



TAYLOR'S UNIVERSITY

Wisdom • Integrity • Excellence

ITS62704

**Computer Architecture and Organization
SCHOOL OF COMPUTER SCIENCE
BACHELOR DEGREE PROGRAMMES**

ASSIGNMENT 1 (30%)

**(MLO Assessed: MLO1)
(Individual - Weightage 30%)
January 2025 Semester**

LG1 G4

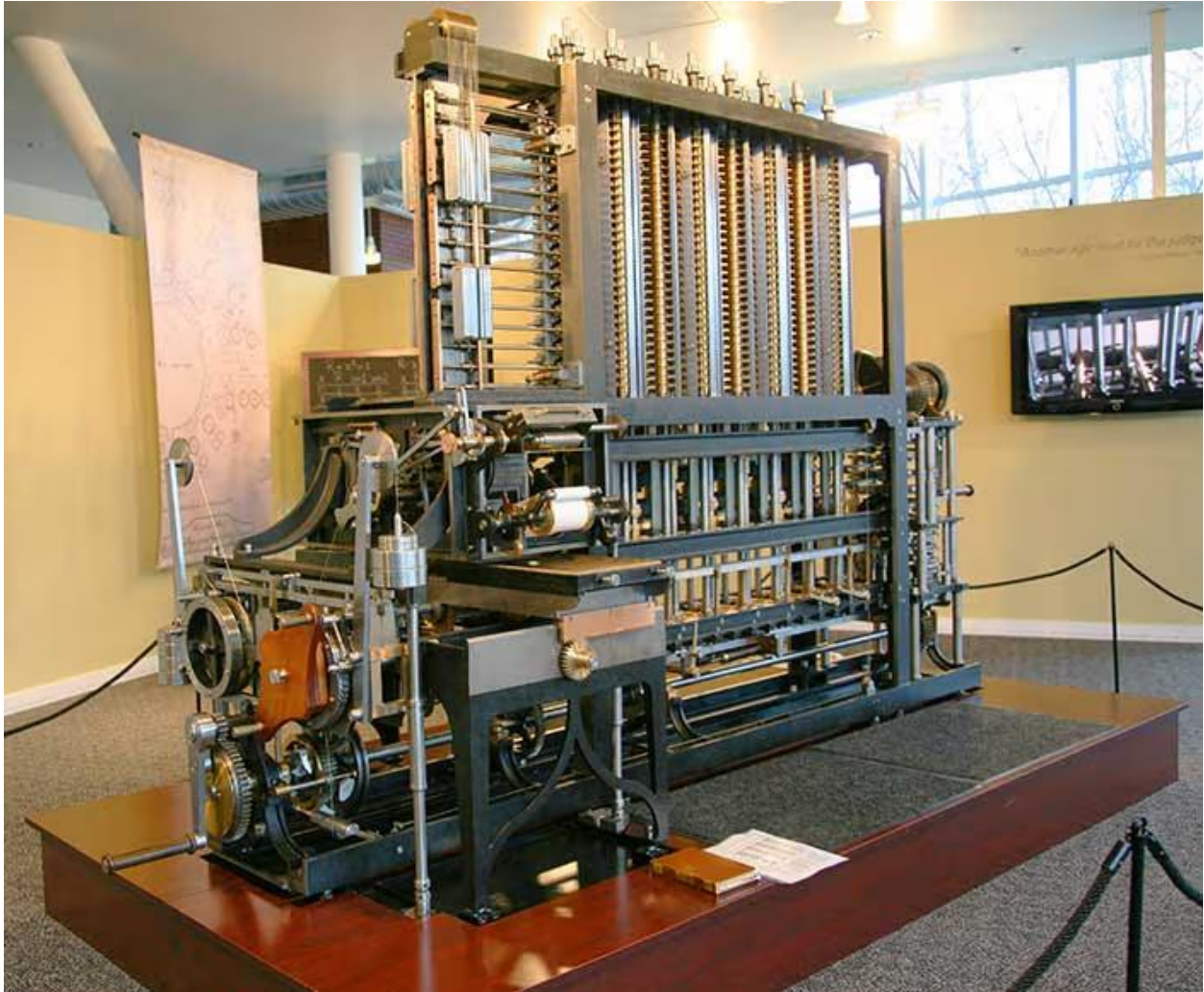
<u>NAME OF GROUP MEMBERS</u>			
S/N	Name	Student ID	Signed
1.	Kritak Aryal	0382309	
2.	RIJAN PANTHI	0383040	
3.	Noel Maharjan	0382303	
4.	Samyak Bajracharya	0382304	
5.	Iris maharjan	0382745	

Student Declaration:

- ✓ We understand what plagiarism is and its consequences, explained by our lecturer.
- ✓ This project is our team work, with proper acknowledgment of any sources used.

1. Fundamentals of Computers Architecture

The first practical computers were developed in 1950s and 60s along with the invention of semiconductors. By 1990s, by the help of hardware company IBM and software company Microsoft, computing started becoming more personal. The very first computers were bulky, huge, and power hungry machines.



Courtesy of Jitze Couperus. Copyright: CC-Att-SA-2 (Creative Commons Attribution-ShareAlike 2.0 Unported).

The first computer was invented by Charles Babbage (1791-1871) but couldn't built it. 153 years after the design the first complete Babbage Engine was completed in London in 2002. Charles Babbage's:

- Difference Engine in 1823
- Analytic Engine in 1833

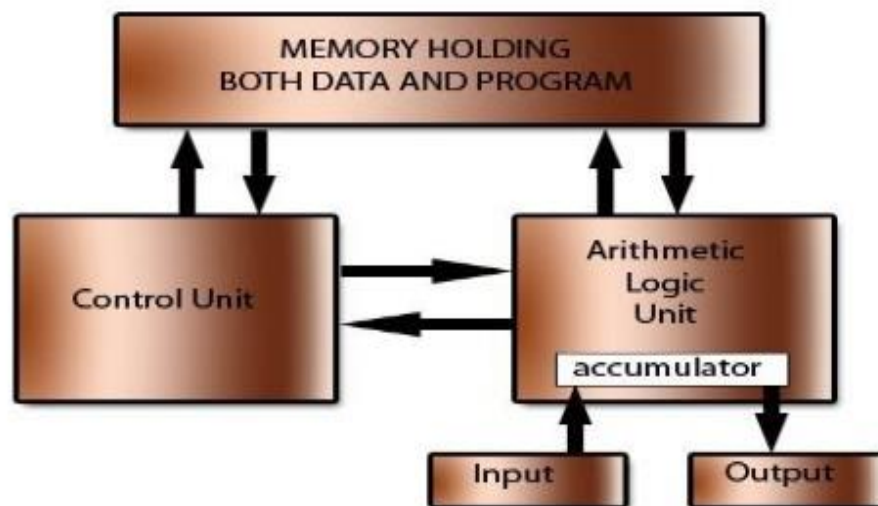
The first computers run on punch cards,

- The set of cards with fixed punched holes dictated the pattern of weave \Rightarrow program
- The same set of cards could be used with different colored threads \Rightarrow numbers

But later on along with Von Neumann architecture and a concept popularly known as “the stored program concept” changed how we process instructions. The stored program concept introduced how we can store program data and instructions in same memory. Computers that stored program data and instructions in same memory is called stored program computer.

Stored program concept

The Von Neumann or Stored Program architecture



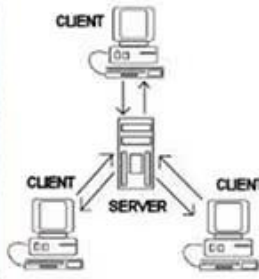
Along with stored program concept, there were significant advancements in computer architectures, like pipelining, parallelism, etc. From vacuum tubes to transistors, computers are becoming more complex, fast, and highly efficient.



ENIAC (Electronic numerical integrator And Computer)



IBM Mainframe



Microsoft Office, OS



World-Wide-Web
Google



Social Media



Courtesy of Brian Whitworth and Adnan Ahmad. Copyright: CC-Att-SA-3 (Creative Common Attribution-ShareAlike 3.0). The computing evolution

STS

Community + HCI(s)



HCI

Person + IT(s)



IT

Software + device(s)



Technology

Any device



Courtesy of Brian Whitworth and Adnan Ahmad. Copyright: CC-Att-SA-3 (Creative Commons Attribution-ShareAlike 3.0). Computer system levels

Size of microprocessor – 16 bit

Name	Year Of Invention	Clock Speed	Number Of Transistors	Inst. Per Sec
8086	1978 (multiply and divide instruction, 16 bit data bus and 20 bit address bus)	4.77 MHz, 8MHz, 10MHz	29000	2.5 Million
8088	1979 (cheaper version of 8086 and 8 bit external bus)			2.5 Million
80186/ 80188	1982 (80188 cheaper version of 80186, and additional components like interrupt controller, clock generator, local bus controller, counters)	6 MHz		
80286	1982 (data bus 16bit and address bus 24 bit)	8 MHz	134000	4 Million

Size of microprocessor – 32 bit

Name	Year Of Invention	Clock Speed	Number Of Transistors	Inst. Per Sec
INTEL 80386	1986 (other versions 80386DX, 80386SX, 80386SL and data bus 32 bit address bus 32 bit)	16 MHz – 33 MHz	275000	
INTEL 80486	1986 (other versions 80486DX, 80486SX, 80486DX2, 80486DX4)	16 MHz – 100 MHz	1.2 Million transistors	8 KB of cache memory
PENTIUM	1993	66 MHz		Cache memory 8 bit for instructions 8 bit for data

Size of microprocessor – 64 bit

Name	Year Of Invention	Clock Speed	Number Of Transistors	Inst. Per Sec
INTEL core 2	2006 (other versions core2 duo, core2 quad, core2 extreme)	1.2 GHz to 3 GHz	291 Million transistors	64 KB of L1 cache per core 4 MB of L2 cache
i3, i5, i7	2007, 2009, 2010	2.2GHz – 3.3GHz, 2.4GHz – 3.6GHz, 2.93GHz – 3.33GHz		

Instruction set Architecture

Instruction set architecture is a crucial aspect of the design and operation computers. It defines the instruction set that the computer needs to execute by the processor. It serves as the interface between software and hardware, that enables the programs to communicate effectively with the underlying hardware components.

