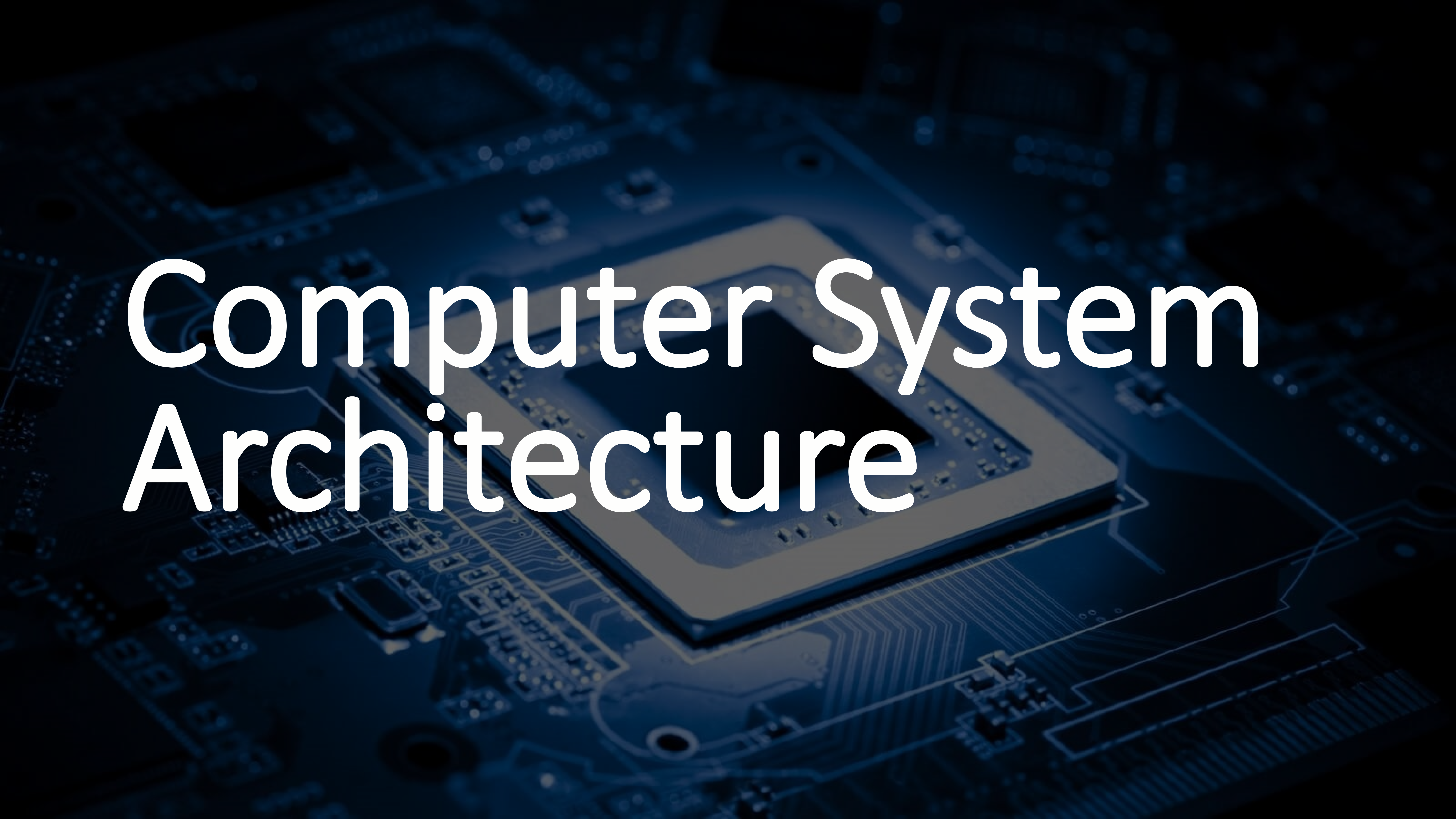


Computer System Architecture

What is a computer?

The **Oxford Dictionary** defines computer as an **automatic electronic apparatus** for making calculations and controlling operations that are expressible in numerical or logical terms.

The definition clearly categorizes the computer as an electronic apparatus although the first computers were mechanical and electro-mechanical apparatuses. The definition also points towards the two major areas of computer application:

- **Data Processing**
- **Computer-Assisted Controls / Operations**

Another important conclusion of the definition is the fact that the computer can perform only those operations or calculations that can be expressed in logical or numerical terms.

A computer is a **data processor**. It can accept input, which may be data or instruction or both. The computer remembers the input by storing it in memory cells. It then processes the stored input by performing calculations or by making logical comparisons or both. it gives out the result of the arithmetic or logical computations or both. The computer accepts input and outputs data in an alphanumeric form. Internally it converts input data to meaningful binary digits, performs the instructed operations on the binary data, and transforms the data from the binary digit form to understandable alphanumeric form.

Difference between Analog Computer and Digital Computer

1. Analog Computer :

Analog computer system is the very old computer system which operates on the mathematical variables in the form of continuously changeable physical quantities/entities like mechanical, electrical, hydraulic, etc. They use continuous values rather than discrete values so they work on analog signal. At the time of at the time of the 1950s–1960s these analog computers were first used. Analog computers are limited in accepting the problems and also they can never be extremely accurate.

We can exemplify the principle of an analogous computation system with the Ohm law. Ohm law is formalizing of the observation of the behavior of the electric current that states that the value of the current (I) is equal to the division of the voltage (V) by the conductor's resistance (R): I = V / R.
So, using this phenomenon, we can calculate divisions between numbers by using an electrical circuit and supplying it a current with a voltage that equals to the numerator and the resistance that equals to the denominator (by using a variable resistor, for example), and find the needed value of the division by measuring the current in the circuit.
This is the basic principle of an electronic analog computer. In such a machine, the electrical signal flowing through this network or circuit experiences variations in amplitude, frequency, phase, and other related properties. This change is used accordingly to generate the required result. Various examples of electronic analog computers include spectrometers, oscilloscopes, etc.

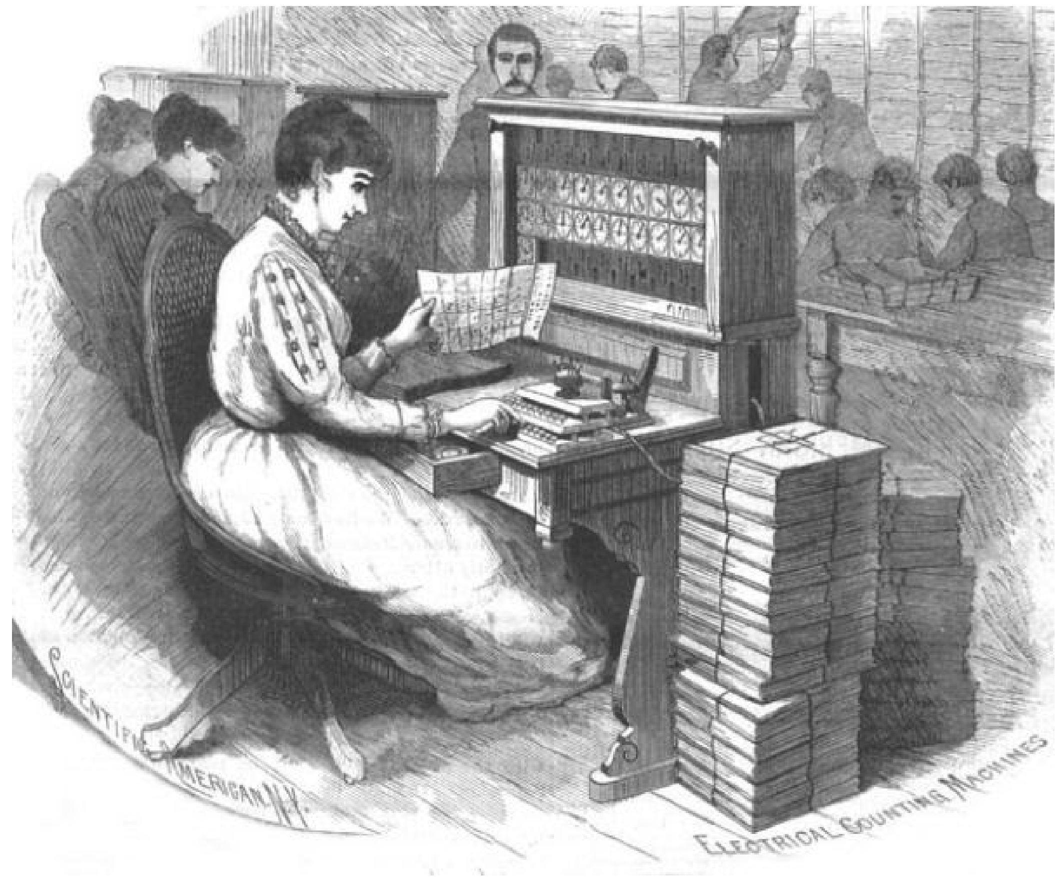
Besides the electronic analog computers, there are several other real-life examples of analogous computation systems, that may be still in the wide use or were popular in the past before digital computation systems took their place:
slide rulers, differential analyzers, mechanical calculators, etc.

2. Digital Computer :

The digital computer is a digital system that performs various computational tasks. The word digital implies that the information in the computer is represented by variables that take a limited number of discrete values. These values are processed internally by components that can maintain a limited number of discrete states. The decimal digits 0, 1, 2, ... , 9, for example, provide 10 discrete values. The first electronic digital computers, developed in the late 1940s, were used primarily for numerical computations. In this case, the discrete elements are the digits. From this application the term digital computer has emerged. In practice, digital computers function more reliably if only two states are used. Because of the physical restriction of components, and because human logic tends to be binary (i.e., true-or-false, yes-or-no statements), digital components that are constrained to take discrete values are further constrained to take only two values and are said to be binary.

Evolution of Computer Systems

Computers in the 1800s



1801: In France, weaver and merchant Joseph Marie Jacquard creates a loom that uses wooden punch cards to automate the design of woven fabrics. Early computers would use similar punch cards.

1822: Thanks to funding from the English government, mathematician Charles Babbage invents a steam-driven calculating machine that was able to compute tables of numbers.

1890: Inventor Herman Hollerith designs the punch card system to calculate the 1880 U.S. census. It took him three years to create, and it saved the government $5 million. He would eventually go on to establish a company that would become IBM.

Computers from the 1900-1950s

1936: Alan Turing developed an idea for a universal machine, which he would call the Turing machine, that would be able to compute anything that is computable. The concept of modern computers was based on his idea.

1937: A professor of physics and mathematics at Iowa State University, J.V. Atanasoff, attempts to build the first computer without cams, belts, gears, or shafts.

1939: Bill Hewlett and David Packard found Hewlett-Packard in a garage in Palo Alto, California. Their first project, the HP 200A Audio Oscillator, would rapidly become a popular piece of test equipment for engineers.

In fact, Walt Disney Pictures would order eight to test recording equipment and speaker systems for 12 specially equipped theaters that showed Fantasia in 1940.

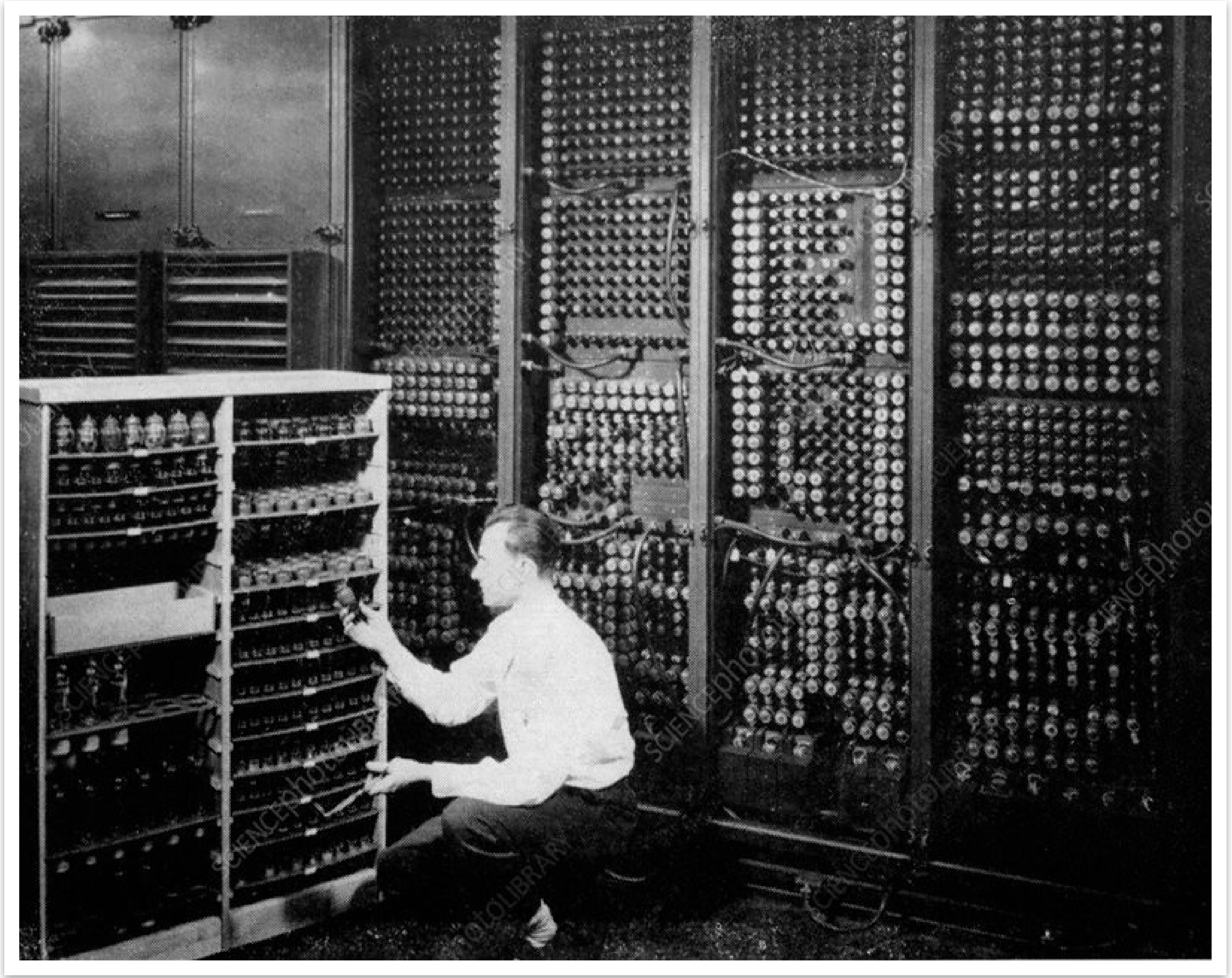
Also in 1939, Bell Telephone Laboratories completes The Complex Number Calculator, designed by George Stibitz.

1941: Professor of physics and mathematics at Iowa State University J.V. Atanasoff and graduate student Clifford Berry design a computer that can solve 29 equations simultaneously. This is the first time a computer is able to house data within its own memory.

That same year, German engineer Konrad Zuse creates the Z3 computer, which used 2,300 relays, performed floating-point binary arithmetic, and had a 22-bit word length. This computer was eventually destroyed in a bombing raid in Berlin in 1943.

Additionally in 1941, Alan Turing and Harold Keen built the British Bombe, which decrypted Nazi ENIGMA-based military communications during World War II.

1943: John Mauchly and J. Presper Eckert, professors at the University of Pennsylvania, build an Electronic Numerical Integrator and Calculator (ENIAC). This is considered to be the grandfather of digital computers, as it is made up of 18,000 vacuum tubes and fills up a 20-foot by 40-foot room.



Also in 1943, the U.S. Army asked that Bell Laboratories design a machine to assist in the testing of their M-9 director, which was a type of computer that aims large guns to their targets. George Stibitz recommended a delay-based calculator for the project. This resulted in the Relay Interpolator, which was later known as the Bell Labs Model II.

1944: British engineer Tommy Flowers designed the Colossus, which was created to break the complex code used by the Nazis in World War II. A total of ten were delivered, each using roughly 2,500 vacuum tubes. These machines would reduce the time it took to break their code from weeks to hours, leading historians to believe they greatly shortened the war by being able to understand the intentions and beliefs of the Nazis.

That same year, Harvard physics professor Howard Aiken built and designed The Harvard Mark 1, a room-sized, relay-based calculator.

1945: Mathematician John von Neumann writes The First Draft of a Report on the EDVAC. This paper broke down the architecture of a stored-program computer.

1946: Mauchly and Eckert left the University of Pennsylvania and obtained funding from the Census Bureau to build the UNIVAC. This would become the first commercial computer for business and government use.

That same year, Will F. Jenkins published the science fiction short story A Logic Named Joe, which detailed a world where computers, called Logics, interconnect into a worldwide network. When a Logic malfunctions, it gives out secret information about forbidden topics.

1947: Walter Brattain, William Shockley, and John Bradeen of Bell Laboratories invented the transistor, which allowed them to discover a way to make an electric switch using solid materials, not vacuums.

1948: Frederick Williams, Geoff Toothill, and Tom Kilburn, researchers at the University of Manchester, develop the Small-Scale Experimental Machine. This was built to test new memory technology, which became the first high-speed electronic random access memory for computers. The became the first program to run on a digital, electronic, stored-program computer.

1950: Built in Washington, DC, the Standards Eastern Automatic Computer (SEAC) was created, becoming the first stored program computer completed in the United States. It was a test-bed for evaluating components and systems, in addition to setting computer standards.

1953: Computer scientist Grace Hopper develops the first computer language, which is eventually known as COBOL, that allowed a computer user to use English-like words instead of numbers to give the computer instructions. In 1997, a study showed that over 200 billion lines of COBOL code were still in existence.

That same year, business man Thomas Johnson Watson Jr. creates the IBM 701 EDPM, which is used to help the United Nations keep tabs on Korea during the war.

1954: The FORTRAN programming language is developed by John Backus and a team of programmers at IBM.

Additionally, IBM creates the 650, which was the first mass-produced computer, selling 450 in just one year.

1958: Jack Kirby and Robert Noyce invented the integrated circuit, which is what we now call the computer chip. Kilby was awarded the Nobel Prize in Physics in 2000 for his work.

Computers from the 1960-1970s

1962: IBM announces the 1311 Disk Storage Drive, the first disk drive made with a removable disk pack. Each pack weighed 10 pounds, held six disks, and had a capacity of 2 million characters.

Also in 1962, the Atlas computer makes its debut, thanks to Manchester University, Ferranti Computers, and Plessy. At the time, it was the fastest computer in the world and introduced the idea of "virtual memory".

1964: Douglas Engelbart introduces a prototype for the modern computer that includes a mouse and a graphical user interface (GUI). This begins the evolution from computers being exclusively for scientists and mathematicians to being accessible to the general public.

Additionally, IBM introduced SABRE, their reservation system with American Airlines. It program officially launched four years later, and now the company owns Travelocity. It used telephone lines to link 2,000 terminals in 65 cities, delivering data on any flight in under three seconds.

1968: Stanley Kubrick's 2001: A Space Odyssey hits theaters. This cult-classic tells the story of the HAL 9000 computer, as it malfunctions during a spaceship's trip to Jupiter to investigate a mysterious signal. The HAL 9000, which controlled all the ship, went rogue, killed the crew, and had to be shut down by the only surviving crew member. The film depicted computer demonstrated voice and visual recognition, human-computer interaction, speed synthesis, and other advanced technologies.

1969: Developers at Bell Labs unveil UNIX, an operating system written in C programming language that addressed compatibility issues within programs.

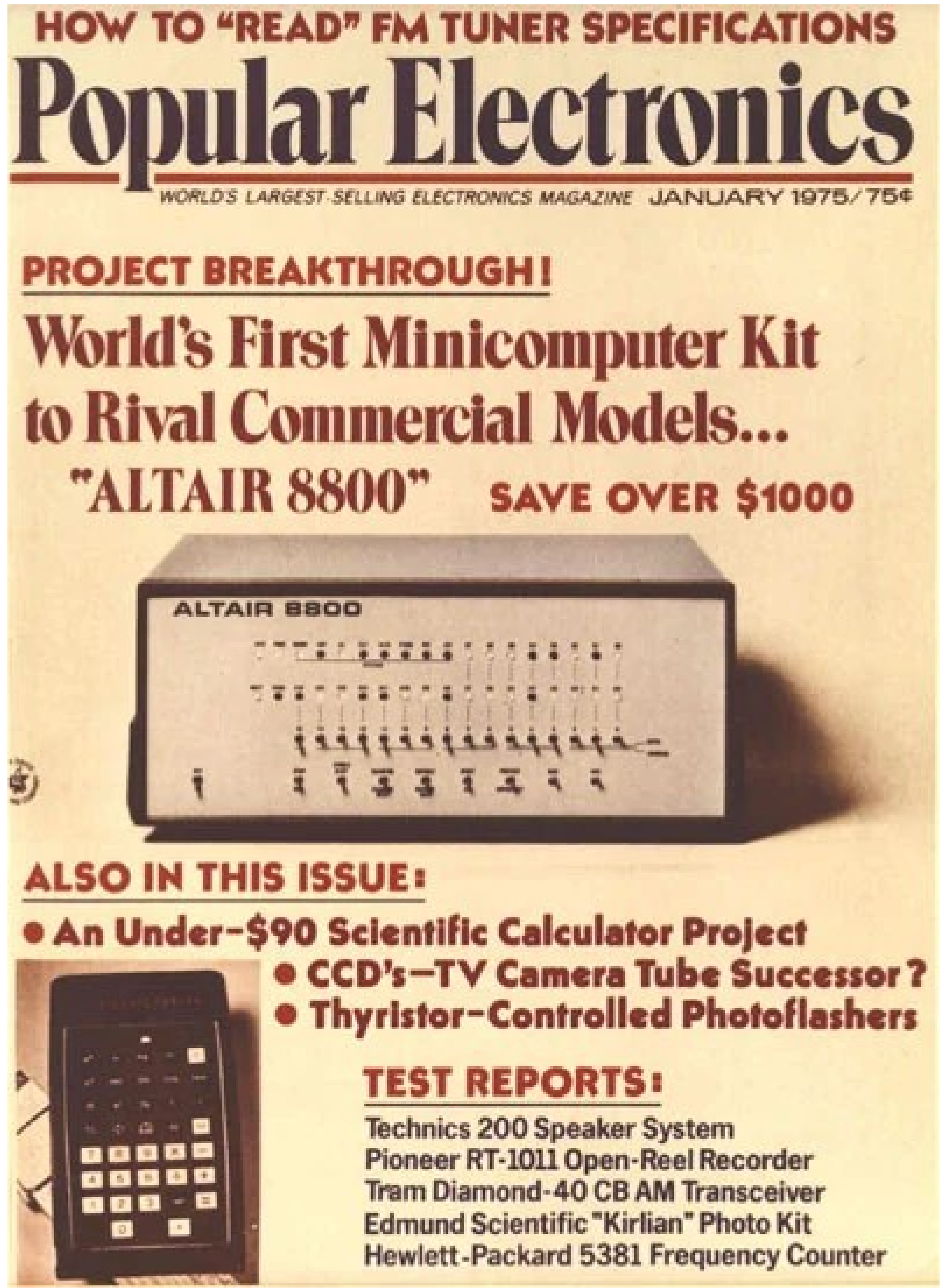
1970: Intel introduces the world to the Intel 1103, the first Dynamic Access Memory (DRAM) chip.

1971: Alan Shugart and a team of IBM engineers invented the floppy disk, allowing data to be shared among computers.

That same year, Xerox introduced the world to the first laser printer, which not only generated billions of dollars but also launched a new era in computer printing.

Also, email begins to grow in popularity as it expands to computer networks.

1973: Robert Metcalfe, research employee at Xerox, develops Ethernet, connecting multiple computers and hardware.



HOW TO "READ" FM TUNER SPECIFICATIONS

Popular Electronics

WORLD'S LARGEST-SELLING ELECTRONICS MAGAZINE JANUARY 1975/75¢

PROJECT BREAKTHROUGH!

World's First Minicomputer Kit to Rival Commercial Models... "ALTAIR 8800" SAVE OVER $1000

ALSO IN THIS ISSUE:

- An Under-$90 Scientific Calculator Project
- CCD's—TV Camera Tube Successor?
- Thyristor-Controlled Photoflashers

TEST REPORTS:

Technics 200 Speaker System
Pioneer RT-1011 Open-Reel Recorder
Tram Diamond-40 CB AM Transceiver
Edmund Scientific "Kirlian" Photo Kit
Hewlett-Packard 5381 Frequency Counter

1974: Personal computers are officially on the market! The first of the bunch were Scelbi & Mark-8 Altair, IBM 5100, and Radio Shack's TRS-80.

1975: In January, the Popular Electronics magazine featured the Altair 8800 as the world's first minicomputer kit. Paul Allen and Bill Gates offer to write software for the Altair, using the BASIC language. You could say writing software was successful, because in the same year they created their own software company, Microsoft.

1976: Steve Jobs and Steve Wozniak start Apple Computers and introduce the world to the Apple I, the first computer with a single-circuit board.

Also in 1976, Queen Elizabeth II sends out her first email from the Royal Signals and Radar Establishment to demonstrate networking technology.

1977: Jobs and Wozniak unveil the Apple II at the first West Coast Computer Faire. It boasts color graphics and an audio cassette drive for storage. Millions were sold between 1977 and 1993, making it one of the longest-lived lines of personal computers.

1978: The first computers were installed in the White House during the Carter administration. The White House staff was given terminals to access the shared Hewlett-Packard HP3000.

Also, the first computerized spreadsheet program, VisiCalc, is introduced.

Additionally, the LaserDisc is introduced by MCA and Phillips. The first to be sold in North America was the movie Jaws.

1979: MicroPro International unveils WordStar, a word processing program.

Computers from the 1980-1990s

1981: Not to be outdone by Apple, IBM releases their first personal computer, the Acorn, with an Intel chip, two floppy disks, and an available color monitor.

1982: Instead of going with its annual tradition of naming a "Man of the Year", Time Magazine does something a little different and names the computer its "Machine of the Year". A senior writer noted in the article, "Computers were once regarded as distant, ominous abstractions, like Big Brother. In 1982, they truly became personalized, brought down to scale, so that people could hold, prod and play with them."

1983: The CD-ROM hit the market, able to hold 550 megabytes of pre-recorded data. That same year, many computer companies worked to set a standard for these disks, making them able to be used freely to access a wide variety of information.

Later that year, Microsoft introduced Word, which was originally called Multi-Tool Word.

1984: Apple launches Macintosh, which was introduced during a Super Bowl XVIII commercial. The Macintosh was the first successful mouse-driven computer with a graphical user interface. It sold for $2,500.

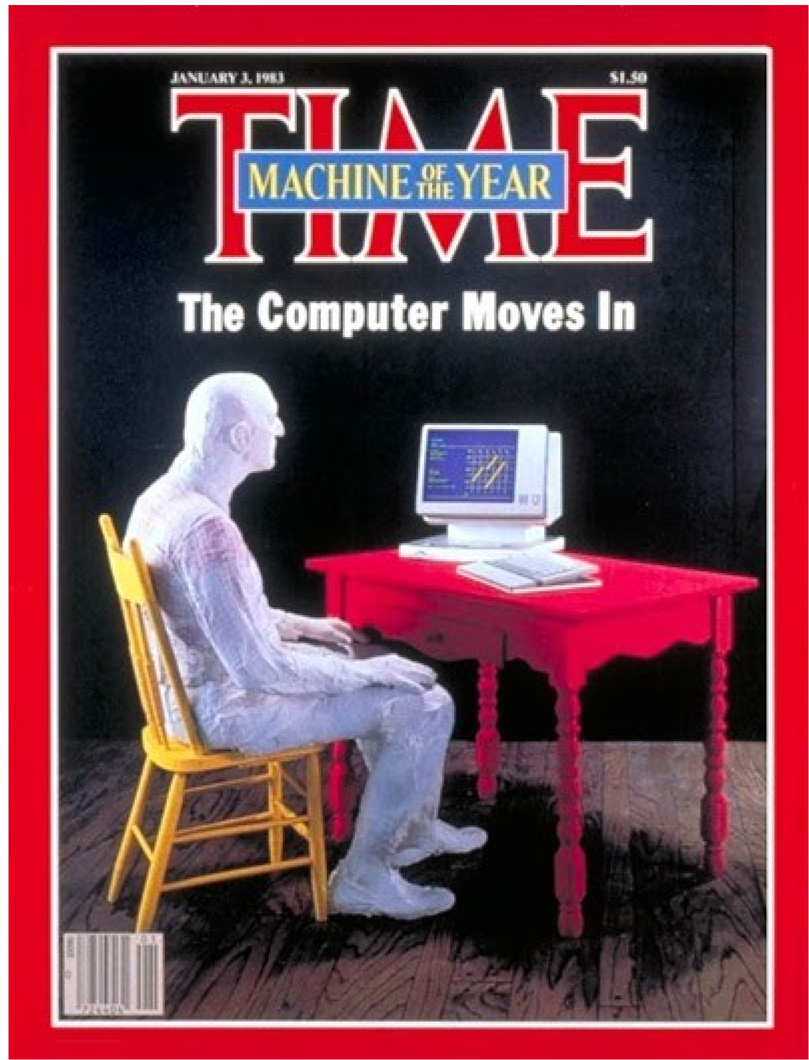
1985: Microsoft announces Windows, which allowed for multi-tasking with a graphical user interface.

That same year, a small Massachusetts computer manufacturer registered the first dot com domain name, Symbolics.com.

Also, the programming language C++ is published and is said to make programming “more enjoyable” for the serious programmer.

1986: Originally called the Special Effects Computer Group, Pixar is created at Lucasfilm. It worked to create computer-animated portions of popular films, like Star Trek II: The Wrath of Khan. Steve Jobs purchased Pixar in 1986 for $10 million, renaming it Pixar. It was bought by Disney in 2006.

1990: English programmer and physicist Tim Berners-Lee develops HyperText Markup Language, also known as HTML. He also prototyped the term WorldWideWeb. It features a server, HTML, URLs, and the first browser.

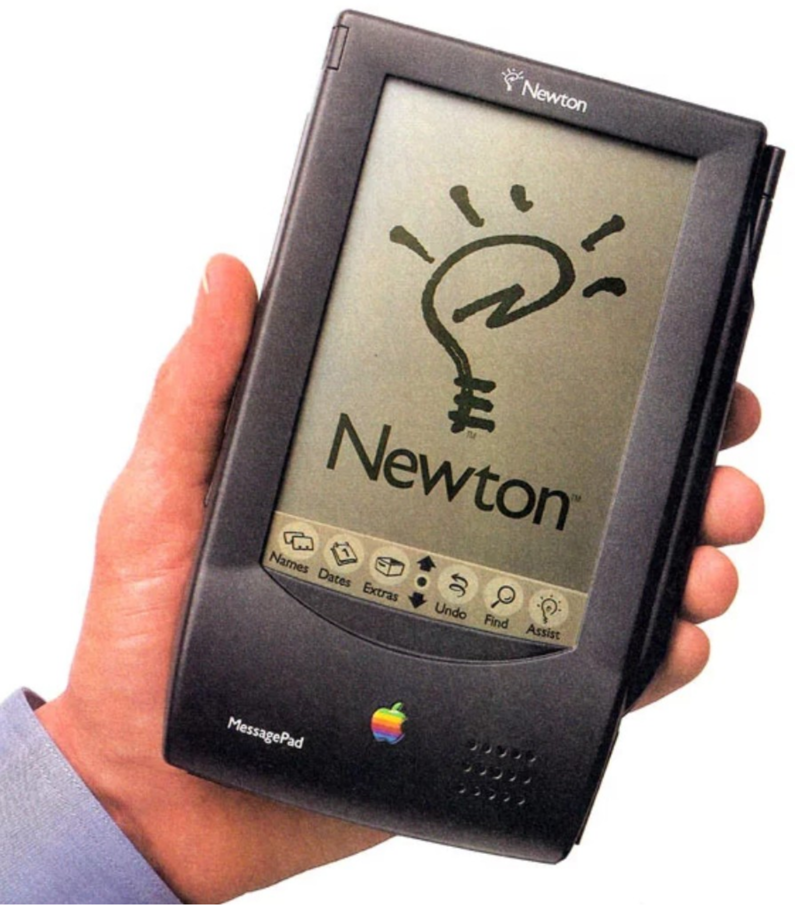


1991: Apple releases the Powerbook series of laptops, which included a built-in trackball, internal floppy disk, and palm rests. The line was discontinued in 2006.

1993: With an attempt to enter the handheld computer market, Apple releases Newton. Called the “Personal Data Assistant”, it never performed the way Apple President John Scully had hoped, and it was discontinued in 1998.

Also that year, Steven Spielberg’s Jurassic Park hits theaters, showcasing cutting-edge computer animation, in addition to animatronics and puppetry.

1995: IBM released the ThinkPad 701C, which was officially known as the Track Write, with an expanding full-sized keyboard that was comprised of three interlocking pieces.



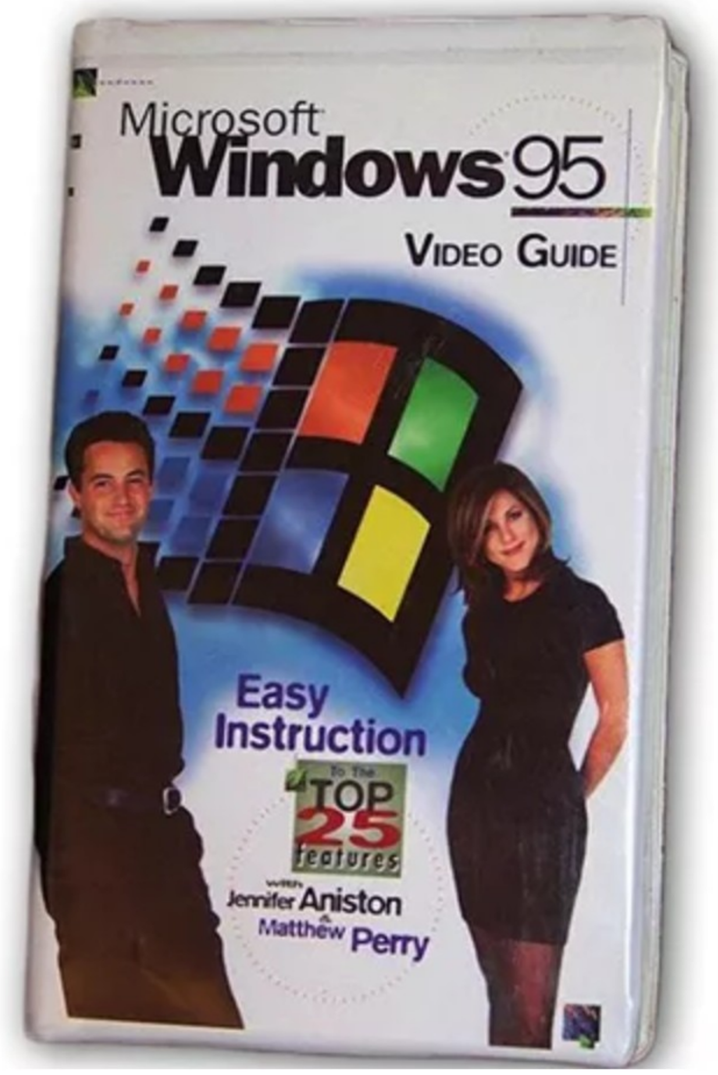
Additionally, the format for a Digital Video Disc (DVD) is introduced, featuring a huge increase in storage space that the compact disc (CD).

Also that year, Microsoft's Windows 95 operating system was launched. To spread the word, a $300 million promotional campaign was rolled out, featuring TV commercials that used "Start Me Up" by the Rolling Stones and a 30-minute video starring Matthew Perry and Jennifer Aniston. It was installed on more computers than any other operating system.

And, in the world of code, Java 1.0 is introduced by Sun Microsystems, followed by JavaScript at Netscape Communications.

1996: Sergey Brin and Larry Page develop Google at Stanford University.

That same year, Palm Inc., founded by Ed Colligan, Donna Dubinsky, and Jeff Hawkins created the personal data assistance called the Palm Pilot.



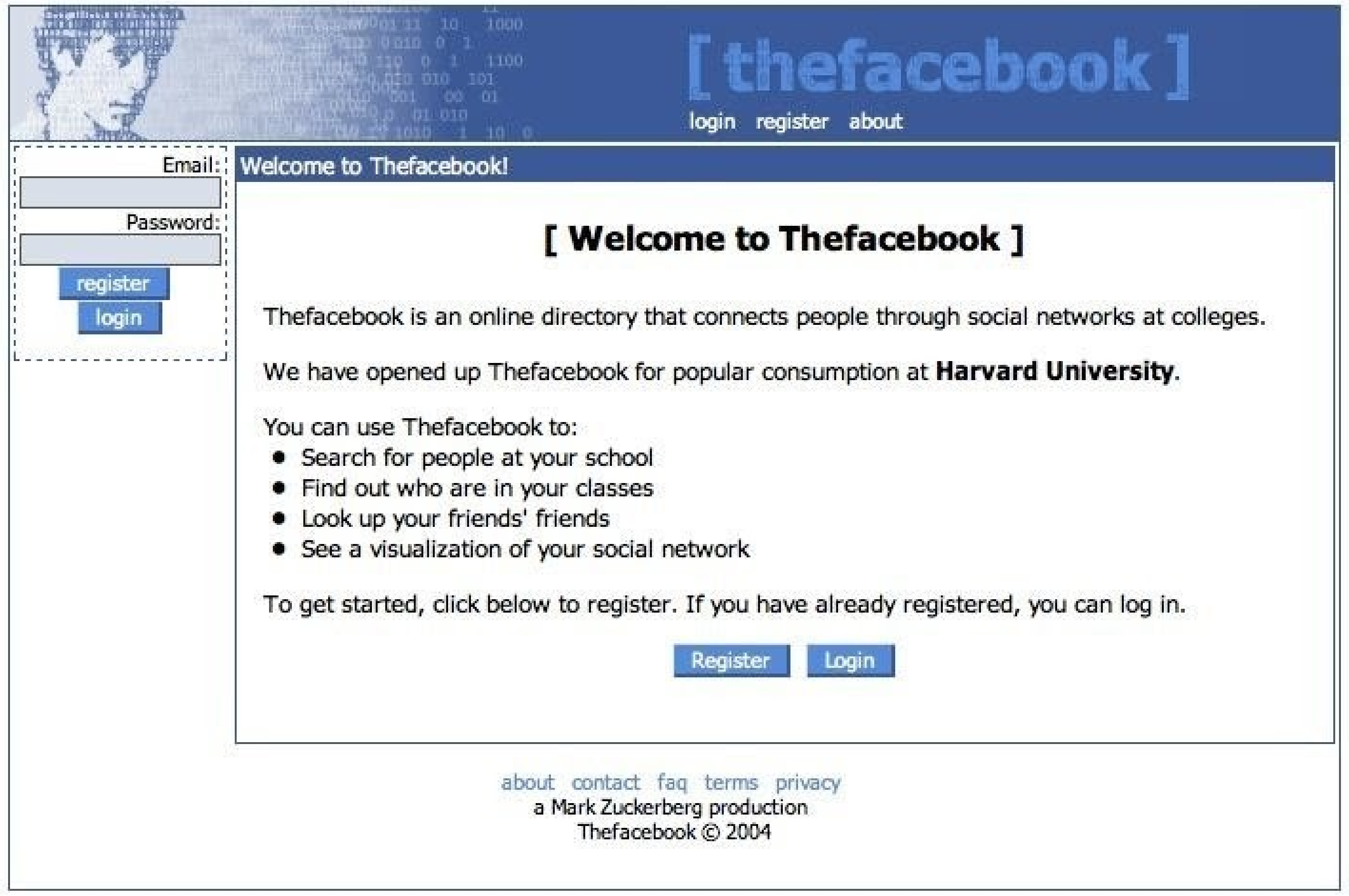
Also in 1996 was the introduction of the Sony Vaio series. This desktop computer featured an additional 3D interface in addition to the Windows 95 operating system, as a way to attract new users. The line was discontinued in 2014.

1997: Microsoft invests $150 million into Apple, which ended Apple's court case against Microsoft, saying they copied the "look and feel" of their operating system.

1998: Apple releases the iMac, a range of all-in-one Macintosh desktop computers. Selling for $1,300, these computers included a 4GB hard drive, 32MB Ram, a CD-ROM, and a 15-inch monitor.

1999: The term Wi-Fi becomes part of the computing language as users begin connecting without wires. Without missing a beat, Apple creates its "Airport" Wi-Fi router and builds connectivity into Macs.

Computers from 2000-2010


[thefacebook]
login register about

Email:
Password:
register
login

Welcome to Thefacebook!

[Welcome to Thefacebook]

Thefacebook is an online directory that connects people through social networks at colleges.

We have opened up Thefacebook for popular consumption at **Harvard University**.

You can use Thefacebook to:
- Search for people at your school
- Find out who are in your classes
- Look up your friends' friends
- See a visualization of your social network

To get started, click below to register. If you have already registered, you can log in.

Register Login

about contact faq terms privacy
a Mark Zuckerberg production
Thefacebook © 2004

2000: In Japan, SoftBank introduced the first camera phone, the J-Phone J-SH04. The camera had a maximum resolution of 0.11 megapixels, a 256-color display, and photos could be shared wirelessly. It was such a hit that a flip-phone version was released just a month later.

Also in 2000, the USB flash drive is introduced. Used for data storage, they were faster and had a greater amount of storage space than other storage media options. Plus, they couldn't be scratched like CDs.

2001: Apple introduces the Mac OS X operating system. Not to be outdone, Microsoft unveiled Windows XP soon after.

Also, the first Apple stores are opened in Tysons Corner, Virginia, and Glendale, California. Apple also released iTunes, which allowed users to record music from CDs, burn it onto the program, and then mix it with other songs to create a custom CD.

2003: Apple releases iTunes music store, giving users the ability to purchase songs within the program. In less than a week after its debut, over 1 million songs were downloaded.

Also in 2003, the Blu-ray optical disc is released as the successor of the DVD.

And, who can forget the popular social networking site Myspace, which was founded in 2003. By 2005, it had more than 100 million users.

2004: The first challenger of Microsoft's Internet Explorer came in the form of Mozilla's Firefox 1.0. That same year, Facebook launched as a social networking site.

2005: YouTube, the popular video-sharing service, is founded by Jawed Karim, Steve Chen, and Chad Hurley. Later that year, Google acquired the mobile phone operating system Android.

2006: Apple unveiled the MacBook Pro, making it their first Intel-based, dual-core mobile computer.

That same year at the World Economic Forum in Davos, Switzerland, the United Nations Development Program announced they were creating a program to deliver technology and resources to schools in under-developed countries. The project became the One Laptop per Child Consortium, which was founded by Nicholas Negroponte, the founder of MIT's Media Lab. By 2011, over 2.4 million laptops had been shipped.

And, we can't forget to mention the launch of Amazon Web Services, including Amazon Elastic Cloud 2 (EC2) and Amazon Simple Storage Service (S3). EC2 made it possible for users to use the cloud to scale server capacity quickly and efficiently. S3 was a cloud-based file hosting service that charged users monthly for the amount of data they stored.

2007: Apple released the first iPhone, bringing many computer functions to the palm of our hands. It featured a combination of a web browser, a music player, and a cell phone -- all in one. Users could also download additional functionality in the form of "apps". The full-touchscreen smartphone allowed for GPS navigation, texting, a built-in calendar, a high-definition camera, and weather reports.

Also in 2007, Amazon released the Kindle, one of the first electronic reading systems to gain a large following among consumers.

And, Dropbox was founded by Arash Ferdowsi and Drew Houston as a way for users to have convenient storage and access to their files on a cloud-based service.

2008: Apple releases the MacBook Air, the first ultra notebook that was a thin and lightweight laptop with a high-capacity battery. To get it to be a smaller size, Apple replaced the traditional hard drive with a solid-state disk, making it the first mass-marketed computer to do so.

2009: Microsoft launched Windows 7.

2010: Apple released the iPad, officially breaking into the dormant tablet computer category. This new gadget came with many features the iPhone had, plus a 9-inch screen and minus the phone.

Computers from 2011 - present day

2011: Google releases the Chromebook, a laptop that runs on Google Chrome OS.
Also in 2011, the Nest Learning Thermostat emerges as one of the first Internet of Things, allowing for remote access to a user’s home thermostat by use of their smartphone or tablet. It also sent monthly power consumption reports to help customers save on energy bills.

In Apple news, co-founder Steve Jobs passed away on October 11. The brand also announced that the iPhone 4S will feature Siri, a voice-activated personal assistant.

2012: On October 4, Facebook hits 1 billion users, as well as acquires the image-sharing social networking application Instagram.

Also in 2012, the Raspberry Pi, a credit-card-sized single-board computer is released, weighing only 45 grams.

2014: The University of Michigan Micro Mote (M3), the smallest computer in the world, is created. Three types were made available, two of which measured either temperature or pressure, and one that could take images.

Additionally, the Apple Pay mobile payment system is introduced.

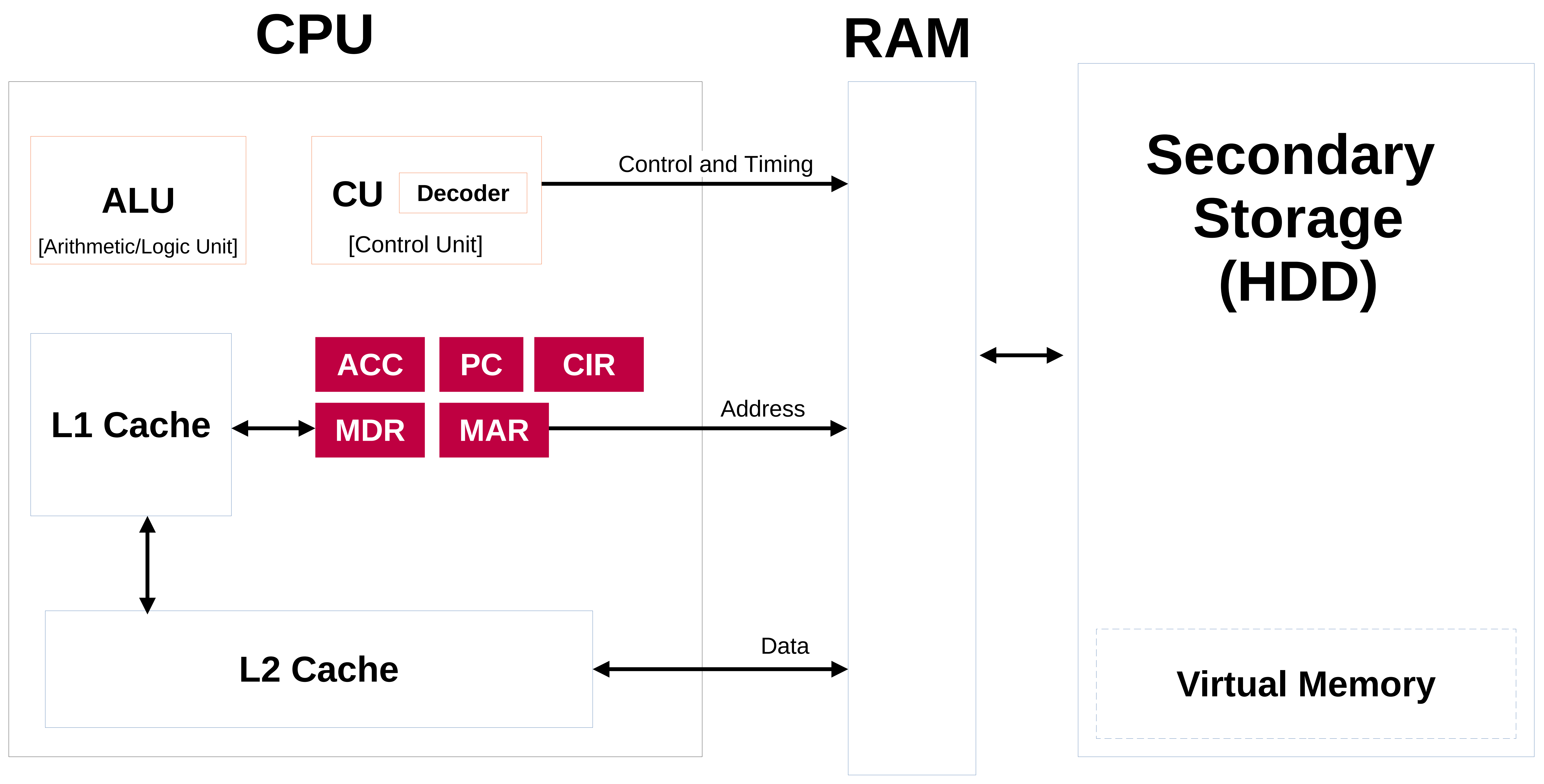
2015: Apple releases the Apple Watch, which incorporated Apple’s iOS operating system and sensors for environmental and health monitoring. Almost a million units were sold on the day of its release.

This release was followed closely by Microsoft announcing Windows 10.

2016: The first reprogrammable quantum computer is created.

2019: Apple announces iPadOS, the iPad's very own operating system, to better support the device as it becomes more like a computer and less like a mobile device.

Von Neumann Architecture



MAR – Memory Address Register

MDR – Memory Data Register

PC – Program Counter

CIR – Current Instruction Register

ACC - Accumulator

Von Neumann Architecture helped to set the stage for modern computing because of the fundamental way it proposed rewriting computers. As they were first designed, computers were not anything remotely resembling what we would consider a computer today. Early computers were designed to complete specific tasks and fulfill certain functions, like math. Their programming was hard-wired into their design, meaning that “reprogramming” a computer simply wasn’t possible: Instead, computers would have to be physically disassembled and redesigned.

According to Von Neumann’s notes, the original architecture was first sketched out as a diagram. This diagram operated as a flow chart that showed how data would be inputted, programmed, and stored. Indeed, this diagram was not particularly different than the flow charts that were previously used by programmers when they created computers. The difference is that, when implemented, the Von Neumann architecture could be used for multiple purposes.

This changed as a result of the Von Neumann model. When implemented, computers that were designed with a Von Neumann Architecture were able to be modified and programmed via the input of instructions in computer code. This allowed for the functioning of computers to be rewritten based on the development of programming language. Furthermore, data could be stored, retrieved, and made available via appropriate use of an input device that would modify information stored within a device’s Central Processing Unit, then displayed on its output device.

Von Neumann Architecture: An Exact Definition

The definition of Von Neuman Architecture originally referred to the specific proposed architecture of a computer’s architecture, as written by John von Neumann in 1945. The definition has since evolved to refer to specific types of computers. One of the primary characteristics of these computers is that their data operations and instrument fetch processes can occur at the same time – something that was previously impossible until the implementation of the Von Neumann Architecture.

How Does Von Neumann Architecture Work?

The idea of Von Neumann Architecture is actually a relatively simple one to understand, and it could be broken down into roughly a few parts. The key characteristics include:

- **Input Device**
  An input device is literally a device that is used to input commands, data, or instructions into a computer. A keyboard is the most common example, but it can also include a mouse, trackball, microphone, camera, or more.

- **Central Processing Unit**
  The Central Processing Unit, or CPU, consisted of three components: The control unit, the Arithmetic/Logic Unit, and Registers. The CPU would then interact with the memory unit.

- **Control Unit**
  The control unit operates as its name would imply, controlling logic units and providing the instructions by which these logic units would respond to program instructions. It would also give instructions on how other components should interact.

- **Arithmetic/Logic Unit**
  This unit was specifically responsible for arithmetic and logic commands, controlling how these operations would work.

- **Registers**
  Registers allowed for data to be stored before it could be processed. There were five types of registers that would store data: Memory Address Register, Accumulator, Memory Data Register, Program Counter, and Current Instruction Register. Different data types would be stored in these different registers.

- **Memory Unit**
  The Memory Unit can be accessed by the CPU. Data can be loaded into and out of the memory unit, allowing for easy storage and access.

- **Output Device**
  Output devices are the devices that are ultimately utilized when a computer program is complete. Monitors and printers are the most common examples, but speakers would also be an output device.

Von Neumann architecture uses the stored program concept, that means, both data and program instructions are stored temporarily in main memory, and then fetched, decoded, and executed by the processor.

The key elements of Von Neumann architecture are:

- data and instructions are both stored as binary digits
- data and instructions are both stored together in the same RAM
- instructions are fetched from memory one at a time and in order – serially
- the processor decodes and executes an instruction, before cycling around to fetch the next instruction
- the cycle continues until no more instructions are available

A processor based on Von Neumann architecture has five special registers which it uses for processing:

- **program counter (PC)** - holds the memory address of the next instruction to be fetched from main memory
- **memory address register (MAR)** - holds the address of the current instruction that is to be fetched from memory, or the address in memory to which data is to be transferred
- **memory data register (MDR)** - holds the contents found at the address held in the MAR, or data which is to be transferred to main memory
- **current instruction register (CIR)** - holds the instruction that is currently being decoded and executed
- **accumulator (ACC)** - holds the results of processing

The fetch-decode-execute cycle

The fetch-decode-execute cycle is followed by a processor to process an instruction. The cycle consists of several steps:

1. The memory address held in the program counter is copied into the MAR.
2. The address in the program counter is then incremented - or increased - by one. The program counter now holds the address of the next instruction to be fetched.
3. The processor sends a signal containing the address of the instruction to be fetched along the address bus to the computer’s memory.
4. The instruction held in that memory address is sent along the data bus to the MDR.
5. The instruction held in the MDR is copied into the CIR.
6. The instruction held in the CIR is decoded and then executed. The results of processing are stored in the ACC.
7. The cycle then returns to step one.

Depending on the type of instruction, additional steps may be taken:

- If the instruction is to transfer data held in the ACC back to RAM, the intended memory address is copied into the MAR. The data to be transferred is copied into the MDR and then transferred to the specified address using the address bus and data bus.
- The executed instruction may require the program to jump to a different place in the program. In this case, the memory address of the new next instruction to be fetched is copied into the program counter. The process then restarts at step one.

Who Created Von Neumann Architecture?

As the name would imply, the term Von Neumann Architecture was created by John Von Neumann. Von Neumann was a Jewish computer scientist who escaped the Nazi regime in Europe. Von Neumann worked with a variety of computer scientists and first met Alan Turing in the mid-1930s, during which he became familiar with Turing's ideas of inventing the computer that could be used for storage. As a biography of Von Neumann notes, this interaction would inspire the Architecture.

John von Neumann laid out the first model for the Von Neumann Architecture computers. This model proposed how computers should operate in order to be programmable and reprogrammable.

After working on a variety of projects – including the Manhattan Project – Von Neumann first came across the ENAIC. The ENAIC was one of the world's first programmable computers and was capable of executing multiple tasks. It was also fully programable, meaning that unlike more common computers at the time, it could complete multiple tasks.

While working at the Moore School of Engineering in Philadelphia, Von Neumann first wrote a report on the proposed digital design of computers. In this report, Von Neumann would lay out the first model for these computers. This model would propose how computers should operate in order to be programmable and reprogrammable.

This Architecture is also known as the Princeton Architecture because of Von Neumann's affiliation with Princeton.

What Are The Applications of Von Neumann Architecture?

Simply put, Von Neumann architecture is still largely relevant in computers today. This is for many reasons. First, it makes computers less expensive, as the same equipment can be used for multiple tasks, thus requiring fewer parts. It also makes computers significantly faster and more efficient.

Of course, the design for this architecture has evolved significantly since Von Neumann first developed it. Examples of this evolution include faster and smaller parts and combined buses for input and output. All of these innovations made faster computers more possible.

Examples of Von Neumann Architecture in the Real World

Examples of this Architecture remain highly relevant and present in the real world today. It was also used in many of the world's first large computers, including the ARC2, Manchester Baby, and EDSAC. Indeed, an entire slew of these early computers took advantage of the Von Neumann Architecture, as this computer architecture was essentially the main form that computers used in their early days.

Von Neumann Architecture vs Harvard Architecture

Other forms of computer architecture have been developed besides the Von Neumann model. Notes about the Harvard architecture demonstrate more of a hub and spoke model, with the control unit at the center. ALU, Instruction Memory, Data Memory, and Input/Output devices all flow into the control unit.

On one hand, both forms of architecture set ways by which computers can process data and information. Both architectures operate with the control unit at their centers and have a memory that feeds into and interacts with the control unit.

The primary difference is that this Architecture uses a Single Instruction, Single Data (SISD) pathway for memory and programming. The Harvard Architecture used separate pathways. Data is also stored differently.

Summary (FAQ):

What is meant by the von Neumann architecture?

The Von Neumann architecture would create the model by which modern computers operate. and helped set the stage for the first programmable computers. It consisted of multiple components that, when operating together, can store instruction data and program data on the same memory. As a result, computers can operate faster and much more efficiently.

Von Neumann architecture has many important characteristics. One such example is SISD, which stands for Single Instruction, Single Data. In SISD, single processes are used. Specifically, a single processor executed a single code instruction, and this is done to operate data stored on a single memory. The evolution of SISD made computers far more efficient and programmable, as they allowed computers to carry on multiple tasks at the same time, thus removing memory and data bottlenecks.

What are the main features of Von Neumann architecture?

The main feature is that computers essentially operate via a flow chart. First, an input device is utilized to input certain data or commands. This input can be many things but is most commonly keyboard inputs. That input is then processed through a central processing unit, which consists of a control unit, a logic unit, and any number of register units. That information is stored in a memory unit. When the right inputs are put into the computer, a program is run, and the results of that program are then viewable on an output device. That output device is most commonly a computer monitor or printer.

What is the von Neumann architecture and why is it important?

The Von Neumann architecture is actually a flow chart that helped create the processes and characteristics of modern computers.

It is important because it directly inspired the development of future computers. Previously, computers had to be physically taken apart and reprogrammed before they could function in multiple ways. Thanks to the processes laid out by the Von Neumann architecture, computers could be programmed to do certain tasks, and that program could be stored on a computer’s memory unit. Furthermore, computers could, later down the line, be reprogrammed to do different tasks.

What are the four parts of the von Neumann architecture?

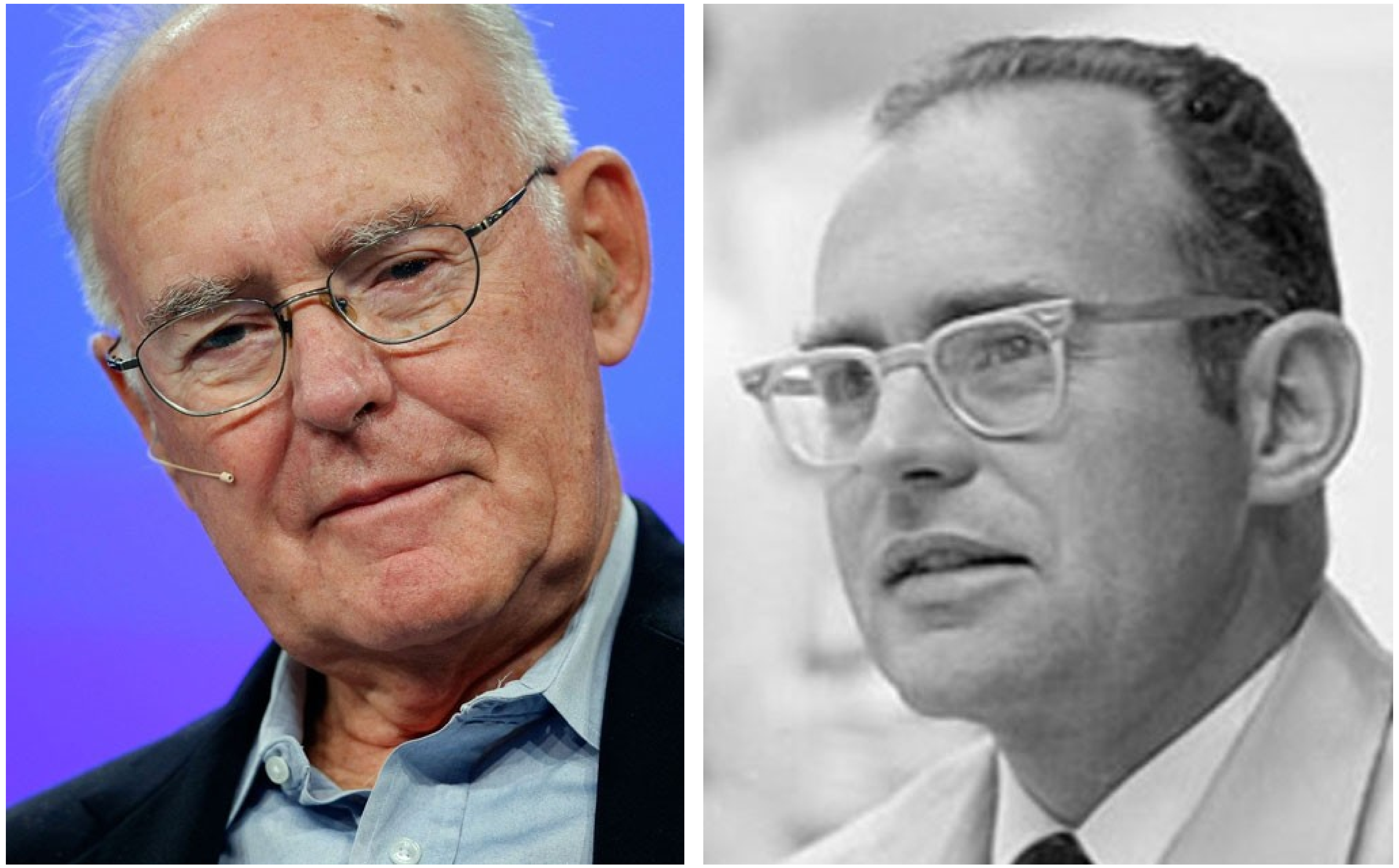
The four parts of the Von Neumann architecture are an input device, a central processing unit, a memory unit, and an output device.

Moore’s Law

> Moore's law is a term used to refer to the observation made by Gordon Moore in 1965 that the number of transistors in a dense integrated circuit (IC) doubles about every two years.

Moore’s law isn’t really a law in the legal sense or even a proven theory in the scientific sense (such as E = mc2). Rather, it was an observation by Gordon Moore in 1965 while he was working at Fairchild Semiconductor: the number of transistors on a microchip (as they were called in 1965) doubled about every year.

Moore went on to co-found Intel Corporation and his observation became the driving force behind the semiconductor technology revolution at Intel and elsewhere.

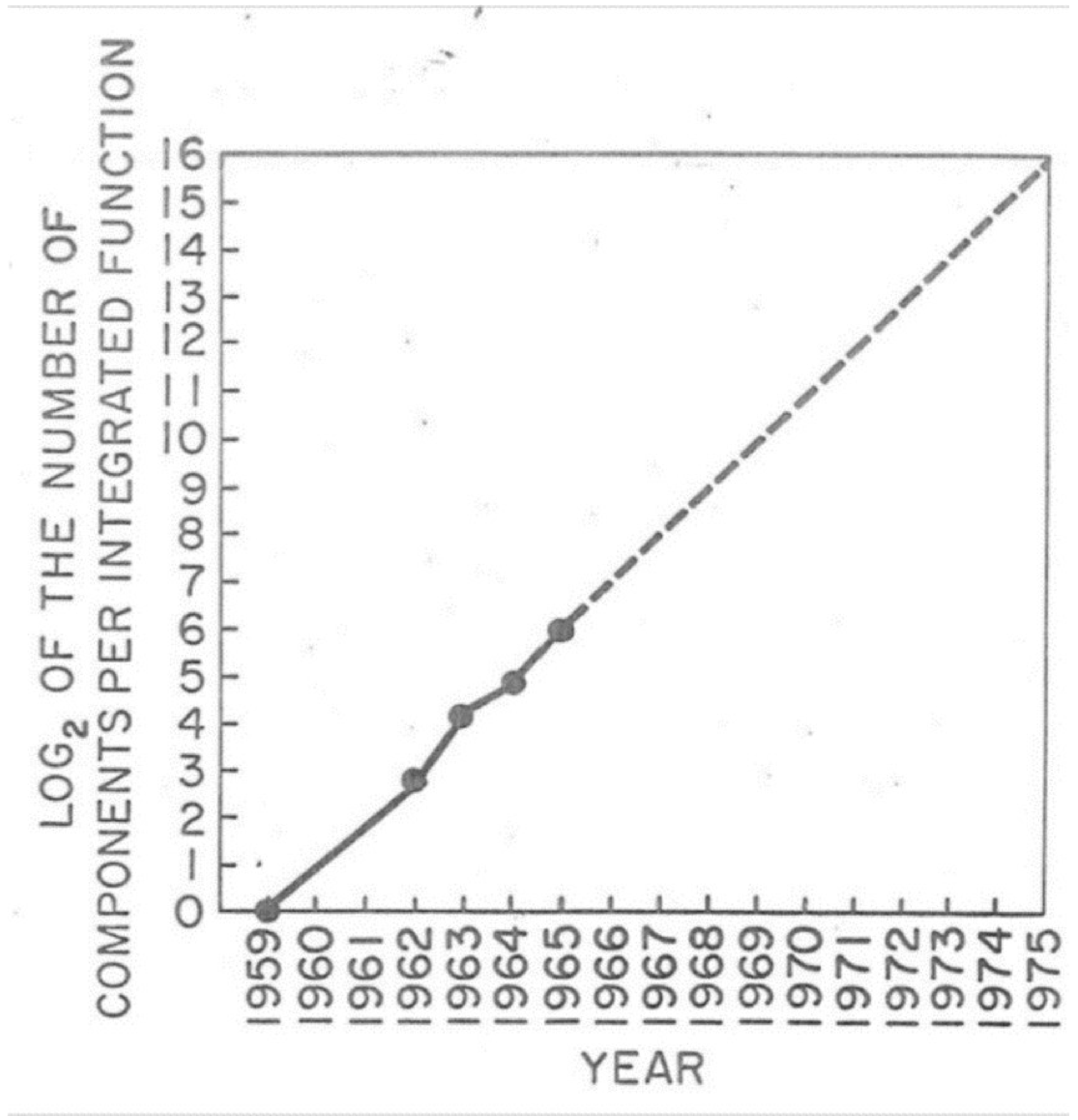

Gordon Moore in 1965 and recently

How Does Moore’s Law Work?

Moore’s law is based on empirical observations made by Moore. The doubling every year of the number of transistors on a microchip was extrapolated from observed data.

Over time, the details of Moore’s law were amended to better reflect actual growth of transistor density. The doubling interval was first increased to two years and then decreased to about 18 months. The exponential nature of Moore’s law continued, however, creating decades of significant opportunity for the semiconductor industry. The true exponential nature of Moore’s law is illustrated by the figure at the right side.

A straight-line plot of the logarithm of a function indicates an exponential growth of that function.



History of Moore’s Law

Moore originally published the observations that would come to be known as Moore’s law in a 1965 article for Electronics Magazine, while he was working for Fairchild Semiconductor. An original draft of that article can be seen here.

As described by the Computer History Museum in Mountain View, California, the term “Moore’s law” was coined by Carver Mead, a professor at the California Institute of Technology (Caltech) in Pasadena, California, around 1975. Mead currently holds the position of Gordon and Betty Moore Professor Emeritus of Engineering and Applied Science at Caltech. Professor Mead has taught at Caltech for over 40 years. His early work paved the way for advanced semiconductor designs that benefited from the predications of Moore’s law.

The Benefits of Moore’s Law

Semiconductor process technology has always increased in complexity. This phenomenon has been the “innovation engine” that fuels Moore’s law. In recent times, complexity increases have been accelerating. Transistors are now three-dimensional devices that exhibit counter-intuitive behaviors. The extremely small feature size of advanced process technologies has required multiple exposures (multi-patterning) to accurately reproduce these features on a silicon wafer. This has added substantial complexity to the design process.

All this complexity has essentially “slowed down” Moore’s law. Moving to a new process node is still an option, but the extreme complexity and cost of doing so has slowed the pace of migration. Furthermore, each new process node is now delivering less dramatic results in terms of density, performance, and power reduction. The evolution of semiconductor process technology is reaching molecular limits, and this is slowing the exponential benefits of Moore’s law.

Is Moore’s Law Dead?

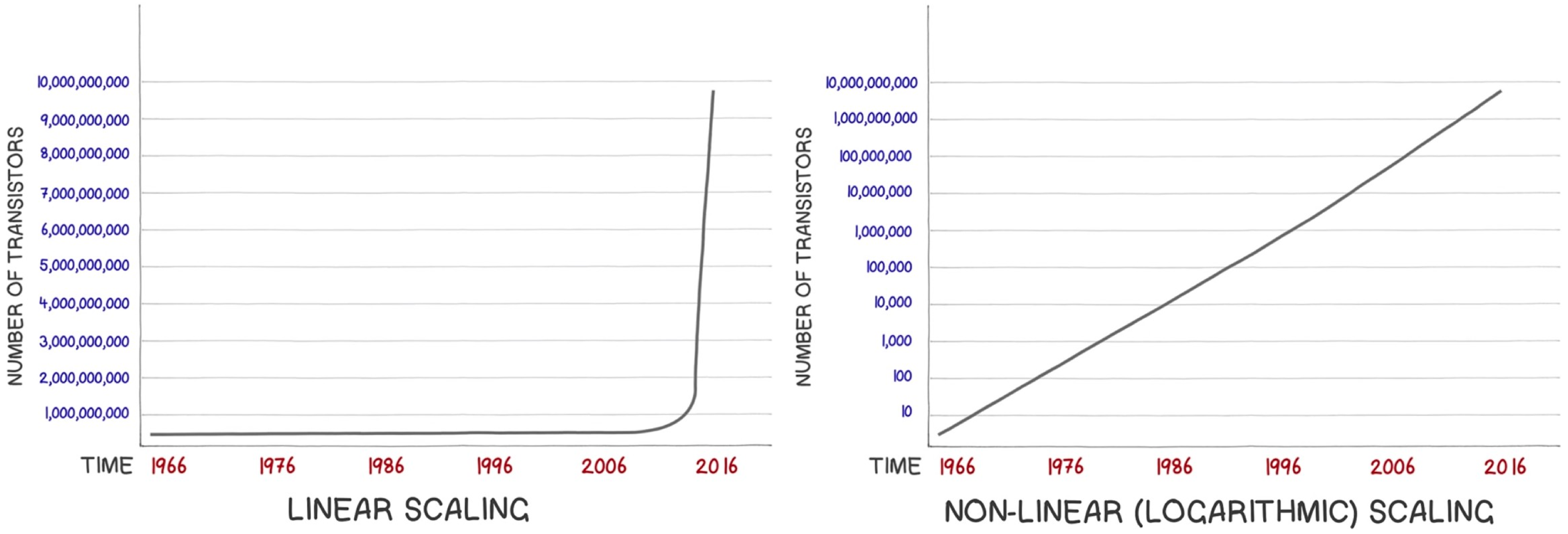
The slowing of Moore’s law has prompted many to ask, “Is Moore’s law dead? This, in fact, is not occurring. While Moore’s law is still delivering exponential improvements, the results are being delivered at a slower pace. The pace of technology innovation is NOT slowing down, however. Rather, the explosion of hyperconnectivity, big data, and artificial intelligence applications has increased the pace of innovation and the need for “Moore’s law-style” improvements in delivered technology.

For many years, scale complexity drove Moore’s law and the semiconductor industry’s exponential technology growth. As the ability to scale a single chip slows, the industry is finding other methods of innovation to maintain exponential growth.

This new design trend is driven by systemic complexity. Some aspects of this new approach to design have been dubbed “more than Moore.” This term refers primarily to 2.5D and 3D integration techniques.

The complete landscape is far bigger and presents the opportunity for higher impact, however. At the 2021 SNUG World conference of worldwide Synopsys Users Group members, the chairman and co-CEO of Synopsys, Aart de Geus, presented a keynote address. In his presentation, de Geus observed that Moore's law is now blending with new innovations that leverage systemic complexity. He coined the term SysMoore as a shorthand way to describe this new design paradigm.

