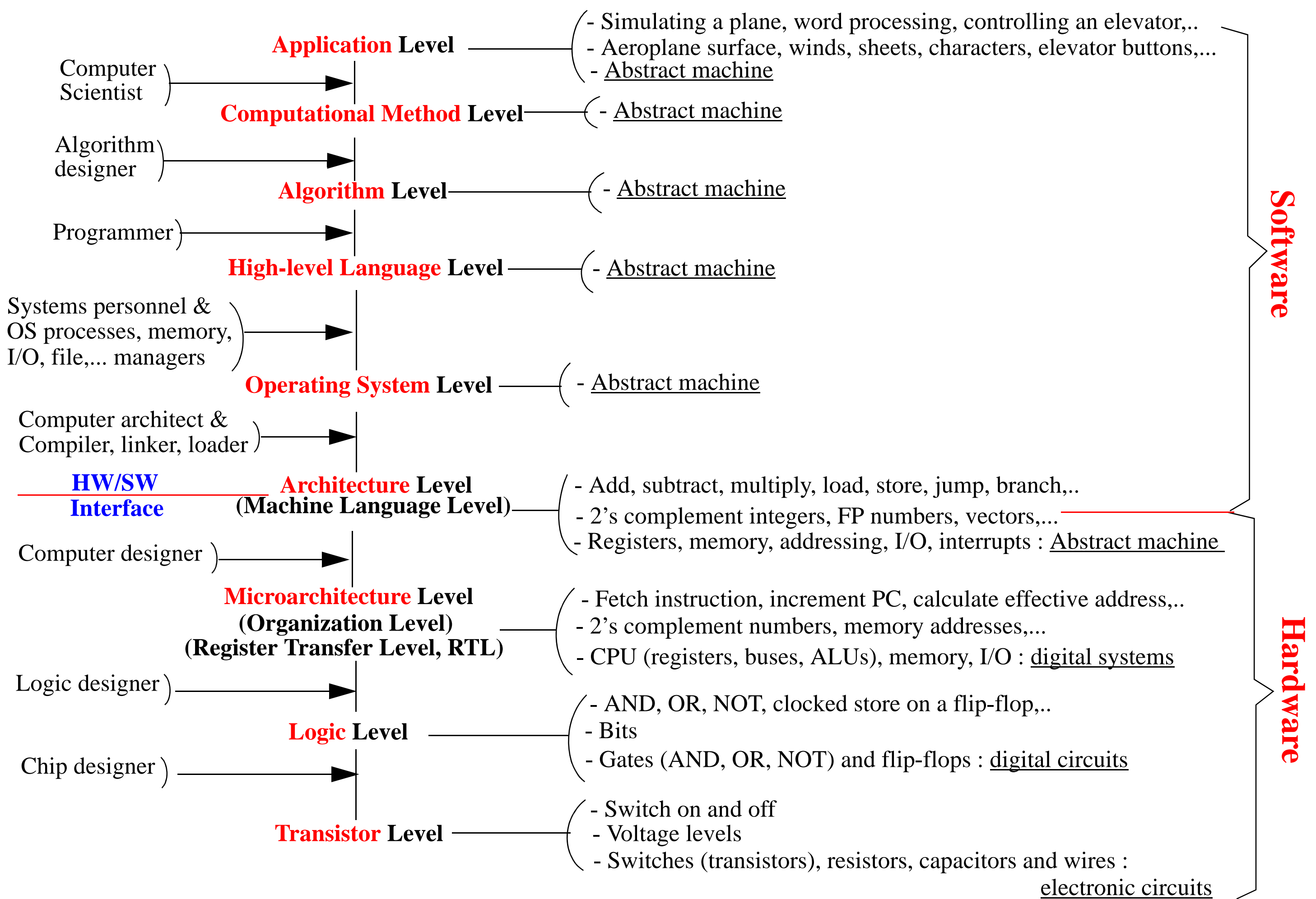
CS2214 COMPUTER ARCHITECTURE & ORGANIZATION SPRING 2012

LAYERED COMPUTER DESIGN

1. Introduction

CS2214 focuses on computer design. It uses the top-down, layered, approach to design and also to improve computers. A computer is designed layer by layer, starting with the top conceptual layer and ending with the bottom layer, the complete design. A layer is implemented by the layer below. There are many possibilities to implement a layer. To decide, **design goals** are used : speed, cost, size, weight, power consumption, reliability, expendability, flexibility and compatibility. CS2214 concentrates on the speed (performance) goal and describes how it can be used to make decisions on the design of a computer named EMY. It will be concluded often that the best (most reliable) computer speed measure is the run (execution) time.

Applications to Transistors



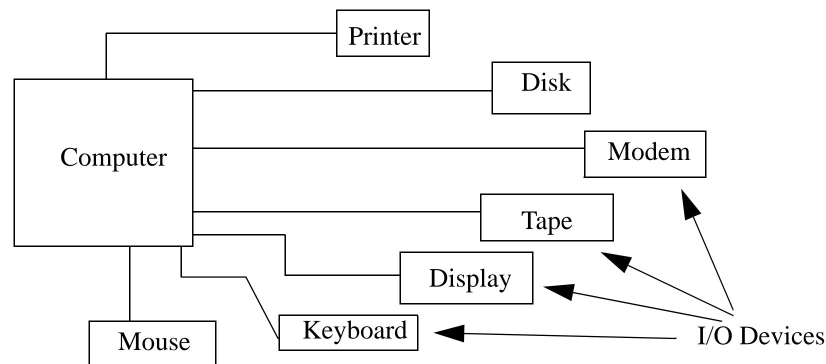
Computer architecture courses cover application, architecture, organization, logic and transistor layers. But, four other layers between the application and architecture layers need to be studied as well for a comprehensive design (see the figure above). To save time, CS2214 will also not consider these four layers much. In the figure above, who/what implements a layer is indicated on the left side. Some layers are shown with three text lines on the right side. The first text line indicates typical operations of the layer, the second text line indicates typical operands of the layer and the third text line indicates typical components of the layer. Finally, it must be noted that ultimately **a computer computes by means of its transistors** turning (switching) on and off : The transistor layer is the complete design !

In Chapters 2, 3 and Appendix A, the architecture layer of the EMY computer is introduced. The architecture is finalized when Input/Output and interrupts are covered in Chapter 8. In chapters 3, 5, 8 (eight) and Appendices B and C the organization and logic levels of the EMY computer are presented. Improvements to the organization in the form of pipelining and a memory hierarchy are introduced in Chapters 6 and 7, respectively. All nine layers shown in the figure above are summarized below. However, before we discuss the layers, we will discuss computer fundamentals, then popular computer classifications and then conclude that the nine layers above give a better view of computers.

1.1. Computer Systems

The fundamentals : **A computer processes digital information**. In order to do that it runs (executes) a machine language program. As an example, when we buy software, such as the Microsoft Word, we buy the machine language program of the Word software. A machine language program manipulates data. A **machine language program** consists of machine language instructions. A **machine language instruction** is a simple command that can be immediately understood by the hardware. It commands the computer to perform a simple operation such as add, subtract, and, shift left, etc. Thus, it can be directly run by the computer (hardware).

Machine language instructions and data are in terms of 1s and 0s and are stored in the memory. It is not possible to distinguish whether a part of the memory has an instruction or a data element by just looking at it. This is a unique property of today's computers and so are called **stored-program computers**. Data and programs are input from input/output (I/O) devices into the computer memory and result data are output to the I/O devices.



Computers are classified with respect to their size, speed and cost as supercomputers, servers, desktop computers and embedded computers. **Supercomputers** are the fastest computers, costing millions of dollars and very large. They are used for scientific applications, such as airplane design, weather forecasting, molecular simulations. Government agencies and large corporations can afford them. **Servers** are large computers that allow multiple users to run general-purpose applications. Companies and universities are typical customers. CS2214 will concentrate on server class computers. **Desktop computers** are single-user machines, intended to run a small numbers applications ranging from email to word processing. **Embedded computers** are very small and control a system they are embedded in. They typically have one application to run which is the control of the system they are in.

1.2. Hardware vs. Software

Another classification is hardware vs. software. **Hardware** is the collection of physical components, such as chips, wires, PCBs, connectors, I/O devices, etc. that form a computer **Software** is the collection of programs on a computer. Software and Hardware are equivalent in that any operation performed by the hardware can be built into software and any operation performed by software can be also directly realized by hardware. Therefore, we have the hardware/software trade-off. This equivalence is under the assumption that there is a basic set of operations implemented in hardware. Decisions on what to include in hardware and software are based on the required speed, cost, reliability, frequency of expected changes, etc.

There are two types of software today : Application and systems. The meaning of the two changes computer to computer. Since we concentrate on large computers, servers, in CS2214, we define **application programs** as those run by ordinary users, such as email, word processing, spreadsheet, simulation programs, etc. **Systems programs** are used to control the hardware to make the computer easy to use, secure and more efficient. Systems software include the operating systems, language translators (compilers, assemblers), linkers, loaders, libraries. They are used by systems people who have special privileges (**access rights**) to use the computer. This distinction is enforced by today's computers in the form of hardware **control states** : user and system states. Application programs are run in the **user state** and if they try to run system software in this mode an **interrupt** (exception) is generated The program is terminated. System programs are run in the **system state**.

Even though software is in machine language, today it is often developed by first writing in a high-level language or an application-oriented language or in assembly language. **High-level languages** include C++, Java, C, Fortran, Cobol, Python, PHP, etc. **Application-oriented languages** contain constructs and keywords to develop a program for a specific class of applications, such as simulating a computer network. **Assembly languages** are related to the architecture of the processor they are targeted for. That is, for a computer with an Intel Pentium processor, one would develop an assembly language program in the Intel assembly language. If the processor is an IBM Power processor, one would write an IBM Power assembly language program.

Since the computer can run only machine language programs, one needs to translate the above programs to machine language programs. To translate from a high-level language program to the machine language program, **compilers** are used : C++ compiler, Java compiler, etc. To translate from an assembly language program to the machine language program, **assemblers** are used : Intel assembler, IBM assembler, etc. To translate from an application-oriented language program to the machine language program, typically preprocessing programs are used to convert to an intermediate form in a high-level language and then they are compiled to the machine language program. Among the three types of languages, application-oriented languages are the highest level, meaning very easy to write and assembly languages are the lowest, meaning hardest to write. Although it is easier to develop application oriented language programs, their corresponding machine code may not be efficient since preprocessors and compilers may not be sophisticated enough to generate an efficient machine code. On the other hand, developing a large assembly language program may not be practical due to the complexity of the language.

The common practice today is that for embedded applications assembly and C programs are developed since embedded programs are not large. For all others high-level and application-oriented languages are used. In order to speed up the assembly language program development, programmers use **pseudoinstructions**. A pseudo instruction is <u>not</u> a real instruction. The CPU <u>cannot</u> execute it. It often requires a complex architectural operation and if it was an actual instruction, it would be a complex (CISC) instruction. In other cases, a pseudo instruction may require a simple architectural operation and would be **one** simple RISC instruction. Why one would have it as a pseudo instruction is then that its <u>syntax</u> is more convenient, i.e. easier to learn and use, as in the case of "MOVE" and "CLEAR"pseudoinstructions asked in Homework I. Several MIPS pseudo instructions are given in the textbook, such as BLT (Branch Less Than) and LI (Load Immediate). Pseudoinstructions used by the programmer are converted to real instructions by the assembler.

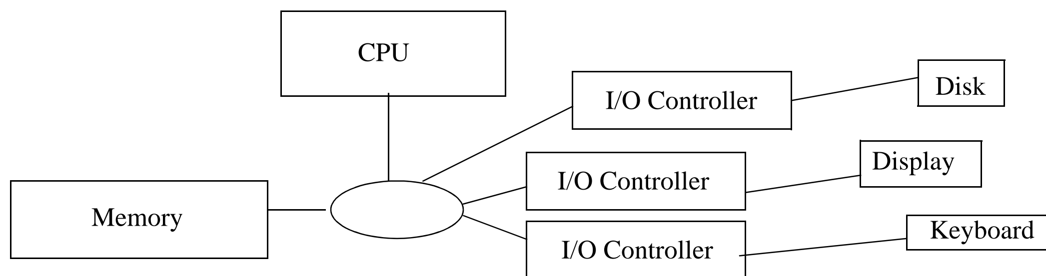
1.3. Architecture vs. Organization (Microarchitecture)

Another computer classification is architecture vs. organization (microarchitecture). The **architecture** is the set of resources visible to the machine language programmer : Registers, the memory, data representations, addressing modes, instructions formats, control states, I/O controllers, interrupts, etc. Although often the architecture is thought to be equivalent to the machine language set of a computer, it is more than that. Still, a major portion of the architecture coverage is devoted the machine language set. A related issue in the past was whether the machine language set should be complex (complex instruction set computer, **CISC**) or simple (reduced instruction set computer, **RISC**). The debate took place in the 1980s and first half of the 1990s. It was resolved as the RISC the winner since it allows more efficient pipelining, leads to simpler hardware and easier increase of the clock frequency. The Intel and Motorola machine language sets are CISC. The Sun is RISC. Why and how the Intel CISC architecture has kept its domi-

nance will be clear later in the semester. But, simply, this has been possible by designing an Intel CPU that converts each Intel CISC instruction to up to three RISC instructions on the fly.