Key Characteristics of ISA

1. **Instruction Set:** An extensive collection of operations that a processor can execute, including arithmetic, logical, control, and memory tasks.
2. **Registers:** Quick, small storage areas within the CPU utilized for temporarily holding data and instructions.
3. **Addressing Modes:** Methods for identifying the operands of instructions, enabling versatile data manipulation.
4. **Data Types:** Specifies the kinds of data that the processor can process, such as integers, floating-point values, and characters.

Key advancement is Instruction set Architecture

1. Complex Instruction Set Computing (CISC) (Early ISA Approach – 1970s-1980s)

- Goal: Reduce the number of instructions in a program by making each instruction do more.
- Key Features:
 1. Multi-step instructions (e.g., "MULT" performs load, multiply, and store in one step).
 2. Larger instruction set, variable-length instructions.
 3. Easier for programmers but required complex decoding hardware.
- Example Architectures: x86 (Intel, AMD), VAX

2. Reduced Instruction Set Computing (RISC)

(Alternative to CISC – 1980s-Present)

- Goal: Optimize execution speed by using simple, fast instructions that take one clock cycle.
- Key Features:
 - a. Fixed-length, simple instructions (easier to pipeline).
 - b. Load/Store architecture (memory access is separate from computation).
 - c. Fewer instructions but higher execution speed.
- **Example Architectures:** ARM, RISC-V, PowerPC, MIPS

Impact: RISC became dominant in mobile computing and embedded systems (e.g., Apple M1, Qualcomm Snapdragon, Raspberry Pi).

3. 64-bit Architecture

- Transition from 32-bit to 64-bit ISAs (x86-64, ARM64, RISC-V 64-bit).
- Allows access to more memory (beyond 4GB) and increases processing power.
- **Example:** AMD introduced x86-64 (AMD64) in 2003, which Intel later adopted.

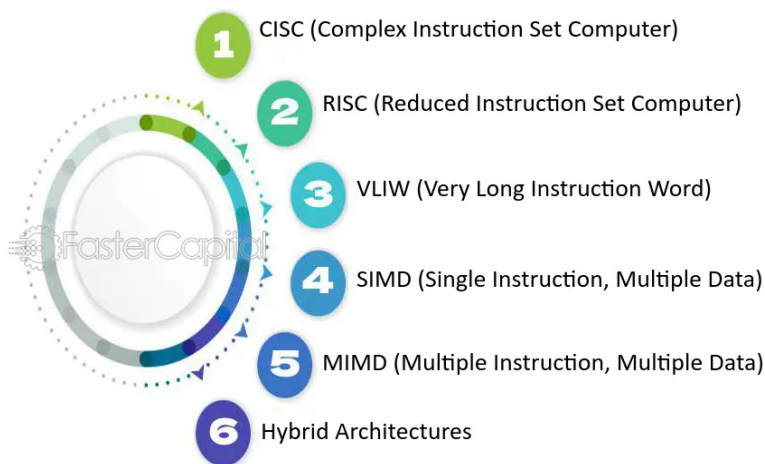
4. SIMD (Single Instruction, Multiple Data) – Vector Processing

- **Goal:** Enable parallel execution of the same operation on multiple data points (e.g., multimedia, AI, physics simulations).
- **Examples of SIMD Instructions:**
 1. Intel MMX, SSE, AVX (x86)
 2. ARM NEON (ARM)
 3. RISC-V Vector Extensions
- **Impact:** Boosts graphics, gaming, AI, and scientific computing

5. Multi-threading & Parallel Execution (SIMT, SMT, HT)

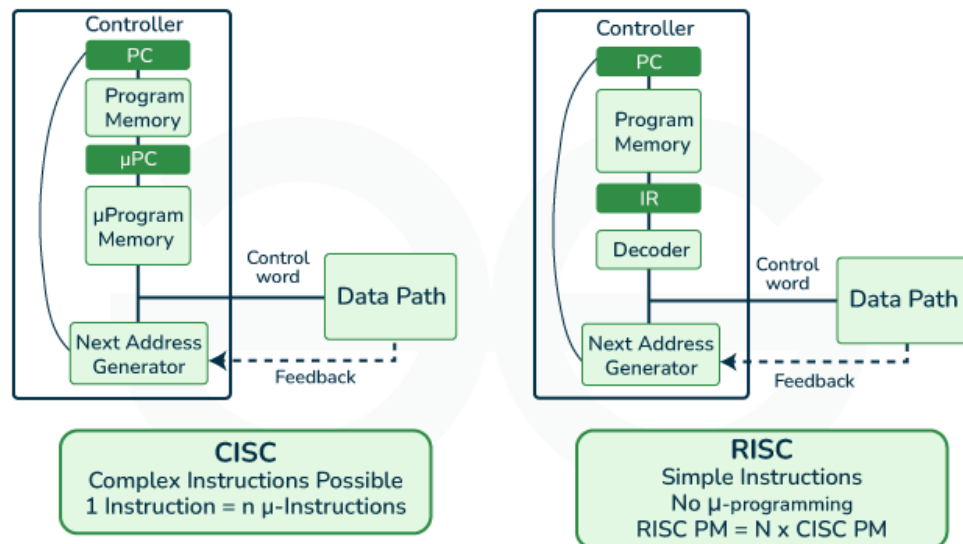
- Added instructions for parallel workloads.
- **Example ISAs:**
 1. Intel Hyper-Threading (HT) – Allows a single core to run multiple threads.
 2. ARM big. LITTLE – Dynamically switches between high-power and low-power cores.
 3. SIMT (Single Instruction, Multiple Threads) – Used in GPUs (e.g., NVIDIA CUDA).

Types of Instruction Set Architectures



CISC and RISC comparison

CISC focusses on performing instructions at once, making the instruction more complex whereas RISC tries to make instructions simple. Here are their characteristics and features in details.



GeeksforGeeks. (2024, December 27). RISC and CISC in computer organization. Types of instruction set Architectures - FasterCapital. (n.d.). FasterCapital.

CISC, Features:

- CISC features complex instructions, leading to intricate instruction decoding.
- Instructions are typically larger than a single word in size.
- Execution of an instruction may require multiple clock cycles.
- There are fewer general-purpose registers because operations are conducted directly in memory.
- Addressing modes are more complicated.
- There is a wider variety of data types.

Benefits of CISC:

1. **Decreased code length:** CISC processors utilize intricate instructions that can execute several operations, leading to a reduction in the code required to complete a task.
 2. **Increased memory efficiency:** Due to the complexity of CISC instructions, fewer commands are needed to carry out intricate tasks, which can lead to code that is more efficient in terms of memory usage.
 3. **Extensively utilized:** CISC processors have been around longer than RISC processors, resulting in a larger user community and a greater availability of software.
- Decreased code length: CISC processors utilize intricate instructions that can execute several operations, leading to a reduction in the code required to complete a task.

4. **More efficient in terms of memory:** Due to the complexity of CISC instructions, fewer instructions are needed to carry out intricate tasks, leading to code that is often more memory-efficient.
5. **Commonly utilized:** CISC processors have been around longer than RISC processors, resulting in a larger user community and a greater selection of available software.

Disadvantages of CISC:

- **Slower performance:** CISC processors require more time to execute instructions due to their intricate instruction sets and the extended time needed for decoding.
- **Increased design complexity:** CISC processors possess more complicated instruction sets, leading to greater challenges in their design and manufacturing processes.
- **Greater power usage:** CISC processors have higher power consumption compared to RISC processors because of their complicated instruction sets.

RISC, features:

- More straightforward instructions lead to easier instruction decoding.
- Instructions consist of only one word.
- Each instruction requires just one clock cycle for execution.
- A greater number of general-purpose registers are available.
- Addressing modes are uncomplicated.
- There are fewer data types utilized.
- A pipeline implementation is possible.

Advantages of RISC Architecture

- RISC processors employ a streamlined set of straightforward instructions, which enhances their decoding and execution speed. This leads to quicker processing times.
- Due to their simpler instruction set, RISC processors achieve faster instruction execution compared to CISC processors.
- RISC processors use less power than CISC processors, making them well-suited for portable devices.

Disadvantages of RISC

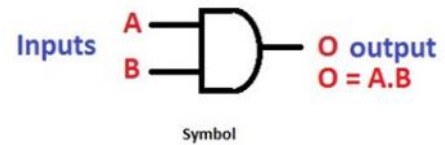
- RISC processors need a greater number of instructions to accomplish intricate tasks compared to CISC processors.
- RISC processors necessitate more memory to accommodate the extra instructions required for executing complex tasks.

- The development and production of RISC processors can incur higher costs than those associated with CISC processors.