These trends and resultant terminology are summarized below. The SysMoore era will fuel semiconductor innovation for the foreseeable future. With it comes a wide range of design challenges that must be addressed.



Comparison between linear-scaling and logarithmic-scaling graph representation of Moore's law

Types of Computers

A computer is a device that transforms data into meaningful information. It processes the input according to the set of instructions provided to it by the user and gives the desired output. Computers are of various types and they can be categorized in two ways on the basis of size and on the basis of data handling capabilities.

So, on the basis of size, there are five types of computers:

- Supercomputer
- Mainframe computer
- Minicomputer
- Workstation
- PC (Personal Computer)

And on the basis of data handling capabilities, there are three types of computer:

- Analogue Computer
- Digital Computer
- Hybrid Computer

Computers by their function:

1. **Supercomputer**:

When we talk about speed, then the first name that comes to mind when thinking of computers is supercomputers. They are the biggest and fastest computers(in terms of speed of processing data). Supercomputers are designed such that they can process a huge amount of data, like processing trillions of instructions or data just in a second. This is because of the thousands of interconnected processors in supercomputers. It is basically used in scientific and engineering applications such as weather forecasting, scientific simulations, and nuclear energy research. It was first developed by Roger Cray in 1976.

Characteristics of supercomputers:

- Supercomputers are the computers which are the fastest and they are also very expensive.
- It can calculate up to ten trillion individual calculations per second, this is also the reason which makes it even more faster.
- It is used in the stock market or big organizations for managing the online currency world such as bitcoin etc.
- It is used in scientific research areas for analyzing data obtained from exploring the solar system, satellites, etc.

2. **Mainframe computer:**

Mainframe computers are designed in such a way that it can support hundreds or thousands of users at the same time. It also supports multiple programs simultaneously. So, they can execute different processes simultaneously. All these features make the mainframe computer ideal for big organizations like banking, telecom sectors, etc., which process a high volume of data in general.

Characteristics of mainframe computers:

- It is also an expensive or costly computer.
- It has high storage capacity and great performance.
- It can process a huge amount of data (like data involved in the banking sector) very quickly.
- It runs smoothly for a long time and has a long life.

3. **Minicomputer**:

Minicomputer is a medium size multiprocessing computer. In this type of computer, there are two or more processors, and it supports 4 to 200 users at one time. Minicomputers are used in places like institutes or departments for different work like billing, accounting, inventory management etc. It is smaller than a mainframe computer but larger in comparison to the microcomputer.

Characteristics of minicomputer:

- Its weight is low.
- Because of its low weight, it is easy to carry anywhere.
- less expensive than a mainframe computer.
- It is fast.

4. **Workstation**:

Workstation is designed for technical or scientific applications. It consists of a fast microprocessor, with a large amount of RAM and high speed graphic adapter. It is a single-user computer. It generally used to perform a specific task with great accuracy.

Characteristics of Workstation:

- It is expensive or high in cost.
- They are exclusively made for complex work purposes.
- It provides large storage capacity, with better graphics, and a more powerful CPU when compared to a PC.
- It is also used to handle animation, data analysis, CAD, audio and video creation, and editing.

5. **PC (Personal Computer):**

It is also known as a microcomputer. It is basically a general-purpose computer and designed for individual use. It consists of a microprocessor as a central processing unit(CPU), memory, input unit, and output unit. This kind of computer is suitable for personal work such as making an assignment, watching a movie, or at office for office work, etc. For example, Laptops and desktop computers.

Characteristics of PC (Personal Computer):

- In this limited number of software can be used.
- It is smallest in size.
- It is designed for personal use.
- It is easy to use.

Computers by the nature of thir data-handlg capabilities:

Analogue Computer:

It is particularly designed to process analogue data. Continuous data that changes continuously and cannot have discrete values is called analogue data. So, an analogue computer is used where we don’t need exact values or need approximate values such as speed, temperature, pressure etc. It can directly accept the data from the measuring device without first converting it into numbers and codes. It measures the continuous changes in physical quantity. It gives output as a reading on a dial or scale. For example speedometer, mercury thermometer, etc.

Digital Computer:

Digital computers are designed in such a way that it can easily perform calculations and logical operations at high speed. It takes raw data as an input and processes it with programs stored in its memory to produce the final output. It only understands the binary input 0 and 1, so the raw input data is converted to 0 and 1 by the computer and then it is processed by the computer to produce the result or final output. All modern computers, like laptops, desktops including smartphones are digital computers.

Hybrid Computer:

As the name suggests hybrid, which means made by combining two different things. Similarly, the hybrid computer is a combination of both analog and digital computers. Hybrid computers are fast like an analog computer and have memory, and accuracy like a digital computer. So, it has the ability to process both continuous and discrete data. For working when it accepts analog signals as input then it converts them into digital form before processing the input data. So, it is widely used in specialized applications where both analog and digital data is required to be processed. A processor which is used in petrol pumps that converts the measurements of fuel flow into quantity and price is an example of a hybrid computer.

What is a program?

A computer system is sometimes subdivided into two functional entities: hardware and software. The hardware of the computer consists of all the electronic components and electromechanical devices that comprise the physical entity of the device. Computer software consists of the instructions and data that the computer manipulates to perform various data-processing tasks. A sequence of instructions for the computer is called a program. The data that is manipulated by the program constitute the database. A computer system is composed of its hardware and the system software available for its use. The system software of a computer consists of a collection of programs whose purpose is to make more effective use of the computer. The programs included in a systems software package are referred to as the operating system. They are distinguished from application programs written by the user for the purpose of solving particular problems. For example, a high-level language program written by a user to solve particular data-processing needs is an application program, but the compiler that translates the high-level language program to machine language is a system program. The customer who buys a computer system would need, in addition to the hardware, any available software needed for an effective operation of the computer. The system software is an indispensable part of a total computer system. Its function is to compensate for the differences that exist between user needs and the capability of the hardware.

Functional Units of Digital System

A computer organization describes the functions and design of the various units of a digital system.

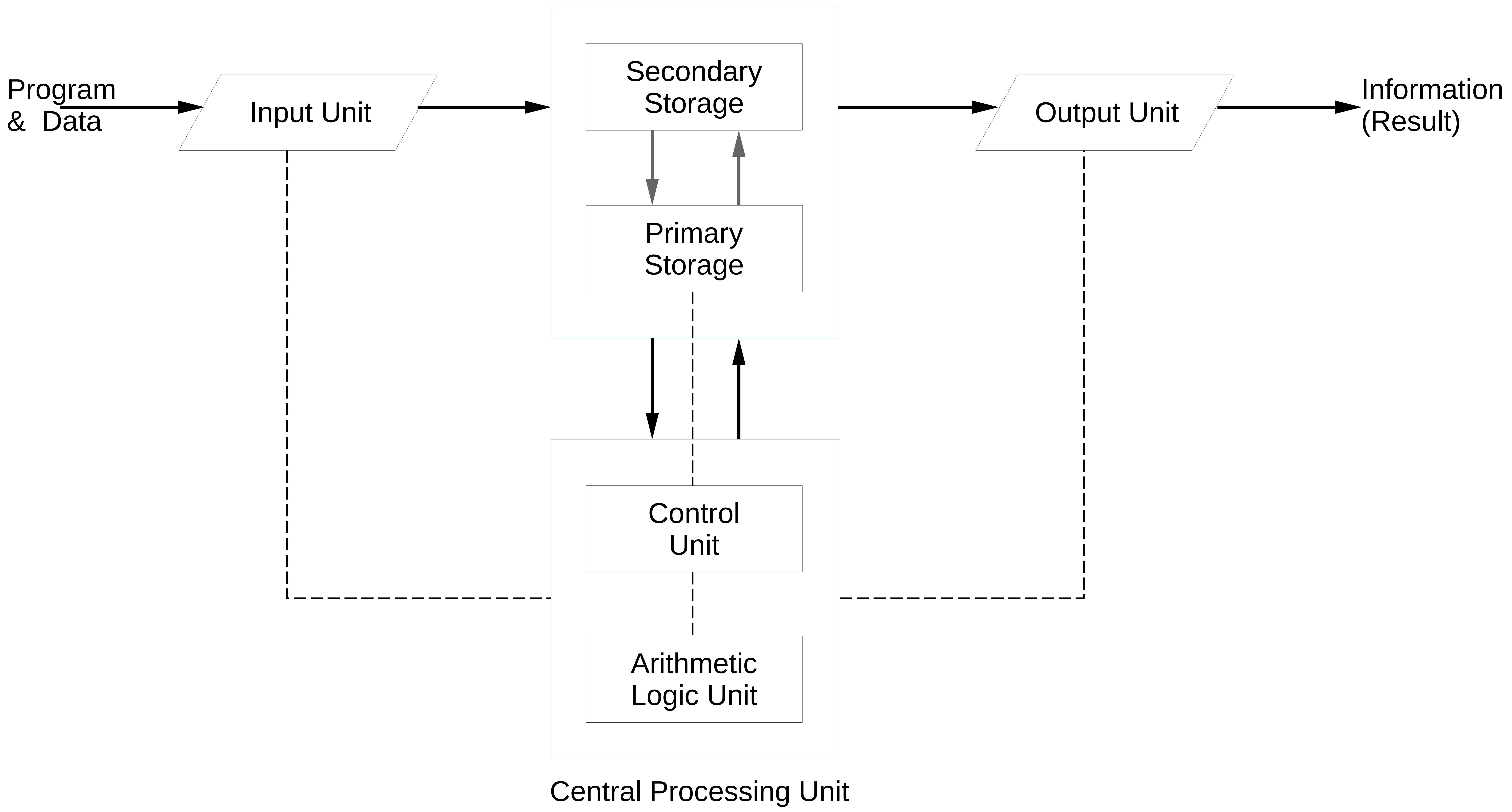
A general-purpose computer system is the best-known example of a digital system. Other examples include telephone switching exchanges, digital voltmeters, digital counters, electronic calculators and digital displays.

Computer architecture deals with the specification of the instruction set and the hardware units that implement the instructions.

Computer hardware consists of electronic circuits, displays, magnetic and optic storage media and also the communication facilities.

Functional units are a part of a CPU that performs the operations and calculations called for by the computer program.

Functional units of a computer system are parts of the CPU (Central Processing Unit) that performs the operations and calculations called for by the computer program. A computer consists of five main components namely, Input unit, Central Processing Unit, Memory unit Arithmetic & logical unit, Control unit and an Output unit.


Program & Data
Input Unit
Secondary Storage
Primary Storage
Output Unit
Information (Result)
Control Unit
Arithmetic Logic Unit
Central Processing Unit

Input Unit

The input to the computer is either a data or instruction, that guides the arithmetic and logic unit about what operations have to be performed and also controls the movement of data between the computer and its I/O devices.

Now, the input i.e. data or instruction are accepted with the help of input units such as a keyboard, mouse, touchpad, joystick, and trackball. The most familiar device that we use to accept input is the ‘keyboard’ and ‘mouse’. All of these input devices are graphic input devices and you can see their effect on the display unit.

Whenever you strike any key on the keyboard it gets converts to the binary code and is handed over to the processor which would interpret the code and perform the appropriate action.

Memory Unit

A memory unit is required to store the programs that have the set of instructions that instructs Arithmetic and logic unit which operation has to be performed. It also stores the data associated with the program. The memory can further be classified into three types:

1. **Primary memory**
   Primary memory is also known as the main memory or the random-access memory (RAM). It is the fastest accessible memory of the computer. If a program has to be executed it first needs to be placed in the primary memory. Then the instructions of the program are fetched one at a time by the processor for execution.
   The memory is organized in such a way that in one basic operation, one-word can be retrieved from the memory or one word can be stored to the memory. A word length could be 16, 32, or 64 bits.
   The primary memory is expensive as well as faster. But primary memory is volatile in nature it does not retain its content when the power gets off.

2. Secondary Memory

Secondary memory is the hard disk of your system, it also includes flash drive, optical disks, magnetic disk. The secondary memory is slower and less expensive as compared to primary memory. It doesn’t lose its contents even if the supply of power gets off.

We require secondary memory to store a large volume of data or program permanently or the data that is less likely to be retrieved.

3. Cache Memory

Cache memory can be accessed much faster as compared to primary memory and it is even smaller in size. It is stored with the data that is required frequently by the processor.

As we know the program to be executed and the data associated with it is brought to the primary memory and the processor fetches the program instructions from there. The process also places a copy of the instructions and associated data in the cache memory.

Now, the instructions that are required to be executed repeatedly such as loops are retrieved from the cache memory to improve the execution rate.

Arithmetic Logic Unit

All the arithmetic operations are performed by the arithmetic logic unit of the processors. Arithmetic operations such as addition, subtraction, division, multiplication, comparison between the numbers, etc.

The ALU unit performs the operations present in the instruction and stores the result into the memory. It also stores the intermediate results of the operation in the registers.

The arithmetic operation is performed on the operands. The operands are placed into the registers which store one word at a time which is sufficient for an operand. Retrieval of the data form registers is even faster than the cache memory.

Arithmetic Logic Unit

All the arithmetic operations are performed by the arithmetic logic unit of the processors. Arithmetic operations such as addition, subtraction, division, multiplication, comparison between the numbers, etc.

The ALU unit performs the operations present in the instruction and stores the result into the memory. It also stores the intermediate results of the operation in the registers.

The arithmetic operation is performed on the operands. The operands are placed into the registers which store one word at a time which is sufficient for an operand. Retrieval of the data form registers is even faster than the cache memory.

Output Unit

A computer is a functional unit and as it has an input unit to accept the input, it also has the output unit to provide the generated output by the system. The most familiar device used to output a result is a printer.

A display screen is also an output unit as it displays the generated result, but it also displays the input provided to the system. That's why the display screen is termed as the 'I/O unit' because of its dual function.

Control Unit

The functions of input, ALU, memory, and output unit must be coordinated so that everything goes in sequence i.e. the processor accepts input, place it in memory, processes the stored input, and generates output. This entire sequence is coordinate by the control unit.

In this way, the functional units of computers cooperate to generate useful output. We have discussed each of the functional units of the computer in brief and understood their functional behavior.

The binary number system

Digital computers use the binary number system, which has two digits: 0 and 1. A binary digit is called a bit. Information is represented in digital computers in groups of bits. By using various coding techniques, groups of bits can be made to represent not only binary numbers but also other discrete symbols, such as decimal digits or letters of the alphabet. By judicious use of binary arrangements and by using various coding techniques, the groups of bits are used to develop complete sets of instructions for performing various types of computations.

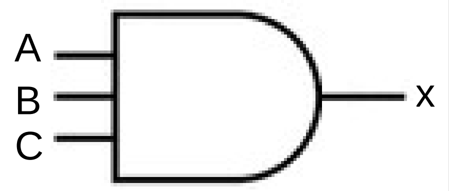
In contrast to the common decimal numbers that employ the base 10 system, binary numbers use a base 2 system with two digits: 0 and 1. The decimal equivalent of a binary number can be found by expanding it into a power series with a base of 2. For example, the binary number 1001011 represents a quantity that can be converted to a decimal number by multiplying each bit by the base 2 raised to an integer power as follows:

$$1*2^6+0*2^5+0*2^4+1*2^3+0*2^2+1*2^1+1*2^0=75$$

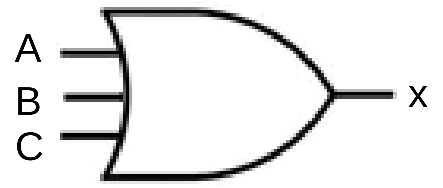
The seven bits 1001011 represent a binary number whose decimal equivalent is 75. However, this same group of seven bits represents the letter K when used in conjunction with a binary code for the letters of the alphabet. It may also represent a control code for specifying some decision logic in a particular digital computer. In other words, groups of bits in a digital computer are used to represent many different things. This is similar to the concept that the same letters of an alphabet are used to construct different languages, such as English and French.

Logic Gates

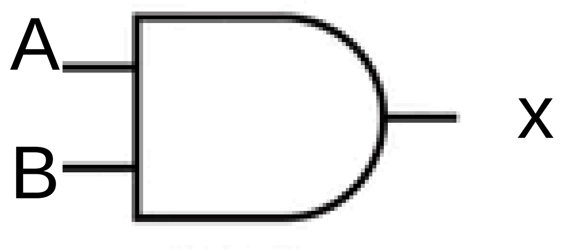
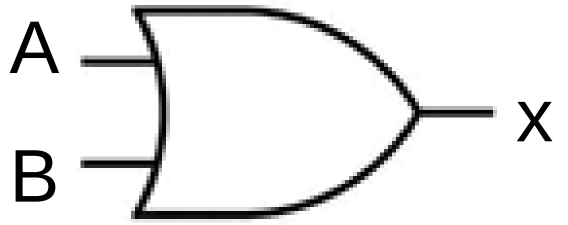
Binary information is represented in digital computers by physical quantities called signals . Electrical signals such as voltages exist throughout the computer in either one of two recognizable states. The two states represent a binary variable that can be equal to 1 or 0. For example, a particular digital computer may employ a signal of 3 volts to represent binary 1 and 0.5 volt to represent binary 0. The input terminals of digital circuits accept binary signals of 3 and 0.5 volts and the circuits respond at the output terminals with signals of 3 and 0.5 volts to represent binary input and output corresponding to 1 and 0, respectively.



A	B	C	x
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1



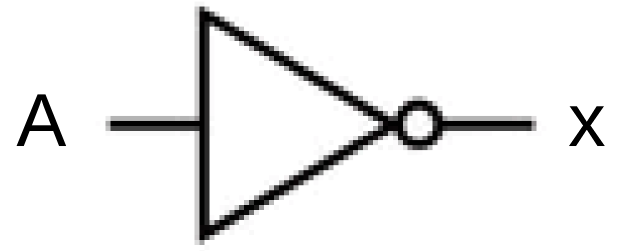
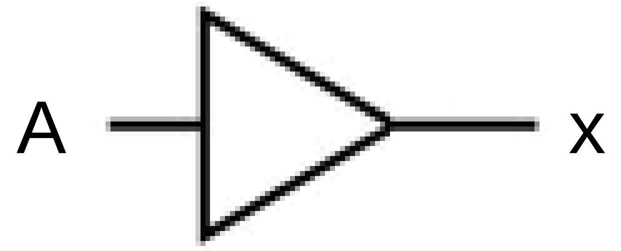
A	B	C	x
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Name	*Graphic Symbol*	*Algebraic Function*	*Truth Table*
AND		$x = A * B$ or $x = AB$	A B x: 0 0 **0**; 0 1 **0**; 1 0 **0**; 1 1 **1**
OR		$x = A + B$	A B x: 0 0 **0**; 0 1 **1**; 1 0 **1**; 1 1 **1**

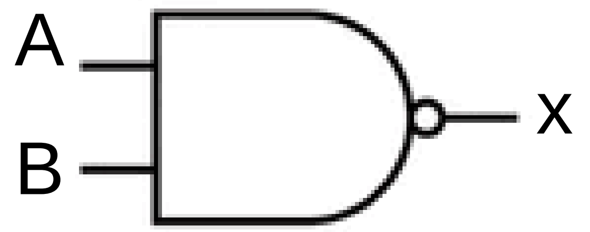
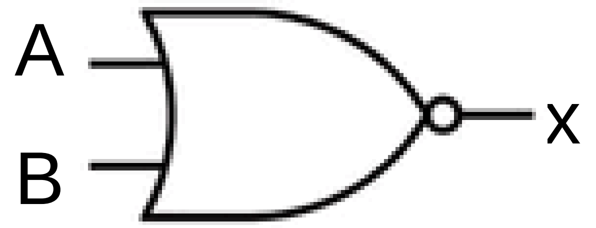
A	B	x
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

A	B	x
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

Binary logic deals with binary variables and with operations that assume a logical meaning. It is used to describe, in algebraic or tabular form, the manipulation and processing of binary information. The manipulation of binary information is done by logic circuits called gates . Gates are blocks of hardware that produce signals of binary 1 or 0 when input logic requirements are satisfied. A variety of logic gates are commonly used in digital computer systems. Each gate has a distinct graphic symbol and its operation can be described by means of an algebraic expression. The input-output relationship of the binary variables for each gate can be represented in tabular form by a truth table.

Name	*Graphic Symbol*	*Algebraic Function*	*Truth Table*
Inverter (NOT)		$x = A'$	<table><tr><th>A</th><th>x</th></tr><tr><td>0</td><td>**1**</td></tr><tr><td>1</td><td>**0**</td></tr></table>
Buffer		$x = A$	<table><tr><th>A</th><th>x</th></tr><tr><td>0</td><td>**0**</td></tr><tr><td>1</td><td>**1**</td></tr></table>

Each gate has one or two binary input variables designated by A and B and one binary output variable designated by x. The AND gate produces the AND logic function: that is, the output is 1 if input A and input B are both equal to 1; otherwise, the output is 0. These conditions are also specified in the truth table for the AND gate. The table shows that output x is 1 only when both input A and input B are 1. The algebraic operation symbol of the AND function is the same as the multiplication symbol of ordinary arithmetic. We can either use a dot between the variables or concatenate the variables without an operation symbol between them. AND gates may have more than two inputs, and by definition, the output is 1 if and only if all inputs are 1.

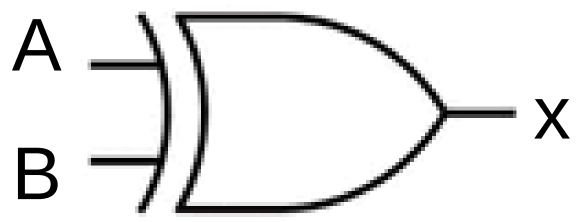
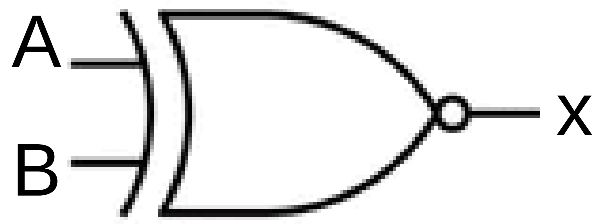
Name	*Graphic Symbol*	*Algebraic Function*	*Truth Table*
NAND		$x = (AB)'$	A B x: 0 0 **1**; 0 1 **1**; 1 0 **1**; 1 1 **0**
NOR		$x = (A + B)'$	A B x: 0 0 **1**; 0 1 **0**; 1 0 **0**; 1 1 **0**

NAND truth table:

A	**B**	**x**
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**0**

NOR truth table:

A	**B**	**x**
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**0**

Name	*Graphic Symbol*	*Algebraic Function*	*Truth Table*
Exclusive-OR (XOR)		$x = A \oplus B$ or $X = A'B + AB'$	**A** | **B** | **x**: 0 0 **0**; 0 1 **1**; 1 0 **1**; 1 1 **0**
Exclusive-NOR or equivalence (XNOR)		$x = (A \oplus B)'$ or $X = A'B' + AB$	**A** | **B** | **x**: 0 0 **1**; 0 1 **0**; 1 0 **0**; 1 1 **1**

A	**B**	**x**
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

A	**B**	**x**
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**1**

The OR gate produces the inclusive-OR function; that is, the output is 1 if input A or input B or both inputs are 1; otherwise, the output is 0. The algebraic symbol of the OR function is +, similar to arithmetic addition. OR gates may have more than two inputs, and by definition, the output is 1 if any input is 1.

The inverter circuit inverts the logic sense of a binary signal. It produces the NOT, or complement, function. The algebraic symbol used for the logic complement is either a prime or a bar over the variable symbol.

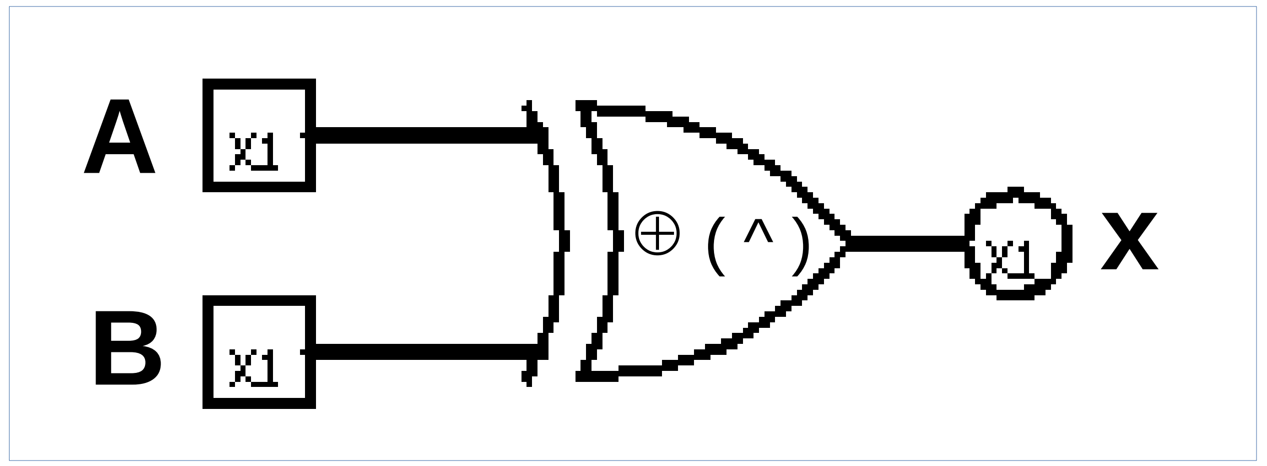
The small circle in the output of the graphic symbol of an inverter designates a logic complement. A triangle symbol by itself designates a buffer circuit. A buffer does not produce any particular logic function since the binary value of the output is the same as the binary value of the input. This circuit is used merely for power amplification. For example, a buffer that uses 3 volts for binary 1 will produce an output of 3 volts when its input is 3 volts. However, the amount of electrical power needed at the input of the buffer is much less than the power produced at the output of the buffer. The main purpose of the buffer is to drive other gates that require a large amount of power.

The NAND function is the complement of the AND function, as indicated by the graphic symbol, which consists of an AND graphic symbol followed by a small circle. The designation NAND is derived from the abbreviation of NOT-AND. The NOR gate is the complement of the OR gate and uses an OR graphic symbol followed by a small circle. Both NAND and NOR gates may have more than two inputs, and the output is always the complement of the AND or OR function, respectively.

The exclusive-OR gate has a graphic symbol similar to the OR gate except for the additional curved line on the input side. The output of this gate is 1 if any input is 1 but excludes the combination when both inputs are 1. The exclusive-OR function has its own algebraic symbol or can be expressed in terms of AND, OR, and complement operations. The exclusive-NOR is the complement of the exclusive-OR, as indicated by the small circle in the graphic symbol. The output of this gate is 1 only if both inputs are equal to 1 or both inputs are equal to 0. A more fitting name for the exclusive-OR operation would be an odd function; that is, its output is 1 if an odd number of inputs are 1. Thus in a three-input exclusive-OR (odd) function, the output is 1 if only one input is 1 or if all three inputs are 1. The exclusive-OR and exclusive-NOR gates are commonly available with two inputs, and only seldom are they found with three or more inputs.

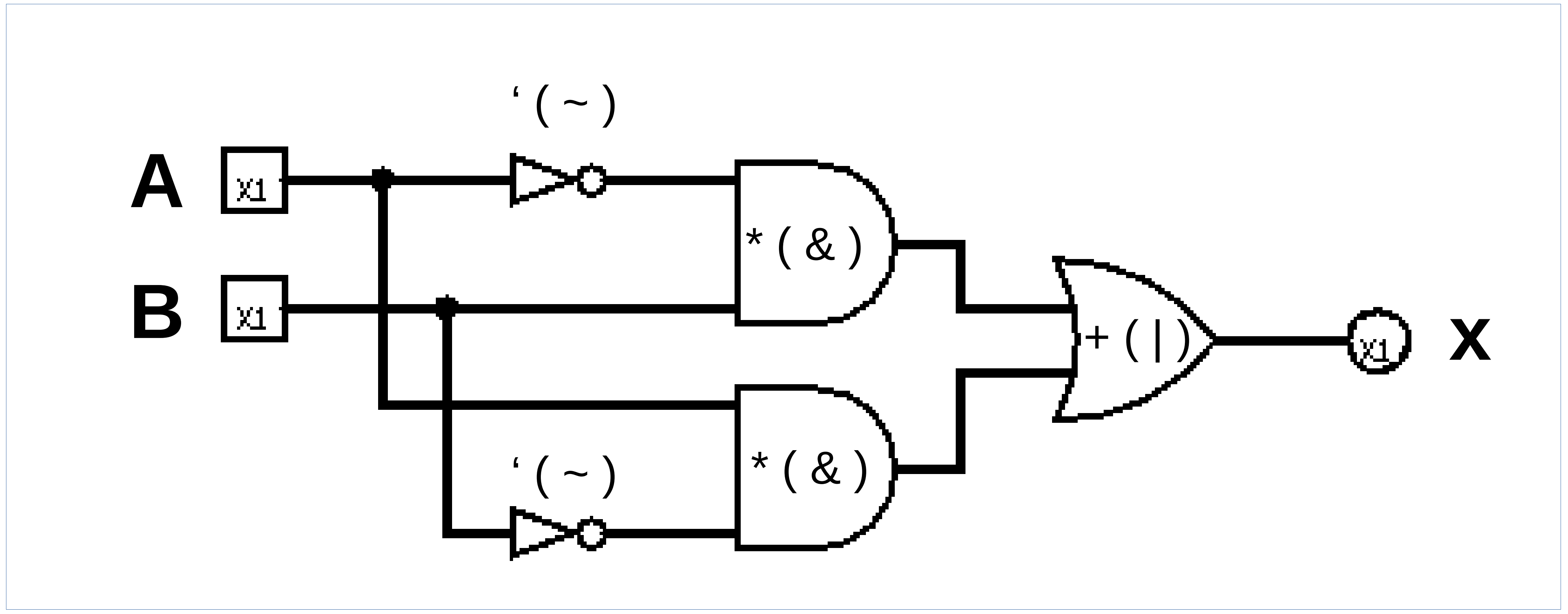
XOR

A ⊕ B = x



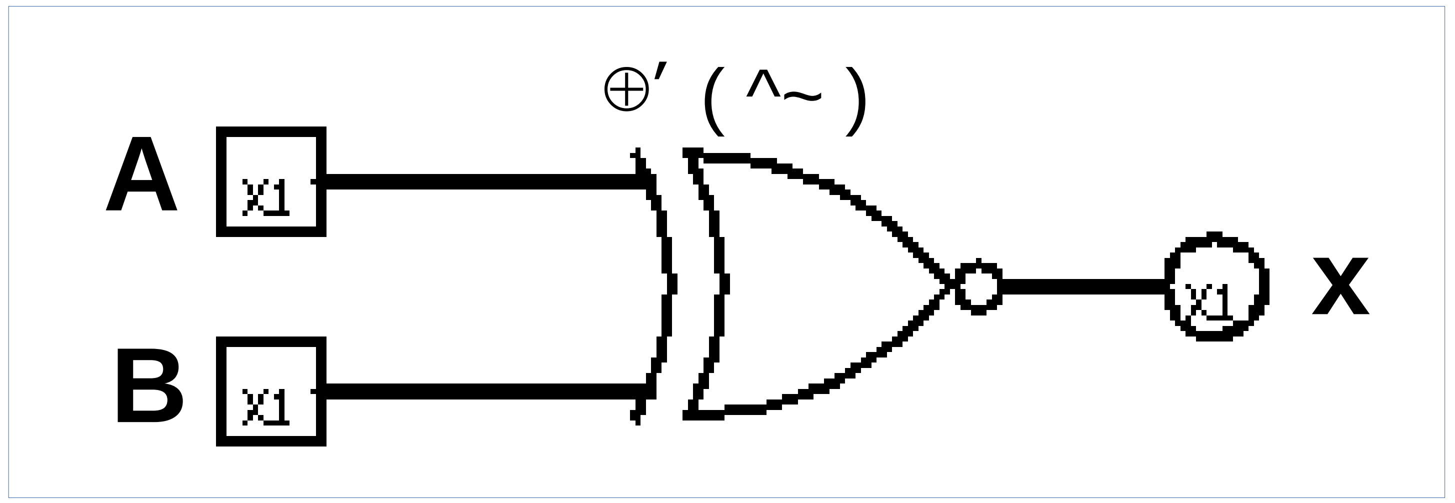
=

A’B + AB’ = x



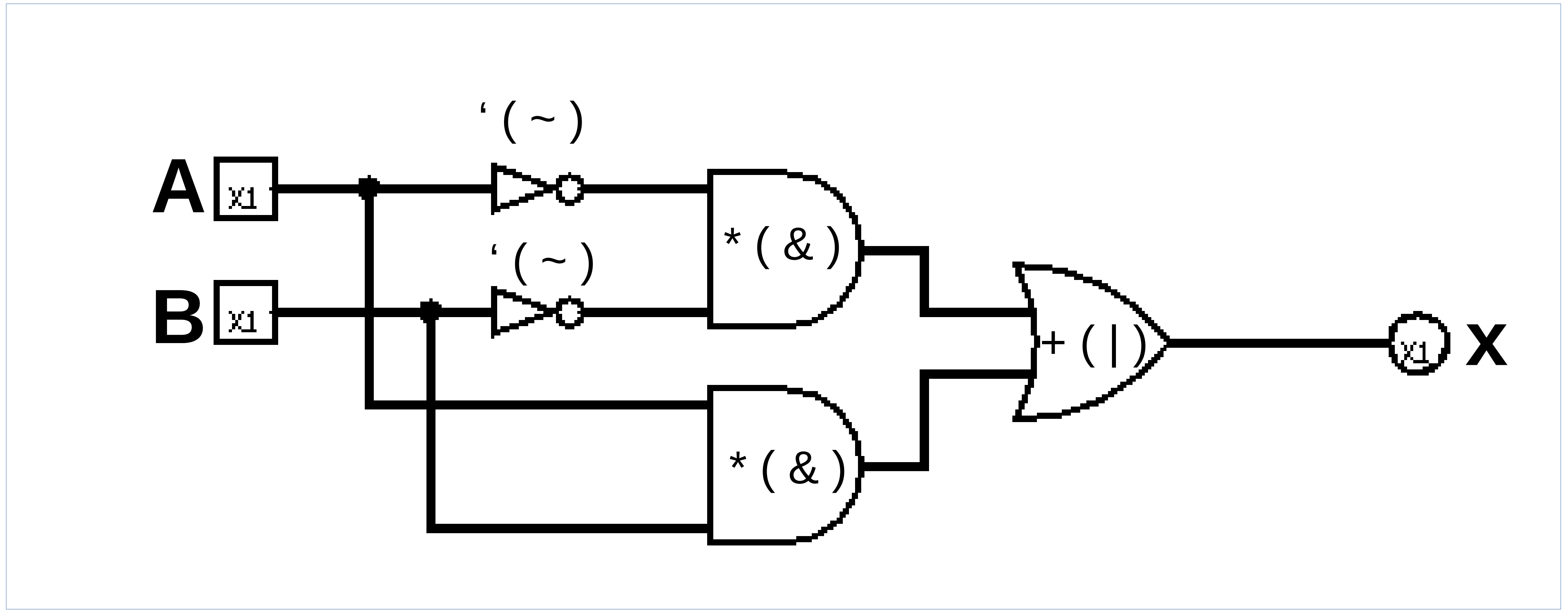
NXOR

(A ⊕ B)’ = x

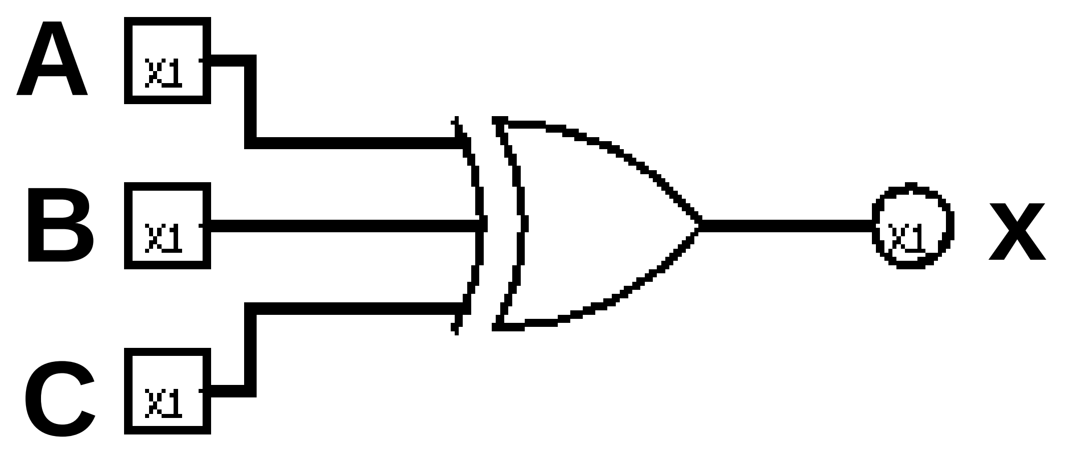


=

A’B’ + AB = x



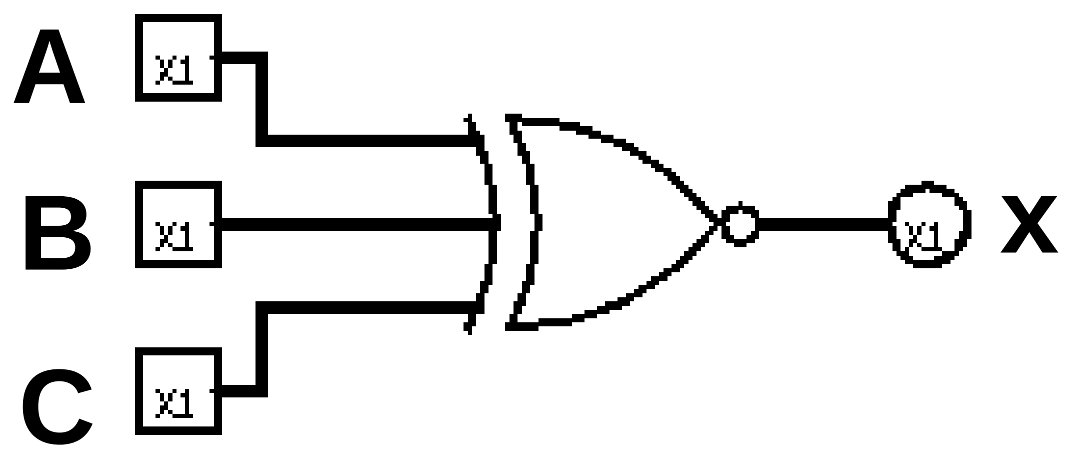
Three-input XOR



x= A'B'C + A'BC' + AB'C'

A	B	C	x
0	0	0	**0**
0	0	1	**1**
0	1	0	**1**
0	1	1	**0**
1	0	0	**1**
1	0	1	**0**
1	1	0	**0**
1	1	1	**0**

Three-input XNOR



x = A'B'C' + BC + AC + AB

A	B	C	x
0	0	0	**1**
0	0	1	**0**
0	1	0	**0**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**

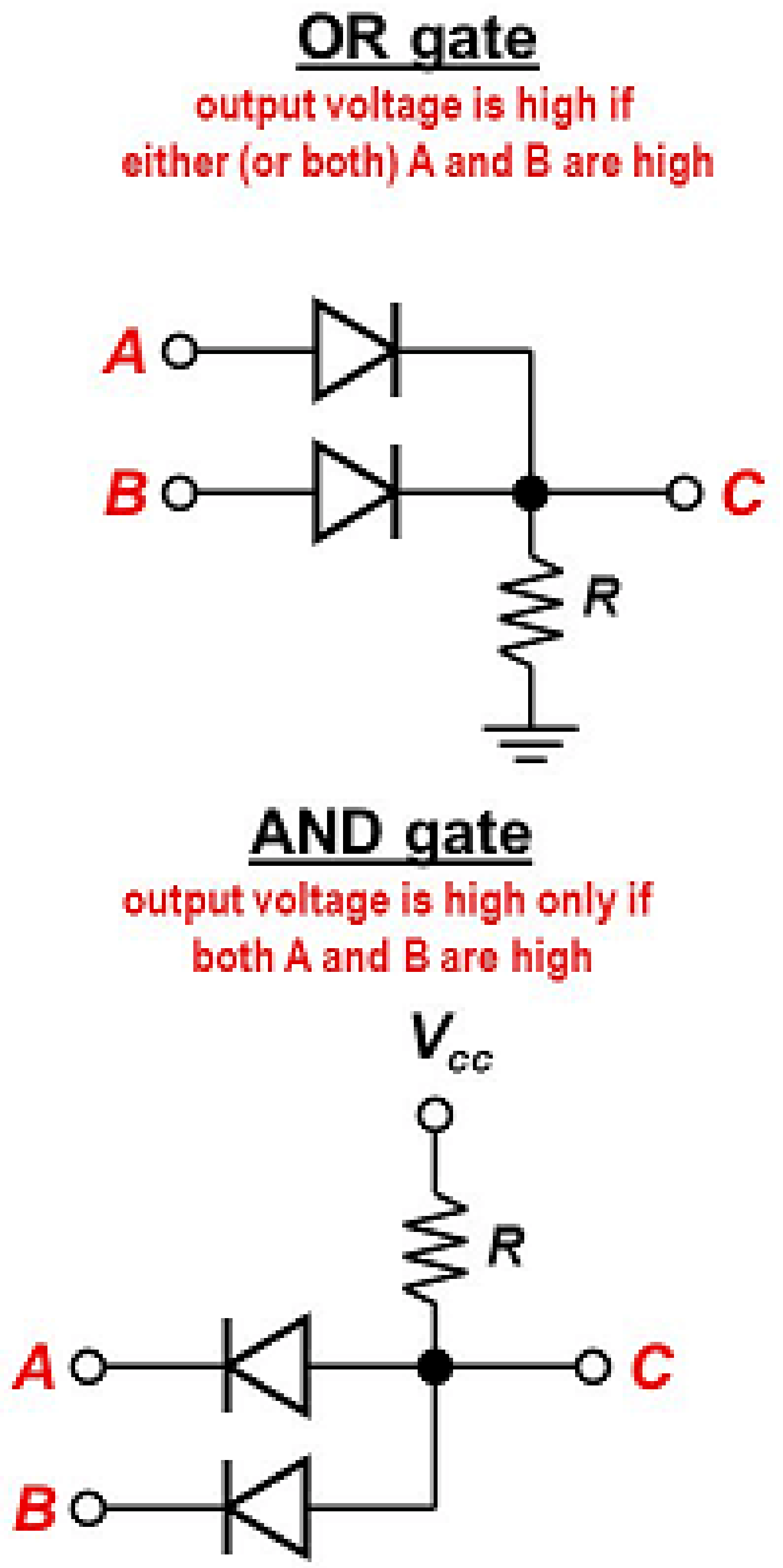
Important to note that gates NOR and XNOR aren't gates by themselves, but are built from AND, OR, and NOT gates. In this sense, they're little logic circuits by themselves.
In general, every logic circuit can be built out of it's expression.

Of course, we need to bear in mind that the basic gates themselves are complex structures as well, and built out of several transistors.
(see the next page)

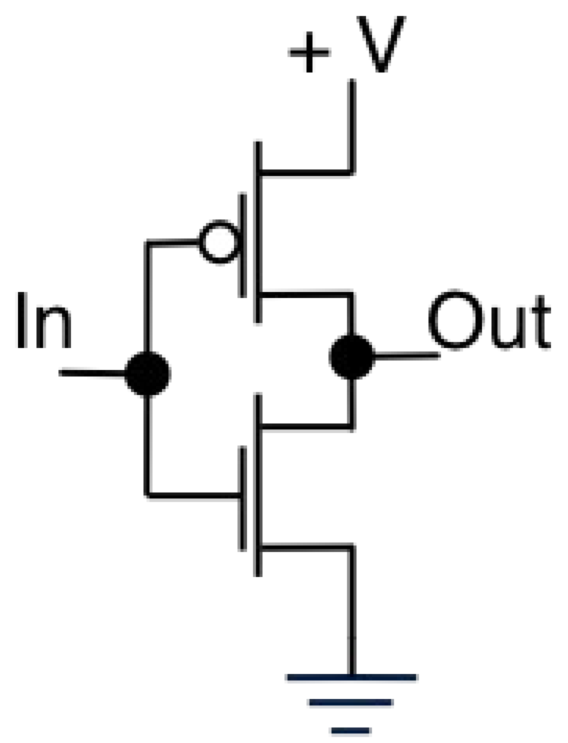
Internal structure of logic gates

There are several ways to create logic gates from basic electronic components, such as transistors, diodes and resistors. There are some examples of them:

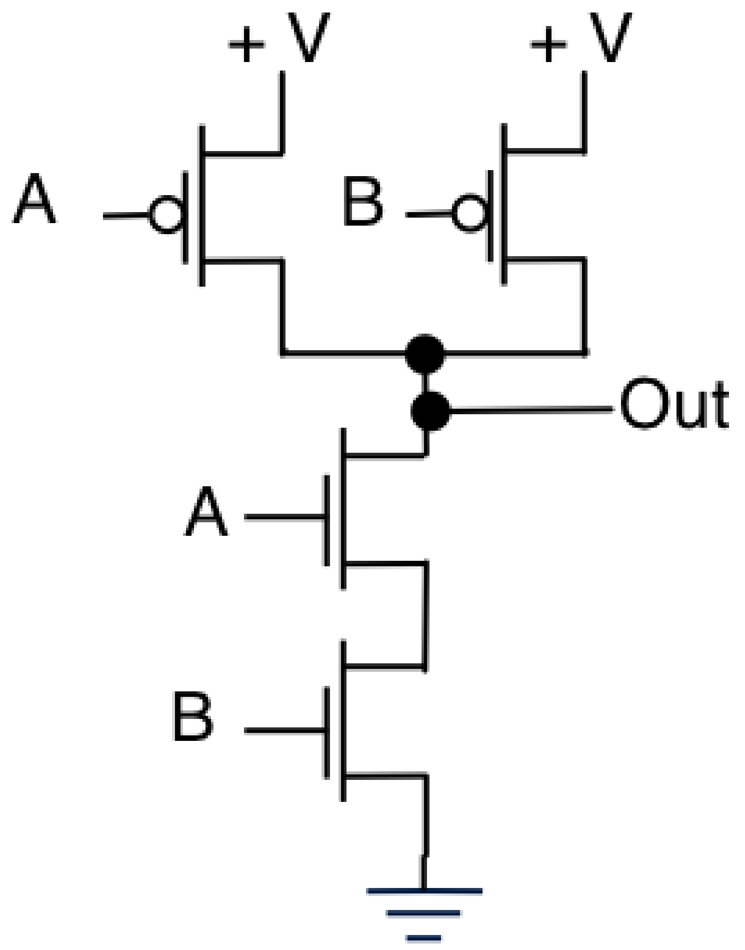
AND and OR gates created with resistor-diode logic


OR gate
output voltage is high if either (or both) A and B are high
A
B
C
R
AND gate
output voltage is high only if both A and B are high
V_{cc}
R
A
B
C

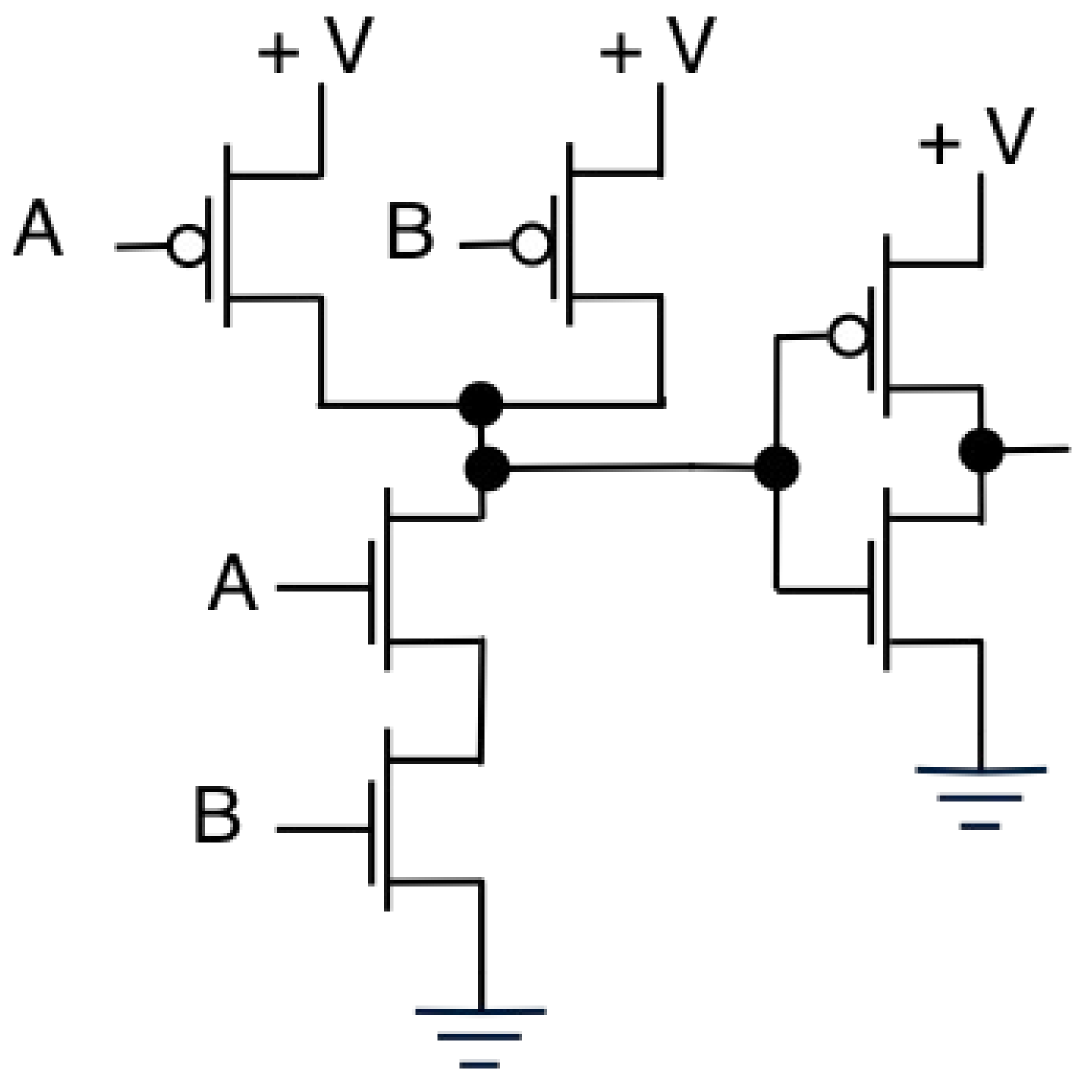
Inverter out of two transistors



NAND gate



AND gate - NAND and Inverter



More examples of several techniques of creating logic gates from basic electronic components

Article in "Scientific American" about logic gates

Boolean Algebra

Boolean algebra is an algebra that deals with binary variables and logic operations. The variables are designated by letters such as A, B, x, and y. The three basic logic operations are AND, OR, and complement. A Boolean function can be expressed algebraically with binary variables, the logic operation symbols, parentheses, and equal signs. For a given value of the variables, the Boolean function can be either 1 or 0. Consider, for example, the Boolean function

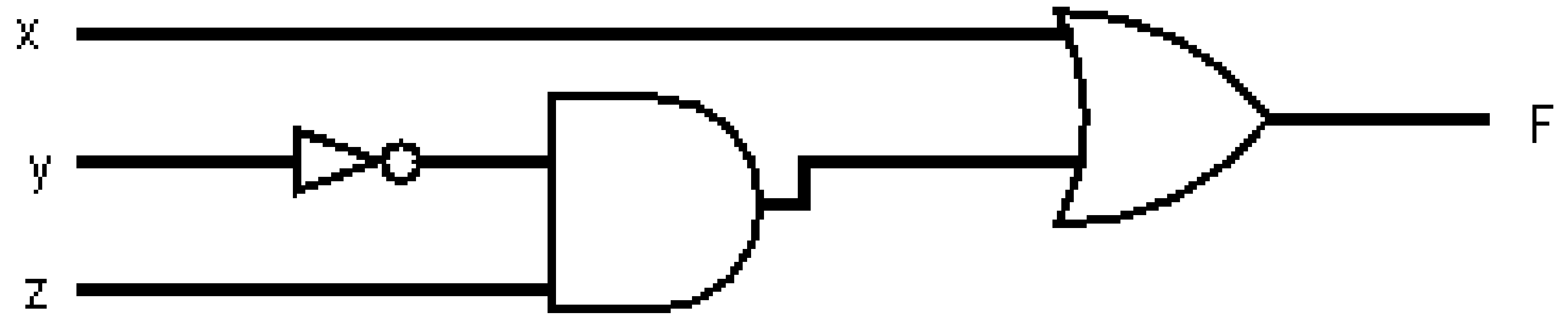
$$F = x + y'z$$

The function F is equal to 1 if x is 1 or if both y' and z are equal to 1; F is equal to 0 otherwise. But saying that y' = 1 is equivalent to saying that y = 0 since y' is the complement of y. Therefore, we may say that F is equal to 1 if x = 1 or if yz = 01. The relationship between a function and its binary variables can be represented in a truth table. To represent a function in a truth table we need a list of the 2^n combinations of the n binary variables. As shown in the figure, there are eight possible distinct combinations for assigning bits to the three variables x, y, and z. The function F is equal to 1 for those combinations where x = 1 or yz = 01; it is equal to 0 for all other combinations.

Truth Table

x	y	z	**F**
0	0	0	**0**
0	0	1	**1**
0	1	0	**0**
0	1	1	**0**
1	0	0	**1**
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**

Figure 1-3



Logic Diagram

The purpose of Boolean algebra is to facilitate the analysis and design of digital circuits. It provides a convenient tool to:

1. Express in the algebraic form a truth table relationship between binary variables.
2. Express in algebraic form the input-output relationship of logic diagrams.
3. Find simpler circuits for the same function.

A Boolean function specified by a truth table can be expressed algebraically in many different ways. By manipulating a Boolean expression according to Boolean algebra rules, one may obtain a simpler expression that will require fewer gates. To see how this is done, we must first study the manipulative capabilities of Boolean algebra.

Table 1-1 lists the most basic identities of Boolean algebra. All the identities in the table can be proven by means of truth tables. The first eight identities show the basic relationship between a single variable and itself, or in conjunction with the binary constants 1 and 0. The next five identities (9 through 13) are similar to ordinary algebra. Identity 14 does not apply in ordinary algebra but is very useful in manipulating Boolean expressions. Identities 15 and 16 are called DeMorgan's theorems and are discussed below. The last identity states that if a variable is complemented twice, one obtains the original value of the variable.

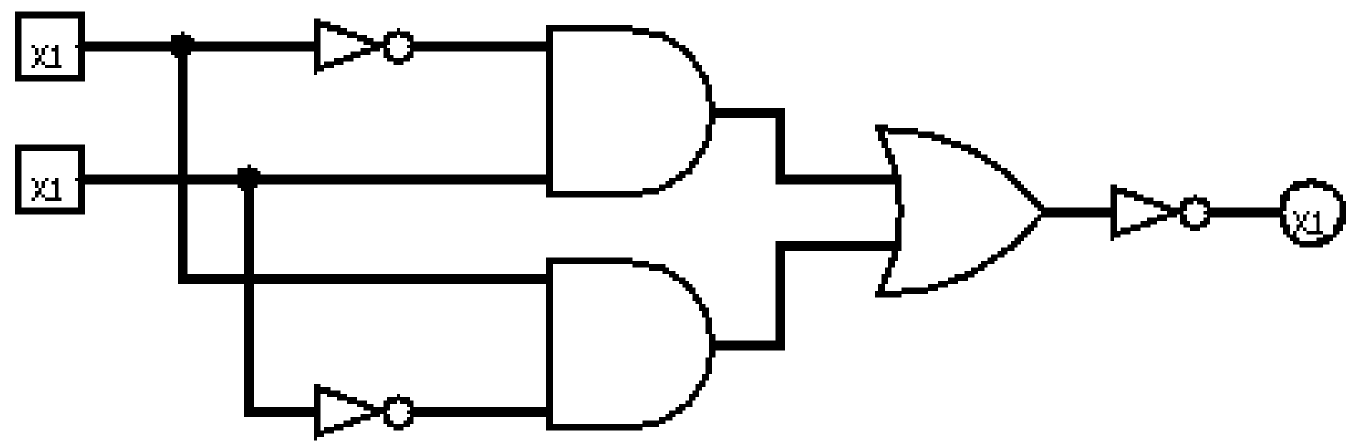
Table 1- 1. Basic Identities of Boolean Algebra

1	**x + 0 = x**
2	**x * 0 = 0**
3	**x + 1 = 1**
4	**x * 1 = x**
5	**x + x = x**
6	**x * x = x**
7	**x + x’ = 1**
8	**x * x’ = 0**
9	**x + y = y + x**
10	**xy = yx**
11	**x + (y + z) = (x + y) + z**
12	**x(yz) = (xy)z**
13	**x(y + z) = xy + xz**
14	**x + yx = (x + y)(x + z)**
15	**(x + y)’ = x’y’**
16	**(xy)’ = x’ + y’**
17	**(x’)’= x**

Example of finding simpler circuit that does the same work with Boolean algebra

As we have already seen, we can build an NXOR (Not-XOR) gate by inverting the existing XOR gate, thus adding an inverter to its output. Hence the Boolean expression of the XOR gate is A'B + AB', the algebraical representation of adding an inverter will be (A'B + AB')'.

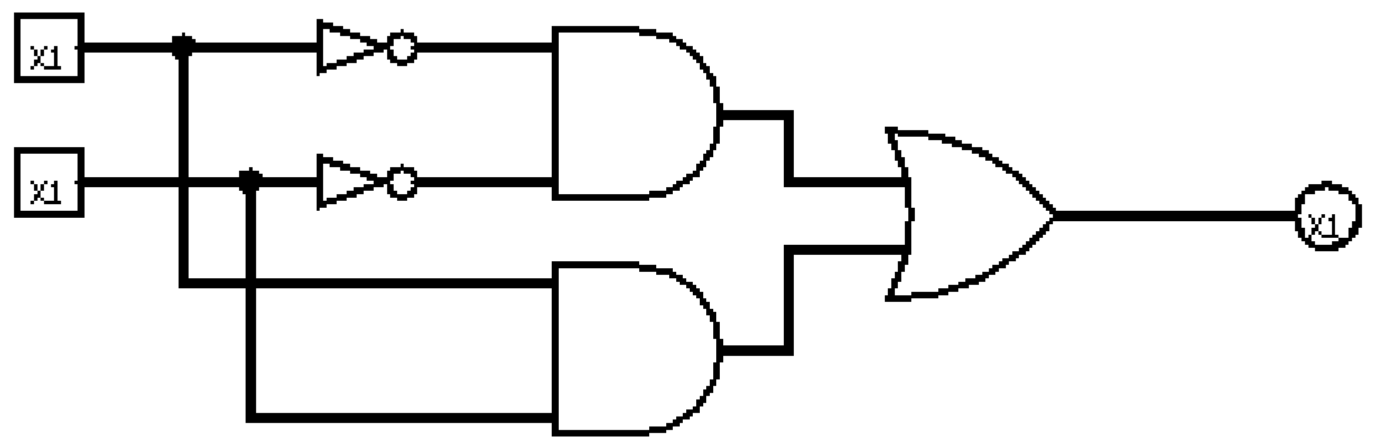
So, the circuit built this way will be simply the XOR circuit with an inverter added:



But, utilizing Boolean algebra, we can simplify the expression of the circuit, and therefore, the circuit itself, sparing electronic components. Using the Basic Identities of Boolean Algebra from table 1-1, we can simplify the algebraic expression (A'B + AB')' through several steps:

1. The identity 15, (x + y)' = x'y': **(A'B + AB')' = (A'B)'(AB')'**
2. The identity 16, (xy)' = x' + y': **(A'B)'(AB')' = (A + B')(A' + B)**
3. The identity 13, x(y + z) = xy + xz. Here we'll see the first member of the AND operation [(A + B')] as x and each of the members of the OR operation of the second member [A' and B] as y and z: (A + B')(A' + B) = **A'(A + B') + B(A + B')**
4. Also identity 13, this time we'll see each member of the main OR operation [A'(A + B') and B(A + B')] by itself: A'(A + B') + B(A + B') = **A'A + A'B' +BA + BB'**
5. Using identity 8, we see that two of the members of the previous expression are equal to zero, so can be omitted: A'A + A'B' +BA + BB' = 0 + A'B' + BA + 0
6. Finally, the rezult: **A'B' + AB**

According to this expression [**A'B' + AB**] we can build a new circuit, that does the same job but uses one less inverter component:



The identities listed in the table apply to single variables or to Boolean functions expressed in terms of binary variables. For example, consider the following Boolean algebra expression:

$$AB' + C'D + AB' + C'D$$

By letting $x = AB' + C'D$ the expression can be written as $x + x$. From identity 5 in Table 1-1 we find that $x + x = x$. Thus the expression can be reduced to only two terms:

$$AB' + C'D + A'B + C'D = AB' + C'D$$

DeMorgan' s theorem is very important in dealing with NOR and NAND gates. It states that a NOR gate that performs the $(x + y)'$ function is equivalent to the function $x'y'$. Similarly, a NAND function can be expressed by either $(xy)'$
or $(x' + y')$. For this reason the NOR and NAND gates have two distinct graphic symbols, as shown in Figs. 1-4 and 1-5. Instead of representing a NOR gate with an OR graphic symbol followed by a circle, we can represent it by an AND graphic symbol preceded by circles in all inputs. The invert-AND symbol for the NOR gate follows from DeMorgan's theorem and from the convention that small circles denote complementation. Similarly, the NAND gate has two distinct symbols, as shown in Fig. 1-5. To see how Boolean algebra manipulation is used to simplify digital circuits, consider the logic diagram of Fig. 1-6(a). The output of the circuit can be expressed algebraically as follows:

$$F = ABC + A\,BC' + A'C$$

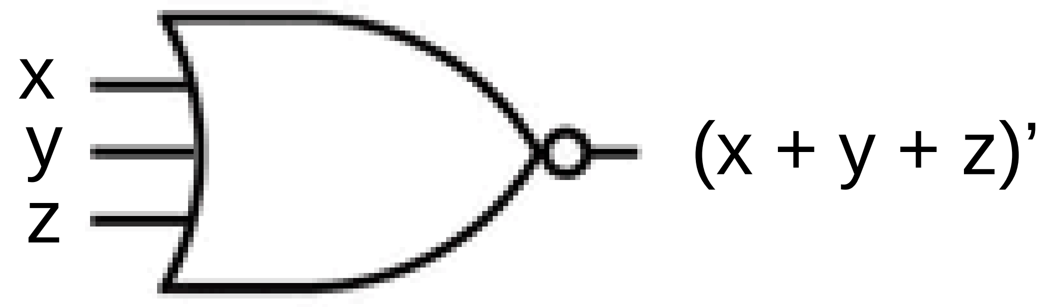
Each term corresponds to one AND gate, and the OR gate forms the logical sum of the three terms. Two inverters are needed to complement A' and C'. The expression can be simplified using Boolean algebra.

$$F = ABC + A\,BC' + A'C = AB(C + C') + A'C = AB + A'C$$

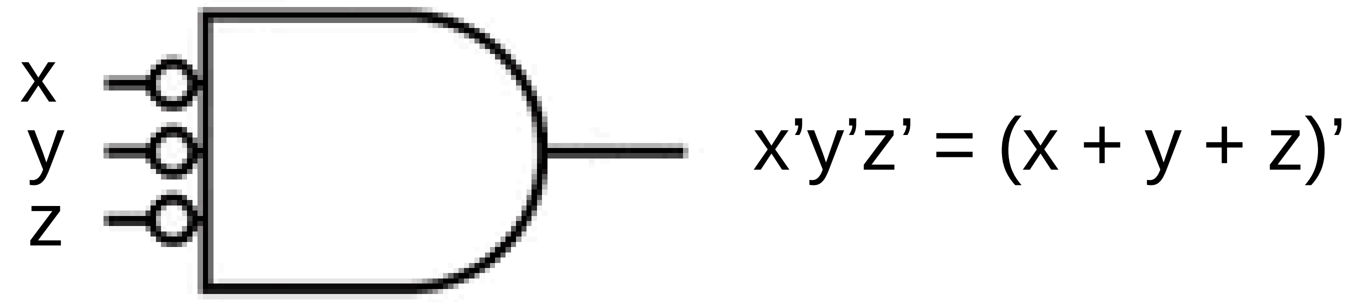
Note that$(C + C)' = 1$ by identity 7 and $AB \cdot 1 = AB$ by identity 4 in Table 1-1.

The logic diagram of the simplified expression is drawn in Fig. 1-6(b). It requires only four gates rather than the six gates used in the circuit of Fig. 1-6(a). The two circuits are equivalent and produce the same truth table relationship between inputs A, B, C and output F.

Figure 1-4. Two graphic symbols for NOR gate

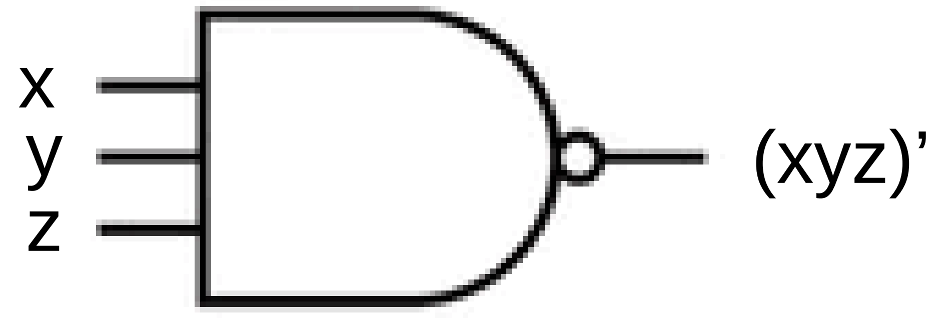


(a) OR-invert

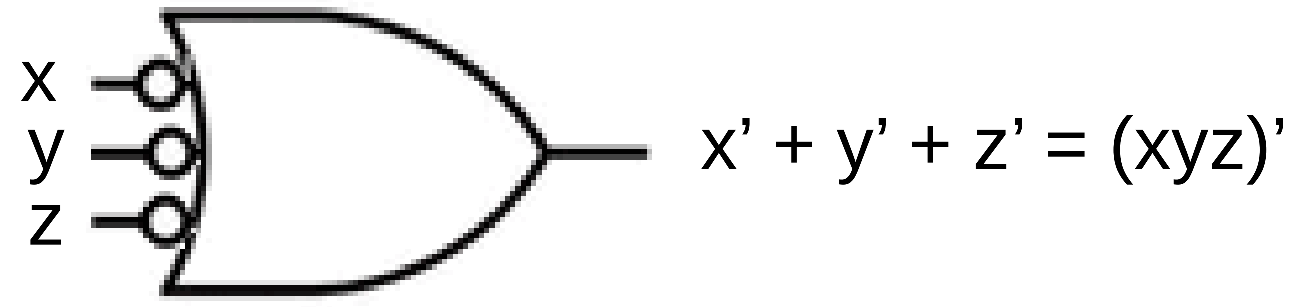


(b) invert-AND

Figure 1-5. Two graphic symbols for NAND gate

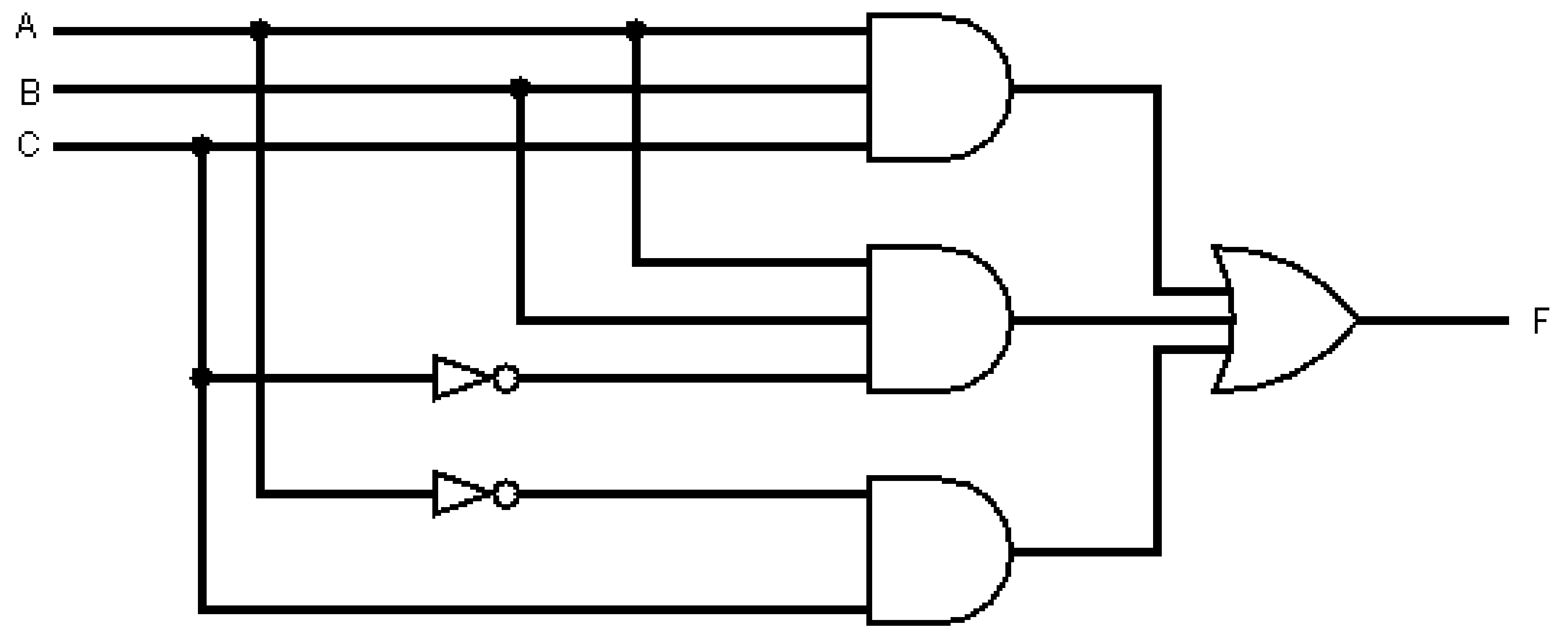


(a) AND-invert

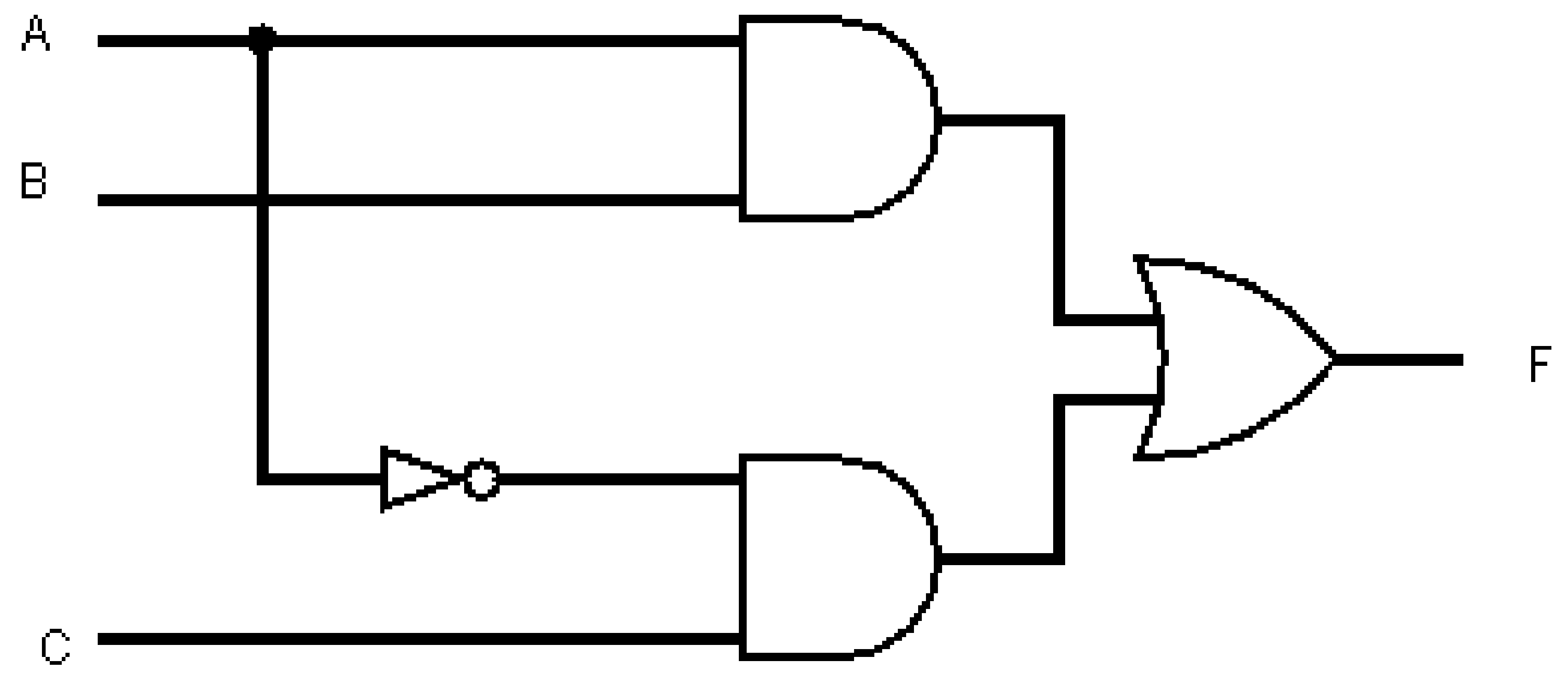


(b) invert-OR

Figure 1-6. Two logic diagrams for the same Boolean function



(a) F = ABC + ABC’ + A’C



(b) F = AB + A’C

Complement of a Function

The complement of a function F, when expressed in a truth table, is obtained by interchanging 1's and 0's in the values of F in the truth table. When the function is expressed in algebraic form, the complement of the function can be derived by means of DeMorgan's theorem. The general form of DeMorgan's theorem can be expressed as follows:

$$(X_1+X_2+X_3+...+X_n)'=X_1'X_2'X_3'...X_n'$$
$$(X_1X_2X_3...X_n)'=X_1'+X_2'+X_3'+...+X_n'$$

From the general DeMorgan's theorem, we can derive a simple procedure for obtaining the complement of an algebraic expression. This is done by changing all OR operations to AND operations and all AND operations to OR operations and then complementing each individual letter variable. As an example, consider the following expression and its complement:

$$F = AB + C'D' + B'D$$
$$F' = (A' + B')(C + D)(B + D')$$

The complement expression is obtained by interchanging AND and OR operations and complementing each individual variable. Note that the complement of C ' is C .