Studying the architecture implies working on machine language programs. But, this is not practical when we design a computer since machine language programs have 1s and 0s. Therefore, in CS2214, we will work on **mnemonic machine language programs**. They are easier to write and in one-to-one correspondence with machine language programs. That is, if one has a mnemonic machine language program, it is very straightforward to obtain the corresponding machine language program. Note that there is often no one-to-one correspondence between assembly language programs and machine language programs.

The **organization** is the set of resources that realizes the architecture which include the CPU, the memory and I/O controllers. These are **digital systems** with registers, buses, ALUs, sequencers, etc. The CPU is responsible for running machine language programs : It runs machine language instructions. Running a machine language instruction is performing a simple operation (command) on data. The memory keeps the programs and data, leading to the stored-program concept of today's computers. I/O controllers interface the I/O devices to the memory and CPU. An I/O controller can control one or more I/O devices. Often the number of I/O devices connected to an I/O controller depends on the speed of I/O devices. A high speed I/O device can be controlled by a single I/O controller while a few slow speed I/O devices can be controlled by a single I/O controller. The stored-program concept and the generic view of a computer organization with at least three digital systems (the CPU, memory and I/O controller) are often attributed to mathematician **John Von Neumann**. However, there is considerable debate on that.



A microprocessor contains at the least the CPU which was the case in the 1970s and early 1980s. Today they include cache memories, bus interfaces, memory management units. High-performance microprocessors from Intel, AMD, Sun, IBM have these functional units. Some other chips in the market today contain memory and even I/O controllers. These are used for embedded applications and called **microcontrollers**, not microprocessors. The reason why the memory and I/O controllers are added is that embedded computers are often required to occupy a small space in the system they are housed in. To reduce the chip count, hence the physical space, this approach is needed.

As the above discussion indicates looking at a computer from different points of view can be at least distractive, if not confusing for beginners of computer design : hardware vs. software, different programming languages, operating systems, compilers, assemblers, architecture vs. organization, etc. That is why the concept of computer layers is used to give a comprehensive view of computers at different complexities or abstraction. Abstraction allows reducing the number of details of a layer with a simpler view. In the computer layers figure on the first page, a layer is abstracted by the layer just above it.

2. Computer Layers

The Application, Computational Method, High-Level Language, Operating Systems and Architecture layers constitute the software layers. The Architecture, Microarchitecture, Logic and Transistor layers constitute the hardware layers. Each layer, except the Application layer, implements the layer above, following the concept of abstraction Clearly, **the Architecture is the hardware/software interface**. A computer architect needs to handle both hardware and software and keep track of advances in both.

2.1. Application Layer :

This layer indicates the set of applications intended for the computer ! Ideally, all applications can be run on a computer. However, in practice the computer is designed to "efficiently" run a subset of them. For example, a computer runs scientific applications, a different computer runs business applications, etc. Our EMY computer will target scientific applications. Specific applications mentioned in the textbook are benchmark suites Linpack, Livermore Loops, Whetstone, Dhrystone, SPEC CPU 2000, SPECWeb and EDN EEMBC (Embedded Microprocessor Benchmark Consortium benchmark of five classes of applications).

2.2. Computational Methods Layer :

This layer is highly theoretical and abstract. The computational method (i) determines characteristics of items (data and other) and work (operations), ii) describes how operations initiate each other during execution, i.e. which operation is followed by which or determining the order of performing operations, and (iii) implicitly determines the amount of parallelism among the operations. Three types of computational methods are frequently covered in the discussion of this topic : control flow, data flow and demand driven.

Today's computers use the **control flow** computational method where the order of operations is specified by the order of instructions in the program. The order implies the execution order and so next instruction to perform is the one that follows the current instruction in the program. If one wants to change the order of execution, explicit control instructions (branch, jump, etc.) must be used, hence the name control flow. This explicit sequence of operations obscures parallelism. Thus, the control-flow is inherently sequential, hindering parallelism and higher speeds. This is the reason why today's supercomputers are very expensive as they need complex compilers, operating systems, hardware and highly trained parallel algorithm designers and programmers to extract parallelism from sequential programs. In **data flow**, an operation starts its execution when all of its operands are available. Since the operand availability determines the order of operations, this method is also called data driven. Many operations can have their operands ready at the same and so they can start execution at the same. Thus, data flow does not hinder parallelism. In fact, the parallelism is explicit to the fullest extend. In **demand driven**, an operation starts when its result is demanded. Many operation results can be demanded at the same and so they can all start execution in parallel. Demand driven computation also has parallelism explicit. Overall, data-flow and demand driven methods are inherently parallel. However, to implement them in full scale today is not efficient given the current technology.

2.3. Algorithm Layer :

The algorithm for an application specifies major steps to generate the output. The algorithm follows the computational method chosen. An algorithm is abstract and short. It is independent of high-level languages. Today, for a single-processor (uniprocessor) computer such as the EMY processor, we write a sequential algorithm in the control-flow method. However, if we have a computer with multiple processors (cores), we write a parallel algorithm but still use the control flow method.

2.4. High-Level Language Layer :

The algorithm developed for an application is coded in a high-level language, such as Fortran, C, C++, Java, etc. Fortran is still the choice of scientific computing, while C is gaining ground. Note that for an algorithm there are different programs possible as each can be in a different high-level language.

2.5. Operating Systems Layer :

This layer interfaces with hardware. That is, it hides hardware details from the programmer and provides, security, stability and fairness in the computing system. Thus, this layer adds **more** code to run on the behalf of the application. The layer also handles interrupts and input/output operations.

2.6. Architecture Layer :

The architecture layer is the hardware/software interface. Its elements include the machine language instruction set, register sets, the memory and Input/Output structures among others. CS2214 discusses this level considerably, starting with the second week of the semester, and so its description here is kept short. Nevertheless, below we give a brief discussion of a dilemma that computer architects had in the 1980s and early 1990s. The discussion is given due to its historical significance :

The dilemma : on the one side, a computer architect would like to include complex instructions (floating-point division, string search, etc.) in the instruction set to perform complex operations directly. The opposite decision is not to include complex instructions : only simple instructions. A missing complex operation is implemented by a piece of code. Obviously, running a code takes longer time than running a single complex instruction, so the simpler machine would be slower than the complex machine. But, a complex architecture results in complex hardware with higher costs, longer development times and difficult upgrading. A simple architecture leads to simpler hardware. But, we need to use a sophisticated compiler to generate an efficient code since an application with complex operations has to be implemented by pieces of code. One can see that the simpler computer is a slow computer when those applications are run.

We must note that both decisions are attractive : a fast machine or a cheaper machine. The fundamental question is this : Are those complex operations needed often ? In other words, are those functions executed often ? If often, a complex architecture is justified : we must make the common case fast.

The division : There have been two camps in computer hardware that promote two different computer architecture philosophies : A Complex Instruction Set Computer, CISC and a Reduced (simple) Instruction Set Computer, RISC. Examples of highly CISC microprocessors are the Intel x86 and Motorola 680X0. Examples of highly RISC microprocessors are MIPS and Sun UltraSPARC. The compromise has been tried in the form of hybrid microprocessors. Highly RISC microprocessors are added CISC features when upgraded. Similarly, highly CISC microprocessors are added RISC features when upgraded. An example of a hybrid microprocessor is the IBM PowerPC microprocessor, which is nevertheless marketed as a RISC microprocessor. Currently, the RISC idea is the favorite since it allows **efficient** pipelining. Even, the Intel x86 architecture relies on RISC execution : Each x86 CISC instruction is converted to up to three RISC operations in the ID cycle and these three RISC operations are executed in the rest of the CPU hardware as if they are in the instruction set. To summarize :

a) There is always a speed/cost trade-off where the higher the speed, the higher the cost.

b) The design of a computer (designing its architecture, organization, logic and chip levels) is based on the targeted application set. That is, we choose the application set for our computer, then design the hardware.

c) Making the common case fast is an attractive design rule as mentioned on page 177 of the textbook. See also pages 176 , 178 and 179 of the textbook.

When textbook problems are analyzed, one can see how computer architects would **ideally** design the architecture of a computer :

- A number of real, commonly used programs are run and a large set of statistics is obtained, such as how often each instruction is executed, which registers are used, etc.
- From the statistics, we decide about which instructions we have to include in the instruction set, how many registers in the register set, types of addressing modes, types of data representations, etc. so that "**we make the common case fast** !"
- We also determine how time consuming it would be if some operations were not implemented by instructions (by hardware), but by software (by functions).

Clearly, the design focuses on application-architecture interactions such that the architecture is "**tuned**" to the applications. For example, our EMY computer is tuned to scientific applications ! Concentrating on one layer at a time (such as the architecture layer) is attractive from computer architecture education point of view. This approach is not attractive practically. Because the resulting computer can violate one or more of the design goals (speed, cost, size, power consumption,...). For example, the computer can be too expensive or too large.

Thus, in practice computer designers work on several levels simultaneously, even though they proceed top-down. When an architectural decision is made, its implications on lower levels are examined. If it is concluded that a particular architectural decision can violate a goal, it is abandoned. For example, if it is decided to allow "word boundary crossings," on the architecture level, we check its implications on the organization level. We might realize that the corresponding hardware is unnecessarily too expensive, so we reverse the decision about word boundary crossings.

2.7. Microarchitecture Layer :

This layer consists of **digital systems**. A computer which is a digital system consists of at least three smaller digital systems : the processor (CPU), the memory and Input/Output controller. A digital system consists of registers, buses, ALUs, sequencers, etc. Other names used for this layer are organization and register transfer level (RTL). The microarchitecture layer is also discussed in depth in CS2214, starting with the fourth week of the semester. Handouts will be distributed during the semester to ensure students understand fundamental microarchitecture concepts.

2.8. Logic Layer :

This layer consists of digital circuits. **Digital circuits** form digital systems of the microarchitecture level. Digital circuits use two types of components : gates and flip-flops. A **gate** outputs 1 or 0, depending on its current input values, i.e. the output now is a function of the inputs now. Most common gates used are AND, OR, NOT, NAND and NOR gates. **A flip-flip stores a single bit**. To store the bit (1 or 0), a clock signal is used. The rising or falling edge of the clock stores the bit. Most common flip-flops used are D and JK flip-flops. A flip-flop is implemented by using a few gates. Then, we can state that all digital circuits consist of gates ! Note that a flip-flop is **not** a memory. The memory chip design is different from the flip-flop design.

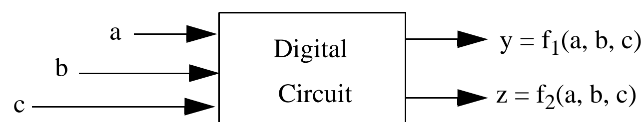
There are two types of digital circuits. A **combinational circuit** contains gates. A combinational circuit changes its output right after an input is changed : the output now is a function of the inputs now. Combinational circuits cannot store information. Examples of combinational circuits are adders, multipliers, comparators, etc. **Sequential circuits** contain gates and flip-flops. They store past inputs : the output now is a function of inputs now and past inputs. Examples of sequential circuits are counters, registers, shift registers, sequencers.

The Logic layer will be discussed less than the architecture and microarchitecture layers in CS2214. It will be the main topic of interest when we cover hardwiring, microprogramming and high-speed arithmetic circuits. Below, we give a brief introduction to digital logic.

2.8.1. Introduction to Digital Logic

In this section, we present formal digital circuit fundamentals needed to implement such structures as registers, buses, ALUs and sequencers. Digital circuits consist of gates and flip-flops. There are two types of digital circuits : Combinational circuits and sequential circuits. Combinational circuits use only gates while sequential circuits use both gates and flip-flops. Most real life circuits are sequential circuits. Combinational circuits are more specific purpose.