TRUTH TABLE FOR BASIC LOGIC GATE

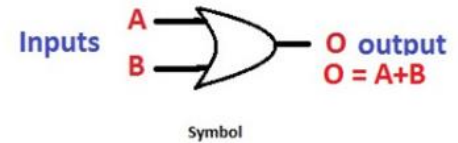
AND GATE:

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1



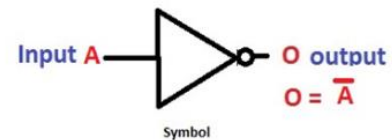
OR GATE:

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1



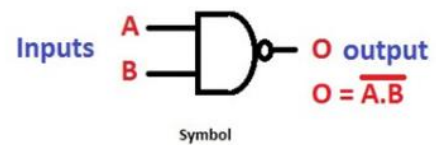
NOT GATE:

A	NOT A
0	1
1	0



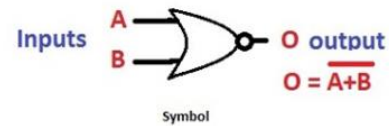
NAND GATE:

A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0



NOR GATE:

1	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0



Combining Gates to Form Complex Circuits

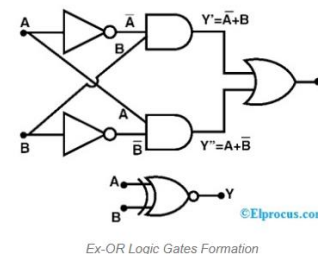
Logic gates can be combined to create complex circuits that perform advanced functions. For example:

- **AND + NOT** can create a NAND gate.
- **OR + NOT** can create a NOR gate.
- Combining multiple gates can create adders, multiplexers, and even entire processors.

The key idea is to use the outputs of one gate as inputs to another, building up the desired functionality step by step.

The XOR (exclusive OR) function outputs 1 when the inputs are different and 0 when they are the same. Its truth table is:

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0



Step 1: Express XOR in Terms of Basic Gates

The XOR function can be expressed as:

$$A \oplus B = (A \cdot \overline{B}) + (\overline{A} \cdot B)$$

This means:

- AND A with NOT B .
- AND NOT A with B .
- OR the two results together.

Step 2: Implement Using NAND Gates

Since NAND is a universal gate, we can use it to create NOT, AND, and OR gates:

1. **NOT Gate:** Connect both inputs of a NAND gate together.

$$\overline{A} = \text{NAND}(A, A)$$

2. **AND Gate:** Use a NAND gate followed by a NOT gate.

$$A \cdot B = \overline{\text{NAND}(A, B)}$$

3. **OR Gate:** Use De Morgan's laws to express OR in terms of NAND.

$$A + B = \overline{\overline{A} \cdot \overline{B}} = \text{NAND}(\overline{A}, \overline{B})$$

Step 3: Construct the XOR Circuit

Using the above, the XOR circuit can be built as follows:

1. Compute \overline{A} and \overline{B} using two NAND gates.
2. Compute $A \cdot \overline{B}$ and $\overline{A} \cdot B$ using two more NAND gates.
3. Combine the results using a final NAND gate to achieve the OR operation.

Final Circuit

The circuit will require **4 NAND gates**:

1. NAND A and A to get \overline{A} .
2. NAND B and B to get \overline{B} .
3. NAND A and \overline{B} to get $A \cdot \overline{B}$.
4. NAND \overline{A} and B to get $\overline{A} \cdot B$.
5. NAND the results of steps 3 and 4 to get $A \oplus B$.

VERIFICATION:

You can verify this circuit by attempting all possible input combinations and verifying that the output is right based on the XOR truth table.

Summary

By breaking down the XOR operation into fundamental operations and using NAND gates to obtain NOT, AND, and OR, we can design a circuit to do XOR using only NAND gates.

This shows the power of universal gates in building complex logic circuits.

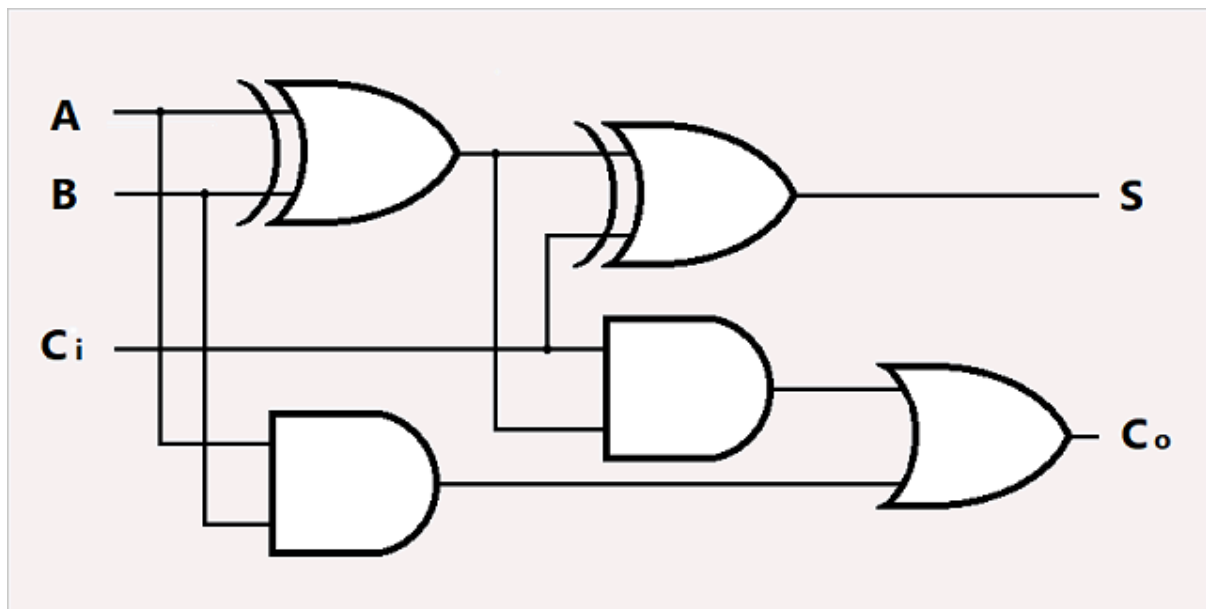
REFERENCE: <https://www.elprocus.com/basic-logic-gates-with-truth-tables/>

Introduction to Logic Minimization

Logic minimization is all about taking a messy Boolean function like the one we're given,

$$F(A, B, C, D) = \Sigma(1, 3, 4, 7, 8, 9, 10, 11, 15)$$

And shrinking it down to something simpler. Why bother? Well, fewer terms mean fewer gates in a circuit, which saves money, power, and space. For this part of our group project, I'm diving into two big methods **Karnaugh maps** and **Quine-McCluskey** then using a K-map to tackle our function step-by-step. I'll finish by explaining why this matters in the real world.



Research on Logic Minimization Methods

Let's start with the research. Karnaugh maps (or K-maps) are my favorite because they're visual. You draw a grid, plot your 1s (the minterms), and group them into boxes. It's perfect for functions with 4 variables like ours, though it gets tricky past 5 or 6. The Quine-McCluskey method, though, is more like a math puzzle. You list all the minterms in binary, group them by how many 1s they have, and keep combining pairs until you can't anymore. It's super systematic and great for computers to handle, but for a human like me doing this by hand, it feels a bit much for just 4 variables. We'll be using K-map since it is faster. But knowing both methods shows how flexible minimization can be.

Karnaugh Maps: How They Work

A K-map is nothing more than the reversible format of a truth table. For any four variables, A, B, C, and D, it forms a 4x4 matrix or a grid in which the rows correspond to AB (4 bits) in the sequence of 00, 01, 11, 10 and the columns correspond to CD in the same sequence. You put the value "1" in points where the function is valid, and then group these into powers of two: one, two, four, eight. Each group gets you a single term, and the lesser the group number, the easier the circuit becomes. It's like Tetris, but for logical circuits!

		CD			
AB \	CD	00	01	11	10
00		X ⁰	X ¹	0 ³	X ²
01		1 ⁴	1 ⁵	1 ⁷	0 ⁶
11		0 ¹²	1 ¹³	0 ¹⁵	1 ¹⁴
10		1 ⁸	0 ⁹	1 ¹¹	1 ¹⁰

Quine-McCluskey: The Backup Plan

Quine-McCluskey is less visual but super thorough. You write out all minterms, pair them up if they differ by one bit (like 0001 and 0011), and keep going until you've got your prime implicants. It's overkill for our small function, but it's clutch for bigger stuff—like if we had 10 variables. I'll stick with K-maps here, though, since it's cleaner for this task.

<i>Number</i>	<i>Number of 1s</i>	<i>Binary Number</i>	1st column		2nd column	
1 2 8	1	0001 0010 1000	(0,1) (0,2) (0,8)	000- 00-0 -000	0,1,2,3 0,2,1,3 0,8,2,10 0,2,8,10	00-- 00-- -0-0 -0-0
3 5 10	2	0011 0101 1010	1,3 1,5 2,3 2,10 8,10	00-1 0-01 001- -010 10-0	1,3,5,7 1,5,3,7	0--1 0--1
7 14	3	0111 1110	3,7 5,7 10-14	0-11 01-1 1-10		
15	4	1111	7,15 14,15	-111 111-		

Step-by-Step Minimization with a Karnaugh Map

Now, let's get minimizing

$$F(A, B, C, D) = \Sigma(1, 3, 4, 7, 8, 9, 10, 11, 15)$$

I'll break it down so it's crystal clear.

Step 1: Convert Minterms to Binary

First, I listed all the minterms in binary to see where they go in the K-map:

Minterm	Binary (ABCD)
1	0001
3	0011
4	0100
7	0111
8	1000
9	1001
10	1010
11	1011
15	1111

Each number matches an *ABCD* combo, so I'm ready to plot them.