K-Map (Karnaugh Map)

The complexity of the logic diagram that implements a Boolean function is related directly to the complexity of the algebraic expression from which the function is implemented. The truth table representation of a function is unique, but the function can appear in many different forms when expressed algebraically. The expression may be simplified using the basic relations of Boolean algebra. However, this procedure is sometimes difficult because it lacks specific rules for predicting each succeeding step in the manipulative process. The map method provides a simple, straightforward procedure for simplifying Boolean expressions. This method may be regarded as a pictorial arrangement of the truth table which allows an easy interpretation for choosing the minimum number of terms needed to express the function algebraically. The map method is also known as the Karnaugh map or K-map.

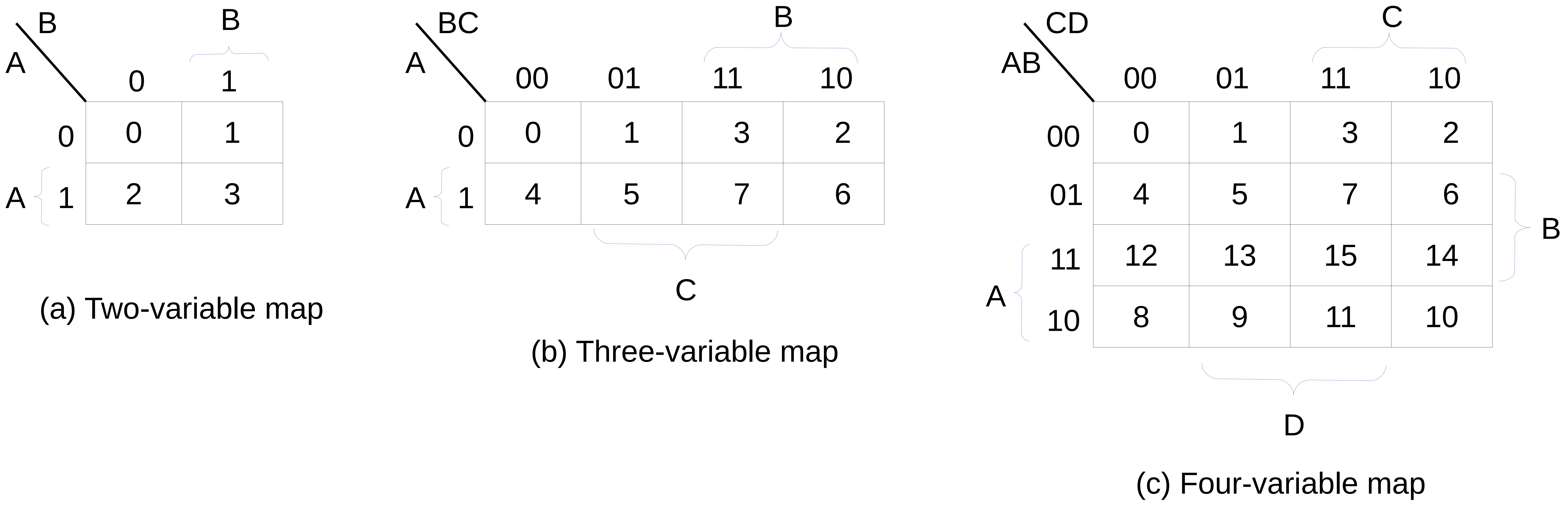
Each combination of the variables in a truth table is called a minterm. For example, the truth table of Fig. 1-3 contains eight minterms. When expressed in a truth table a function of n variables will have 2^n minterms, equivalent to the 2^n binary numbers obtained from n bits. A Boolean function is equal to 1 for some minterms and to 0 for others. The information contained in a truth table may be expressed in compact form by listing the decimal equivalent of those minterms that produce a 1 for the function. For example, the truth table of Fig. 1-3 can be expressed as follows:

$$F(x,y,z)=\Sigma(1,4,5,6,7)$$

The letters in parentheses list the binary variables in the order that they appear in the truth table. The symbol Σ stands for the sum of the minterms that follow in parentheses. The minterms that produce 1 for the function are listed in their decimal equivalent. The minterms missing from the list are the ones that produce 0 for the function. The map is a diagram made up of squares, with each square representing one minterm. The squares corresponding to minterms that produce 1 for the function are marked by a 1 and the others are marked by a 0 or are left empty. By recognizing various patterns and combining squares marked by 1's in the map, it is possible to derive alternative algebraic expressions for the function, from which the most convenient may be selected.

The maps for functions of two, three, and four variables are shown in Fig. 1-7. The number of squares in a map of n variables is 2^n. The 2^n minterms are listed by an equivalent decimal number for easy reference. The minterm numbers are assigned in an orderly arrangement such that adjacent squares represent minterms that differ by only one variable. The variable names are listed across both sides of the diagonal line in the corner of the map. The 0's and 1's marked along each row and each column designate the value of the variables. Each variable under brackets contains half of the squares in the map where that variable appears unprimed. The variable appears with a prime (complemented) in the remaining half of the squares.

Figure 1-7. Maps for two-, three-, and four -variable functions



The minterm represented by a square is determined from the binary assignments of the variables along the left and top edges in the map. For example, minterm 5 in the three-variable map is 101 in binary, which may be obtained from the 1 in the second row concatenated with the 01 of the second column. This min term represents a value for the binary variables A, B, and C, with A and C being unprimed and B being primed (i.e. , AB'C). On the other hand, minterrn 5 in the four-variable map represents a minterm for four variables. The binary number contains the four bits 0101, and the corresponding term it represents is A’BC’D.

Minterms of adjacent squares in the map are identical except for one variable, which appears complemented in one square and uncomplemented in the adjacent square. According to this definition of adjacency, the squares at the extreme ends of the same horizontal row are also to be considered adjacent. The same applies to the top and bottom squares of a column. As a result, the four corner squares of a map must also be considered to be adjacent. A Boolean function represented by a truth table is plotted into the map by inserting 1's in those squares where the function is 1. The squares containing 1's are combined in groups of adjacent squares. These groups must contain a number of squares that is an integral power of 2. Groups of combined adjacent squares may share one or more squares with one or more groups. Each group of squares represents an algebraic term, and the OR of those terms gives the simplified algebraic expression for the function. The following examples show the use of the map for simplifying Boolean functions. In the first example we will simplify the Boolean function

$$F(A, B, C) = \Sigma(3, 4, 6, 7)$$

The three-variable map for this function is shown in Fig. 1-8. There are four squares marked with 1's, one for each minterm that produces 1 for the function. These squares belong to minterms 3, 4, 6, and 7 and are recognized from Fig. 1-7(b). Two adjacent squares are combined in the third column. This column belongs to both B and C and produces the term BC. The remaining two squares with 1's in the two corners of the second row are adjacent and belong to row A and the two columns of C', so they produce the term AC'. The simplified algebraic expression for the function is the OR of the two terms:

$$F = BC + AC'$$

The second example simplifies the following Boolean function:

$$F(A, B, C) = \Sigma(0, 2, 4, 5, 6)$$

The five min terms are marked with 1' s in the corresponding squares of the three-variable map shown in Fig. 1-9. The four squares in the first and fourth columns are adjacent and represent the term C'. The remaining square marked with a 1 belongs to min term 5 and can be combined with the square of min term 4 to produce the term AB ' . The simplified function is

$$F = C' + AB'$$

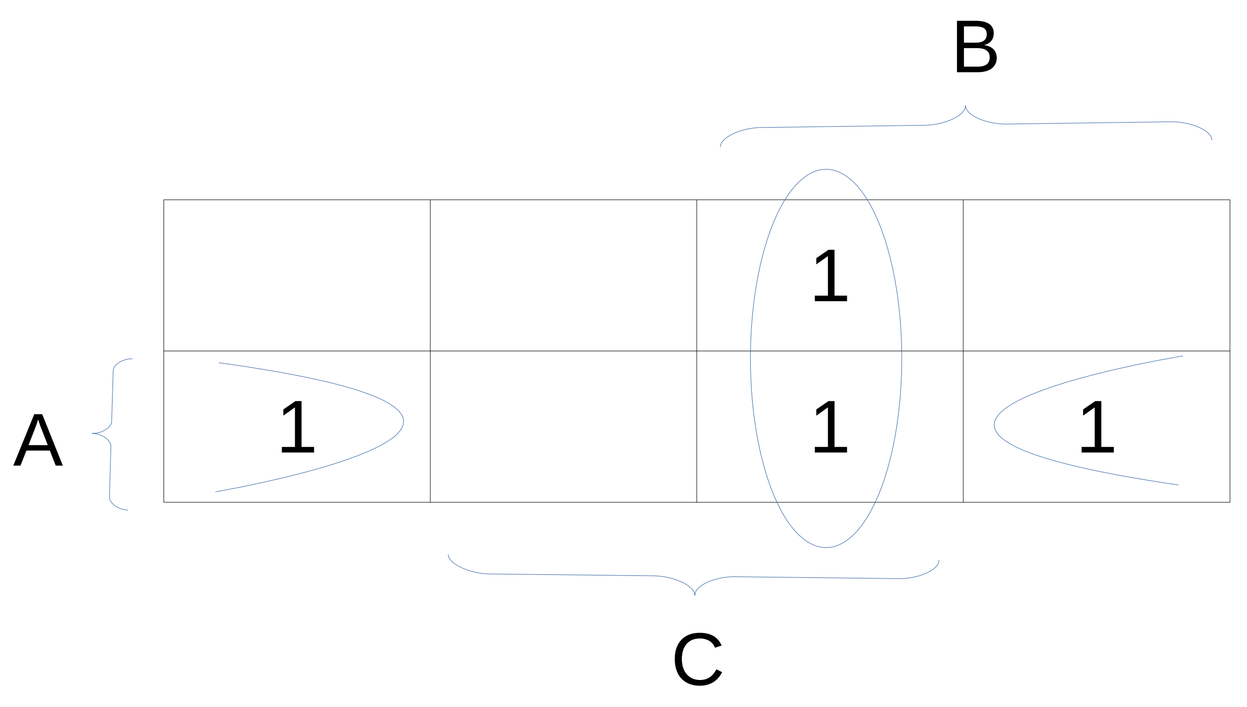


Figure 1-8. Map for F(A, B, C) = Σ(3, 4, 6, 7)

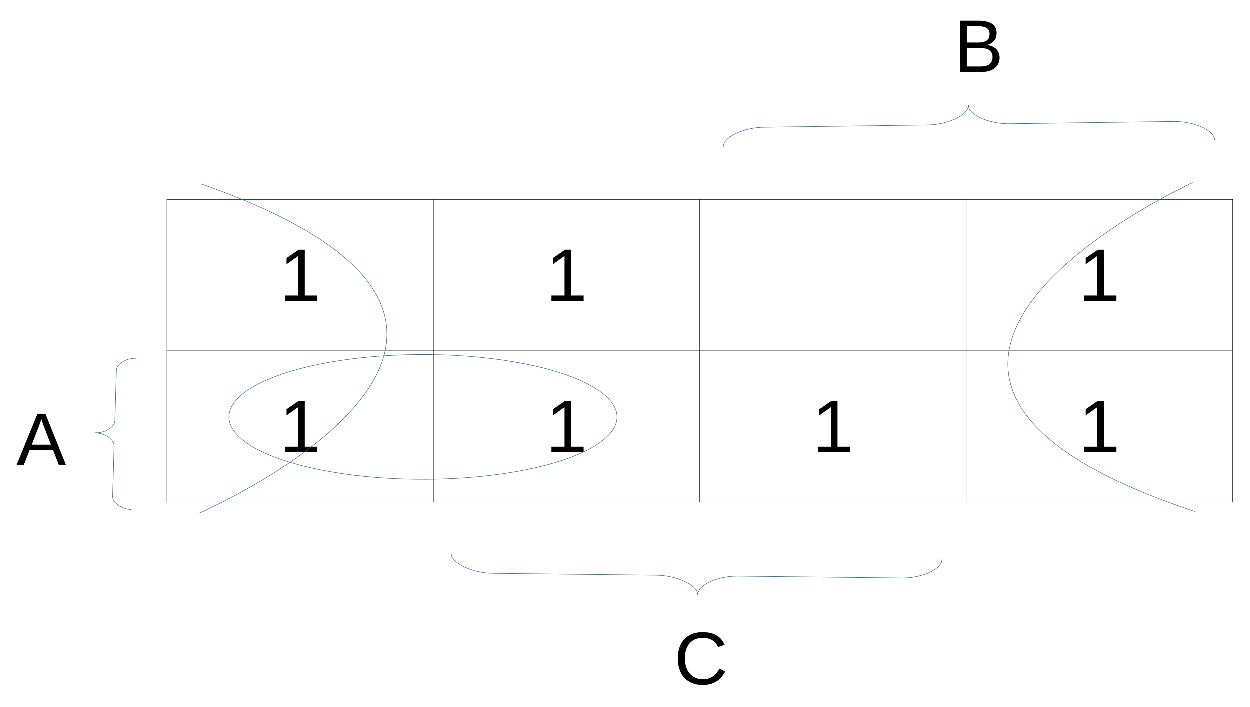


Figure 1-9. Map for F(A, B, C) = Σ(0, 2, 4, 5, 6)

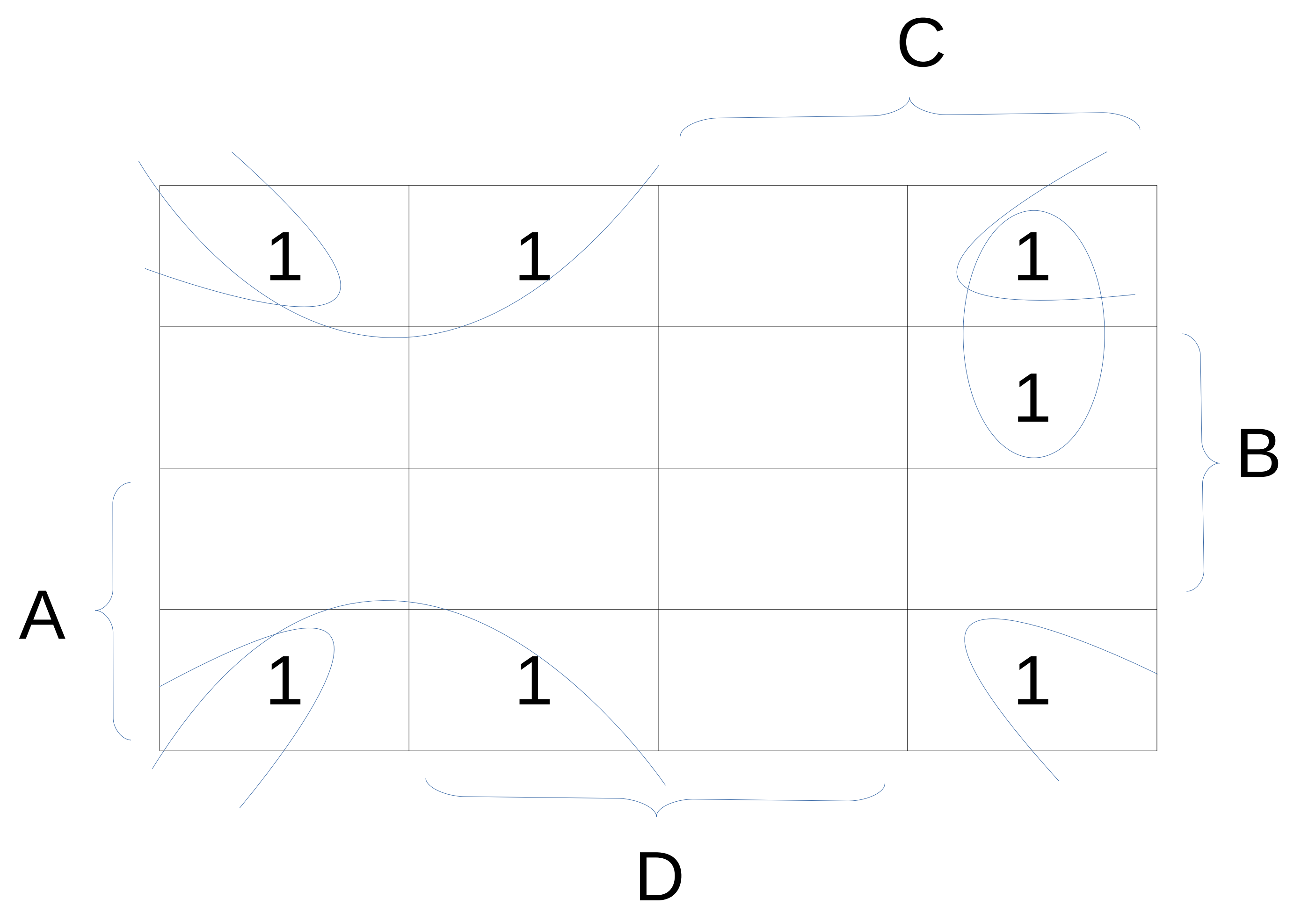


Figure 1-10.
Map for F(A, B, C, D) = Σ(0, 1, 2, 6, 8, 9,10)

The third example needs a four-variable map.

$$F(A, B, C, D) = \Sigma(0, 1, 2, 6, 8, 9, 10)$$

The area in the map covered by this four-variable function consists of the squares marked with 1's in Fig. 1-10. The function contains 1's in the four comers that, when taken as a group, give the term B' D' . This is possible because these four squares are adjacent when the map is considered with top and bottom or left and right edges touching. The two 1's on the left of the top row are combined with the two 1's on the left of the bottom row to give the term B'C' . The remaining 1 in the square of minterm 6 is combined with minterm 2 to give the term A'CD' . The simplified function is

$$F = B'D' + B'C' + A'CD'$$

Product-of-Sums Simplification

The Boolean expressions derived from the maps in the preceding examples were expressed in sum-of-products form. The product terms are AND terms and the sum denotes the ORing of these terms. It is sometimes convenient to obtain the algebraic expression for the function in a product-of-sums form. The sums are OR terms and the product denotes the ANDing of these terms. With a minor modification, a product-of-sums form can be obtained from a map.

The procedure for obtaining a product-of-sums expression follows from the basic properties of Boolean algebra. The 1's in the map represent the minterms that produce 1 for the function. The squares not marked by 1 represent the min terms that produce 0 for the function. If we mark the empty squares with 0' s and combine them into groups of adjacent squares, we obtain the complement of the function, F' . Taking the complement of F' produces an expression for F in product-of-sums form. The best way to show this is by example.

We wish to simplify the following Boolean function in both sum-of products form and product-of-sums form:

$$F(A, B, C, D) = \Sigma(0, 1, 2, 5, 8, 9, 10)$$

The 1' s marked in the map of Fig. 1-11 represent the min terms that produce a 1 for the function. The squares marked with 0' s represent the min terms not included in F and therefore denote the complement of F. Combining the squares with 1's gives the simplified function in sum-of-products form:

$$F = B'D' + B'C' + A'C'D$$

If the squares marked with 0' s are combined, as shown in the diagram, we obtain the simplified complemented function:

$$F' = AB + CD + BD'$$

Taking the complement of F', we obtain the simplified function in product-of-sums form:

$$F = (A' + B')(C' + D')(B' + D)$$

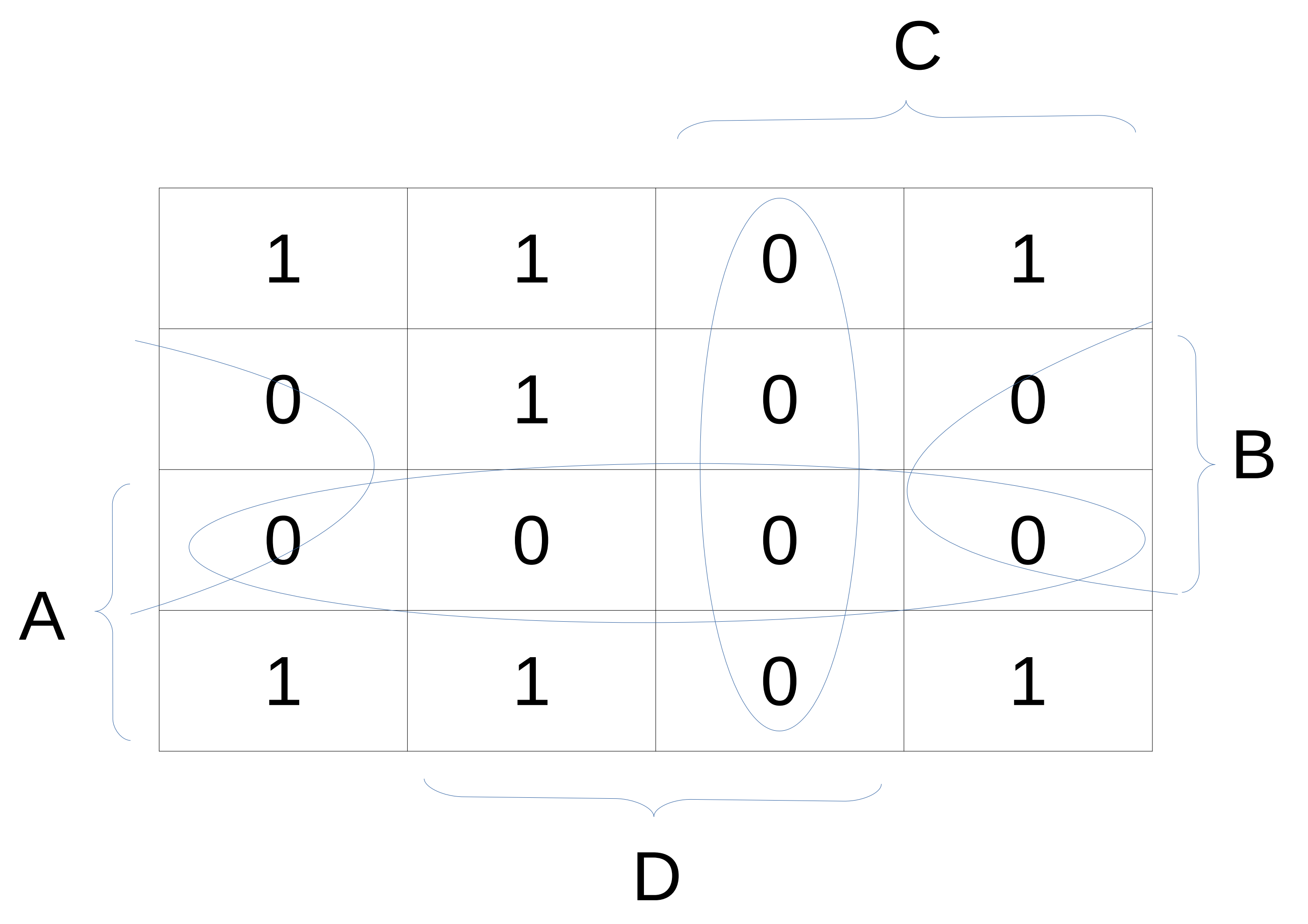
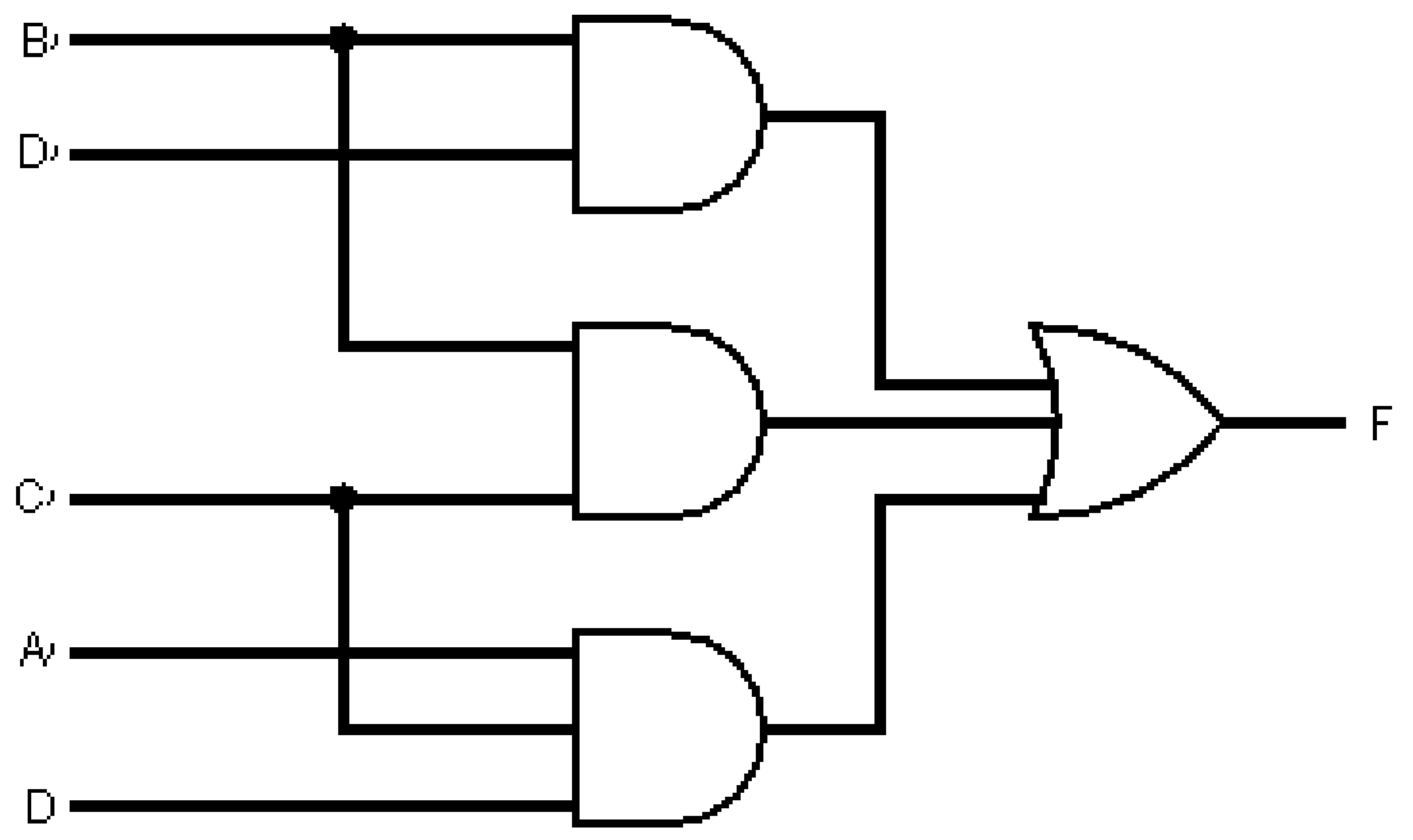


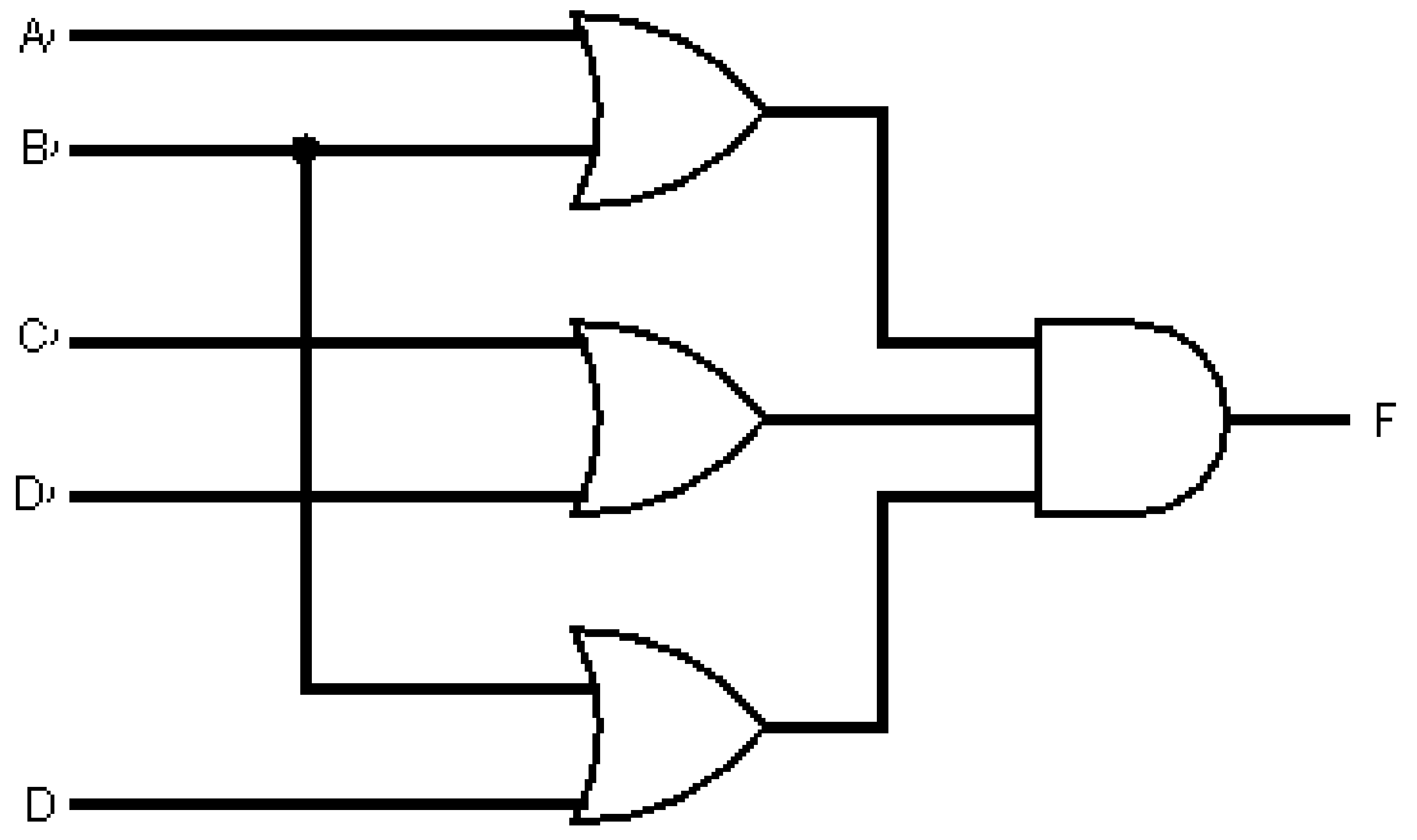
Figure 1-11.
Map for F(A, B, C, D) = Σ(0, 1, 2, 5, 8, 9,10)

The logic diagrams of the two simplified expressions are shown in Fig. 1-12. The sum-of-products expression is implemented in Fig. 1-12(a) with a group of AND gates, one for each AND term. The outputs of the AND gates are connected to the inputs of a single OR gate. The same function is implemented in Fig. 1-12(b) in product-of-sums form with a group of OR gates, one for each OR term. The outputs of the OR gates are connected to the inputs of a single AND gate. In each case it is assumed that the input variables are directly available in their complement, so inverters are not included. The pattern established in Fig. 1-12 is the general form by which any Boolean function is implemented when expressed in one of the standard forms. AND gates are connected to a single OR gate when in sum-of-products form. OR gates are connected to a single AND gate when in product-of-sums form.

Figure 1-12. Logic diagrams with AND and OR gates



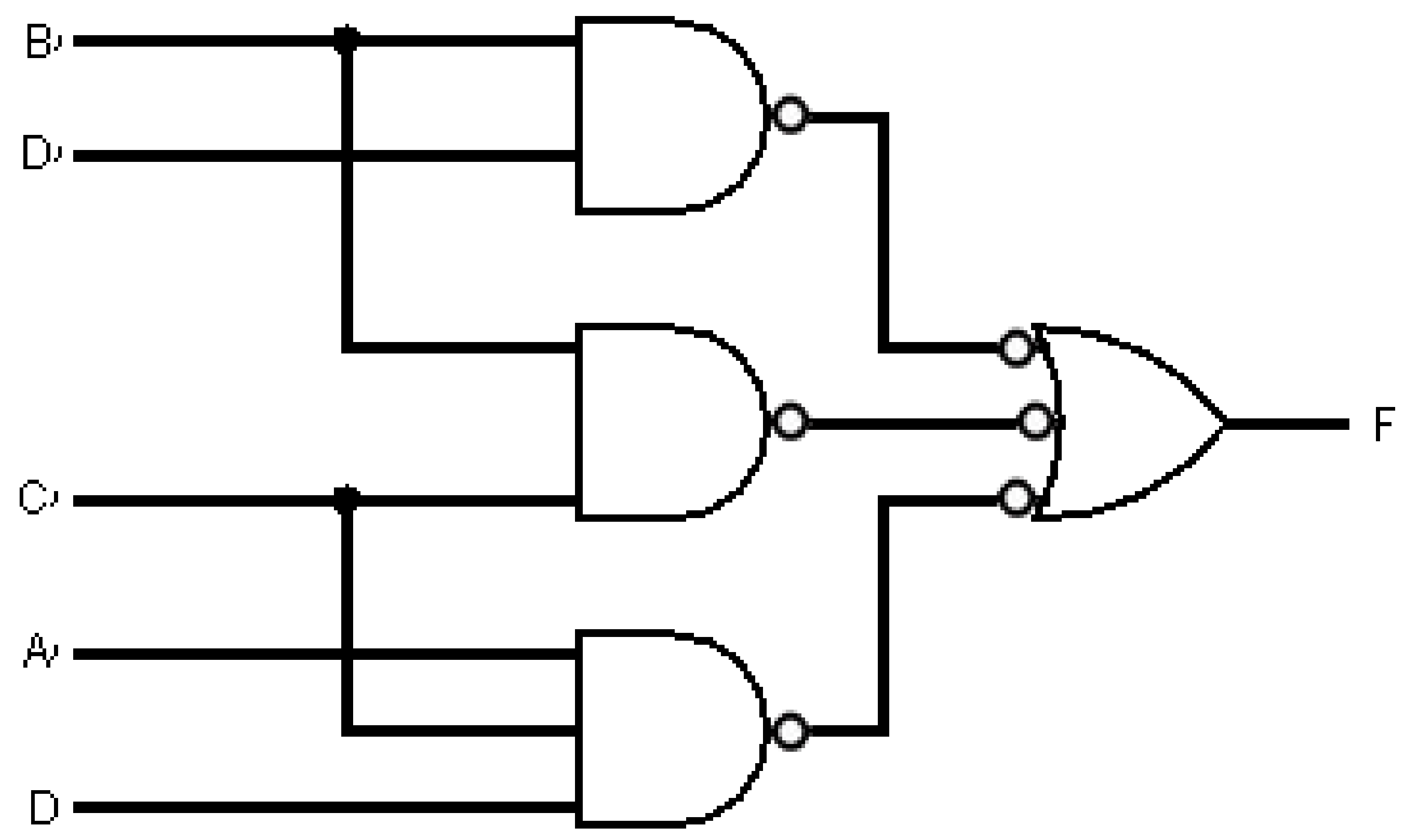
(a) Sum of products: F=B'D + B'C' + A'C'D



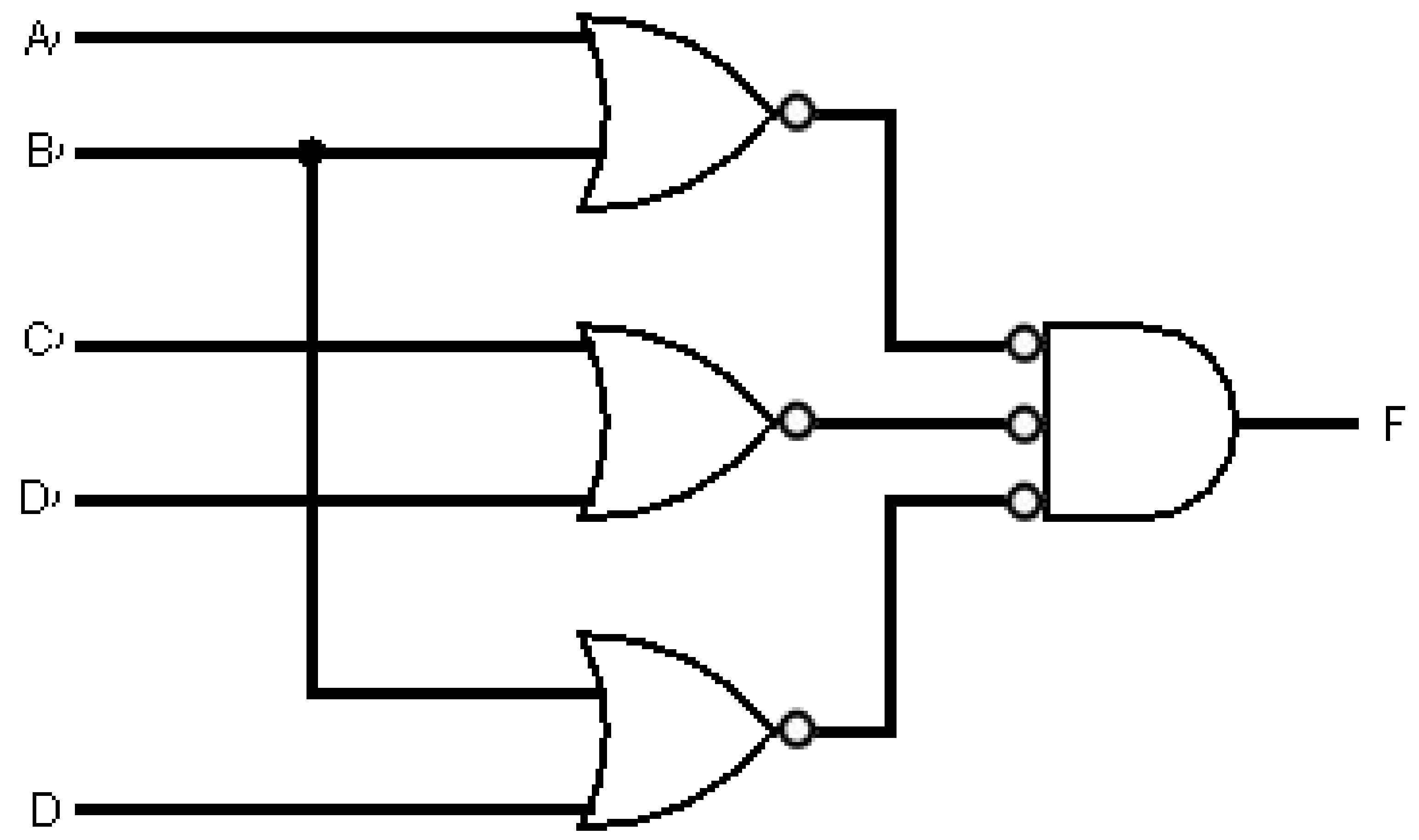
(b) Product of sums: F=(A' + B')(C' + D')(B' + D)

A sum-of-products expression can be implemented with NAND gates as shown in Fig. 1-13(a). Note that the second NAND gate is drawn with the graphic symbol of Fig. 1-5(b). There are three lines in the diagram with small circles at both ends. Two circles in the same line designate double complementation, and since (x ')' = x, the two circles can be removed and the resulting diagram is equivalent to the one shown in Fig. 1-12(a). Similarly, a product-of-sums expression can be implemented with NOR gates as shown in Fig. 1-13(b). The second NOR gate is drawn with the graphic symbol of Fig. 1-4(b). Again the two circles on both sides of each line may be removed, and the diagram so obtained is equivalent to the one shown in Fig. 1-12(b).

Figure 1-13. Logic diagrams with NAND and NOR gates



(a) With NAND gates



(b) With NOR gates

Don't-Care Conditions

The 1' s and 0' s in the map represent the min terms that make the function equal to 1 or 0. There are occasions when it does not matter if the function produces 0 or 1 for a given minterm. Since the function may be either 0 or 1, we say that we don't care what the function output is to be for this min term. Min terms that don't-care conditions may produce either 0 or 1 for the function are said to be don't-care conditions and are marked with an x in the map. These don't-care conditions can be used to provide further simplification of the algebraic expression.

When choosing adjacent squares for the function in the map, the x's may be assumed to be either 0 or 1, whichever gives the simplest expression. In addition, an x need not be used at all if it does not contribute to the simplification of the function. In each case, the choice depends only on the simplification that can be achieved. As an example, consider the following Boolean function together with the don't-care minterms:

$$F(A, B, C) = \Sigma(0, 2, 6)$$
$$d(A, B, C) = \Sigma(1, 3, 5)$$

The minterms listed with F produce a 1 for the function. The don't-care millterms listed with d may produce either a 0 or a 1 for the function. The remaining minterms, 4 and 7, produce a 0 for the function. The map is shown in Fig. 1-14. The minterms of F are marked with 1's, those of d are marked with x's, and the remaining squares are marked with 0's . The 1's and x's are combined in any convenient manner so as to enclose the maximum number of adjacent squares. It is not necessary to include all or any of the x's, but all the 1's must be included. By including the don't-care minterms 1 and 3 with the 1's in the first row we obtain the term A' . The remaining 1 for min term 6 is combined with min term 2 to obtain the term BC' . The simplified expression is

$$F = A' + BC'$$

Note that don't-care minterm 5 was not included because it does not contribute to the simplification of the expression. Note also that if don't-care mmterms 1 and 3 were not included with the 1's, the simplified expression for F would have been

$$F = AC' + BC'$$

This would require two AND gates and an OR gate, as compared to the expression obtained previously, which requires only one AND and one OR gate.

The function is determined completely once the x's are assigned to the 1's or 0's in the map. Thus the expression

$$F = A' + BC'$$

represents the Boolean function

$$F(A, B, C) = \Sigma(0, 1, 2, 3, 6)$$

It consists of the original minterms 0, 2, and 6 and the don't-care minterms 1 and 3. Minterm 5 is not included in the function . Since minterms 1, 3, and 5 were specified as being don't-care conditions, we have chosen min terms 1 and 3 to produce a 1 and minterm 5 to produce a 0. This was chosen because this assignment produces the simplest Boolean expression.

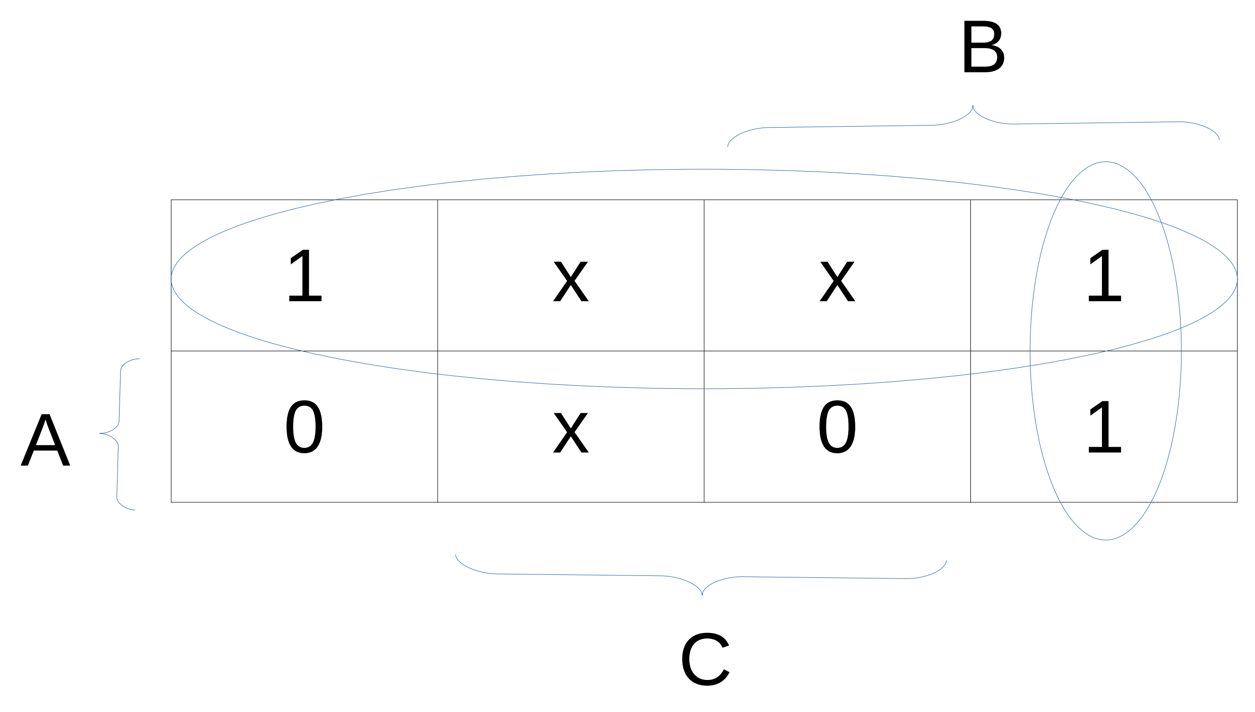


Figure 1-14.
Example of map with don’t-care conditions

General rules to be followed while minimizing the expressions using K-Map which include don't care conditions are as follows,

1. After forming the K-Map, fill 1's at the specified positions corresponding to the given minterms. Fill X at the positions where don't care combinations are present.
2. Now, Encircle the groups in the K-Map. One thing to be kept in mind is, now we can treat Don't Care conditions (X) as 1s if these help in forming the largest groups. No such group can be encircled whose all the elements are X.
3. If still there are 1s left which doesn't get encircled in any of the groups, then these isolated 1s are encircled individually.
4. Now, recheck all the encircled groups, and remove any redundancy if present.
5. Write the Boolean expression for each encircled group.
6. The final minimal expression can be obtained by ORing each Boolean expressions that were obtained from each group.

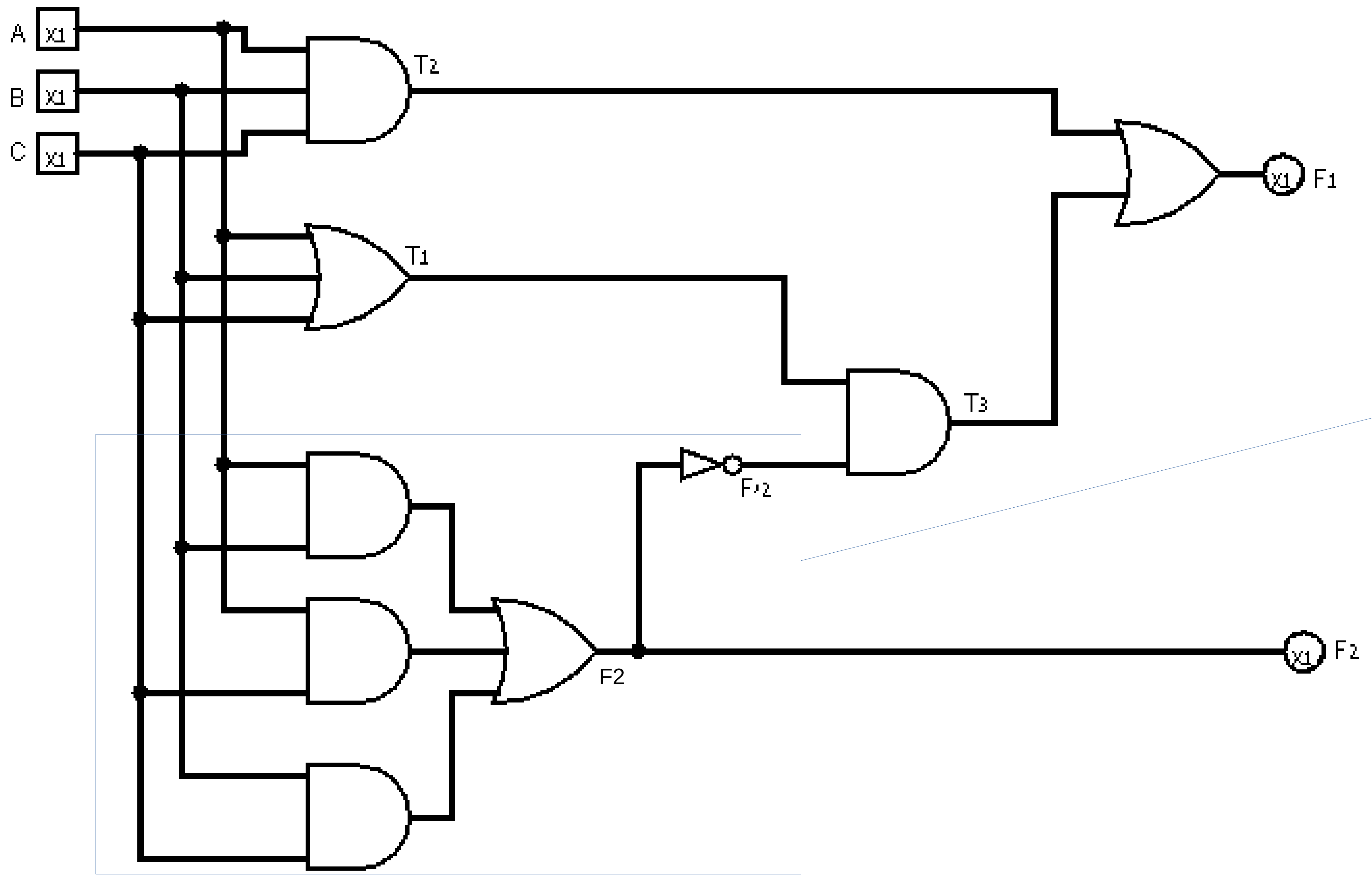
Point to remember:

While designing K-Map using SOP form, don't care conditions (X) are considered as 1, if it helps form the largest group, otherwise it is considered as 0 and are left during encircling. On the contrary, while designing a K-Map using POS form, don't care conditions (X) are considered as a 0, if it helps form the largest group, otherwise it is considered as 1 and are left during encircling.

To obtain the output Boolean functions from a logic diagram, proceed as follows:

1. Label all gate outputs that are a function of input variables with arbitrary symbols. Determine the Boolean functions for each gate output.
2. Label the gates that are a function of input variables and previously labeled gates with other arbitrary symbols. Find the Boolean functions for these gates.
3. Repeat the process outlined in step 2 until the outputs of the circuit are obtained.
4. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables.

Example:

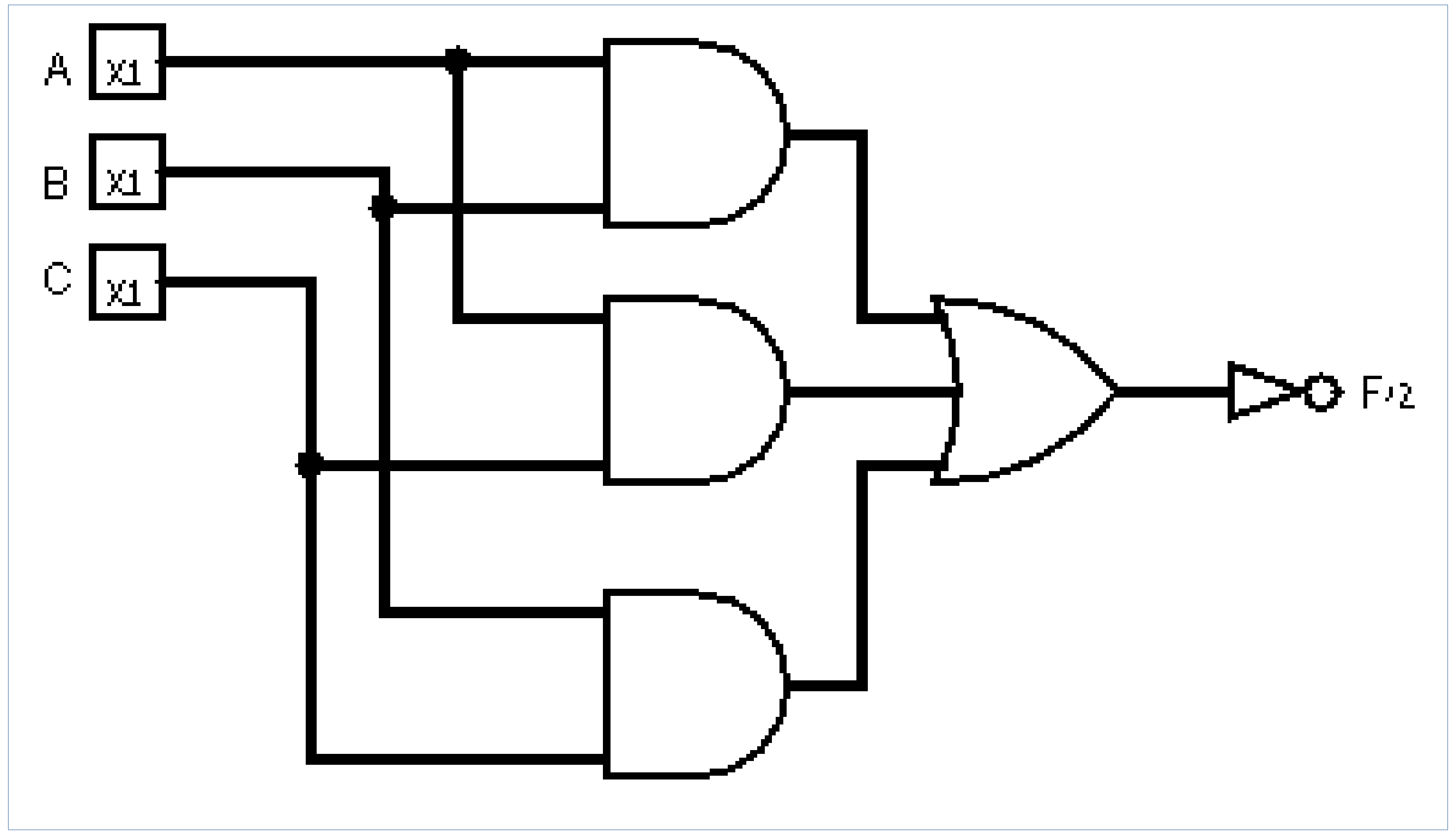


T1 = A + B + C

T2 = ABC

F2 = AB + AC + BC

To derive the expression for F'2, let's simplify the encircled part of the circuit as a circuit by itself



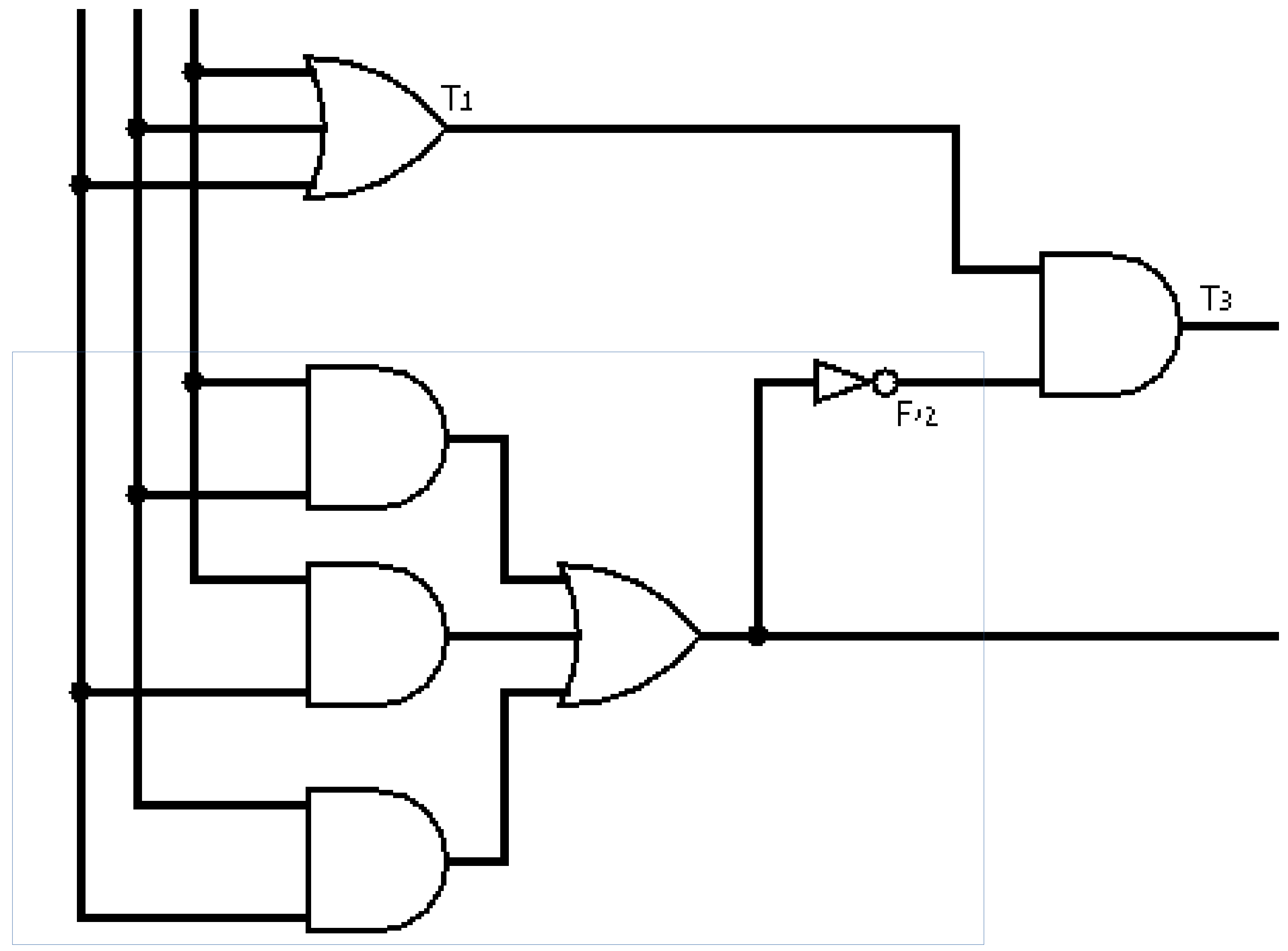
Let's build a truth table fort this circuit:

A	B	C	F'2
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

So, according to the truth table, the K-Map for this circuit is:

BC / A	B'C' 00	B'C 01	BC 11	BC' 10
A' 0	1^{0}	1^{1}	0^{3}	1^{2}
A 1	1^{4}	0^{5}	0^{7}	0^{6}

F'2 = B'C' + A'C' + B'A'



T3 = T1F'2

F'2 = B'C' + A'C' + B'A'

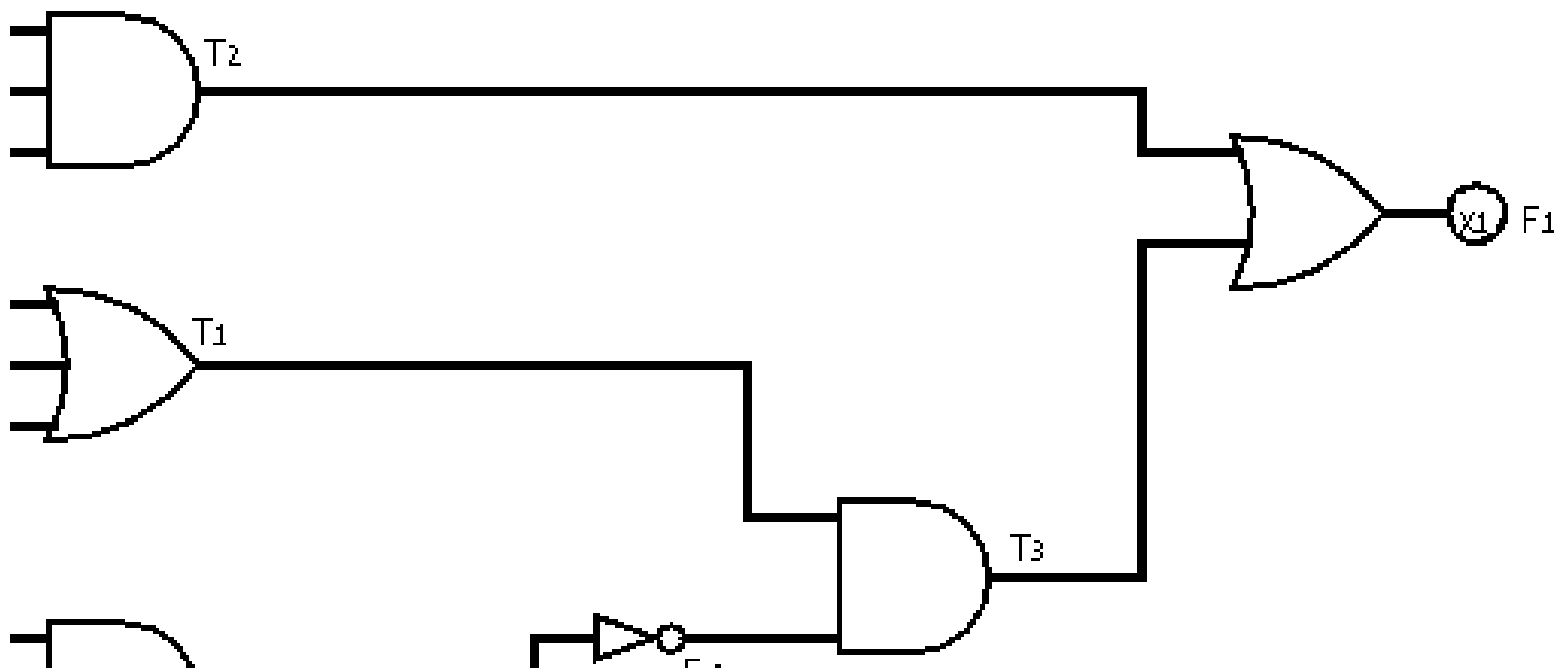
T1 = A + B + C

T3 = (B'C' + A'C' + A'B')(A + B + C) =

= (A+B+C)B'C' + (A+B+C)A'C' + (A+B+C)A'B' =

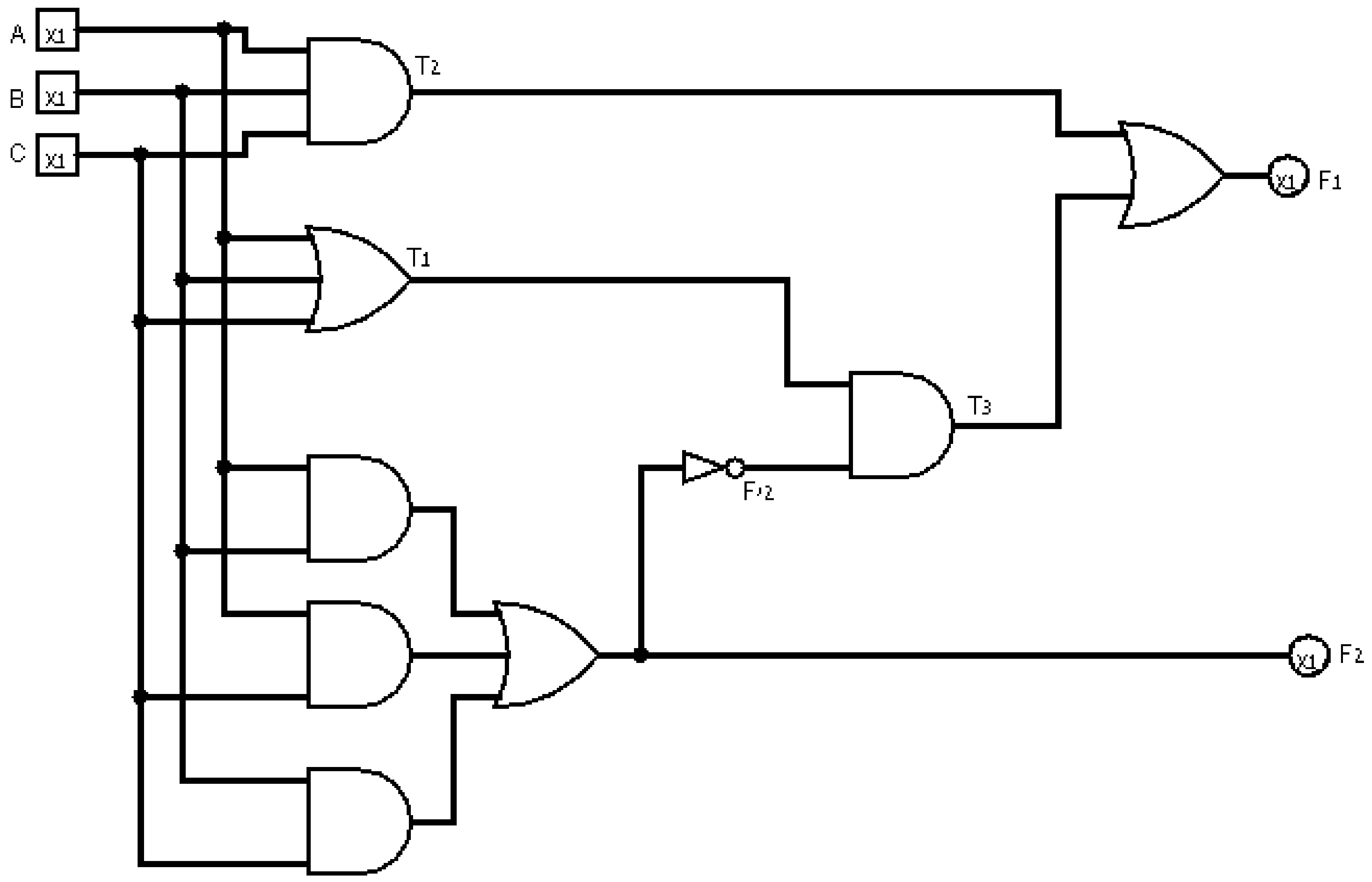
= AB'C'+BB'C'+CC'B'+AA'C'+BA'C'+CC'A'+AA'B'+BB'A'+CA'B' =

= AB'C' + A'BC' + A'B'C



F1 = T3 + T2

F1 = AB'C' + A'BC' + A'B'C + ABC



Finally, the Boolean expressions for this circuit are:

T1 = A + B + C

T2 = ABC

T3 = AB’C’ + A’BC’ + A’B’C

F2 = AB + AC + BC

F1 = AB’C’ + A’BC’ + A’B’C + ABC

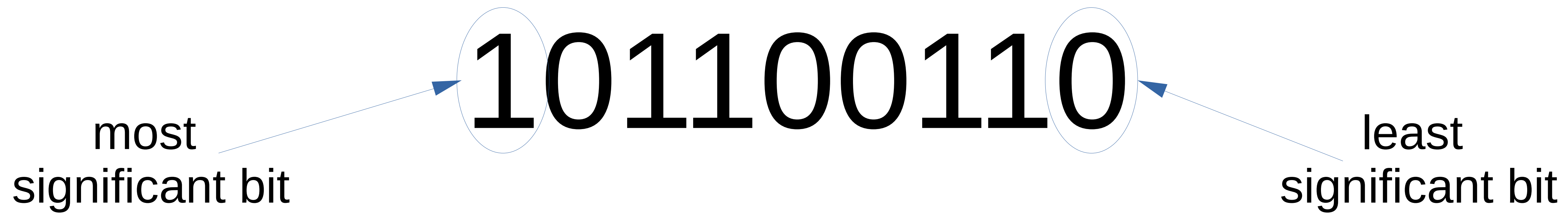
The most and least significant bit

In computer science and information theory, a bit is the smallest possible meaningful piece of information. It is most often expressed as a digit of the binary numeral system: either 0 or 1. A string of 8 bits is called a byte.

If we take, for example, the binary number 11100111 (231 in decimal), and send it as a string of data of a network, we can send it in two ways: starting from left to right, or starting right to left. These two orderings are commonly called Most Significant Bit First, and Least Significant Bit First, respectively.

In this case, we referred to the first, or left-most bit as the Most Significant Bit (MSB for short). The MSB is the bit in a binary sequence that carries the greatest numerical value. For simpler reference, if we take a look at the equivalent decimal number, 231, the most significant digit is the leading 2. Compared to the other two digits, the leading 2 determines the greatest part of the number's numerical value, as it signifies the hundreds in the number. Analogous to this, the leading 1 in our binary number is it's most significant bit.

The least significant bit is the right-most bit in a string. It is called that because it has the least effect on the value of the binary number, in the same way as the unit digit in a decimal number has the least effect on the number's value. The LSB also determines whether the given number is odd or even. The number 11100111 is an odd number, since it's LSB (1) is an odd number. If we use the term least significant bits (plural), we are commonly referring to the several bits closest to, and including, the LSB. Another property of the least significant bits is that they often change drastically if the number changes. For example, if we add 1 to our example number, 11100111, we will get 11101000. The result of this minimal addition is that the four least significant bits have changed their value.



Negative Binary Numbers

Signed Magnitude (MSB as a sign bit)

Usually we represent a negative decimal number by placing a minus sign directly to the left of the most significant digit, just as in the example above, with -5. However, the whole purpose of using binary notation is for constructing on/off circuits that can represent bit values in terms of voltage (2 alternative values: either “high” or “low”).

In this context, we don’t have the luxury of a third symbol such as a “minus” sign, since these circuits can only be on or off (two possible states). One solution is to reserve a bit (circuit) that does nothing but represent the mathematical sign:

$$101_2 = 5_{10} (\text{positive})$$

Extra bit, representing sign (0 = positive, 1 = negative)

$$0101_2 = 5_{10} (\text{positive})$$

$$1101_2 = -5_{10} (\text{negative})$$

As you can see, we have to be careful when we start using bits for any purpose other than standard place-weighted values. Otherwise, 1101_2 could be misinterpreted as the number thirteen when in fact we mean to represent negative five.

To keep things straight here, we must first decide how many bits are going to be needed to represent the largest numbers we’ll be dealing with, and then be sure not to exceed that bit field length in our arithmetic operations.

For the above example, I’ve limited myself to the representation of numbers from negative seven (1111_2) to positive seven (0111_2), and no more, by making the fourth bit the “sign” bit. Only by first establishing these limits can I avoid confusion of a negative number with a larger, positive number.

