A SunCam online continuing education course

Microcontrollers: the Arithmetic Logic Unit

by

Mark A. Strain, P.E.

Microcontrollers: the Arithmetic Logic Unit
A SunCam online continuing education course

Table of Contents

Introduction 1
Binary Numbering System 1
  Decimal Notation 2
  Binary Notation 2
    Binary to Decimal 2
    Decimal to Binary 3
  Negative numbers in Binary 4
    Ones' Complement 4
    Twos' Complement 5
Logic Operations in ALUs 7
  Binary Logic 7
  Fundamental Logic Gates 8
    The AND Gate 9
    The OR Gate 11
    The NOT Gate 13
  Combined Logic Gates 15
    The XOR Gate 15
    The NAND Gate 16
    The NOR Gate 16
    The XNOR Gate 17
Arithmetic Operations in ALUs 18
  Adder 19
    Half Adder 19
    Full Adder 20
    Ripple Carry Adders 22
Arithmetic Logic Unit 23
Design and Structure of an ALU 24
  Two-Bit ALU 25
  Instruction Set and Opcodes 28
ALU Integration in the CPU 31
  Cycles of the Central Processing Unit 31
The Fetch-Decode-Execute Cycle 32
  Instruction Fetch in a CPU 33
    Key Functions of Instruction Fetch 33
  Instruction Decode in a CPU 34
    Key Functions of an Instruction Decoder 35
  Instruction Execute Cycle in a CPU 36
History 37
Summary 41
References 43

Microcontrollers: the Arithmetic Logic Unit
A SunCam online continuing education course

Introduction

The arithmetic logic unit (ALU) is the central core of a central processing unit (CPU). The ALU is simply a digital circuit that performs arithmetic and logical operations on binary numbers. They are combinational logic circuits which means that their outputs change asynchronously in response to changes to their inputs. ALU circuits perform operations on integer binary numbers. A floating point unit (FPU) is used to do operations on floating point numbers (or non-integers). This course focuses on the ALU.

An ALU has two integer inputs called operands and another input called an opcode. The opcode instructs the ALU which instruction to perform (like addition, subtraction, decrement, increment, AND, OR, NOT, XOR, etc.). The opcode code is a binary code that comes from the instruction set (or program) that is being executed. The instructions or program will contain both the operands or numbers to be used and the opcode that tells the ALU what to do with the numbers, eg. add the numbers. These instructions are usually written in a higher level programming language and are stored in the computer's main memory. A compiler will compile the higher level language program and convert it to machine code. The computer executes one instruction at a time.

The CPU will go through a fetch-decode-execute cycle for each instruction. It will fetch an instruction from the program in memory, it will decode the instruction into binary codes that the ALU can use, and it will execute the instruction.

Binary Numbering System

In everyday life we are used to the numbering system known as the decimal numeral system. Our common numbering system is based on Arabic numerals (or symbols). Our numbering system is base-10 which means there are ten symbols (0, 1, 2, 3, 4, 5, 6, 7, 8 and 9) used to represent every possible combination of numbers. Our numeral system is based on the number ten probably because long ago we discovered that we each have ten fingers which are useful tools to count on when doing simple math.

Computer systems and other digital systems use a numbering system based on a number other than ten. Digital systems (such as a computer central processing unit) use a numbering system based on the number two. This base-2 numbering system

is called the binary system. The binary system uses two symbols (0 and 1) to represent every possible combination of numbers.

Decimal Notation

When we write decimal (base-10) numbers, we use positional notation. This means that each digit in a number is multiplied by a specific power of ten. The powers of ten (exponents) are positive on the left side of the decimal point and the powers of ten (exponents) are negative on the right side of the decimal point.

For example, consider the decimal number 3854:

$= 3(10^3) + 8(10^2) + 5(10^1) + 4(10^0)$
$= 3000 + 800 + 50 + 4$
$= 3854$

Now consider the decimal number 1256.79:

$= 1(10^3) + 2(10^2) + 5(10^1) + 6(10^0) + 7(10^{-1}) + 9(10^{-2})$
$= 1000 + 200 + 50 + 6 + 0.7 + 0.09$
$= 1256.79$

…	10^5	10^4	10^3	10^2	10^1	10^0	.	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	…
			1	2	5	6	.	7	9				

Binary Notation

The binary system also uses positional notation. Each digit in the base-2 numbering system is multiplied by a specific power of two. Just as in the base-10 system, the powers of two are positive on the left side of the binary point and the powers of two are negative on the right side of the binary point.

Binary to Decimal

Each digit is a multiple of a power of 2. All digits to the left of the decimal point are positive powers of 2 and all digits to the right of the decimal point are negative powers of 2.