The input-output relationship of a digital circuit is important when it is studied. The input-output relationship treats the digital circuit as a black box with inputs and outputs. It relates the output to the inputs : every output is described as a **function** of the inputs. A function is a **mathematical entity** that precisely describes how an output is determined by its inputs. For example, in the figure below, the digital circuit shown as a black box has three inputs and two outputs (two digital functions) :



Functions f_1 and f_2 precisely indicate when outputs y and z are 1 and when they are 0. That is, f_1 specifies the input-output relationship of y to inputs a, b and c and f_2 specifies the input-output relationship of z to inputs a, b and c. A function for a simple combinational circuit is represented by expressions, minterm lists, truth tables, etc. For complex combinational circuits, we do **not** use functions, but **operation tables**. A function for a simple sequential circuit is represented by expressions, state tables and state diagrams. For complex sequential circuits we do *not* use functions, but **operation diagrams**. Due to historical reasons, digital circuits are called **switching circuits**, digital circuit functions are called **switching functions** and the algebra to design combinational circuits is called **Switching Algebra**. Below, we first discuss combinational circuits and then sequential circuits.

2.8.2. Combinational Circuits

A combinational circuit changes its output when its inputs are changed. This is because it uses gates which change their outputs when their inputs are changed. The time it takes for the gate output to change is several hundred pico seconds today. Therefore, a combinational circuit will change its output in a time duration in terms of nano seconds. There is no time dimension in combinational circuits, since they change their outputs if inputs are changed.

A number of combinational circuits are frequently used in digital systems : Multiplexers, decoders, encoders, demultiplexers, adders, comparators and parity checkers. Their design has been studied extensively in literature. Students can take a look at books on digital logic to understand their operation. Typical combinational design follows the rule of thumb that if it has 4 inputs or less it is immediately designed by using Switching Algebra. Otherwise, the circuit is partitioned into smaller and smaller blocks until they have 4 inputs or less or until they are popular digital circuits and there are available designs for them.

2.8.2.1. Switching Algebra

George Boole, in 1854, introduced a systematic treatment of logic and developed for this purpose an algebraic system now called **Boolean Algebra**. This algebraic system, **(S ; + ; . ; - ; 0 ; 1)**, consists of a set *S* of elements, **binary** operations "+" (called "plus") and "." (called "dot") and a **unary** operation "-" (called "complement") and at least two elements "0" and "1" in the set *S*. E. V. Huntington in 1904 defined a **many-valued Boolean Algebra** with the following postulates (axioms) :

PI) There exists a set *S* of elements which satisfies the principle of substitution under the equivalence relation "=" that if k = m, then "m" may be substituted for "k" in any expression containing "k," without affecting the validity of the expression.

PII) a) The set *S* is closed with respect to the "+" operation : if k, m Œ S, then (k + m) Œ S

PII) b) The set *S* is closed with respect to the "." operation : if k, m Œ S, then (k + m) Œ S

PIIIa) k + 0 = k — 0 is the identity element with respect to "+"
PIIIb) k . 1 = k — 1 is the identity element with respect to "."

PIVa) k + m = m + k — "+" is commutative
PIVb) k . m = m . k — "." is commutative

PVa) k + (m . p) = (k + m) . (k + p) — "+" is distributive over "."
PVb) k . (m + p) = (k . m) + (k . p) — "." is distributive over "+"

PVIa) For every element k of set *S*, there exists an element $\overline{k}$ (complement of k) of set *S*, such that k + ($\overline{k}$) = 1
PVIb) For every element k of set *S*, there exists an element $\overline{k}$ (complement of k) of set *S*, such that k . ($\overline{k}$) = 0

Above, we do not mention the number of elements in *S* and how the operators manipulate the elements of *S*. In fact, one can formulate many Boolean Algebras depending on the number of elements in *S* and operator definitions. Among all Boolean Algebras, the **two-valued (two-element) Boolean Algebra** also known as **Switching Algebra** is the most widely known and studied. Switching Algebra was developed by C. Shannon in 1938.

Switching Algebra is defined by

- Algebraic system (**0** ; **1** ; + ; **.** ; **-**) and
- The six postulates above under the condition that
  - The following operator rules on (0 ; 1) apply, i.e. operator definitions :

k	m	k.m
0	0	0
0	1	0
1	0	0
1	1	1

Definition of the **AND** operator

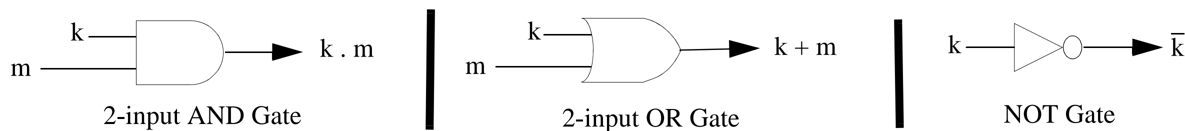
k	m	k+m
0	0	0
0	1	1
1	0	1
1	1	1

Definition of the **OR** operator

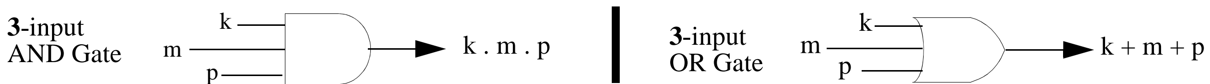
k	$\overline{k}$
0	1
1	0

Definition of the **Complement** (**NOT, Invert**) operator

Today, each operator above is directly implemented by **electronic circuits**. That is, a digital electronic circuit with transistors, capacitors, resistors and other electronic components is designed to perform the specific logic operation. Each such digital electronic circuit for an operator is called "**gate**." The figure below shows the **schematic** symbols of gates for the three operators above.



AND and OR gates can have ***any number of inputs****, as long as there are at least two inputs.* However, a NOT gate *always has a* ***single input****.*



There are **other** operators, such as **NAND**, **NOR**, **EXOR** and **EXNOR**, that are implemented by gates.

Today, on a chip electronic circuits with **transistors** implement several gates to millions of gates. There are chips, such as microprocessor chips, with hundreds of millions of transistors. Chip densities have increased at the rate of **Moore's Law** since 1960s : **The number of transistors on a chip doubles every two years.**

Since 1938, theorems have been developed in Switching Algebra. These theorems follow and satisfy the postulates and operator definitions given above :

TI) Idempotency :
a) $k + k = k$
b) $k . k = k$

TII) Null elements
a) $k + 1 = 1$
b) $k . 0 = 0$

TIII) Absorption :
a) $k + (k . m) = k$
b) $k . (k + m) = k$

TIV) Involution : $((\overline{\overline{k}})) = k$

TV) Associativity :
a) $k + (m + p) = (k + m) + p = k + m + p$
b) $k . (m . p) = (k . m) . p = k . m . p$

TVI)
a) $k + ((\overline{k}).m) = k + m$
b) $k . ((\overline{k}) + m) = k . m$

TVII) DeMorgan's theorems :
a) $\overline{(k + m)} = \overline{k} \, . \, \overline{m}$
b) $\overline{(k \, . \, m)} = \overline{k} + \overline{m}$

TVIII) Consensus theorem :
a) $(k \, . \, m) + ((\overline{k}) \, . \, p) + (m \, . \, p) = (k \, . \, m) + ((\overline{k}) \, . \, p)$
b) $(k + m) \, . \, ((\overline{k}) + p) \, . \, (m + p) = (k + m) \, . \, ((\overline{k}) + p)$

The duality principle :
A postulate or theorem can be obtained from another postulate or theorem by interchanging the binary operators "+" and "**.**" and the identity elements "**0**" and "**1**."

The AND operator symbol :
In order to reduce the number of symbols in expressions, we will **not** show the "." symbol between variables. We will imply there is an AND operation between them : a.b.c = a b c

Precedence rules :
In order to reduce the number of parentheses, we have the following set of precedence rules that indicates which sub-expression or operator to evaluate next :

- Evaluate the expression inside a pair of parentheses
- Evaluate NOT
- Evaluate AND
- Evaluate OR

Example : $(\overline{a}).(((b.(\overline{c})) + (b.c)) + ((a.c).(b + (\overline{b}))) = \overline{a}(b\,\overline{c} + bc) + ac(b + \overline{b})$

The truth table for a combinational circuit shows the output value for every input combination. For example, for the 4-input imaginary circuit below, the following truth table relates the output to the inputs. The truth table describes the function, the input/output relationship :

Truth Table :

	A B C D	**f(A, B, C, D)**
0	0 0 0 0	0
1	0 0 0 1	1
2	0 0 1 0	0
3	0 0 1 1	1
4	0 1 0 0	0
5	0 1 0 1	1
6	0 1 1 0	0
7	0 1 1 1	1
8	1 0 0 0	0
9	1 0 0 1	1
10	1 0 1 0	0
11	1 0 1 1	0
12	1 1 0 0	0
13	1 1 0 1	1
14	1 1 1 0	0
15	1 1 1 1	1

Traditionally, the OR operation is called "**sum**" and the AND operation is called "**product**" since the OR performs similar to the sum operation and the AND performs similar to a multiply operation. One can develop an expression that focuses on the 1s of the output such that it would have as many terms as there are 1s on the output. Each term is an input combination that generates a 1 for the output. Such an expression would have product terms summed and is called the canonical SOP expression. Each product term is canonical product term and has all the inputs of the functions. For example, on the above truth table, there are seven 1s, therefore the canonical SOP expression would have seven canonical product terms. The canonical SOP expression of the above truth table is the following :

$$f(\mathbf{A, B, C, D}) = \overline{A}\,\overline{B}\,\overline{C}\,D + \overline{A}\,\overline{B}\,C\,D + \overline{A}\,B\,\overline{C}\,D + \overline{A}\,B\,C\,D + A\,\overline{B}\,\overline{C}\,D + A\,B\,\overline{C}\,D + A\,B\,C\,D$$

The expression implements the function because when a canonical product generates a 1, the whole expression becomes 1 : 1 OR any term is 1. All we have to do is to show that each canonical product term corresponds to an input combination. Here, we show the correspondence between the first canonical product term and the top input combination that generates a 1 on the truth table. The first term which is $\overline{A}\,\overline{B}\,\overline{C}\,D$ generates a 1, if $\overline{A}$, $\overline{B}$, $\overline{C}$ and D are all 1s. In order for this to happen, A, B and C must zero and D must be 1 so that $\overline{0}$ AND $\overline{0}$ AND $\overline{0}$ AND 1 is 1 AND 1 AND 1 AND 1 which is 1. We see that if an input is complemented, we need to have the input as 0, otherwise 1 to determine the input combination. Then, the input combination that correspondents to $\overline{A}\,\overline{B}\,\overline{C}\,D$ is 1 since it is 0001 on the truth table. The other canonical product terms are for the remaining six input combinations 3, 5, 7, 9, 13 and 15 :

$\overline{A}\,\overline{B}\,\overline{C}\,D$	$\overline{A}\,\overline{B}\,C\,D$	$\overline{A}\,B\,\overline{C}\,D$	$\overline{A}\,B\,C\,D$	$A\,\overline{B}\,\overline{C}\,D$	$A\,B\,\overline{C}\,D$	$A\,B\,C\,D$
0 0 0 1	0 0 1 1	0 1 0 1	0 1 1 1	1 0 0 1	1 1 0 1	1 1 1 1
1	3	5	7	9	13	15

Another function representation is the minterm list that contains the minterms of the function. A minterm is an input combination that generates a 1 for the output. Since a canonical product term also has the same property, we can state that a minterm is a canonical product term. The minterm list is directly obtained from the truth table. For the above circuit, the minterm list is as follows :

The minterm list :

$$\mathbf{f(A, B, C, D)} = \sum m(1,3,5,7,9,13,15)$$

If we have a canonical expression or **any** expression for a function, we can use Switching Algebra to simplify it. One can also **Karnaugh** maps to obtain minimal expressions. Below, we give the simplification of a complex expression that describes a function :