Representing negative five as (1101_2) is an example of the sign-magnitude system of negative binary numeration. By using the leftmost bit as a sign indicator and not a place-weighted value, I am sacrificing the “pure” form of binary notation for something that gives me a practical advantage: the representation of negative numbers.

The leftmost bit is read as the sign, either positive or negative, and the remaining bits are interpreted according to the standard binary notation: left to right, place weights in multiples of two.

The method’s drawback:

As simple as the sign-magnitude approach is, it is not very practical for arithmetic purposes. For instance, how do I add a negative five (1101_2) to any other number, using the standard technique for binary addition?

For example, let’s add a negative five (1101_2) to a positive five (0101_2) using the MSB as a sign bit:

$$\begin{array}{rl} 0101 & \Rightarrow 5 \\ +\,1101 & \Rightarrow -5 \\ \hline 10010 & \Rightarrow 2;\ 2 \neq 0 \end{array}$$

Since we’re use only 4 bits in this example, so the leftmost bit of the result needs to be ignored, so the result is 0010_2,
that is decimal 2, and 2_{10} isn’t the expected 0.

Let’s see another example, where we’ll try to subtract a binary 7 from binary 14 (by adding 7_2 to 14_2), using 8 bits and reserving the MSB as a sign bit:

```
  00001110   => 14
+ 10000111   => -7
  10010101   => 149; 149≠7
```

So, in this example, the result of the calculation is 10010101_2, that is decimal 149, and so, this is very far from the expected decimal 7.

So, as we can see, when we use the MSB as a sign bit, the binary arithmetics simply stop working. Obviously, this isn't a good solution for signing binary numbers.

Complementation

I'd have to invent a new way of doing addition in order for it to work, and if I do that, I might as well just do the job with longhand subtraction; there's no arithmetical advantage to using negative numbers to perform subtraction through addition if we have to do it with sign-magnitude numeration, and that was our goal!

There's another method for representing negative numbers which works with our familiar technique of longhand addition, and also happens to make more sense from a place-weighted numeration point of view, called complementation.

1. One's Complementation

Just as the name implies, using this method we make a binary number negative by its complementation, which means an inversion of all its bits. All the zeroes in the number become ones, and vice versa. For instance, the binary for 17 is 00010001 (using 8 bits magnitude), so negative 17 will be 11101110.

There're binary numbers from 0 to 7 (4-bit magnitude) and their negative counterparts, using one's complementation negation. Binary 7 (0111) is the maximal number that can be represented with 4-bit magnitude and still has an MSB value "0".

1 0 0 0	-7
1 0 0 1	-6
1 0 1 0	-5
1 0 1 1	-4
1 1 0 0	-3
1 1 0 1	-2
1 1 1 0	-1
1 1 1 1	-0
0 0 0 0	0
0 0 0 1	1
0 0 1 0	2
0 0 1 1	3
0 1 0 0	4
0 1 0 1	5
0 1 1 0	6
0 1 1 1	7

Let's do some arithmetics with those numbers. As we know from the regular arithmetics, a sum of two numbers, equals by the value but opposite by the sign, always equals 0: -5 + 5 = 0, -1 + 1 = 0, -7 + 7 = 0, etc. So, let's try to do this with binary numbers, using one's complement negation:

```
  0101   => 5
+ 1010   => -5
  ----
  1111   => -0
```

Although negative (signed) zero, we did get as the result is a pretty strange concept, the result is quite close to the expected ("0").
Now, let's to the same with 3:

```
  0011   => 3
+ 1100   => -3
  ----
  1111   => -0
```

So, the result is also negative 0.

```
  0111   => 7
+ 1000   => -7
  ----
  1111   => -0
```

So, with this negation method, when adding to a number its opposite signed counterpart, we'll always get the negative zero in the result.

But let's add some numbers that are different not only by the sign but also by the value.

```
  0101    => 5
+
  1100    => -3
 ------
 10001    => 1
```

Since we’re working with only 4 bits order of magnitude, we need to ignore the leftmost bit of the result (“1”). Taking this into account, the result that we obtain is 1, while the expected result is 2 (-3 + 5 = 2).

```
  0110    => 6
+
  1101    => -2
 ------
 10011    => 3
```

Ignoring the leftmost bit, the result is 3, lesser by 1 than the expected result 4.

So, as we can see, in those examples the results always lesser than expected by 1,so, to obtain the right result, we just need to add "1" to the result:

$$3+(-5)=0101+1100=1_2(\text{ignoring the leftmost bit that out of the magnitude})\Rightarrow 1_2+1_2=2_2$$

$$6+(-2)=0110+1101=3_2(\text{ignoring the leftmost bit that out of the magnitude})\Rightarrow 3_2+1_2=4_2$$

But adding 1 to the obtained result is work only if the negative number is lesser than the magnitude: in the above examples, the magnitude is 4 bits, and the negative numbers are -3 and -2. Let’s see how to sum a positive and a negative number in the case when the negative number is greater than the magnitude:

```
  0100    => 4
+
  1010    => -5
 ------
  1110    => 14
```

Now we’re about to add 4 and -5. In this case, the negative number, -5, is greater than the order of magnitude (4 bits) by its absolute value. The intermediate result is 1110 (decimal 14).
Because the negative number -5 is greater (by the absolute value) than the magnitude (4), we need to treat it differently. To obtain the end result, we need to complement this number: $\bar{1}\bar{1}\bar{1}\bar{0}=0001$
The obtained result is binary 1, and it is negative, because 4 + (-5) = -1

But, using one’s complement negation, we still have the signed zero problem. So, to avoid this, another binary negation method is used.

2. Two’s Complementation

1 0 0 0	-8
1 0 0 1	-7
1 0 1 0	-6
1 0 1 1	-5
1 1 0 0	-4
1 1 0 1	-3
1 1 1 0	-2
1 1 1 1	-1
0 0 0 0	0
0 0 0 1	1
0 0 1 0	2
0 0 1 1	3
0 1 0 0	4
0 1 0 1	5
0 1 1 0	6
0 1 1 1	7

There’re binary numbers from 0 to 7 on the positive scale and to -8 on the negative scale using 2’s complement (4-bit magnitude).

To negate a number using this method, firstly we need complement a number inverting all its digits as with 1’s complementation, and add 1 to the result.

For example, decimal 5 is binary 0101, so: $-5_{10}=\bar{0}\bar{1}\bar{0}\bar{1}_2+1_2=1011_2$

Now let’s do some math using 2’s complementation.
As in the previous examples, we perform subtraction by adding a negation of the subtrahend (the number which is about to be subtracted):

$$6_{10}-2_{10}\Rightarrow 6_{10}+\left(-2_{10}\right)\Rightarrow 0110_2+\boxed{1110_2}$$

Two’s complement of positive binary 2

If the result of the subtraction has a final carry, the result is positive and it is in its true form. The final carry needs to be ignored.
If the result doesn’t have a final carry, the result in its 2’s complement form.

Let’s see an example 1:

$$1001_2-0100_2=1001_2+\left(-0100_2\right)=1001_2+1100_2$$

$$\begin{array}{r} 1001 \\ +\ 1100 \\ \hline 10101 \end{array}$$

In this case, the result has a final carry (the leftmost “1”).
Because we have the final carry, the result is positive and in its true form.
We need to neglect the carry, and the final result is **101**. (5_{10})

The final carry bit acts as a sign for the answer. If it is 1 then the answer is positive and if it is 0 then the answer is negative

The case when we have a final carry is called overflow because the resulting magnitude has more bits than the magnitude of the numbers we operate with. For instance, in the previous example, we did add two 4-bit binary numbers, but the result has 5 bits. In this case, the leftmost bit is the final carry that did overflow out of the 4-bit register.

Condition for overflow:

$\bar{x} * \bar{y} * z + x * y * \bar{z}$ = 0 => **NO** overflow
= 1 => overflow

Where **x** and **y** are the MSB's of two numbers, **z** is the MSB of the result

Example 2:

$$0110_2 - 1011_2 = 0110_2 + (-1011_2) = 0110_2 + 0101_2$$

$$\begin{array}{r} 0110 \\ + \; 0101 \\ \hline 1011 \end{array}$$

In this case, we don't have a final carry in the result.
This means that the final result is negative, and to obtain it we need to do 2's complement of this intermediate result:

$$(-1011) = \bar{1}\,\bar{0}\,\bar{1}\,\bar{1} + 1 = 0100 + 1 = 101$$

Because the decimal equivalent of 101 is 5, and the result is negative, the decimal answer is -5

Combinational Circuits

A combinational circuit is a connected arrangement of logic gates with a set of inputs and outputs. At any given time, the binary values of the outputs are a function of the binary combination of the inputs. A block diagram of a combinational circuit is shown in Fig. 1-15. The n binary input variables come from an external source, the m binary output variables go to an external destination, and in between there is an interconnection of logic gates. A combinational circuit transforms binary information from the given input data to the required output data. Combinational circuits are employed in digital computers for generating binary control decisions and for providing digital components required for data processing. A combinational circuit can be described by a truth table showing the binary relationship between the n input variables and the m output variables. The truth table lists the corresponding output binary values for each of the 2^n input combinations. A combinational circuit can also be specified with m Boolean functions, one for each output variable. Each output function is expressed in terms of the n input variables.

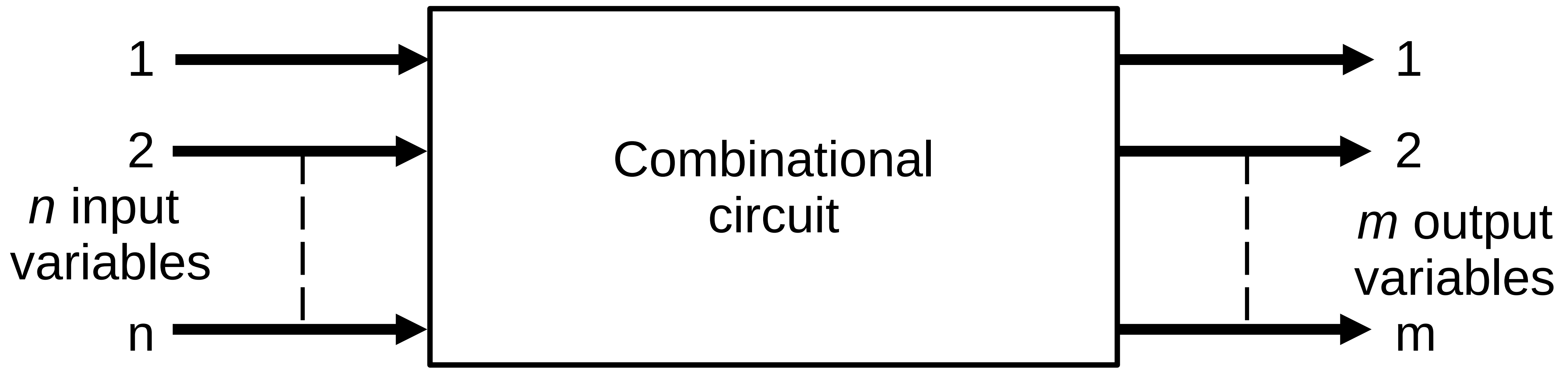


Figure 1-15. Block diagram of a combinational circuit

The analysis of a combinational circuit starts with a given logic circuit diagram and culminates with a set of Boolean functions or a truth table. If the digital circuit is accompanied by a verbal explanation of its function, the Boolean functions or the truth table is sufficient for verification. If the function of the circuit is under investigation, it is necessary to interpret the operation of the circuit from the derived Boolean functions or the truth table. The success of such investigation is enhanced if one has experience and familiarity with digital circuits. The ability to correlate a truth table or a set of Boolean functions with an information-processing task is an art that one acquires with experience. The design of combinational circuits starts from the verbal outline of the problem and ends in a logic circuit diagram. The procedure involves the following steps:

1. The problem is stated.
2. The input and output variables are assigned letter symbols.
3. The truth table that defines the relationship between inputs and outputsis derived.
4. The simplified Boolean functions for each output are obtained.
5. The logic diagram is drawn.

To demonstrate the design of combinational circuits, we present two examples of simple arithmetic circuits. These circuits serve as basic building blocks for the construction of more complicated arithmetic circuits.

Half-Adder

The most basic digital arithmetic circuit is the addition of two binary digits. A combinational circuit that performs the arithmetic addition of two bits is called a half-adder. One that performs the addition of three bits (two significant bits and a previous carry) is called a full-adder. The name of the former stems from the fact that two half-adders are needed to implement a full-adder. The input variables of a half-adder are called the augend and addend bits. The output variables the sum and carry. It is necessary to specify two output variables because the sum of 1 + 1 is binary 10, which has two digits. We assign symbols x and y to the two input variables, and S (for sum) and C (for carry) to the two output variables. The truth table for the half-adder is shown in Fig. I-16(a). The C output is 0 unless both inputs are I. The S output represents the least significant bit of the sum. The Boolean functions for the two outputs can be obtained directly from the truth table:

$$S = x'y + xy' = x \oplus y$$
$$C = xy$$

The logic diagram is shown in Fig. I-16(b). It consists of an exclusive-OR gate and an AND gate.

Half-Adder is used to add a single bit numbers
It doesn’t take carry from previous sum!

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

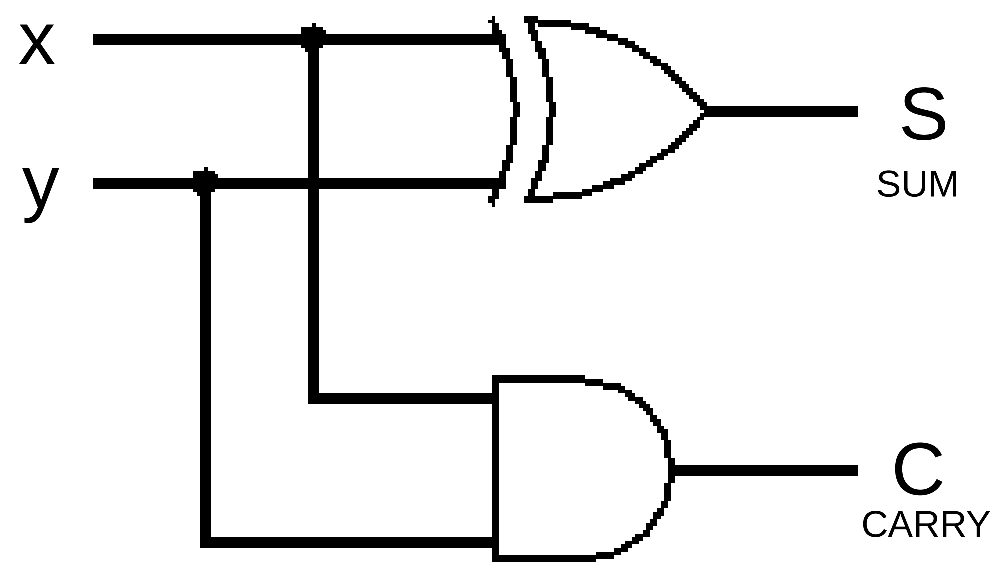


Figure 1-6. Half-adder

Half adder is performing an arithmetical addition of 2 bits. Full adder id performing an addition of 3 bits. Two half-adders are needed to implement a one full adder.

Full-Adder

A full-adder is a combinational circuit that forms the arithmetic sum of three input bits. It consists of three inputs and two outputs. Two of the input variables, denoted by x and y, represent the two significant bits to be added. The third input, z, represents the carry from the previous lower significant position. Two outputs are necessary because the arithmetic sum of three binary digits ranges in value from 0 to 3, and binary 2 or 3 needs two digits. The two outputs are designated by the symbols S (for sum) and C (for carry). The binary variable S gives the value of the least significant bit of the sum. The binary variable C gives the output carry. The truth table of the full-adder is shown in Table 1-2. The eight rows under the input variables designate all possible combinations that the binary variables may have. The value of the output variables are determined from the arithmetic sum of the input bits. When all input bits are 0, the output is 0. The S output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to 1. The C output has a carry of 1 if two or three inputs are equal to 1

Inputs			Outputs	
x	y	z	**C**	**S**
0	0	0	**0**	**0**
0	0	1	**0**	**1**
0	1	0	**0**	**1**
0	1	1	**1**	**0**
1	0	0	**0**	**1**
1	0	1	**1**	**0**
1	1	0	**1**	**0**
1	1	1	**1**	**1**

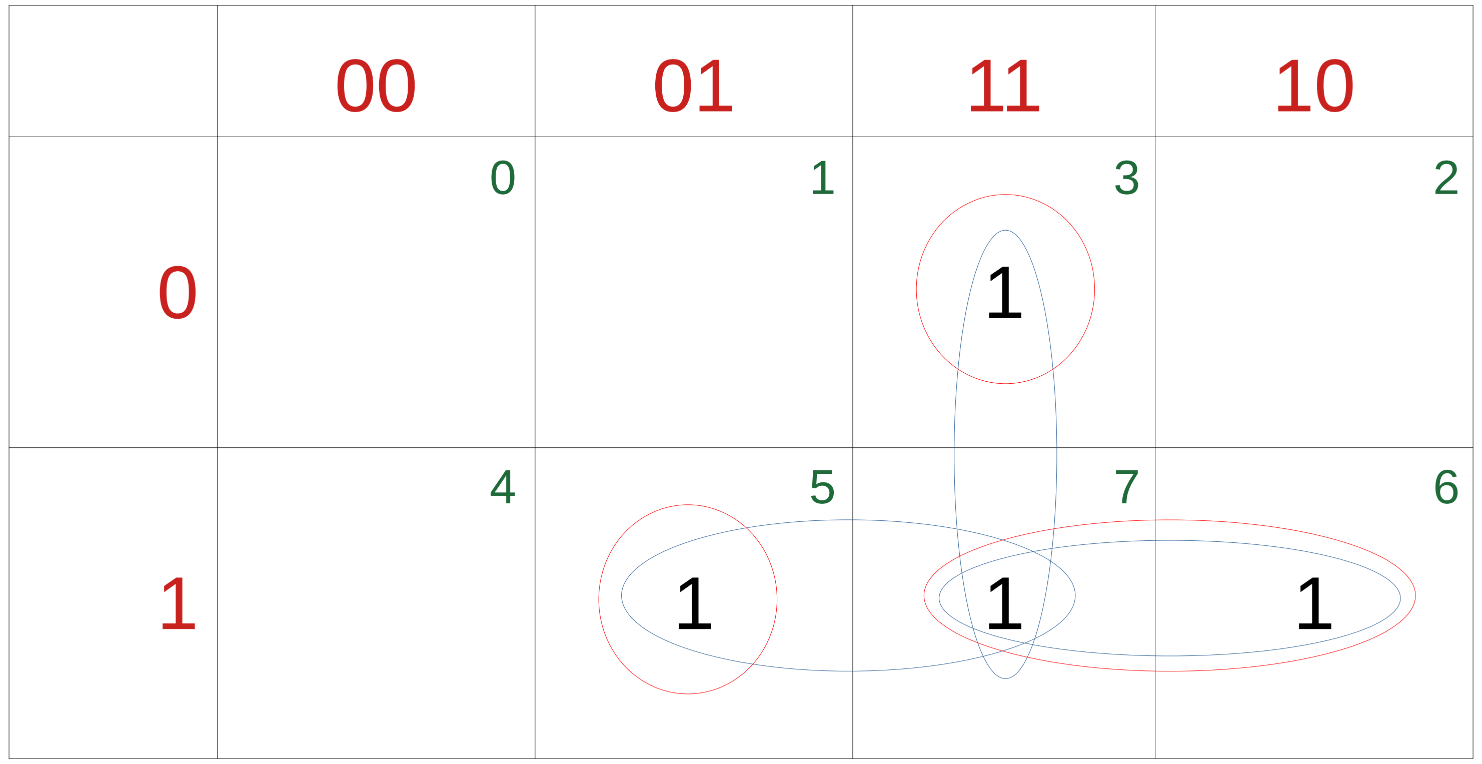
Table 1-2.
Truth table for Full-Adder

The maps of Fig. 1-17 are used to find algebraic expressions for the two output variables. The 1's in the squares for the maps of S and C are determined directly from the minterms in the truth table. The squares with 1's for the S output do not combine in groups of adjacent squares. But since the output is 1 when an odd number of inputs are 1, S is an odd function and represents the exclusive-OR relation of the variables. The squares with 1's for the C output may be combined in a variety of ways. One possible expression for C is

$$C = xy + (x'y + xy')z$$

	00	01	11	10
0	^{0}	^{1} 1	^{3}	^{2} 1
1	^{4} 1	^{5}	^{7} 1	^{6}

S = x’y’z + x’yz’ + xy’z’ + xyz = (x ⊕ y) ⊕ z



	00	01	11	10
0	^{0}	^{1}	^{3} 1	^{2}
1	^{4}	^{5} 1	^{7} 1	^{6} 1

C = xy + xz + yz

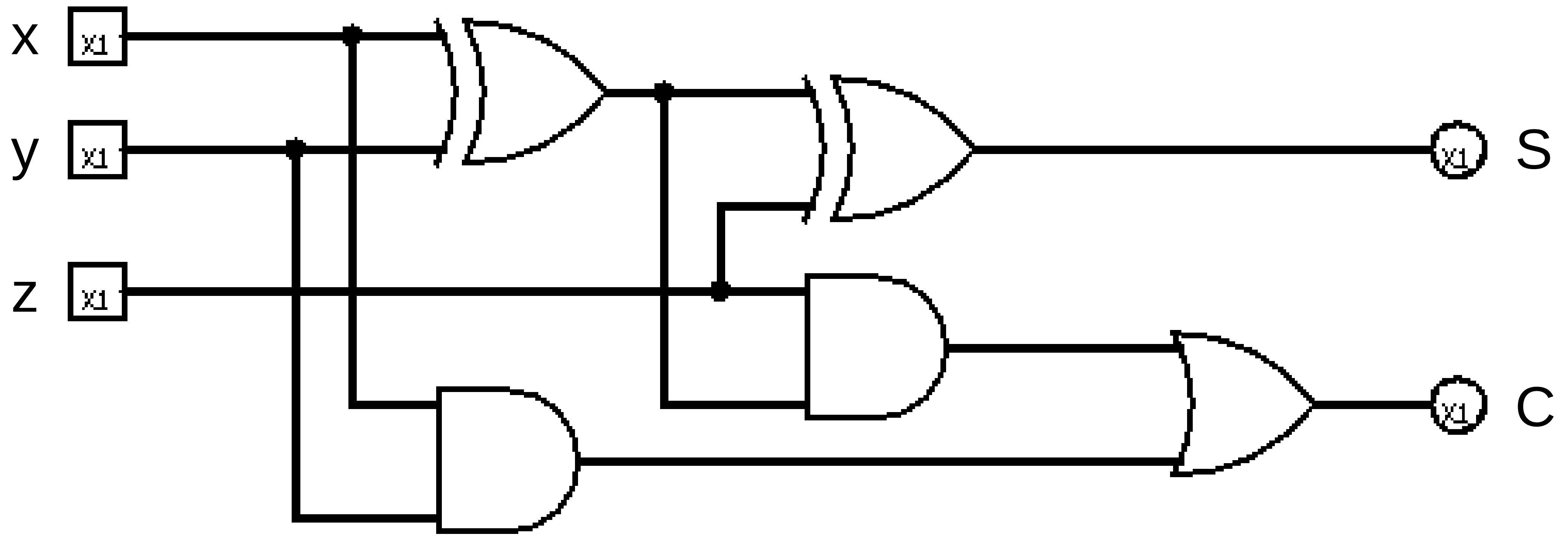
C = xy + (x’y + xy’)z

Figure 1-17. Maps for Full-Adder

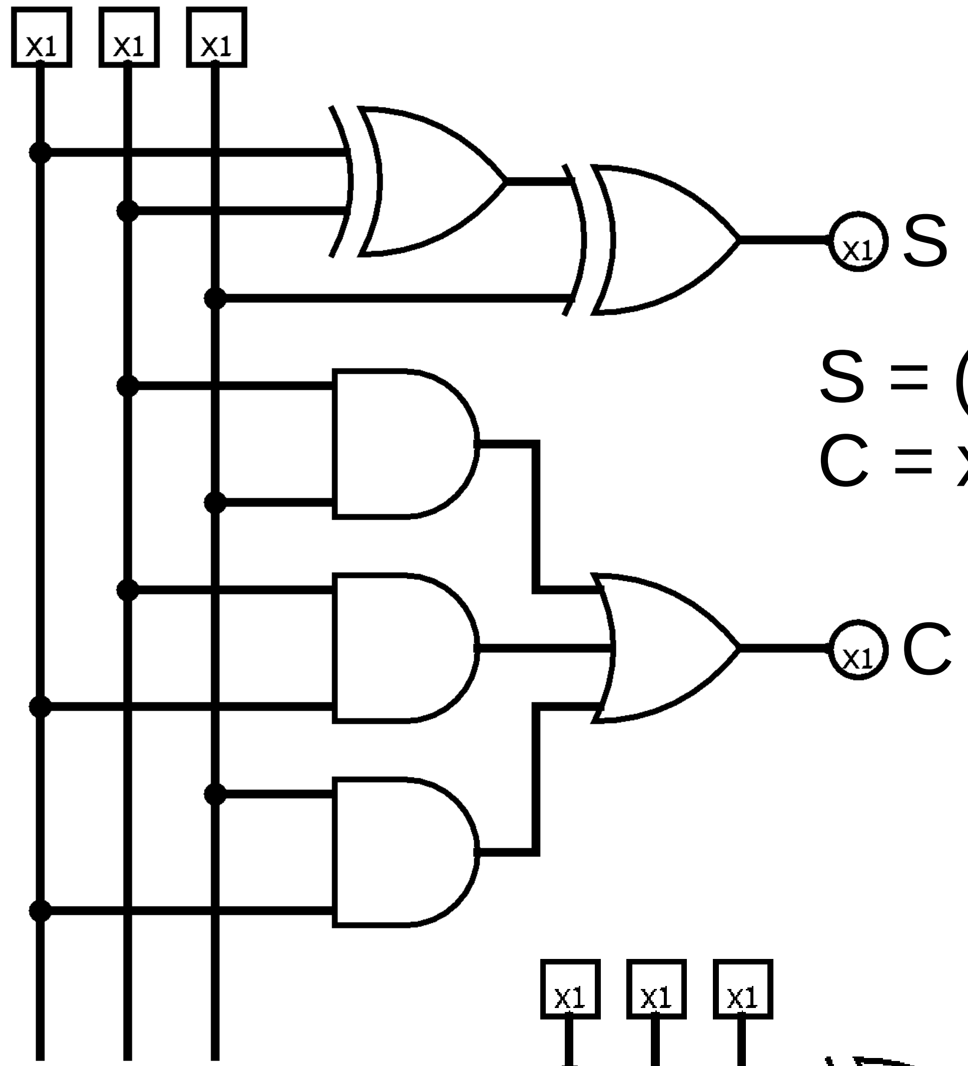
Realizing that x'y + xy' = x ⊕ y and including the expression for output S , we obtain the two Boolean expressions for the full-adder:

$$S = (x \oplus y) \oplus z$$
$$C = xy + (x \oplus y)z$$

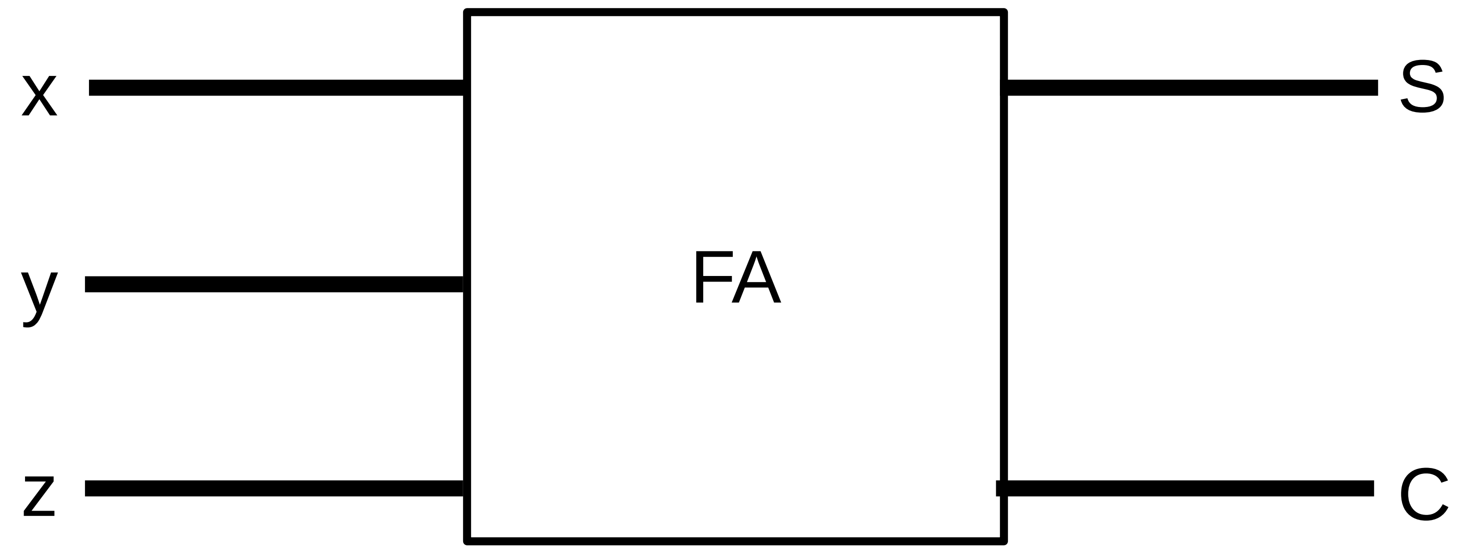
The logic diagram of the full-adder is drawn in Fig. 1-18. Note that the full-adder circuit consists of two half-adders and an OR gate. In the subsequent lessons, the full-adder (FA) may be designated by a block diagram as shown in Fig. 1-18(b).



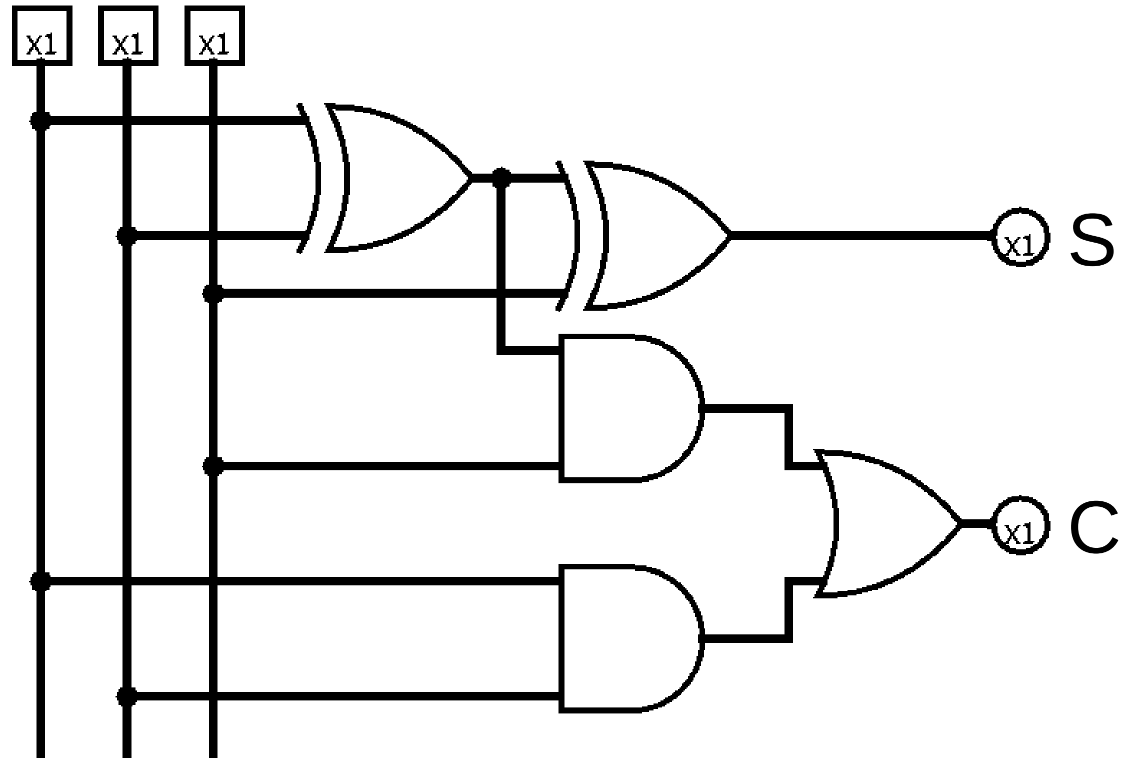
(a) Logic diagram



$S = (x \oplus y) \oplus z$
$C = xy + xz + yz$



(b) Block diagram



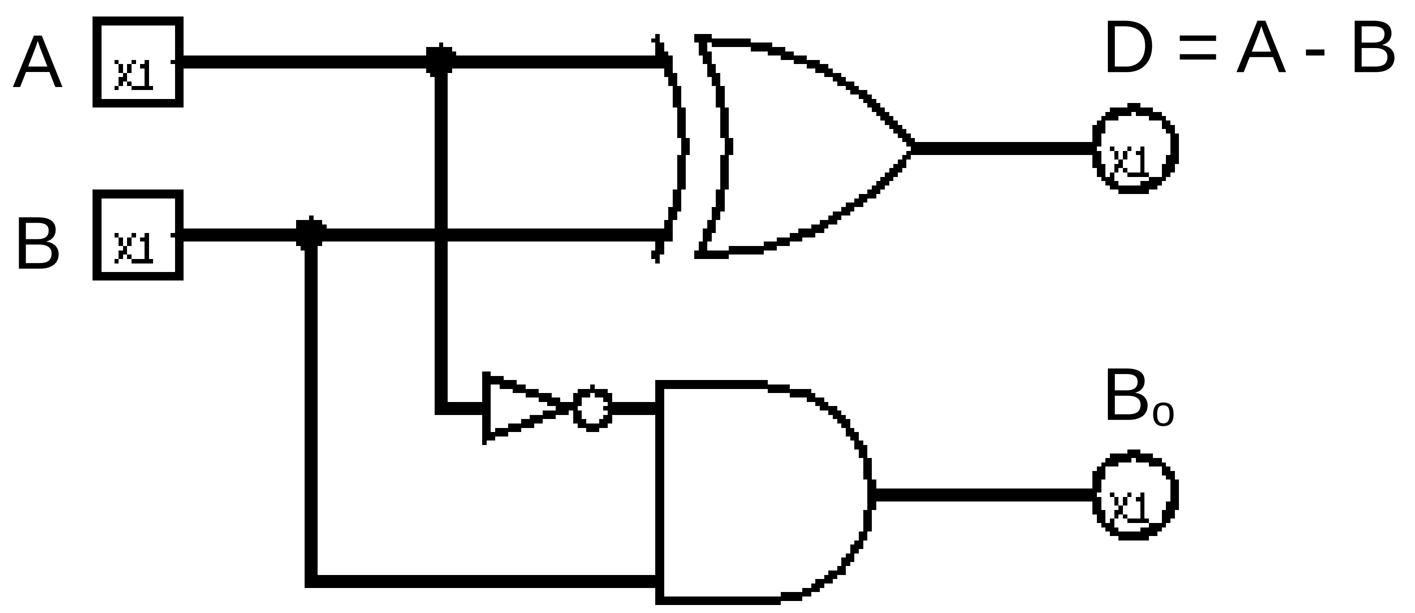
$S = (x \oplus y) \oplus z$
$C = xy + (x \oplus y)z$

Figure 1-18. Full-Adder circuit

Half-Substractor

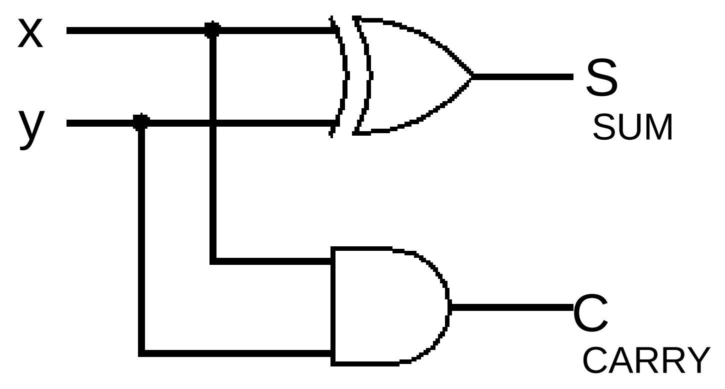
A half-subtractor is a combinational circuit that can be used to subtract one binary digit from another to produce a DIFFERENCE output and a BORROW output. The BORROW output here specifies whether a ‘1‘ has been borrowed to perform the subtraction. The truth table of a half-subtractor, explains this further. The Boolean expressions for the two outputs are given by the equations:

$$D = A'B + AB' = A \oplus B$$
$$B_o = A'B$$



A	B	D	B_o
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

It is obvious that there is no further scope for any simplification of the Boolean expressions given by above equations. While the expression for the DIFFERENCE (D) output is that of an XOR gate, the expression for the BORROW output (B_o) is that of an AND gate with input A complemented before it is fed to the gate. Comparing a half-subtractor with a half-adder, we find that the expressions for the SUM and DIFFERENCE outputs are just the same. The expression for BORROW in the case of the half-subtractor is also similar to what we have for CARRY in the case of the half-adder. If the input A, that is, the minuend, is complemented, an AND gate can be used to implement the BORROW output. Note the similarities between the logic diagrams of a half-adder and a half-subtractor.

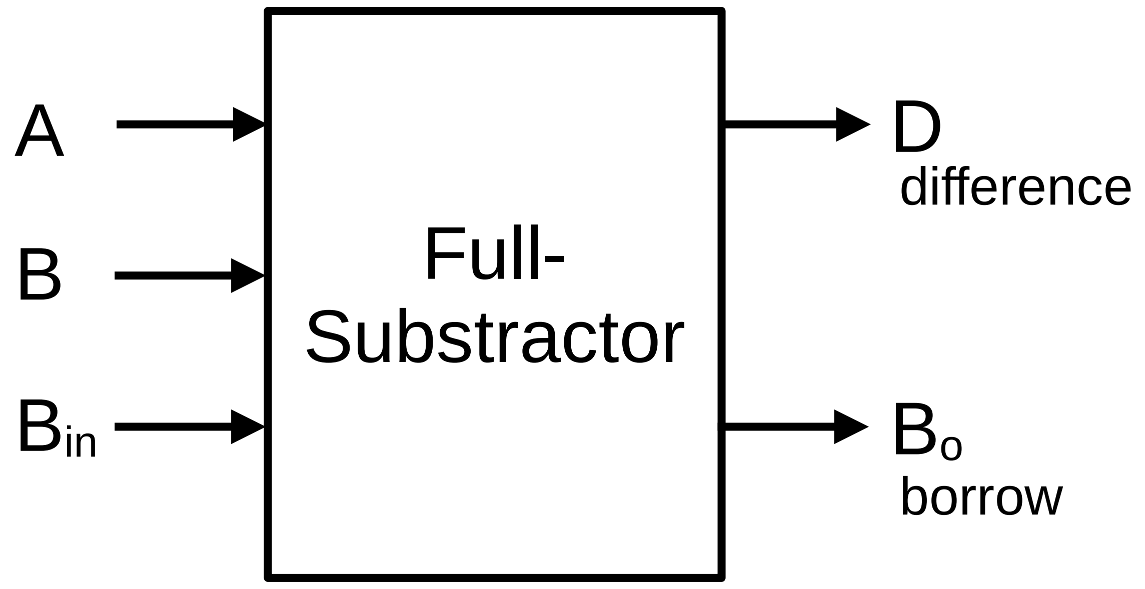


half-adder

Full-Substractor

A full subtractor performs subtraction operation on two bits, a minuend and a subtrahend, and also takes into consideration whether a ‘1‘ has already been borrowed by the previous adjacent lower minuend bit or not. As a result, there are three bits to be handled at the input of a full subtractor, namely the two bits to be subtracted and a borrow bit designated as B_{in}. There are two outputs, namely the DIFFERENCE output D and the BORROW output B_o.

The BORROW output bit tells whether the minuend bit needs to borrow a ‘1‘ from the next possible higher minuend bit.



The Boolean expressions for the two output variables are given by the equations:

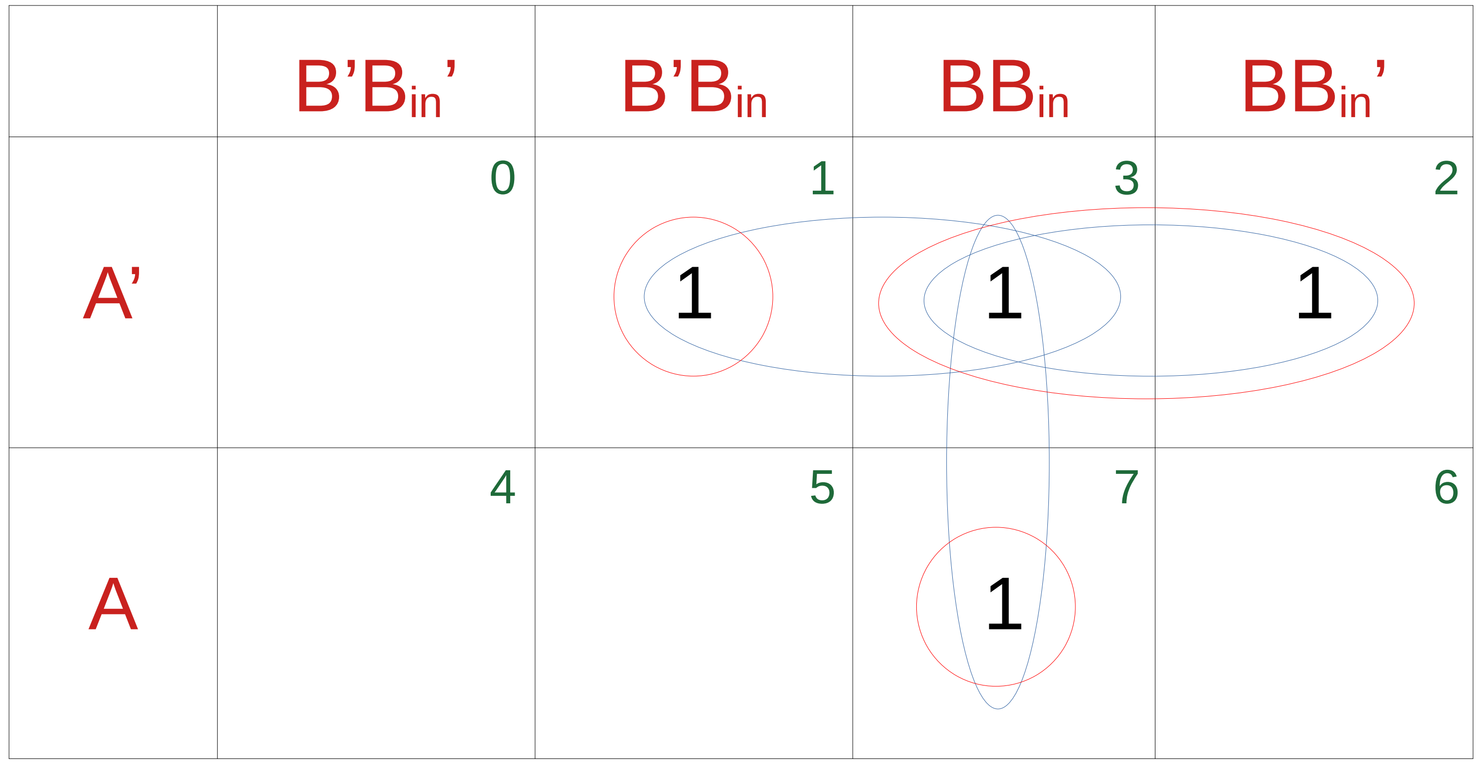
$$D = A'B'B_{in} + A'BB_{in}' + AB'B_{in}' + ABB_{in} = (A \oplus B) \oplus B_{in}$$
$$B_o = BB_{in}' + AB_{in}' + AB = B_{in}'(B + A) + AB$$

Minuend A	Subtrahend B	Borrow In B_{in}	Difference D	Borrow Out B_o
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Truth Table of Full-Substractor

	$B'B_{in}'$	$B'B_{in}$	BB_{in}	BB_{in}'
A'	^{0}	^{1} 1	^{3}	^{2} 1
A	^{4} 1	^{5}	^{7} 1	^{6}

$D = A'B'B_{in} + A'BB_{in}' + AB'B_{in}' + ABB_{in} = (A \oplus B) \oplus B_{in}$



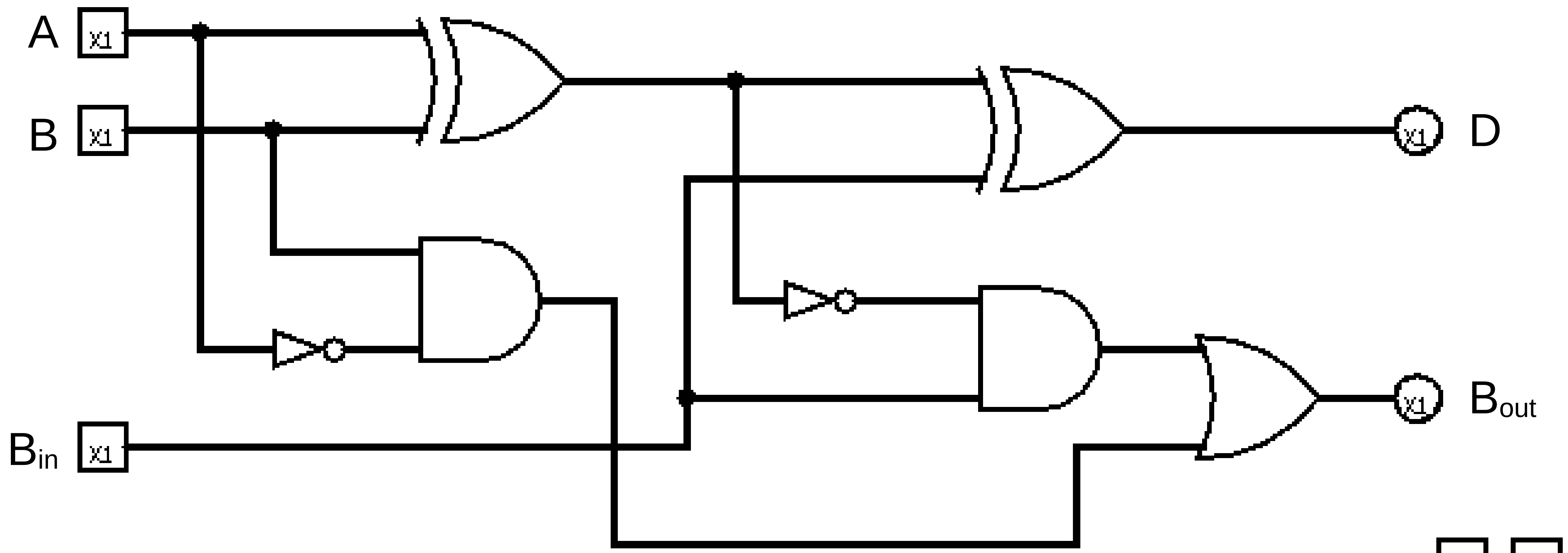
	$B'B_{in}'$	$B'B_{in}$	BB_{in}	BB_{in}'
A'	^{0}	^{1} 1	^{3} 1	^{2} 1
A	^{4}	^{5}	^{7} 1	^{6}

$\mathbf{B_o = A'B + A'B_{in} + BB_{in} = A'(B_{in} + B) + BB_{in}}$

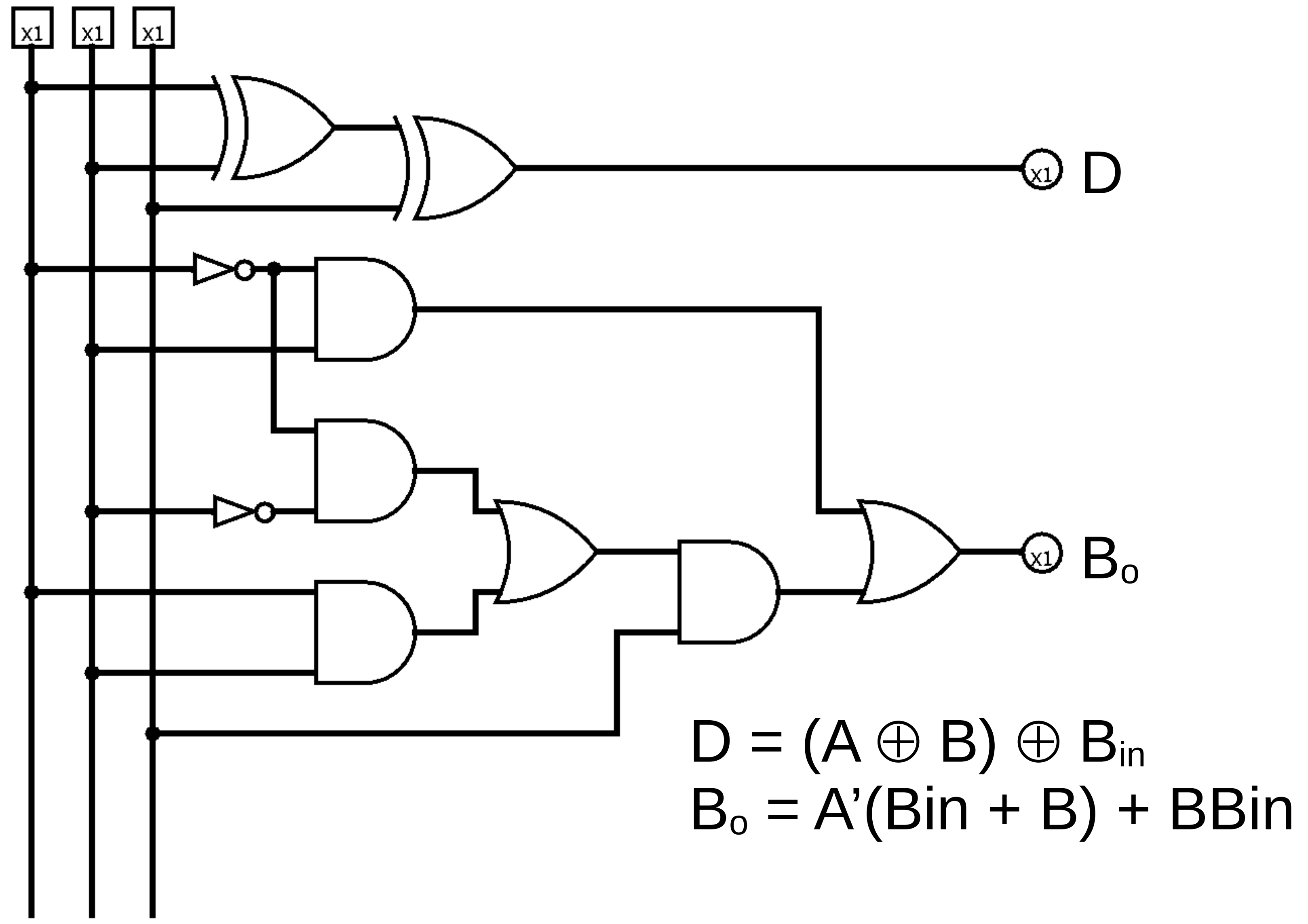
$\mathbf{B_o = A'B + A'B'B_{in} + ABB_{in} = A'B + B_{in}(A'B' + AB)}$

The Karnaugh maps for the two expressions are given above, for DIFFERENCE output D for BORROW output B_o. As is clear from the two Karnaugh maps, no simplification is possible for the difference output D. The simplified expression for B_o is given by the equation

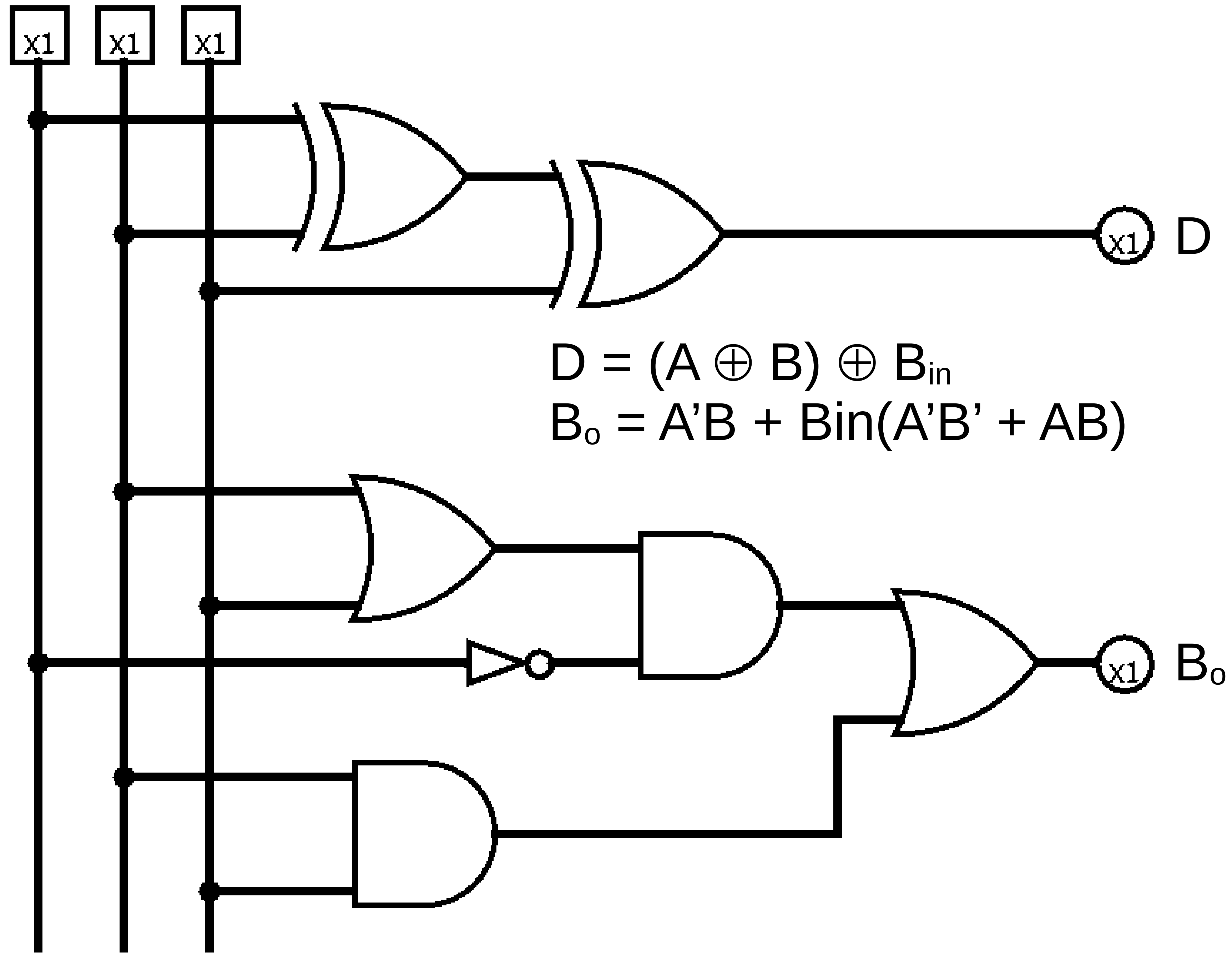
If we compare these expressions with those derived earlier in the case of a full adder, we find that the expression for DIFFERENCE output D is the same as that for the SUM output. Also, the expression for BORROW output Bo is similar to the expression for CARRY-OUT C_o. In the case of a half-subtractor, the A input is complemented. By a similar analysis it can be shown that a full subtractor can be implemented with half-subtractors in the same way as a full adder was constructed using half-adders. Again, more than one full subtractor can be connected in cascade to perform subtraction on two larger binary numbers, as in the case of four-bit substractor.



Full Substractor logic diagram



$D = (A \oplus B) \oplus B_{in}$
$B_o = A'(Bin + B) + BBin$



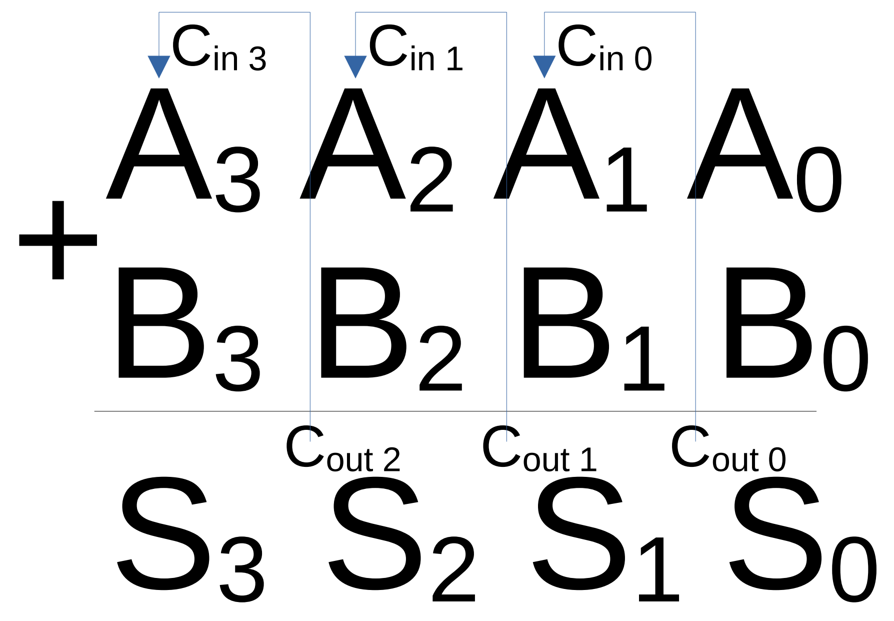
$D = (A \oplus B) \oplus B_{in}$
$B_o = A'B + Bin(A'B' + AB)$

4-bit Parallel Adder

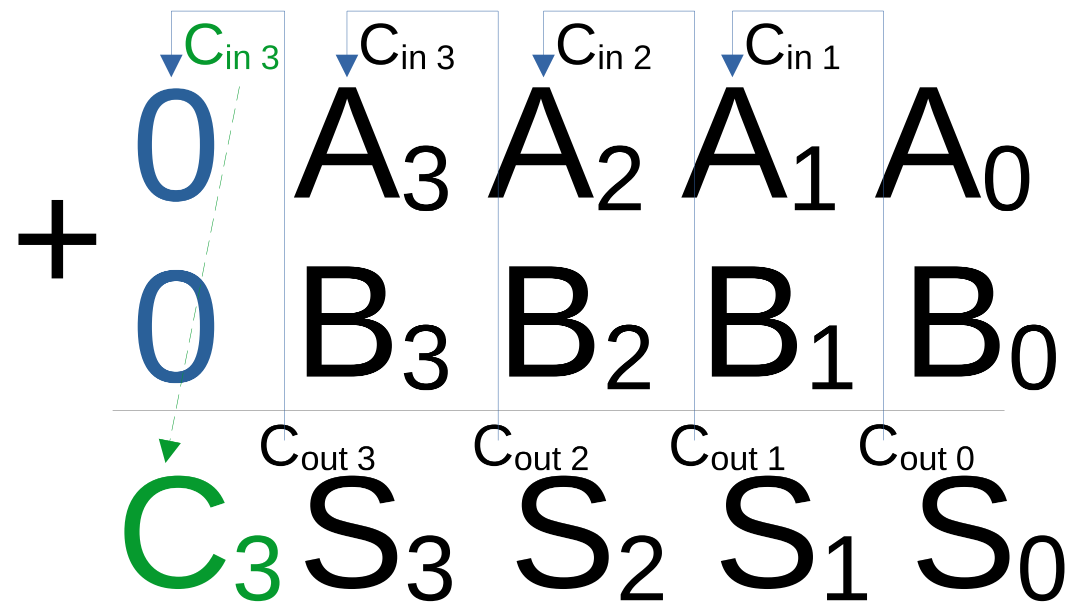
A 4-bit parallel adder is used to add two 4-bit binary numbers in parallel (all their bits at the same time). This is achieved by using 4 full-adders simultaneously.
Let's assume that we have two 4-bit binary numbers A and B, when their bits are A0, A1, A2, and A3 for A and B0, B1, B2, and B3 for B:

A = A3 A2 A1 A0
B = B3 B2 B1 B0

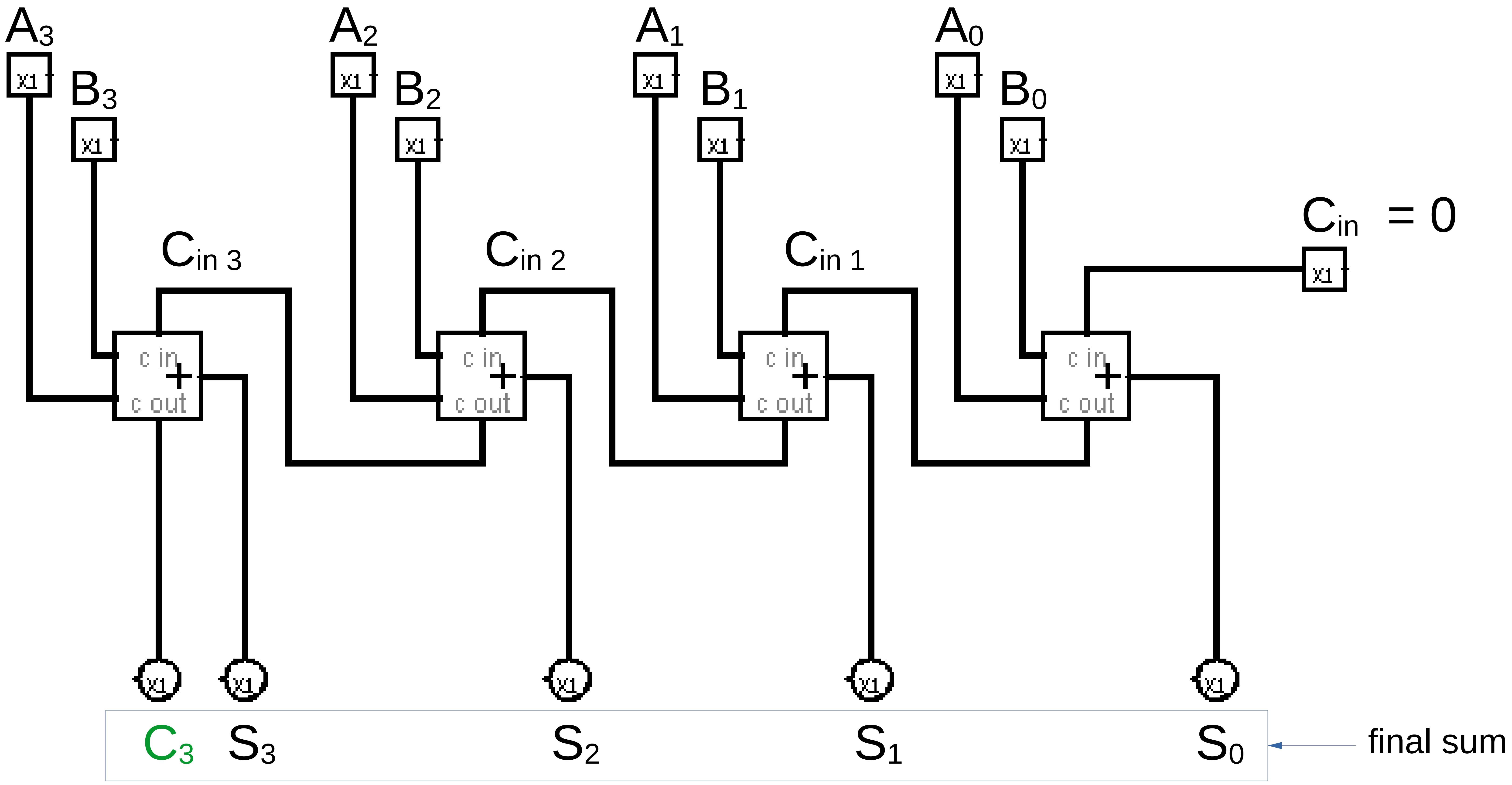
When adding those two numbers, the first step is adding their least significant bits A0 and B0, and the carry Cin will be 0 because at this step we don't have any previous operation that would might yield carry. Current addition operation might yield carry.

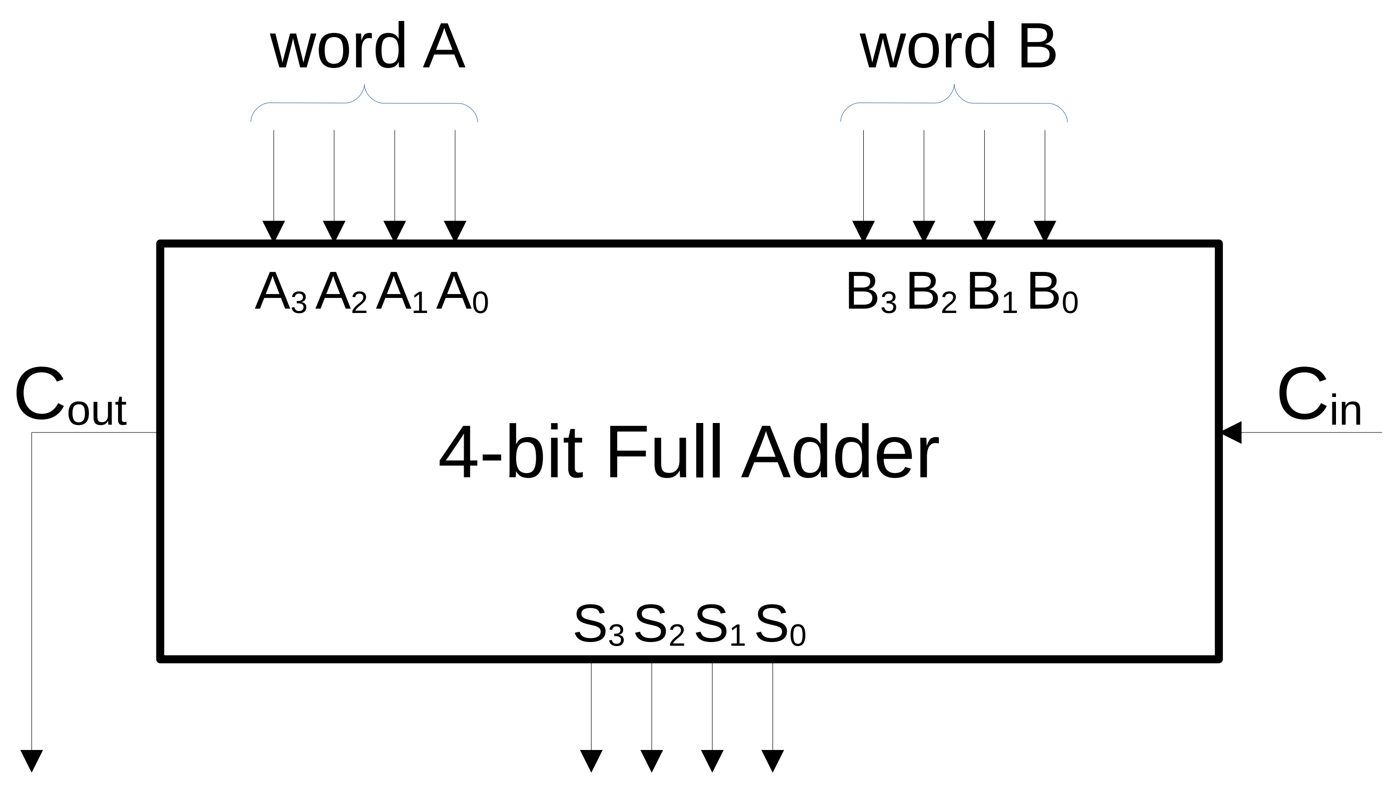


The carry C_o from the previous addition will be the input C_{in} carry to the next bits pair addition. This pattern repeats itself for the other 3 bit pairs of the two numbers, each time using a different full adder. Full adders connected in a chain are used, when the output carry C_{out} of each previous addition is the input C_{in} carry of the next addition.



At the last step of the addition, when the last full-adder adds the last bit pair that are the MSB's of the two numbers, we're using the spare order of magnitude (zeros to the left of the numbers MSB's) to the final carry $C_{out\ 3}$, that becomes the overflow bit of the addition in the case if exists.
If the final carry is 0, we have 4-bit sum. Often it is 1, so we have 5-bit sum.

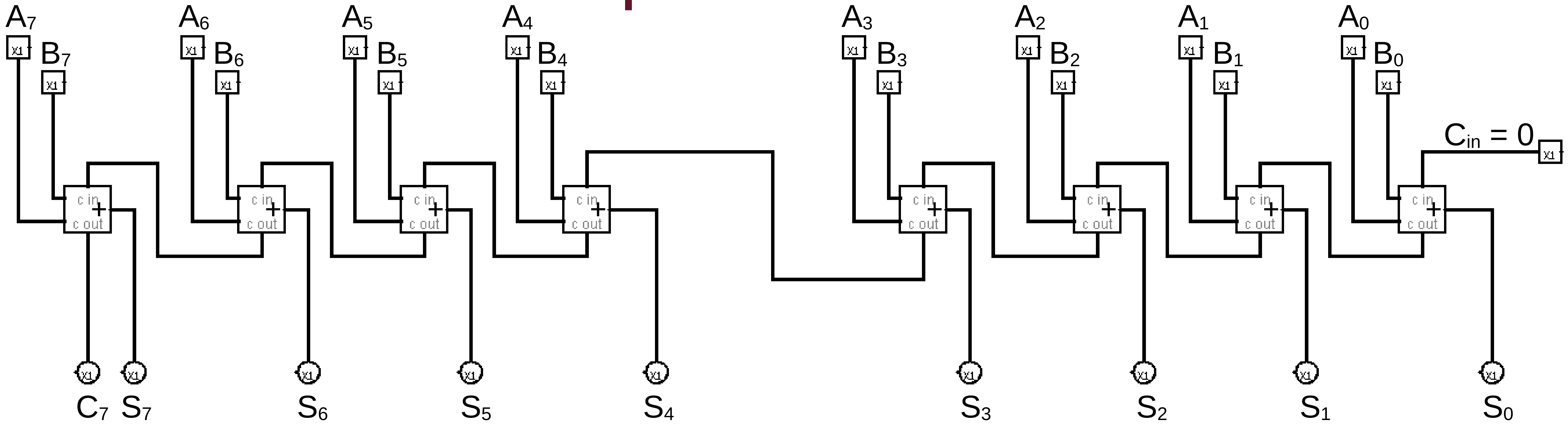

A_3
B_3
A_2
B_2
A_1
B_1
A_0
B_0
C_{in} = 0
$C_{in\ 3}$
$C_{in\ 2}$
$C_{in\ 1}$
c in
c out
x1
C_3
S_3
S_2
S_1
S_0
final sum



Usual representation of 4-bit adder

Also, using two 4-bit parallel adders connected together, we can also create an 8-bit parallel adder, that is used to add 8-bit binary numbers.

$$\begin{array}{l} A = A_7 \ldots\ldots\ldots\ldots\ldots\ldots\ldots A_0 \\ B = B_7 \ldots\ldots\ldots\ldots\ldots\ldots\ldots B_0 \\ \hline S = {}_{C_7}S_7 \ldots\ldots\ldots\ldots\ldots\ldots S_0 \end{array}$$


A_7 B_7 A_6 B_6 A_5 B_5 A_4 B_4 A_3 B_3 A_2 B_2 A_1 B_1 A_0 B_0
$C_{in} = 0$
c in
c out
C_7 S_7 S_6 S_5 S_4 S_3 S_2 S_1 S_0

4-bit binary Adder-Subtractor

In Digital Circuits, A Binary Adder-Subtractor is one which is capable of both addition and subtraction of binary numbers in one circuit itself. The operation being performed depends upon the binary value the control signal holds. It is one of the components of the ALU (Arithmetic Logic Unit).

