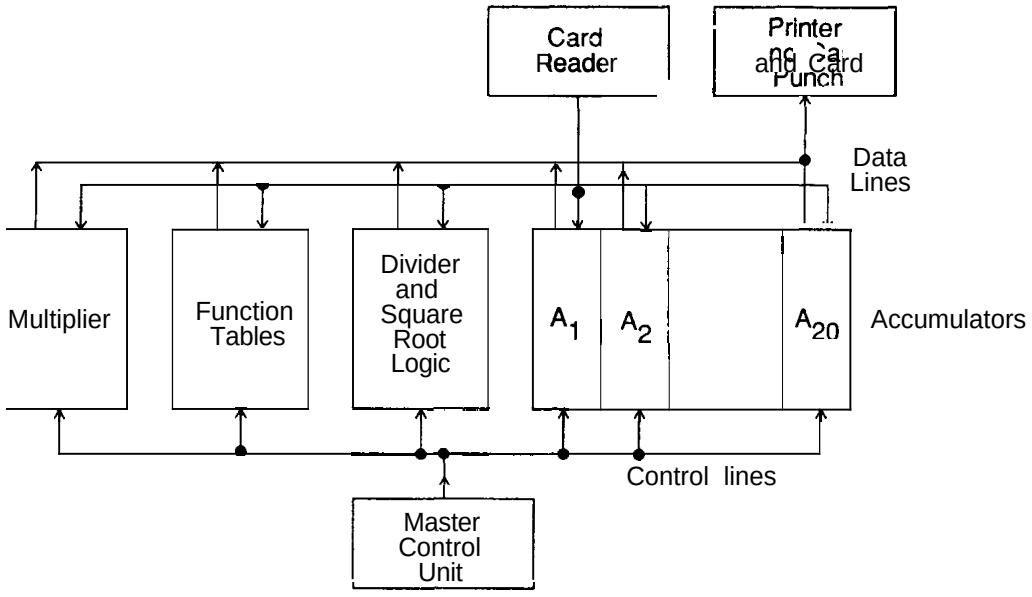


Figure 1.2. Block Diagram of ENIAC.

the contents of the memory during program execution. It is interesting to note that, in the Harvard Mark I and other machines (see the block diagram of **ENIAC** in Figure 1.2), the data was kept separate from the program. This mechanism, often called the Harvard architecture, can be found in many systems today, especially systems like real time digital signal processing units. The benefit is that there are independent paths to data memory and control memory, and both can be used simultaneously; this leads to a higher effective system speed. The cost of this mechanism is that two separate memory units must be provided, with their data paths, addressing decoders, and other costs. This not only results in a higher costs associated with the memory, but also imposes different limits on the system. That is, one program may utilize all of the available program space and need more, while not making use of all of the data memory; another program may require very little program memory, but need more data memory than is available. The next architectural change combined the two memories, which permitted the program to be modified as mentioned above, but which also allowed the available memory to be used by program or data as required by the system.

In **1945** John von Nuemann, a Hungarian-born mathematician who was a consultant working with Mauchly and Eckert on the **ENIAC,** proposed the creation of a new system, the **EDVAC** (electronic discrete variable computer). This system was to operate on what is called the stored program concept, where the program and data share the same memory, and thus the program could be modified to extend the possible execution modes. Although there is evidence that this concept did not uniquely originate with von Neumann and his colleagues, his name is most often attached to it. The **EDVAC** system was developed at the University of Pennsylvania, and differed in many respects from the previous systems. Like Atanasoff's machine, it utilized a binary number system to represent the information. The storage area was much larger than earlier systems, capable of storing **1,024,** or 1K, words of information. In addition, the system had a secondary storage unit capable of storing **20K** words. Both of these memories were made from serial delay lines, the main memory from mercury delay lines, and the larger storage unit from magnetic delay lines. Because of the serial nature of the delay

line storage, and to minimize hardware costs, the arithmetic was performed in a serial fashion, working on a single bit at a time. The words were 44 bits long, and there were three basic types on instructions. Arithmetic instructions were of the form:

$$A_1 \quad A_2 \quad A_3 \quad A_4 \quad OP$$

The OP identified the operation to be performed (+, −, x, or ÷), and the A_i specified the addresses involved. The function was performed on the information stored at the locations specified by A_1 and A_2, and the result was placed at the location specified by A_3. The next instruction to execute was found at location A_4. The format for the conditional branch instructions was similar:

$$A_1 \quad A_2 \quad A_3 \quad A_4 \quad C$$

If the number stored at A_1 is not less than the number stored at A_2, the next instruction to execute is located at A_3; otherwise, the next instruction to execute is located at A_4. The other type of instruction was an input/output instruction of the following format:

$$A_1 \quad \{1,2\},N \quad A_3 \quad A_4 \quad W$$

If the {1,2} choice was a 1, then the words from A_1 to A_3 were stored on the delay line wire N. If the {1,2} choice was a 2, then information from delay line wire N was transferred to locations starting at A_1 and ending at A_3. Again, the next instruction to execute was located at A_4. Actual input and output operations moved information directly to and from the delay lines.

A number of observations can be made about this system, two of which we will identify here. First, this was a memory-to-memory architecture, and no registers were involved in the instructions. Second, there was no default "next" instruction; each instruction identified the location of the instruction which was to follow. These architectural decisions resulted in some unique system characteristics, and incurred one set of costs. In comparing architectures, much can be learned by comparing their use of specific system resources. For example, instructions that required four separate addresses necessitated instruction words long enough to include all four identifiers; for 1K locations, this was 10 bits per address, or 40 bits total for the four addresses. Thus, only four bits were left to identify the instruction itself.

In 1946, von Neumann, along with Arthur W. Burks and Herman H. Goldstine, made a proposal to the Army for a new computer system that combined many of the characteristics of the previous machines and added some new concepts. This machine was called the IAS after the Institute for Advanced Studies at Princeton, where the work was done. The machine was worked on for many years, and finally became operational in 1952. This system formed a basis for many of the computers that followed, so we will describe some of its characteristics.

The unit was constructed from a few basic modules: the memory, the arithmetic units, the control unit, and the input/output capabilities. Since IAS was principally a computer, the four primary arithmetic functions were supplied in specialized hardware. One of the operands required for the arithmetic functions was located in a predefined register, the other operand was obtained from

memory, as with EDVAC. The result of the operation was placed in the accumulator. Thus, only one address was required with the instruction, and the instruction length could be correspondingly smaller. Also, instead of having each instruction identify the location of the subsequent instruction, the next instruction to execute was assumed to be located in the next location in memory; this further reduced the address requirement in an instruction. This type of organization became known as a single address machine, since only one address was required in any instruction.

The memory of the IAS system was provided by an array of X-Y cathode ray tubes, each storing a 64×64 array of bits. Thus, the memory had 4,096 locations for storage of either data or instructions; an address to specify a unique location in the memory required 12 bits. Transfer of information between the memory and the other portions of the system occurred over a parallel path, which provided a higher speed system than the serial information transfers used in EDVAC.

The word size was selected based on the expected numerical accuracy required by the workload, in conjunction with the number of bits required to represent the instruction and the address. The formats are shown in Figure 1.3. Eight bits were selected to represent the operation code of the instruction, although fewer bits could have been used. The single address required by an instruction required an additional 12 bits. These two elements could be contained in a word size of 20 bits, but the arithmetic precision offered by 20-bit words was not sufficient to solve the problems for which it was intended. Therefore, the word length was extended to 40 bits, and two instructions were included in each program word in memory. The data was represented in a fixed point scheme, with a sign bit, an assumed radix point, and 39 bits of fractional data. This format also permitted the bits to represent integers, if appropriate assumptions were made about the data manipulation techniques.