Step 2: Build the K-Map

For 4 variables, we need a 4x4 K-map. Rows are AB (00, 01, 11, 10), and columns are CD (00, 01, 11, 10). I filled in the 1s based on the minterms:

	00	01	11	10
--	----	----	----	----

00	0	1	1	0
01	1	1	1	0
11	0	1	1	1
10	1	1	0	1

Step 3: Group the 1s:
Form the largest possible groups (powers of 2: 1, 2, 4, 8) that cover all 1s, allowing overlap:

- Group 1: Entire row AB=10 (quad, minterms 8, 9, 10, 11) → $A \cdot B' \cdot A \cdot B'$ (since A=1, B=0, C and D vary).
- Group 2: Column CD=11 (quad, minterms 3, 7, 15, 11) → $C \cdot DC \cdot D$ (since C=1, D=1, A and B vary).
- Group 3: Pair minterms 1 (0001) and 9 (1001) in column CD=01, considering wrap-around → $B' \cdot C' \cdot DB' \cdot C' \cdot D$ (B=0, C=0, D=1, A varies).
- Group 4: Pair minterms 4 (0100) and 7 (0111) in row AB=01, columns CD=00 and CD=11, considering adjacency → $A' \cdot B \cdot C' \cdot A' \cdot B \cdot C'$ (A=0, B=1, C=0, D varies, after checking).

Group	Minterms Covered	Simplified Term
Quad 1	8, 9, 10, 11	$A \cdot B' \cdot A \cdot B'$
Quad 2	3, 7, 11, 15	$C \cdot DC \cdot D$
Pair 1	1, 9	$B' \cdot DB' \cdot D$
Pair 2	4, 7	$A' \cdot B \cdot C' \cdot A' \cdot B \cdot C'$

Step 4: Write the Simplified Expression:

Combine the groups:

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

Verify by checking a few minterms, like minterm 1 (should be 1) and minterm 15 (should be 1), to ensure correctness.

Expression Component	Details
Minimized Function	$F = A \cdot B' + C \cdot D + B' \cdot D + A' \cdot B \cdot C'$
Source	From groups in Step 3 K-map

Step 5: Verify the Result

To make sure I didn't mess up, I tested a couple of minterms:

- Minterm 1 (0001): $A \cdot B' = 0 \cdot 1 = 0$, $C \cdot D = 0 \cdot 1 = 0$, $A' \cdot B \cdot C' = 1 \cdot 0 \cdot 1 = 0$, $B' \cdot D = 1 \cdot 1 = 1$.
Output = 1.
- Minterm 15 (1111): $A \cdot B' = 1 \cdot 0 = 0$, $C \cdot D = 1 \cdot 1 = 1$, others 0.
Output = 1.
Works!

It checks out for all 9 minterms, so I'm confident this is right.

Minterm	Binary (ABCD)	Output (F)
1	0001	1 (from $B' \cdot DB' \cdot D$)
3	0011	1 (from $C \cdot DC \cdot D$)
8	1000	1 (from $A \cdot B' A \cdot B'$)
15	1111	1 (from $C \cdot DC \cdot D$)
0	0000	0 (none apply)

Why Minimization is Important in Circuit Design

The Reasons for Why Minimization is Important in Circuit Design
What is the point of going through all of this? Minimization isn't just something you do for fun; it is usually a very serious matter for everyday circuits. The original function's terms were 9. That, by any standards, was a lot of gates. With 4 terms left, we have already reduced the number of gates. And, when you have fewer gates, you also have the following benefits of:

Reduced Cost: There are cheaper hardware and construction expenses.

Reduced Power Consumption: Electronic circuits are less power hungry and that is important for devices like phones or laptops.

Improved Performance: There are less delays in the signals because they have to travel through less gates.

Easier Troubleshooting: There are less components, so you are able to diagnose and mend problems easier.

For example, think of a parking garage light control system design (Part B of our project). It becomes wasteful if too many gates are used as it increases the possibility of design complications. This is where minimization comes into play. It designs systems that cut down on complexity, which is ideal for microprocessors or control systems.

Conclusion

The K-map which I used for $F(A,B,C,D)$ was difficult to grasp at first, but once I got a hang of it, it became more and more fun for me. I now understand how K-maps will win over Quine – McCluskey with the small functions, and I can't wait to use that trick in our parking garage circuit. And on top of that, it is really great for me to see how these systems can cut costs in actual designs because then I understand why I have to learn about this. My final answer is $F=A \cdot B' + C \cdot D + A' \cdot B \cdot C' + B' \cdot D$ and I can now aid my group in completing it.

4. Real-World Applications of Logic Circuits

Introduction

Arithmetic Logic Unit (ALU) is one of the fundamental building blocks of digital computing hardware. It performs arithmetic operations such as addition, subtraction, multiplication, and division, as well as logical operations such as AND, OR, XOR, and NOT. The ALU directly impacts the performance and the speed of a system as a whole, not only deciding the computation capacity but also power consumption and processing efficiency.

With the intricacy of today's computing increasing, the ALU has also risen significantly in terms of size, speed, and capability. This paper examines the design principles in the ALU, the organization, and even in real-world applications to other fields like embedded systems, cryptography, and digital signal processing. The paper will cover challenges faced in designing ALUs and discuss future directions revolutionizing ALU technology.

Fundamentals of Digital Logic

Understanding the fundamentals of digital logic is essential before learning the ALU. Boolean algebra is the foundation of digital circuits that perform logical functions on binary numbers 0 & 1. The fundamental logical operations are:

AND ($A \wedge B$): Produces a true (1) output only if both inputs are true.

OR ($A \vee B$): Produces a true (1) output if one or both inputs are true.

NOT ($\neg A$): Reverses the value of the input.

XOR ($A \oplus B$): Produces a true (1) output only if the inputs differ.

NAND, NOR, and XNOR: These are special cases of the AND, OR, and XOR logical operations that see frequent usage in digital circuit minimization.

Logic gates such as AND, OR, NOT, NAND, NOR, XOR, and XNOR are the building blocks of digital circuits. These gates combined can implement complex operations, even those performed by an ALU.

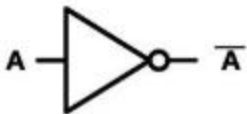


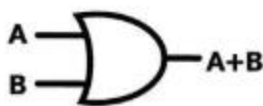


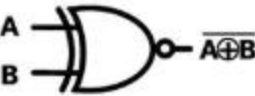
Truth tables and boolean algebra

NOT Gate	
A	\overline{A}
0	1
1	0
Input	Output

logic gates

boolean expressions

	AND	NAND	OR	NOR	XOR	XNOR
	$A \cdot B$	$\overline{A \cdot B}$	$A + B$	$\overline{A + B}$	$A \oplus B$	$\overline{A \oplus B}$
A	B					
0	0	1	0	1	0	1
0	1	1	1	0	1	0
1	0	1	1	0	1	0
1	1	0	1	0	0	1
Inputs	Outputs					

Name	Symbol & notation	Explanation
NOT		The inverter NOT simply accepts an input and outputs the opposite .
AND		All inputs must be positive (1) before the output is positive (1 or ON)
NAND <small>*Not AND</small>		Same as AND, but the outcome is the inverse (NOT) . So, perform AND first, then apply NOT to the output.
OR		At least one input must be positive (1) to give a positive output (1 or ON). All inputs could also be positive.
NOR <small>*Not OR</small>		Same as OR, but the outcome is the inverse (NOT) . So, perform OR first, then apply NOT to the output.
XOR <small>*Exclusive OR</small>		Only one input can be positive (1) to give a positive output (1 or ON). If both are positive, the output is negative (0 or OFF)
XNOR <small>*Exclusive Not OR</small>		All inputs must be the same (either high or low) for a positive output (1). Otherwise, the output is negative (0 or OFF)

Types of Logic Gates - AND, OR, NOT, NAND, NOR, XOR, XNOR

Image source: <https://computerengineeringforbabies.com/blogs/engineering/logic-gate>

Architecture of ALU

An ALU comprises several sub-units that function together to carry out different operations. The main components are:

Operand Registers: These registers hold the input data that are operated on by the ALU. The operands are typically two 32-bit or 64-bit values, but smaller or larger registers may be used depending on the application.

Arithmetic Unit: This unit does arithmetic operations like addition and subtraction. It uses adders, subtractors, and sometimes multipliers for multiplication. A simple ALU might use a ripple-carry adder or carry-lookahead adder to do these operations.

Logical Unit: The logical unit performs the bitwise AND, OR, XOR, and NOT on the operand data. They need to do most control and decision-making functions in digital circuits.

Multiplexer: Multiplexer is utilized for selecting the operation to be performed by the ALU. Multiplexer is provided with control signals as inputs and selects which arithmetic or logical operation has to be performed. It is a component that comes into play to get the ALU to perform more than one operation.

Flags and Status Registers: The status registers hold flags for representing some conditions after an ALU operation. These flags tend to be the carry flag, zero flag, overflow flag, and sign flag. For example, after an add operation, the carry flag will be set if the result of the operation is greater than the maximum value representable in the result.