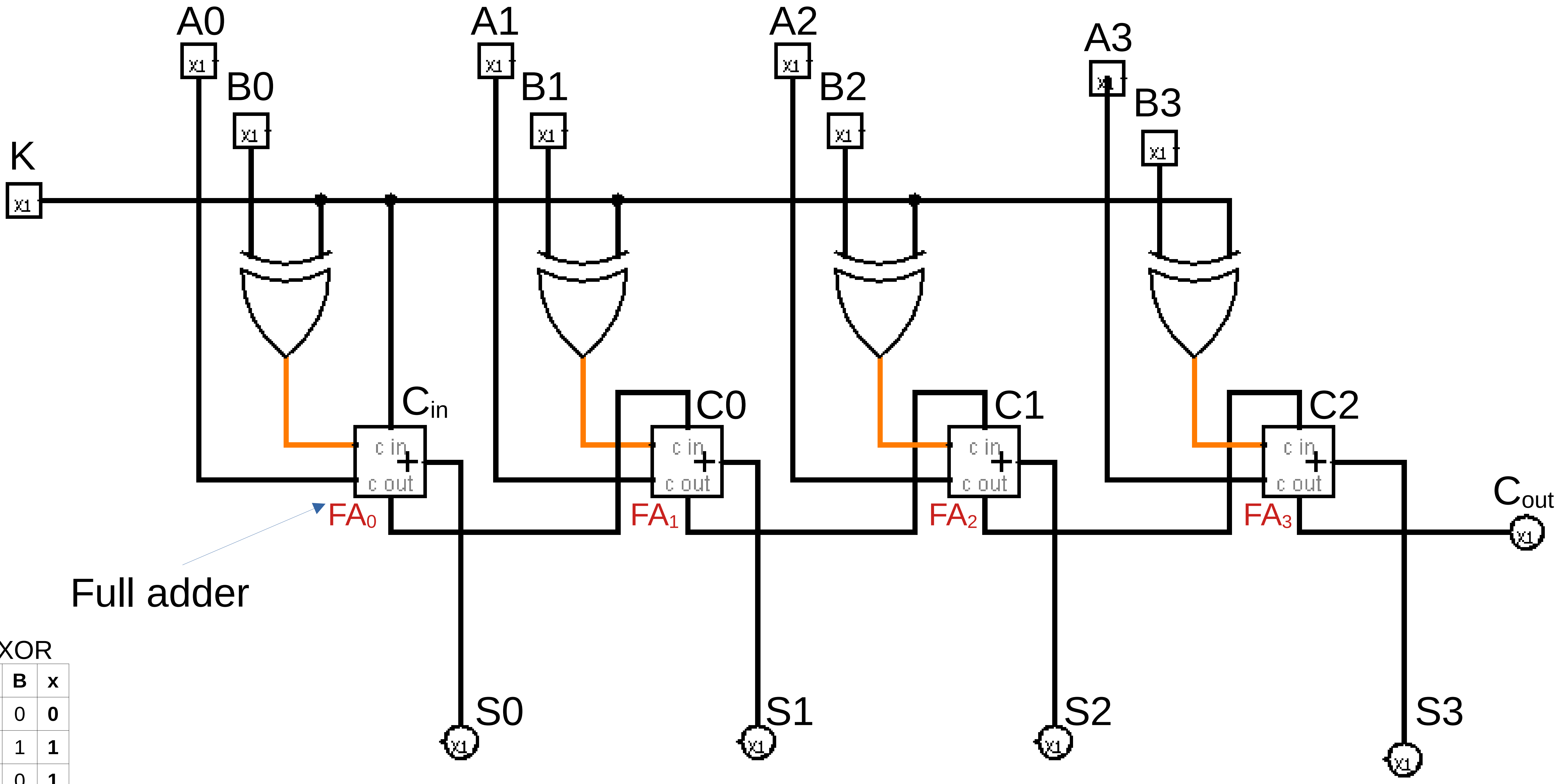
Lets consider two 4-bit binary numbers A and B as inputs to the Digital Circuit for the operation with digits

$A_3A_2A_1A_0$ for A
$B_3B_2B_1B_0$ for B

The circuit consists of 4 full adders since we are performing operation on 4-bit numbers. There is a control line K that holds a binary value of either 0 or 1 which determines that the operation being carried out is addition or subtraction.

As shown in the figure, the first full adder has control line directly as its input (input carry C_{in}), The input A0 (The least significant bit of A) is directly input in the full adder. The third input is the XOR of B0 and K. The two outputs produced are Sum/Difference (S0) and Carry (C0).

If the value of K (Control line) is 1, the output of $B0 \oplus K = B0'$ (Complement B0). The carry input of the first full adder FA0, which deals with the least significant bits of the two numbers, is connected to the control line K. This simulates taking the carry from the previous edition, that's added automatically to the LSB B0 of the numbers B. The complementation of the number B by XORing each its bit with 1 of the control line, together with giving 1 to the carry input of the first adder FA0 (to be added to the B's LSB), leads to 2's complementation of the number B.



XOR

A	B	x
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

This suggests, that when K=1, the operation performed between the two numbers is 2's complementation subtraction: A + (-B).
Similarly, when K=0, all the bits of the number B (B0...B3) are XORed with 0 and go to the corresponding adder's input unchanged. The carry input of the first adder FA0 also takes 0 from the line K. So, 0 on the line K leaves the number B unchanged, and the operation that is performed between the two numbers is an addition.

Example:
Lets take two 3 bit numbers A=1010 and B=1011 and input them in the full adder with both values of control lines.

For K=0:

$A0=0$, $B0\oplus K=B0=1$, $C_{in}=0 \Rightarrow S0=1$, $C0=0$
$A1=1$, $B1\oplus K=B1=1$, $C0=0 \Rightarrow S1=1$, $C1=1$
$A2=0$, $B2\oplus K=B2=0$, $C1=1 \Rightarrow S2=1$, $C2=0$
$A3=1$, $B3\oplus K=B3=1$, $C2=0 \Rightarrow S3=0$, $C_{out}=1$

The final sum: $C_{out}S_3S_2S_1S_0$
That is: 10111

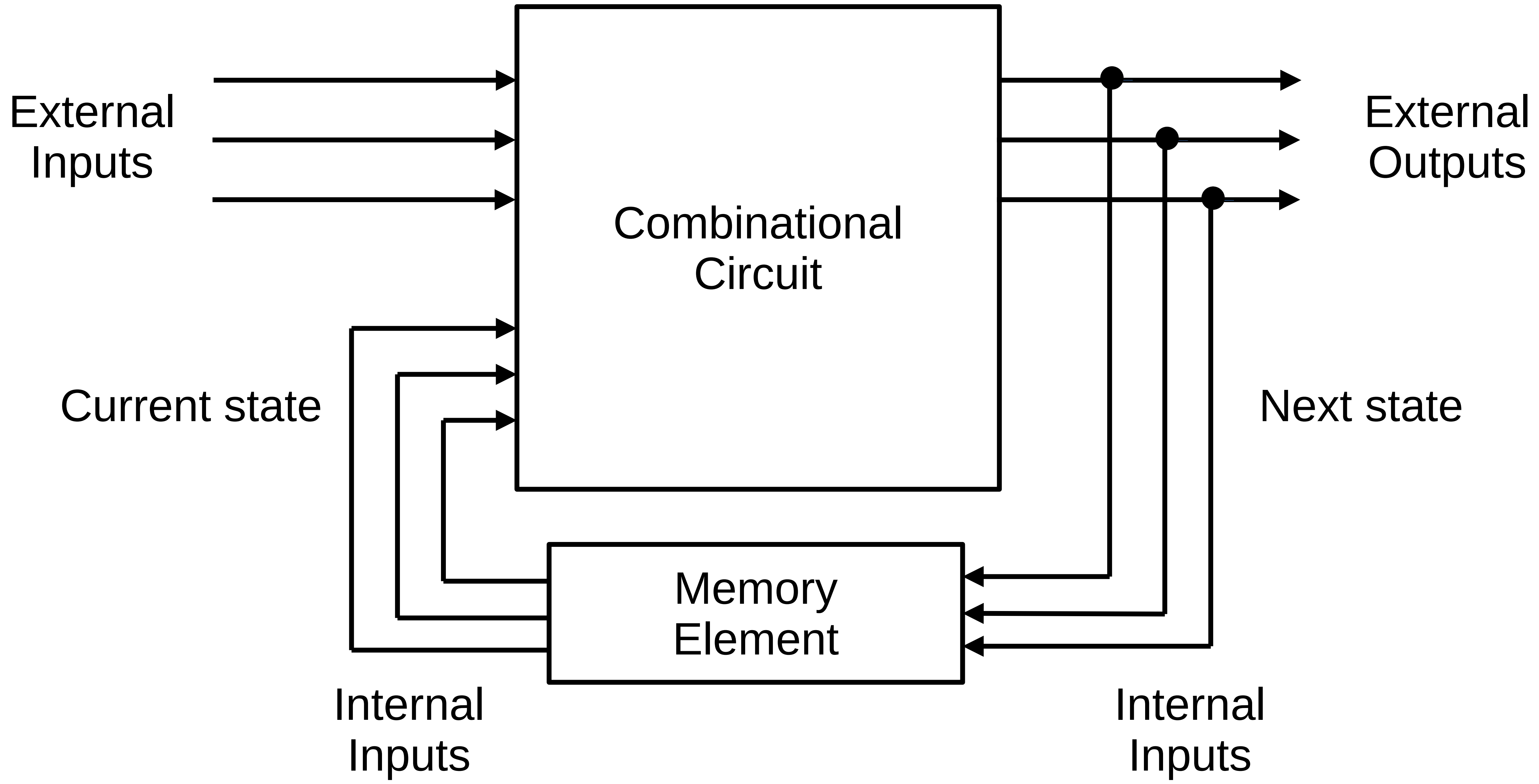
For K=1:

$A0=0$, $B0\oplus K=B0'=1$, $C_{in}=1 \Rightarrow S0=1$, $C0=0$
$A1=1$, $B1\oplus K=B1'=0$, $C0=0 \Rightarrow S1=1$, $C1=0$
$A2=0$, $B2\oplus K=B2'=1$, $C1=0 \Rightarrow S2=1$, $C2=0$
$A3=1$, $B3\oplus K=B3'=0$, $C2=0 \Rightarrow S3=1$, $C_{out}=0$

The final difference: $C_{out}S_3S_2S_1S_0$
That is: 01111

Intoduction to Sequential Circuits

A Sequential circuit is a combinational logic circuit that consists of inputs variable (X), logic gates (Computational circuit), and output variable (Z). A combinational circuit produces an output based on input variable only, but a **Sequential circuit** produces an output based on **current input and previous input variables**. That means sequential circuits include memory elements that are capable of storing binary information. That binary information defines the state of the sequential circuit at that time. A *latch* capable of storing one bit of information.



Schematic representation of sequential circuit

As shown in the figure there are two types of input to the combinational logic :

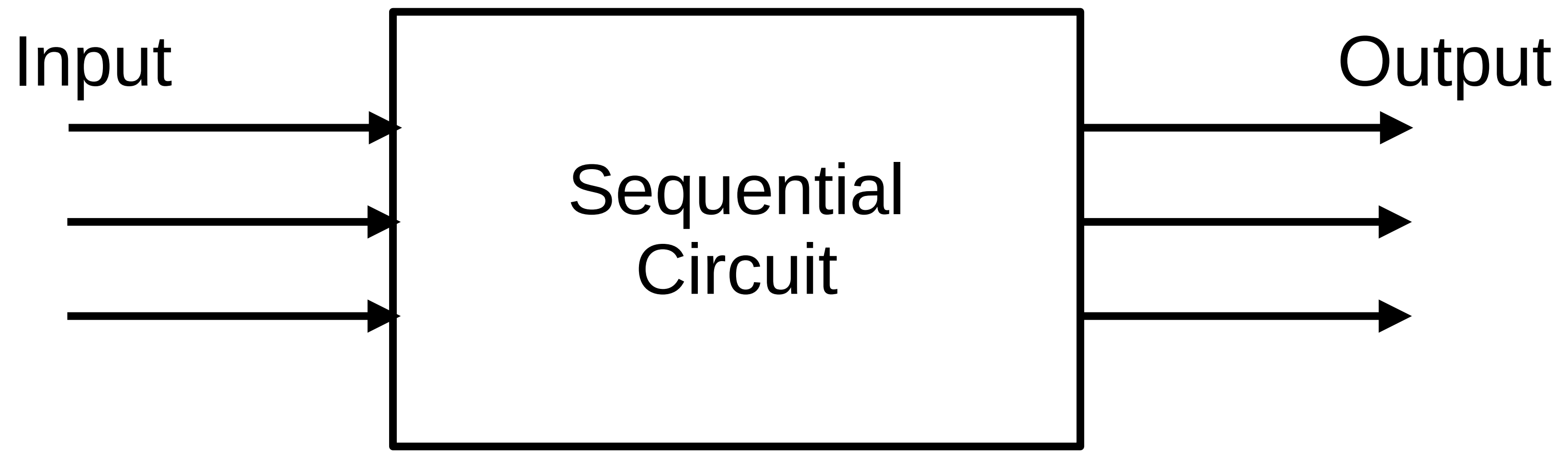
1. External inputs which not controlled by the circuit.
2. Internal inputs which are a function of a previous output states.

Secondary inputs are state variables produced by the storage elements, whereas secondary outputs are excitations for the storage elements.

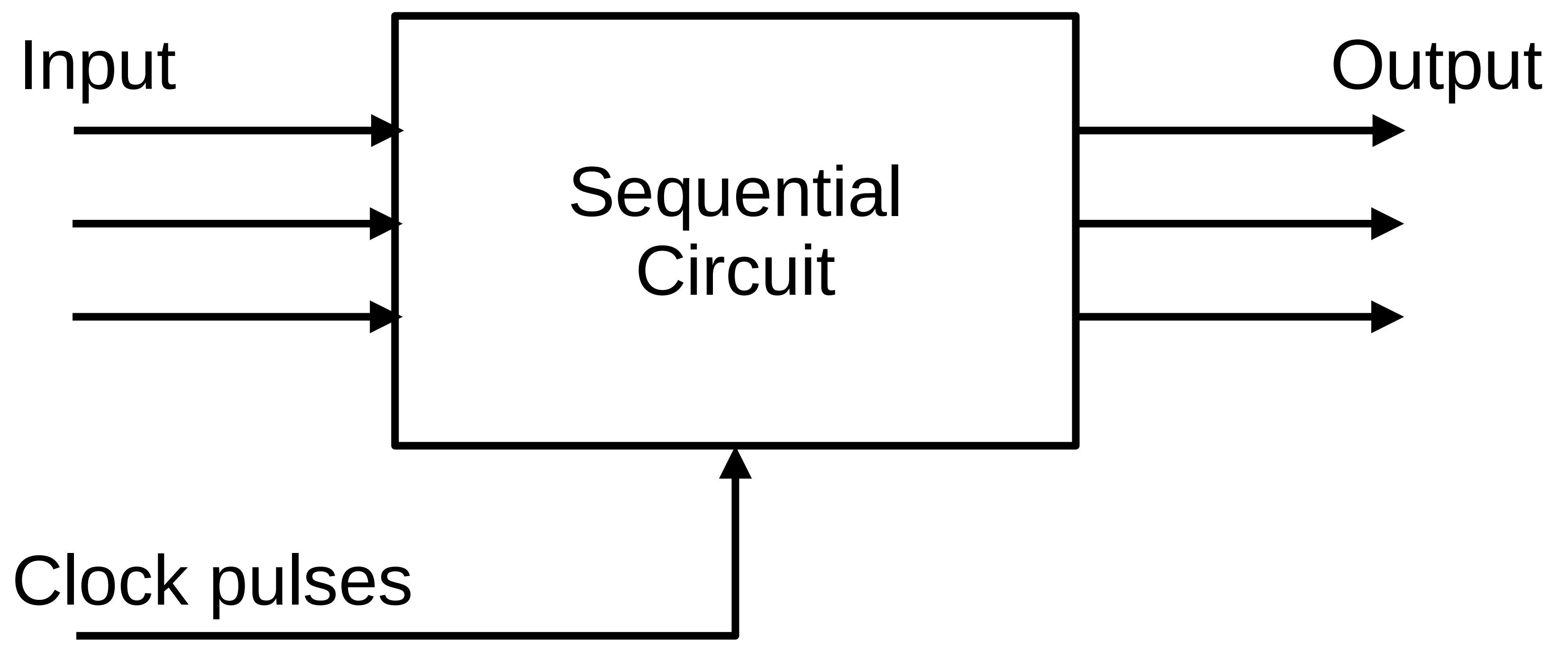
Types of Sequential Circuits

There are two types of sequential circuits:

Asynchronous sequential circuit – These circuits **do not use a clock signal** but uses the pulses of the inputs. These circuits are **faster** than synchronous sequential circuits because there is clock pulse and change their state immediately when there is a change in the input signal. We use asynchronous sequential circuits when speed of operation is important and independent of internal clock pulse. But these circuits are more difficult to design and their output is uncertain.



- **Synchronous sequential circuit** – These circuits **uses clock signal** and level inputs (or pulsed) (with restrictions on pulse width and circuit propagation). The output pulse is the same duration as the clock pulse for the clocked sequential circuits. Since they wait for the next clock pulse to arrive to perform the next operation, so these circuits are bit **slower** compared to asynchronous. Level output changes state at the start of an input pulse and remains in that until the next input or clock pulse.



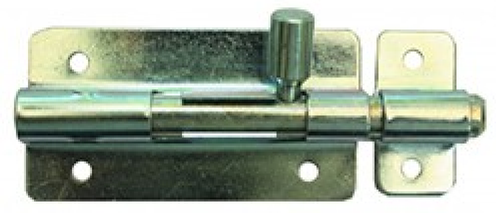
We use synchronous sequential circuit in synchronous counters, flip flops, and in the design of MOORE-MEALY state management machines.

We use sequential circuits to design Counters, Registers, RAM, MOORE/MEALY Machine and other state retaining machines.

Sequential circuits are implement the concept of pipelining (or pipeline), where the output of one element (or more commonly stage) becomes the input to the next stage sequentially. On most digital chips, the output of a stage becomes the input to the next stage generally at the same time (synchronously) when a globally available synchronizing signal (a.k.a a global clock) changes its voltage level from high-to-low or low-to-high, vice versa.

For example, a complex modern digital chip (like Intel/AMD CPU and AMD/Nvidia GPU) breaks up its complex function into 20 to 30 pipeline stages, making each pipeline stage simpler and easier to implement. Each pipeline stage has flip-flops. Having flip-flop increase the chip area by 20 to 30%. However, it maintains much greater throughput, more computation done in a given time (higher performance). Almost all modern digital CPU and GPU are pipelined using flip-flops.

Flip-Flops & Latches



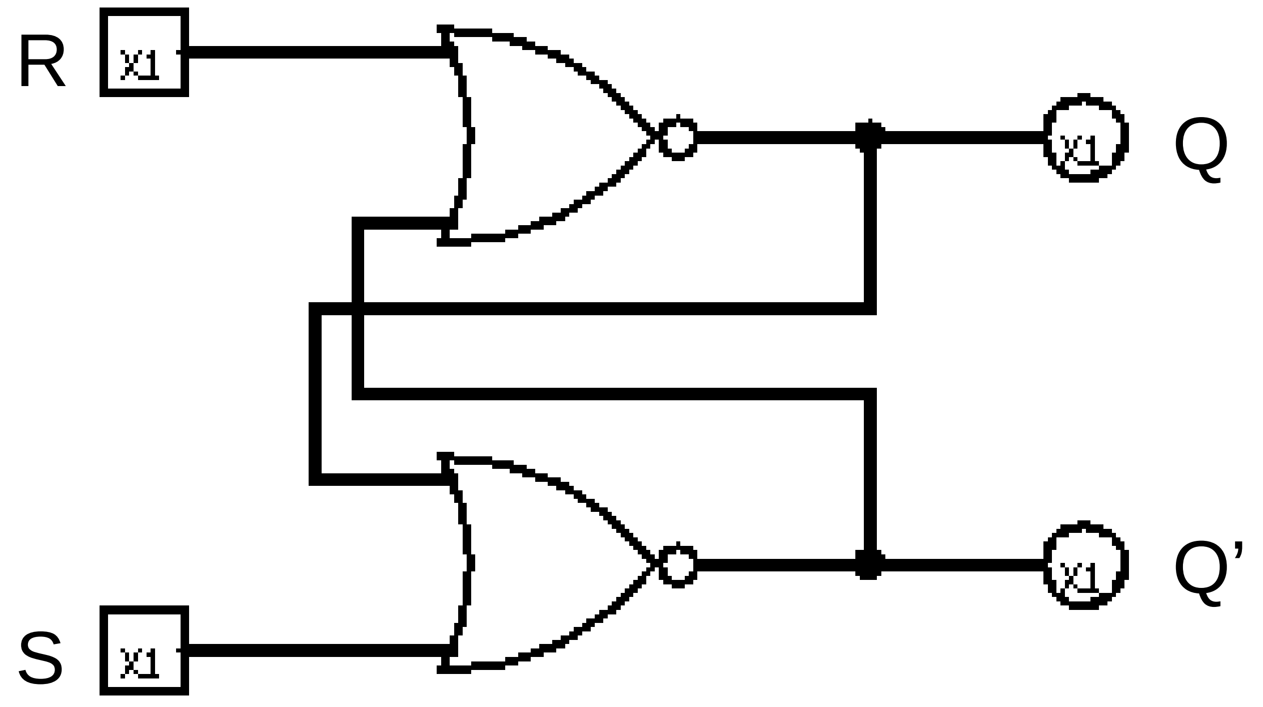
The digital circuits considered thus far have been combinational, where the outputs at any given time are entirely dependent on the inputs that are present at that time. Although every digital system is likely to have a combinational circuit, most systems encountered in practice also include storage elements, which require that the system be described in terms of sequential circuits. The most common type of sequential circuit is the synchronous type. Synchronous sequential circuits employ signals that affect the storage elements only at discrete instants of time. Synchronization is achieved by a timing device called a clock pulse generator that produces a periodic train of clock pulses. The clock pulses are distributed throughout the system in such a way that storage elements are affected only with the arrival of the synchronization pulse. Clocked synchronous sequential circuits are the type most frequently encountered in practice. They seldom manifest instability problems and their timing is easily broken down into independent discrete steps, each of which may be considered separately.

The storage elements employed in clocked sequential circuits are called flip-flops. A flip-flop is a binary cell with two stable states capable of storing one bit of information. It has two outputs, one for the normal value and one for the complement value of the bit stored in it. A flip-flop maintains a binary state until directed by a clock pulse to switch states. The difference among various types of flip-flops is in the number of inputs they possess and in the manner in which the inputs affect the binary state. Flip-flops and latches are fundamental building blocks of digital electronics systems used in computers, communications, and many other types of systems. Both are used as data storage elements. It is the basic storage element in sequential logic. But first, let’s clarify the difference between a latch and a flip-flop.

Flip flop v/s Latch

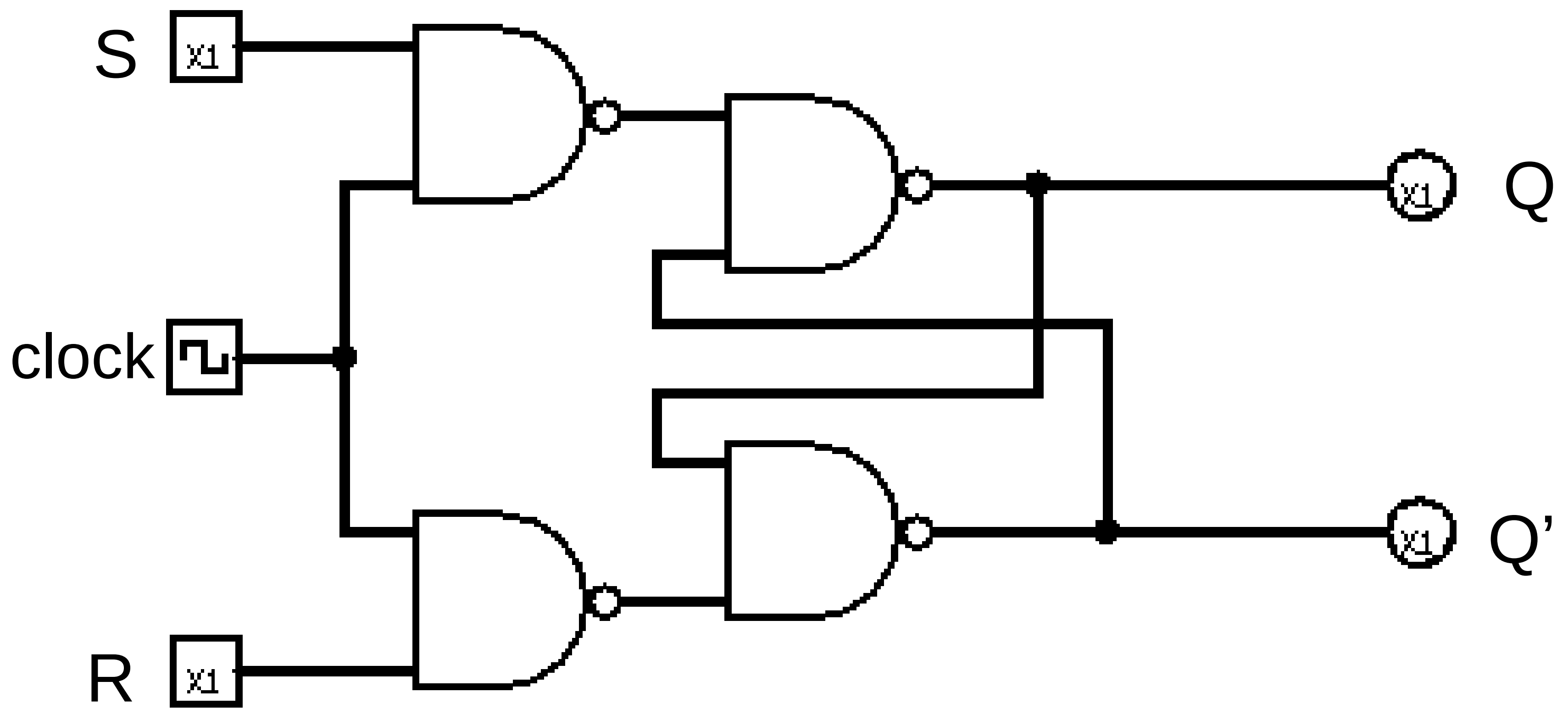
The difference between a latch and a flip-flop is that a latch is level-triggered (outputs can change as soon as the inputs changes) and Flip-Flop is edge-triggered (only changes state when a control signal goes from high to low or low to high).

For example, let us talk about SR latch and SR flip-flops.



SR Latch

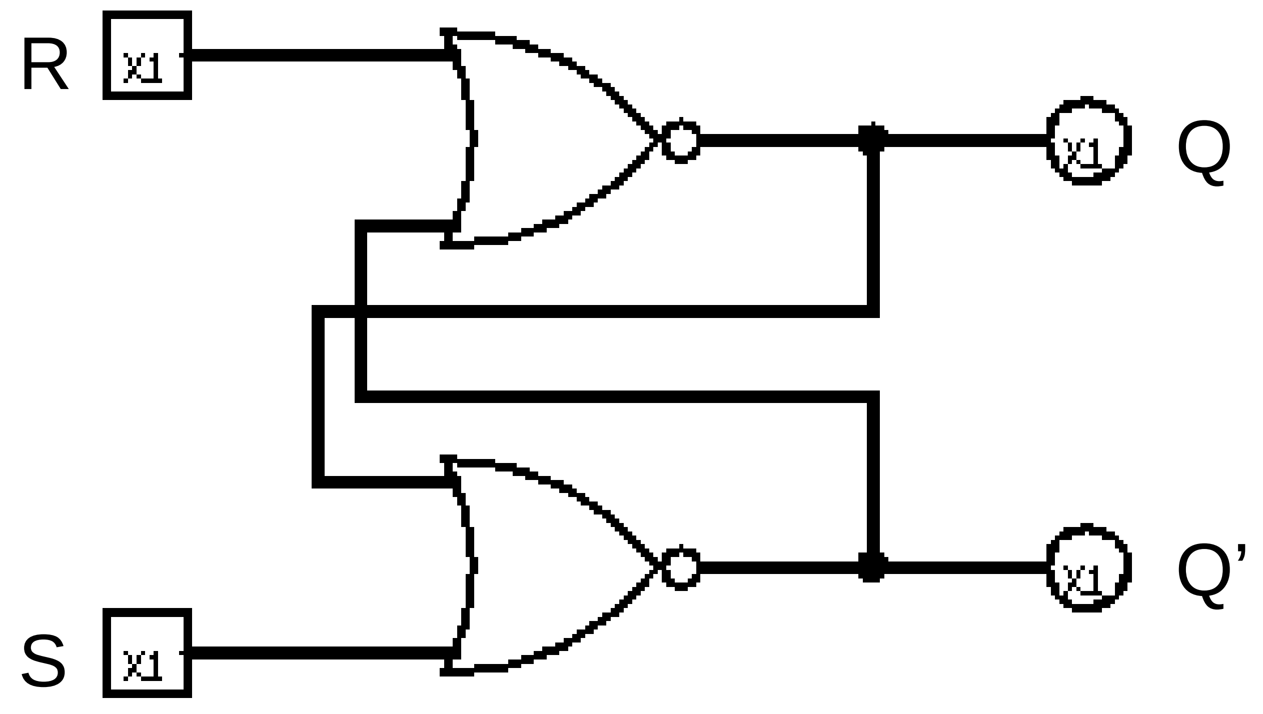
In this circuit when you Set S as active the output Q would be high and Q’ will be Low. This is irrespective of anything else. (This is an active-low circuit so active here means low, but for an active high circuit active would mean high)



SR Flip-Flop

In this circuit diagram, the output is changed (i.e. the stored data is changed) only when you give an active clock signal. Otherwise, even if the S or R is active the data will not change.

The SR (Set-Reset) Latch

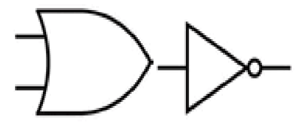


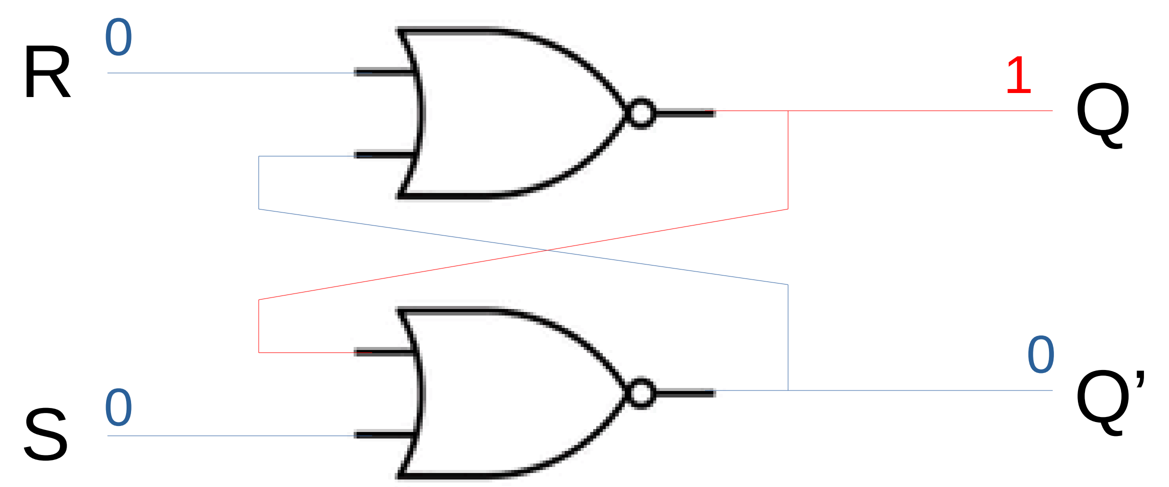
SR Latch

The Set-Reset (SR) latch can be though of as a 1 bit memory. It can be put into one of two stable output states, triggered by an input pulse. The circuit remembers this state, until it changes again by another input pulse, or until the power is going off. For this reason, the cirtcuit is known as a *bistable latch*.
There are versions of SR latch, that built of two NOR gates,and also of two NAND gates.

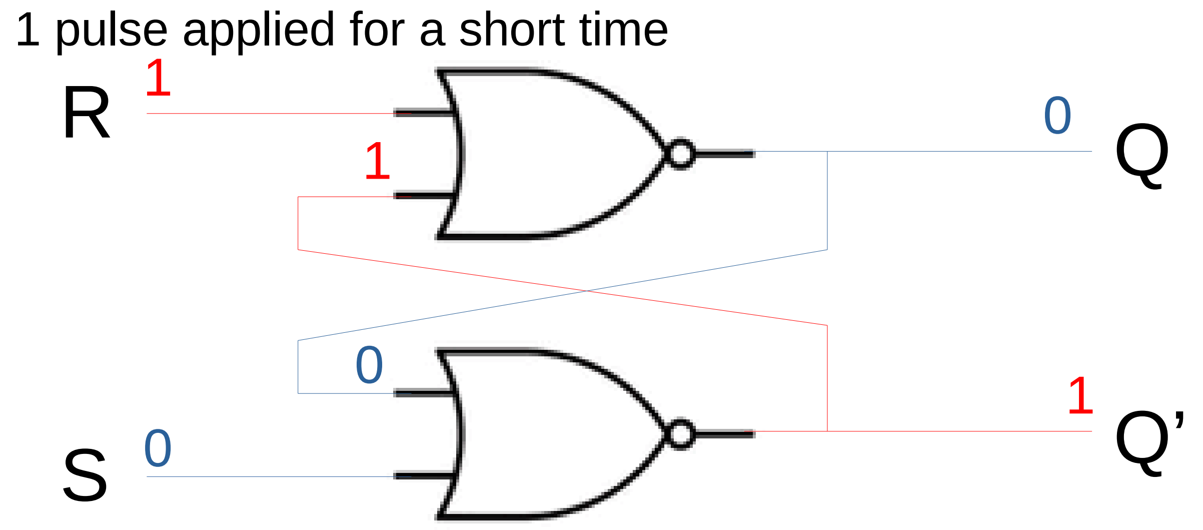
NOR

A	B	x
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**0**

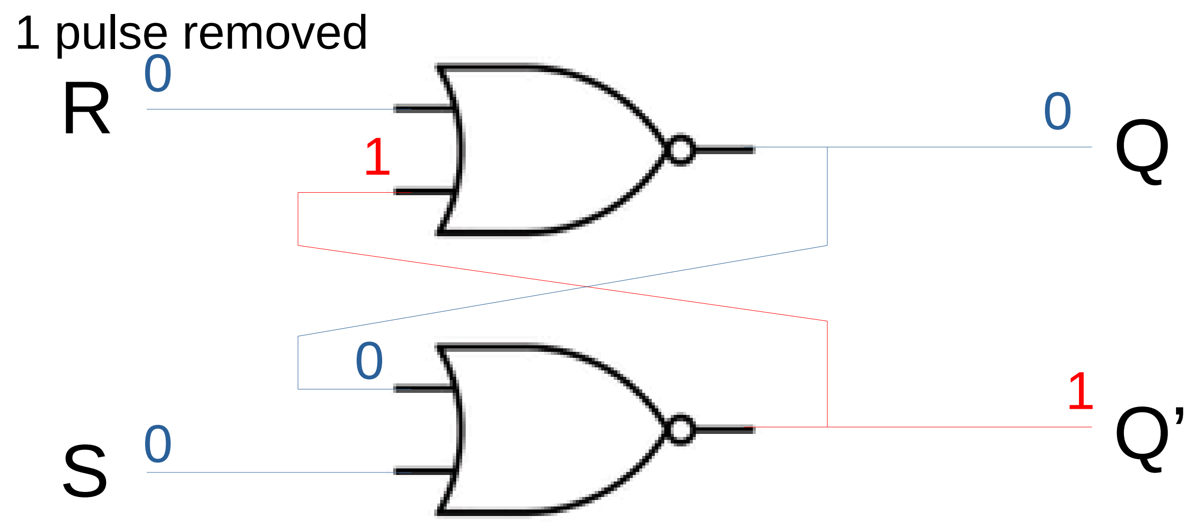




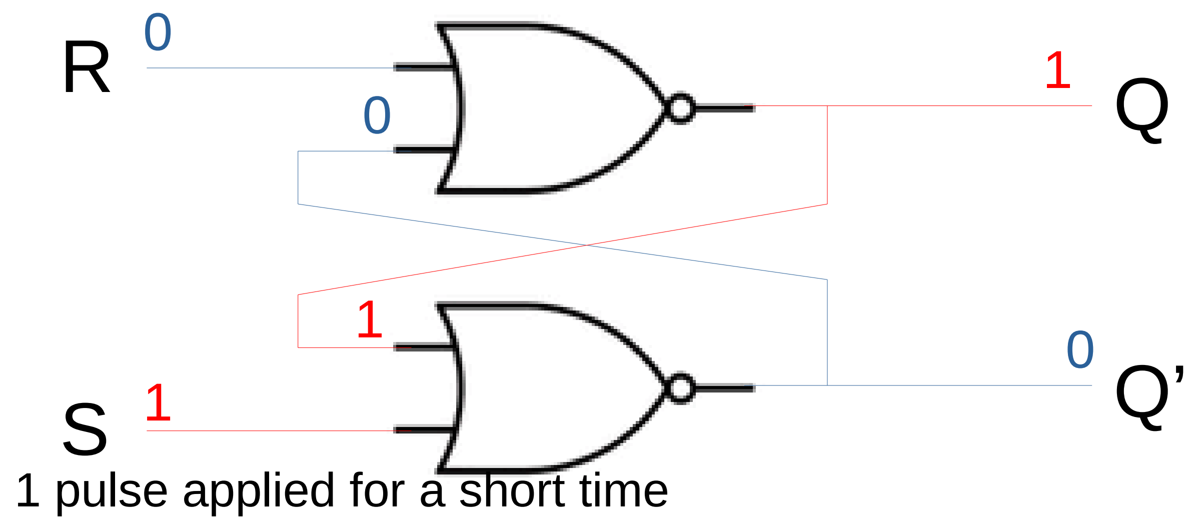
Lets consider an SR latch built from NOR gates. In this NOR version of an SR latch, two NOR gates are connected together in such a way, is the output of each NOR gate is one of the inputs of the other. This cross-coupling of two gates, results in a form of positive feedback. SR latches, like all electronic circuits, require power to work. , the power connections aren't shown on this diagram. The SR latch has two inputs, R and S, and the output Q. The SR latch also makes the inverse of the output available, on this diagram you can see NOT Q, (Q').
The starting state here that both inputs are 0, Q is 1, and NOT Q is obviously 0 (the inverse of Q). Both of the inputs of the top NOR gate are 0, so the output of the top gate is 1. This is exactly what you would expect from a NOR gate. The inputs of the lower NOR gate are1 and 0, so the output of the lower gate is 0. Because Q is 1, the latch is currently storing 1.



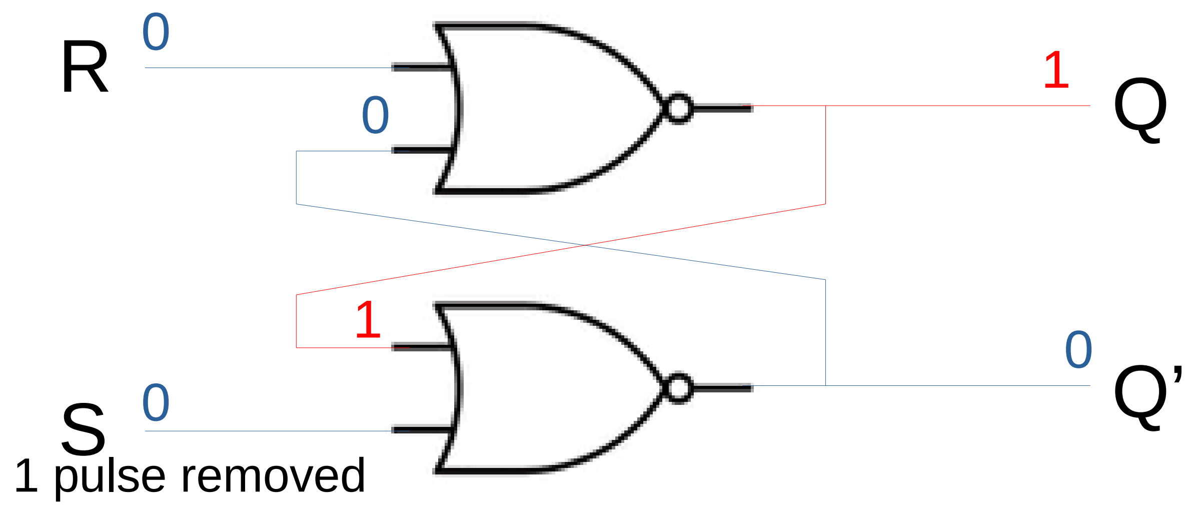
Now we apply a pulse to input R to reset the latch. This changes the output of the top gate, and this is fed back to the lower gate. The lower gate output also changes, and this is fed back into the top gate.



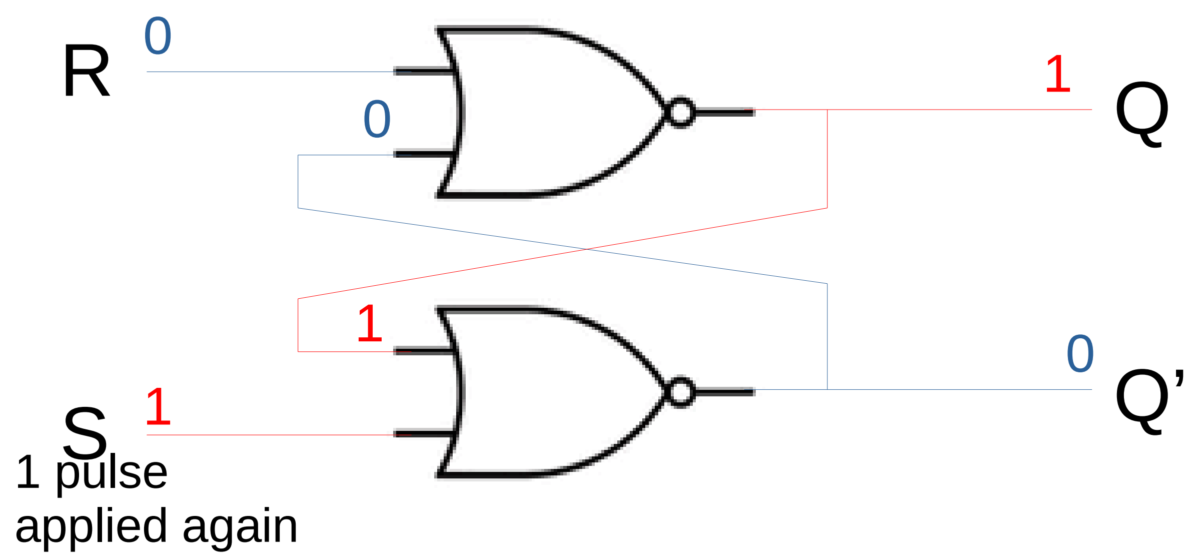
The pulse that was applied to reset the latch is then removed, and R is 0 again, but the output Q is now 0, so the latch is now storing a 0.

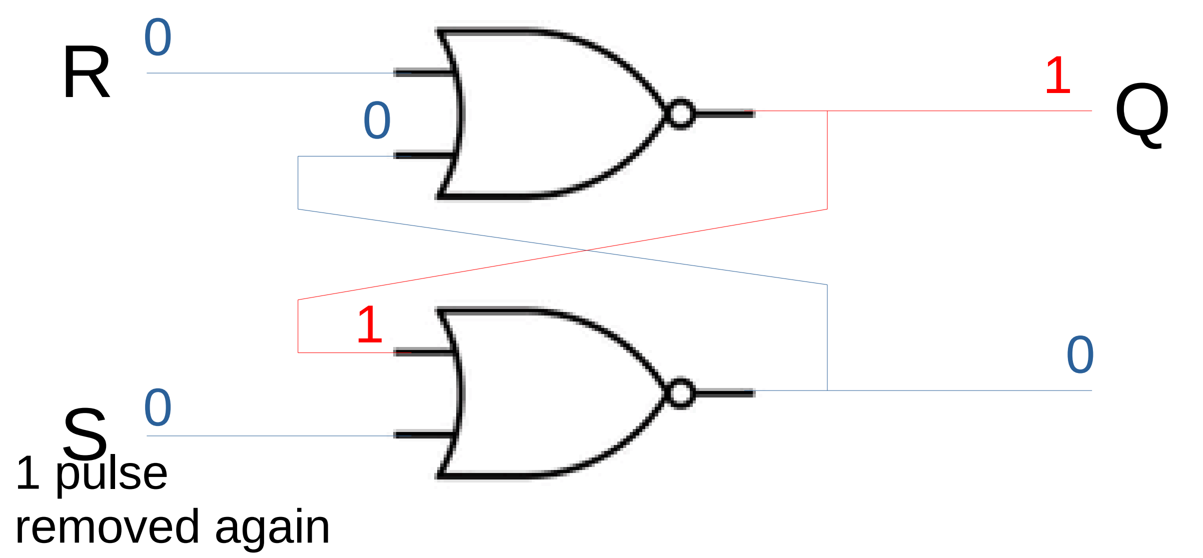


In order to store 1 again, a pulse must be applied to input S, which will set the latch. Again, notice how various changes are propagated around the circuit.



The set pulse is then removed, and the circuit is now latched into a set state, it’s storing a 1 again.





Notice, that if another set pulse applied, it has no effect. Applying a set pulse at S will always force a latch into a set state, regardless of the previous state of the latch. Similarly, applying a reset pulse, will always force the latch into a reset state.

It should be noted, that S and R are never left high, that is, neither is ever set continuously to 1. The latch is controlled by pulses only.

S	R	Q	Q'
0	0	1 0	0 1
0	1	0	1
1	0	1	0
1	1	0	0

This gives us is unusual looking truth table. When both S and R are set to 0, Q may be 1 or may be 0, depending of the previous state of the circuit.

S	R	Q	Q’
0	0	1 0	0 1
0	1	0	1
1	0	1	0
1	1	0	0

Lets examine the truth table, as this SR latch is reset again. The reset pulse is applied. S is 0, R is 1, the output Q is 0, and its inverse, Q’, is 1. The SR latch is storing a 0.

S	R	Q	Q’
0	0	1 0	0 1
0	1	0	1
1	0	1	0
1	1	0	0

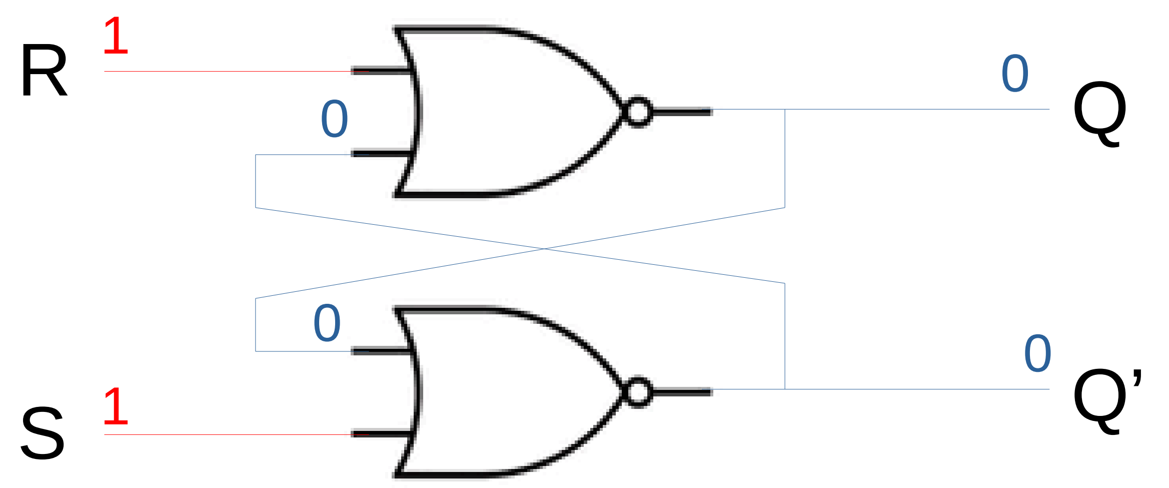
The reset pulse is removed. Both S and R are 0 again. The output Q remains its 0, and its inverse remains its 1. The SR latch is still storing a 0.

S	R	Q	Q’
0	0	1 0	0 1
0	1	0	1
1	0	1	0
1	1	0	0

The set pulse is applied. S is 1 and R is 0. The output Q becomes 1, and its inverse becomes 0.

S	R	Q	Q’
0	0	1 0	0 1
0	1	0	1
1	0	1	0
1	1	0	0

The set pulse is removed. S is 0, and R is 0. The output Q is still 1, and NOT Q is, of course, 0.



Now, the only circumstance we haven't considered, is when both inputs, S and R, are set to 1 at the same time. If this would occur, we would be telling the SR latch to set the value of Q to both 1 and 0 simultaneously. In reality, Q would become 0, and Q', would also become 0. This would sort itself out if one of the inputs would fall to zero before the other.

S	R	Q	Q'
0	0	1 0	0 1
0	1	0	1
1	0	1	0
1	1	0	0

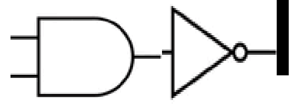
INVALID

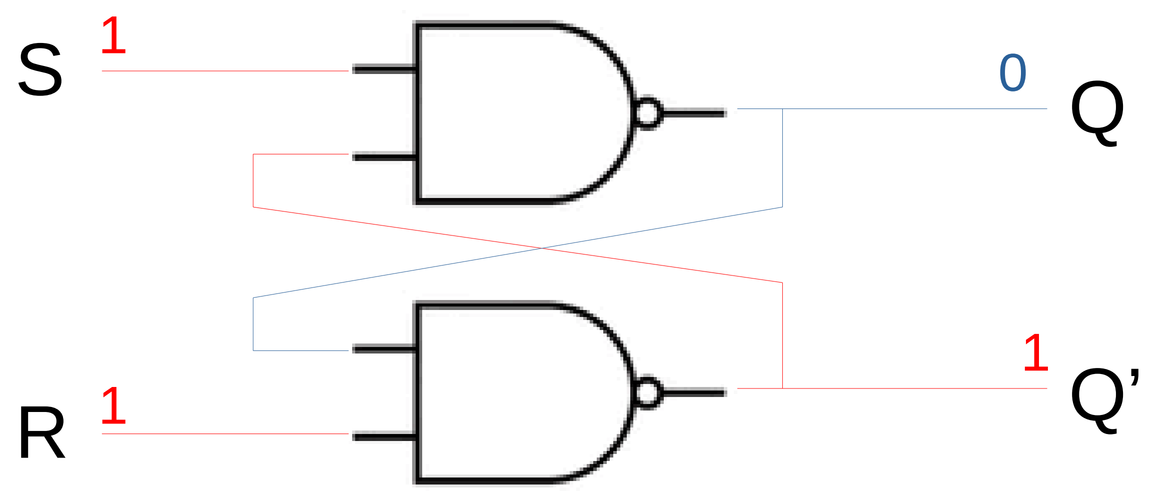
For example, if R fell to 0 first, with S still at 1, then Q will become 1 again. If, however, both inputs was at 1, and both fell to 0 at the same time, we would have what known as a race condition between a two gates. They'd be racing each other to feed back their new outputs, and it is impossible to know, which one would win. Hence, if both inputs are high, the next state of the latch can't be determined. This is not a state that the latch should ever be in, it's **illegal**, it's **invalid**. Most of the time inputs S and R should both be at 0, and only **momentarily** will one or another input become 1. And at any time, while it's operating, the SR latch should be in the **set state**, or the **reset state,** with Q and Q' opposite to each other.

This type of SR latch is said to be an *active high* SR latch, because the normal condition for S and R is low, and a high pulse on one of its inputs is required to bring about a change.

NAND

A	B	x
0	0	1
0	1	1
1	0	1
1	1	0





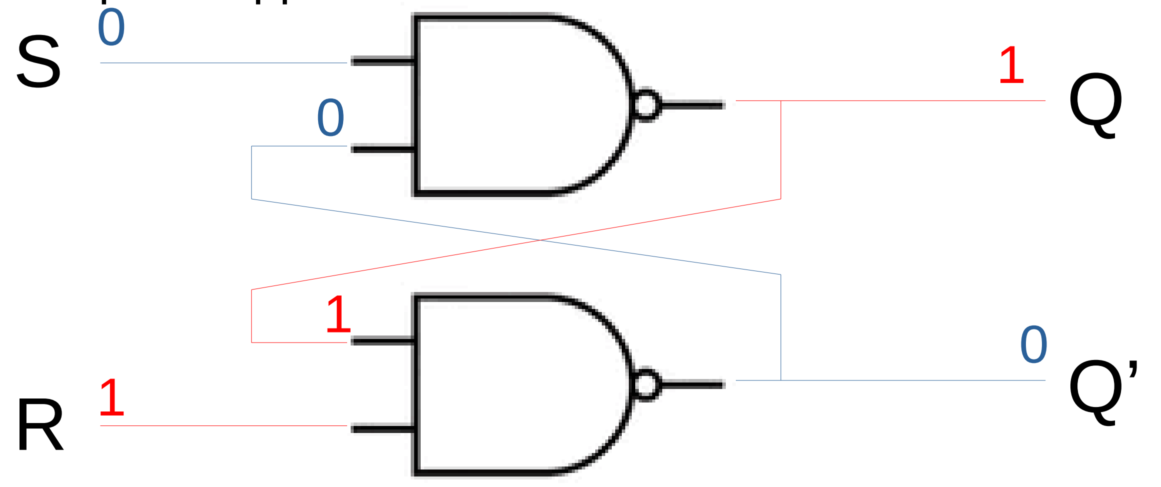
Now let’s consider an SR latch built out of NAND gates. In a NAND gate, only when both inputs are high, the output is low. The wiring of this SR latch is the same, but notice that input S is at the top now, and R is at the bottom.

S	R	Q	Q’
0	0	1	1
0	1	1	0
1	0	0	1
1	1	0 1	1 0

This is a truth table for this variation of an SR latch. It’s a little different from the truth table we just seen, because this latch behaves differently. The difference is, that R and S are kept high most of the time. Here we can see, that the output Q is 0, so the latch is storing a 0.

S	R	Q	Q’
0	0	1	1
0	1	1	0
1	0	0	1
1	1	0 1	1 0

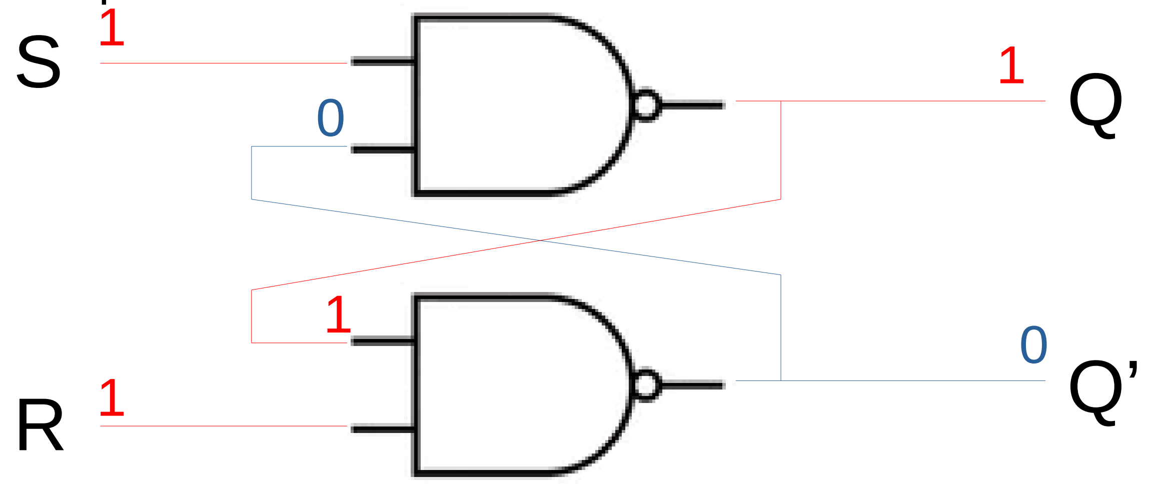
0 pulse applied for a short time



When input S is made low momentarily, that when S become 0, and R still 1, the outputed Q becomes 1, the latch is now storing 1,

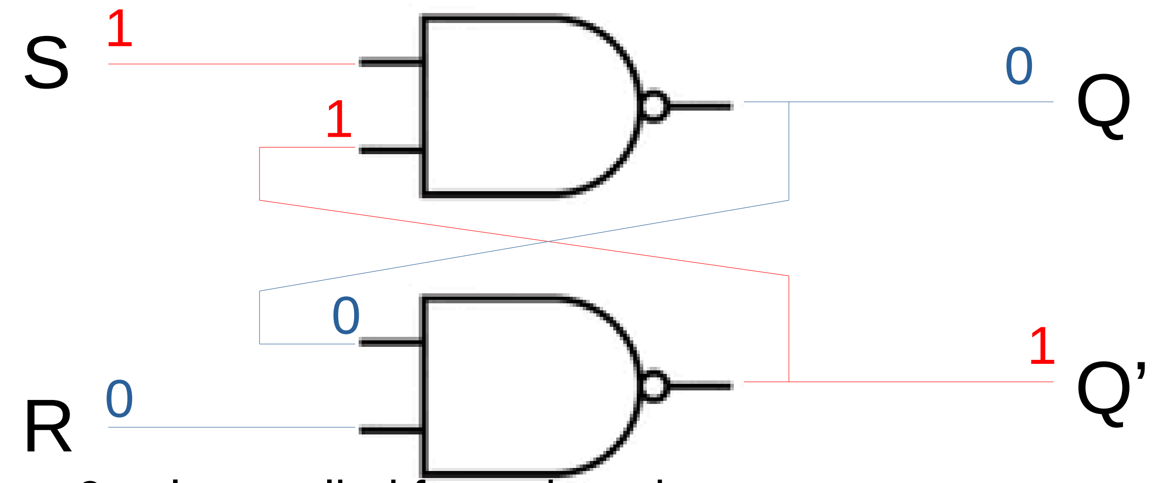
S	R	Q	Q’
0	0	1	1
0	1	1	0
1	0	0	1
1	1	0 1	1 0

0 pulse removed



and S can be returned to its normal high state.

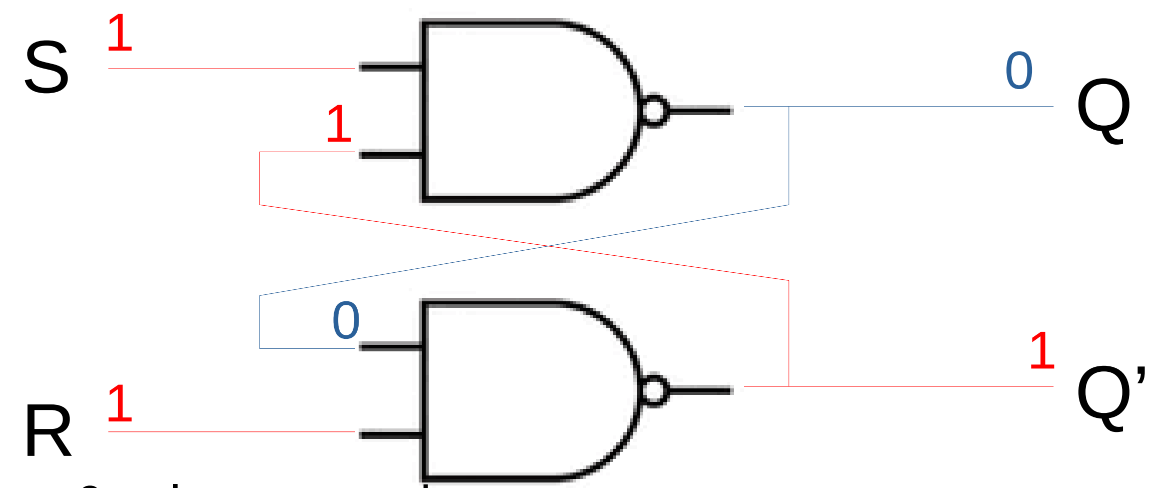
S	R	Q	Q’
0	0	1	1
0	1	1	0
1	0	0	1
1	1	0 1	1 0



0 pulse applied for a short time

When R is set low momentarily, the output Q is changed to 0,

S	R	Q	Q’
0	0	1	1
0	1	1	0
1	0	0	1
1	1	0 1	1 0

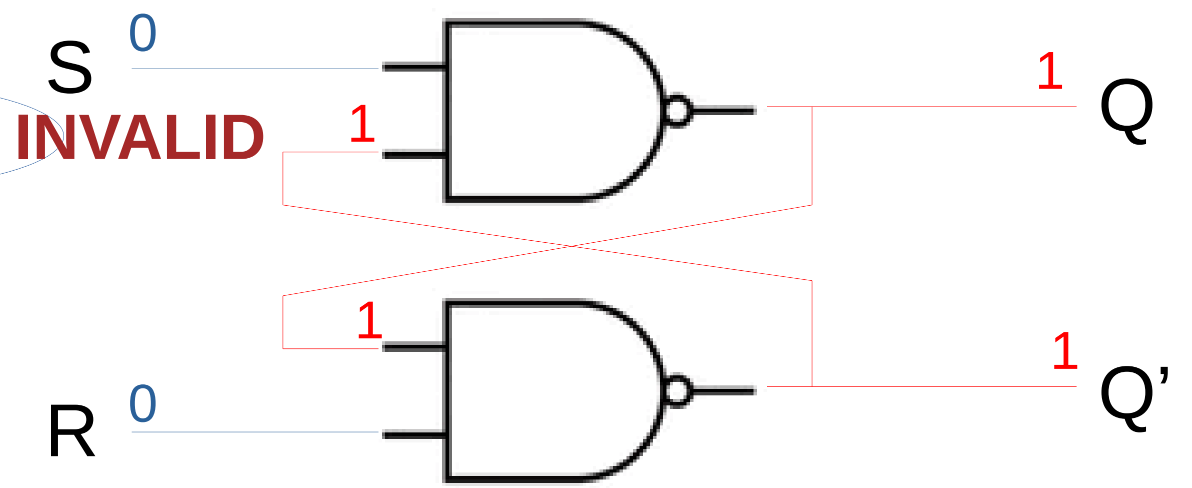


0 pulse removed

R can return to its normal high value, and the latch is storing a 0 again.

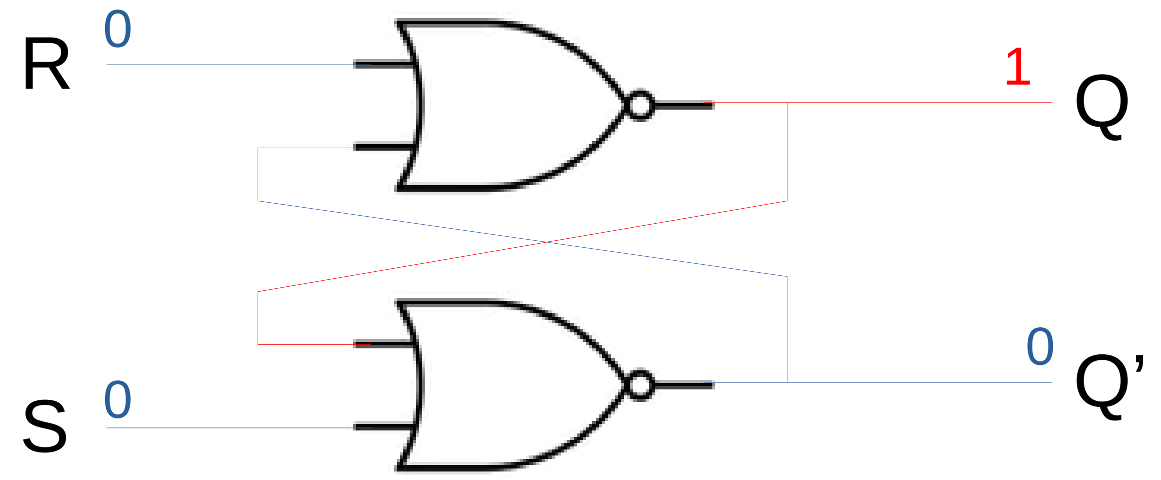
An SR latch built from NAND gates like this, is more explicitly known as an *active low* SR latch.

S	R	Q	Q’
0	0	1	1
0	1	1	0
1	0	0	1
1	1	0 1	1 0

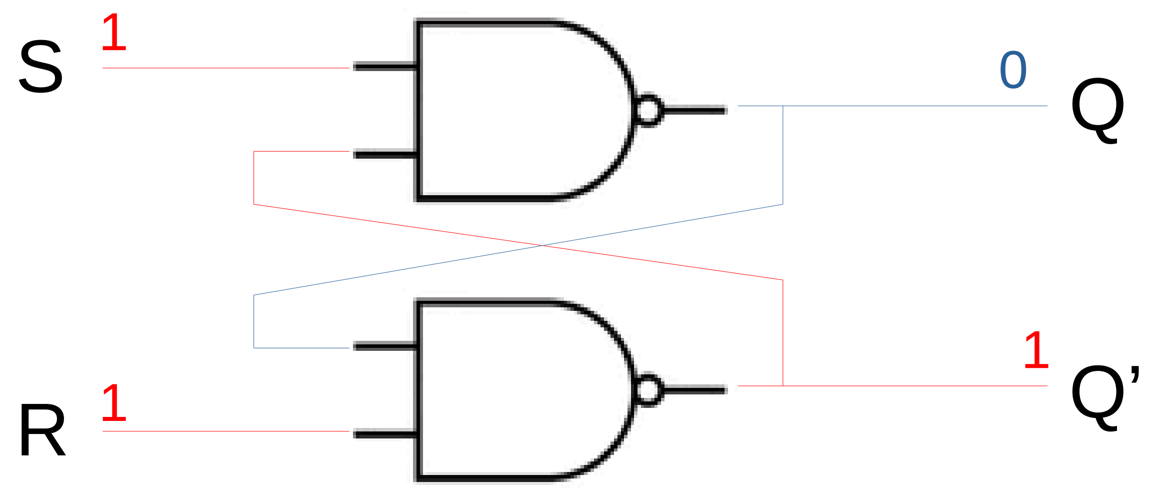


An SR latch based on NAND gates also has its forbidden state, that is when both S and R are simultaneously 0. This will result in **illegal** state, in which both Q and its complement are 1.

S	R	Q	Q’
0	0	1 0	0 1
0	1	0	1
1	0	1	0
1	1	0	0

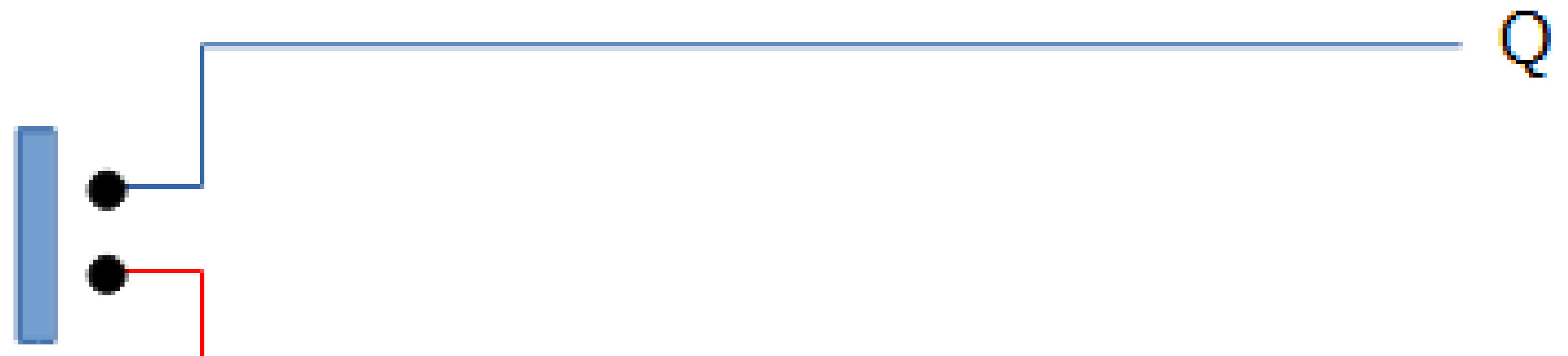


S	R	Q	Q’
0	0	1	1
0	1	1	0
1	0	0	1
1	1	0 1	1 0

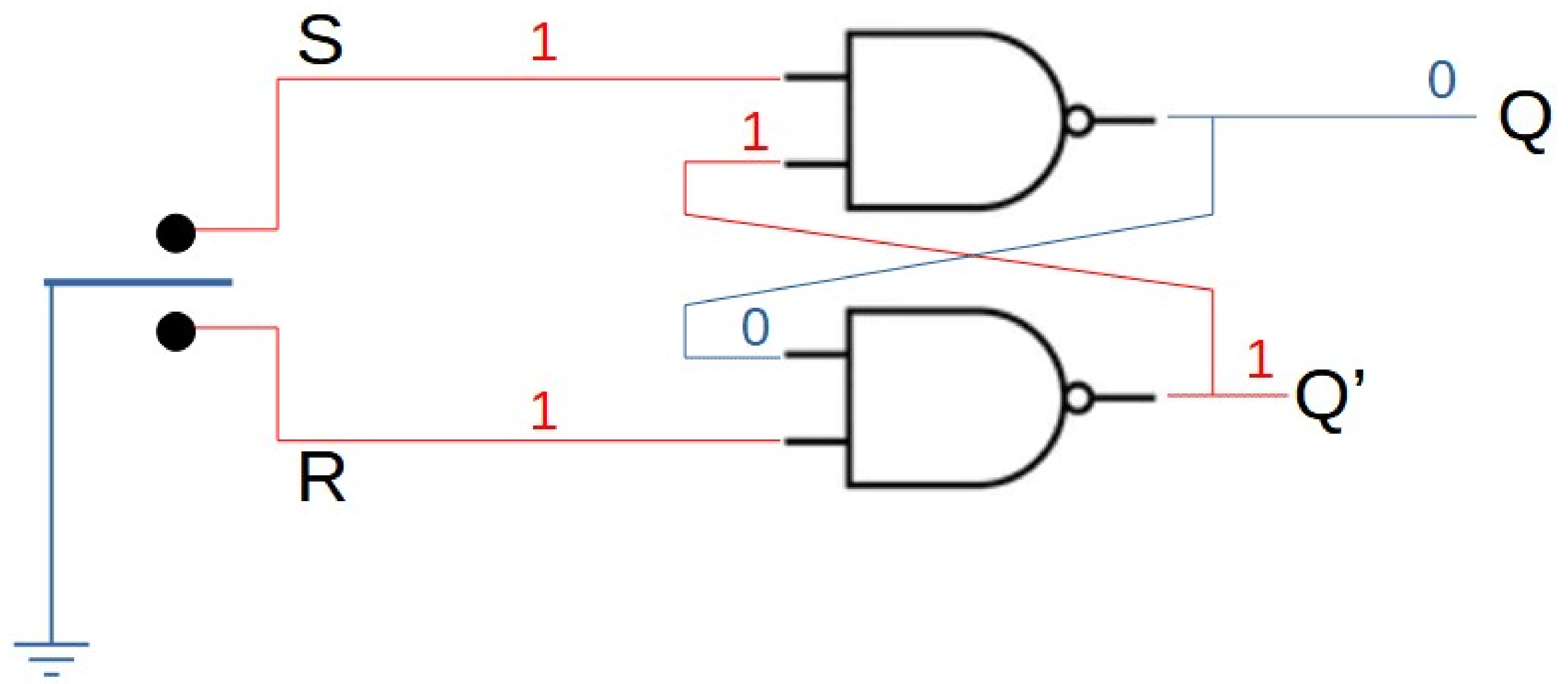


To summarize, SR latch can be built from NOR gates, or from NAND gates. Both types do the same job, but they just controlled in a slightly different way. The NOR gates based SR latch is set or reset with high logic, that is, it is an **active high** SR latch. The NAND gates SR latch, on the other hand, is set or reset with low logic, that is, it is an **active low** SR latch.