The instructions of IAS are included in Table 1.1; the original nomenclature has been changed to a more descriptive method similar to the instruction sets of more recent machines. The instructions identify the manipulations that can be controlled by a programmer in moving data in the system. A block diagram of the organization of IAS, its modules and their interconnection paths is given in Figure 1.4. Table 1.1 identifies several types of instructions that move information within the system. The data transfer instructions move information between the memory and the two data registers: the ACCUMULATOR and the MQ (multiplier-quotient) register. The arithmetic instructions operate on the data

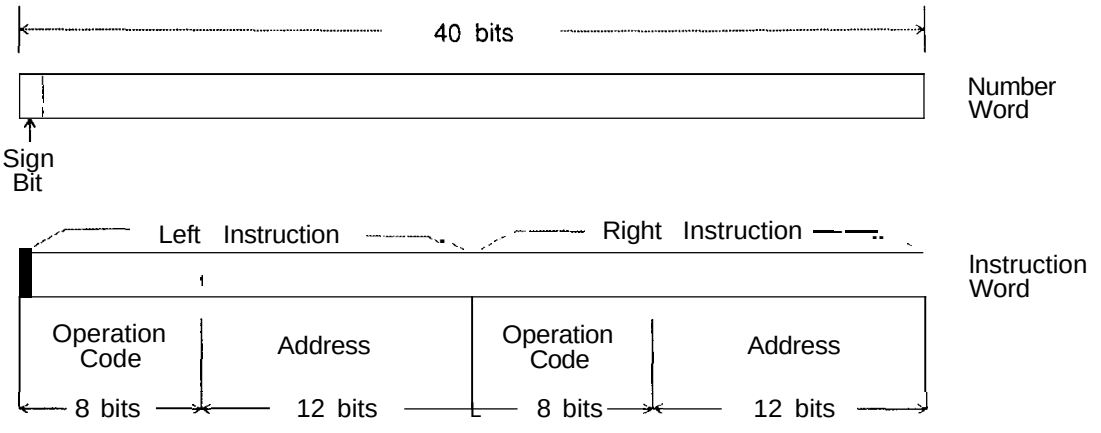


Figure 1.3. Data and Instruction Formats for IAS.

Table 1.1 Instructions for **IAS.**

Data transfer instructions	
Instruction	*Description*
LDA X	Load **ACCUMULATOR** with value stored at location X.
LDAM X	Load **ACCUMULATOR** with negative of value stored at location X.
ABS X	Load **ACCUMULATOR** with absolute value of number stored at location X.
ABSM X	Load **ACCUMULATOR** with negative of absolute value of number stored at location X.
LDM X	Load **MQ** register with value stored at location X.
MQA	Load **ACCUMULATOR** with value stored in **MQ** register.
STOR X	The value of the **ACCUMULATOR** is transferred to location X.

Arithmetic instructions	
Instruction	*Description*
ADD X	Add number stored at location X to **ACCUMULATOR.**
SUB X	Subtract number stored at location X from **ACCUMULATOR.**
ADDABS X	Add absolute value of number stored at location X to **ACCUMULATOR.**
SUBABS X	Subtract absolute value of number stored at location X from **ACCUMULATOR.**
MULT X	Multiply the number stored in **MQ** register by value stored in location X, leave 39 most significant bits in **ACCUMULATOR,** and leave 39 least significant bits in **MQ** register.
DIV X	Divide value in **ACCUMULATOR** by value stored at location X; leave remainder in **ACCUMULATOR** and quotient in **MQ** register.
LFTSHFT	Multiply the number in the **ACCUMULATOR** by 2, leaving it there.
RGTSHFT	Divide the number in the **ACCUMULATOR** by 2, leaving it there.

Jump instructions	
Instruction	*Description*
JMPL X	Next instruction to execute is in most significant half of location X.
JMPR X	Next instruction to execute is in least significant half of location X.

Conditional branch instructions	
Instruction	*Description*
BRANCHL X	If number in **ACCUMULATOR** is nonnegative, next instruction to execute is in most significant half of location X.
BRANCHR X	If number in **ACCUMULATOR** is nonnegative, next instruction to execute is in least significant half of location X.

Address modification instructions	
Instruction	*Description*
CADRL X	The address bits (12 least significant bits) of the most significant half of location X are replaced with the 12 least significant bits of the **ACCUMULATOR.**
CADRR X	The address bits (12 least significant bits) of the least significant half of location X are replaced with the 12 least significant bits of the **ACCUMULATOR.**

located in the data registers; operands are retrieved from memory as needed, and the results are left in the registers. The jump instructions and the branch instructions allow program control to be moved to another location in the memory by

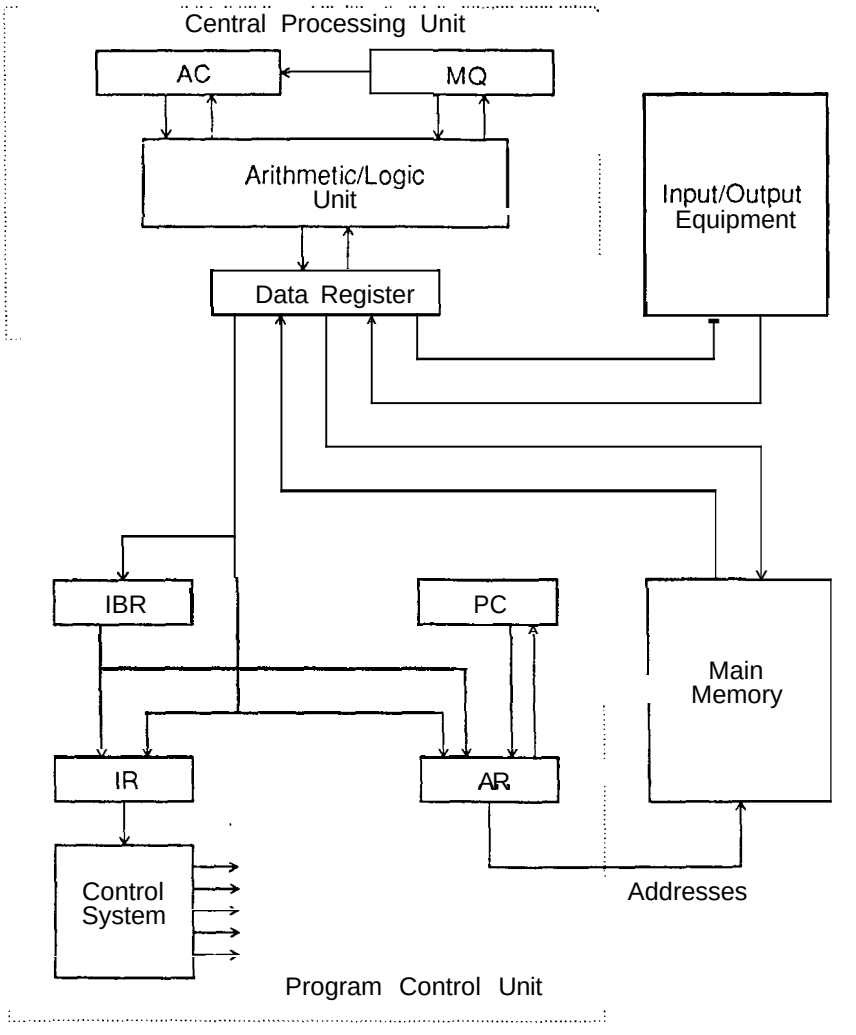


Figure 1.4. Block Diagram Representation of IAS.

modifying the contents of the control registers. The address modificatiori instructions allow a program to dynamically modify the instruction stream. The registers in the program control portion of the system include the program counter (PC), the address register **(AR),** the instruction register (IR), and the instruction buffer register **(IBR).** The PC is responsible for identifying the location in memory where the next instruction will be found. The contents of the PC and the instruction stream are used by the **AR,** which specifies the address to be used in main memory. Memory interaction is accomplished by transferring information to and from the data register. When an instruction pair is extracted from memory, the active instruction is directed to the **IR,** and the other instruction is sent to the **IBR** to await the time of execution. Note that instructions listed do not support input/output operations; transferring information to and from **IAS** was accomplished by moving blocks of data to and from main memory via the registers in the data processing unit.

Note that the instruction set had three different ways to deal with the locations in memory. If the location was used in a data transaction, the entire word was used. If the location was used as the target of a jump or a conditional branch, then the appropriate half of the word was utilized. Finally, modification of the address bits of one the two instructions is possible.

Although this machine was more capable in many respects than the earlier machines, a number of shortcomings became evident as the machine was used. The address modification mechanism was awkward to utilize in an efficient way, and later machines extended the accessing methods to facilitate identification of operands. One of the obvious omissions is a method of structuring programs, that is, a subroutine call-return mechanism. This would facilitate using a single section of code to implement often occurring functions. The scientific nature of the expected workload is evident from the instruction set; programming logical or nonnumerical types of operations was somewhat difficult to accomplish. Systems that handled these problems in a different way, as well as those using different technologies, became more prevalent as new systems moved into another generation of machines.

1.2.2 The second generation (1952–1963)

Many of the first machines were single systems, created for a specific problem or set of problems. As the possible applications of the systems became evident, commercial systems were created. IBM and Sperry enjoyed the most commercial success, being joined later by other firms. The second generation of machines brought to light the methods and lessons learned earlier, together with new technology for both functional units and storage. The transistor, invented in the late 1940s, became one of the principle active devices in computer systems, although tubes continued to play a role. The use of transistors greatly reduced the power required to run a computer system, as well as increasing the speed of operation. Core memories provided a faster, more reliable storage medium for the main memories needed in the new systems. New methods were used to identify the location of operands used in the transactions in the machine. Floating point arithmetic was introduced to remove from the computer user the burden of scaling all of the data and arithmetic to fit the available operations. This period also saw the start of computer languages, such as Fortran and Cobol, which allowed the users of the machines to create programs without knowing all of the details of the internal operations of the machines. Independent input/output processors removed the time-consuming transfer of information to and from the system from the CPU (central processing unit) itself; this allowed the CPU to spend its time doing useful work. These systems also provided the user with some system software: batch processing facilities, libraries, and compilers.