Consider the binary number 1011:

$= 1(2^3) + 0(2^2) + 1(2^1) + 1(2^0)$
$= 8 + 0 + 2 + 1$
$= 11$ (base-10)

Now consider the binary number 10101101.101:

$= 1(2^7) + 0(2^6) + 1(2^5) + 0(2^4) + 1(2^3) + 1(2^2) + 0(2^1) + 1(2^0) + 1(2^{-1}) + 0(2^{-2}) + 1(2^{-3})$

$= 128 + 0 + 32 + 0 + 8 + 4 + 0 + 1 + 1/2 + 0 + 1/8$

$= 173.625$ (base-10)

…	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	…
		1	0	1	0	1	1	0	1	.	1	0	1			

Decimal to Binary

It is often necessary to represent a decimal fraction in binary. The integer part (to the left of the decimal point) is converted to binary by continually dividing by 2 until you get to 1. The fractional part is converted to binary by continually multiplying by 2 until you get 0 or a repeating sequence or you get tired.

Consider the decimal number 142.378:

First, convert the integer part:

Divide by 2	Remainder
142 / 2	0
71 / 2	1
35 / 2	1
17 / 2	1
8 / 2	0
4 / 2	0
2 / 2	0
1	1

Now, read the remainder part backwards and that is the binary representation of the integer part of the number: 10001110

Next, convert the fractional part. Start with the number to the right of the decimal point (0.378). Multiply the number times 2 and record what is to the left of the decimal place after this operation. Then take this number and discard whatever is to the left of the decimal place and continue.

$0.378 * 2 = 0.756 \rightarrow 0$
$0.756 * 2 = 1.512 \rightarrow 1$
$0.512 * 2 = 1.024 \rightarrow 1$

0.024 * 2 = 0.048 → 0
0.048 * 2 = 0.096 → 0
0.096 * 2 = 0.192 → 0
0.192 * 2 = 0.384 → 0
0.384 * 2 = 0.768 → 0
0.768 * 2 = 1.536 → 1
0.536 * 2 = 1.072 → 1
0.072 * 2 = 0.144 → 0
0.144 * 2 = 0.288 → 0
0.288 * 2 = 0.576 → 0
0.576 * 2 = 1.152 → 1
0.152 * 2 = 0.304 → 0

…

Now, read the number forwards and that is the binary representation of the fractional part of the number: 0.011000001100010

Therefore, 142.378 = 10001110.011000001100010…

Negative numbers in Binary

In mathematics, negative decimal numbers can be represented by using a minus "-" sign. In digital circuits numbers are represented only by a sequence of bits, either a 0 or a 1 and no plus or minus sign. The most popular methods of representing signed numbers in binary are the ones' complement and twos' complement methods. These methods allow subtraction to be performed by adding the complement of a number instead of subtracting the number. The utilization of ones' complement and twos' complement greatly simplifies digital circuits since addition is a fundamental operation.

Ones' Complement

The ones' complement of a binary number is obtained by negating the number by inverting all of the bits. This is accomplished by changing all of the 0s into 1s and all of the 1s into 0s. The ones' complement of a number behaves as the negative of the original number. An addition operation is a fundamental operation in digital systems. There is no fundamental subtraction operation. Subtraction of two numbers is equivalent to adding one number to the negative of the other number. Therefore, any subtraction operation is equivalent to inverting one of the numbers and adding it to the other number.

Consider taking the ones' complement of the number 0000 1100 (base-2):

0000 1100

Invert all of the bits

1111 0011

Ones' complement is seldom used in digital systems because when a ones' complement number is added to another number, the result is offset by –1. In other words, the result of a subtraction operation (using ones' complement) is off by –1.

Consider subtracting 8 from 12 using ones' complement:

ones' complement of 8:

0000 1000

Invert all of the bits

1111 0111

Now subtract 8 from 12:

12 – 8 = 4

Add the ones' complement of 8 to 12: (12 – 8)

```
 0000 1100
 1111 0111
-----------
 0000 0011
```

= 3 (base-10)

Note that the result is off by one. This problem is resolved by performing a twos' complement operation instead of a ones' complement.

Twos' Complement

Twos' complement representation of a binary number has widespread use in digital systems. It solves the problem of the –1 offset that a ones' complement produces.

Consider the same subtraction operation as above, but this time using twos' complement of 8:

0000 1000

Invert all of the bits and add one

```
1111 0111
0000 0001
----------
1111 1000
```

(Note: Binary digits are often grouped in groups of four for readability.)