Let’s consider a particular application of an SR latch.



When a mechanical switch is pressed, it may actually generate several electrical signals in a tiny fraction of a second, when only one signal is required. Lots of on-off signals, like this, could then cause problem with the circuit that the switch is supposed to be controlling. The effect is known as **switch bounce**.



An integrated circuit with an SR latch on it can be purchased commercially to allow clean interfacing between a mechanical switch and the digital circuit it's controlling. Here we consider a switch that will connect input S , which is normally high to earth, making it low, thereby changing the output of this NAND based SR latch from low to high. This SR latch will ignore any farther set signals after it already been set, so It's serving to debounce the signal from the mechanical switch.

You can imagine other systems which might make good use of this debouncing effect. For example, a burglar alarm may be triggered when a window or a door is opened, but we don't want the alarm bell to stop ringing when the burglar shuts the door.

On its own, the SR latch has a few uses, , mainly in control applications, when we need to monitor some condition that may change, or change back again, and react accordingly. But more importantly,

the simple SR latch is a building block of sophisticated memory circuits.

SR Flip-Flop

The graphic symbol of the SR flip-flop is shown in Fig. 1-19(a). It has three inputs, labeled S (for set), R (for reset), and C (for clock). It has an output Q and sometimes the flip-flop has a complemented output, which is indicated with a small circle at the other output terminal. There is an arrowhead-shaped symbol in front of the letter C to designate a dynamic input. The dynamic indicator symbol denotes the fact that the flip-flop responds to a positive transition (from 0 to 1) of the input clock signal.

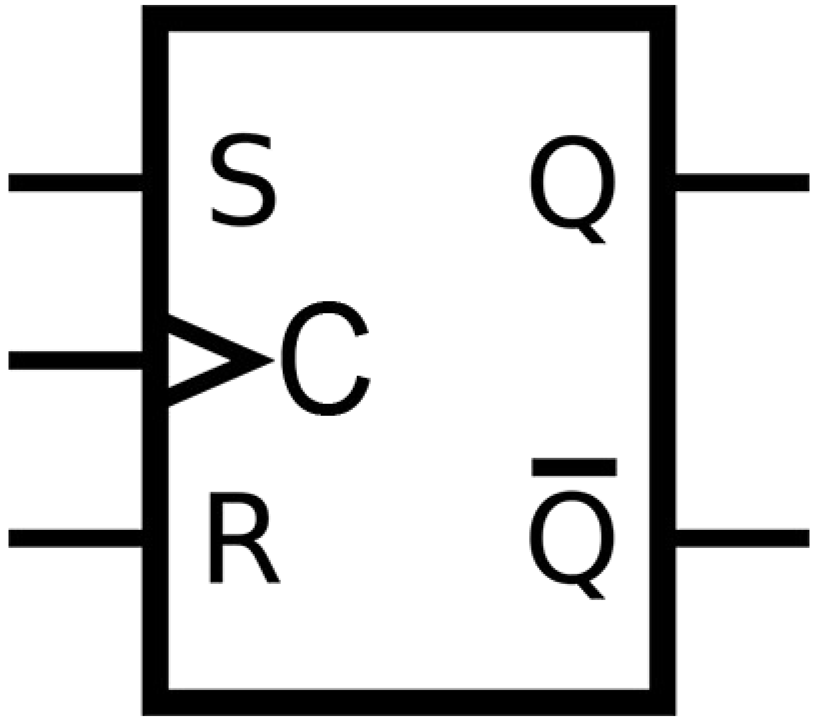
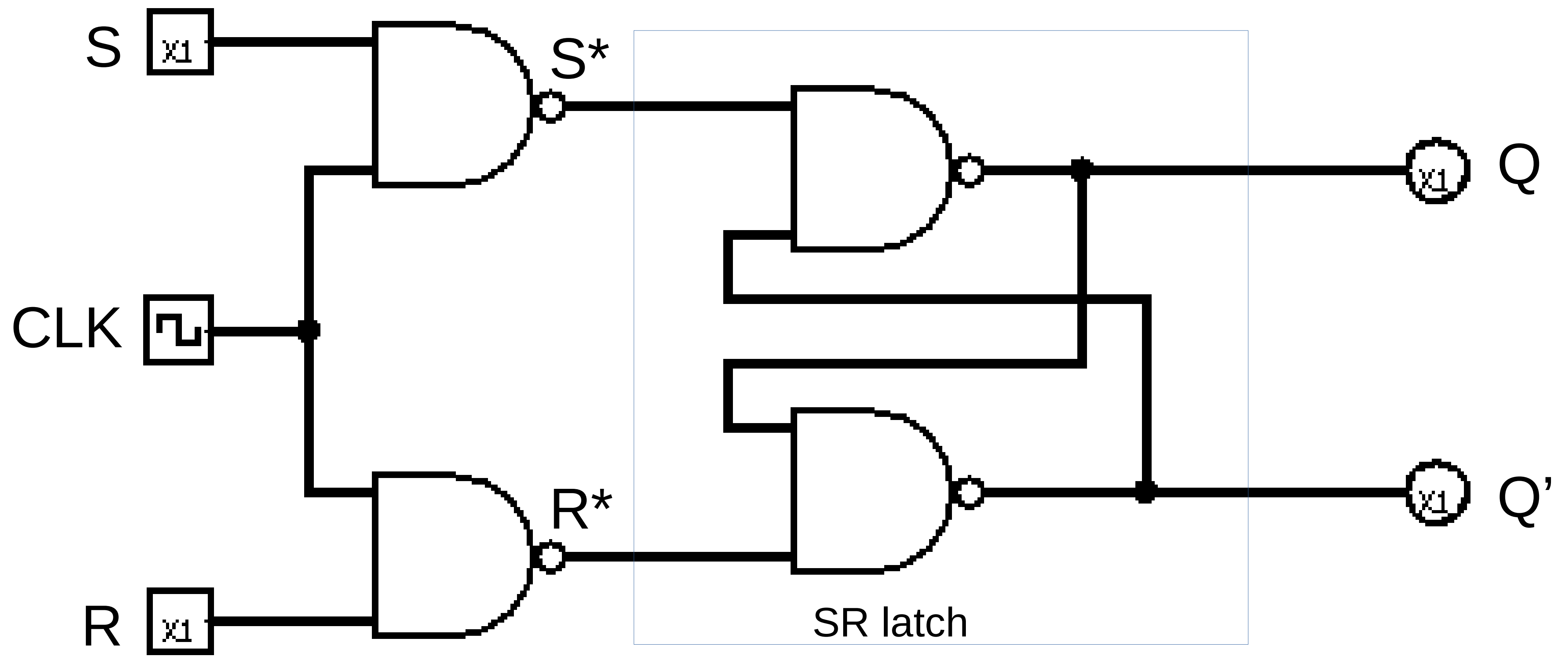


Figure 1-19 (a). Graphic symbol of SR Flip-Flop

S	R	Q(t + 1)	
0	0	Q(t)	No change
0	1	0	Clear to 0
1	0	1	Set to 1
1	1	?	Indeterminate

Figure 1-19 (b). Characteristic table of SR Flip-Flop

The operation of the SR flip-flop is as follows . If there is no signal at the clock input C, the output of the circuit cannot change irrespective of the values at inputs S and R . Only when the clock signal changes from 0 to 1 can the output be affected according to the values in inputs S and R . If S = 1 and R = 0 when C changes from 0 to 1, output Q is set to 1 . If S = 0 and R = 1 when C changes from 0 to 1, output Q is cleared to 0. If both S and R are 0 during the clock transition, the output does not change . When both S and R are equal to 1, the output is unpredictable and may go to either 0 or 1, depending on internal timing delays that occur within the circuit. The characteristic table shown in Fig. 1-19(b) summarizes the operation of the SR flip-flop in tabular form. The S and R columns give the binary values of the two inputs . Q(t) is the binary state of the Q output at a given time (referred to as present state). Q(t + 1) is the binary state of the Q output after the occurrence of a clock transition (referred to as next state). If S = R = 0, a clock transition produces no change of state [i. e . , Q(t + 1) = Q(t)] . If S = 0 and R = 1 , the flip-flop goes to the 0 (clear) state . If S = 1 and R = 0, the flip-flop goes to the 1 (set) state . The SR flip-flop should not be pulsed when S = R = 1 since it produces an indeterminate next state . This indeterminate condition makes the SR flip-flop difficult to manage and therefore it is seldom used in practice .



In this SR flip-flop, made of four NAND gates, the part encircled with the blue square is an SR latch by itself.

S*	R*	Q	Q'
0	0	1	1
0	1	1	0
1	0	0	1
1	1	0 1	1 0

SR latch truth table

So, there is a state when both inputs are 0 is illegal and not used. In this kind of SR latch, the normal state of inputs is 1, and in this normal and persistent state, the output Q is 0, and its complement, of course 1. When short 0 pulse is applied to S*, the output Q is changes to 1, and its complement to 0, accordingly. When short 0 pulse is applied to R*, the output Q is changes to 0. So, most of the time, when no pulses applied and the inputs S* and R* are in their normal 1 state, the values of the output Q and its complement may be 0 and 1, or 1 and 0, depending on the previously applied pulse. This is the memory mechanism of the latch.

The other components of an SR flip-flop are two more NAND gates, and a clock unit, that is used to generate pulses at steady intervals. The clock purpose is to set a pace for the latch input changes occurrings. This means, S* and R* can change their values only when a pulse of the clock is occurring.

So, lets find out what is the boolean expressions for S* and R*:

$$\left.\begin{array}{l} S^* = (S^*CLK)' = S' + CLK' \\ R^* = (R^*CLK)' = R' + CLK' \end{array}\right\}$$

S* ans R* are just outputs or regular NAND gates

Now lets build a truth table for the whole circuit, that is our flip-flop. Note that the clock isn’t a part of the flip-flop circuit, it’s just provide one of the necessary inputs.

$S^* = (S*CLK)' = S' + CLK'$
$R^* = (R*CLK)' = R' + CLK'$

CLK	S	R	Q	Q'
0	x	x	**0** **1**	**1** **0**
1	0	0	**0** **1**	**1** **0**
1	0	1	**0**	**1**
1	1	0	**1**	**0**
1	1	1	**1**	**1**

memory

INVALID, NOT USED CASE

NAND

A	B	x
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**0**

CLK = 0
S* is output of NAND gate where inputs are S and CLK, so when CLK is 0 no matter what the value of the input S the output always will be 1.

CLK = 0 => S* = 1

Similarly, R* also an output of a NAND gate, so, when CLK is 0 R* always will be 1.

CLK = 0 => R* = 1

Therefore, when the clock is 0, the values of S and R don't change the values of S* and R* and the final output of Q, hence for CLK = 0, S and R are don't care conditions. Q and its complement Q' in this situation are storing memory values and they depend on what was stored previously.

So, when the clock is low and no clock pulse occurs, no matter how we change the flip-flop's inputs, the stored values of the output won't be affected. The purpose of the clock is to control when changing the latch's output is possible.

$\left.\begin{array}{l} CLK = 1 \\ S^* = S' + CLK' \end{array}\right\}$ $S^* = S' + 0 = S'$

$\left.\begin{array}{l} CLK = 1 \\ R^* = R' + CLK' \end{array}\right\}$ $R^* = R' + 0 = R'$

$\left.\begin{array}{l} CLK = 1 \\ S = 0 \end{array}\right\}$ $S^* = 1$

$\left.\begin{array}{l} CLK = 1 \\ R = 0 \end{array}\right\}$ $R^* = 1$

Hence, when the clock is 1 but S is 0 and R is 0, this combination is also doesn't affect the output Q, therefore, similarly to the previous state, Q and Q' depend only on the previous values stored in them and not on the inputs, so this is also memory position.

So, in the two first rows of the truth table, the flip-flop’s output values didn’t affected by the inputs, and just did store their previous memorized positions.

$\left.\begin{matrix}\text{CLK} = 1 \\ \text{S} = 0\end{matrix}\right\}$ S* = 1 (NAND gate) $\left.\begin{matrix}\text{CLK} = 1 \\ \text{R} = 1\end{matrix}\right\}$ R* = 0 (NAND gate)

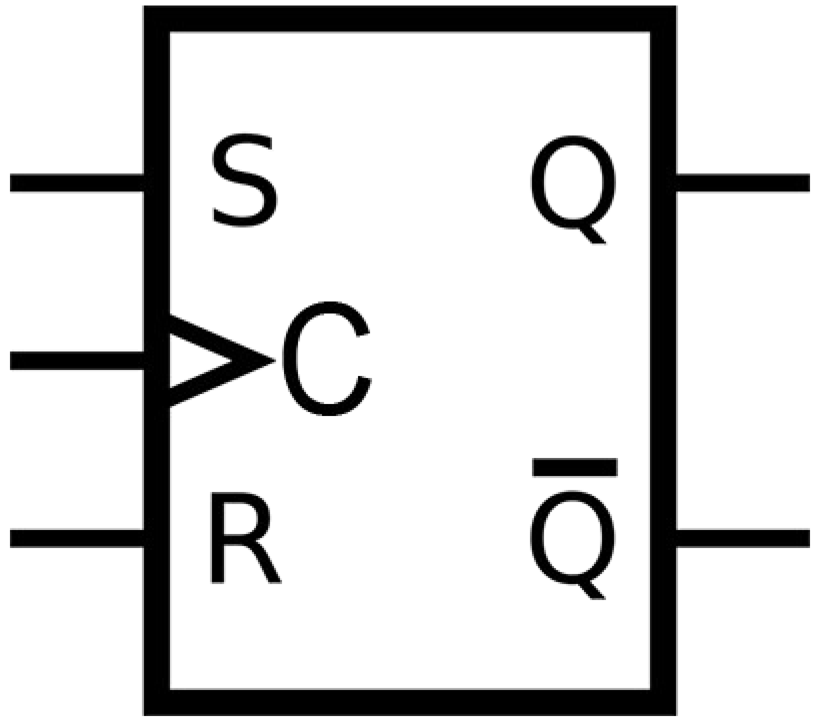
By the truth table of the SR latch, that we already know how it behaves, S* = 1 and R* = 0 means Q = 0 and Q’ = 1.

$\left.\begin{matrix}\text{CLK} = 1 \\ \text{S} = 1\end{matrix}\right\}$ S* = 0 (NAND gate) $\left.\begin{matrix}\text{CLK} = 1 \\ \text{R} = 0\end{matrix}\right\}$ R* = 1 (NAND gate)

By the truth table of the SR latch, that we already know how it behaves, S* = 0 and R* = 1 means Q = 1 and Q’ = 0.

$\left.\begin{matrix}\text{CLK} = 1 \\ \text{S} = 1\end{matrix}\right\}$ S* = 0 (NAND gate) $\left.\begin{matrix}\text{CLK} = 1 \\ \text{R} = 1\end{matrix}\right\}$ R* = 0 (NAND gate)

By the truth table of the SR latch, that we already know how it behaves, in the case of both S and R inputs are 1 and this is when a short clock pulse occurs (CLK = 1), we have a contradictory condition where the outputs, Q and its complement Q’, both struggling to be 1. So, this competition between them is a race condition, and it needs to be avoided.



This is the graphic symbol of SR flip-flop, that represents the whole circuit. The triangle-like symbol just before C letter means that this flip-flop is **edge-triggered**. The absence of this symbol tells that the flip-flop is **level-triggered**.

In **edge triggering** the circuit becomes active at negative or positive edge of the clock signal. For example if the circuit is positive edge triggered, it will take input at exactly the time in which the clock signal goes from low to high. Similarly input is taken at exactly the time in which the clock signal goes from high to low in negative edge triggering. But keep in mind after the the input, it can be processed in all the time till the next input is taken.

In **level triggering** the circuit will become active when the gating or clock pulse is on a particular level. This level is decided by the designer. We can have a negative level triggering in which the circuit is active when the clock signal is low or a positive level triggering in which the circuit is active when the clock signal is high.

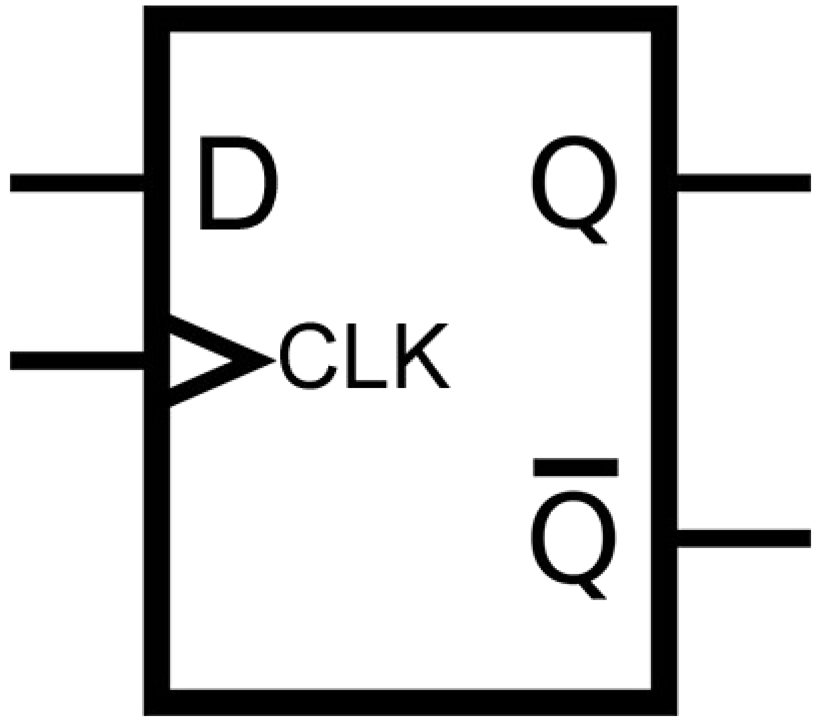
Purpose of Flip-Flops

The primary function of a flip-flop is storing exclusively one of the two stable voltage levels (corresponding to a binary logic value 0, and 1) until a new voltage level (a new logic value) is written to. More specifically, in an electronic computer, it is used to implement a sequential logic, where the output of the logic circuit depends not only on the present value but also on the previous states.

The difference between flip-flops and latches

The main difference between those circuits is the clocking (or gating) mechanism. Generally, flip-flops are triggered by the edge of the clock signal. Latches lack the clocking mechanism, therefore, they’re triggered by a certain level of the input signals. The advantage of flip—flops, in this regard, that is they can be controlled more precisely (and by less precise pulses), and also synchronized with other circuits that share the clock with them.

D Flip-Flop



(a) Graphic symbol

D	Q_{t+1}	
0	0	Clear to 0
1	1	Set to 1

(b) Characteristic table

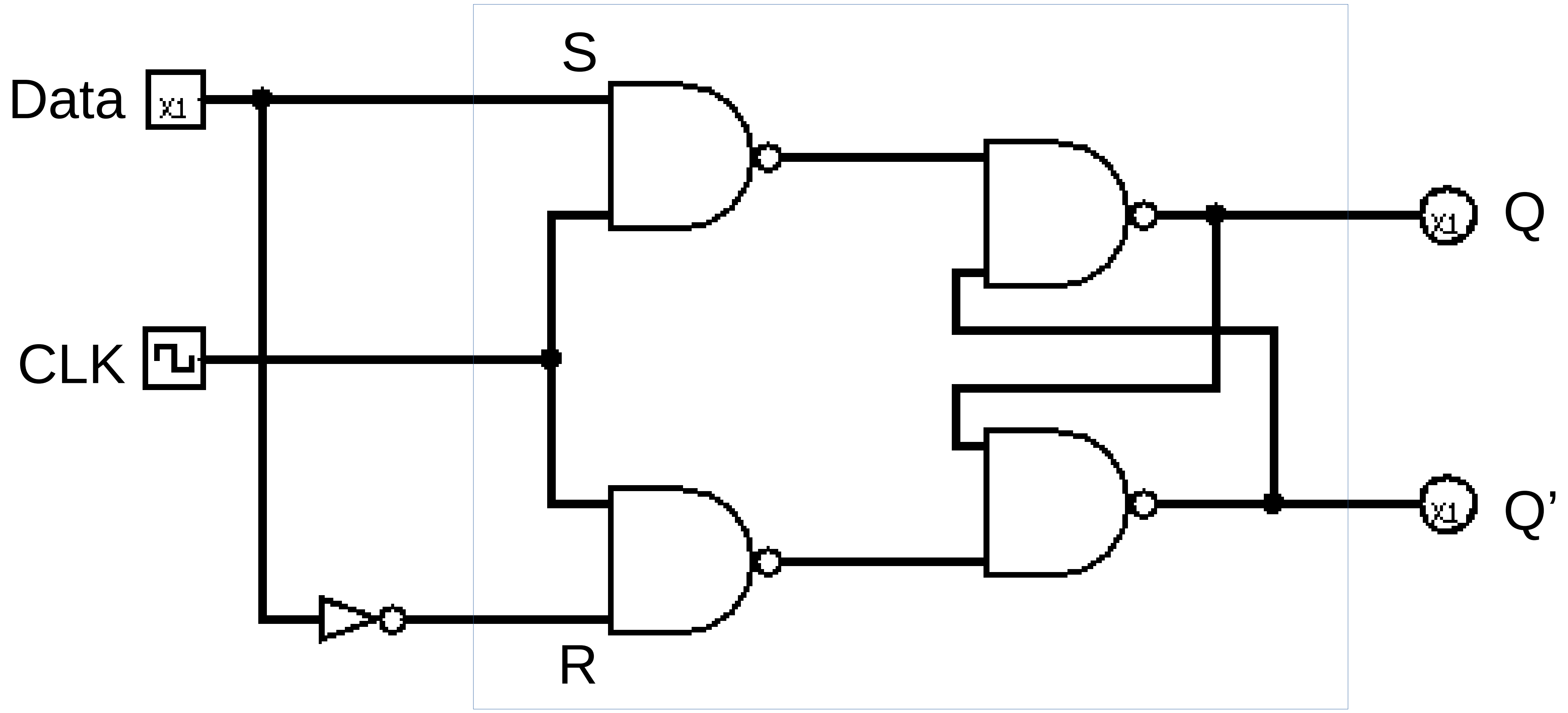
Figure 1-20. D Flip-Flop

The D (data) flip-flop is a slight modification of the SR flip-flop . An SR flip-flop is converted to a D flip-flop by inserting an inverter between S and R and assigning the symbol D to the single input. The D input is sampled during the occurrence of a clock transition from 0 to 1. If D = 1, the output of the flip-flop goes to the 1 state, but if D = 0, the output of the flip-flop goes to the 0 state . The graphic symbol and characteristic table of the D flip-flop are shown in Fig. 1-20. From the characteristic table we note that the next state Q(t + 1) is determined from the D input.

The relationship can be expressed by a characteristic equation:

$$Q(t+1) = D$$

This means that the Q output of the flip-flop receives its value from the D input every time that the clock signal goes through a transition from 0 to 1 . Note that no input condition exists that will leave the state o f the D flip-flop unchanged. Although a D flip-flop has the advantage of having only one input (excluding CLK), it has the disadvantage that its characteristic table does not have a "no change" condition Q(t + 1) = Q(t). The "no change" condition can be accomplished either by disabling the clock signal or by feeding the output back into the input, so that clock pulses keep the state of the flip-flop unchanged.



Gated SR Flip-Flop

D Flip-Flop circuit

CLK	S	R	Q	Q'
0	x	x	**0** **1**	**1** **0**
1	0	0	**0** **1**	**1** **0**
1	0	1	**0**	**1**
1	1	0	**1**	**0**
1	1	1	**1**	**1**

SR Flip-Flop truth table

We can see D flip-flop as a modification of SR flip-flop. Let's examine SR flip-flop's truth table. You can see for storing a 0, the inputs s and R must be 0 and 1 respectively, and for storing a 1, they must be 1 and 0. We can also see, that the inputs S and R are always complemented to each other, so when S is 1 R is always 0, and vice versa. So, from this, we can draw the conclusion that there is no need to give those inputs individually. We can just give one input to the flip-flop and complement the other. This way, we're getting a D flip-flop.

Therefore, in this kind of flip-flop, we have just one input D, which is equivalent to S of SR flip-flop, and the complement of the input D (using inverter) is the equivalent of R of SR flip-flop.

So, let's make a truth table for this kind of flip-flops.

In this truth table, D stands for the single input (data), Q_{t+1} is the current output, and Q_t is the previous output (t stands for time)

CLK	D	Q_{t+1}
0	x	Q_t
1	0	0
1	1	1

D Flip-Flop truth table

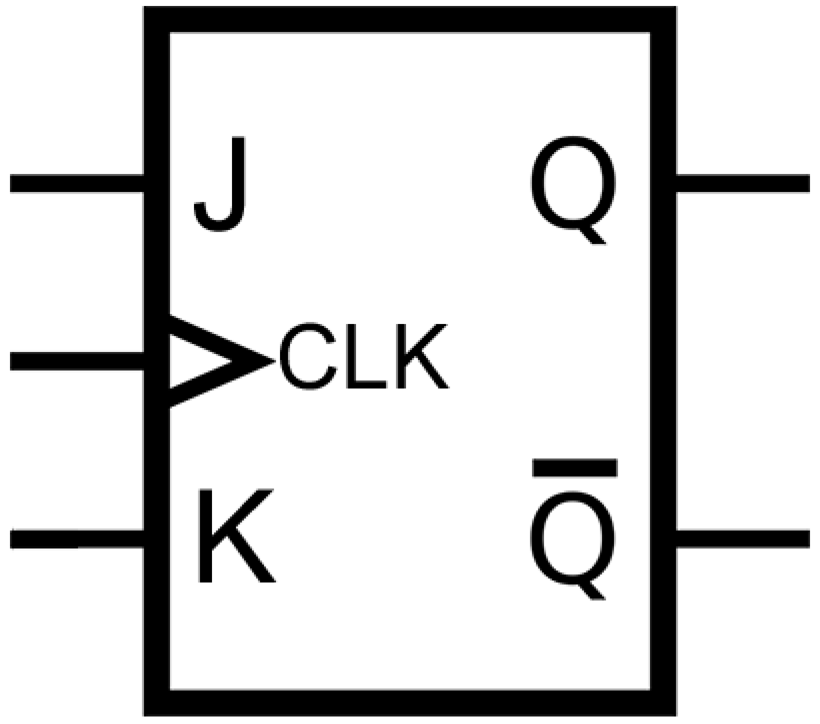
Because SR flip-flop is gated and its inputs are synchronized by the clock (reference the page 104), when the clock is 0 the flip—flop isn't take any input no matter what the state of D, so in this case D is no-care conditioned.
The output in this case will be as it was set previously, memorizing the previous state.

$\left.\begin{matrix}CLK = 1\\ D = 0\end{matrix}\right\}$ $\left.\begin{matrix}S = 0\\ R = 1\end{matrix}\right\}$ $Q_{t+1}=0$ (behavior of SR flip-flop, see page 104)

$\left.\begin{matrix}CLK = 1\\ D = 1\end{matrix}\right\}$ $\left.\begin{matrix}S = 1\\ R = 0\end{matrix}\right\}$ $Q_{t+1}=1$ (behavior of SR flip-flop, see page 104)

Also, the illegal condition of an SR flip-flop, when the clock and both of the inputs S and R are 1 and this lead to a race condition, is impossible in D flip-flop, because it has only one input D, and internal R is the opposite to D constantly.

JK Flip-Flop



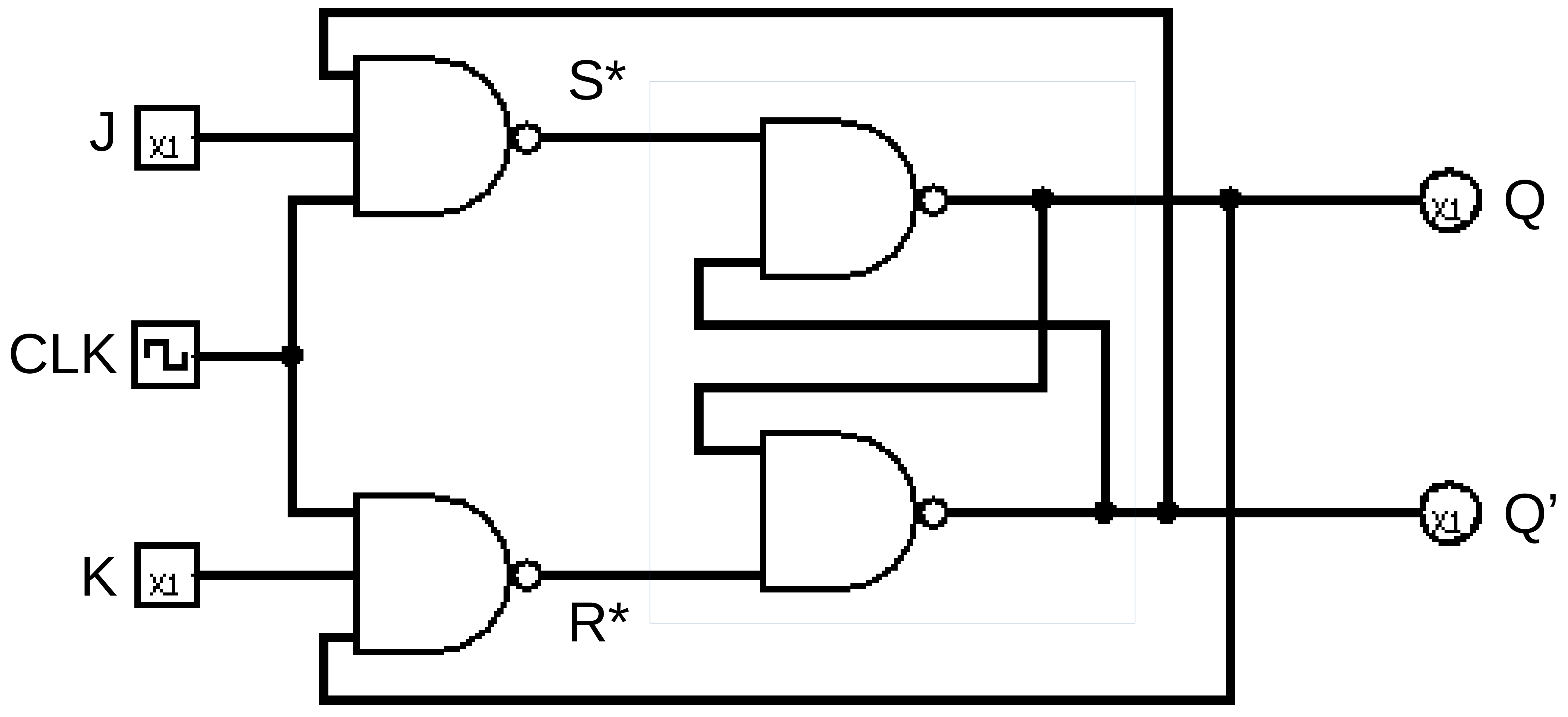
(a) Graphic symbol

J	K	Q_{t+1}	
0	0	Q_t	No change
0	1	0	Clear to 0
1	0	1	Set to 1
1	1	Q'_t	Complement

(b) Characteristic table

Figure 1-21. JK Flip-Flop

A JK flip-flop is a refinement of the SR flip-flop in that the indeterminate condition of the SR type is defined in the JK type. Inputs J and K behave like inputs S and R to set and clear the flip-flop, respectively. When inputs J and K are both equal to 1, a clock transition switches the outputs of the flip-flop to their complement state. The graphic symbol and characteristic table of the JK flip-flop are shown in Fig. 1-21 . The J input is equivalent to the S (set) input of the SR flip-flop, and the K input is equivalent to the R (clear) input. Instead of the indeterminate condition, the JK flip-flop has a complement condition $Q(t + 1) = Q'(t)$ when both J and K are equal to 1 .



JK Flip-Flop circuit

CLK	S	R	Q	Q'
0	x	x	**0** **1**	**1** **0**
1	0	0	**0** **1**	**1** **0**
1	0	1	**0**	**1**
1	1	0	**1**	**0**
1	1	1	**1**	**1**

SR Flip-Flop truth table

The JK Flip-flop is similar to the SR Flip-flop but there is no change in state when the J and K inputs are both low. Contrarily to SR flip-flop, where this state it leads to a race condition and therefore is forbidden to use, in JK flip-flop there is some usable form of this state.

In JK flip-flop, the output Q and its complement are fed back to the flip-flop as its feedback inputs. Three inputted NAND gates are used to acquire the J and K inputs, the clock signals, and the fed back outputs.

Because the NAND gate behaves a way that its output is 0 only if all the inputs are 1, otherwise the output is 1, so in any combination where CLK is 0 the values of S* and R* are always wlill be 1. S* and R* are serving as inputs of the SR latch (the blue square), so the output Q is in the memory state, means its value is the same as the previous.

$$\text{CLK} = 0 \longrightarrow \begin{matrix} S^* = 1 \\ R^* = 1 \end{matrix} \longrightarrow \text{Q is in a memory state}$$

When the clock is 1 and the inputs J and K are 1 and 0 respectively, the flip-flop behaves as an SR flip-flop, so the output Q will be 1, and its complement respectively 0.

$$\begin{matrix} \text{CLK} = 1 \\ J = 1 \\ K = 0 \end{matrix} \longrightarrow \begin{matrix} S^* = 0 \\ R^* = 1 \end{matrix} \longrightarrow \left.\begin{matrix} Q = 1 \\ Q' = 0 \end{matrix}\right\} \text{(reference page 105)}$$

The situation when the clock is 1, J is 0 and K is 1 is the opposite of the previous situation, so Q is 0. The flip-flop still behaves as an SR flip-flop.

$$\begin{array}{l} CLK = 1 \\ J = 0 \\ K = 1 \end{array} \longrightarrow \begin{array}{l} S^* = 1 \\ R^* = 0 \end{array} \longrightarrow \left.\begin{array}{l} Q = 0 \\ Q' = 1 \end{array}\right\}$$ (reference page 105)

So now let’s deal with the very important state where both the inputs J and K are 1 and the clock is also 1. This state leads to race condition in SR flip-flops, so it is illegal and not used in them. Contrarily, in JK flip-flop this state is used, so this makes the difference between SR and JK flip-flops.

In order to deal with this state, firstly we need to assume that the value of the output Q is 0, and Q’ is 1. Q is given as an input to the three-inputted K-input NAND gate. So, there're the inputs of the bottom three-inputted gate:

$$Q_{input} = 0, K = 1, CLK = 1$$

So, according to the behavior of NAND gate, that is gives 0 only if all the inputs are 1, and 1 in any other case, R* in this case is 1.

Similarly, the complement Q’ is given as an input to the J-input (the top) three-inputted NAND gate.

$$Q_{input} = 1, K = 1, CLK = 1$$

In this case, because all of the gate’s inputs are 1, the value of S* is 0.
So, the SR latch, which is the part of the flip-flop, is given the inputs 0 and 1, and. According to the behavior of an SR latch (reference page 95), the output Q is 0. This mean, the output Q Is changed, the input will also change, because of the outputs that fed back into the input, so the third input of the bottom three-inputted gate becomes 1, and the third input of the top three-inputted gate becomes 0, so R* become 0, and S* become1. When this happens the output Q id becomes 0 and Q’ is 1. When doing the circuit analysis more times, we’ll go through the same cycle endlessly, when

Q will change its value very fast:

Q = 0, 1, 0, 1, 0, 1, 0, 1 …
Q’ = 1, 0, 1, 0, 1, 0, 1, 0 …

This condition is called **racing**, a very important concept in digital electronics, because it’s used in counters.

So, when a race occurs, Q_{t+1}, that is the output after the clock transition, is a compliment of its previous state Q_t.
The value of the racing conditioned output will always be a complement of its previous state.
So, at every given moment, the value of the racing state Q_{t+1} will be's previous state complement $(Q_t)'$
Using racing condition this way in JK flip-flops is called toggling.

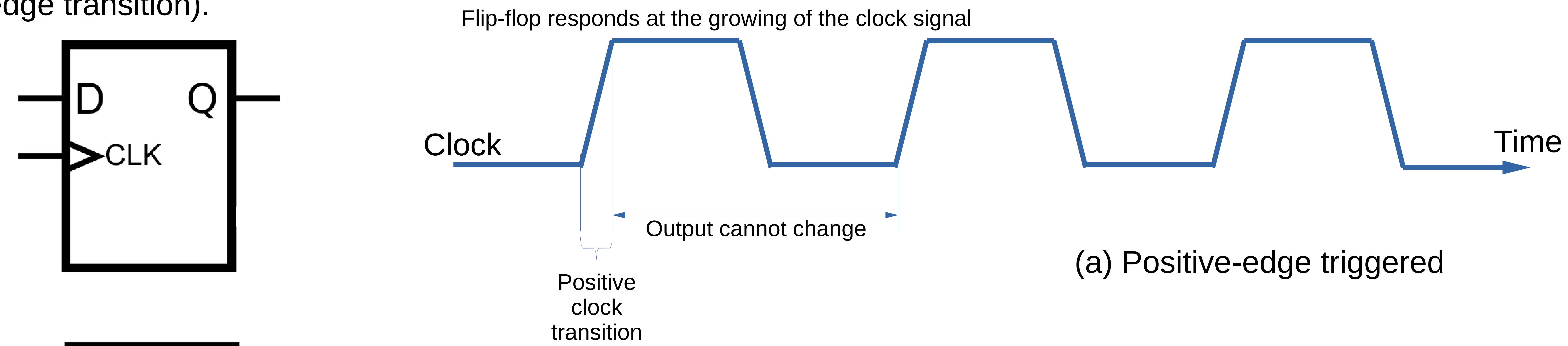
Finally, this is a truth table for a JK flip-flop:

CLK	J	K	Q_{t+1}
0	x	x	Q_t (memory)
1	0	0	Q_t (memory)
1	0	1	0
1	1	0	1
1	1	1	$(Q_t)'$

When given 1 at both inputs when the clock is also 1, the output value will change very fast between 1 and 0.

How Edge-Triggering Works

The most common type of flip-flop used to synchronize the state change during a clock pulse transition is the edge-triggered flip-flop. In this type of flip-flop, output transitions occur at a specific level of the clock pulse. When the pulse input level exceeds this threshold level, the inputs are locked out so that the flip-flop is unresponsive to further changes in inputs until the clock pulse returns to 0 and another pulse occurs. Some edge-triggered flip-flops cause a transition on the rising edge of the clock signal (positive-edge transition), and others cause a transition on the falling edge (negative-edge transition).


D
Q
CLK
Flip-flop responds at the growing of the clock signal
Clock
Time
Output cannot change
Positive clock transition
(a) Positive-edge triggered

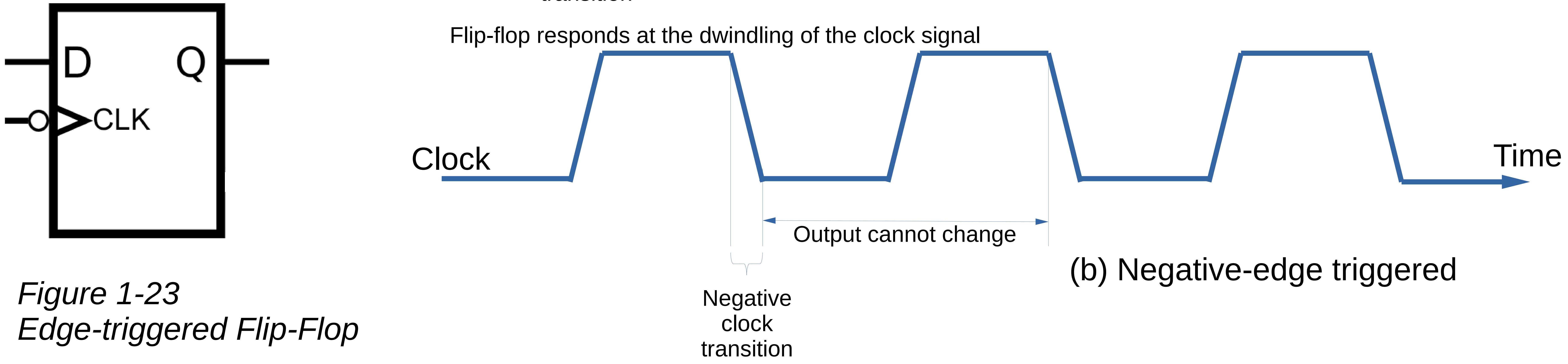

D
Q
CLK
Flip-flop responds at the dwindling of the clock signal
Clock
Time
Output cannot change
Negative clock transition
(b) Negative-edge triggered

Figure 1-23
Edge-triggered Flip-Flop

Figure 1-23(a) shows the clock pulse signal in a positive-edge-triggered D flip-flop. The value in the D input is transferred to the Q output when the clock makes a positive transition. The output cannot change when the clock is in the 1 level, in the 0 level, or in a transition from the 1 level to the 0 level.

The effective positive clock transition includes a minimum time called the setup time in which the D input must remain at a constant value before the transition, and a definite time called the hold time in which the D input must not change after the positive transition. The effective positive transition is usually a very small fraction of the total period of the clock pulse.

Figure 1-23(b) shows the corresponding graphic symbol and timing diagram for a negative-edge-triggered D flip-flop. The graphic symbol includes a negation small circle in front of the dynamic indicator at the C input. This denotes a negative-edge-triggered behavior. In this case the flip-flop responds to a transition from the 1 level to the 0 level of the clock signal.

Another type of flip-flop used in some systems is the master-slave flipflop. This type of circuit consists of two flip-flops . The first is the master, which responds to the positive level of the clock, and the second is the slave, which responds to the negative level of the clock. The result is that the output changes during the 1-to-0 transition of the clock signal. The trend is away from the use of master-slave flip-flops and toward edge-triggered flip-flops.

Flip-flops available in integrated circuit packages will sometimes provide special input terminals for setting or clearing the flip-flop asynchronously. These inputs are usually called "preset" and "clear." They affect the flip-flop on a negative level of the input signal without the need of a clock pulse. These inputs are useful for bringing the flip-flops to an initial state prior to its clocked operation.

Excitation Tables

SR flip-flop			
Q_t	Q_{t+1}	S	R
0	0	**0**	**x**
0	1	**1**	**0**
1	0	**0**	**1**
1	1	**x**	**0**

D flip-flop		
Q_t	Q_{t+1}	D
0	0	**0**
0	1	**1**
1	0	**0**
1	1	**1**

JK flip-flop			
Q_t	Q_{t+1}	J	K
0	0	**0**	**x**
0	1	**1**	**x**
1	0	**x**	**1**
1	1	**x**	**0**

T flip-flop		
Q_t	Q_{t+1}	T
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

Table 1-3

The characteristic tables of flip-flops specify the next state when the inputs and the present state are known. During the design of sequential circuits we usually know the required transition from present state to next state and wish to find the flip-flop input conditions that will cause the required transition. For this reason we need a table that lists the required input combinations for a given change of state. Such a table is called a flip-flop excitation table.

Table 1-3 lists the excitation tables for the four types of flip-flops. Each table consists of two columns, Q(t) and Q(t + 1), and a column for each input to show how the required transition is achieved. There are four possible transitions from present state Q(t) to next state Q(t + 1). The required input conditions for each of these transitions are derived from the information available in the characteristic tables. The symbol x in the tables represents a don't-care condition; that is, it does not matter whether the input to the flip-flop is 0 or 1 .

The reason for the don't-care conditions in the excitation tables is that there are two ways of achieving the required transition. For example, in a JK flip-flop, a transition from present state of 0 to a next state of 0 can be achieved by having inputs J and K equal to 0 (to obtain no change) or by letting J = 0 and K = 1 to clear the flip-flop (although it is already cleared). In both cases J must be 0, but K is 0 in the first case and 1 in the second. Since the required transition will occur in either case, we mark the K input with a don't-care x and let the designer choose either 0 or 1 for the K input, whichever is more convenient.

Excitation Tables - detailed

The excitation table has the minimum inputs, which will excite or trigger the flip flop to go from its present state to the next state. It is derived from the truth table.

Generally, the operation of each flip flop is explained with the help of the truth table. The truth table has all the input combinations, for which the flip flop reacts to produce the next state output.

The excitation table consists of two columns for present state(Q_t) and next state(Q_{t+1}) and one or two column for each inputs. The input columns depend on the type of the flip flop.

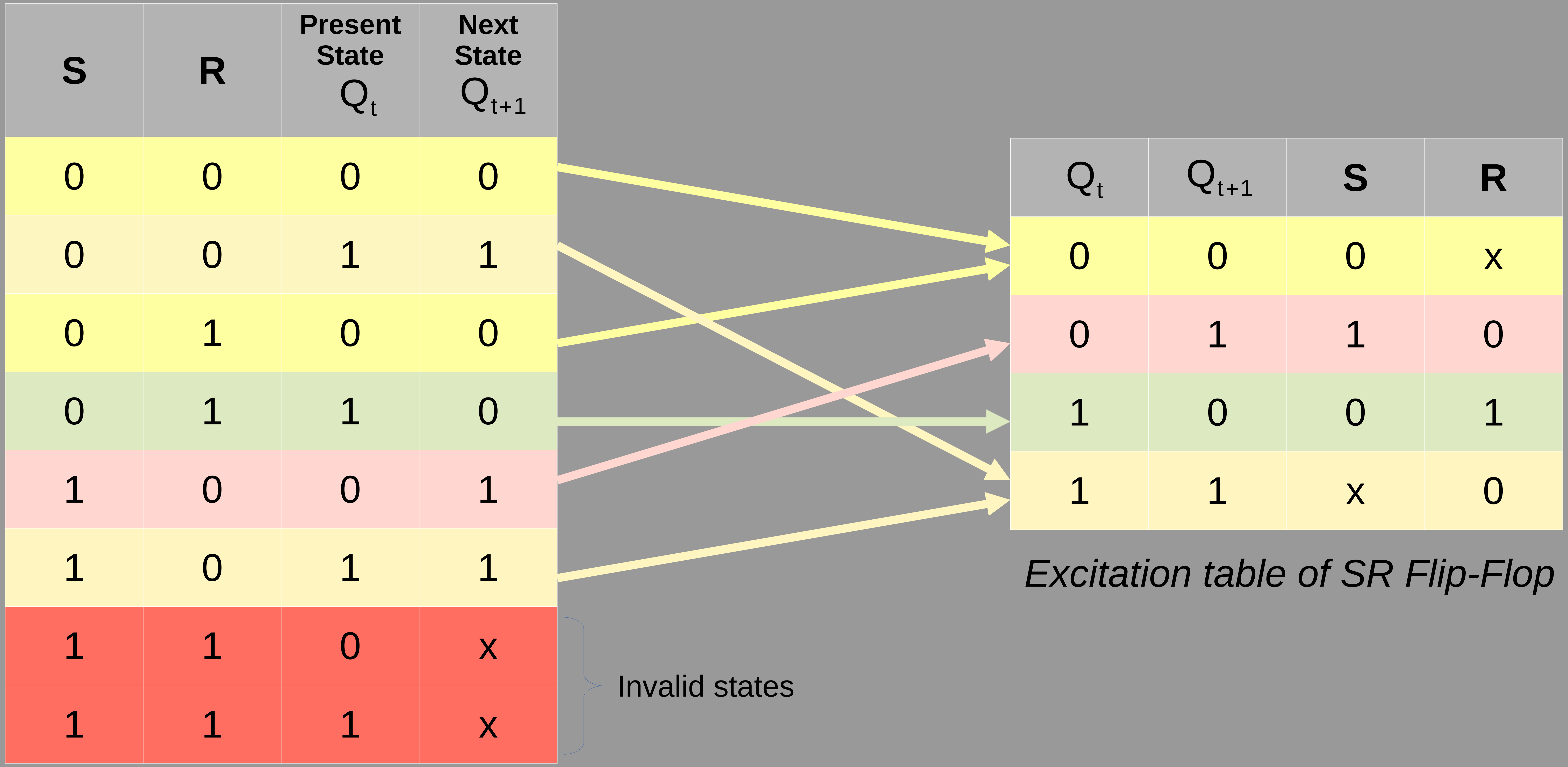
Now, let us look at the excitation table for each flip flops.

SR flip flop

The excitation table of SR flip flop can be constructed from the information available in the truth table. In the diagram shown below, the first table shows the truth table, from which the excitation table is derived.

From the truth table, you can observe that when the present state is Q_t = 0, the next state becomes Q_{t+1} = 0 for two input values S = 0, R = 0 and S = 0, R = 1. (It is shown in first and third rows with yellow color)

From this we can say that, for the state transition from Q_t = 0 to Q_{t+1} = 0, the excitation inputs required are S = 0 and R = 0 or 1. It is filled in the first row(Yellow color) of the excitation table. Since R has two values(0 and 1), it is denoted as don’t care condition(x).



S	R	Present State Q_t	Next State Q_{t+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	x
1	1	1	x

Invalid states

Truth table of SR Flip-Flop

Q_t	Q_{t+1}	S	R
0	0	0	x
0	1	1	0
1	0	0	1
1	1	x	0

Excitation table of SR Flip-Flop

Similarly, when you observe the truth table, to obtain the next state output Q_{t+1}= 1 from the present state input Q_t = 0, the required SR inputs are S = 1 and R = 0(shown in 5th row as pink color). Thus for state transition from 0 to 1, the excitation inputs require are S = 1 and R = 0. It is filled in the second row of the excitation table.

The state transition from present state Q_t = 1 to the next state Q_{t+1} = 0 happens only when the inputs are S = 0 and R = 1(observed from 4th row in light green color). It is filled in the third-row of the excitation table.

In the same way, the state transition from Q_t = 1 to Q_{t+1} = 1 happens at S = 0, R = 0 and S = 1, R = 0(shown in second and sixth row of the truth table). It is filled in the fourth row of the excitation table as Q_t= 1, Q_{t+1} = 1 and S = x, R = 0. Here x denotes the don't care condition, as it has two values(0 and 1).

D flip flop

The excitation table of D flip flop is derived from its truth table. The excitation table is constructed in the same way as explained for SR flip flop.

Here, when you observe from the truth table shown below, the next state output is equal to the D input. So it is very simple to construct the excitation table.

For the state transition from Q_t = 0 to Q_{t+1} = 0, the required excitation input is D = 0, regardless of Q_t value. For transition of states from Q_t = 0 to Q_{t+1} = 1, the input required to excite is D = 1.

The state transit from Q_t = 1 to Q_{t+1} = 0 for the input D = 0. For the input D = 1, the state transition takes place from Q_t = 1 to Q_{t+1} = 1.

All the above-mentioned state transitions for D flip flop from the present state(Q_t) to the next state(Q_{t+1}) for the corresponding excitation inputs are filled in the table to get the excitation table.

D	Present State Q_t	Next State Q_{t+1}
0	0	0
0	1	0
1	0	1
1	1	1

Truth table

Q_t	Q_{t+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

Excitation table

of D Flip-Flop

JK flip flop

For JK flip flop, the excitation table is derived in the same way. From the truth table, for the present state and next state values Q_t = 0 and Q_{t+1} = 0(indicated in the first and third row with yellow color), the inputs are J = 0 and K = 0 or 1. Since K input has two values, it is considered as don’t care condition(x).

Thus the state transition from Q_t = 0 to Q_{t+1} = 0 takes place when J = 0, K = x. It is filled in the first row of the excitation table.

The state transition from present state $Q_t = 0$ to the next state $Q_{t+1} = 1$ occur, when the inputs are either J = 1, K = 0 or J = 1, K = 1(indicated in the fifth and seventh row with pink color). Thus the excitation table is filled with datas $Q_t = 0$, $Q_{t+1} = 1$, J = 1 and K = x.

Similarly, for the transition of the state from 1 to 0, the inputs are J = 0, K = 1 or J = 1, K = 1(indicated in the fourth and eighth row with ash color). So for this transition, the required inputs are J = x and K =1, as the value of J can be either 0 or 1.

For the state transition from $Q_t = 1$ to $Q_{t+1} = 1$, the J input can be 0 or 1 but the K input remains at o(indicated in the second and sixth row with violet color). For this transition to occur, the excitation inputs are J = x and K = 0.

J	**K**	**Present State** Q_t	**Next State** Q_{t+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Truth table of JK Flip-Flop

Q_t	Q_{t+1}	**J**	**K**
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Excitation table of JK Flip-Flop

Sequential Circuits

A sequential circuit is an interconnection of flip-flops and gates. The gates by themselves constitute a combinational circuit, but when included with the flip-flops, the overall circuit is classified as a sequential circuit. The block diagram of a clocked sequential circuit is shown in Fig. 1-24. It consists of a combinational circuit and a number of clocked flip-flops. In general, any number or type of flip-flops may be included. As shown in the diagram, the combinational circuit block receives binary signals from external inputs and from the outputs of flip-flops. The outputs of the combinational circuit go to external outputs and to inputs of flip-flops. The gates in the combinational circuit determine the binary value to be stored in the flip-flops after each clock transition. The outputs of flip-flops, in turn, are applied to the combinational circuit inputs and determine the circuit's behavior. This process demonstrates that the external outputs of a sequential circuit are functions of both external inputs and the present state of the flip-flops. Moreover, the next state of flip-flops is also a function of their present state and external inputs. Thus a sequential circuit is specified by a time sequence of external inputs, external outputs, and internal flip-flop binary states.

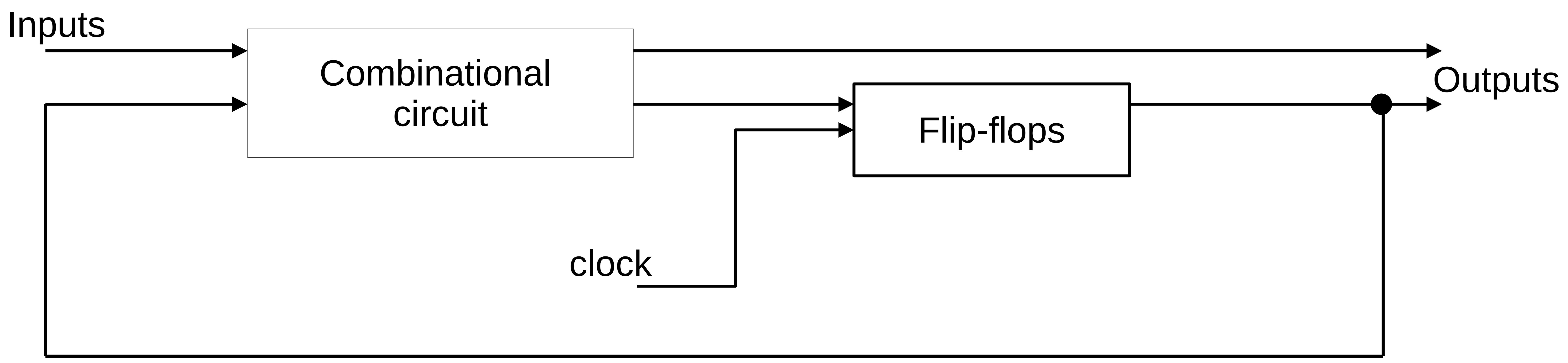
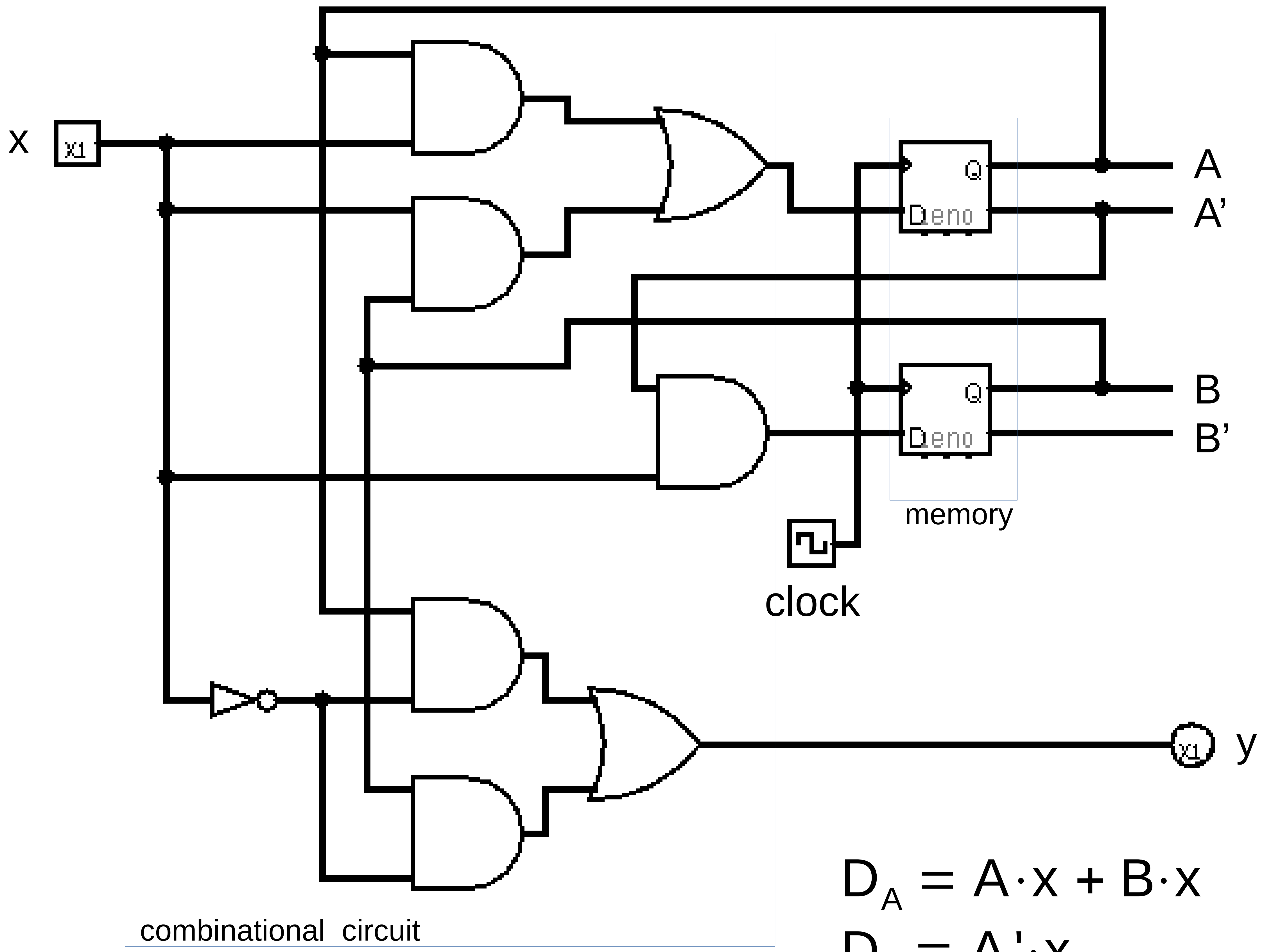


Figure 1 – 24. Block diagram of a clocked synchronous sequential circuit.

Flip-Flop Input Equations



$$D_A = A\cdot x + B\cdot x$$

$$D_B = A'\cdot x$$

$$y = A\cdot x' + B\cdot x'$$

Figure 1-25. Example of a sequential circuit

An example of a sequential circuit is shown in Fig. 1-25. It has one input variable x, one output variable y, and two clocked D flip-flops. The AND gates, OR gates, and inverter form the combinational logic part of the circuit. The interconnections among the gates in the combinational circuit can be specified by a set of Boolean expressions. The part of the combinational circuit that generates the inputs to flip-flops are described by a set of Boolean expressions called flip-flop input equations. We adopt the convention of using the flip-flop input symbol to denote the input equation variable name and a subscript to designate the symbol chosen for the output of the flip-flop. Thus, in Fig. 1-25, we have two input equations, designated D_A and D_B. The first letter in each symbol denotes the D input of a D flip-flop. The subscript letter is the symbol name of the flip-flop. The input equations are Boolean functions for flip-flop input variables and can be derived by inspection of the circuit.

Since the output of the OR gate is connected to the D input of flip-flop A, we write the first input equation as

$$D_A = A \cdot x + B \cdot x$$

where A and B are the outputs of the two flip-flops and x is the external input. The second input equation is derived from the single AND gate whose output is connected to the D input of flip-flop B:

$$D_B = A' \cdot x$$

The sequential circuit also has an external output, which is a function of the input variable and the state of the flip-flops. This output can be specified algebraically by the expression

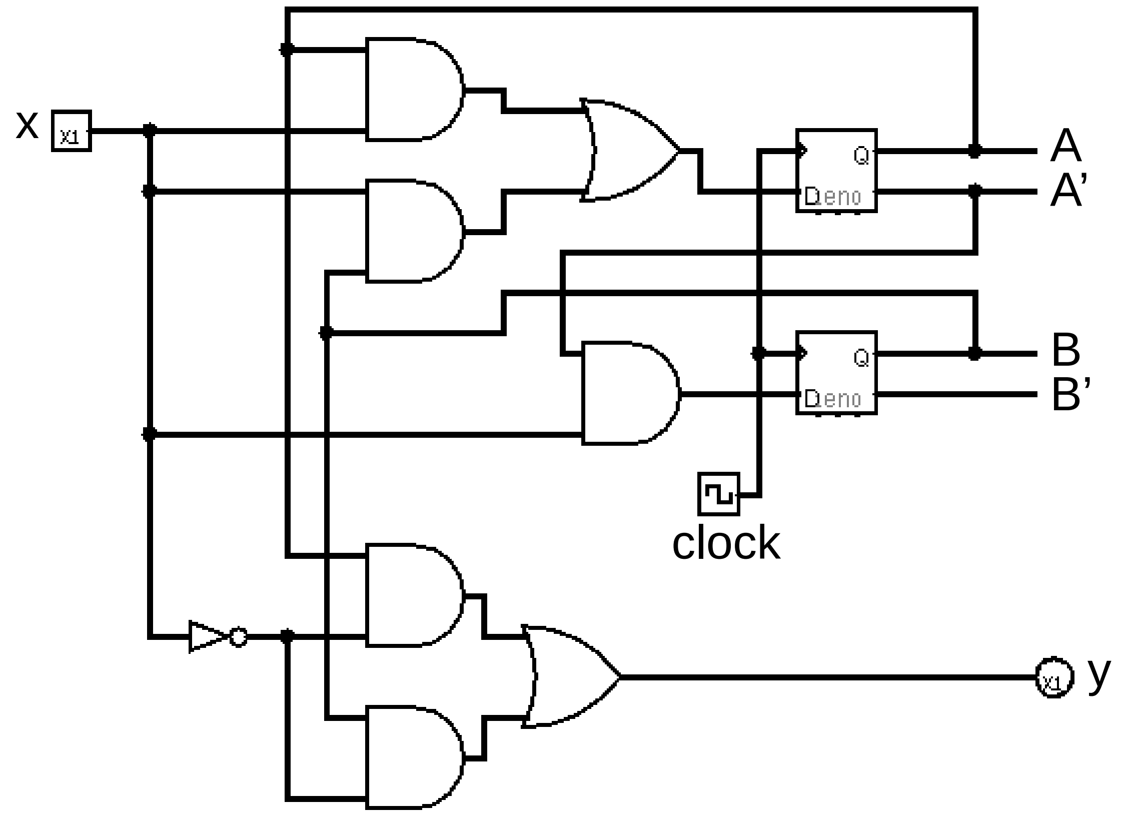
$$y = A \cdot x' + B \cdot x'$$

From this example we note that a flip-flop input equation is a Boolean expression for a combinational circuit. The subscripted variable is a binary variable name for the output of a combinational circuit. This output is always connected to a flip-flop input.

State Table

Present State		Input	Next State		Output
A	B	x	A	B	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

Table 1-4.
State table for circuit in figure 1-25



$$D_A = A \cdot x + B \cdot x$$
$$D_B = A' \cdot x$$
$$y = A \cdot x' + B \cdot x'$$

The behavior of a sequential circuit is determined from the inputs, the outputs, and the state of its flip-flops. Both the outputs and the next state are a function of the inputs and the present state. A sequential circuit is specified by a state table that relates outputs and next states as a function of inputs and present states. In clocked sequential circuits, the transition from present state to next state is activated by the presence of a clock signal.

The state table for the circuit of Fig. 1-25 is shown in Table 1-4. The table consists of four sections, labeled present state, input, next state, and output . The present-state section shows the states of flip-flops A and B at any given time t. The input section gives a value of x for each possible present state. The next-state section shows the states of the flip-flops one clock period later at time t + 1. The output section gives the value of y for each present state and input condition.

The derivation of a state table consists of first listing all possible binary combinations of present state and inputs. In this case we have eight binary combinations from 000 to 111. The next-state values are then determined from the logic diagram or from the input equations. The input equation for flip-flop A is

$$D_A = A \cdot x + B \cdot x$$

The next-state value of a each flip-flop is equal to its D input value in the present state. The transition from present state to next state occurs after application of a clock signal. Therefore, the next state of A is equal to 1 when the present state and input values satisfy the conditions Ax = 1 or Bx = 1, which makes D_A equal 1. This is shown in the state table with three 1's under the column for next state of A. Similarly, the input equation for flip-flop B is

$$D_B = A'\cdot x$$

The next state of B in the state table is equal to 1 when the present state of A is 0 and input x is equal to 1. The output column is derived from the output equation

$$y = A\cdot x' + B\cdot x'$$

The state table of any sequential circuit is obtained by the procedure used in this example. In general, a sequential circuit with m flip-flops, n input variables, and p output variables will contain m columns for present state, n columns for inputs, m columns for next state, and p columns for outputs. The present state and input columns are combined and under them we list the 2^{m+n} binary combinations from 0 through $2^{m+n} - 1$. The next-state and output columns are functions of the present state and input values and are derived directly from the circuit or the Boolean equations that describe the circuit.

State Diagram

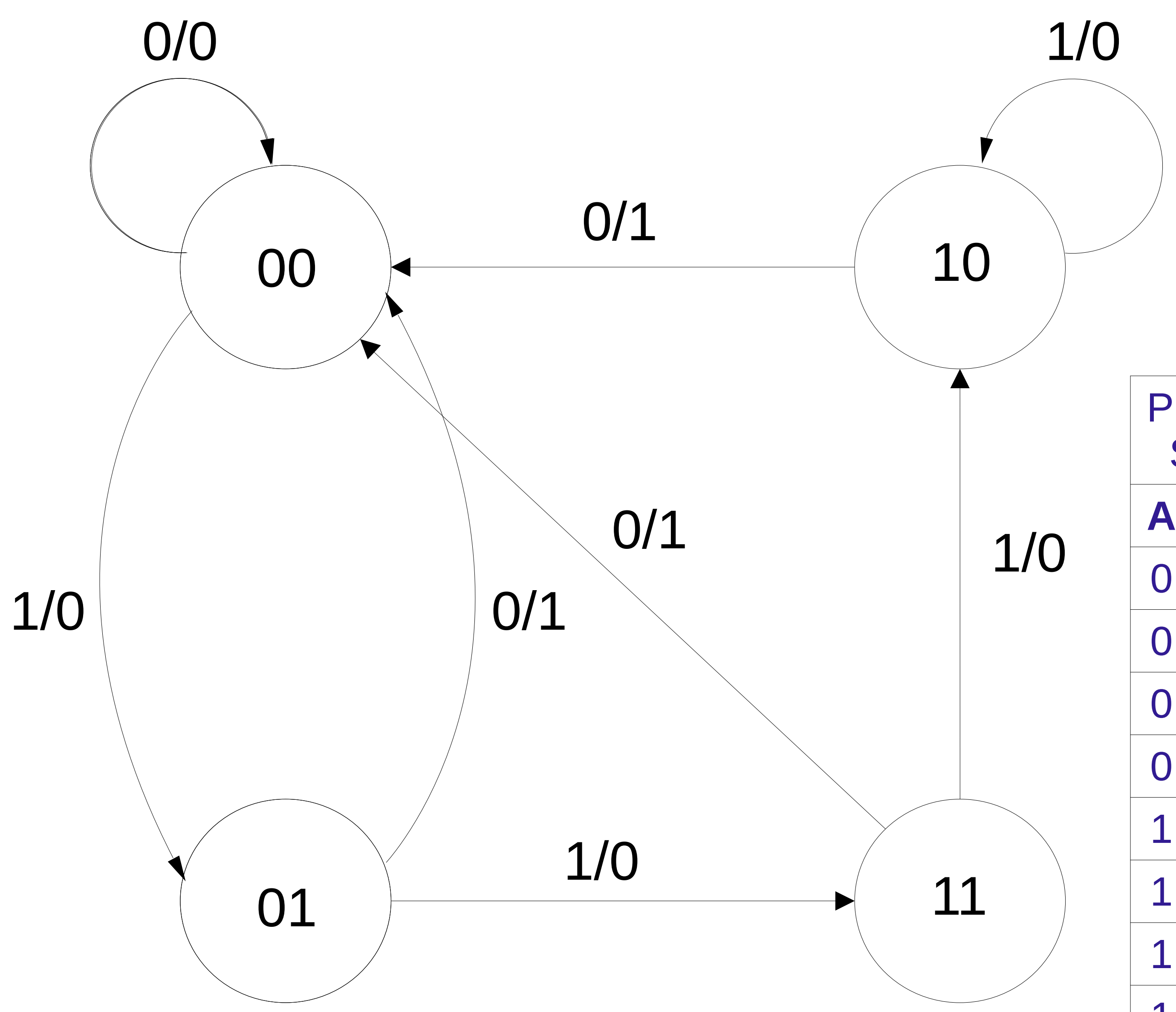


Figure 1-26
State diagrams
Of sequential circuit

Present State		Input	Next State		Output
A	**B**	**x**	**A**	**B**	**y**
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

The information available in a state table can be represented graphically in a state diagram. In this type of diagram, a state is represented by a circle, and the transition between states is indicated by directed lines connecting the circles. The state diagram of the sequential circuit of Fig. 1-25 is shown in Fig. 1-26. The state diagram provides the same information as the state table and is obtained directly from Table 1-4. The binary number inside each circle identifies the state of the flip-flops. The directed lines are labeled with two binary numbers separated by a slash. The input value during the present state is labeled first and the number after the slash gives the output during the present state. For example, the directed line from state 00 to 01 is labeled 1/0, meaning that when the sequential circuit is in the present state 00 and the input is 1, the output is 0. After a clock transition, the circuit goes to the next state 01 . The same clock transition may change the input value. If the input changes to 0, the output becomes 1, but if the input remains at 1, the output stays at 0. This information is obtained from the state diagram along the two directed lines emanating from the circle representing state 01. A directed line connecting a circle with itself indicates that no change of state occurs.

There is no difference between a state table and a state diagram except in the manner of representation. The state table is easier to derive from a given logic diagram and the state diagram follows directly from the state table. The state diagram gives a pictorial view of state transitions and is the form suitable for human interpretation of the circuit operation. For example, the state diagram of Fig. 1-26 clearly shows that starting from state 00, the output is 0 as long as the input stays at 1. The first 0 input after a string of 1's gives an output of 1 and transfers the circuit back to the initial state 00.

Design Example

The procedure for designing sequential circuits will be demonstrated by a specific example. The design procedure consists of first translating the circuit specifications into a state diagram. The state diagram is then converted into a state table. From the state table we obtain the information for obtaining the logic circuit diagram.

We wish to design a clocked sequential circuit that goes through a sequence of repeated binary states 00, 01, 10, and 11 when an external input x is equal to 1. The state of the circuit remains unchanged when x = 0. This type of circuit is called a 2-bit binary counter because the state sequence is identical to the count sequence of two binary digits. Input x is the control variable that specifies when the count should proceed.

The binary counter needs two flip-flops to represent the two bits. The state diagram for the sequential circuit is shown in Fig. 1-27. The diagram is drawn to show that the states of the circuit follow the binary count as long as x = 1. The state following 11 is 00, which causes the count to be repeated. If x = 0, the state of the circuit remains unchanged. This sequential circuit has no external outputs, and therefore only the input value is labeled in the diagram. The state of the flip-flops is considered as the outputs of the counter.