The influence of the first generation on the machines of the second generation is evident by comparing block diagrams of machines from each era. Figure 1.5 gives a block diagram representation of the IBM 7094. The IBM 709X series of machines were 36-bit systems, with the data formats shown in Figure 1.6. The system is a single address machine, but with this system the address register can specify information based not only on the contents of the program counter, but also on the content of index registers and combinations of registers. This additional capability adds to the flexibility of the system, but it also requires additional complexity in the control unit and in the specification method. The instruction format has expanded to fill the entire 36-bit word; the operation code must not only identify the desired function, but also the manner in which the address must be treated to identify the location of the operand. This requires several bits of the operation code to identify one of the eight index registers, and the manner in which the address is to be constructed. In addition to the instruction format, Figure 1.6 identifies two different data formats, one for integer data, and one for

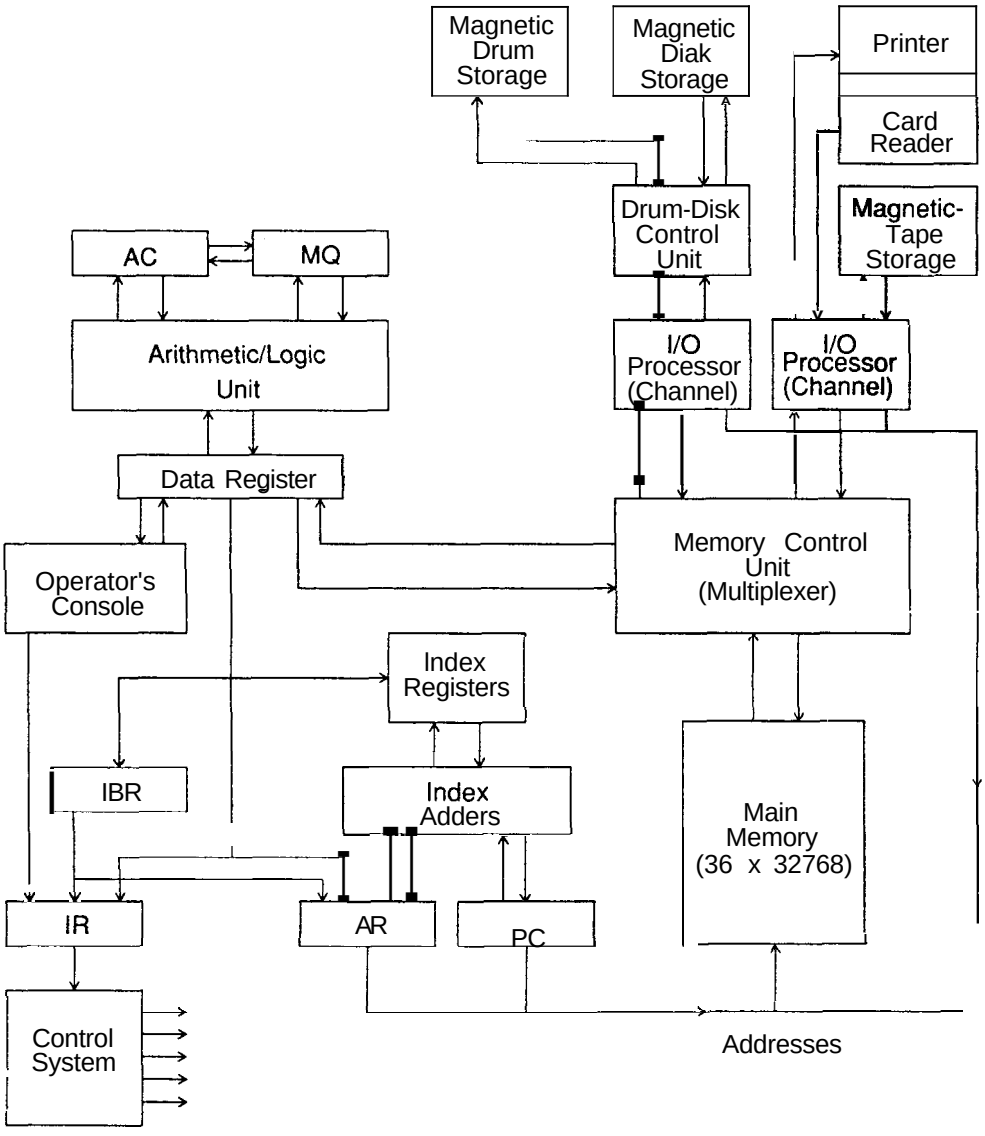


Figure 1.5. Block Diagram Representation of the IBM 7094.

floating point data. The integer capability allows for "normal" incremental functions, and the floating point capability provides for a combination of large and small values to be used with minimal user worry. Also, a double precision floating point format provides for a double length mantissa.

The IBR in this system is an instruction backup register, but it serves the same purpose as the IBR in the IAS. However, in this case, the provision arises not because two instructions can fit into the characteristic word length of the system, but rather because the data path from main memory to the central processing unit is 72 bits wide, and so two transfers can be effected simultaneously. The AR and the PC specify the address and the location in memory where the program is executing. The AC and MQ registers provide a similar function to the corresponding registers in the IAS system.

Another difference in the system capabilities is evident by examining the peripheral devices and their communication paths to the system. The I/O processors, called channels, have the responsibility of coordinating the transfer of

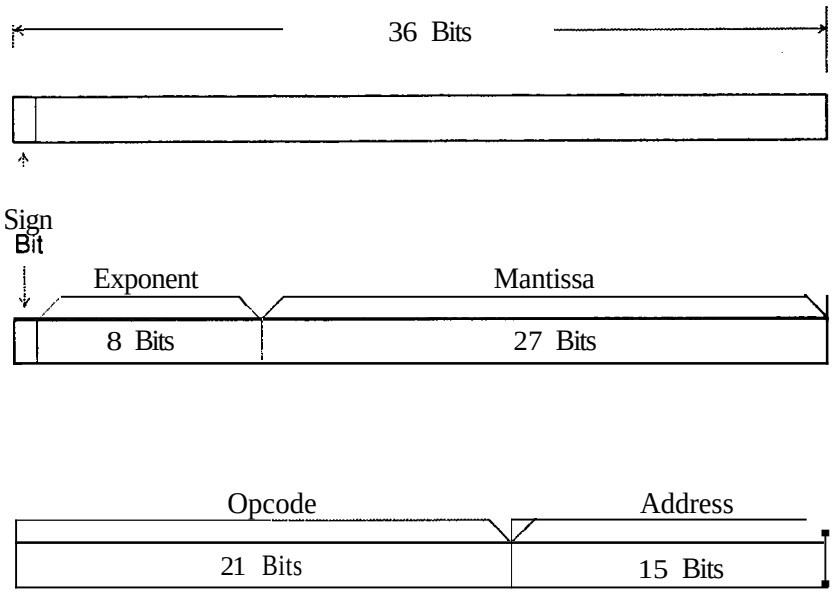


Figure 1.6. Data and Instruction Formats for IBM 709X.

information to and from the mass storage devices and the input/output devices. These transfers are initiated by action of the CPU, but are carried out by the channels. When the transfer has been completed, the channel has the capability to interrupt the action of the CPU, which will indicate the completion of the specified action.

The more elaborate I/O system also added to the flexibility and efficiency of the system. In earlier systems, the operation of the device was strictly single user — whoever had physical access to the machine controlled the operation of the system. The system then was dedicated to the task of one user, until that user finished and relinquished the machine. This resulted in a large amount of dead time when the system was idle. With the second generation systems, programs were collected together into a "batch," and then fed one at a time into the memory and executed. This increased the apparent speed of the machine, because it minimized the time the central processing system was idle.

The creation of the transistor and related technology provided higher performance in a much smaller package. The availability of these devices prompted efforts to create machines capable of much higher execution rates than the "normal" computer systems. These machines are called supercomputers, and involve a variety of techniques to improve the processing speed. One of the first efforts was the LARC system (Livermore Atomic Research Computer), made by UNIVAC. Another early system was the IBM 7030, also called the Stretch. These systems pushed the technology to create faster systems, and also explored the use of parallelism to increase system speed.