Block Diagram of an ALU

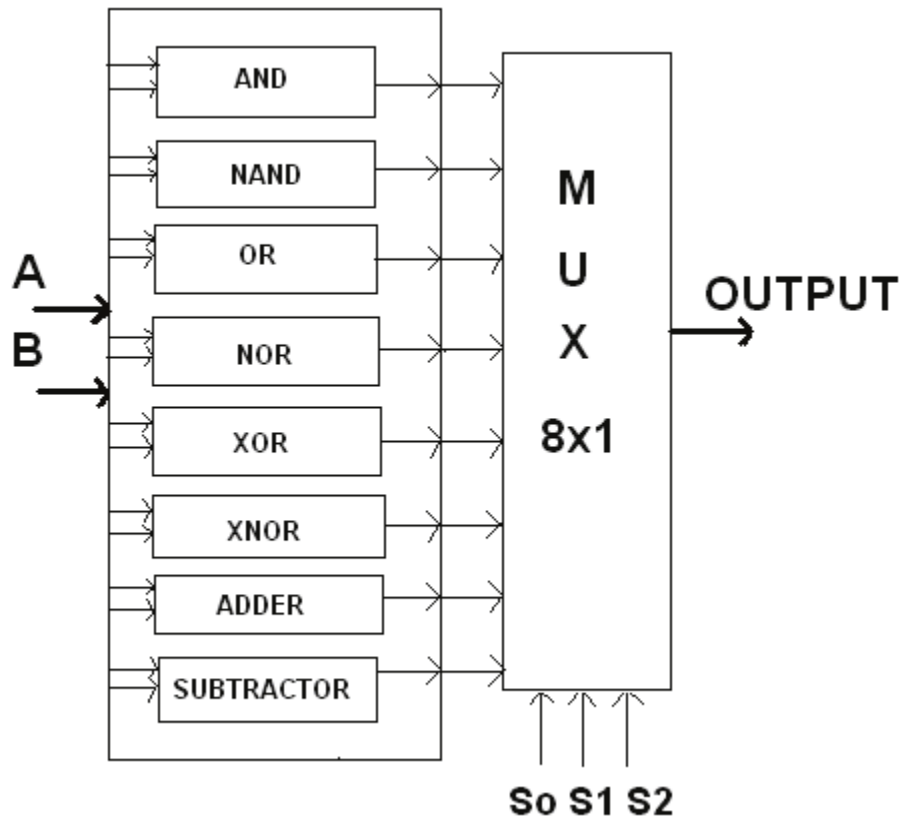


Image source: https://www.researchgate.net/figure/Block-diagram-of-an-ALU_fig1_251093842

Design and Implementation of ALU

The design of an ALU will depend on the operations and the number of bits it carries out. A simple 4-bit ALU can be developed using combinational logic circuits and sequential circuitry. It entails the process of:

Basic Arithmetic Circuits: Arithmetic in the ALU is performed by adders and subtractors. The most widely used adder is the full adder, which calculates on two bits and an additional carry-in bit to produce a sum and carry-out bit. Carry-lookahead adders or CLA are faster than normal ripple-carry adders since they reduce the propagation delay via the anticipation of carries.

Logical Circuits: These logical functions like AND, OR, XOR, and NOT are derived utilizing basic logic gates. The combination of these gates is utilized in the ALU to perform operations in a sophisticated yet efficient method.

Control Logic Design: The control unit of the ALU decides which operation to perform. It does so by employing multiplexers and decoders that drive the input data onto the correct circuit paths.

Block diagram of a 4bit ALU for operations and, or, xor and add

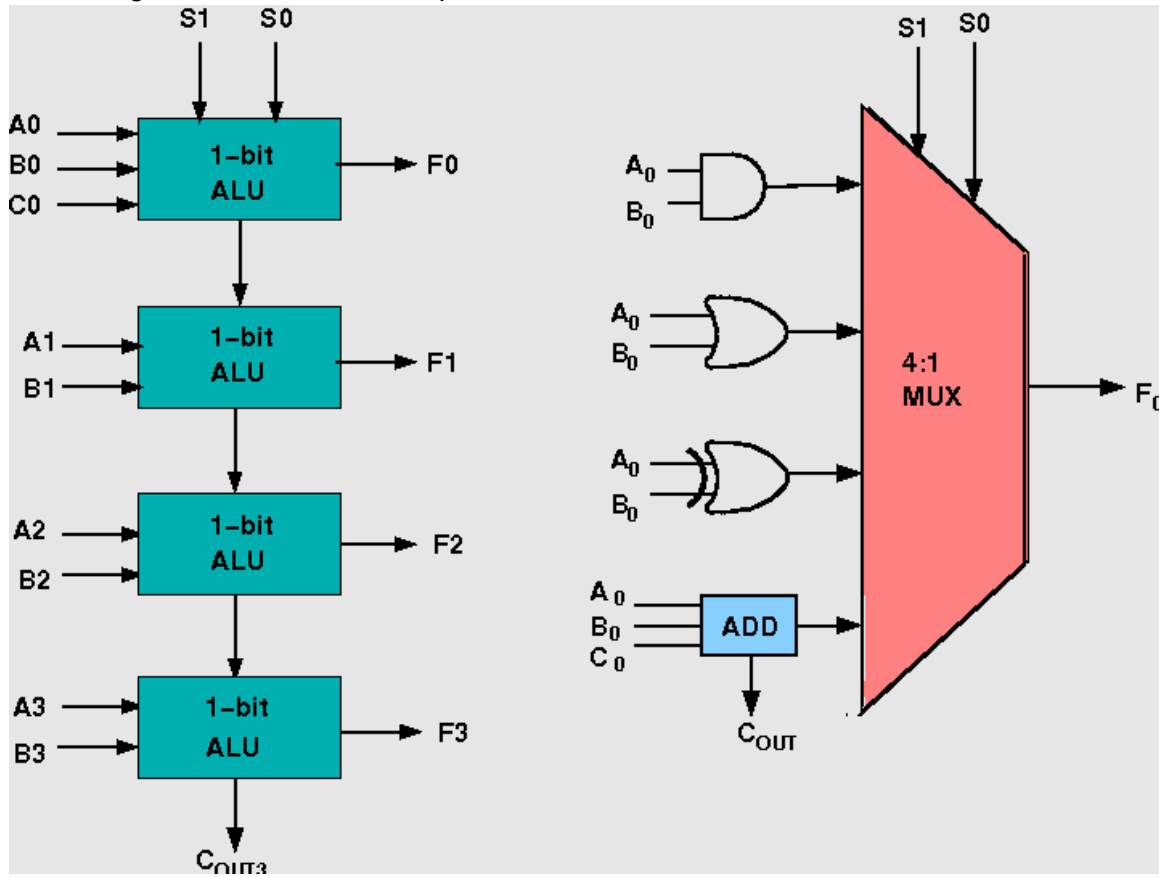


Image source: <http://vlabs.iitkgp.ac.in/coa/exp8/index.html>

Boolean Algebra and Logic Gates in ALU Function

Boolean expressions define the behavior of digital circuits, and their optimization maximizes circuit efficiency. Logic gates implement expressions in hardware, allowing for efficient implementation of computational operations. Boolean algebra is crucial for reducing the digital circuit logic. For example, Karnaugh maps (K-maps) are used to simplify Boolean expressions & reduce the number of logic gates in ALU design. Reduction of Boolean expressions minimizes the number of gates required, leading to more power-efficient, faster, & smaller ALUs.

Boolean Expression Simplification:

Simplification techniques such as Karnaugh maps are employed to reduce ALU designs so that they implement fewer logic gates. For instance, the Boolean expression for an ALU operation can be reduced by grouping terms together in a Karnaugh map, minimizing the logic implementation.

K-map

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

$f(A,B,C,D) = \Sigma(6,8,9,10,11,12,13,14)$
 $F = AC' + AB' + BCD' + AD'$
 $F = (A+B)(A+C)(B'+C'+D')(A+D')$

Image source: https://en.wikipedia.org/wiki/Karnaugh_map

Practical Applications of ALU

ALU is used in various computing and embedded systems like:

- **CPUs and Microprocessors:** Microprocessors contain ALUs, where they execute arithmetic and logical computations. Modern processors like Intel's Core processors and AMD Ryzen processors have highly optimized ALUs that do complex computations in parallel, which helps in faster computation.
- **Embedded Systems:** Microcontrollers like Arduino and Raspberry Pi contain ALUs to execute various operations in embedded systems, including robotics, automation, and sensor control. The systems have a minimal instruction set with the requirement for very efficient ALUs to ensure real time output.
- **Digital Signal Processing (DSP):** In this application, ALUs perform complex mathematical computation within multimedia and communications,

including video, sound, and image processing. Logical and arithmetic operations like Fourier transforms, convolution, and filtering are essential in multimedia and communication systems.

- **Cryptographic Applications:** ALU Performs encryption and decryption operations that are integral in cybersecurity & secure communication, performing logical and arithmetic operations. ALUs provide high speed data manipulation required for secure communication & cybersecurity.