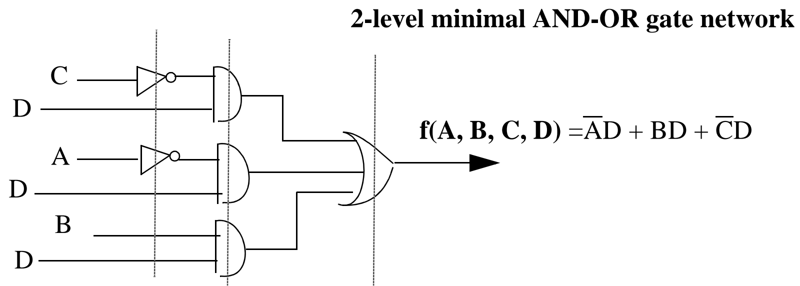
$\mathbf{f(A, B, C, D)} = D(AB + \overline{C}) + \overline{A}BCD + \overline{A}\,\overline{B}\,D$	**A nonminimal expression for function f**
$= ABD + \overline{C}D + \overline{A}D(\overline{B} + BC)$	$k(m + s) = km + ks$
$= ABD + \overline{C}D + \overline{A}D(\overline{B} + C)$	$k + \overline{k}m = k + m$
$= ABD + \overline{C}D + \overline{A}\,\overline{B}\,D + \overline{A}CD$	$k(m + s) = km + ks$
$= ABD + \overline{A}\,\overline{B}\,D + D(\overline{C} + C\overline{A})$	$k(m + s) = km + ks$
$= ABD + \overline{A}\,\overline{B}\,D + \overline{C}D + \overline{A}D$	$k + \overline{k}m = k + m$ & $k(m + s) = km + ks$
$= ABD + \overline{C}D + \overline{A}D(1 + \overline{B})$	$k(m + s) = km + ks$
$= ABD + \overline{C}D + \overline{A}D$	$k + 1 = 1$ & $k1 = k$
$= \overline{C}D + D(\overline{A} + AB)$	$k(m + s) = km + ks$
$= \mathbf{\overline{C}D + \overline{A}D + BD}$	$k + \overline{k}m = k + m$ & $k(m + s) = km + ks$

expressions for function f

Minimal SOP expression

Once we have the minimal SOP expression, we draw the combinational circuit (the gate network) as the last step. The resulting gate network is what we call the 2-level AND-OR gate network. Below, we show the minimal 2-level AND-OR gate network for the above function :

2-level minimal AND-OR gate network



Note that the above circuit has **three** levels : a level of inverters, a level of AND gates and a level of the OR gate. However,, these gate networks are still called 2-level AND-OR gate networks and we will keep that name. The rea-

son why we try to obtain SOP expressions is that they are implemented by 2 (3)-level gate networks that are the fastest possible we can have as explained below. Note that a 1-level gate network has only one gate which cannot be useful for real-life applications.

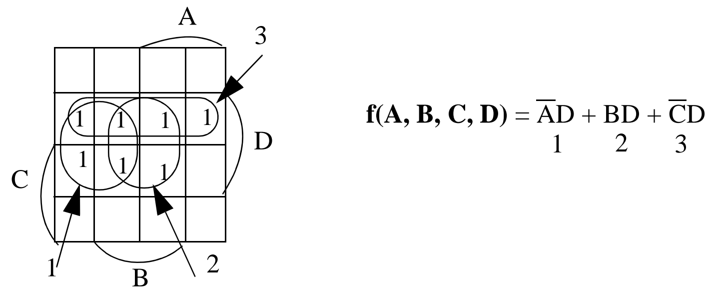
In summary, a function, an input/output relationship, for a combinational circuit can be expressed in different formats, that include the truth table, the minterm list, the minimal sum-of-products (SOP) expression, the canonical SOP expression and the Karnaugh map. Among these representations, the minimal SOP expression and the minimal POS expressions are the designs of the circuits. This is because, they directly describe circuits (gate networks) and as the word “minimal” implies, the circuits are minimal. Why SOP and POS expressions are worked on is that they lead to 2 (3)-level gate networks that are the fastest combinational circuits. Often the SOP expression is preferred when circuits are designed since it looks like an ordinary algebra expression. For the sake of brevity, we will not discuss POS expressions here. Students are referred to books on digital logic.

2.8.2.2. Karnaugh Map Simplifications

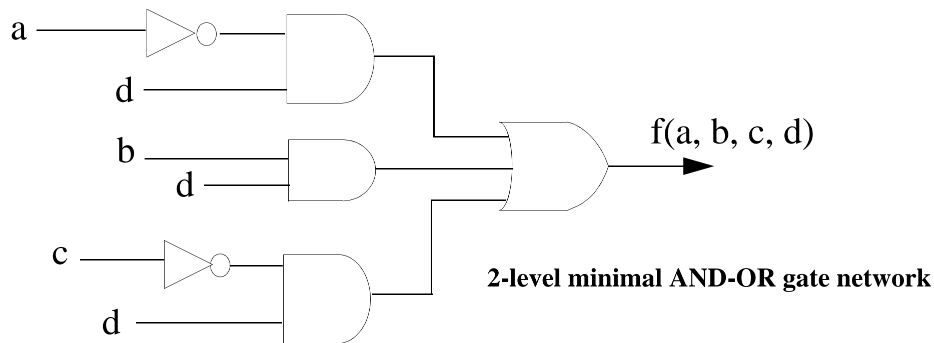
Canonical SOP and canonical POS expressions are almost all the time not minimal. We need to obtain minimal expressions from them. One can use Switching Algebra at the expense of slow and error prone simplification process or the Karnaugh map technique. To use the Karnaugh map technique (to obtain the minimal SOP expression) we start with the minterm list. One can use the Karnaugh map technique to obtain minimal POS expressions as well. In this one starts with the maxterm list. We will not discuss POS expression simplifications via Karnaugh maps.

There is a standard K map for a certain number of inputs. In this section, we will have K maps for four-input cases. The K-map method starts with placing the minterms on the map. Then, one combines logicall/vertically adjacent 2^k minterms and ontains a product term. If there are n inputs and ^{k} minterms are combined, the product term must have (n - k) variables. This can be used to verify that the combination is legal. Note that minterms along the edges are adjacent. One has to cover all the minterms of the function to get the minimal SOP expression. Therefore, the goal is to cover the minterms by using minimum number of largest combinations.

The minimal SOP expression for the above functions is as obtained below :

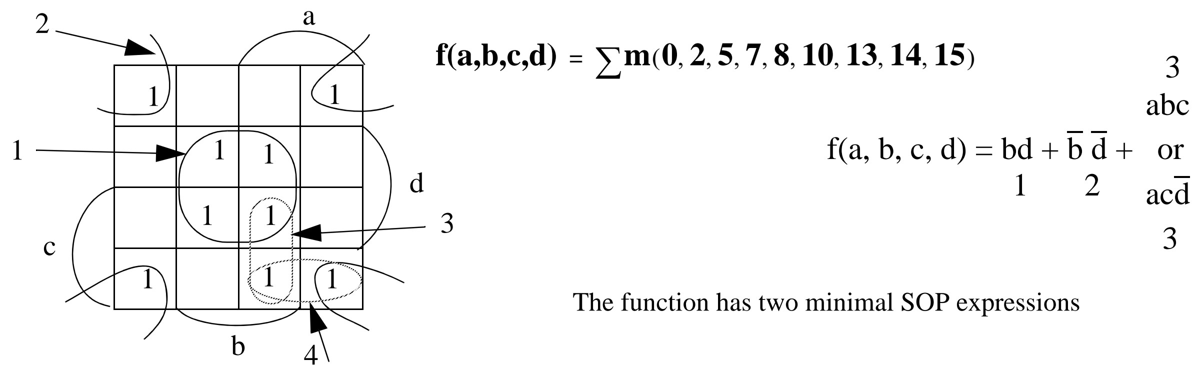


The corresponding 2-level AND-OR gate is as follows :

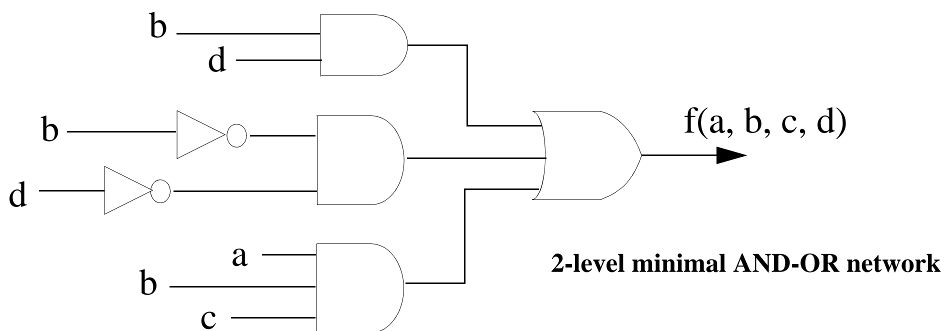


2-level minimal AND-OR gate network

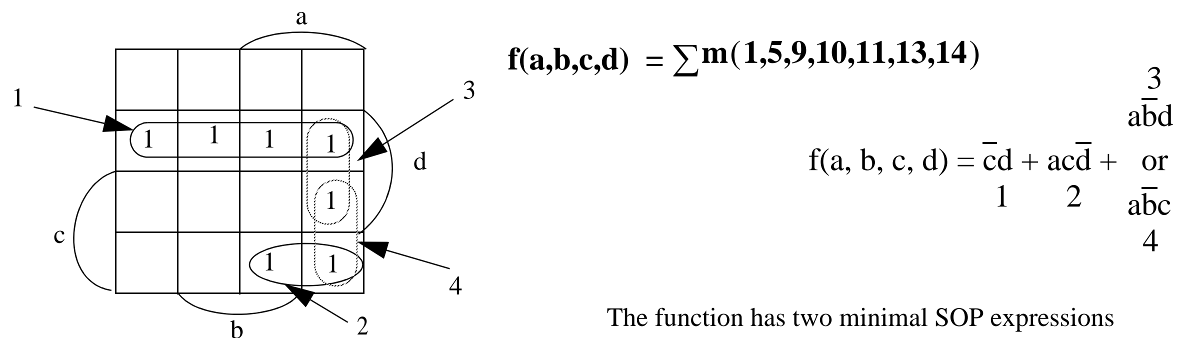
Another example to obtain the minimal SOP expression of a function by using the Karnaugh-map method :



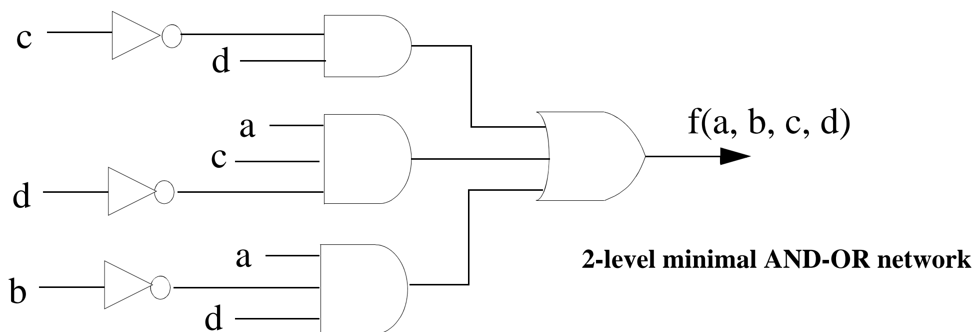
Either circuit is minimal. We choose the one with combinations 1, 2 and 3 :



The last example to obtain the minimal SOP expression by using the Karnaugh-map method :

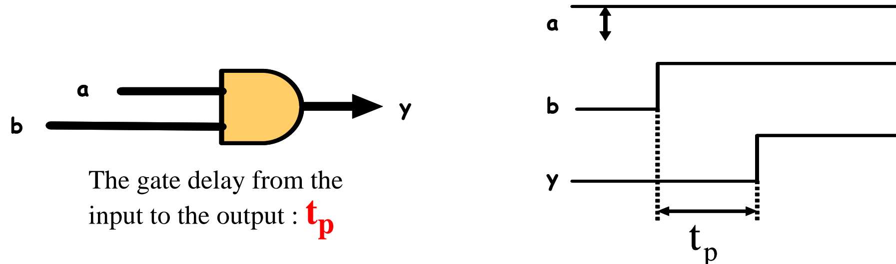


Either circuit is minimal. We choose the one with combinations 1, 2 and 3 :



2.8.2.3. Combinational Circuit Speed

Gates in combinational circuits output values based on the inputs. If an input is changed, the output changes after a short delay which is called propagation (gate) delay, t_p. This delay is a function of electical properties of the gate, including the **process** of the gate. We describe the process in the Transistor Layer section. Today, this delay is a few nanoseconds or less. Note that when we mention the speed of a gate, we mean its gate delay : **the shorter the delay, the faster the gate is**.