The parallelism at this stage took two basic forms: overlapped instruction execution and the use of parallel processing elements. Overlapping the fetch and execute portions of the computational process resulted in an apparent speed increase by doing more than one thing at at time. While one instruction was being executed, the next instruction was being fetched from memory. Higher degrees of overlap could be achieved by dividing the processing into even smaller pieces, The use of multiple processors allowed one program to execute on one

processor while another executed on a second processor. The benefit came when the systems resources (memory, I/O processors, disks, etc.) were more fully utilized because of the increased processing. These early attempts at supercomputing led the way to further advances in the next set of computers.

1.2.3 The third generation (1962–1975)

Early in the 1960s the promise of semiconductor technology began to make itself felt. Integrated circuits, which combined many transistors in a single chip, reduced the size and cost of computer systems. Not only did the integrated circuits have a great impact on the logic, but semiconductor memories became a significant factor in the creation of computer systems. These memories would eventually replace core memories as the primary memory element in a computer. The high speed memories provided the needed technology to implement a technique known as microprogramming. This technique had been proposed by Maurice Wilkes in England as early as 1951, but the technology was not available to effectively utilize it. However, with memory speeds an order of magnitude faster than the main memory speed, microprogramming was widely used. The regularity involved in microprogramming allowed the complexity of the instruction sets increase, without an undue increase in the complexity of the control system needed to assert the control signals of the system.

The utilization of a CPU was raised above that of the second generation systems by means of multiprogramming, in which the system resources are shared among several programs on "time-shared" basis. This resulted in part from the advances made in'the set of programs that controlled the operation of the system, which have become known as the operating system (OS). Operating systems continue to provide additional capabilities, such as improved compilers, shared libraries, utilities, and accounting information.

The concurrent use of hardware segments in parallel or pipelined processing was utilized in a variety of ways. Again, the objective of the mechanism was to increase the apparent speed of the system. Along with the increased processing speeds, a new technique for numerical programming was introduced, which is called vector processing. A vector processor seeks to enhance system speed by organizing the information into uniform sets called vectors, and applying the same operation to all of the elements of the vector. This saves time because a single instruction is used to specify many operations, and because the hardware can be organized to take advantage of the one-after-another nature of the operands being used in the vector operations.

One of the most prolific systems of this period was the IBM 360. This system is interesting from a number of aspects, one of which is the manner in which it came into existence. In 1964 an article in the IBM *Journal* [AmBl64] described the various members of the 360 family. To this point in time, and in many cases after, members of a "family" of computers came into existence as newer technology made speed improvements possible, and as a customer base made investment in the new machines reasonable. However, with the announcement of the 360 series of computers, the various members of the family were identified, and their specific characteristics enumerated. Thus, if a customer wanted one level of performance, one machine was purchased; a different level of performance, either higher or lower, dictated a different choice. However, from the view of an assembly language programmer, the systems were identical. This initiated into the computer jargon the phrase, "instruction set architecture," which refers to the

appearance of the machine, not as defined by the functional units and interconnections of the hardware, but rather the apparent functional units and interconnections that are activated or controlled by the instruction set. This leaves the hardware designer free to create a system that uses whatever techniques are economically or technically justifiable to create a system with a certain set of characteristics.

From the creation of the notion of an instruction set architecture, there have been two uses of the term "architecture": one is to describe the actual functional elements of a system and their interconnections; the other is to describe the apparent structure of the machine as defined by the instruction set. In this text we will be examine the instructions used to define a system, but also the interconnection methods and some of their implications.

A system level block diagram of a typical IBM 360 installation is shown in Figure 1.7. The system is controlled by the CPU, which not only manipulates the data in the main memory, but also controls the action of the I/O channels. The channels transfer information directly to and from the memory, and the CPU is informed of the completion of the specified operation by use of an interrupt. Two different types of channels are indicated, one for high speed operation (selector channel), and one for low to medium speed operation (multiplexer channel). The channels handle device specific I/O requirements, and communicate with each device over a common set of interface lines.

The instruction set architecture of an IBM 360 system is shown in Figure 1.8. The fundamental size of the system is 32 bits, which was chosen for a number of pragmatic reasons. This width is a multiple of 4 bits, which is the size of the representation of BCD (binary coded decimal) digits; instructions are

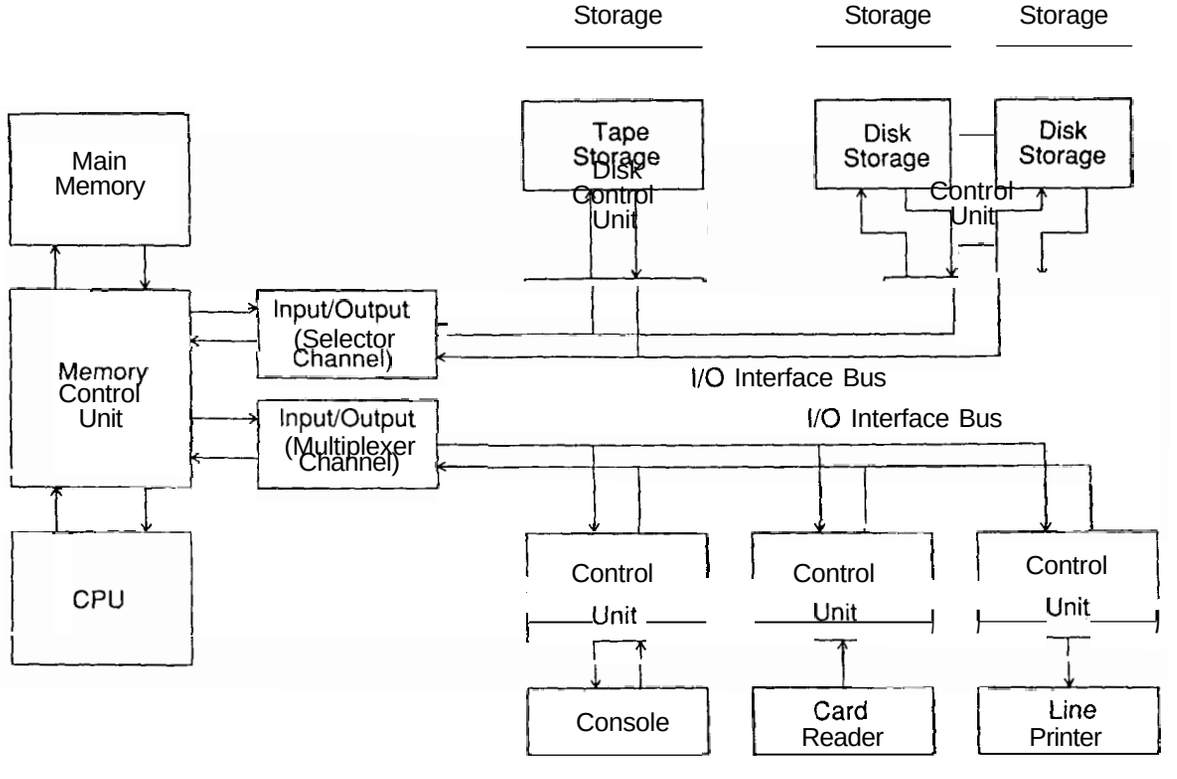


Figure 1.7. Block Diagram Representation of an IBM 360 Installation.

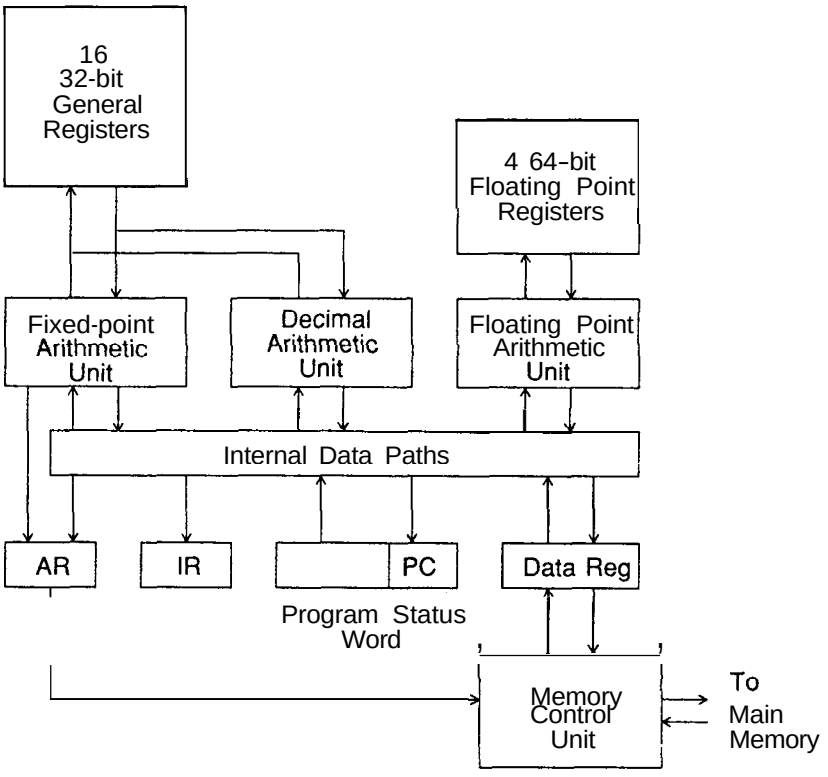


Figure 1.8. Instruction Set Architecture of an IBM 360 CPU.

available to operate on BCD digits. It is also a multiple of eight bits, which is used for character storage. Integers can be represented with either one or two words, for single or double precision information. Floating point numbers can be represented with one, two, or four words, for a variety of numerical capabilities. Finally, instructions are composed of two, four, or six bytes, and will fit easily into a 32-bit format.