ALU Operation Selection Flowchart

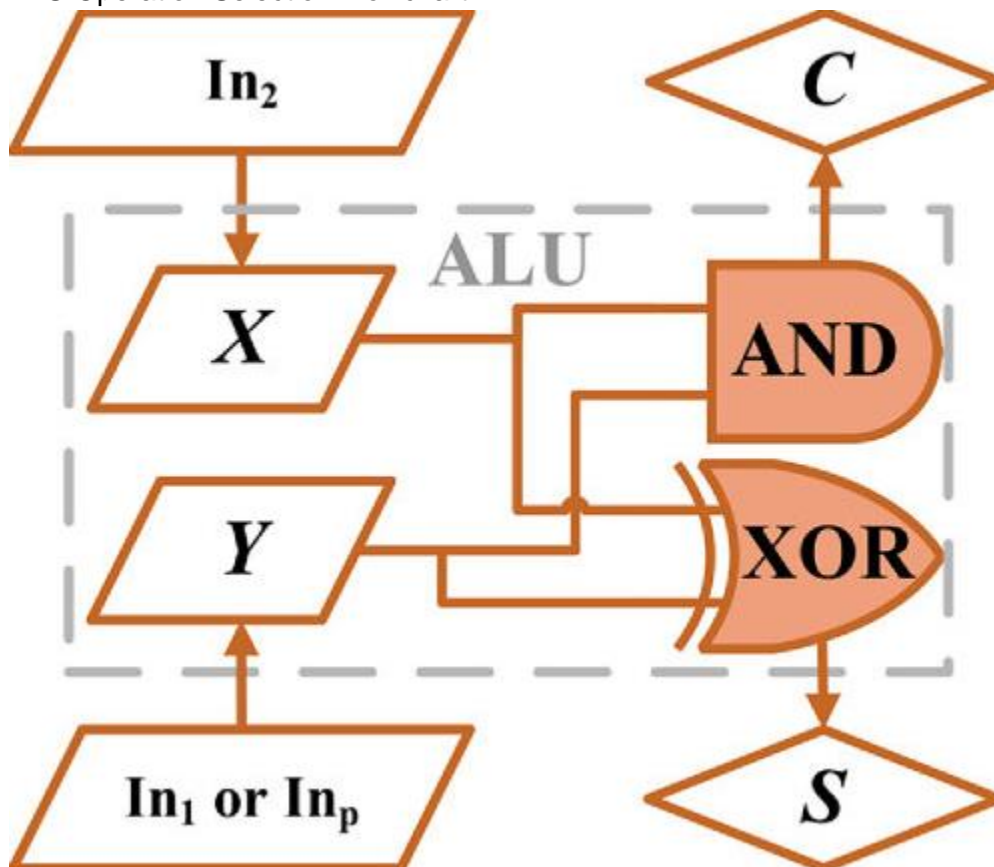


Image source: https://www.researchgate.net/figure/Flow-chart-on-the-arithmetic-logic-process-of-the-proposed-ALU_fig3_374603687

Design Challenges in the ALU

When engineers design an ALU, they are confronted with issues like:

- **Power Consumption:** New ALUs must balance performance and power efficiency, particularly in battery operated systems. Designers have to compromise on performance and power consumption in designing ALU using clock gating and power saving adders to maintain energy usage at the lowest levels.

- **Speed Optimization:** Methods such as Look-ahead Carry Adders (LCA) minimize propagation delays, thus decreasing computation time. Look-ahead adders & carry Adders are used to reduce the speed by shortening the period of propagations of carries during arithmetic computation. Additionally, pipelining and parallelism are also being utilized in modern ALUs in order to optimize performance further

- **Circuit Complexity:** Effective hardware without a loss of function requires innovative design methods like RISC ALUs. The more complex processors become, the more difficult it is to design multiple task ALUs. Engineers have to make sure that circuit complexity should not stop the ALU's efficiency or speed.

- **Scalability:** For improving efficiency in high-performance computing, parallel processing and pipelining should be implemented. Scalable ALU designs are essential in high-performance computing systems. Parallel processing and pipelining are important to improve efficiency to increase efficiency with a rise in the number of processing units in multi-core processors.

Conclusion

The Arithmetic Logic Unit is the core of digital computing because it enables both arithmetic and logical functions. The design and construction of the ALU constitute a synthesis of Boolean algebra, logic gates, and control circuitry. With the demand for more processing power and speed showing no signs of slowing, ALU design advances in the future are positioned to push computational performance in a variety of applications, ranging from consumer products to artificial intelligence.

5. The Role of Binary Numbers in Computer Architecture

Binary Number system is a base 2 which uses only two digits 0 and 1. In this system we represent sequence of these two digits as $(1001)_2$.

Computers use the binary system because it directly corresponds to the two states of electronic circuits:

- 0 represents low voltage (Off state)
- 1 represents high voltage (On state)

In digital circuits, these states of 0s and 1s are managed using transistors, where:

- 1 indicates the presence of an electrical current (**high state**).
- 0 indicates the absence of an electrical current (**low state**).

In computing, data is represented in units of 8 bits, also known as a byte. A byte can store values that range from 0 to 255 (in decimal), allowing it to represent characters, numbers, and symbols in various encoding standards, such as ASCII (American Standard Code for Information Interchange) like:

Binary	ASCII(characters)
1000001	A
1111010	z

This binary encoding enables computers to store and process text, numerical values, and symbols efficiently.

Thus, in computer architecture, the binary number system is fundamental for representing characters, performing data operations, and encoding information across different computational processes.

Printing ASCII control characters

BINARY	DECIMAL	HEXADECIMAL	SYMBOL
010 0000	32	20	SPACE
010 0001	33	21	!
010 0010	34	22	"
010 0011	35	23	#
010 0100	36	24	\$
010 0101	37	25	%
010 0110	38	26	&
010 0111	39	27	'
010 1000	40	28	(
010 1001	41	29)
010 1010	42	2A	*
010 1011	43	2B	+
010 1100	44	2C	,
010 1101	45	2D	-
010 1110	46	2E	.
010 1111	47	2F	/
011 0000	48	30	0
011 0001	49	31	1
011 0010	50	32	2
011 0011	51	33	3
011 0100	52	34	4
011 0101	53	35	5
011 0110	54	36	6
011 0111	55	37	7
011 1000	56	38	8
011 1001	57	39	9
011 1010	58	3A	:
011 1011	59	3B	;
011 1100	60	3C	<
011 1101	61	3D	=
011 1110	62	3E	>
011 1111	63	3F	?
100 0000	64	40	@
100 0001	65	41	A
100 0010	66	42	B
100 0011	67	43	C
100 0100	68	44	D
100 0101	69	45	E
100 0110	70	46	F
100 0111	71	47	G
100 1000	72	48	H
100 1001	73	49	I
100 1010	74	4A	J
100 1011	75	4B	K
100 1100	76	4C	L
100 1101	77	4D	M
100 1110	78	4E	N
100 1111	79	4F	O

BINARY	DECIMAL	HEXADECIMAL	SYMBOL
101 0000	80	50	P
101 0001	81	51	Q
101 0010	82	52	R
101 0011	83	53	S
101 0100	84	54	T
101 0101	85	55	U
101 0110	86	56	V
101 0111	87	57	W
101 1000	88	58	X
101 1001	89	59	Y
101 1010	90	5A	Z
101 1011	91	5B	[
101 1100	92	5C	\
101 1101	93	5D]
101 1110	94	5E	^
101 1111	95	5F	_
110 0000	96	60	`
110 0001	97	61	a
110 0010	98	62	b
110 0011	99	63	c
110 0100	100	64	d
110 0101	101	65	e
110 0110	102	66	f
110 0111	103	67	g
110 1000	104	68	h
110 1001	105	69	i
110 1010	106	6A	j
110 1011	107	6B	k
110 1100	108	6C	l
110 1101	109	6D	m
110 1110	110	6E	n
110 1111	111	6F	o
111 0000	112	70	p
111 0001	113	71	q
111 0010	114	72	r
111 0011	115	73	s
111 0100	116	74	t
111 0101	117	75	u
111 0110	118	76	v
111 0111	119	77	w
111 1000	120	78	x
111 1001	121	79	y
111 1010	122	7A	z
111 1011	123	7B	{
111 1100	124	7C	
111 1101	125	7D	}
111 1110	126	7E	~
111 1111	127	7F	DEL

© 2021 TECHTARGET. ALL RIGHTS RESERVED

The Role of binary in computer architectures

Binary can be used in representing and storing data, performing calculations, retrieving data and controlling and organizing tasks. Since there is only two state 0 as low voltage “off” and 1 as high voltage “on” state, we can use this feature of binary to perform calculations using components like transistors and circuits.

We can also represent signed numbers using representations like sign and magnitude representation, 1's complement and 2's complement. Of all, Two's complement is commonly favored in contemporary systems due to its advantages over other methods. It simplifies arithmetic operations and eliminates the issue of having two representations for zero.

Binary bit Operations:

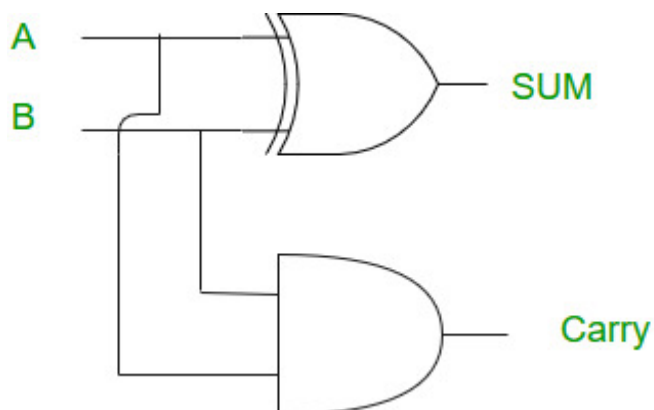
- **Addition:**
 - $1 + 1 = 0$, 1 carried over
 - $1 + 0 = 1$,
 - $0 + 0 = 0$

Example:

$$\begin{array}{r} 1101 \text{ (13)} \\ + 1011 \text{ (11)} \\ \hline 1|1000 \text{ (24)} \end{array}$$

- **Ones column:** $1 + 1 = 10$ (write down 0, carry-over 1)
- **Twos column:** $0 + 1 + 1 = 10$ (write down 0, carry-over 1)
- **Fours column:** $1 + 0 + 1 = 10$ (write down 0, carry-over 1)
- **Eights column:** $1 + 1 + 1 = 11$ (write down 1, carry-over 1)
- **Sixteenths column:** 1 (write down 1), exceeds the current bit size.