The gate delay from the input to the output : t_p

The gate delay is one of the **three** factors that determine the speed of a combination circuit. The other two factors are the longest path from an input to the output, i.e. the number of gate levels and wire delays betwen the gates. 2-level gate networks have the shortest path from an input to the output and so they are the fastest circuits. Wire delays also contribute to the speed since it takes some time for signals to travel from one gate to another. Although, 2-level gate networks are satisfactory for high speed, they result in expensive circuits.

2.8.2.4. Examples of Combinational Circuit Design

Below, we give examples of designing simple combinational circuits, algebraic simplifications and Karnaugh map simplification so that students understand the purpose of the postulates and theorems .

A (1-bit) 2-to-1 Multiplexer

A 1-bit 2-to-1 Multiplexer (MUX) is a selector which selects one of the two inputs based on a select signal. As seen below, it has three inputs and one output. Two inputs (b and c) are data inputs one of which is output. The third input (a) is the control input, the select input. The single output is always equal to either b or c at any time. The MUX is a 1-bit MUX since when an input is selected, there is only one data line selected. As the gate network shows, the MUX has three gate delays. .

a, b, c → 1-bit 2-to- 1 MUX → y(a, b, c)

If a = 0 then y = b
a = 1 then y = c

	a b c	y(a, b, c)
0	0 0 0	0
1	0 0 1	0
2	0 1 0	1
3	0 1 1	1
4	1 0 0	0
5	1 0 1	1
6	1 1 0	0
7	1 1 1	1

$$y(a,b,c) = \sum m(2,3,5,7) = \underbrace{\bar{a}\,b\,\bar{c}}_{2:\ 010} + \underbrace{\bar{a}\,b\,c}_{3:\ 011} + \underbrace{a\,\bar{b}\,c}_{5:\ 101} + \underbrace{a\,b\,c}_{7:\ 111}$$

$$y(a,b,c) = \bar{a}\,b\,(\bar{c} + c) + a\,c\,(\bar{b} + b) \quad \rightarrow \quad k(m+p) = km + kp$$

$$= \bar{a}\,b\,1 + a\,c\,1 \quad \rightarrow \quad k + \bar{k} = 1$$

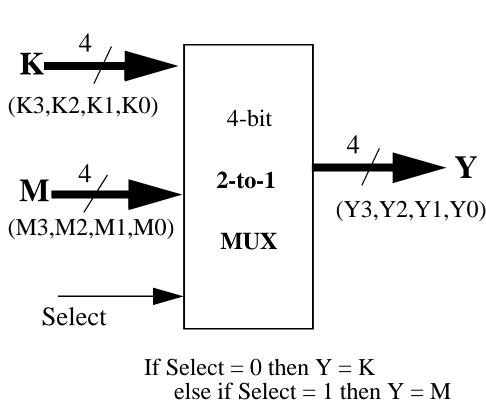
$$= \bar{a}\,b + a\,c \quad \rightarrow \quad k1 = k$$

a → NOT → $\bar{a}$; b, $\bar{a}$ → AND; c, a → AND; both → OR → y

One can develop 2-bit 2-to-1 MUXes, 4-bit 2-to-1 MUXes, etc, by using a number of 1-bit 2-to-1 MUXes as described in the next section. Note also that there are k-bit 4-to-1 MUXes, k-bit 8-to-1 MUXes, etc.

A 4-bit 2-to-1 Multiplexer

A 4-bit 2-to-1 MUX has two sets of data inputs. Each set of data inputs has four bits. Thus, the MUX has four outputs carrying the values of the four input lines selected. The MUX has 9 inputs and 4 outputs. The single input is Select and the 4-bit inputs are K and M. The 4-bit otput is Y. It outputs K if Select is 0 and outputs M if Select is 1. The black box view and implementation of the 4-bit 2-to-1 MUX is given below.



Since there are 9 inputs, we need to partition it into simpler pieces !
We have to obtain the operation table of the 4-bit 2-to-1 MUX :

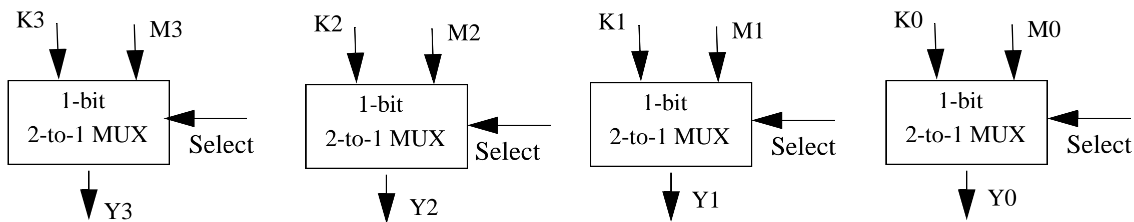
Select	Operation
0	Y = K
1	Y = M

The major operations are not clear on this operation table. We need to get a different, more detailed operation table :

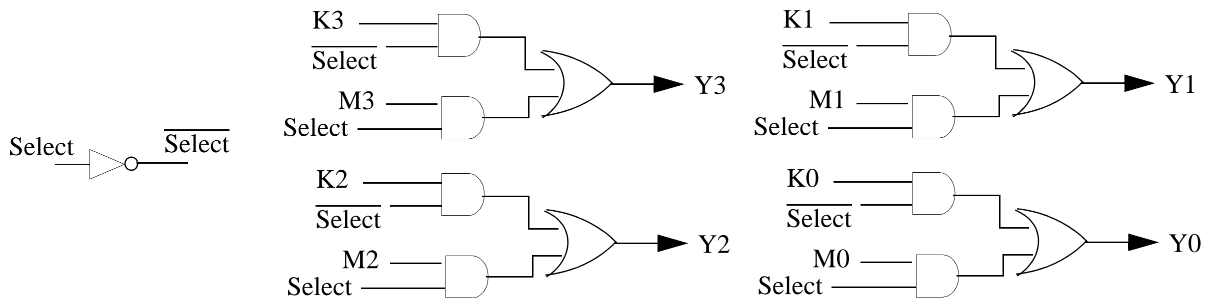
Select	Operation
0	Y3 = K3 ; Y2 = K2 ; Y1 = K1 ; Y0 = K0
1	Y3 = M3 ; Y2 = M2 ; Y1 = M1 ; Y0 = M0

There are four identical major operations : 1-bit 2-to-1 MUXing !

We partition the 4-bit 2-to-1 MUX into four blocks. Each block is a 1-bit 2-to-1 MUX which we have designed by using Switching Algebra : It has three inputs and one output :

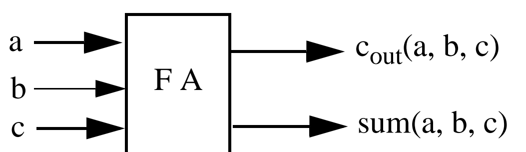


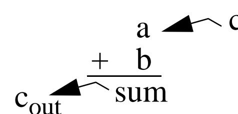
The 4-bit 2-to-1 MUX is then as follows :



A 1-bit Adder, Full Adder

A 1-bit adder, a Full ADDer (FA) adds two 1-bit numbers plus a carry input. Therefore, it adds three bits. It has 3 inputs and 2 outputs as shown below.





We obtain the truth table from which we obtain the canonical SOP expressions :

	a b c	$c_{out}(a, b, c)$	sum(a, b, c)
0	0 0 0	0	0
1	0 0 1	0	1
2	0 1 0	0	1
3	0 1 1	1	0
4	1 0 0	0	1
5	1 0 1	1	0
6	1 1 0	1	0
7	1 1 1	1	1

$$c_{out}(a,b,c) = \sum m(3, 5, 6, 7) \;\; \begin{array}{cccc} 3 & 5 & 6 & 7 \\ 0\,1\,1 & 1\,0\,1 & 1\,1\,0 & 1\,1\,1 \end{array} \;\; \bar{a}\,b\,c + a\,\bar{b}\,c + a\,b\,\bar{c} + a\,b\,c$$

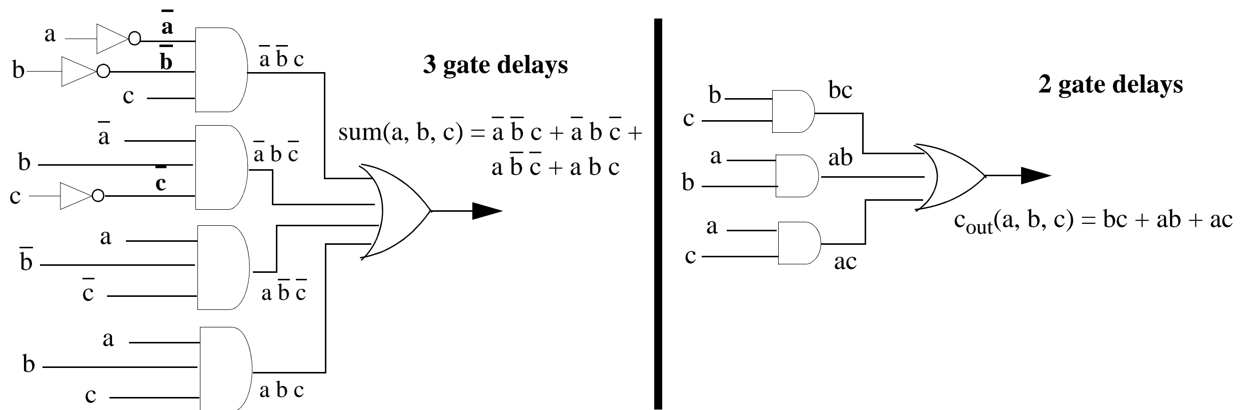
$$sum(a,b,c) = \sum m(1, 2, 4, 7) \;\; \begin{array}{cccc} 1 & 2 & 4 & 7 \\ 0\,0\,1 & 0\,1\,0 & 1\,0\,0 & 1\,1\,1 \end{array} \;\; \bar{a}\,\bar{b}\,c + \bar{a}\,b\,\bar{c} + a\,\bar{b}\,\bar{c} + a\,b\,c$$

The canonical sum(a, b, c) expression is also the minimal expression. It cannot be simplified :

$sum(a, b, c) = \bar{a}\,\bar{b}\,c + \bar{a}\,b\,\bar{c} + a\,\bar{b}\,\bar{c} + a\,b\,c$

$$\begin{aligned} c_{out}(a, b, c, d) &= \bar{a}\,b\,c + a\,\bar{b}\,c + a\,b\,\bar{c} + a\,b\,c \\ &= bc(\bar{a} + a) + a\,\bar{b}\,c + a\,b\,\bar{c} && k(m+p) = km + kp \\ &= bc + a\,\bar{b}\,c + a\,b\,\bar{c} && k+\bar{k} = 1 \;\&\; k1 = k \\ &= c(b + a\,\bar{b}) + a\,b\,\bar{c} && k(m+p) = km + kp \\ &= c(b + a) + a\,b\,\bar{c} && k + \bar{k}m = k + m \\ &= bc + ac + a\,b\,\bar{c} && k(m+p) = km + kp \\ &= b(c + a\bar{c}) + ac && k(m+p) = km + kp \\ &= b(c + a) + ac && k + \bar{k}m = k + m \\ &= bc + ab + ac && k(m+p) = km + kp \end{aligned}$$

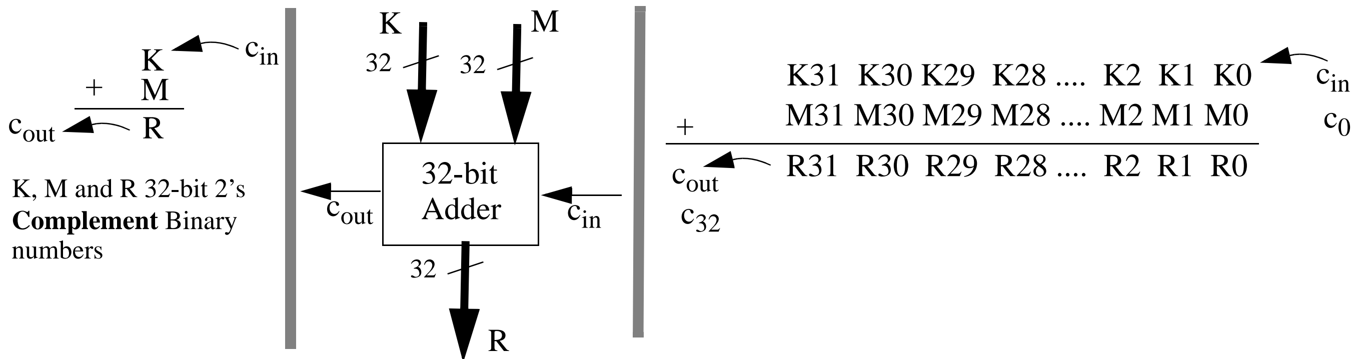
The 2-level AND-OR gate networks are as follows :