The instruction set processor of an IBM 360 identifies 16 registers used for storage of general purpose values. These can hold data to be used in normal calculations, or they can be used to hold addresses that identify the location of operands in main memory. In addition to the general registers, there are four 64-bit floating point registers. These are used to hold operands in the execution of floating point operations. The AR is used to identify the location of information in main memory, which is accessed via the memory control unit. The IR holds the instruction being executed, and the PC is used to identify locations in the execution of the program. The PC is considered part of a set of information known as the program status word (PSW). The PSW contains not only the PC, but also information about the current status of the system, such as whether the last operation resulted in a positive or negative number, and so on. Also shown in the figure are the apparent communication paths connecting the various units. These connections may or may not be extant, which also applies to the functional units. In smaller systems the functional units appear to be there, because the instructions function as expected. However, the actions may be orchestrated by a microprogrammed control system which utilizes a single arithmetic/logic unit to accomplish all of the various capabilities of the system. However, on the higher end models the functional units are there as expected, providing additional speed to the system.

Along with the advances in hardware, operating system advances continued to modify the manner in which users interacted with the computer systems. During this time, concepts were developed that led to the implementation of virtual memory systems. These systems present a uniform view to user programs of the available system memory, and then map address requests in the user space (virtual addresses) into the actual location of the information (real addresses). The operating system also presented to users an easily manageable access to the file system for program and data storage, to the system utilities (editors, file manipulation, etc.), and to the workhorse programs (compilers, etc.) As the systems became more readily controlled and managed, the application areas in which computers were utilized expanded.

As a result of the use of semiconductor technology to reduce the space and power requirements required by computer logic, small computers could be created with very useful capabilities. The result was the minicomputer, which provided computational and control capabilities for a variety of applications. These systems utilized the same concepts as the larger machines, but worked on smaller quantities of information, such as 12- or 16-bit words. This proliferation of machines made the capabilities of computing systems available to a larger community of users. The computer lost some of its mystery, and became an inexpensive, useful tool in the solution of a large variety of problems in science, industry. business, and education.

1.2.4 Additional generations of computers (1974–?)

Each advance in technology brings with it computer systems with more capabilities. As the amount of logic and memory contained in a single integrated circuit continued to increase, a new generation of computers emerged. This is sometimes called the fourth generation of computers. These computers utilize semiconductor devices almost exclusively for main memory as well as the active logic required in the CPU and I/O controllers. The use of virtual memory systems has become a standard feature of systems, in both large and small computers. The proliferation of active devices and semiconductor memories has allowed the virtual memory techniques to be applied to high speed memories, called cache memories. These units provide a relatively small high speed memory between the CPU and main memory. The net result is an increase in the apparent system speed, since the amount of time that a processor is idle decreases.

As the relative cost of the hardware portion of a system continues to decrease, techniques that were at one time reserved for high performance machines are routinely applied to smaller systems. The complexity available in integrated circuits has allowed creating entire systems on a single chip. These processors have progressed from the 4- or 8-bit processing elements, which were available in the mid-1970s, to systems with complete 32-bit computer systems on a single chip, such as the Motorola 68000 series of processors, which form a popular 32-bit processor system. Contained within the chip are not only the registers and arithmetic units of a complex processing element, but also the virtual memory facility and the interface to the physical system memory. Along with the immense amount of processing capability, memory technology also provides storage capacities unavailable in previous systems. In the late 1980s, a single chip can now provide up to 4 megabits of storage capability, and 16-megabit memories are in the experimental stage.

The availability of low cost, high performance processors and memory devices resulted in personal computers for the home or office, in workstation computers for engineering and scientific use, and in multiprocessing systems useful in many areas of computing. With the proliferation of computers have come different ways to provide communication between processors, such as token ring networks, collision detection, common wire networks, broadcast networks, and direct connection networks. Fiber optics have opened new dimensions in speeds and capabilities, and other communication mechanisms continue to evolve. Each of these advances opens new doors to computer architects, and application of the new technologies will result in even more exciting systems.

Another generation of computers, sometimes called the fifth generation, is the target of different research and development efforts. This generation of machines is not identified by the technology used to implement it, but rather by the capabilities of the machine. The target is to create a system that is oriented to human interaction, so that minimal specific knowledge on the part of the user is, necessary to make use of the system. This "user friendly" type of a system will result in improved abilities to use computers to meet needs in all endeavors which can make use of computers. In addition, this new generation will be capable of handling immense amounts of data. This will facilitate not only the traditional mathematically intense programming, but also areas such as artificial intelligence and natural language translation.

As advances in technology continue to give us new capabilities and improved tools to work with, our challenge is to create systems that will not only solve an immediate need, but which will also be capable of growth to solve future needs.

1.3. Computer Design and Architecture: The Organization of This Book

The preceeding brief history highlights the fact that the various architectures and design techniques used in system implementation change and grow as more experience is gained and as the available technology changes. A number of different design issues are involved in any design, and what is a good design or a good architecture will not be the same from one implementation to another. How "good" a design or an architecture is depends on how well it matches the goals of the system, and the intended application area of the system will have a great impact on the characteristics of the unit. To compare one technique against another requires that a metric or a set of metrics be chosen, and that the relative performance of the systems be compared using the chosen metrics as a measure of how "good" the system is. The number and type of metrics chosen will directly impact the comparison method and results of the evaluation. We will not try to establish a specific set of metrics; rather, we will present different rnetrics throughout the text to provide a basis for evaluation. The task of a system architect is to select the set of metrics that most closely reflects the goals for the system.

An example of the evaluation mechanism and the metrics involved is available with the numerical calculation of the Fourier transform. The Fourier transform can be used as a tool in a number of areas of research, and it has provided a wealth of knowledge in a variety of fields. The transform is simply defined, and utilizes a set of input values to create a set of output values. The

complexity arises because each of the input values must interact with all other input values, which results in a computation requiring on the order of N^2 multiplications, which we represent as $O(N^2)$. In the mid-1960s a number of researchers independently developed an algorithm, or a set of algorithms, known as the fast fourier transform (FFT). The FFT follows a carefully planned pattern to allow the interaction between inputs required by the Fourier transform, but the operations are done in such a way that some values are used a number of times. The net result is a reduction in the number of multiplications required, from an algorithm that requires $O(N^2)$ multiplies, to an algorithm which requires only $O(N \times \log_2 N)$ multiplies. This greatly increased the size of transforms that could be economically produced, since it reduced the amount of computer time required for the calculation of a transform. The choice of the number of multiplies as the metric was reasonable in 1965, since the multiply was the most time-consuming portion of the algorithm. However, in today's technology, the multiply can be done very rapidly with inexpensive hardware, so other metrics could be more useful and result in algorithms that exhibit better performance than the straightforward FFT algorithm. For example, instead of using a base 2 radix algorithm, a base 4 radix algorithm could be used. This would actually result in more multiplies, but fewer data transfers would be required to complete the calculation. By using both the multiply time and the data transfer time, and the number of each required, better choices can be made for the architecture of an algorithm or a computer system created to do FFTs. A clearer picture of the overall response of the system will be obtained as more metrics are included in the evaluation.

The design process involves application of basic principles to solve engineering problems. In the process of learning the basic principles of computer design and the application of those principles to solve the problems posed by computer systems, it is not enough to simply explain the principle and to assume that the application of the principle will automatically follow. For that reason, this text makes extensive use of examples to illustrate the principles being discussed. It is hoped that the examples chosen and the design methods used will illustrate not only the principle, but the application of that principle as well. Thus, not only do the examples include block diagram representations of a solution, but the example will also, where appropriate, carry the application of the principle to a hardware level, so that a real implementation is presented. In some cases, the hardware included in the example is extensive, and the actual schematics and further explanations are included in Appendix B.