Now subtract 8 from 12:

$$12 - 8 = 4$$

Add the twos' complement of 8 to 12

```
0000 1100
1111 1000
----------
0000 0100
```

= 4 (base-10)

Now consider subtracting 23 from 17: (17 – 23)

twos' complement of 23:

```
0001 0111
```

Invert all of the bits and add one

```
1110 1000
0000 0001
----------
1110 1001
```

Now subtract 23 from 17:

$$17 - 23 = -6$$

Add the twos' complement of 23 to 17

```
0001 0001
1110 1001
----------
1111 1010
```


= –6 (base-10)

This is a negative number since the sign bit (most significant bit) is set. The number (without the sign) may be determined by taking the twos' complement of the result:

twos' complement of 1111 1010:

```
1111 1010
```

Invert all of the bits and add one

```
0000 0101
0000 0001
----------
0000 0110
```

= 6 (base-10)

So the answer is –6.

Logic Operations in ALUs

Logic operations in an ALU are fundamental because they allow CPUs to perform essential operations on data, which are crucial for executing programs and making decisions. CPUs operate using binary numbers. Logic operations directly manipulate these binary values. Logic operations allow for the manipulation of data, such as setting, clearing, bit flipping, and masking bits. These operations are fundamental for tasks like data encoding, data decoding, encryption, error detection, and error correction. Logic operations allow for control flow mechanisms made possible by conditional statements and loops to determine the flow of program execution. Branching operations by comparing two values to decide which branch of code to execute involves logic operations.

Addition and subtraction are performed using logic gates in a full adder and performing twos' complement using logic operations. Bit shifting and bit rotation are logic operations essential for multiplication and division by powers of two. Algorithms involving cryptography and data compression rely on logic operations.

Binary Logic

Binary logic uses variables that are either in a high state (or a logic "1") or in a low state (or a logic "0"). This makes the use of the binary numbering system perfect for digital systems like the ones used in computer systems. A switch is either on or off; it can implement two discrete logic states, logic "1" and logic "0". The switching function in digital systems is implemented by the use of a transistor. A transistor, when biased properly, can act as a digital switch. Digital systems use thousands, millions and sometimes billions of transistors to implement the complex logic of the central processing unit of a simple microcontroller to a complex multicore microprocessor.

Every binary logical condition must assume a logic value 0 or 1. There must be a way to combine different complex logical conditions to provide a logical result. The complex logical conditions are represented by logical functions and implemented with electrical circuits. Each logic function has its own special symbol and each has its own specific behavior.

The basic building blocks of a microcontroller or microprocessor are called logic gates. These gates are basic electrical circuits that have at least one input and only one output. The input and output values are logical values true (or 1) and false (or 0).

Gates have no memory; their output depends only on the value of the inputs. A gate's output is sometimes called its logical function. The relationship of a logic gate's output versus its inputs is best described by a truth table. A truth table lists every possible combination of inputs (in order) in tabular form and presents the corresponding output value in a separate column.

The following is an example of a truth table with two inputs (A and B) and one output (F). The table lists every possible combination of inputs in order and each associated output.

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

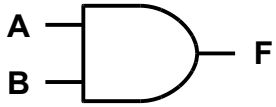
Figure 1 - Truth Table

Fundamental Logic Gates

The three most basic logic functions are AND, OR, NOT. Any logical function can be implemented using these three different types of gates.

The AND Gate

The AND gate implements the AND function. The AND logic function is represented as F = AB, where F is the output and A and B are the inputs. The operator is sometimes represented as a dot, A•B, but is most often represented with no operator, AB. The output is a logic 1 only if both inputs are logic 1. The following shows the symbol for an AND gate and its associated truth table.



A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

Figure 2 - AND Gate (F = AB)

An AND gate can be thought of as two switches (inputs A and B) connected in series with a power source such as a battery or power supply and with the output such as a lamp. The output lamp is not illuminated unless both the switches are closed.

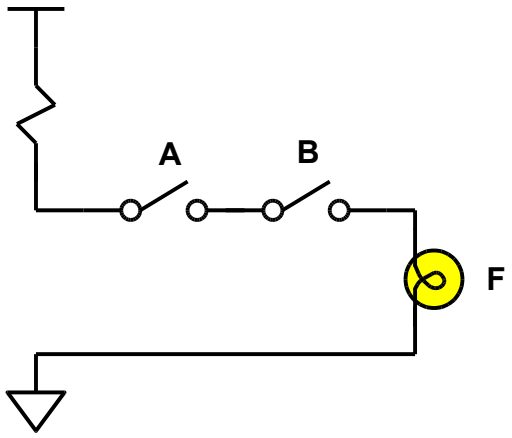


Figure 3 - AND gate composed of switches

A simple way to implement an AND gate is by connecting two NPN transistors in series with a power source and the output. The inputs A and B are the base connections of the two transistors.