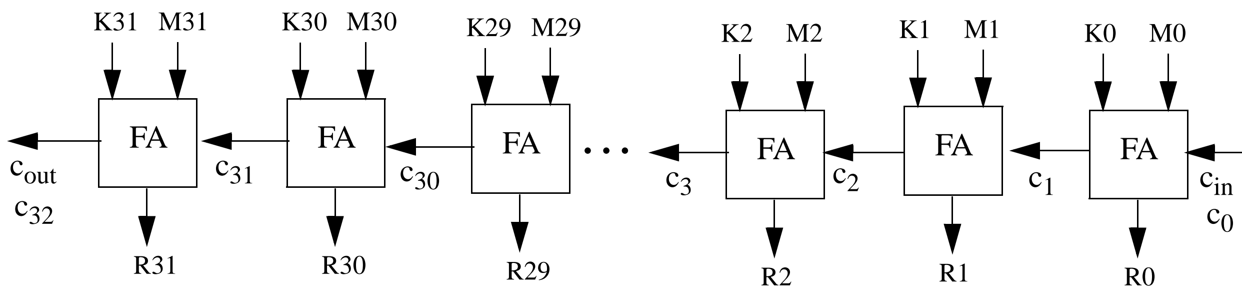
Therefore, a Full Adder takes 3 gate delays to generate the sum output (sum(a, b, c)) and two gate delays to generate the carry out (c_{out}(a, b, c)).

A 32-bit Ripple-Carry Adder

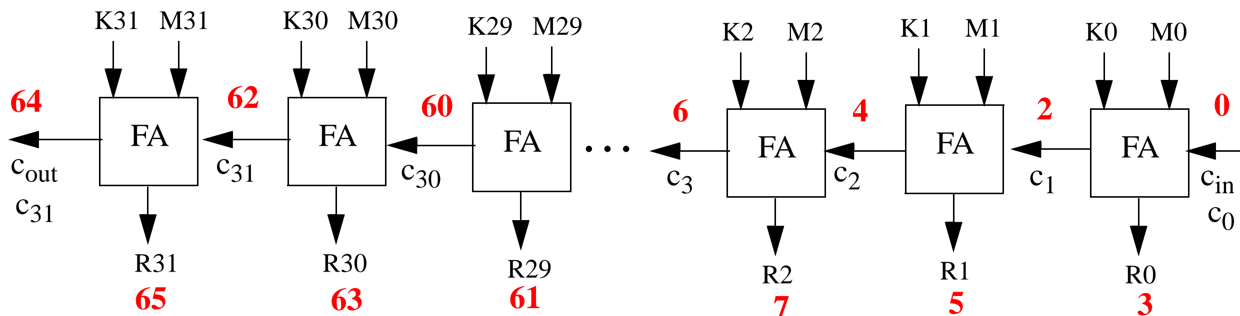
An important component of a digital system is the ALU which has an adder, multiplier, AND, OR and other functional units. The adder is the most critical one since its speed partly determines the clock frequency of the digital system. CPUs have typically a 32-bit adder which has to complete its operation in one or a few clock periods. Thus, a high-speed adder has to be designed.



A 32-bit adder adds two 32-bit numbers plus a carry input. It has 65 inputs and 33 outputs. Our starting point to design a high-speed adder is a 32-bit Ripple-Carry Adder which is the slowest that can be designed. The Ripple-Carry Adder has 32 1-bit adders, known as Full Adders :



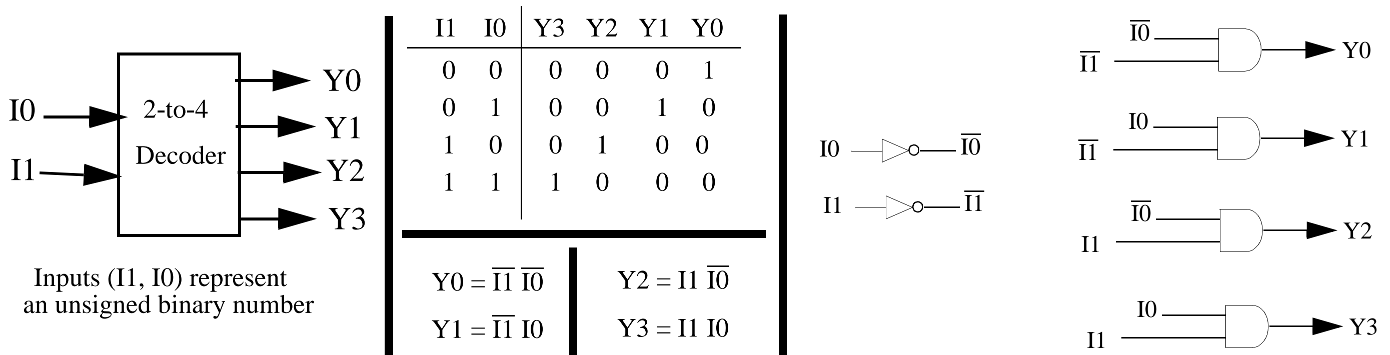
Each one of our 32 Full Adders have the above two gate networks. By using these two gate delays, we can obtain the worst case addition time for our 32-bit Ripple-Carry Adder :



Our 32-bit Ripple-Carry Adder takes 65 gate delays to calculate the sum. If a gate delay is 1ns, the addition time is 65ns. This is **very long** for today's standards. We need to improve the timing. We will do that by designing a 32-bit Carry-Lookahead Adder in class.

A 2-to-4 Decoder

The most common decoder is the binary decoder which has "k" data inputs and 2^k outputs. If thedecoder is a 2-to-4 decoder, then "k" is 2 and so there are $2^2 = 4$ outputs. The "k" inputs represent an unsigned binary number. The outputs decode the unsigned number represented by the "k" inputs.For example if the inputs represent $(3)_{10}$, Output line 3 is 1 and the other outputs lines are 0. Below the development of the 2-to-4 decoder is shown.



I1	I0	Y3	Y2	Y1	Y0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

The decoder outputs require at most two gate levels to generate the outputs. Therefore, the decoder is fast. Note that there are 3-to-8, 4-to-16, etc. decoders whose operation and implementation follow similarly. We will use a 4-to-16 decoder when we implement hardwiring later in the semester. Today's memory chips (DRAM, SRAM, ROM, etc.) have large binary decoders.

For a k-input decoder, there are 2^k AND gates. Each input is connected to half the number of AND gates. For small size decoders, this is not a major problem. But, for large decoders, it is a problem which is called "**fan-out**." The fan-out of line is a number which indicates how many inputs can be connected to it. If the number is exceeded, electrically, there are problems and so the circuit may not work. It is because of this reason that the decoders of memory chips have their gate networks with more than two levels to reduce the fan-out requirement. However, with more levels, the decoder is slower, therefore, the memory chip is slower.

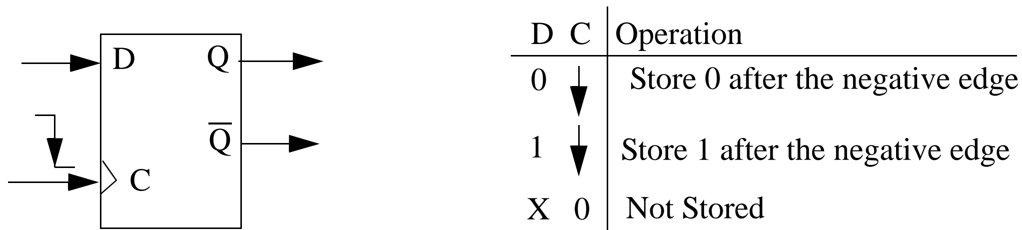
2.8.3. Sequential Circuits

A sequential circuit consists of flip-flops and gates. It has flip-flops to store bits, meaning past inputs. Therefore, a sequential circuit output depends on the present inputs and also past inputs. This means sequential circuits have the time dimension. A flip-flop operates different from a gate such that it stores a bit, if it receives an "edge" on the clock signal. The edge is either a high-to-low transition of the signal (negative edge) or the low-to-high transition of the signal (positive edge). A flip uses only one type of these two edges. For example, if a flip-flop stores when it receives a negative edge, then we say it is negative-edge triggered.

There are several types of flip-flops. One that is used to implement registers is the Data (**D**) flip-flop. Another frequently used flip-flop is the **J-K** flipflop used to implement counters. A D flip-flop has a single data input which is stored when the clock edge is received. A J-K flip-flop has two inputs and is stored a bit when a clock edge is received. The timing of the edge is controlled by a Store control signal generated by the control unit.

2.8.3.1. Flip-Flops

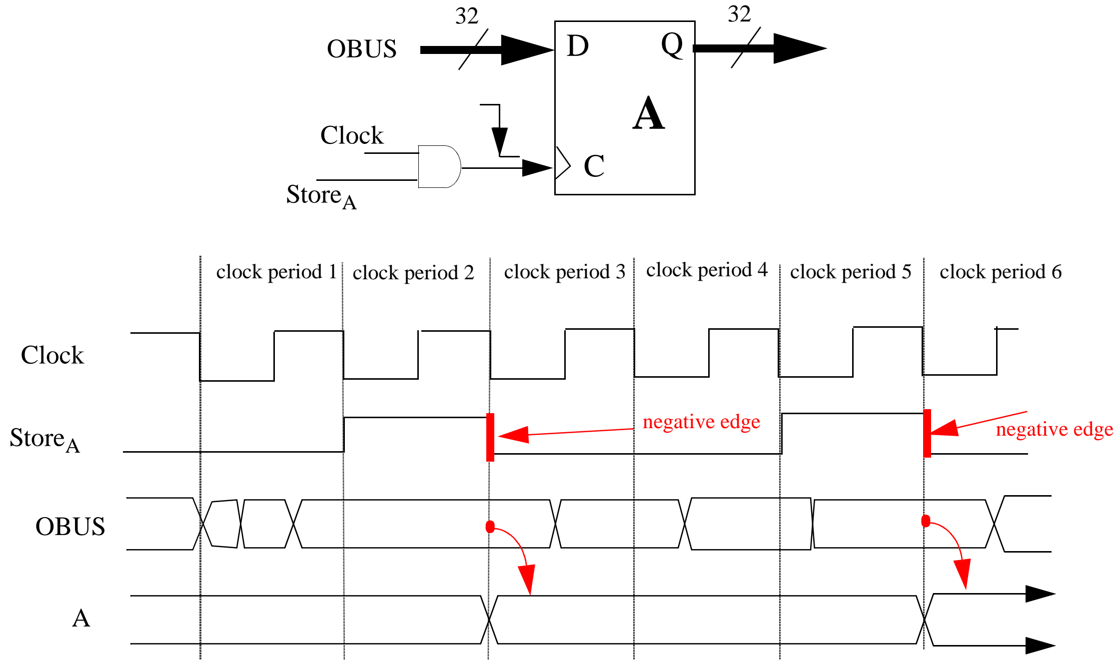
A flip-flop today has two outputs : One uncomplemented, Q, and the other one complemented, $\overline{Q}$ as shown below. The triangle symbol next to the clock input, C, indicates it is an edge triggered clock input. The high-to-low transition symbol indicates the flip-flop is stored when there is a **negative** edge. What is stored on the flip-flop depends on the data input, D. According to the operation table below, when there is a negative edge on the clock input and the D input is 0, we store 0 after the negative edge, typically, a few nano seconds after the negative edge. If the D input is 1 at the edge, we store 1 after the edge. Finally, the last row, indicates at all other times (when there is no negative edge), the D input is ignored. This is also known as "Don't Care" and is shown by an "X" symbol. Below, we give the operation table of a **negative**-edge triggered D flip-flop