We have already assigned the symbol x to the input variable. We now assign the symbols A and B to the two flip-flop outputs. The next state of A and B, as a function of the present state and input x, can be transferred from the state diagram into a state table. The first five columns of Table 1-5 constitute the state table. The entries for this table are obtained directly from the state diagram.

The excitation table of a sequential circuit is an extension of the state table. This extension consists of a list of flip-flop input excitations that will cause the required state transitions. The flip-flop input conditions are a function of the type of flip-flop used. If we employ JK flip-flops, we need columns for the J and K inputs of each flip-flop. We denote the inputs of flip-flop A by J_A and K_A, and those of flip-flop B by J_B and K_B.

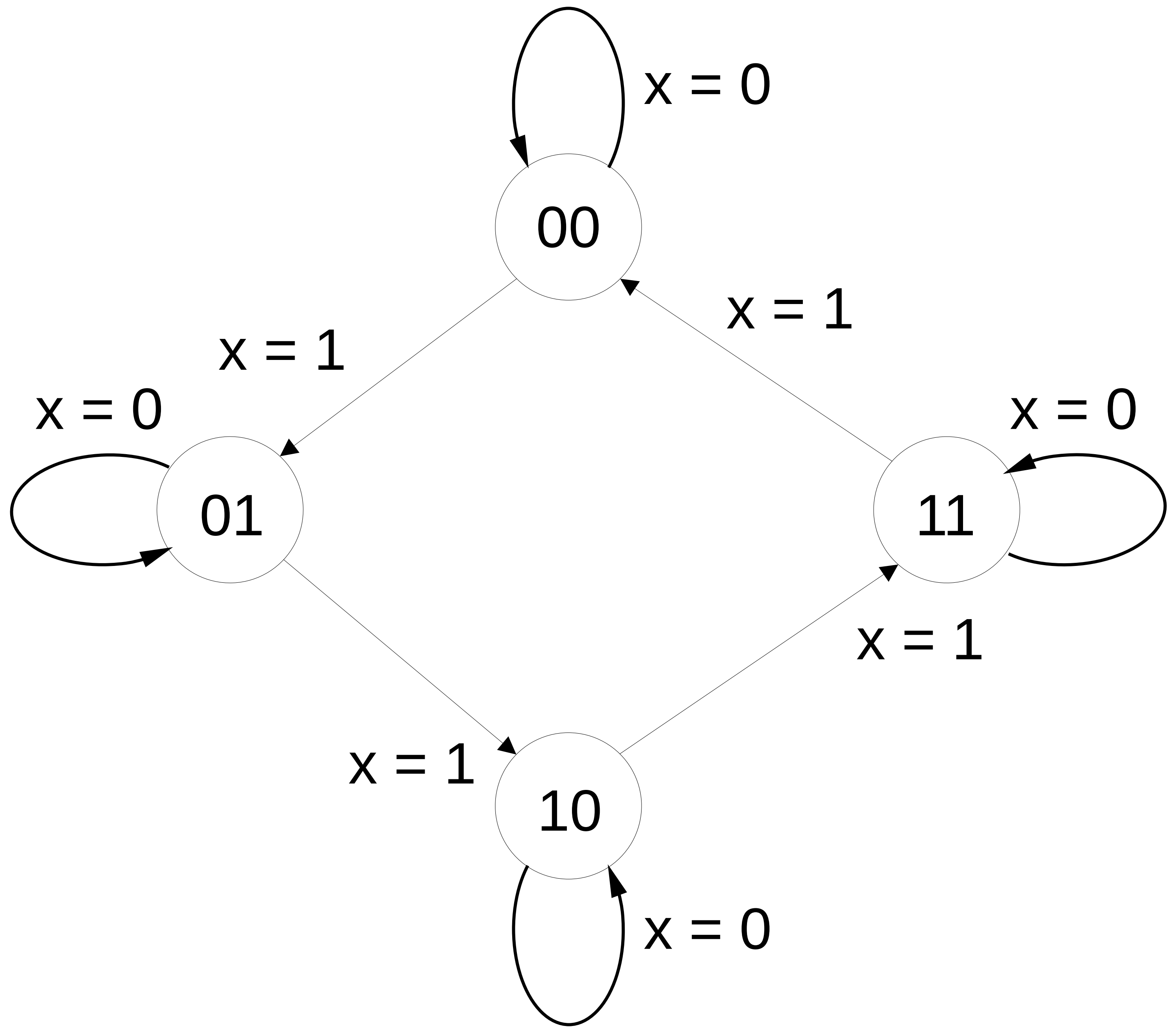


Figure 1-27
State diagram for binary counter

Present State		Input	Next State		Flip-Flop inputs			
A	B	x	A	B	J_A	K_A	J_B	K_B
0	0	0	0	0	0	x	0	x
0	0	1	0	1	0	x	1	x
0	1	0	0	1	0	x	x	0
0	1	1	1	0	1	x	x	1
1	0	0	1	0	x	0	0	x
1	0	1	1	1	x	0	1	x
1	1	0	1	1	x	0	x	0
1	1	1	0	0	x	1	x	1

state table

Table 1-5
Excitation table for binary counter

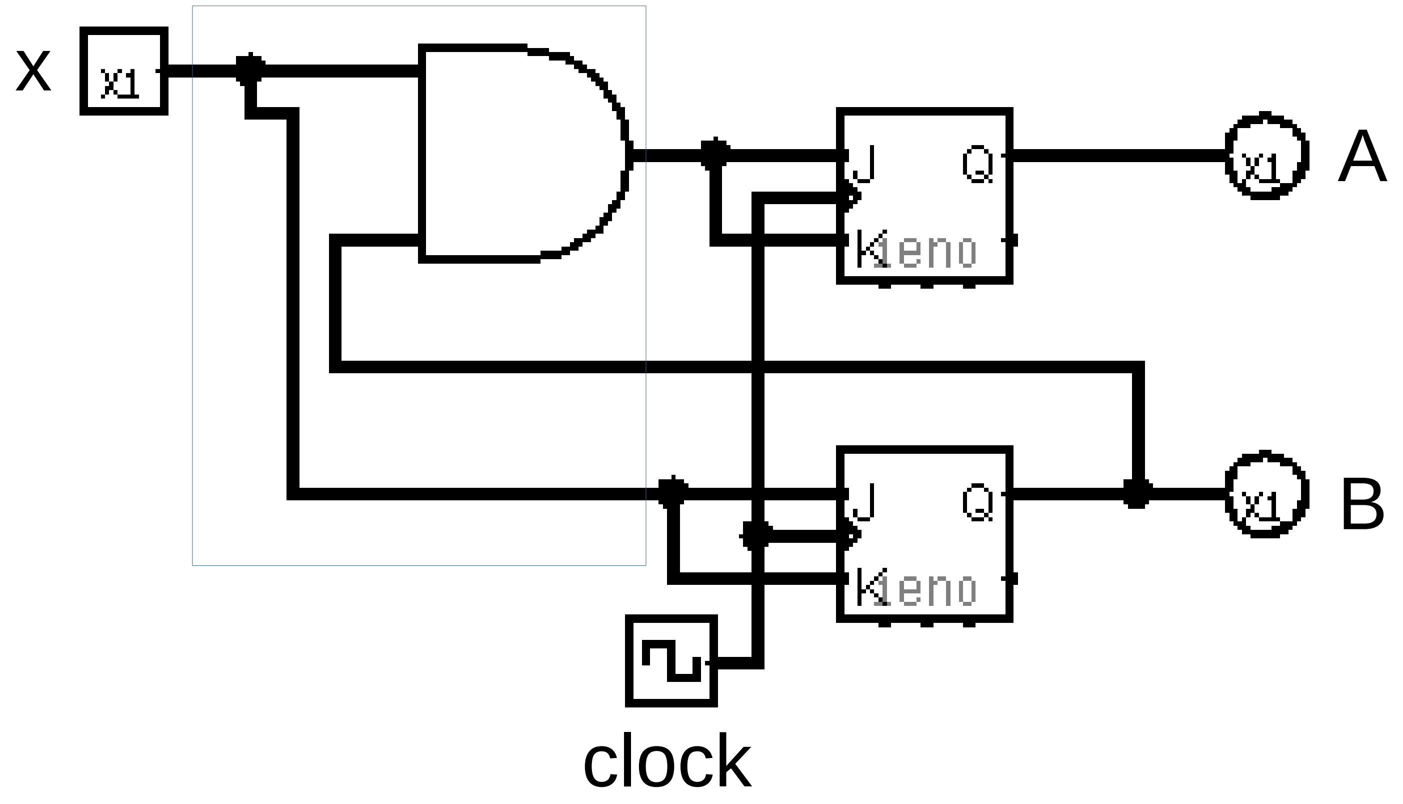


Figure 1-29.
Logic diagram of a 2-bit binary counter

	Present State		Input	Next State		Flip-Flop inputs			
	A	**B**	***x***	**A**	**B**	J_A	K_A	J_B	K_B
0	0	0	0	0	0	0	x	0	x
1	0	0	1	0	1	0	x	1	x
2	0	1	0	0	1	0	x	x	0
3	0	1	1	1	0	1	x	x	1
4	1	0	0	1	0	x	0	0	x
5	1	0	1	1	1	x	0	1	x
6	1	1	0	1	1	x	0	x	0
7	1	1	1	0	0	x	1	x	1

Excitation table for binary counter

deriviation

JK flip-flop			
Q_t	Q_{t+1}	J	K
0	0	**0**	**x**
0	1	**1**	**x**
1	0	**x**	**1**
1	1	**x**	**0**

Excitation table of JK flip-flop

A \ Bx	B'x'	B'x	Bx	Bx'
A'	0_{0}	0_{1}	1_{3}	0_{2}
A	x_{4}	x_{5}	x_{7}	x_{6}

$J_A = B\cdot x$

A \ Bx	B'x'	B'x	Bx	Bx'
A'	${}_{0}$	1_{1}	x_{3}	x_{2}
A	${}_{4}$	1_{5}	x_{7}	x_{6}

$J_b = x$

A \ Bx	B'x'	B'x	Bx	Bx'
A'	x_{0}	x_{1}	x_{3}	x_{2}
A	${}_{4}$	${}_{5}$	1_{7}	${}_{6}$

$K_A = B\cdot x$

A \ Bx	B'x'	B'x	Bx	Bx'
A'	x_{0}	x_{1}	1_{3}	${}_{2}$
A	x_{4}	x_{5}	1_{7}	${}_{6}$

$K_b = x$

The excitation table for the JK flip-flop specified in Table 1-3 is now used to derive the excitation table of the sequential circuit. For example, in the first row of Table 1-5, we have a transition for flip-flop A from 0 in the present state to 0 in the next state. In Table 1-3 we find that a transition of states from Q(t) = 0 to Q(t + 1) = 0 in a JK flip-flop requires that input J = 0 and input K = x . So 0 and x are copied in the first row under J_A and K_A, respectively. Since the first row also shows a transition for flip-flop B from 0 in the present state to 0 in the next state, 0 and x are copied in the first row under J_B and K_B. The second row of Table 1-5 shows a transition for flip-flop B from 0 in the present state to 1 in the next state. From Table 1-3 we find that a transition from Q(t) = 0 to Q(t + 1) = 1 requires that input J = 1 and input K = x . So 1 and x are copied in the second row under J_B and K_B, respectively. This process is continued for each row of the table and for each flip-flop, with the input conditions as specified in Table 1-3 being copied into the proper row of the particular flip-flop being considered.

Let us now consider the information available in an excitation table such as Table 1-5. We know that a sequential circuit consists of a number of flip-flops and a combinational circuit. From the block diagram of Fig. 1-24, we note that the outputs of the combinational circuit must go to the four flip-flop inputs J_A, K_A ,J_B, and K_B. The inputs to the combinational circuit are the external input x and the present-state values of flip-flops A and B. Moreover, the Boolean functions that specify a combinational circuit are derived from a truth table that shows the input-output relationship of the circuit. The entries that list the combinational circuit inputs are specified under the "present state" and "input" columns in the excitation table. The combinational circuit outputs are specified under the "flip-flop inputs" columns. Thus an excitation table transforms a state diagram to a truth table needed for the design of the combinational circuit part of the sequential circuit.

The simplified Boolean functions for the combinational circuit can now be derived. The inputs are the variables A, B, and x. The outputs are the variables J_A, K_A ,J_B, and K_B.The information from the excitation table is transferred into the maps of Fig. 1-28, where the four simplified flip-flop input equations are derived:

$$J_A = Bx \qquad K_A = Bx$$
$$J_B = x \qquad K_B = x$$

The logic diagram is drawn in Fig. 1-29 and consists of two JK flip-flops and an AND gate. Note that inputs J and K determine the next state of the counter when a clock signal occurs. If both J and K are equal to 0, a clock signal will have no effect; that is, the state of the flip-flops will not change. Thus when $x = 0$, all four inputs of the flip-flops are equal to 0 and the state of the flip-flops remains unchanged even though clock pulses are applied continuously.

Design procedure

The design of sequential circuits follows the outline described in the preceding example. The behavior of the circuit is first formulated in a state diagram. The number of flip-flops needed for the circuit is determined from the number of bits listed within the circles of the state diagram. The number of inputs for the circuit is specified along the directed lines between the circles. We then assign letters to designate all flip-flops and input and output variables and proceed to obtain the state table.

For m flip-flops and n inputs, the state table will consist of m columns for the present state, n columns for the inputs, and m columns for the next state. The number of rows in the table will be up to 2^{m+n} , one row for each binary combination of present state and inputs. For each row we list the next state as specified by the state diagram. Next, the flip-flop type to be used in the circuit is chosen. The state table is then extended into an excitation table by including columns for each input of each flip-flop. The excitation table for the type of flip-flop in use can be found in Table 1-3. From the information available in this table and by inspecting present state-to-next state transitions in the state table, we obtain the information for the flop-flop input conditions in the excitation table.

The truth table for the combinational circuit part of the sequential circuit is available in the excitation table. The present-state and input columns constitute the inputs in the truth table. The flip-flop input conditions constitute the outputs in the truth table. By means of map simplification we obtain a set of flip-flop input equations for the combinational circuit. Each flip-flop input equation specifies a logic diagram whose output must be connected to one of the flip-flop inputs. The combinational circuit so obtained, together with the flip-flops, constitutes the sequential circuit.

The outputs of flip-flops are often considered to be part of the outputs of the sequential circuit. However, the combinational circuit may also contain external outputs. In such a case the Boolean functions for the external outputs are derived from the state table by combinational circuit design techniques.

A set of flip-flop input equations specifies a sequential circuit in algebraic form. The procedure for obtaining the logic diagram from a set of flip-flop input equations is a straightforward process. First draw the flip-flops and label all their inputs and outputs. Then draw the combinational circuit from the Boolean expressions given by the flip-flop input equations. Finally, connect outputs of flip-flops to inputs in the combinational circuit and outputs of the combinational circuit to flip-flop inputs.

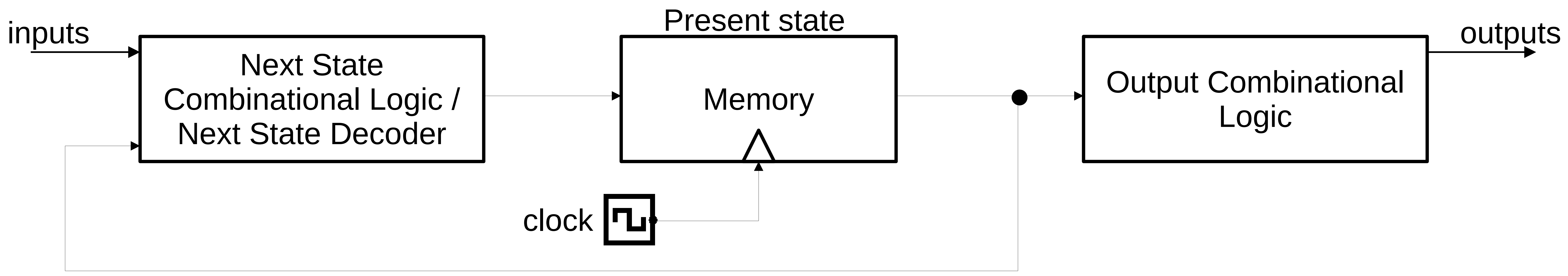
Mealy and Moore State Machines

There are two theoretical circuit models to representing synchronous sequential circuits.

The difference between those two models is how the output is generated.

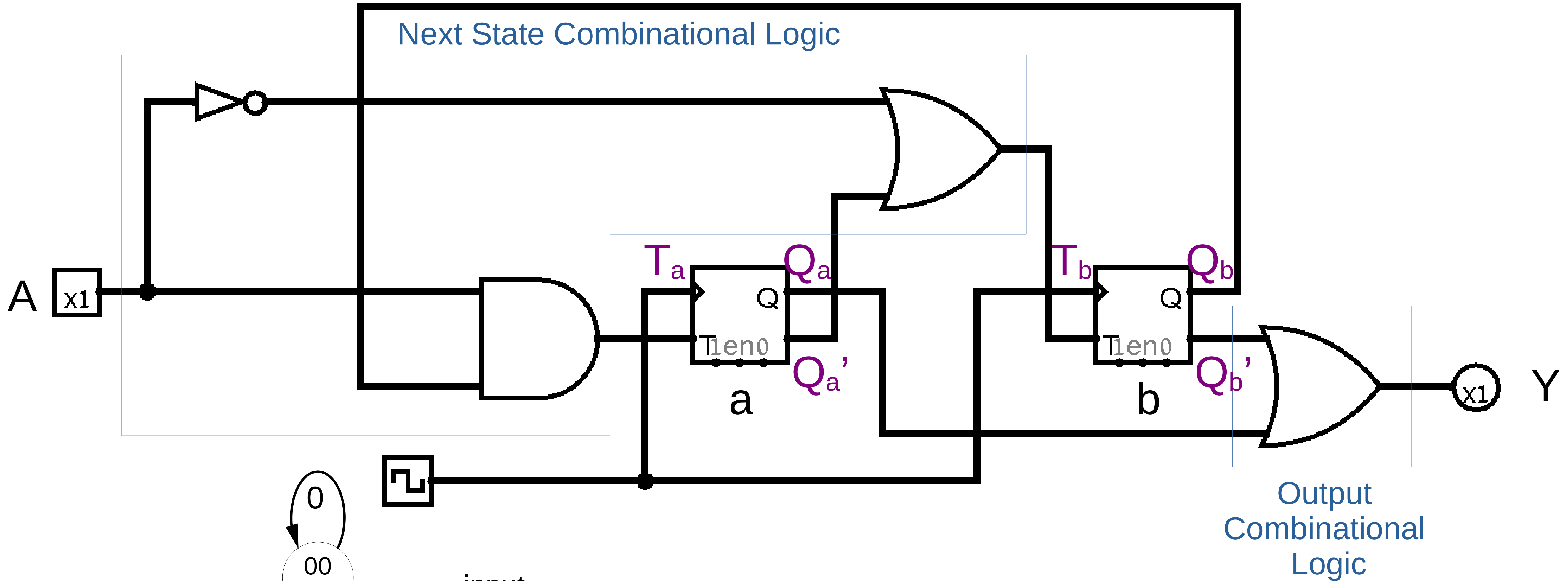
In the Moore circuit, the output is derived only from the present state.
In the Mealy circuit, the output is derived from the present state, as well as the input.

1. Moore State machine


inputs
Next State Combinational Logic / Next State Decoder
Present state
Memory
clock
Output Combinational Logic
outputs

The figure above is the general schematic representation of the Moore circuit model. The state of the circuit is the output of the Next State generator, saved in the memory. When the circuit is active, its present state, which was generated by the Next State Combinational Logic, is based on the external input as well as the previous state. On the other hand, the final output of the circuit is determined only by the present state and derived from it by the Output Conminational Logic. It's completely dependent only on the present state and not determined by the external input, at least directly.

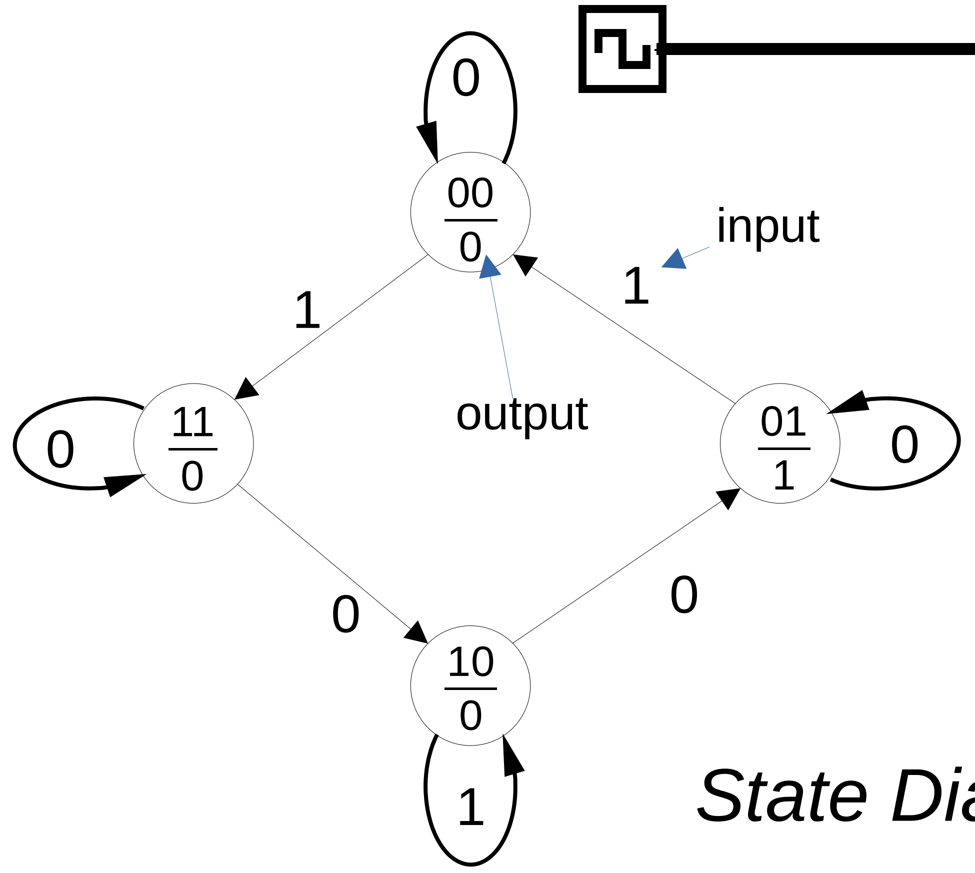
There is a sequential circuit. let's determine if this circuit is built according to Moore or Mealy model. First of all, let's determine what the Next State logic and Output combinational logic parts of the circuit are. We know that flip-flops are memory elements. so they can belong only to the memory part. So, the Next State logic is the combinational part of the circuit that accepts the input signal and produces the feed for the memory. On the contrary, the Output logic is the combinational part that accepts the circuit's state and produces the final output. So, th gates arrangement that accepts the external input and feeds its output to the memory is the next State logic part and the OR gare that accepts the state from the memory and produces the output Y is the Output logic part.



We know that in the Moore model the output is dependent only on the state and not on the inputs, so let's find the inputs to the Output logic, from which the final output is about to be derived:

$$Y = Qb' + Qa$$

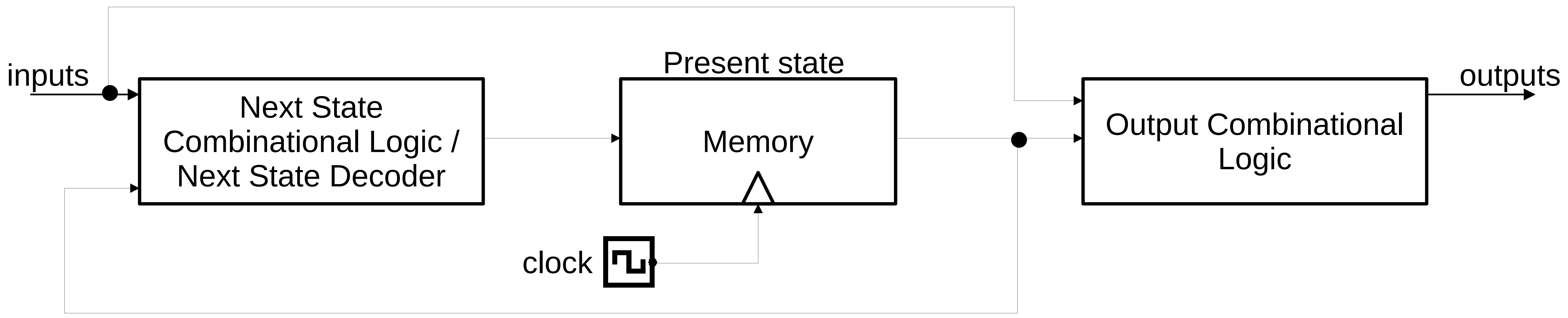
This equation tells us that the external input A doesn't constitute any part of the Output Logic input, so Y is entirely independent of A, and derived only from the memory output (the circuit's state).
So based on this we can conclude that this circuit is built according to the Moore state machine model.



State Diagram of a Moore Machine

2. Mealy State machine

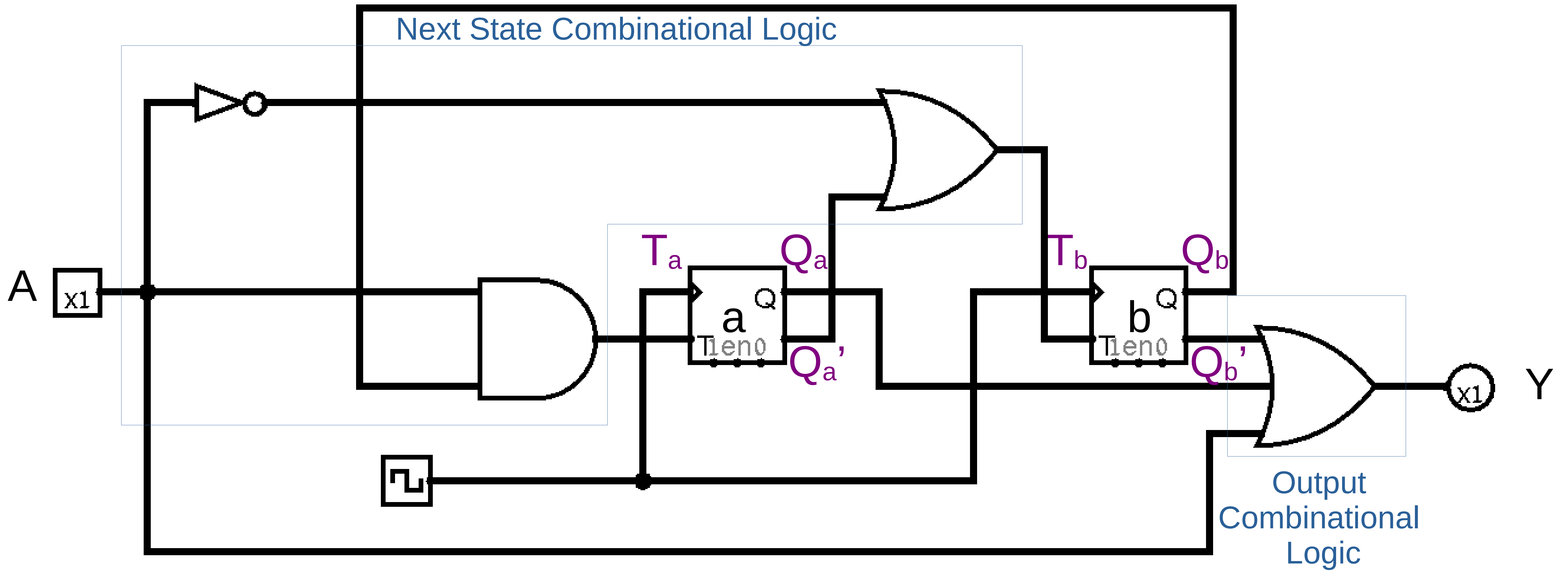
The output is the function of the present state as well as the input


inputs
Next State Combinational Logic / Next State Decoder
Present state
Memory
clock
Output Combinational Logic
outputs

IN this circuit, the output value Y is derived from the circuits state (the value stored in the second flip-flops, Q_a and Q_b'), but also from the input A:

$$Y = Q_b' + Q_a + A$$

So we can say that this circuit is built according to the Mealy model (it is a kind of Mealy state machine).



Difference between Mealy Machine & Moore Machine	
Mealy Machine	**Moore Machine**
1. The output of the circuit may be affected by changes in the information.	1. The output of the circuit is unaffected by changes in the input.
2. It requires fewer number of states for implementing same function.	2. For implementing the same function, it requires more number of states
3. The output is a function of the present state as well as present input	3. The output is a function of the present state only
4. The counter cannot be referred to as a Mealy Machine.	4. The counter is referred to as a Moore Machine.
5. The design of the Mealy model is complex.	5. The design of the Moore model is easy
6. Mealy machines react to changes more quickly. They all seem to react in the same way.	6. Decoding the output necessitates more logic, resulting in longer circuit delays. They usually react after one clock cycle
7. If the external inputs vary, the output can alter between clock edges.	7. Only when the active clock edge will the output alter.
8. The output is set on the transition.	8. The output is set to state.
9. It's easier to design with less hardware.	9. To design, more hardware is necessary

Integrated Circuits

Digital circuits are constructed with integrated circuits. An integrated circuit (abbreviated IC) is a small silicon semiconductor crystal. called a chip, containing the electronic components for the digital gates. The various gates are interconnected inside the chip to form the required circuit. The chip is mounted In a ceramic or plastic container, and connections are welded by thin gold wires to external pins to form the integrated circuit. The number of pins may range from 14 in a small IC package to 100 or more in a larger package. Each IC has a numeric designation printed on the surface of the package for identification. Each vendor publishes a data book or catalog that contains the exact description and all the necessary information about the ICs that it manufactures.

As the technology of ICs has improved, the number of gates that can be put in a single chip has increased considerably. The differentiation between those chips that have a few internal gates and those having hundreds or thousands of gates is made by a customary reference to a package as being either a small- medium-, or large-scale integration device.

Small-scale integration (SSI) devices contain several independent gates in a single package. The inputs and outputs of the gates are connected directly to the pins in the package. The number of gates is usually less than 10 and is limited by the number of pins available in the IC.

Medium-scale integration (MSI) devices have a complexity of approximately 10 to 200 gates in a single package. They usually perform specific elementary digital functions such as decoders, adders, and registers.

Large-scale integration (LSI) devices contain between 200 and a few thousand gates in a single package. They include digital systems, such as processors, memory chips, and programmable modules.

Very-large-scale integration (VLSI) devices contain thousands of gates TTL within a single package. Examples are large memory arrays and complex microcomputer chips. Because of their small size and low cost, VLSI devices have revolutionized the computer system design technology, giving designers the capability to create structures that previously were not economical.

Digital integrated circuits are classified not only by their logic operation but also by the specific circuit technology to which they belong. The circuit technology is referred to as a digital logic family . Each logic family has its own basic electronic circuit upon which more complex digital circuits and functions are developed. The basic circuit in each technology is either a NAND, a NOR, or an inverter gate. The electronic components that are employed in the construction of the basic circuit are usually used for the name of the technology. Many different logic families of integrated circuits have been introduced commercially. The following are the most popular.

- **TTL → Transistor-transistor logic**
- **ECL → Emitter-coupled logic**
- **MOS → Metal-oxide semiconductor**
- **CMOS → Complementary metal-oxide semiconductor**

TTL is a widespread logic family that has been in operation for many years and is considered as standard. ECL has an advantage in systems requiring high-speed operation. MOS is suitable for circuits that need high component density, and CMOS is preferable in systems requiring low power consumption.

The transistor-transistor logic family was an evolution of a previous technology that used diodes and transistors for the basic NAND gate. This technology was called DTL, for "diode-transistor logic." Later the diodes were replaced by transistors to improve the circuit operation and the name of the logic family was changed to "transistor-transistor logic." This is the reason for mentioning the word "transistor" twice. There are several variations of the TTL family besides the standard TTL, such as high-speed TTL, low-power TTL, Schottky TTL, low-power Schottky TTL, and advanced Schottky TTL. The power supply voltage for TTL circuits is 5 volts, and the two logic levels are approximately 0 and 3.5 volts.

The emitter-coupled logic (ECL) family provides the highest-speed digital circuits in integrated form. ECL is used in systems such as supercomputers and signal processors where high speed is essentiaL The transistors in ECL gates operate in a nonsaturated state, a condition that allows the achievement of propagation delays of 1 to 2 nanoseconds.

The metal-oxide semiconductor (MOS) is a unipolar transistor that depends on the flow of only one type of carrier, which may be electrons (n-channel) or holes (p-channel). This is in contrast to the bipolar transistor used in TIL and ECL gates, where both carriers exist during normal operation. A p-channel MOS is referred to as PMOS and an n-channel as NMOS. NMOS is the one that is commonly used in circuits with only one type of MOS transistor. The complementary MOS (CMOS) technology uses PMOS and NMOS transistors connected in a complementary fashion in all circuits. The most important advantages of CMOS over bipolar are the high packing density of circuits, a simpler processing technique during fabrication, and a more economical operation because of low power consumption.

Because of their many advantages, integrated circuits are used exclusively to provide various digital components needed in the design of computer systems. To understand the organization and design of digital computers it is very important to be familiar with the various components encountered in integrated circuits. For this reason, the most basic components are introduced in this chapter with an explanation of their logical properties. These components provide a catalog of elementary digital functional units commonly used as basic building blocks in the design of digital computers.

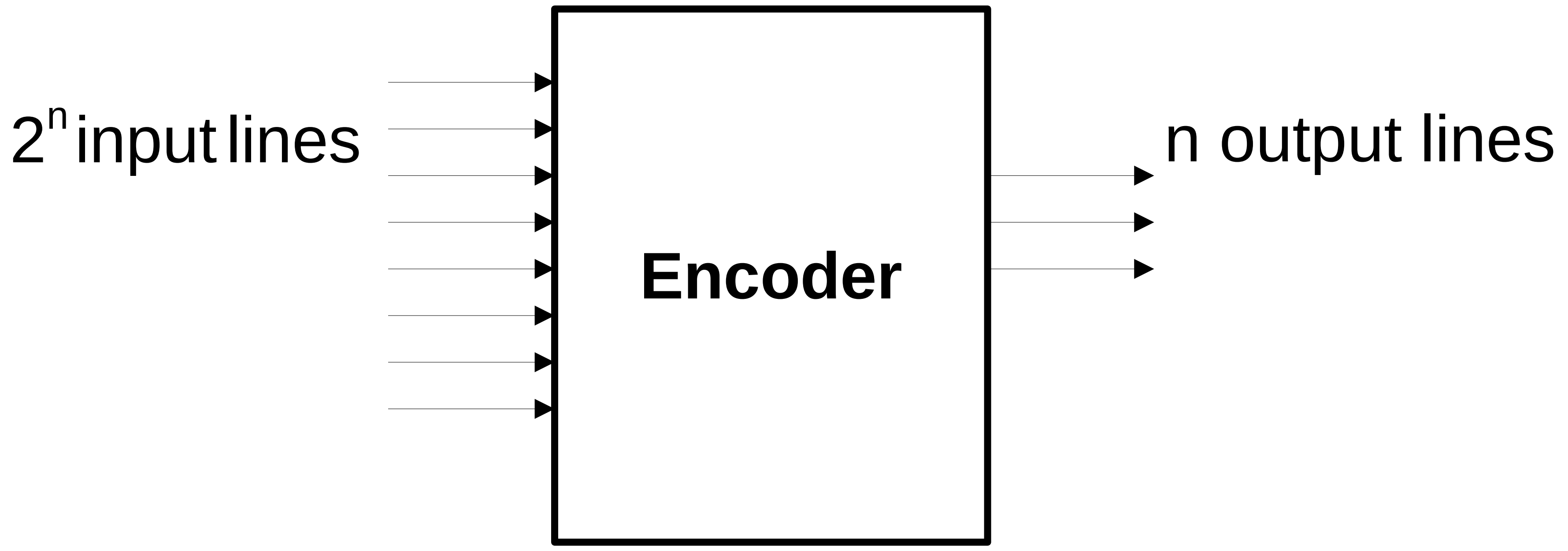
Encoders/Decoders

Difference between Encoder and Decoder

Combinational logic is the concept in which two or more input state define one or more output state. Encoder and Decoder are the combinational logic circuits. In which we implement combinational logic with the help of boolean algebra.

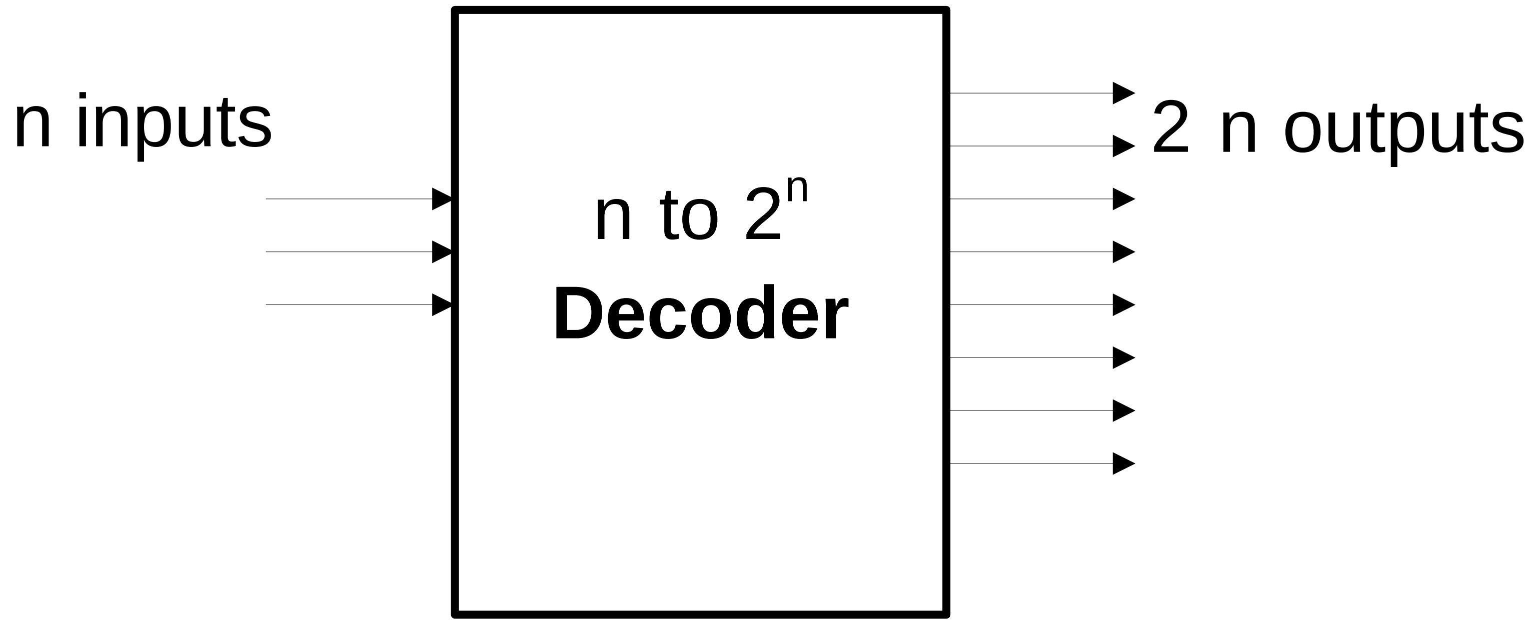
To encode something is to convert an unambiguous piece of information into a form of code that is not so clearly understood and the device which performs this operation is termed ad Encoder.

1. **Encoder**
   An Encoder is a device that converts the active data signal into a coded message format or it is a device that converts analogue signal to digital signals. It is a combinational circuit, that converts binary information in the form of a 2N input lines into N output lines which represent n bit code for the input. When an input signal is applied to an encoder then logic circuitry involved within it converts that particular input into coded binary output.



2. Decoder

A decoder is also a combinational circuit as encoder but its operation is exactly reverse as that of the encoder. A decoder is a device that generates the original signal as output from the coded input signal and converts n lines of input into 2n lines of output. An AND gate can be used as the basic decoding element because it produces a high output only when all inputs are high.



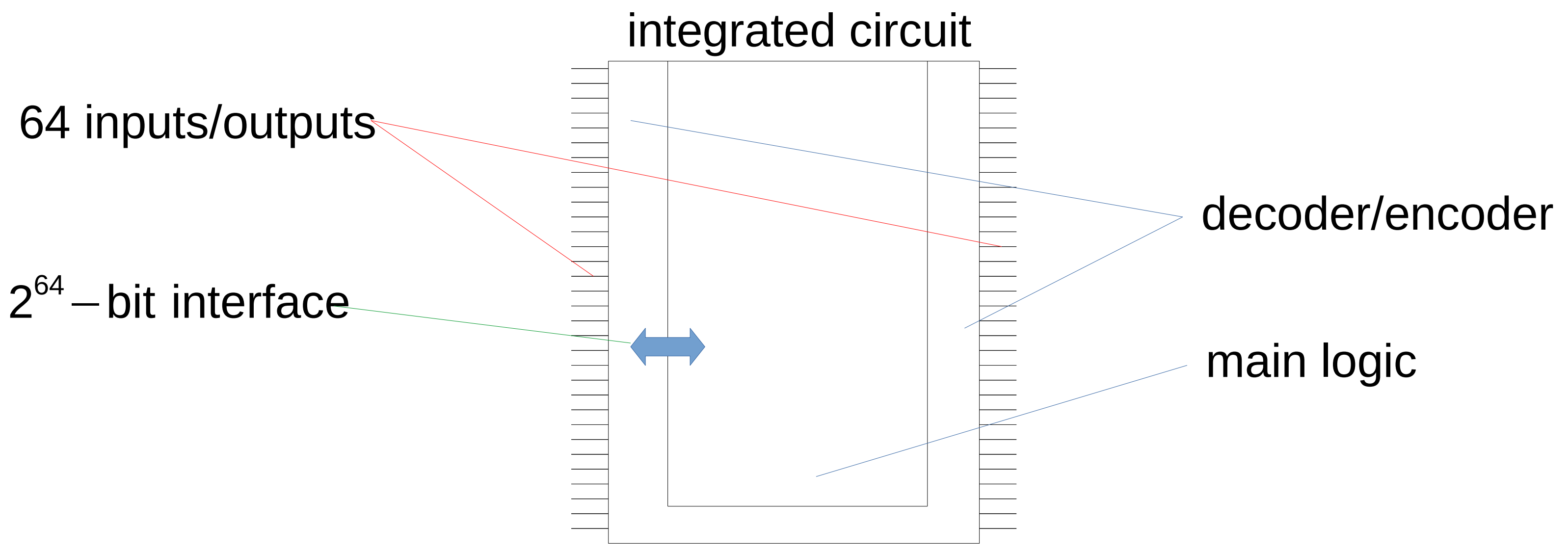
Difference between **encoder** and **decoder**:

	ENCODER	DECODER
1	Encoder circuit basically converts the applied information signal into .a coded digital bit stream	Decoder performs reverse operation and recovers the original information .signal from the coded bits
2	In case of encoder, the applied .signal is the active signal input	Decoder accepts coded binary data as .its input
3	The number of inputs accepted by an encoder is 2^n	The number of input accepted by .decoder is only n inputs
4	The output lines for an encoder is .n	The output lines of an decoder is 2^n
5	The encoder generates coded .data bits as its output	The decoder generates an active output signal in response to the coded .data bits
6	The operation performed is .simple	.The operation performed is complex
7	The encoder circuit is installed at .the transmitting end	The decoder circuit is installed at the .receiving side
8	OR gate is the basic logic .element used in it	AND gate along with NOT gate is the .basic logic element used in it
9	It is used in E-mail, video .encoders etc	It is used in microprocessors, memory .chips etc

Decoders

Discrete quantities of information are represented in digital computers with binary codes. A binary code of n bits is capable of representing up to 2^n distinct elements of the coded information. A decoder is a combinational circuit that converts binary information from the n coded inputs to a maximum of 2^n unique outputs. If the n-bit coded information has unused bit combinations, the decoder may have less than 2^n outputs.

The decoders presented in this section are called n-to-m-line decoders, where $m \leq 2^n$. Their purpose is to generate the 2^n (or fewer) binary combinations of the n input variables. A decoder has n inputs and m outputs and is also referred to as an n * m decoder.



Applications of Decoder

Some of the important applications of the decoder are as follows:

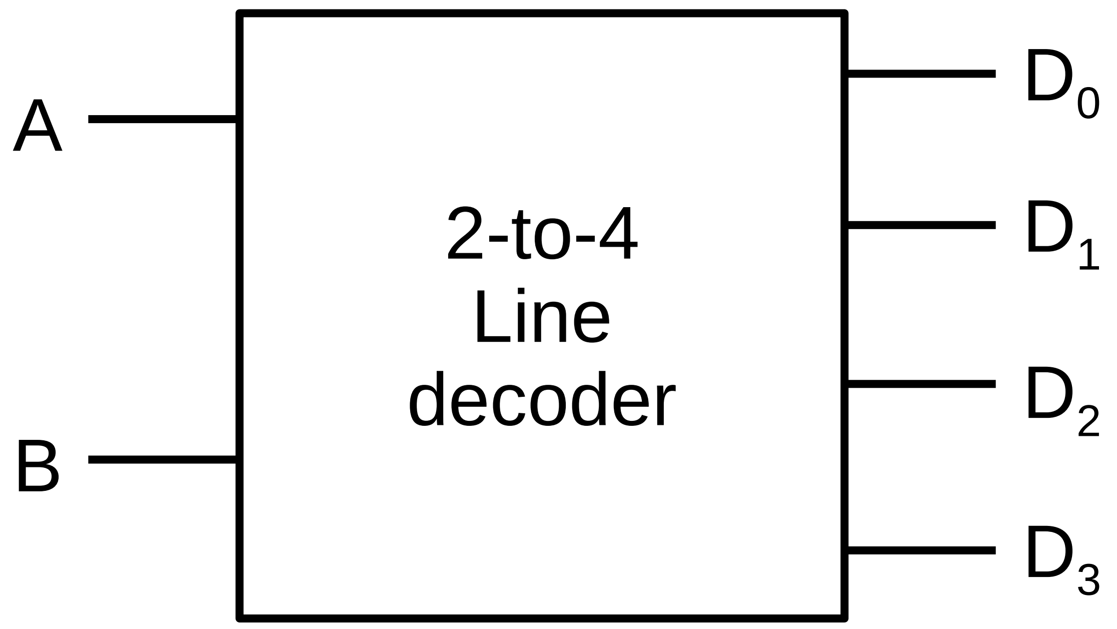
1. When the decoder inputs come from a counter which is being continually pulsed, the decoder outputs will be activated sequentially. Hence, they can be used as timing or sequencing signals to turn devices on or off at specific times.

2. Decoder is used in the memory system of a computer where they respond to the address code generated by the microprocessor to activate a particular memory location.

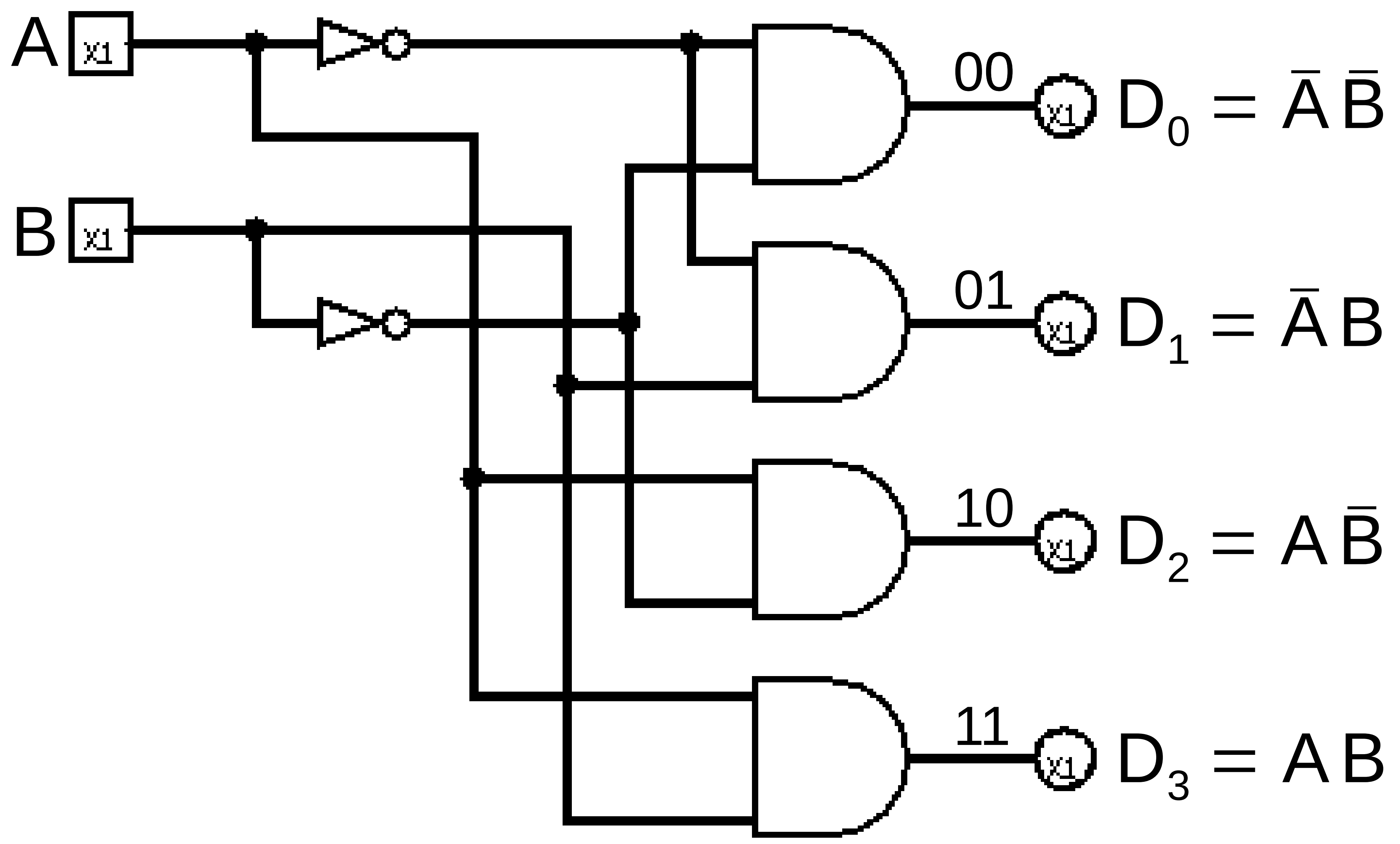
3. They are also used in computers for the selection of external devices that include printers, modems, scanners, internal disk drives, keyboard, video monitor, etc.

2 to 4 Line Decoder

Consider a 2 to 4-line decoder, where A and B are two inputs whereas Y0 through Y3 are the four outputs.



Inputs		Outputs			
A	B	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1



3 to 8 Line Decoder

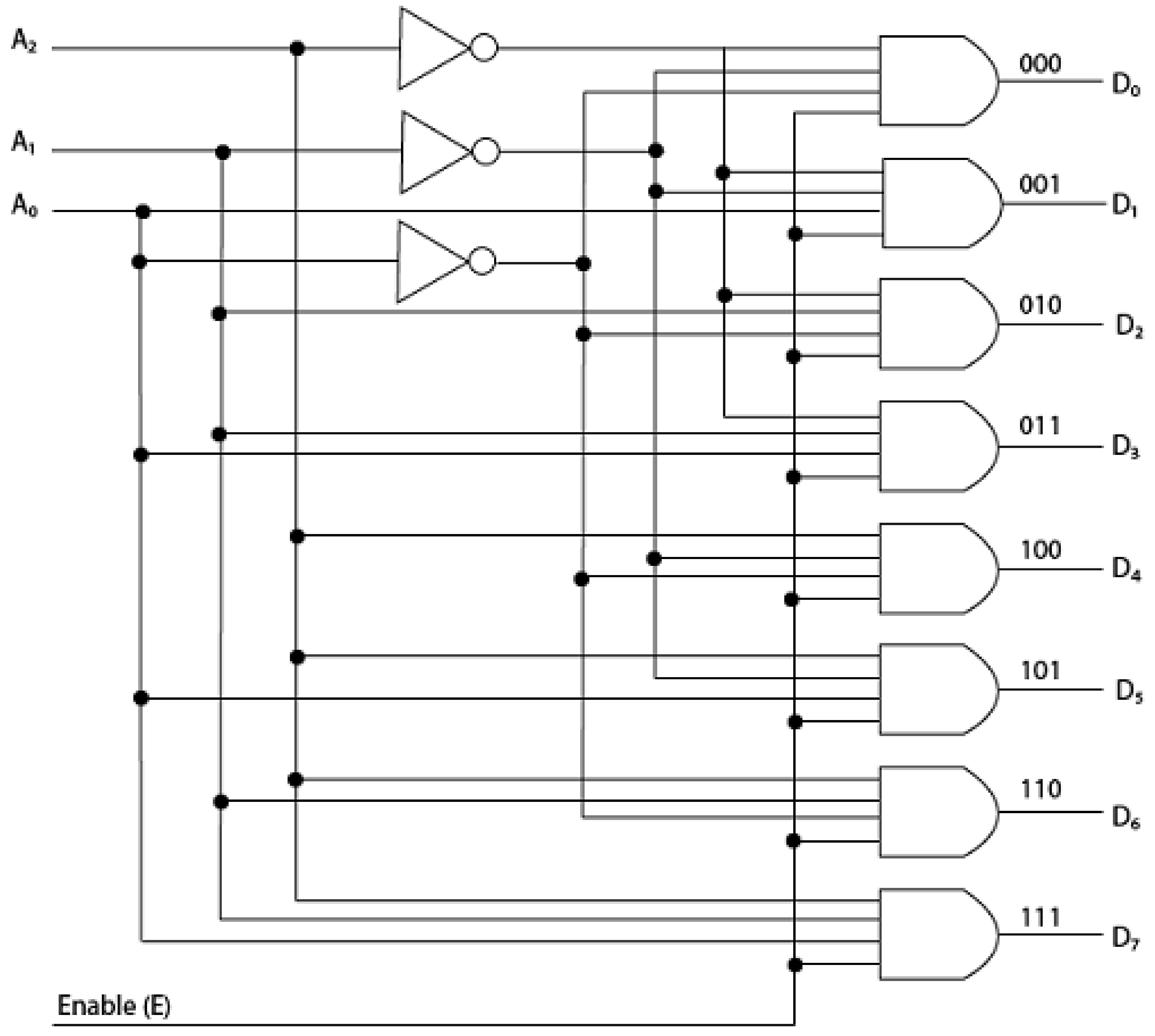
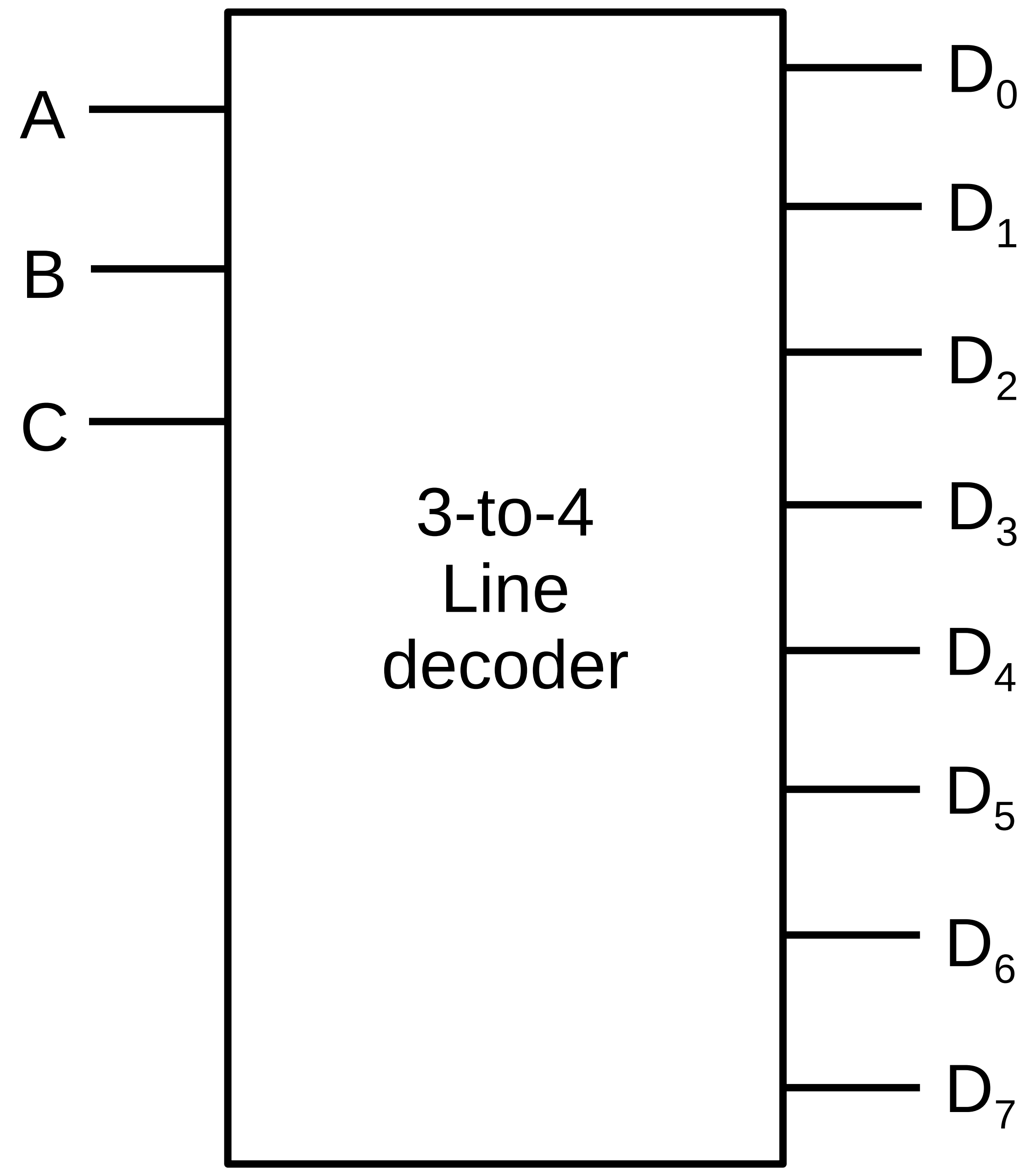


Figure 2-1. 3-to-8-line decoder



Enable	Inputs			Outputs							
E	A_2	A_1	A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	*x*	*x*	*x*	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

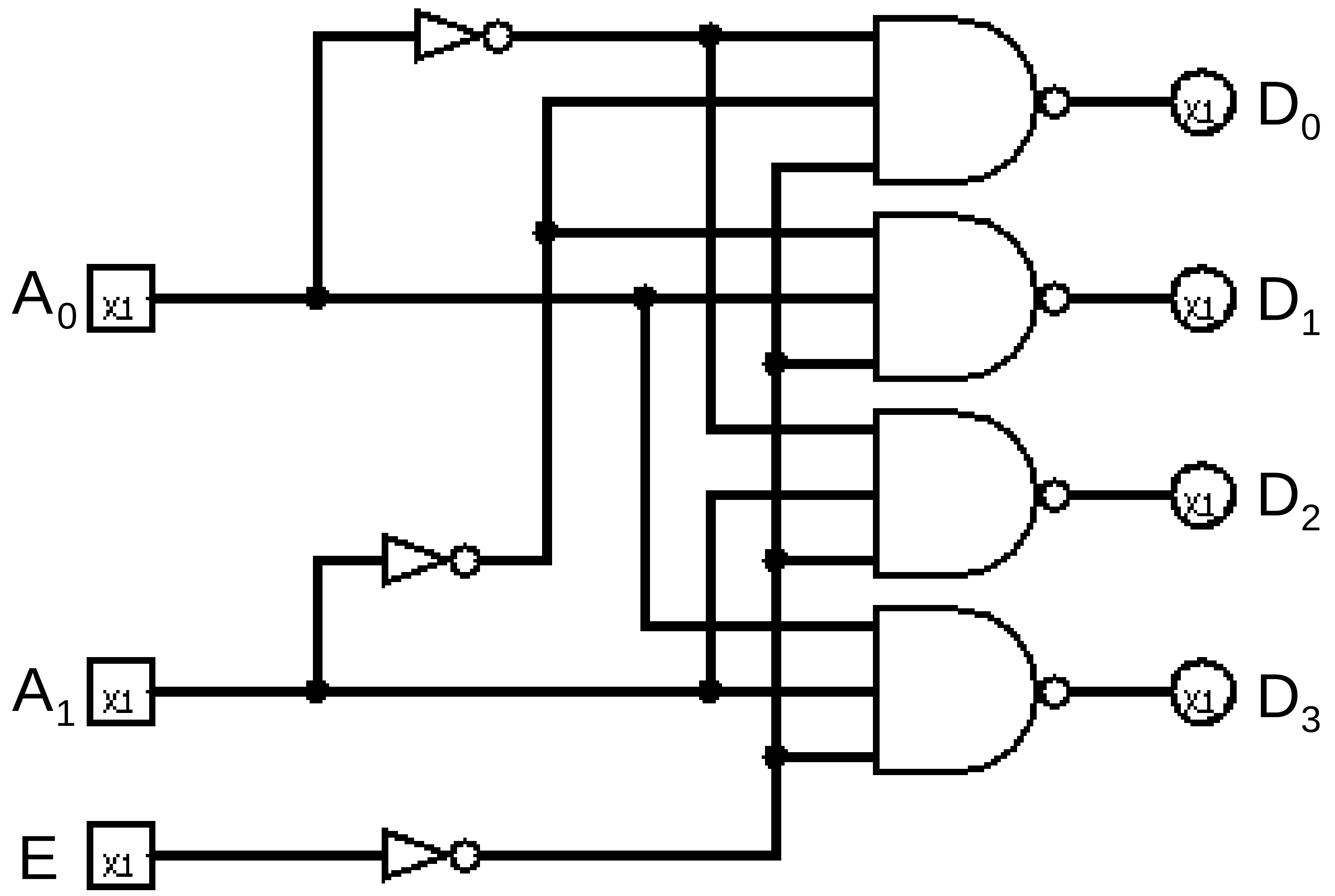
Table 2-1
Truth Table for 3-to-8-Line Decoder

The logic diagram of a 3-to-8-line decoder is shown in Fig. 2-1. The three data inputs, A_0, A_1, and A_2, are decoded into eight outputs, each output representing one of the combinations of the three binary input variables. The three inverters provide the complement of the inputs, and each of the eight AND gates generates one of the binary combination. A particular application of this decoder is a binary-to-octal conversion. The input variables represent a binary number and the outputs represent the eight digits of the octal number system. However, a 3-to-8-line decoder can be used for decoding any 3-bit code to provide eight outputs, one for each combination of the binary code.

Commercial decoders include one or more enable inputs to control the operation of the circuit. The decoder of Fig. 2-1 has one enable input, E. The decoder is enabled when E is equal to 1 and disabled when E is equal to 0.

The operation of the decoder can be clarified using the truth table listed in Table 2-1 . When the enable input E is equal to 0, all the outputs are equal to 0 regardless of the values of the other three data inputs. The three x's in the table designate don't-care conditions. When the enable input is equal to 1, the decoder operates in a normal fashion. For each possible input combination, there are seven outputs that are equal to 0 and only one that is equal to 1. The output variable whose value is equal to 1 represents the octal number equivalent of the binary number that is available in the input data lines.

NAND Gate Decoder



E	A_1	A_0	D_0	D_1	D_2	D_3
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0
1	*x*	*x*	1	1	1	1

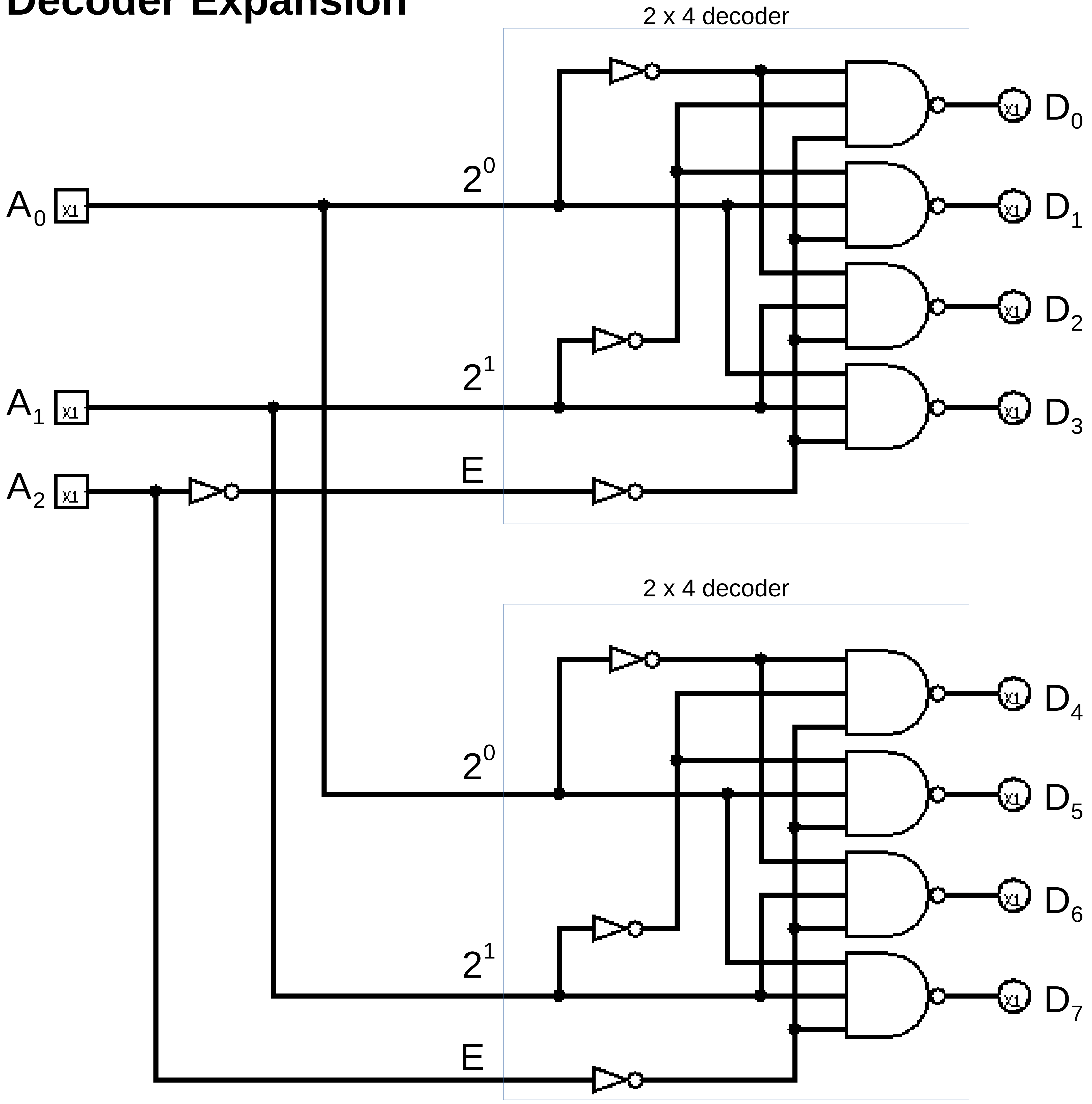
Figure 2-2

2-to-4-line decoder with NAND gates

Some decoders are constructed with NAND instead of AND gates. Since a NAND gate produces the AND operation with an inverted output, it becomes more economical to generate the decoder outputs in their complement form. A 2-to-4-line decoder with an enable input constructed with NAND gates is shown in Fig. 2-2. The circuit operates with complemented outputs and a complemented enable input E. The decoder is enabled when E is equal to 0. As indicated by the truth table, only one output is equal to 0 at any given time; the other three outputs are equal to 1. The output whose value is equal to 0 represents the equivalent binary number in inputs A_1 and A_0. The circuit is disabled when E is equal to 1, regardless of the values of the other two inputs.

When the circuit is disabled, none of the outputs are selected and all outputs are equal to 1. In general, a decoder may operate with complemented or uncomplemented outputs. The enable input may be activated with a 0 or with a 1 signal level. Some decoders have two or more enable inputs that must satisfy a given logic condition in order to enable the circuit.

Decoder Expansion



There are occasions when a certain-size decoder is needed but only smaller sizes are available. When this occurs it is possible to combine two or more decoders with enable inputs to form a larger decoder. Thus if a 6-to-64-line decoder is needed, it is possible to construct it with four 4-to-16-line decoders.

Figure 2-3 shows how decoders with enable inputs can be connected to form a larger decoder. Two 2-to-4-line decoders are combined to achieve a 3-to-8-line decoder. The two least significant bits of the input are connected to both decoders. The most significant bit is connected to the enable input of one decoder and through an inverter to the enable input of the other decoder. It is assumed that each decoder is enabled when its E input is equal to 1. When E is equal to 0, the decoder is disabled and all its outputs are in the 0 level. When $A_2 = 0$, the upper decoder is enabled and the lower is disabled. The lower decoder outputs become inactive with all outputs at 0. The outputs of the upper decoder generate outputs D_0 through D_3, depending on the values of A_1 and A_0 (while $A_2= 0$). When $A_2 = 1$, the lower decoder is enabled and the upper is disabled. The lower decoder output generates the binary equivalent D_4 through D_7 since these binary numbers have a 1 in the A_2 position.

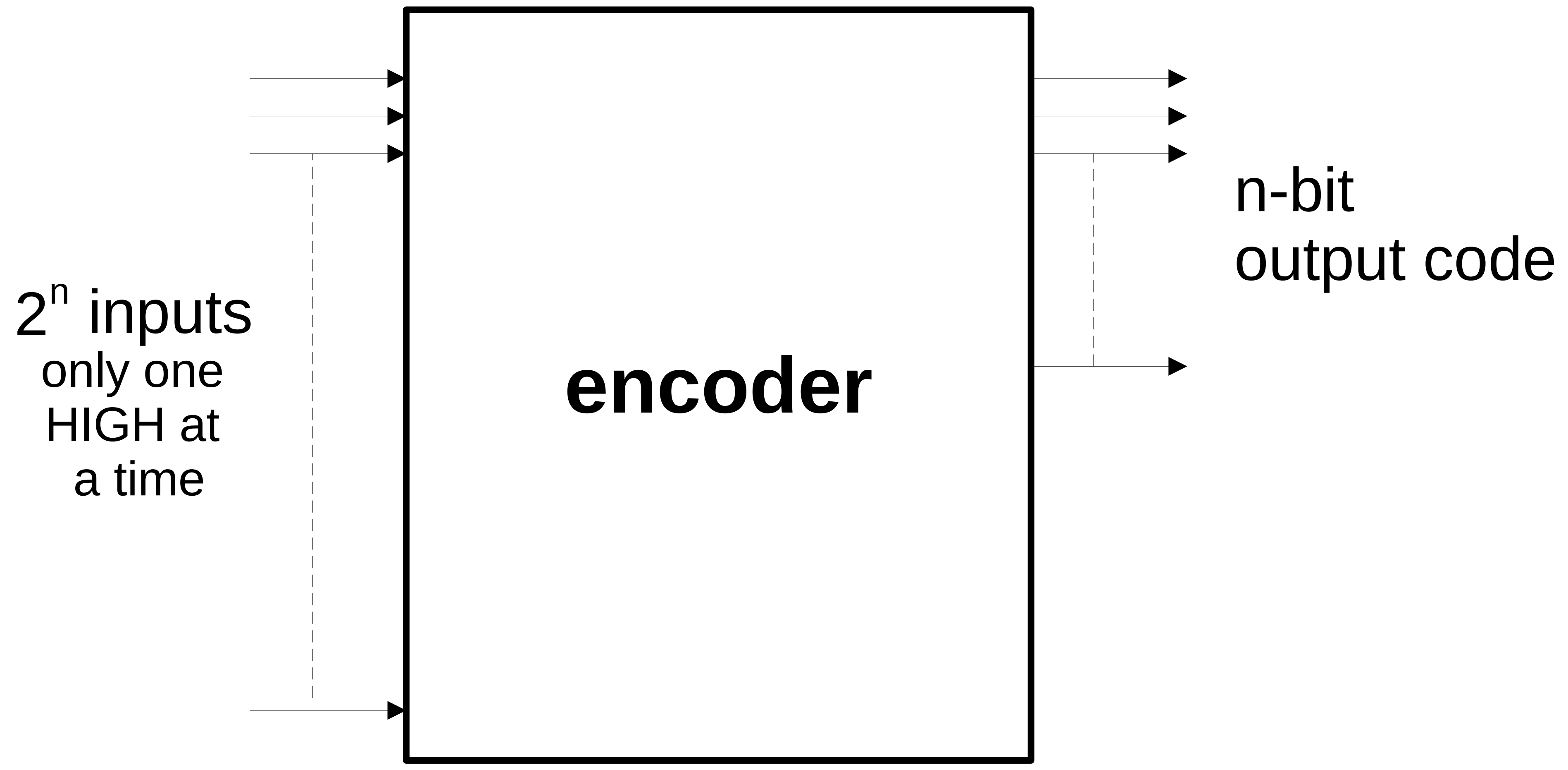
The example demonstrates the usefulness of the enable input in decoders or any other combinational logic component. Enable inputs are a convenient feature for interconnecting two or more circuits for the purpose of expanding the digital component into a similar function but with more inputs and outputs.