The design examples follow the ideas detailed by Fletcher [Flet80] in his design text. Uniform application of these ideas will result in designs that are not only functionally correct, but are also easily understood and debugged. Some of the specific ideas we will mention here include the consistent use of:

- The shape of gates to identify function.

  The use of polarized mnemonics.
- The use of logical state indicators.
- The use of incompatibility triangles.

All of these ideas are related to one another, and all deal with the issue of communication of ideas.

In the creation of a design to perform a specific function, a designer can implement a system that performs the desired work and matches all system

criteria. But if the designer is unable to communicate these ideas and this design to anyone else, then no benefit is derived. One of the greatest challenges encountered by technically oriented people is to communicate their ideas not only to their colleagues, but also, perhaps more importantly, to the people who would benefit from application of those ideas. By following a consistent method for providing drawings and presenting designs, the ideas and concepts can be more readily understood.

Random logic is used extensively in computer design, not only to create the functional units of a system, such as adders and registers, but also to test conditions and create control signals. The logic provides an active function, and is not simply created to check for "true" or "false" conditions. This active nature of logic is reflected in the creation of the designs used in this book. We do not refer to signals as "true" or "false," but rather, a signal is either **ASSERTED** or **UNASSERTED. A** signal will be **ASSERTED** when the conditions required for the appropriate action are satisfied. If these conditions are not satisfied, then the signal will be **UNASSERTED.** In order to identify the action of the signal, polarized mnemonics will be used; that is, the names used to identify a signal will also identify its function. For example, a signal used to enable a data value onto a bus might be called **DATA–ENABLE.** If a name is too long, it can be shortened in a manner that will maintain the information. For example, the previous name could be shortened to D_ENBL. The other information that needs to be added to the name is an indication of the binary level at which the signal will be **ASSERTED.** This is done by appending to the end of the name a polarization indicator. If the signal is **ASSERTED** when the voltage of the line is high, then a hyphen and the letter H is added to the name. If the signal is **ASSERTED** when the voltage of the line is low, then a hyphen and the letter L is added to the name. If the data enable line is asserted low, then the name could be D_ENBL-L. The consistent use of polarized mnemonics will provide a basis for better understanding of the design and for better communication. It also will help the designer when, six months later, the design is revisited.

When representing the gates used to perform the logic work in a digital system, the shape of the gate should indicate the function being performed. The three basic shapes are the **AND** shape, the **OR** shape, and the **BUFFER** shape:

AND shape

OR **shape**

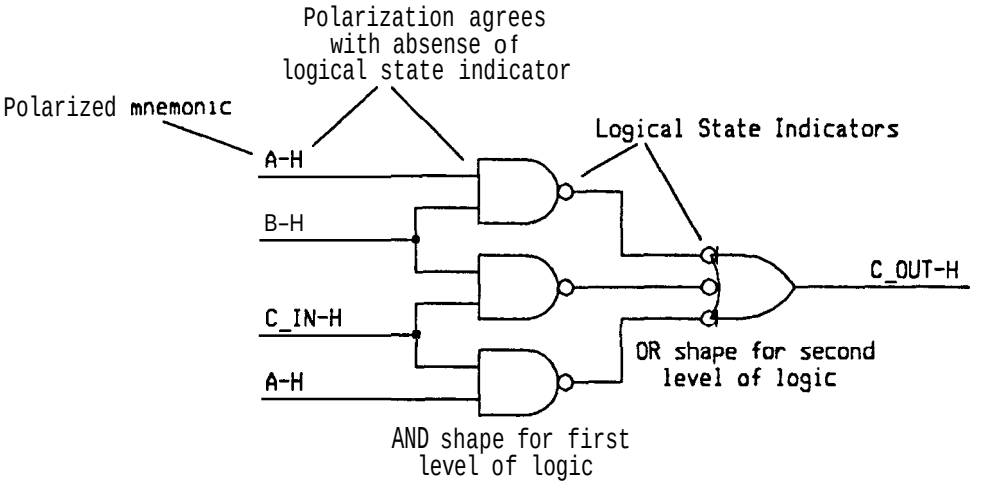
BUFFER shape

Like the names of signal lines, the inputs and outputs of a logic gate should identify the assertion levels involved. This is done by the use of logical state indicators, which are the "bubbles" appearing on some of the leads of a gate or a logical block. If a logical state indicator is present, then that line is asserted low. If a logical state indicator is not present, then that line is asserted high. For example, consider the equation used to implement the carry of a full adder:

$$C_{OUT} = A \cdot B + B \cdot C_{IN} + A \cdot C_{IN}$$

We know from basic digital derign implementations that this can be implemented with two levels of **NAND** gates, one for the ANDing function and one for the

ORing function. However, when the shape of the gate, the use of polarized mnemonics, and the use of logical state indicators all come into play, then the gating will appear like:



The various configurations for the basic gates is shown in Figure 1.9. Each can be used in either an ANDing or an ORing function, and the buffer can provide level conversion or not, as needed by the logic.

As can be seen from the gating for the carry shown above, the normal situation is for an output with a logical state indicator to supply a signal to an input

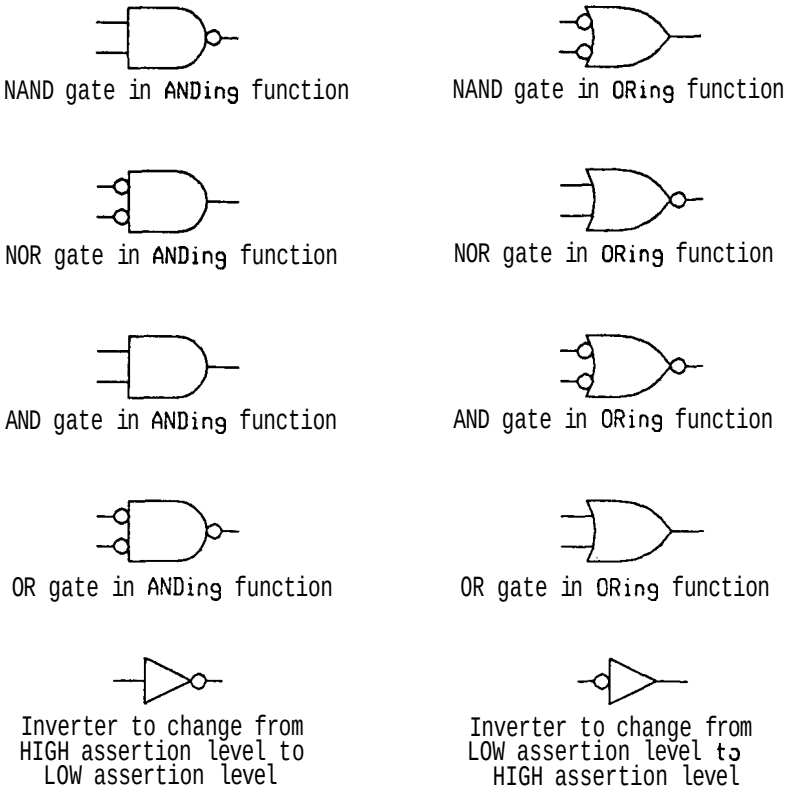


Figure 1.9. Basic Gates in and Their DeMorgan Representations.

with a logical state indicator. Also, an output with no logical state indicator will normally drive inputs that do not have a logical state indicator. However, sometimes this does not happen, and this condition is called an incompatibility. There are two times when this incompatibility will occur: at the input of an ANDing function and at the input of an ORing function. Consider the two functions below:

The first function shows an incompatible signal at the input of an OR function. When the signal (ENBL-H) is asserted, then the output (OUT-H) will be the logical OR of the remaining inputs. When the signal is not asserted, then OUT-H will be asserted, regardless of the level of the other inputs. Thus, an incompatibility at the input of an ORing function provides an enabling action: when the signal is asserted, the OR function is enabled; when the signal is not asserted, the OR function is disabled. The second function shows an incompatible signal at the input of an AND function. This time the incompatibility provides a disabling function. When the signal is asserted, the AND function is disabled, and the output will not be asserted, regardless of the level of the other inputs. When either of these cases arise, the fact that the designer created the incompatibility on purpose is indicated by the use of the small incompatibility triangle at the input. This is a signal from the creator of the design to anyone looking at the circuit that a high asserted signal feeding a low 'asserted input, or a low asserted signal that provides input to a high asserted signal, is not an oversight or a mistake, but rather a result of the design process.