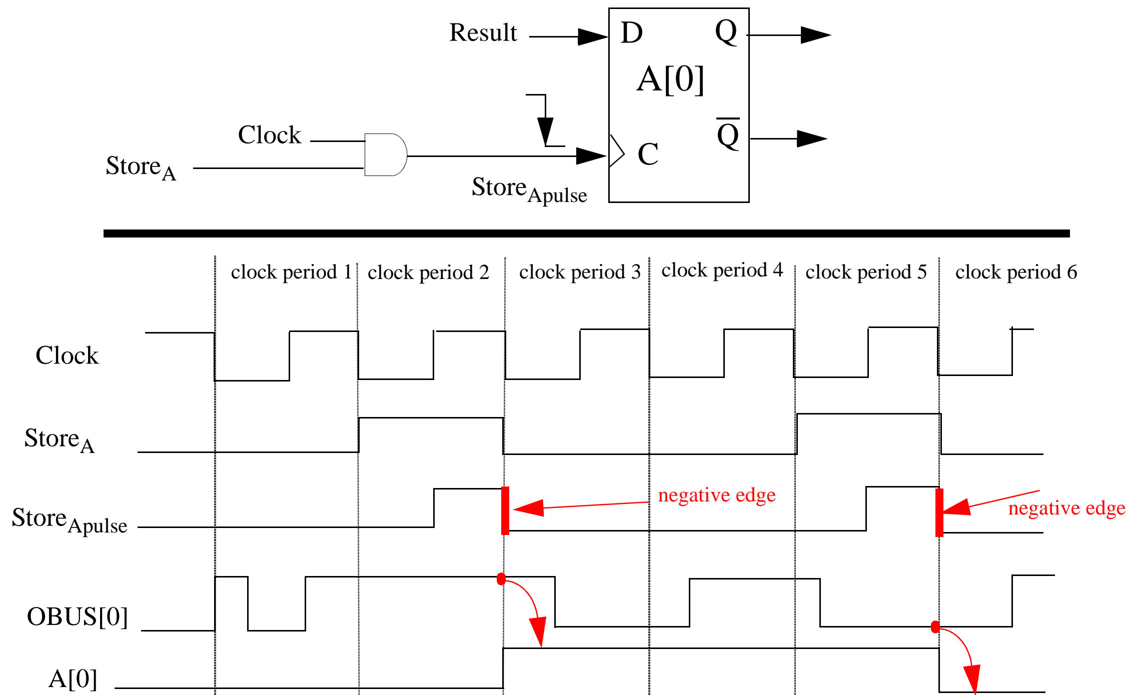
D	C	Operation
0	↓	Store 0 after the negative edge
1	↓	Store 1 after the negative edge
X	0	Not Stored

2.8.3.2. Registers

A register is a sequential circuit used to store data temporarily. It is stored data by applying a clock edge at the end of the clock period it needs to be stored. Note the a register which is stored a value in a particular clock period actually gets the value in the beginning of the following clock period. The example below shows an imaginary 32-bit register named **A** which is stored a value when its $Store_A$ signal is 1 in clock periods 2 and 5. The D flip-flops of the register receive the edge at the end of clock periods 2 and 5.



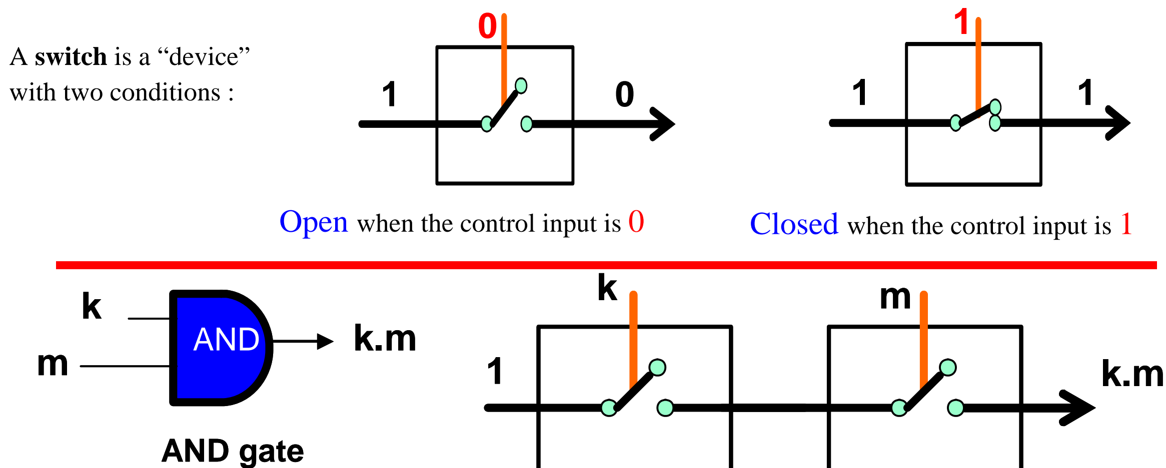
We study how the D flip-flop above can be used to store bits by using one of the bits of OBUS shown for the register example above. The rightmost flip-flop A[0] and how it is stored are shown below. Note that we AND the clock signal and a Store signal so that we do not store every clock period, but when it is necessary.



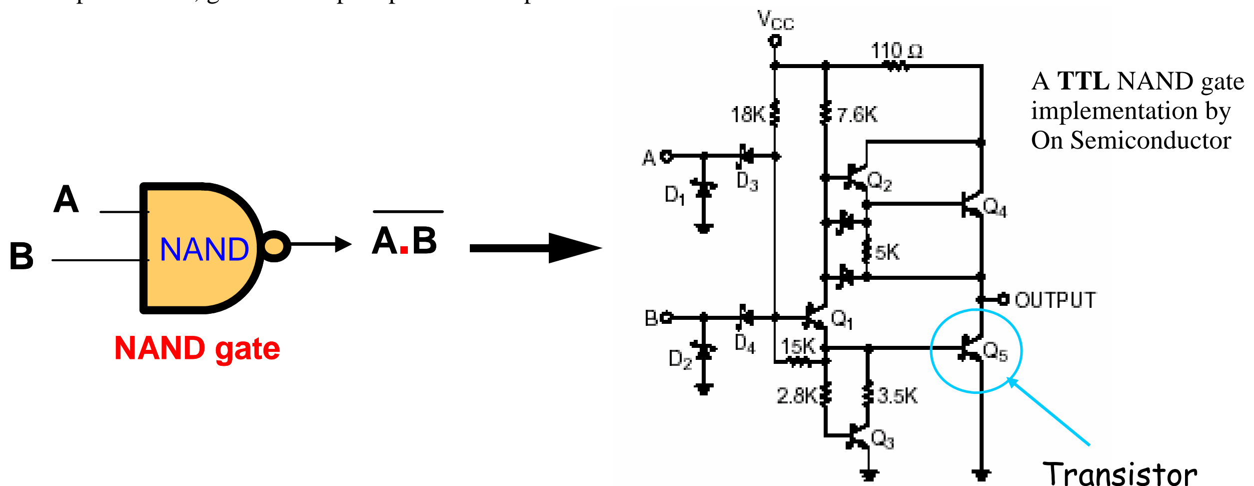
The D input is used only when there is a negative edge on the clock input. Therefore, D line changes do not affect the output all the time which is unlike the gate operation. Note that on the timing diagram, it looks like output A[0] changes at the same time the negative edge occurs. Actually, it changes a few nano seconds after the negative edge. Similarly, if the input seems to change at the same time there is a negative edge, the value stored is the value right before the negative edge. For example, if OBUS[0] changed from 1 to 0 at the end of clock period 2, the value that is stored is the value right before the edge which is 1.

2.9. Transistor Layer :

This layer consist of digital electronic circuits. **Digital electronic circuits** are used to build digital circuits. That is, digital electronic circuits implement gates (also flip-flops). Digital electronic circuits consist of transistors, resistors, capacitors, diodes, etc. **Transistors** are the main component and so this level is often called the transistor level. Transistors in these circuits are used as on-off switches. The switches are turned on and off by control inputs. The figure below shows on-off switches and how these switches are used to implement an AND gate as an example.



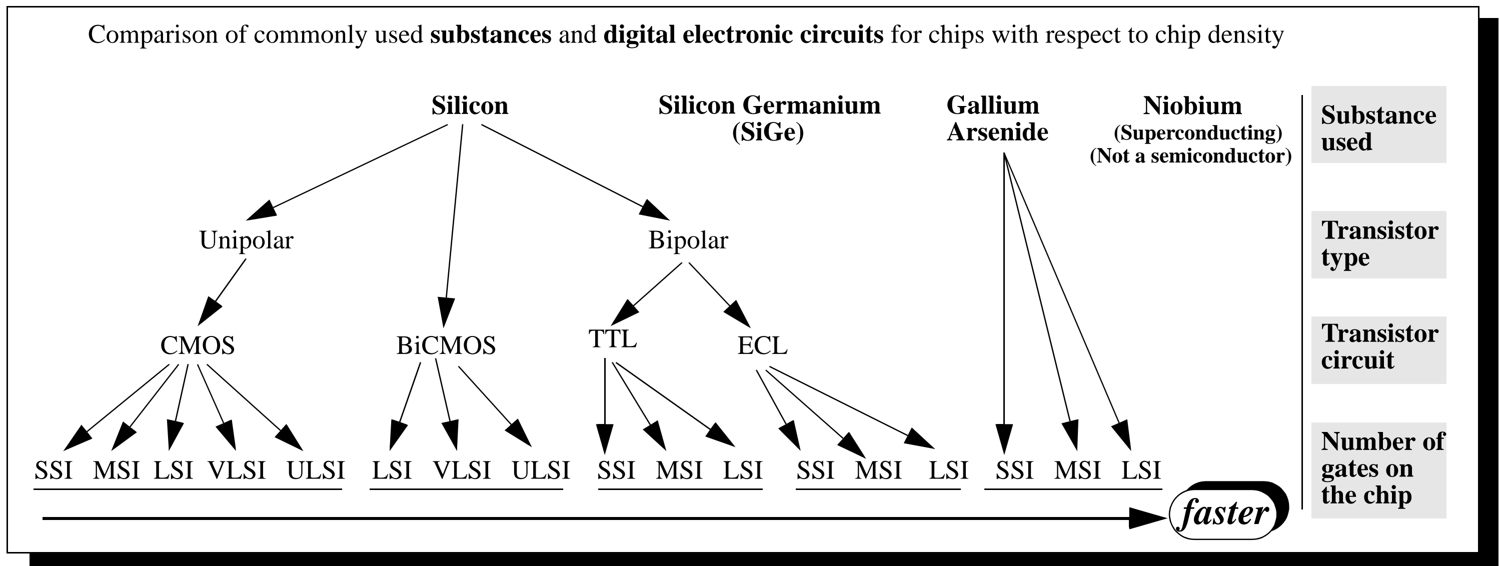
The figure below gives an example of a gate implementation where the gate is a 2-input **TTL** NAND gate. The implementation of the NAND gate by transistors, resistors, etc. is shown next to the gate. TTL is a chip technology and is described below. Today, digital electronic circuits, i.e. circuits with transistors, resistors, capacitors, etc., are on chips. That is, gates and flip-flops are on chips.



We use **semiconductor** substances to implement transistors. That is, today's chips are semiconductor chips. Silicon and Gallium Arsenide are examples of semiconductor substances. Each substance has its own speed, cost, power consumption figures. The most common substance is Silicon which is found in sea sand. This is why Silicon chip prices are so **low**.

Chip design is constrained by design goals : speed, cost, power consumption, size, weight, reliability, etc. Before the design is started, we determine these constraints and then design the product. We try not to exceed the constraints, by

using the right number of gates and flip-flops and right digital electronic implementations. However, it is not easy to satisfy them as they conflict with each other. For example, the higher the speed, the higher the cost and power consumption. Hence, a study of spectrum of choices from semiconductor substances to chip densities is needed. The figure below shows the spectrum of substances and their relative speed for today's digital electronic circuits.



Unipolar/bipolar transistors and other electronic components (resistors, capacitors, diodes,...) are used to implement transistor circuits, such as CMOS, TTL, ECL and BiCMOS. By using a transistor circuit, we implement a single gate. For example, a CMOS AND gate, a TTL AND gate, etc. The reason for using resistors, capacitors, diodes besides transistors for a gate is first for the correct usage of transistors and second to maintain the signal integrity, hence operational stability of the gate.

The number of electronic components on a chip depends on the intended functionality : the more functionality, the more components. A widely used classification of integration of components on chips is given on Table 1 below. The earliest chips from the 1960s were SSI chips and some of them are still used today. The current state of the art microprocessors have more than 1 billion components. The integration level for these high-density chips is beyond ULSI but no new name is agreed upon it yet.