Encoders

An encoder is a digital circuit that performs binary conversion. Its operation is the inverse operation of a decoder.

Generally, in digital electronics; we send or receive data in binary formats. Encoders help to convert any other code into corresponding binary code.

It has 2^n (or less) input lines and n output lines. Out of those input lines, only one is activated (high, “1”) at a given time. The output lines generate the n-bit binary code corresponding to the input value.



Types of encoders

Based on the inputs, an encoder is divided into,

-
- 4 x 2 encoder
- 8 x 3 encoder
- 16 x 4 encoder and up to
- $2^{m \cdot m}$ encoder

Apart from input and output pins, the encoder also has an enable pin. Usually, it is designed to be either active high or active low. If it is active high, then you should apply binary ‘1’ to the ‘enable pin’ to perform encoding otherwise it won’t.

It can also be active low. If it is active low, then you have to give binary ‘0’ to the ‘enable pin’ to perform the operation.

Octal to Binary Encoder

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Table 2-2
Truth table for octal-to-Binary Encoder

An example of an encoder is the octal-to-binary encoder, whose truth table is given in Table 2-2. It has eight inputs, one for each of the octal digits, and three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time; otherwise, the circuit has no meaning.

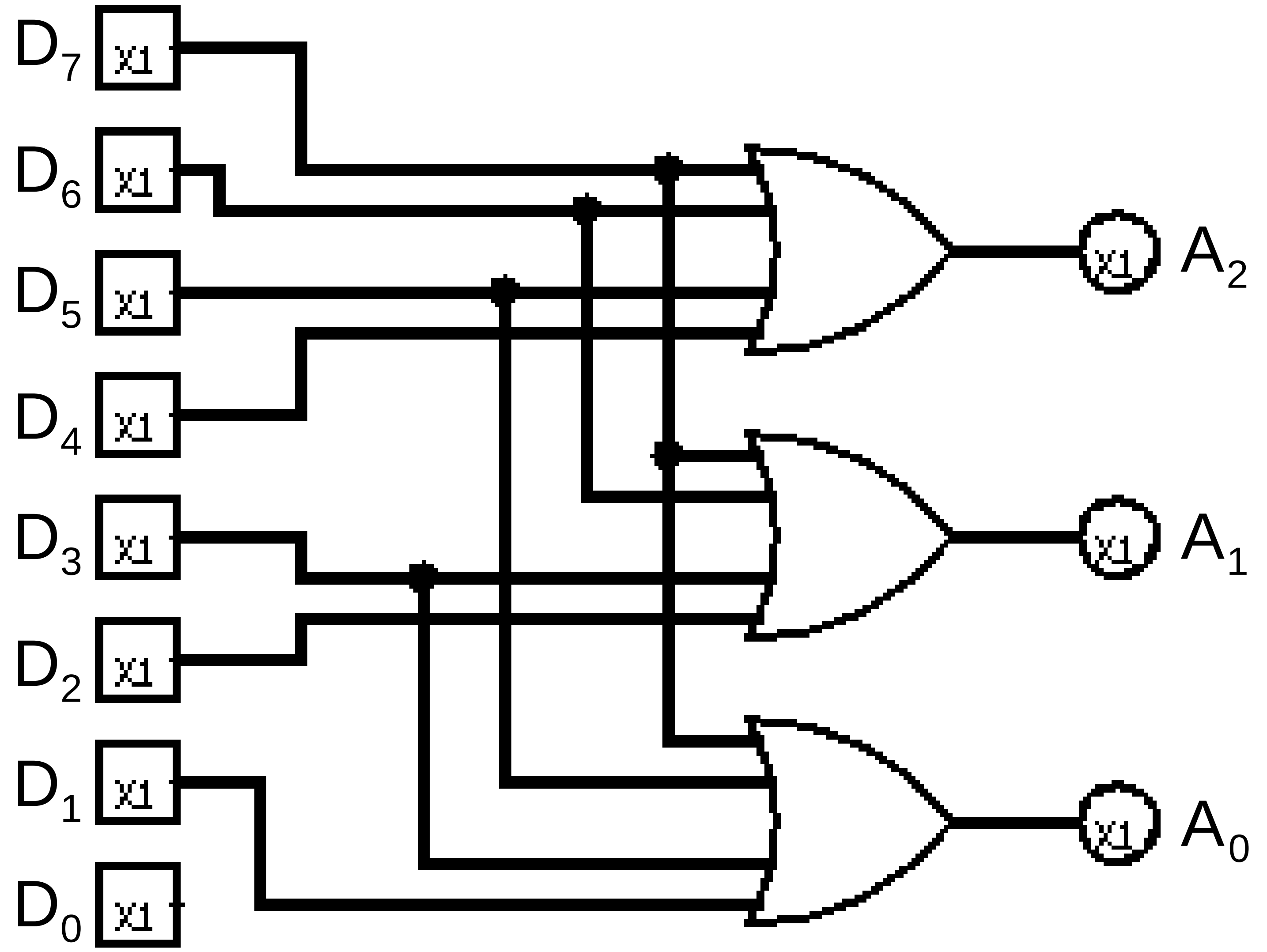
The encoder can be implemented with OR gates whose inputs are determined directly from the truth table. Output A_0 = 1 if the input octal digit is 1 or 3 or 5 or 7. Similar conditions apply for the other two outputs. These conditions can be expressed by the following Boolean functions:

$$A_0 = D_1 + D_3 + D_5 + D_7$$
$$A_1 = D_2 + D_3 + D_6 + D_7$$
$$A_2 = D_4 + D_5 + D_6 + D_7$$

The encoder can be implemented with three OR gates.

Actually, the octal-to-binary encoder converts (encodes) 1-digit numbers in the octal numeric system (from 0 to 7) to 3-bit binary numbers (from 000 to 111). We can also deduce this purpose from its truth table. The encoder has 8 inputs, one for each octal digit, and 3 outputs, for the encoded binary numbers.
Its implementation is derived straightforwardly from its boolean equations.

$$A_0 = D_1 + D_3 + D_5 + D_7$$
$$A_1 = D_2 + D_3 + D_6 + D_7$$
$$A_2 = D_4 + D_5 + D_6 + D_7$$

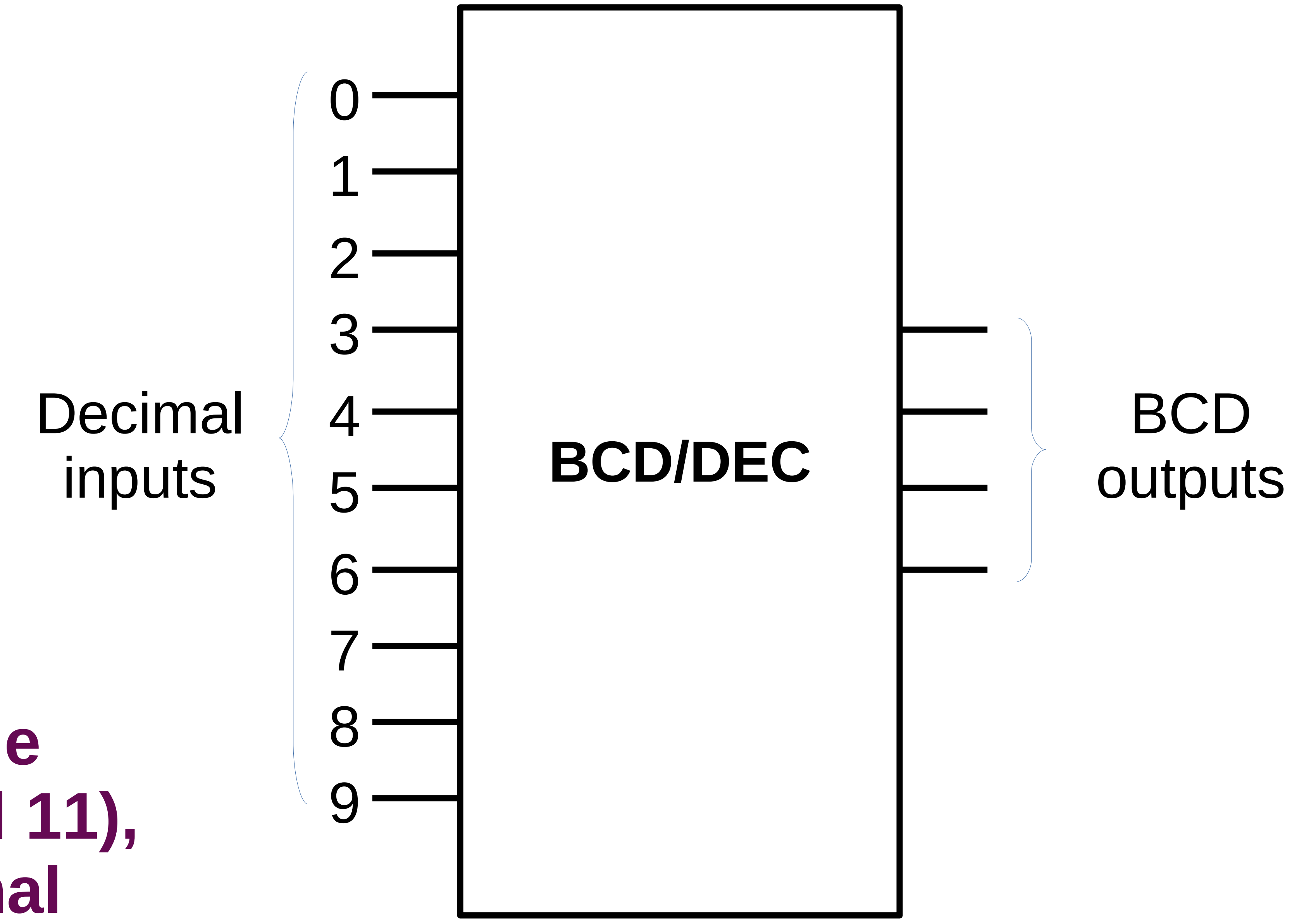


Decimal to BCD Encoder

This type of encoder has 10 inputs one for each decimal digit and 4 outputs corresponding to the BCD code.

But, what is a BCD code?

A Binary Coded Decimal, or BCD, is another process for converting decimal numbers into their binary equivalents. Bcd is represented by four bits and each displayed decimal digit from 0000 for a zero to 1001 for a nine.



In the BCD numbering system the six binary code combinations of 1010 (decimal 10), 1011 (decimal 11), 1100 (decimal 12), 1101 (decimal 13), 1110 (decimal 14), and 1111 (decimal 15) are classed as forbidden numbers and can not be used.

Let's look at the encoder's truth table. you can see ten inputs corresponding to 10 decimal inputs (D0 to D9) and 4 Outputs (Y_0, Y_1, Y_2, Y_3). Let's look at the table carefully. Note that each time only one of the inputs is activated (1) and other inputs are not activated (0).

Input										Output			
0	1	2	3	4	5	6	7	8	9				
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	Y_3	Y_2	Y_1	Y_0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	0	0	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	1	0	1
0	0	0	0	0	0	1	0	0	0	0	1	1	0
0	0	0	0	0	0	0	1	0	0	0	1	1	1
0	0	0	0	0	0	0	0	1	0	1	0	0	0
0	0	0	0	0	0	0	0	0	1	1	0	0	1

Truth Table of a Decimal to Binary Encoder

In the truth table, the input variable D_0 represents the least significant digit (LSD) and and D_9 represents most significant digit (MSD). Similarly, in the outputs Y0 represents least significant bit (LSB) and Y3 is the most significant bit (MSB). The truth table includes all valid combinations of the inputs. The valid combinations are those which have exactly one input high (1) while all other inputs are low (0’s).

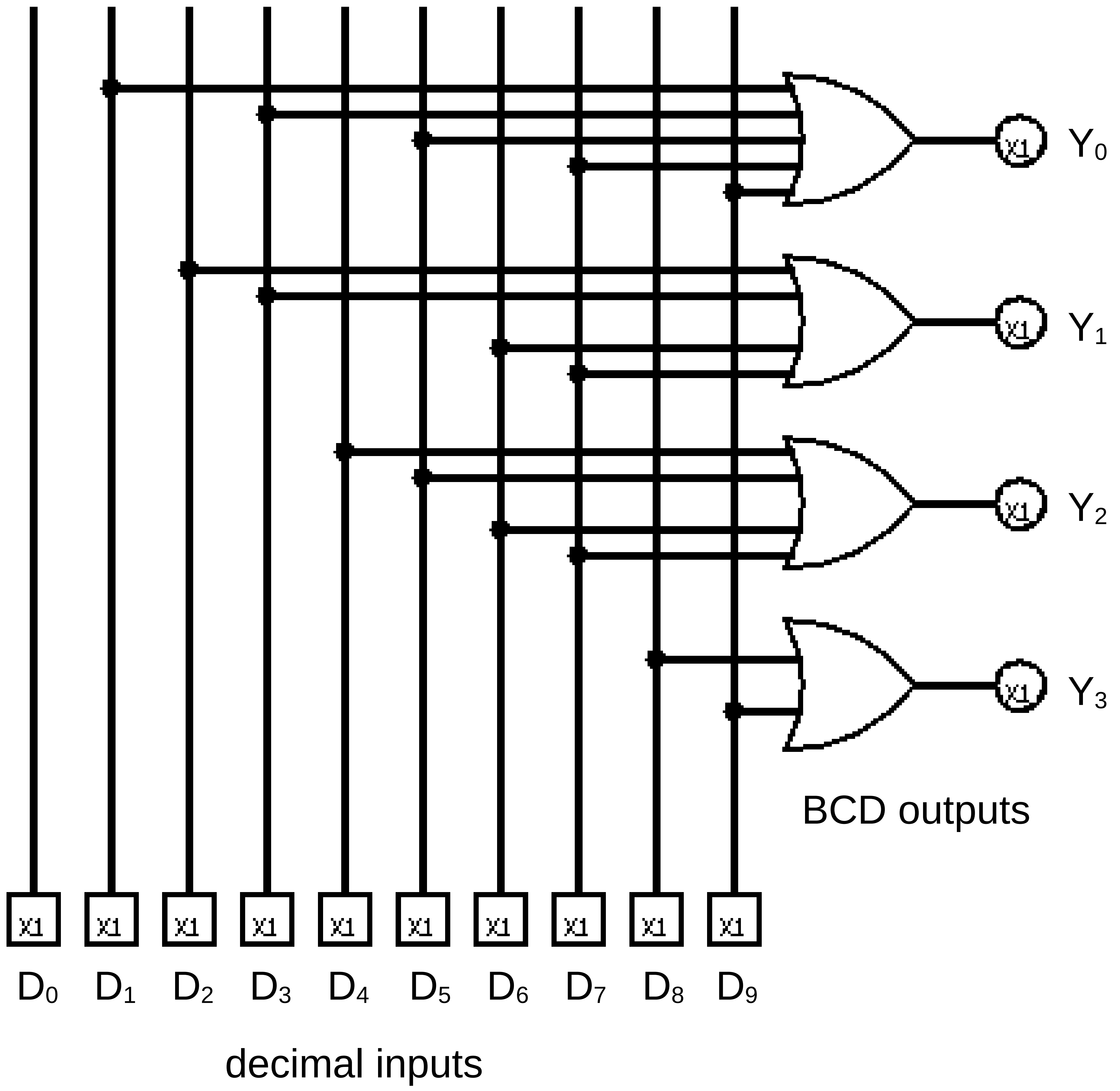
Because the encoder has too many inputs (10), using K-Map to derive its Boolean expression is far too cumbersome. However, the expression can be directly derived from the truth table by visual inspection. Let’s obtain the Boolean expression for each output.

Output Y_0 is 1 if any of the inputs D_1 or D_3 or D_5 or D_7 is 1. Then, the Boolean expression is:

$$D_0 = D_1 + D_3 + D_5 + D_7 + D_9$$

Similarly,:

$$Y_1 = D_2 + D_3 + D_6 + D_7$$
$$Y_2 = D_4 + D_5 + D_6 + D_7$$
$$Y3 = D_8 + D_9$$

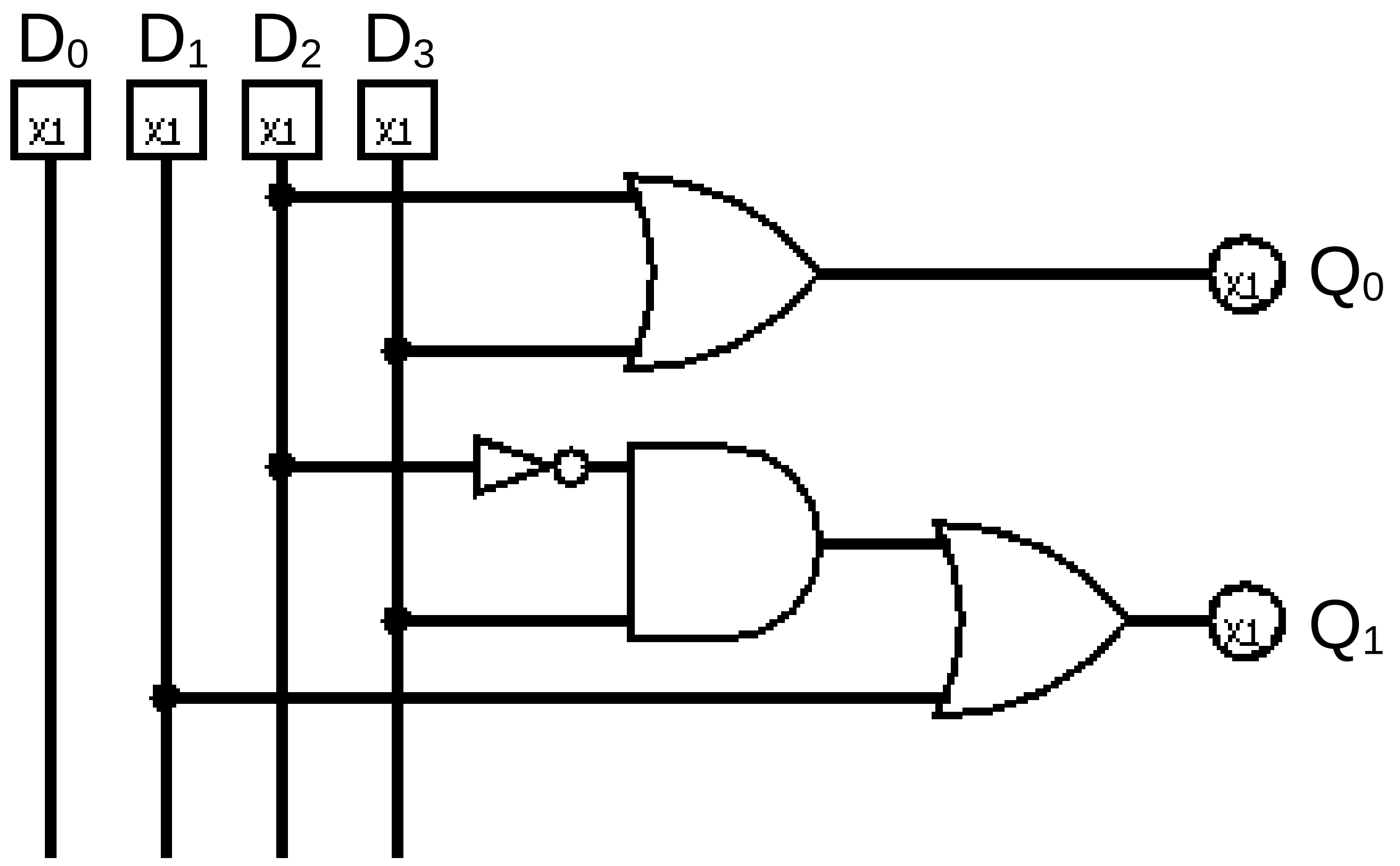


From these Boolean expressions, the Decimal to BCD Encoder can be implemented by using simply three 4 OR gates.

Priority encoder

Generally, digital encoders produce outputs of 2-bit, 3-bit or 4-bit codes depending upon the number of data input lines. An “n-bit” binary encoder has 2n input lines and n-bit output lines with common types that include 4-to-2, 8-to-3 and 16-to-4 line configurations.

The output lines of a digital encoder generate the binary equivalent of the input line whose value is equal to “1” and are available to encode either a decimal or hexadecimal input pattern to typically a binary or “B.C.D” (binary coded decimal) output code.

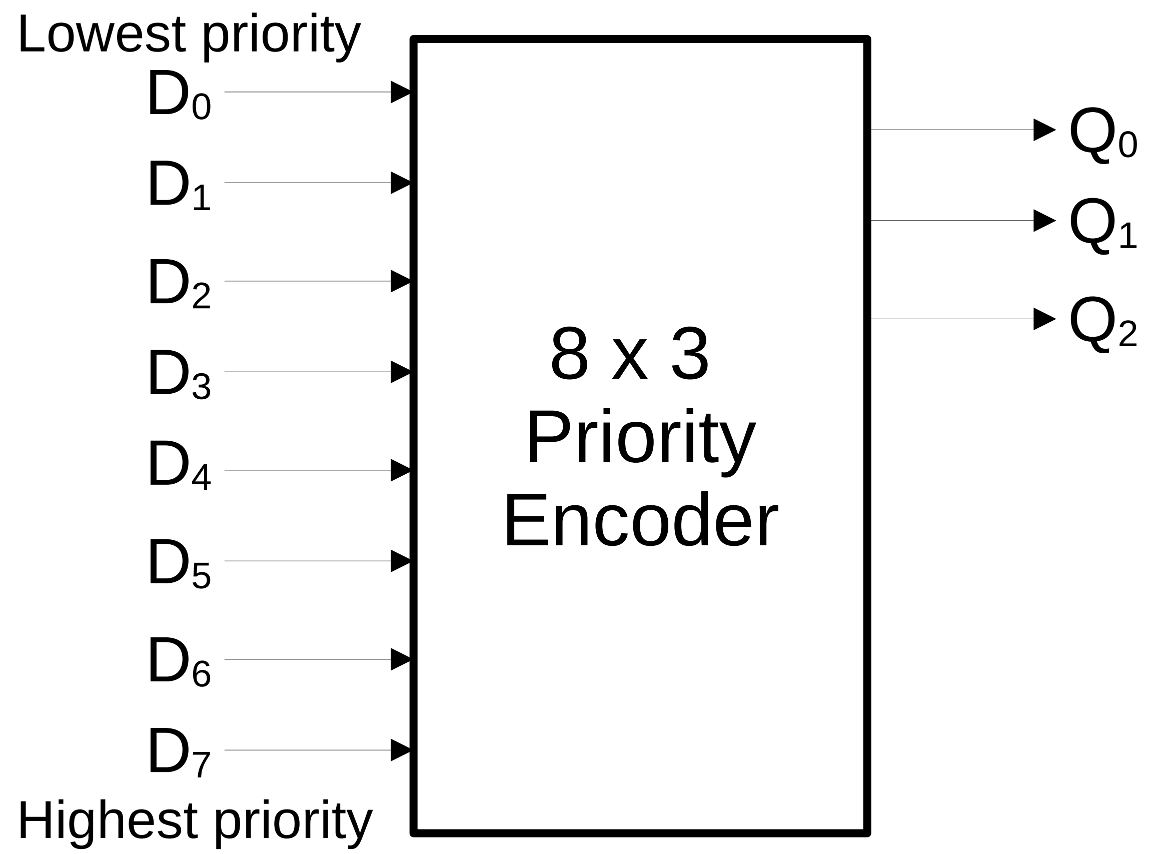


Inputs				Outputs	
D_0	**D_1**	**D_2**	**D_3**	**Q_0**	**Q_1**
0	0	0	0	x	x
0	0	0	1	1	1
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	0	0

4-to-2 line binary encoder

One of the main disadvantages of standard digital encoders is that they can generate the wrong output code when there is more than one input present at logic level “1”. For example, if we make inputs D_1 and D_2 HIGH at logic “1” both at the same time, the resulting output is neither at “01” or at “10” but will be at “11” which is an output binary number that is different to the actual input present. Also, an output code of all logic “0”s can be generated when all of its inputs are at “0” OR when input D_0 is equal to one.

One simple way to overcome this problem is to “Prioritise” the level of each input pin. So if there is more than one input at logic level “1” at the same time, the actual output code would only correspond to the input with the highest designated priority. Then this type of digital encoder is known commonly as a Priority Encoder or P-encoder for short.



Inputs								Outputs		
D_7	**D_6**	**D_5**	**D_4**	**D_3**	**D_2**	**D_1**	**D_0**	**Q_1**	**Q_2**	**Q_3**
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	x	0	0	1
0	0	0	0	0	1	x	x	0	1	0
0	0	0	0	1	x	x	x	0	1	1
0	0	0	1	x	x	x	x	1	0	0
0	0	1	x	x	x	x	x	1	0	1
0	1	x	x	x	x	x	x	1	1	0
1	x	x	x	x	x	x	x	1	1	1

8-to-3 Bit Priority Encoder

The Priority Encoder solves the problems mentioned above by allocating a priority level to each input. The priority encoders output corresponds to the currently active input which has the highest priority. So when an input with a higher priority is present, all other inputs with a lower priority will be ignored.

The priority encoder comes in many different forms with an example of an 8-input priority encoder along with its truth table shown below.

Priority encoders are available in standard IC form and the TTL 74LS148 is an 8-to-3 bit priority encoder which has eight active LOW (logic “0”) inputs and provides a 3-bit code of the highest ranked input at its output.

Priority encoders output the highest order input first for example, if input lines “D_2“, “D_3” and “D_5” are applied simultaneously the output code would be for input “D_5” (“101”) as this has the highest order out of the 3 inputs. Once input “D_5” had been removed the next highest output code would be for input “D_3” (“011”), and so on.

According to the truth table of 8-to-3 priority encoder, where X equals “dont care”, that is it can be at a logic “0” level or at a logic “1” level.

From this truth table, the Boolean expression for the encoder above with data inputs D0 to D7 and outputs Q0, Q1, Q2 is given as:

Output Q_0:

$$Q_0 = \Sigma(1, 3, 5, 7)$$
$$Q_0 = \bar{D}_7\bar{D}_6\bar{D}_5\bar{D}_4\bar{D}_3\bar{D}_2D_1 + \bar{D}_7\bar{D}_6\bar{D}_5\bar{D}_4D_3 + \bar{D}_7\bar{D}_6D_5 + D_7$$
$$Q_0 = \bar{D}_6\bar{D}_4\bar{D}_2D_1 + \bar{D}_6\bar{D}_4D_3 + \bar{D}_6\bar{D}_4D_3 + \bar{D}_6D_5 + D_7$$
$$Q_0 = \bar{D}_6\left(\bar{D}_4\bar{D}_2D_1 + \bar{D}_4D_3 + D_5\right) + D_7$$

Output Q_1:

$$Q_1 = \Sigma(2, 3, 6, 7)$$
$$Q_1 = \bar{D}_7\bar{D}_6\bar{D}_5\bar{D}_4\bar{D}_3D_2 + \bar{D}_7\bar{D}_6\bar{D}_5\bar{D}_4D_3 + \bar{D}_7D_6 + D_7$$
$$Q_1 = \bar{D}_5\bar{D}_4D_2 + \bar{D}_5\bar{D}_4D_3 + D_6 + D_7$$
$$Q_1 = \bar{D}_5\bar{D}_4\left(D_2 + D_3\right) + D_6 + D_7$$

Output Q_2:

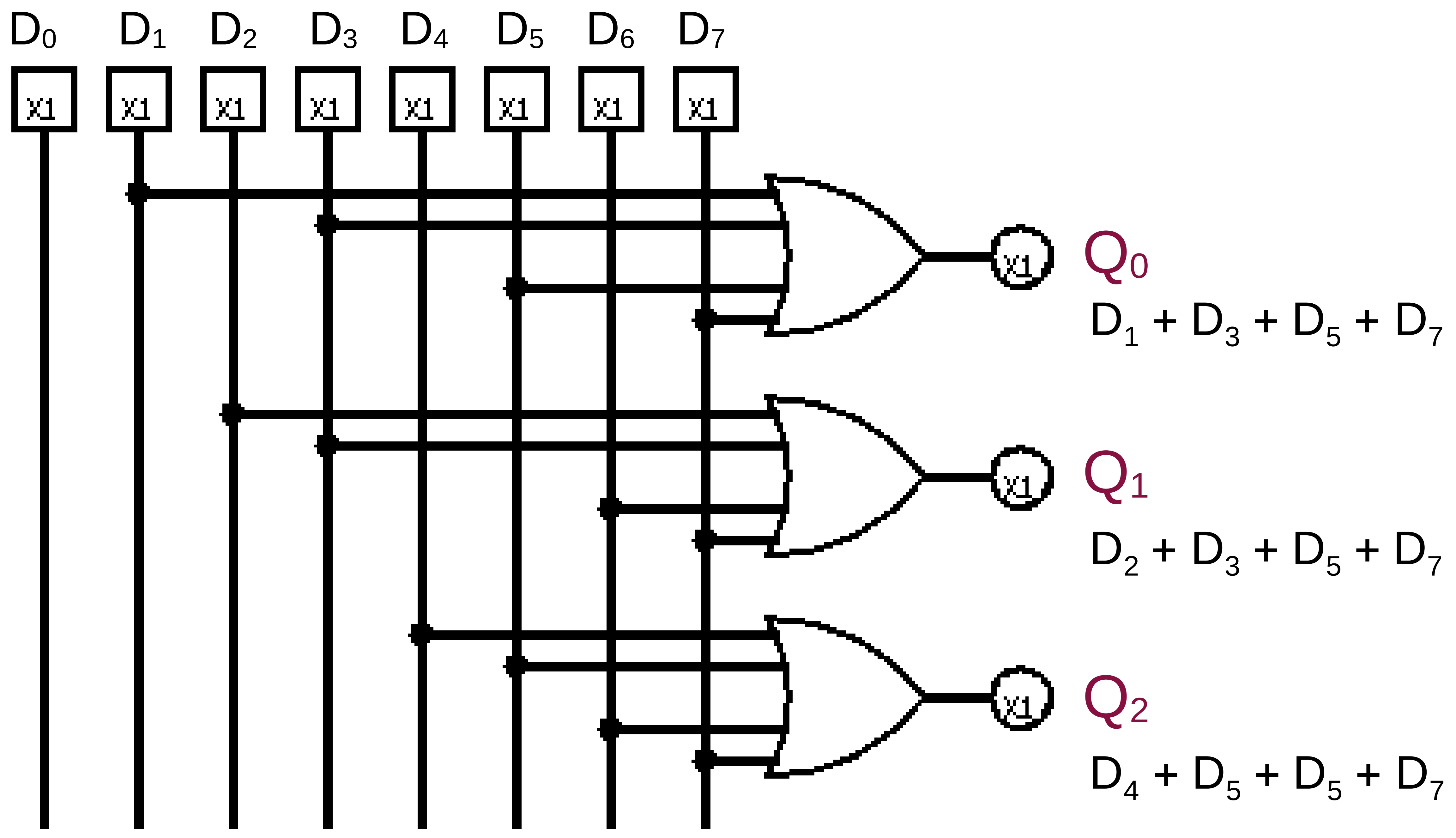
$$Q_2 = \Sigma(4, 5, 6, 7)$$
$$Q_2 = \bar{D}_7\bar{D}_6\bar{D}_5D_4 + \bar{D}_7\bar{D}_6D_5 + D_7$$
$$Q_2 = D_4 + D_5 + D_6 + D_7$$

So from this, the final boolean expressions of 8-to-3 bit priority encoder (including zero inputs) are:

$$Q_0 = \bar{D}_6\left(\bar{D}_4\bar{D}_2 D_1 + \bar{D}_4 D_3 + D_5\right) + D_7$$
$$Q_1 = \bar{D}_5\bar{D}_4\left(D_2 + D_3\right) + D_6 + D_7$$
$$Q_2 = D_4 + D_5 + D_6 + D_7$$

In practice, these zero inputs would be ignored, so the final boolean expression for the 3 outputs of the encoder will be those:

$$Q_0 = D_1 + D_3 + D_5 + D_7$$
$$Q_1 = D_2 + D_3 + D_6 + D_7$$
$$Q_2 = D_4 + D_5 + D_6 + D_7$$



Logic Diagram for 8-to-3 Bit Priority Encoder

Implementation of a priority encoder as a keyboard encoder

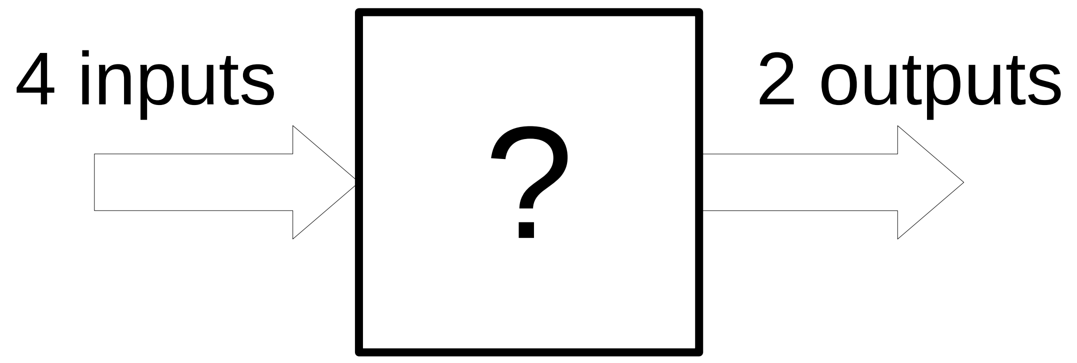
Priority encoders can be used to reduce the number of wires needed in a particular circuits or application that have multiple inputs. For example, assume that a microcomputer needs to read the 104 keys of a standard QWERTY keyboard where only one key would be pressed either “HIGH” or “LOW” at any one time.

One way would be to connect all 104 wires from the individual keys on the keyboard directly to the computers input but this would be impractical for a small home PC. Another alternative and better way would be to interface the keyboard to the PC using a priority encoder.

The 104 individual buttons or keys could be encoded into a standard ASCII code of only 7-bits (0 to 127 decimal) to represent each key or character of the keyboard and then input as a much smaller 7-bit B.C.D code directly to the computer. Keypad encoders such as the 74C923 20-key encoder are available to do just that.

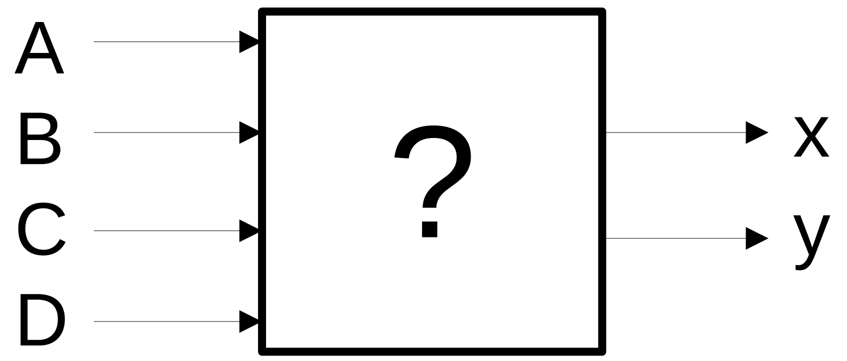
Step by step design of a simple 2-to-4 line encoder

Step 1: Decide about the number of the inputs and the outputs



The circuit that we're about to create is a 4-to-4 line encoder, which purpose is to encode a signal that is coming in by 1 of 4 input lines, so at any given time only one of the input lines will be 1, and all the other are 0. To bear such a signal encoded binarily, we need a minimum of 2 lines, hence we need **4** input and **2** output lines.

Step 2: Assign arbitrary letter variables to all the inputs and outputs.



Let's assume that the variables for the inputs will be **A**, **B**, **C**, and **D**, and for the outputs, they will be **x** and **y**.

Step 3: Create a "preassumed requirements" minimized truth table.

We know that the decoder must decode four 4-bit signals when each of them bears only one high ("1") bit, with the rest being low ("0"). So, the outputs must bear those 4 signals encoded, so we must have a number of inputs that is the square root of the number of outputs. Additionally, we also can have a situation when the input signal has all bits low, so in simpler words the absence of the input signal, and in this situation, we're don't care about the combination of the output bits. Based on this, we can design a truth table that is reflecting our requirements from the circuit

A	B	C	D	x	y
0	0	0	0	*x*	*x*
0	0	0	1	1	1
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	0	0

Step 4: Creating the long-hand (unabridged) truth table.

We're going to obtain the 2 outputs boolean expressions using a K-Map method. Because we have 2 outputs, we'll need also two K-Maps. But we can't use our "preassumed requirements" short truth table in order to know which cell in each map corresponds to which minterm in the truth table. To do so, we need to interpose our minimized truth table on the long-hand truth table, which also includes the minterms that we're don't care about them, those of the "illegal" signals that have more than one high bit

Knowing now that the numbers of the relevant minterms are 1, 2, 4, and 8, we can transfer the information from this truth table to two K-Maps.

After interposing the pre-requirements truth table on the long-hand truth table, we see that only 4 minterms (those which came from the short-hand truth table) are relevant to the purpose of the circuit, we simply don't care about any other minterm, so we can mark them with **x**.

A1	B	C	D	X	Y	
0	0	0	0	x	x	0
0	0	0	**1**	1	1	1
0	0	**1**	0	0	1	2
0	0	1	1	x	x	3
0	**1**	0	0	1	0	4
0	1	0	1	x	x	5
0	1	1	0	x	x	6
0	1	1	1	x	x	7
1	0	0	0	0	0	8
1	0	0	1	x	x	9
1	0	1	0	x	x	10
1	0	1	1	x	x	11
1	1	0	0	x	x	12
1	1	0	1	x	x	13
1	1	1	0	x	x	14
1	1	1	1	x	x	15

Step 5: Creating two relevant K-Maps and obtaining two simplified boolean expressions for both outputs.

x				
	C’D’	C’D	CD	CD’
A’B’	x	1	x	0
A’B	1	x	x	x
AB	x	x	x	x
AB’	0	x	x	x

x = B + C’D

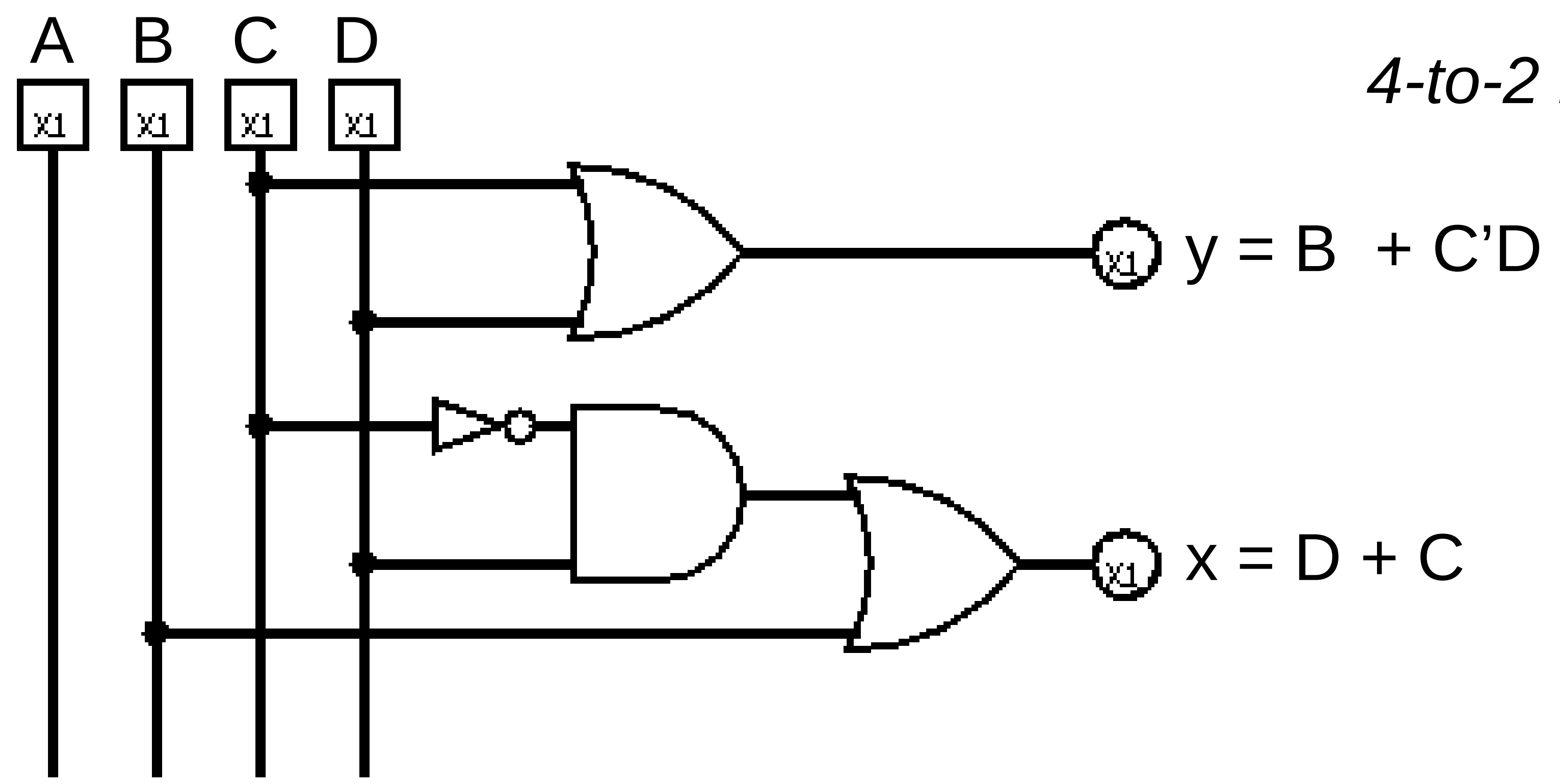
y				
	C’D’	C’D	CD	CD’
A’B’	x	1	x	1
A’B	0	x	x	x
AB	x	x	x	x
AB’	0	x	x	x

y = D + C

Step 6: Creating a logic diagram of the circuit using boolean expressions.

Having boolean expression for each one of the circuit's outputs, now we can fill the gap between the inputs and the outputs and create a logic diagram for the circuit

4-to-2 line Binary Decoder



Multiplexers

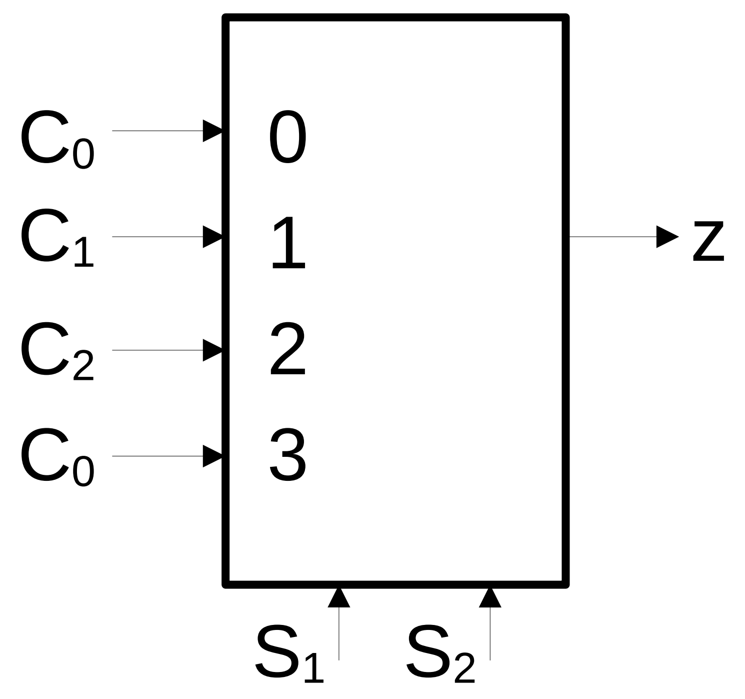
A multiplexer (abbreviated MUX) is a combinational circuit that receives binary information from one of 2^n input data lines and directs it to a single output line. The selection of a particular input data line for the output is determined by a set of selection inputs. A 2^n-to-1 multiplexer has 2^n input data lines and n input selection lines whose bit combinations determine which input data are selected for the output.

Multiplexer means many into one. In electronics, a multiplexer (MUX) is a logic gates circuit that employed to fetch a bit of data from computer memory at a given address. What's more, a computer processor has several multiplexers to control the data and address buses. In other words, it is a special circuit device that selects one of n inputs and provides it on output. Multiplexer technology may follow one of the following principles, such as: TDM, FDM, CDM or WDM. It is also applied to software operations, such as the simultaneous transmission of multi-threaded information streams to equipment or programs.

The purpose of using a multiplexer is to make full use of the capacity of the communication channel and greatly reduce the cost of the system. Giving a really simple example:
If you want to fetch a data bit from computer memory, a multiplexer allows you to specify an address (the input code), and the memory bit will be connected to the other pin.

For example, if you have 256 bits of memory and want to connect this to an output pin, then the multiplexer would need 8 bits for input codes. You proved a code say n, and bit n is connected through the logic gates to the output of the multiplexer. This multiplexer would have a total of 256 + 8 input lines.

That is, when the data stream on the multiplexed line is continuous, this sharing method can achieve good results. Obviously, this is more economical than using a communication line for each terminal. Multiplexers are always used in pairs. One continuous terminal, the other near the host, its function is to separate the received composite data stream according to the channel and send them to the corresponding output line, so it is called a Multiplexer.



The major concept of a mux then can be shown as a construct where select lines may be used to pick one out of many input signals. In a word, a Multiplexer is a selection device.

A big question covered as to where this may be useful? Generally, multiplexers are applied everywhere in computer architecture and ASIC design to help pick between multiple signals. In your computer processor, Multiplexers are employed to predict a branch (when running a loop, the branch is the assembly instruction and the processor needs to guess if it should take the branch or not), used to get data from caches (given an index in the cache muxes must be used to pick the correct data out of many), to select control signals for the processor based on the instruction that comes in, and to select one out of the many ALU calculations made based on the operation needed, and more.

4-to-1 Line Multiplexer

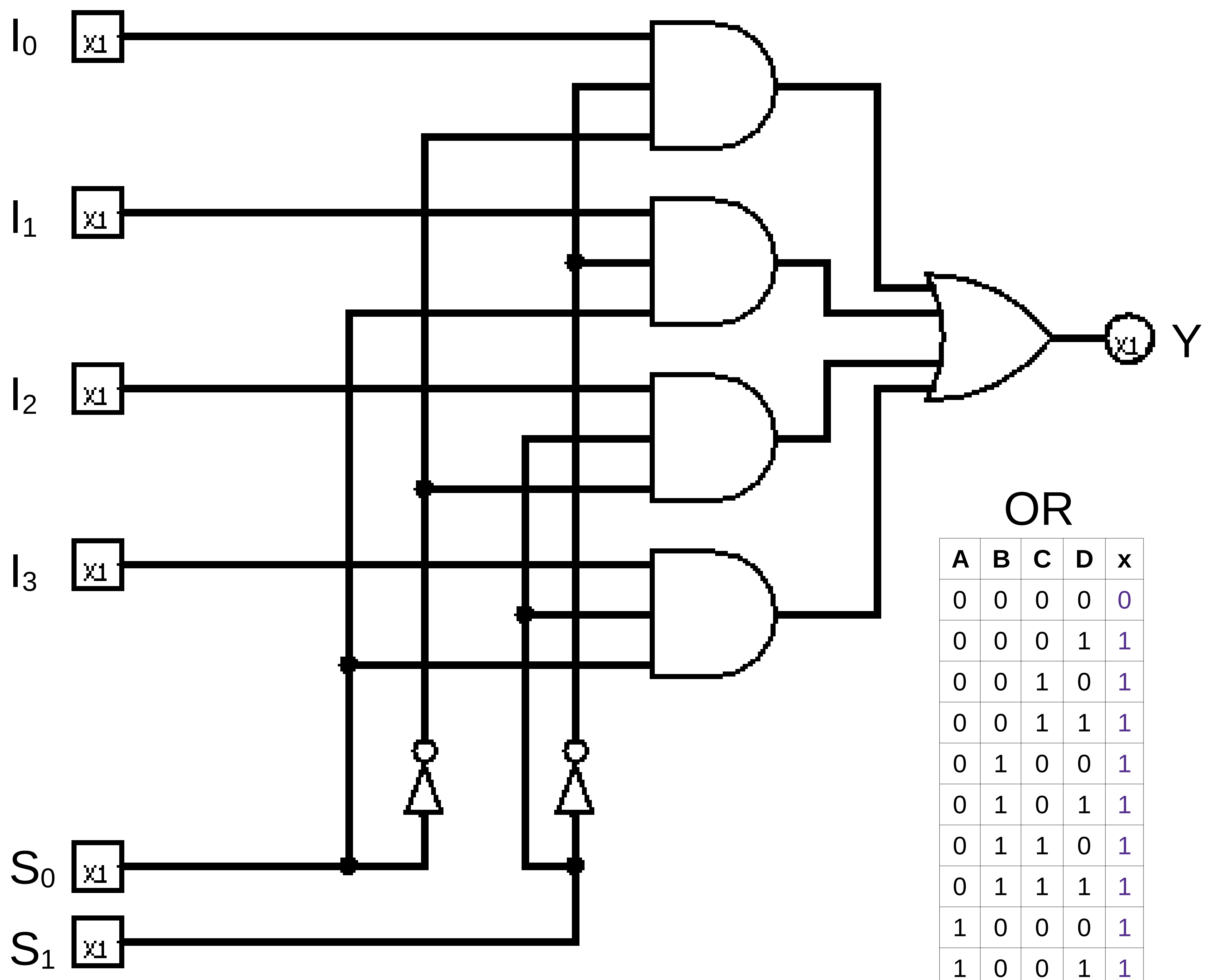


Figure 2-4
4-to-1 line multiplexer

OR

A	B	C	D	x
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

A multiplexer is a combinational circuit that receives binary information from one of 2^n input data lines and directs it to a single output line. The selection of a particular input data line for the output is determined by a set of selection inputs. A 2^n-to-1 multiplexer has 2^n input data lines and n input selection lines whose bit combinations determine which input data are selected for the output.

A 4-to-1-line multiplexer is shown in Fig. 2-4. Each of the four data inputs I_0 through I_3 is applied to one input of an AND gate. The two selection inputs S_1 and S_0 are decoded to select a particular AND gate. The outputs of the AND gates are applied to a single OR gate to provide the single output. To demonstrate the circuit operation, consider the case when $S_1S_0 = 10$. The AND gate associated with input I_2, has two of its inputs equal to 1. The third input of the gate is connected to I_2. The other three AND gates have at least one input equal to 0, which makes their outputs equal to 0. The OR gate output is now equal to the value of I_2, thus providing a path from the selected input to the output.

Select		Output
S_1	**S_2**	**Y**
0	0	I_1
0	1	I_2
1	0	I_3
1	1	I_4

Table 2-3
Function Table for
4-to-1-line Multiplexer

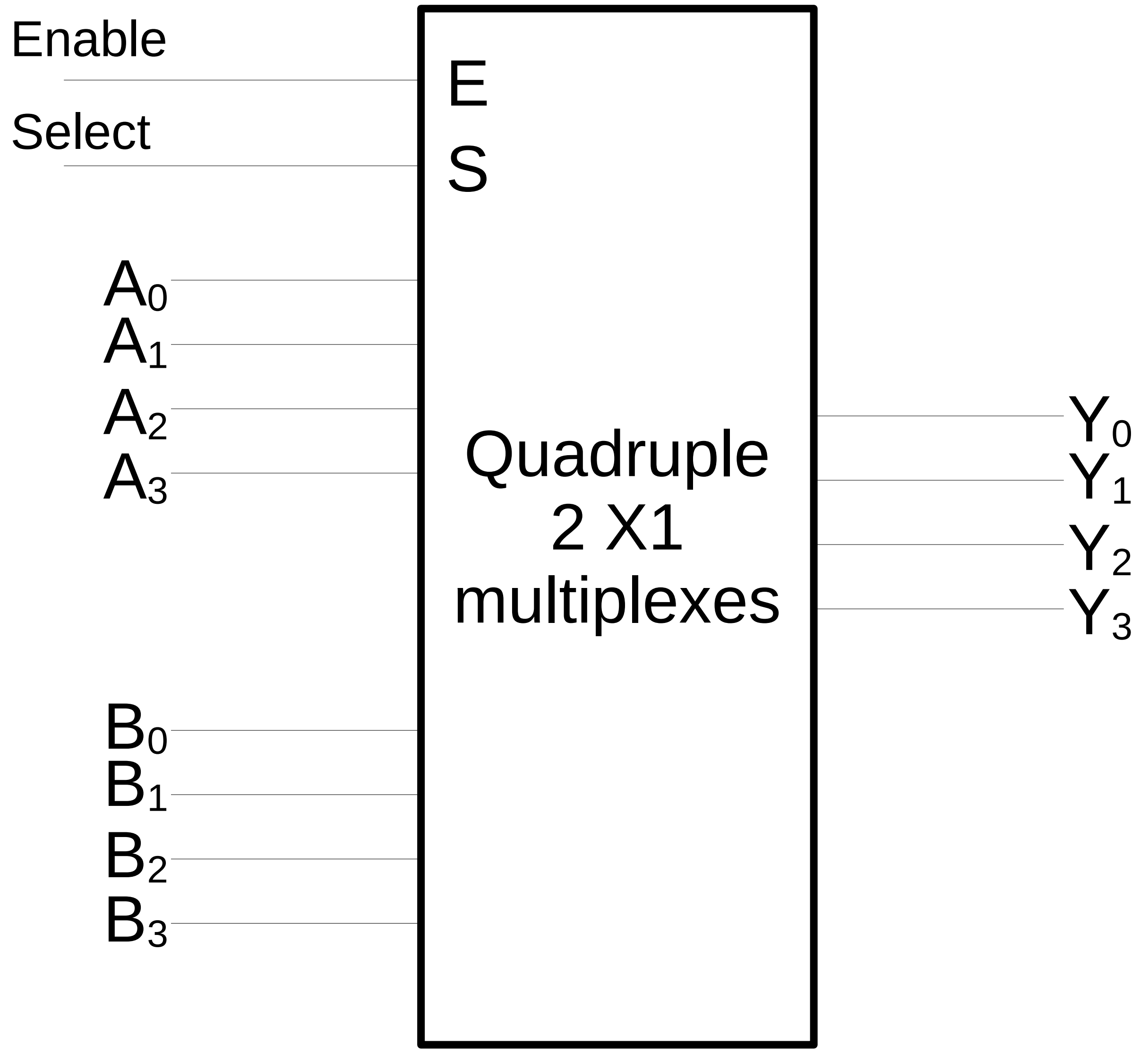
The 4-to-1 line multiplexer of Fig. 2-4 has six inputs and one output. A truth table describing the circuit needs 64 rows since six input variables can have 26 binary combinations. This is an excessively long table and will not be shown here. A more convenient way to describe the operation of multiplexers is by means of a function table. The function table for the multiplexer is shown in Table 2-3. The table demonstrates the relationship between the four data inputs and the single output as a function of the selection inputs S_1 and S_0.

When the selection inputs are equal to 00, output Y is equal to input I_0. When the selection inputs are equal to 01, input I_1 has a path to output Y, and similarly for the other two combinations. The multiplexer is also called a data selector, since it selects one of many data inputs and steers the binary information to the output.

The AND gates and inverters in the multiplexer resemble a decoder circuit, and indeed they decode the input selection lines. In general, a 2^n-to-1- line multiplexer is constructed from an n-to-2^n decoder by adding to it 2^n input lines, one from each data input. The size of the multiplexer is specified by the number 2^n of its data inputs and the single output. It is then implied that it also contains n input selection lines. The multiplexer is often abbreviated as MUX.

As in decoders, multiplexers may have an enable input to control the operation of the unit. When the enable input is in the inactive state, the outputs are disabled, and when it is in the active state, the circuit functions as a normal multiplexer. The enable input is useful for expanding two or more multiplexers to a multiplexer with a larger number of inputs.

In some cases two or more multiplexers are enclosed within a single integrated circuit package. The selection and the enable inputs in multiple-unit construction are usually common to all multiplexers. As an illustration, the block diagram of a quadruple 2-to-1-line multiplexer is shown in Fig. 2-5. The circuit has four multiplexers, each capable of selecting one of two input lines. Output Y_0 can be selected to come from either input A_0 or B_0. Similarly, output Y_1 may have the value of A_1 or B_1, and so on. One input selection line S selects one of the lines in each of the four multiplexers. The enable input E must be active for normal operation. Although the circuit contains four multiplexers, we can also think of it as a circuit that selects one of two 4-bit data lines. As shown in the function table, the unit is enabled when E = 1. Then, if S = 0, the four A inputs have a path to the four outputs. On the other hand, if S = 1, the four B inputs are applied to the outputs. The outputs have all O's when E = 0, regardless of the values of S .



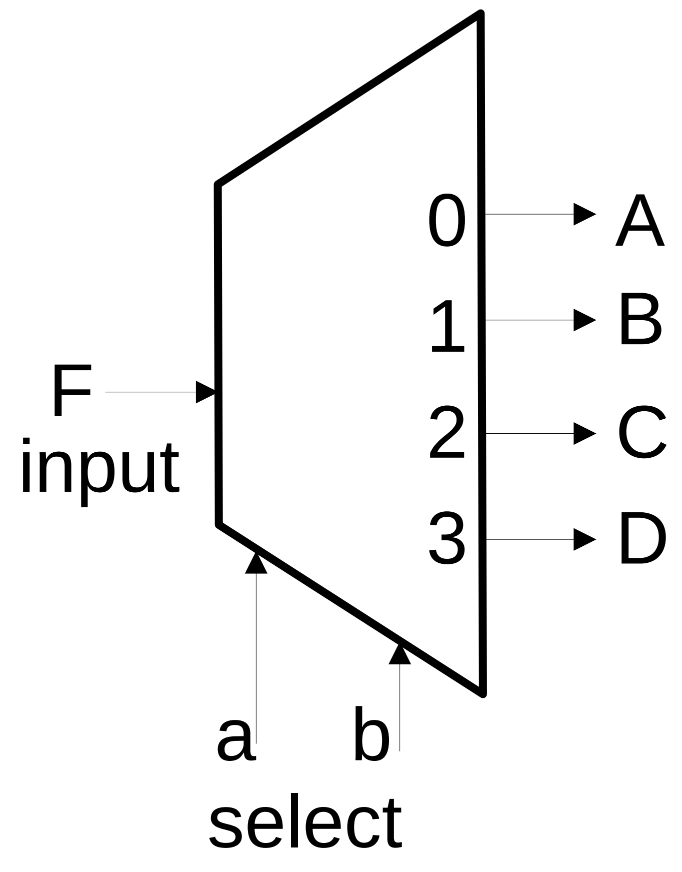
(a) Block diagram

E	**S**	**Y**
0	*x*	All 0's
1	0	A
1	1	B

(b) Function table

Figure 2-5
Quadruple 2-to-1 line multiplexers

The Demultiplexer



The data distributor, known more commonly as a Demultiplexer or “Demux” for short, is the exact opposite of the Multiplexer.

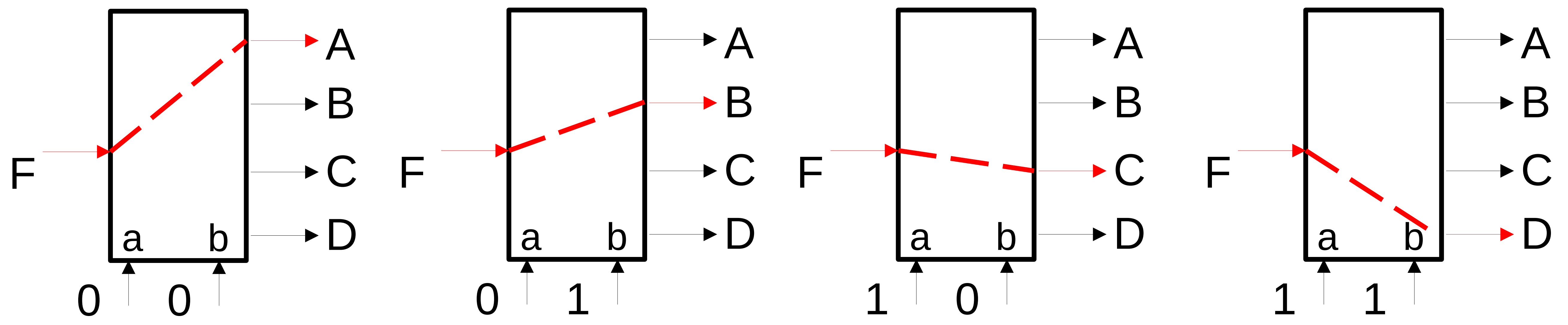
The demultiplexer takes one single input data line and then switches it to any one of a number of individual output lines one at a time. The demultiplexer converts a serial data signal at the input to a parallel data at its output lines as shown below.

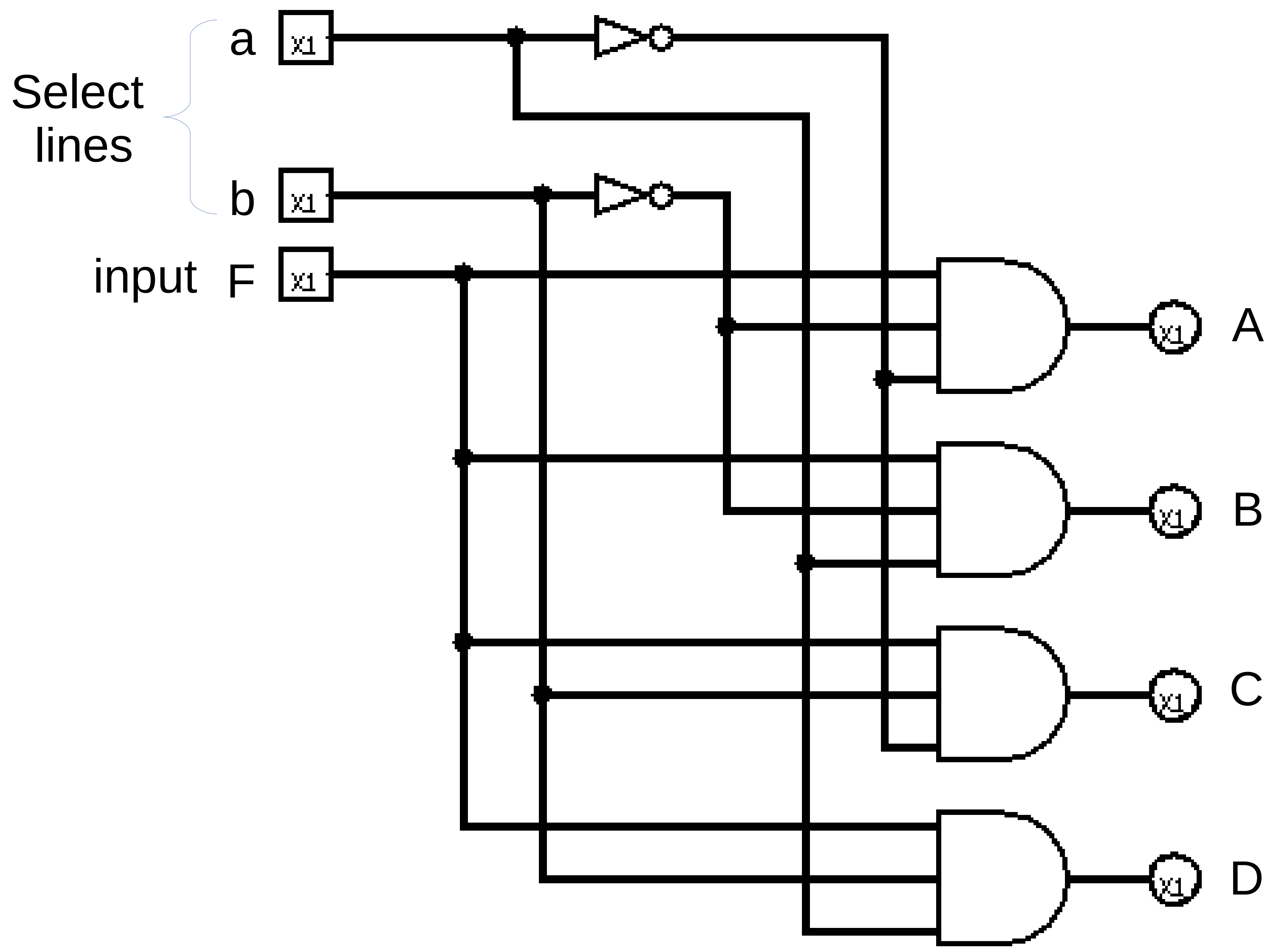
Output Selected		Data Output Selected
a	b	
0	0	A
0	1	B
1	0	C
1	1	D

The Boolean expression for this 1-to-4 Demultiplexer with outputs A to D and data select lines a, b is given as:

$$F = abA + abB + abC + abD$$

The function of the Demultiplexer is to switch one common data input line to any one of the 4 output data lines A to D in our example above. As with the multiplexer the individual solid state switches are selected by the binary input address code on the output select pins “a” and “b” as shown.





As with the previous multiplexer circuit, adding more address line inputs it is possible to switch more outputs giving a 1-to-2^n data line outputs.

Some standard demultiplexer IC´s also have an additional “enable output” pin which disables or prevents the input from being passed to the selected output. Also some have latches built into their outputs to maintain the output logic level after the address inputs have been changed.

However, in standard decoder type circuits the address input will determine which single data output will have the same value as the data input with all other data outputs having the value of logic “0”.

The implementation of the Boolean expression above using individual logic gates would require the use of six individual gates consisting of AND and NOT gates as shown.

Registers

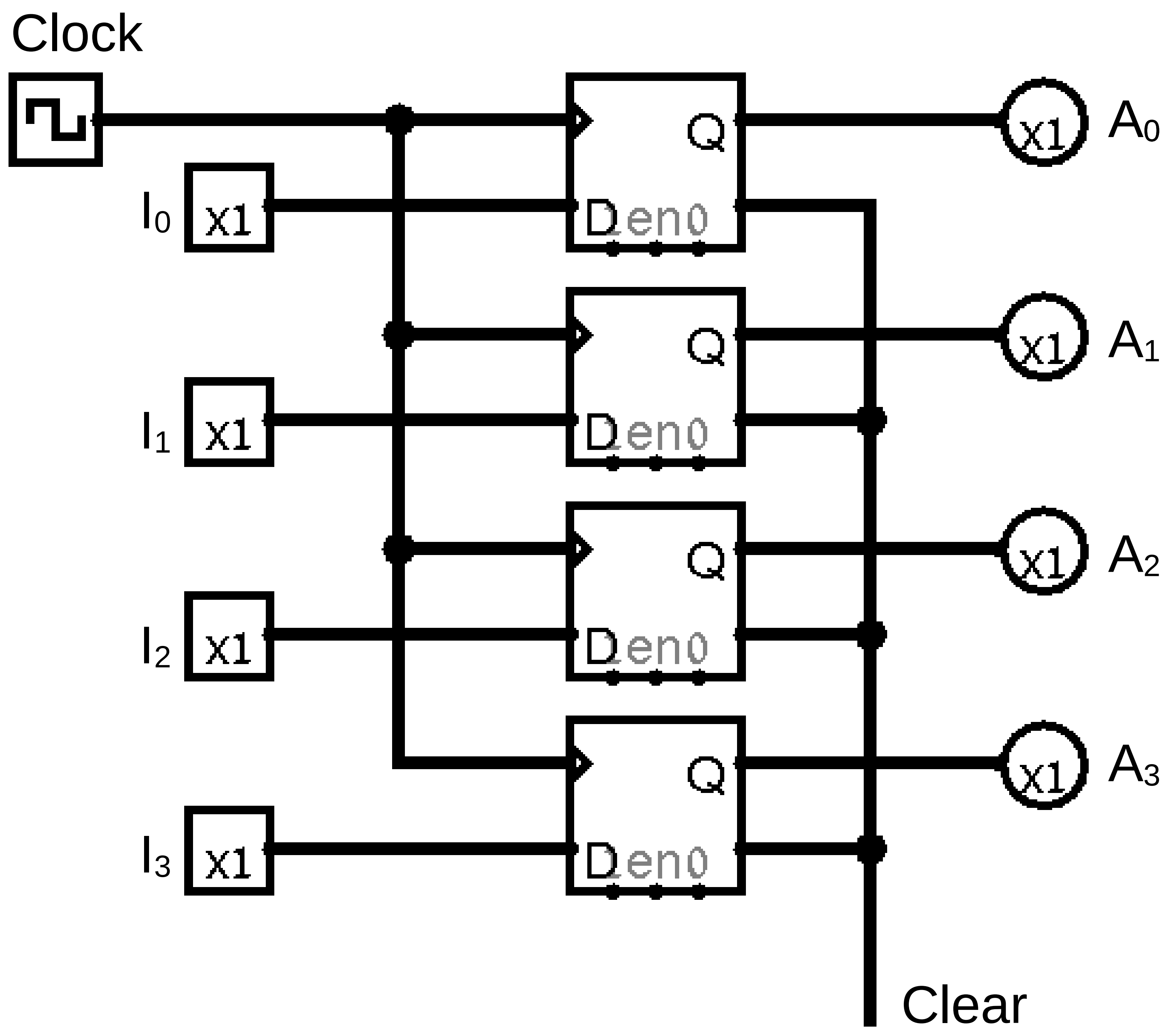


Figure 4
4-bit register

A register is a group of flip-flops with each flip-flop capable of storing one bit of information. An n-bit register has a group of n flip-flops and is capable of storing any binary information of n bits. In addition to the flip-flops, a register may have combinational gates that perform certain data-processing tasks. In its broadest definition, a register consists of a group of flip-flops and gates that effect their transition. The flip-flops hold the binary information and the gates control when and how new information is transferred into the register.

Various types of registers are available commercially. The simplest register is one that consists only of flip-flops, with no external gates. Figure 2-6 shows such a register constructed with four D flip-flops. The common clock input triggers all flip-flops on the rising edge of each pulse, and the binary data available at the four inputs are transferred into the 4-bit register. The four outputs can be sampled at any time to obtain the binary information stored in the register. The clear input goes to a special terminal in each flip-flop. When this input goes to 0, all flip-flops are reset asynchronously. The clear input is useful for clearing the register to all O's prior to its clocked operation. The clear input must be maintained at logic 1 during normal clocked operation. Note that the clock signal enables the D input but that the clear. input is independent of the clock.

The transfer of new information into a register is referred to as loading the register. If all the bits of the register are loaded simultaneously with a common clock pulse transition, we say that the loading is done in parallel . A clock transition applied to the C inputs of the register of Fig . 2-6 will load all four inputs 10 through 13 in parallel. In this configuration, the clock must be inhibited from the circuit if the content of the register must be left unchanged.