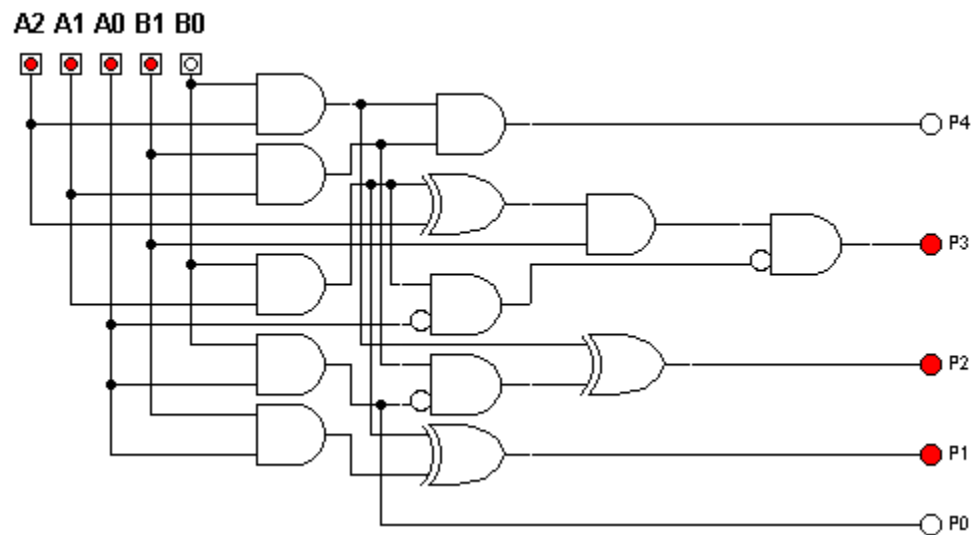
Addition can be performed in computing with the help of half adders and full adders.



- **Multiplication:**

- $1 \times 1 = 1$,
- $1 \times 0 = 0$,
- $0 \times 0 = 0$

$$\begin{array}{r}
 101101 \\
 1101 \\
 \hline
 101101 \\
 000000 \\
 \hline
 0101101 \text{ First Intermediate sum} \\
 101101 \\
 \hline
 11100001 \text{ Second Intermediate Sum} \\
 101101 \\
 \hline
 1001001001 \text{ Final Sum}
 \end{array}$$



- **Subtraction**

1. Find the two's complement of the subtrahend
 - **Invert all bits** (one's complement).
 - **Add 1** to the inverted bits
2. Add the result to the minuend

3. If there is a carry-out, discard it.
4. If no carry-out occurs, the result is negative (already in two's complement form)

Example: 7 - 5 in Binary in following steps:

1. Convert Numbers to Binary

7 → 0111

5 → 0101

2. Find Two's Complement of 5

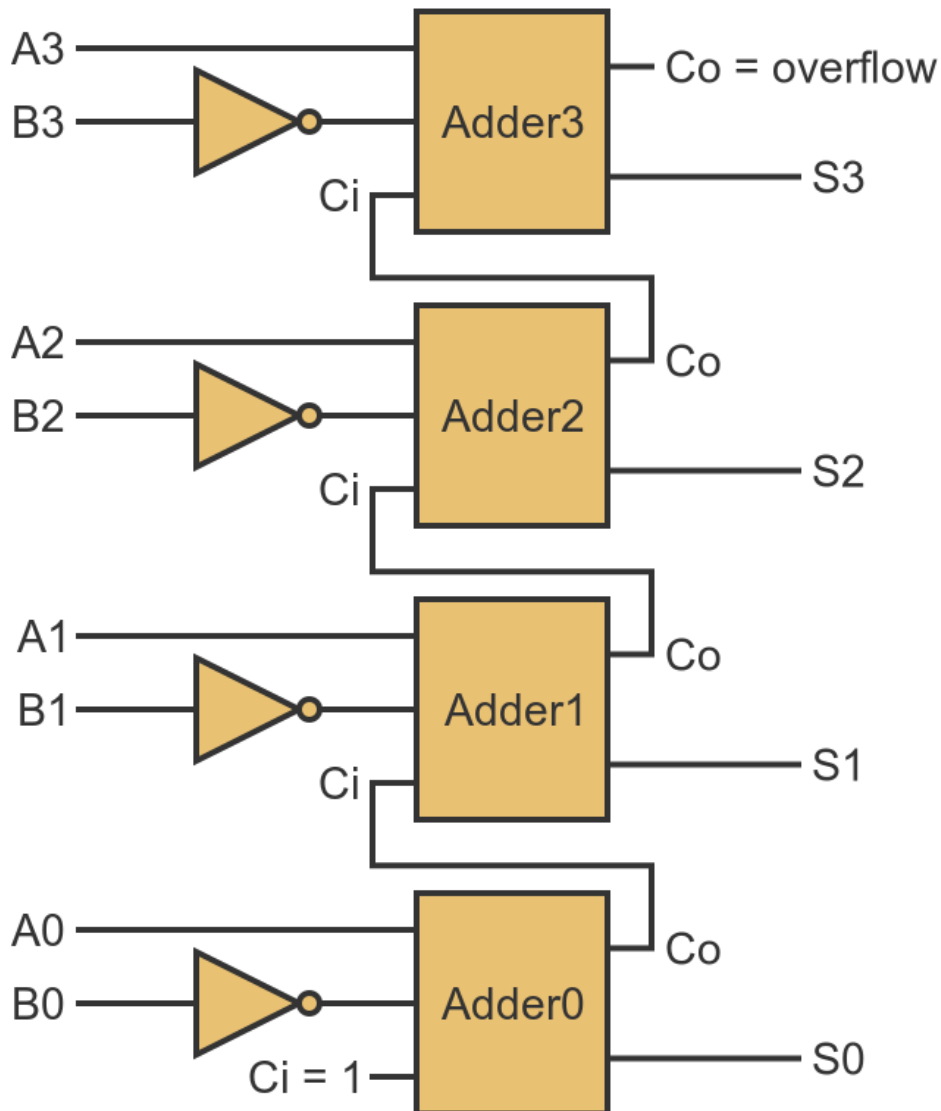
Invert all bits of 5: 0101 → 1010

Add 1: 1010 + 1 = 1011

3. Add to 7

```
  0111 (7)
+ 1011 (-5 in two's complement)
-----
1|0010 (Ignore carry-out → Final answer: 0010)
```

In digital logic, subtraction is possible through addition.



- **Division**

There are various ways of achieving division in digital logics, like:

1. Restoring Division Algorithm:
 - Involves repeated subtraction of the divisor from the dividend.
 - Restores the previous value if the result is negative.
 - Utilizes subtractors and control logic in the Arithmetic Logic Unit (ALU).
2. Non-Restoring Division Algorithm:

- Enhances efficiency by eliminating the restore step.
 - Adjusts the quotient based on the sign of the remainder.
 - Incorporates adders and control units for sign evaluations.
3. Newton–Raphson Division:
 - An iterative method that approximates the reciprocal of the divisor.
 - Multiplies the reciprocal by the dividend to obtain the quotient.
 - Requires multipliers and adders to perform iterative calculations.
 4. Goldschmidt Division:
 - Transforms division into a multiplication problem.
 - Simultaneously scales the dividend and divisor to converge the divisor to 1.
 - Uses parallel multipliers to enhance computation speed.

Example, Divide 13 by 3 in Binary

Dividend (13_{10}) = 1101_2

Divisor (3_{10}) = 11_2

Using **restoring division** algorithm in the following steps:

1. Initialize:

Dividend: 1101_2

Divisor: 11_2

Quotient: 0000 (initially zero)

Remainder: 0000 (initially zero)

2. Bring down the first bit of the dividend.

Remainder: 110 (first three bits of dividend)

Perform subtraction: $110 - 11 = 01$

Quotient: 1 (since subtraction was successful, we write 1)

3. Bring down the next bit of the dividend.

Remainder: 010 (bring down next bit, which is 0)

Perform subtraction: $010 - 11$ (this goes negative)

Restore the remainder: Return to 110 (previous remainder)

Quotient: 10 (since subtraction went negative, we write 0)

4. Bring down the final bit of the dividend.

Remainder: 110 (bring down the final bit, which is 1)

Perform subtraction: $110 - 11 = 01$

Quotient: 101 (successful subtraction, so write 1)

5. Final Result:

Quotient = 101_2 (which is 5 in decimal)

The diagram illustrates a digital architecture for a sorting algorithm. It features several key components:

- Comparators:** Two comparators at the top, labeled '2.1 (n-1) bits Max' and '2.1 n bits Max', receive inputs s_0, s_1, \dots, s_7 and n_0, n_1, \dots, n_7 respectively. They output signals s_0 and s_1 .
- Shift Registers:** Two shift registers, 'No variable n-1 bits left shift engine' and 'No variable n bits left shift engine', receive control signals S_A and S_1 and data from the comparators. They output signals s_0, s_1, \dots, s_7 and n_0, n_1, \dots, n_7 .
- Control and Counting:** A 'Counter i=j-1' and a 'Comparator K' are connected to the shift registers and comparators. They receive a 'Clk' signal and output Ady and Ady_0 .
- Register and Adder:** A 'Register n bits' receives inputs y_0, y_1, \dots, y_7 and a 'Clk' signal. Its output is connected to an 'Adder/subtractor (n+1) bits' block. The adder/subtractor has 'MSB' and 'LSB' outputs and is controlled by an 'Add/sub' signal.
- FG Blocks:** Multiple 'FG' (Function Generator) blocks are distributed throughout the circuit, receiving various inputs and outputs.