These simple procedures will lead to systems that are easily understood and implemented. As an example, consider the task of creating a gating circuit to detect the address 776000_8 on an address bus that consists of 18 lines. One logic circuit to do this is shown in Figure 1.10. The drawings used in this book make extensive use of buses, which are merely a collection of wires. The name of the bus should follow the same polarized mnemonic convention mentioned above. The width of the bus and the range of the elements contained in it are identified by the use of the pair of numbers in parentheses: the ADDRESS(17:0)-H nomenclature identifies that the address lines range from ADDRESS(17)-H to ADDRESS(0)-H. The H on the end identifies the fact that the assertion level is high for this bus. Note also that the elements that are split off from the bus will identify which wire of the bus is involved. The gates used to detect the address include two NAND gates used in an ANDing function, and two OR gates, also used in an ANDing function. The incompatibility triangles identify the fact that the high asserted bus is knowingly directed to the low asserted input. The AND shape of the lower gate indicates that the output will be asserted (high) if ADDRESS(0)-H is not asserted, AND ADDRESS(1)-H is not asserted, AND ADDRESS(2)-H is not asserted, AND ADDRESS(3)-H is not asserted, AND ADDRESS(4)-H is not asserted. The shape of all of the gates in this book can be similarly interpreted to identify the function performed.

As mentioned above, information transfer within a computer system is often performed on a collection of wires called a bus. To provide an additional piece of information about the transfer of information on the bus, the direction of information flow will sometimes be indicated with a small arrow, as shown with the

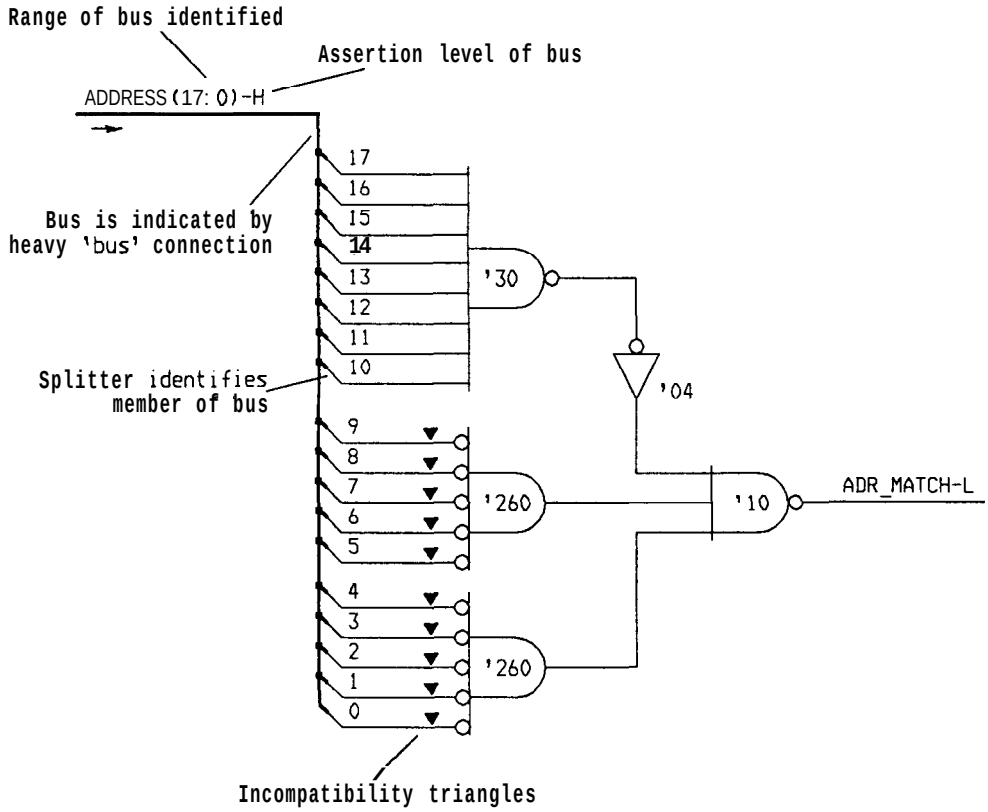


Figure 1.10. Implementation of an Address Decode Logic Circuit.

address bus above. If information can be transferred in both directions along a bus, this is indicated with a double ended arrow.

Since buses will be very important in the structure of systems, and in the logic used to implement examples in the text, a brief word about their function is in order. Buses can be created with a number of techniques, but we will mention only two, both diagramatically shown in Figure 1.11.

The first technique shown in the figure is the open collector method, in which the collector of the final transistor of a gate (or a functional block, such as a register) is not connected internally to any element. The effect of this mechanism is that, when the transistor is tumed "on," the output will sink current, and the output voltage will go to a small value, usually near 0.4V. When the output transistor is not turned "on," no current is requested from the output, and the output voltage level is not influenced by that transistor/gate. Thus, an external pullup of some kind must be provided, so that the output voltage will go high when there is no transistor pulling it down. This mechanism will allow multiple outputs to be connected together; the level of this common node will be allowed to go high only if all output transistors are tumed "off." This ability can be used effectively in a number of circumstances, as we shall see. A gate is often identified as an open collector gate by the presence of a mark at the output of the gate, as shown in Figure 1.11.

The other prevalent mechanism used for busing is also shown in Figure 1.11. This is called a tri-state capability, so named because the output can assume one of three states. Two of the states are the "normal" states of a TTL gate: low

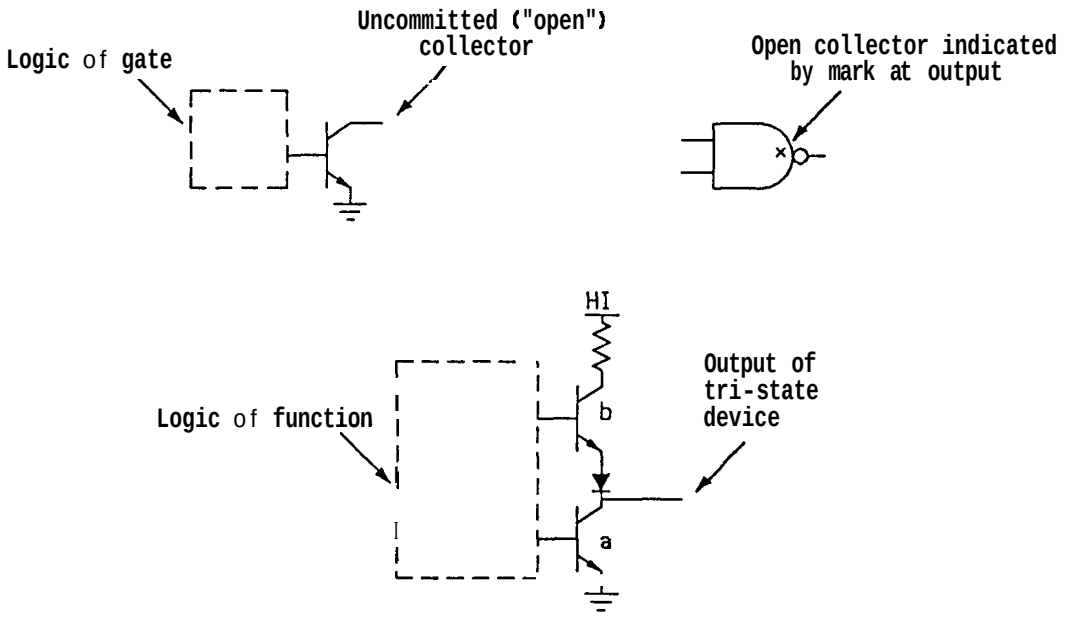


Figure 1.11. Busing Configurations: Open Collector and Tri-state.

and high. The output will be low when the logic of the function creates a situation in which transistor "a" is turned "on." and transistor "b" is turned "off'; the output will be high when transistor "a" is turned "off' and transistor "b" is turned "on." The third state occurs when the logic of the function creates a situation where both transistors "a" and "b" are turned off. In this case, the output is electrically disconnected from the system, since the paths through transistor "a" and through transistor "b" present an extremely high impedance. This third, high impedance state is usually created by an enable (or disable) input to the function.

The enable line used to control the tri-state capabilities of a gate of function can be included with a simple gate or with more complex functions. Figure 1.12 includes several examples of tri-state functions. Figure 1.12(a) shows a buffer shape (from the basic shapes shown above), which also has an enable line. From the presence of the logical state indicator on the enable line, it is evident that the buffer function will occur when the enable line is asserted low; if the enable line is high, then the buffer is electrically disconnected from the wire connected to the output. The buffer shown in Figure 1.12(b) operates in exactly the same fashion, except that the assertion level for the enable line is high. To require an enable line for each output would be excessive, so often a single enable line is used for an entire IC. Such is the situation depicted in Figure 1.12(c), which is often called a tri-state driver: eight buffers are packaged in a single IC, and the enable lines (in this case, asserted low enable lines) are connected together and made available on a single pin. Finally, since it is possible to transfer information in both directions through a tri-state port, Figure 1.12(d) shows a device configured to provide this capability. The device is a transceiver, and both an enable input and a direction input are used. Internal to the transceiver, two tri-state buffers are used for each line, one in either direction (A to B, or B to A). When the enable line is not asserted, the transceiver is electrically disconnected from both the A lines and the B lines. However, when the enable line is asserted, the direction line will determine which set of buffers is enabled.