Table 1: Chip densities for various scales of integration

Scales of Integration (chip density)	Siewiorek et al (1982)	Burger et al (1982)
Small Scale Integration (SSI)	**< 10 gates**	**< 64 components**
Medium Scale Integration (MSI)	**< 100 gates**	**< 2K components**
Large Scale Integration (LSI)	**< 10,000 gates**	**< 64K components**
Very Large Scale Integration (VLSI)	**< 100,000 gates**	**< 2M components**
Ultra Large Scale Integration (ULSI)	**> 100,000 gates**	**> 2M components**

Silicon is the most commonly used substance and used by high-speed microprocessors and high-density memory chips. Silicon is expected to be around 10 to 15 years into the future at least. Table 2 below presents the state of the silicon technology. Silicon transistor circuits (CMOS, TTL,...) have different speed, cost, power consumption figures. A brief description of TTL and CMOS circuits is given below. TTL circuits are used for high-speed, low-cost applications while CMOS is for high-density chips, such as microprocessors and memories (DRAM, SRAM). CMOS circuits are also used for portable applications that require low-power consumption (space, embedded applications). Table 3 compares three most commonly used transistor circuits. CMOS is the preferred transistor circuit to implement microprocessors and high-density memory chips. This is because CMOS circuits consume the least

amount of power among the three. TTL is the most widely available and cheapest one, while ECL is the fastest one. Finally, we list a number of properties of the CMOS technology below.

Table 2: The state of the **silicon** technology

Characteristic	Silicon
Densest chip transistor circuit	**CMOS**
Transistors/chip (density)	**1,500,000,000**
Gate delay	**50 - 500 ps**
Process	**32 nanometer**

Table 3: Summary of characteristics for three commonly used IC logic families

Parameter	TTL	CMOS	ECL
Speed	**Medium**	**Low**	**High**
Power consumption	**Medium**	**Low**	**High**
Chip density	**Medium**	**High**	**Low**
Cost	**Low**	**Medium**	**High**

Complementary Metal Oxide Semiconductor (**CMOS**) features

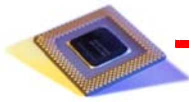
- ➔ CMOS families : 4000 series ; 7400 series : 74HC (High-speed CMOS), 74HCT (High-speed CMOS, TTL compatible), 74AC (Advanced CMOS), 74ACT (Advanced CMOS, TTL compatible), 74FCT (Fast CMOS, TTL compatible), 74FCT-T(Fast CMOS, TTL compatible with TTL V_{OH})
- ➔ Unused gate inputs : do ***not*** leave them unconnected (floating). Tie them to a used input. Also, can connect to 1 or 0 depending on the input characteristic, via a pull-up resistor or pull-down resistor, respectively
- ➔ Gate output circuits :
  - ✦ Regular (do ***not*** short circuit gate outputs)
  - ✦ Tri-state (gate outputs can be short circuited, if only one gate is enabled)
  - ✦ Open-drain (an external pull-up resistor needed. Gate outputs can be short circuited)
- ➔ Electrostatic discharge can damage CMOS chips. Unless properly grounded, one should ***not*** touch CMOS chips

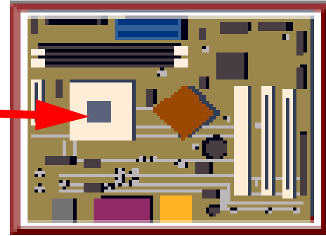
3. The Big Picture : Transistors to Computers

Today, digital electronic circuits (transistor circuits) are on chips. That is, those transistors, resistors, capacitors, etc. are on chips. Chips are on printed circuit boards (PCBs) also known as cards. A PCB can contain tens of chips. The main PCB of a computer is called motherboard which contains the microprocessor and the memory chips. Typically, how many PCBs a non-embedded computer can have in a single cabinet depends on the size of the PCB together with the power and cooling arrangements of the cabinet and the room the cabinet is in. For example, a desktop computer can have two to six PCBs.

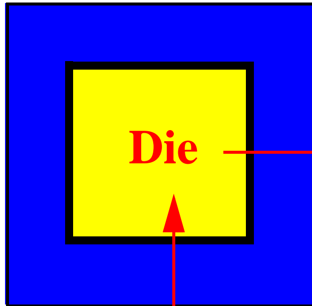
Transistors and electronic components are placed in the center of the chip the area called **die**. That is, the digital circuits implemented by transistors are on the die. Pins (terminals) of the chip allow the components on the die to be accessible from the external world. The die is connected to the pins by means of wires. Dice are placed on a wafer. The number of dice per wafer depends on sizes of the wafer and die. The size of the die depends on the complexity (functionality) of the digital circuit !

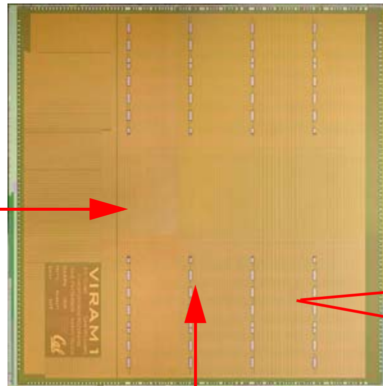
A chip

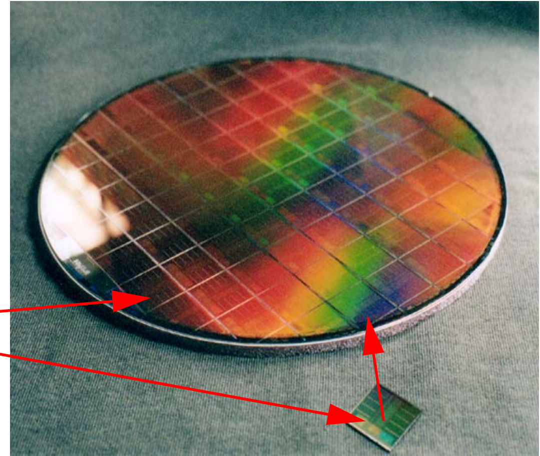




A PCB







Transistors are on the die

The UC Berkeley VIRAM1 **die**

VIRAM1 silicon dice are on the silicon wafer. The wafer contains 72 VIRAM1 dice

Photos by Joseph Gebis : The Berkeley Intelligent RAM (IRAM) Project : http://iram.cs.berkeley.edu

Just as a chip design is constrained by the speed, cost, power consumption, size, weight, reliability, etc., the PCB design is also constrained by the same factors. Before the PCB design is started, we determine these constraints ! Based on them, we go ahead and design the PCB. We keep speed, cost, power consumption, size, weight, etc. of the PCB in mind, by using the right number of chips, chip implementations and wiring. The hierarchy of circuits from chips to the whole system is exemplified by one of the fastest supercomputers, the IBM Blue Gene/L, below.

IBM Deep Computing

Copyright IBM

IBM Research

Blue Gene/L: from chip to system

System
64 Racks, 64x32x32
360 TF/s
32 TB

Rack
32 node cards
5.6 TF/s
512 GB

Node card
(32 chips 4x4x2)
16 compute, 0-2 IO cards
180 GF/s
16 GB

Compute card
2 chips, 1x2x1
11.2 GF/s
1.0 GB

Chip
2 processors
2.8/5.6 GF/s
4 MB

17 V. Salapura and J. Moreira, IBM Blue Gene Tutorial
International Symposium on Computer Architecture ISCA 2006

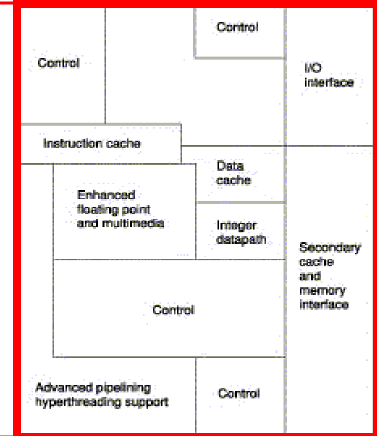
© 2006 IBM Corporation

Permission to use by IBM

A microprocessor chip contains several processors (cores, central processing units, CPUs), cache memories, memory management units (MMUs) and bus interfaces. These are implemented by registers, buses, arithmetic-logic units (ALUs), sequencers and other digital circuits. All of these digital circuits are implemented by gates and flip-flops. Finally, all the gates and flip-flops are implemented by transistor circuits which are on a single die. Therefore, transistor circuits on a die implement a microprocessor. Below the Intel Pentium 4 die is shown.

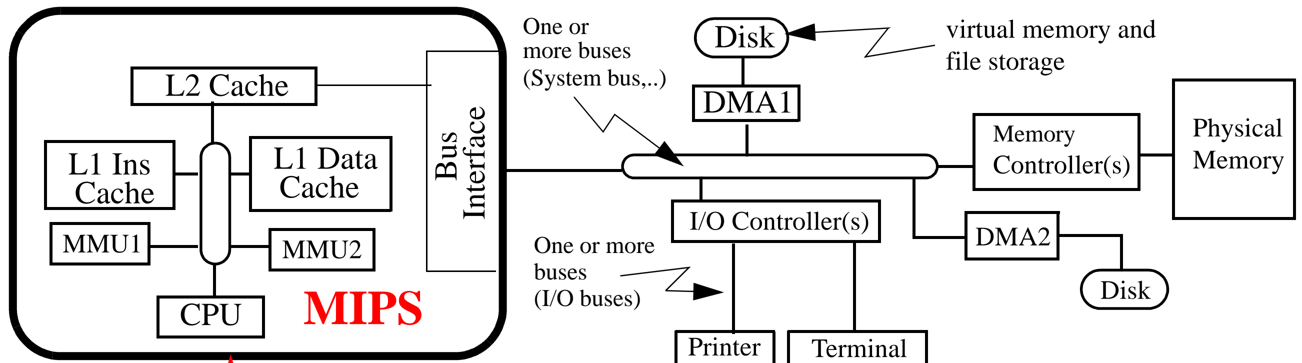
Intel® Pentium® 4 Processor Die on 0.18-micron : http://www.intel.com

Computer Organization and Design The Hardware/Software Interface, David A. Patterson and John L. Hennessy, 3rd edition, Morgan Kaufman, 2005, pp. 21.

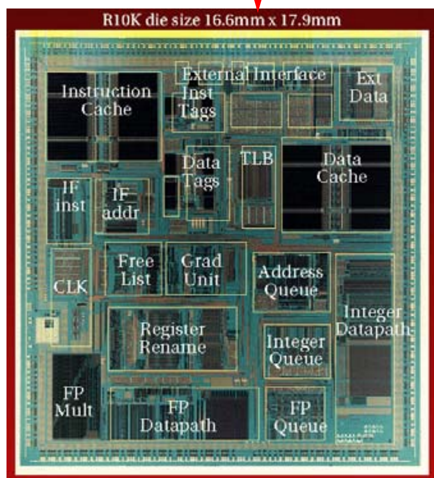


We pack hundreds of millions of transistors on a chip today. The number follows **Moore's Law** : Every two years the number of transistors on a chip doubles. Because we shrink the size of the transistor. What we call the process on Table 2 above is a measure to determine the size of a transistor on a chip. The process is 45 nanometer today and is reduced by one-third every two years, shrinking transistor size. Currently, we have chips with more than one billion transistors. A typical MIPS-based microprocessor organization, the MIPS R10000 die is shown below.

The general computer organization that is also implemented on the MIPS :



The MIPS R10000 die photo http://bwrc.eecs.berkeley.edu/CIC/die_photos/r10k.gif



Power consumption is a major concern today as there are so many transistors on the chip. Power is also related to the clock frequency : **the higher the clock frequency, the higher the power consumption**. When the power consumption is high, the temperature of the chip increases. If a hot chip is not cooled quickly, it will **burn out**. Thus, one has to use **heat sinks**, **fans** or **liquids** to cool the chip. However, cooling adds to the size, weight and cost of the chip and the PCB.

The recent shift in microprocessor design from one processor (core) to multiple processors (cores) on the chip is due to the increased power consumption. Simply put, engineers cannot keep the microprocessor chip at low temperatures with simple cooling techniques when they increased the clock frequency. They have to lower the clock frequency. But, that increases the execution time (CPUtime), meaning slower speeds. The solution to keep the execution time low is by using multiple processors. All the processors execute instructions of the same application, performing more operations per clock period, compensating for the reduced clock rate. Note that a multi-core microprocessor is not a uniprocessor. It is a parallel processing system ! A multi-core chip requires a new CPUtime equation. It cannot use the one given in the textbook. What can it be ?