Binary Arithmetic Implemented in Arithmetic Logic Unit (ALU)

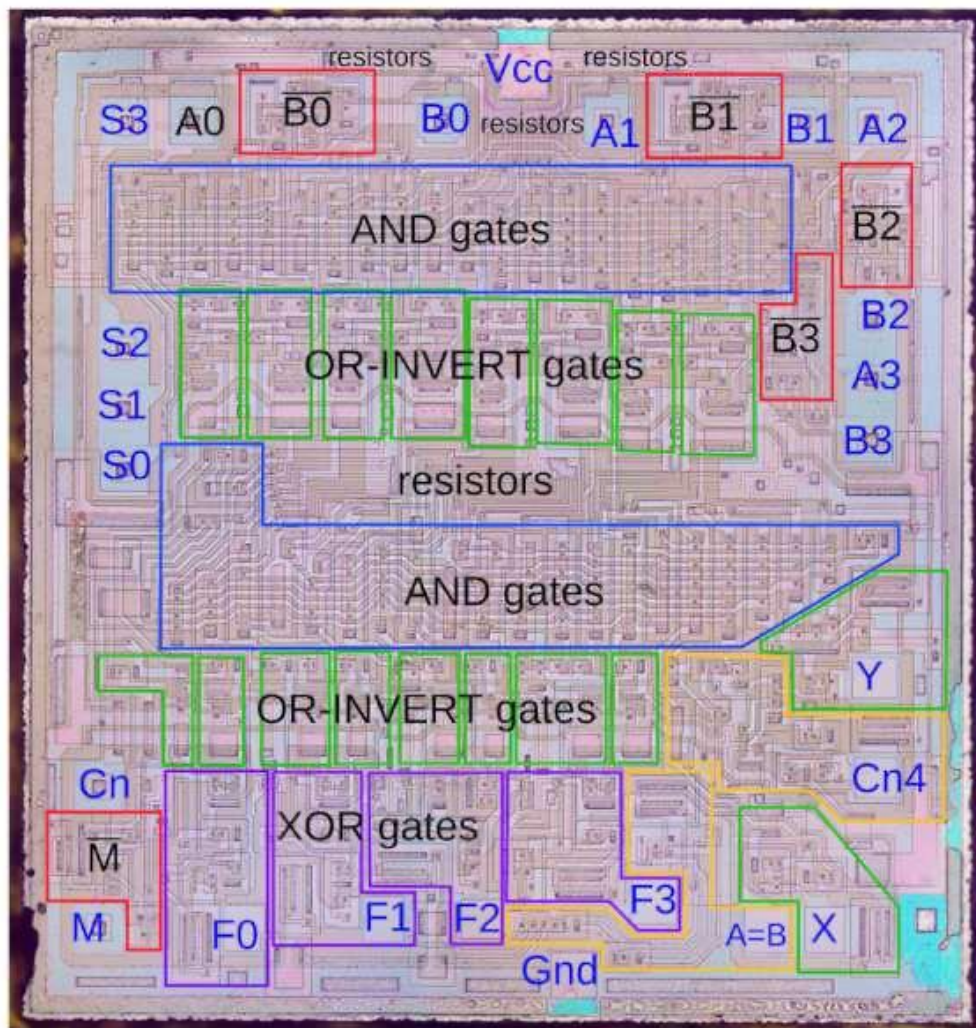
ALU is responsible for performing binary arithmetic and logical operation inside of a CPU. It can process addition, subtraction, multiplication, and division using logic gates, adders, and control circuits.

Major Components of the ALU for Binary Arithmetic:

1. Half Adder & Full Adder (For Addition & Subtraction)
 - The Half Adder performs single-bit binary addition using XOR and AND gates.
 - The Full Adder extends this concept to multi-bit addition by handling carry-over bits.
 - Subtraction is done using Two's Complement, so subtraction circuits often use adders with inverters instead of separate subtractors.
2. Two's Complement Circuit (For Subtraction)
 - The ALU converts subtraction into addition by taking the two's complement of the subtrahend and adding it to the minuend.
 - Uses NOT gates (for bit inversion) and increment circuits (to add 1).
3. Shift-and-Add Multiplication (For Multiplication)
 - Multiplication is performed using bitwise shifting and addition.
 - The ALU checks each bit of the multiplier:
 - If 1, it adds the multiplicand.
 - If 0, it only shifts left (like long multiplication).
 - High-performance processors use Booth's Algorithm to optimize this process.
4. Restoring & Non-Restoring Division (For Division)
 - The ALU uses subtraction-based division algorithms like:
 - Restoring Division (restores remainder if negative).
 - Non-Restoring Division (avoids unnecessary restores for efficiency).
 - More advanced processors use Newton–Raphson and Goldschmidt algorithms for fast division.
5. Logic Gates (For Boolean Operations)
 - The ALU also performs logic operations like AND, OR, XOR, and NOT to manipulate binary values.
 - These operations are used for bitwise operations, comparisons, and decision-making in programs.

How ALU Executes Binary Arithmetic Step-by-Step

1. **Addition:** Uses full adders to sum binary numbers, handling carry propagation.
2. **Subtraction:** Uses two's complement to convert subtraction into addition.
3. **Multiplication:** Uses shift-and-add method, optimized with Booth's Algorithm.
4. **Division:** Uses repeated subtraction or advanced division algorithms.
5. **Logic Operations:** Uses AND, OR, XOR, and NOT gates for bitwise computations.



References

Computer Architecture Fundamentals

https://fastercapital.com/topics/types-of-instruction-set-architectures.html?utm_source=chatgpt.com

Admin. (2024, May 17). Instruction Set Architecture (ISA): the foundation of modern computing - SolveForce Communications. SolveForce. <https://solveforce.com/instruction-set-architecture-isa-the-foundation-of-modern-computing/>

<https://abit.edu.in/wp-content/uploads/2022/07/advanced-computer-architecture.pdf>

The evolution of computing. (n.d.). The Interaction Design Foundation. <https://www.interaction-design.org/literature/book/the-social-design-of-technical-systems-building-technologies-for-communities/the-evolution-of-computing>

Advanced Computer Architecture. (n.d.). [E-book]. Prof.A. Pal, IIT, Kharagpur. <https://abit.edu.in/wp-content/uploads/2022/07/advanced-computer-architecture.pdf>

GeeksforGeeks. <https://www.geeksforgeeks.org/computer-organization-risc-and-cisc/>

Logic gates

Wikipedia contributors. (2025, February 22). Logic gate. Wikipedia. https://en.wikipedia.org/wiki/Logic_gate

GeeksforGeeks. (2025, January 9). Logic Gates Definition, Types, Uses. GeeksforGeeks. <https://www.geeksforgeeks.org/logic-gates/>

Summary of the common Boolean logic gates with symbols and. . . (n.d.). ResearchGate. https://www.researchgate.net/figure/Summary-of-the-common-Boolean-logic-gates-with-symbols-and-truth-tables_fig3_291418819

Binary

GeeksforGeeks. (2024, September 30). Binary Number System. GeeksforGeeks. <https://www.geeksforgeeks.org/binary-number-system/>

Awati, R. (2022, May 25). binary. Whatls. <https://www.techtarget.com/whatis/definition/binary>

Characters - Units and data representation - OCR - GCSE Computer Science Revision - OCR - BBC Bitesize. (2024, May 14). BBC Bitesize.

<https://www.bbc.co.uk/bitesize/guides/zfspfcw/revision/7>

Awati, R. (2022, May 25). binary. Whatls. <https://www.techtarget.com/whatis/definition/binary>

GeeksforGeeks. (2024, December 28). Binary representations in digital logic. GeeksforGeeks. <https://www.geeksforgeeks.org/binary-representations-in-digital-logic/>

Mari, L. (2021, September 8). Digital Power Electronics Basics: Digital (Binary) operation. Technical Articles. <https://eepower.com/technical-articles/digital-electronics-basics-digital-binary-operation/>

GeeksforGeeks. (2024, September 23). Half adder in digital logic. GeeksforGeeks. <https://www.geeksforgeeks.org/half-adder-in-digital-logic/>

Admin. (2020, July 29). Binary Multiplication (Rules and Solved Examples). BYJUS. <https://byjus.com/maths/binary-multiplication/>

Cluster-based evolutionary design of digital circuits using all improved multi-expression programming - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Evolved-3x2-bit-multiplier-13-gates-with-4-levels-using-and-and-with-one-input_fig1_220742040 [accessed 14 Mar 2025]

McBride, M. (n.d.). GraphicMaths - Creating a subtractor with logic gates. <https://graphicmaths.com/computer-science/logic/subtractor/>

Division Circuit Using Reversible Logic Gates - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Proposed-reversible-8-bit-divider-circuit_fig5_324151648 [accessed 14 Mar 2025]

Rao, R. (2024, November 23). Building a computer from scratch: Understanding Arithmetic Logic Unit #2. Medium. <https://medium.com/%40ruthurao/building-a-computer-from-scratch-understanding-arithmetic-logic-unit-2-315ce860c972>

Mixos. (2017, January 13). Inside the 74181 ALU chip: die photos and reverse engineering. Electronics-Lab.com. <https://www.electronics-lab.com/inside-74181-alu-chip-die-photos-reverse-engineering/>