Before continuing, we will mention one final set of devices to be used again and again in the text. These are the register and the latch:

Like the buffer of Figure 1.12(c), the register contains several elements ganged together for a common function. In this case, the joined elements are flip-flops, and in this configuration they are capable of storing a byte, or 8 bits. The clock lead on the register is marked with a special symbol that indicates that the function (in this case, a storage function) occurs on the rising edge of the clock. That is, when a low-to-high transition occurs on the clock line, information available at the input is transferred to the output. The next time at which a

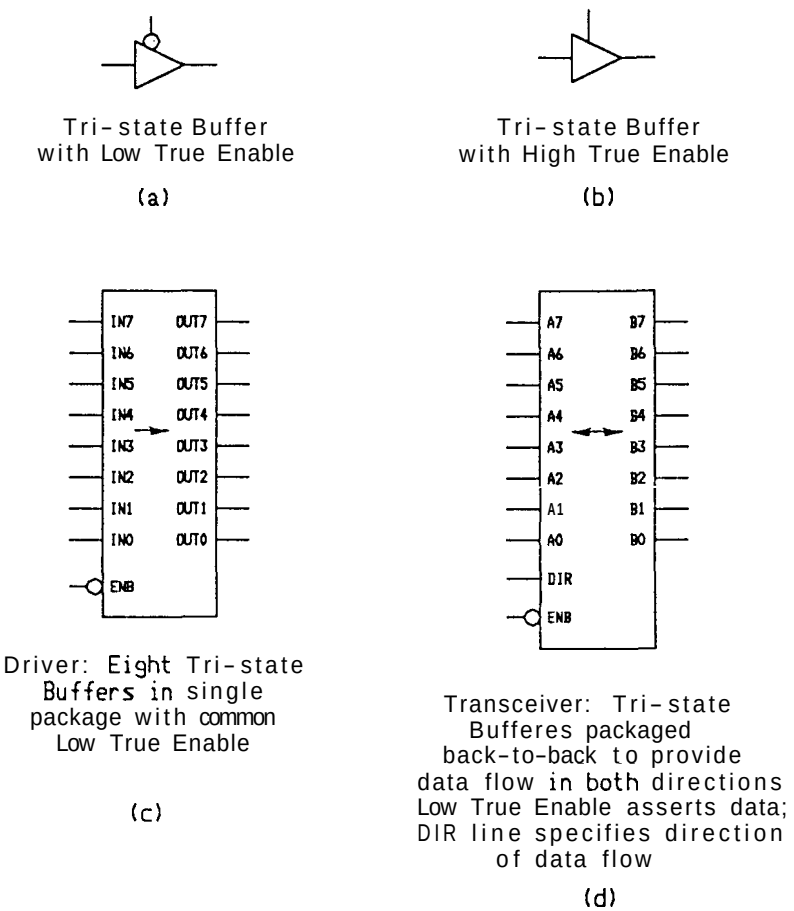


Figure 1.12. Tri-state Buffer Configurations.

data input can have an effect on its corresponding output is when the clock again makes a low-to-high transition.

The latch function. as shown above, can also store a byte. Note, however, that there is no dynamic indication on the enable line of the latch. The latch function is described as follows: as long as the enable line is **NOT ASSERTED** (a low voltage level for this example), the outputs will be maintained at the value previously entered. So long as the enable line is **ASSERTED** (a high voltage level for this example), the output will follow the input, until the enable line is **DEASSERTED,** at which point the information will be stored in the latch. Thus, a register, as used in this text, is an edge-triggered function, while the latch is a level-sensitive function. (Note that latches can be created in other configurations than 8 bits, such as a single bit latch.)

Registers and tri-state devices can be packaged together, which can save board space and power. Registers, tri-state drivers, buffers, and transceivers will be used extensively in the examples in this text. The elements will be identified by their TTL device numbers, but the same functions are often available in other technologies as well. The open collector and tri-state capabilities can be provided with **CMOS** and other technologies, and the functions shown in the examples are often available in device libraries for designing functions on a single IC chip.

From the days of Pascal to the present, computers have been used to work with information, to do arithmetic, and to manipulate a variety of quantities. Some of the machines have implemented schemes to do the work with decimal quantities, but most of them use binary systems to represent the information. In Chapter 2 we look at the representation of information, as well as mechanisms that can be utilized to present different kinds of information. Not all number schemes are created equal, and different methods will lead to different characteristics in the ability of a system to represent values and in the amount of logic required to work in a number system. Once a designer or a system architect knows the limits to the various representations and schemes, then reasonable engineering choices can be made about the use of number systems.

The techniques used to manipulate data is the subject of Chapter 3. Here we examine techniques for doing not only addition and subtraction, but multiplication and division as well. All of the basic operations can be implemented in more than one way, and we will examine some of the assorted implementations. Each method will present a different set of characteristics, and so the choice of a metric will determine which type of algorithm or structure will be best suited for an application.

Armed with the number systems and the arithmetic methods, we are ready to consider the mechanisms used to specify work to be done by a computer. Thus, Chapter 4 looks at instruction specification: what is both useful and necessary at the instruction set level of a computer. Here we also introduce a register transfer language (RTL), which can be used to specify the elementary operations used to accomplish the work of the instruction. Again, the use of metrics, such as the time and number of steps required, provides a basis on which to compare different ideas or implementation techniques. One of the instruction set ideas to be examined is the ongoing debate on the complexity of the instruction set.

The control section of the computer coordinates the action of the various modules making up the machine, the buses, arithmetic units, registers, and so on. The action of the instruction set and the basic register transfers required to implement it are discussed in Chapter 4, and the creation of a control system to implement the instruction set is discussed in Chapter 5. Here we look at different

techniques that can be utilized to assert the control signals required to move data throughout the system. Again, these techniques are represented as both principles and specific examples.

The subject of Chapter 6 is the input/output process — the techniques and methods used to move information to and from the computer. From the earliest days of computing machinery, this has been a requirement: how does the information get transferred to the machine to start a computation, and what is required to get the information out of the system? A number of methods are available, from the simple programmed I/O to the more complex direct memory access. The methods and their relative merits are presented, along with a discussion of different bus techniques utilized in the I/O process.

The storage and retrieval of the information required by a computer system is discussed in Chapter 7. As the technology of memories has changed over the years, the implementations have also changed. Memories have become larger, and this trend will continue for the foreseeable future. In order to inaintain all of the information needed in a system, a hierarchical memory system is utilized. At one end of this system are the registers and cache memory systems needed to keep pace with a high speed processor. At the other end of the system are the slow speed devices, such as tape systems, where immense amounts of information are maintained. In Chapter 7 we will discuss not only the implementation of memories, but also some of the methods used in this hierarchy of storage devices, such as virtual memory techniques and cache memory systems.

The desire for higher performance always pushes designers and system architects to select approaches that most effectively utilize the overall system resources. One mechanism for obtaining higher performance is to utilize concurrent events. That is, if two things can happen simultaneously, then the overall system speed should increase. One method of doing this would be to have parallel computer systems, which greatly increases the complexity of the hardware and software problems needed to utilize a computer system. Another method to achieve some of the benefits of parallel processing is to create a pipeline of events. That is, the processing required by a system is divided into sections, and independent operations are carried out in hardware created to perform each of the sections. This technique is called pipelining, and in Chapter 8 we will examine some of the issues involved. Pipelining will result in performance benefits, but only if we are successful in keeping the pipe busy.

The following chapters provide some insights into the design process, along with some techniques for comparing the benefits of one technique with another. These techniques and ideas will be valid only when fully understood and applied in reasonable ways.

1.4. References and Readings

[AmBl64] Amdahl, G. M., G. A. Blaauw, and F. P. Brooks, Jr., "Architecture of the IBM System/360," IBM *Journal of Research and Development*. Vol. 8, No. 2, April 1964, pp. 87–101.

[Baer84] Baer, J. L., "Computer Architecture," *Computer*. Vol. 17, No. 10, October 1984, pp. 77–87.

[Baer80] Baer, J. L., *Computer Systems Architecture*. Rockville, *MD:* Computer Science Press. 1980.