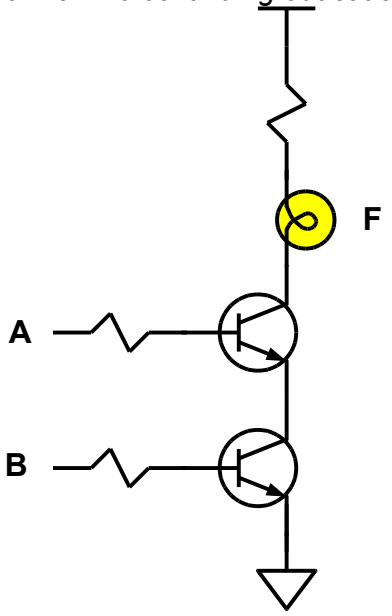
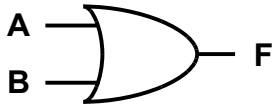


Figure 4 - AND gate composed of transistors

The OR Gate

The OR gate implements the OR function. The OR logic function is represented as F = A + B, where F is the output and A and B are the inputs. The operator is represented by a plus (+) sign. The output is a logic 1 if either of the inputs is a logic 1. The following shows the symbol for an OR gate and its associated truth table.



A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Figure 5 - OR Gate (F = A + B)

An OR gate can be thought of as two switches (inputs A and B) connected in parallel and then connected in series with a power source and the output lamp. The output lamp is illuminated if either switch is closed.

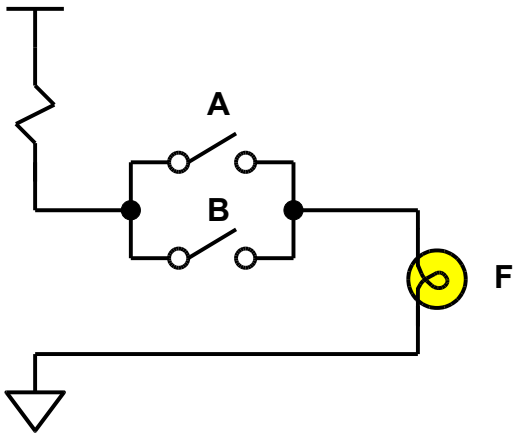


Figure 6 - OR gate composed of switches

A simple way to implement an OR gate is by connecting two NPN transistors in parallel whose inputs A and B are the base connections. The parallel combination of the two transistors is connected in series with the power source and the output.

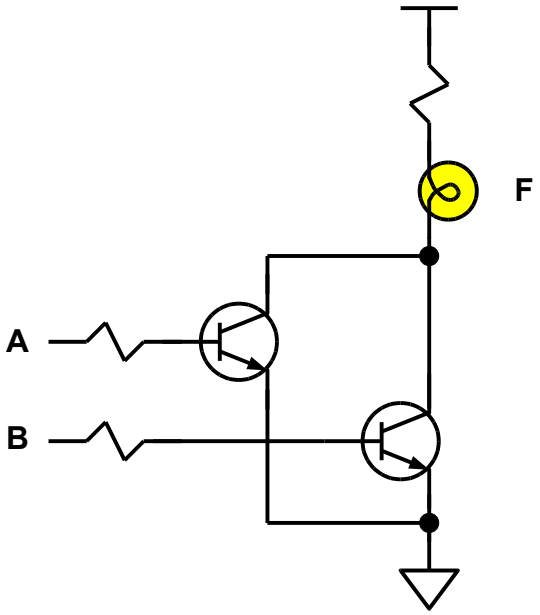
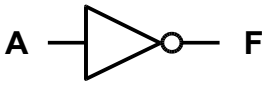


Figure 7 - OR gate composed of transistors

The NOT Gate

The NOT gate is sometimes called an inverter. It implements the NOT (or invert) function. The NOT gate has only a single input. Its logic function is represented as F = A'. The operator is represented by a single tick mark (') after the variable or as a bar (–) over the variable. The output is the inverse of the input. The following shows the symbol for a NOT gate and its associated truth table.



A	F
0	1
1	0

Figure 8 - NOT Gate (F = A')

A NOT gate or an inverter is as simple as a normally open input switch connected in parallel with the output. The switch is connected in series with the power source and a resister. When the switch is open the current runs from the power source, through the resistor, through the lamp and finally to ground. When the switch is closed, the current is rerouted directly to ground, bypassing the lamp. Thus, when the switch is open (or off), the output is on (or high), and when the switch is closed (or on), the output is off (or low).

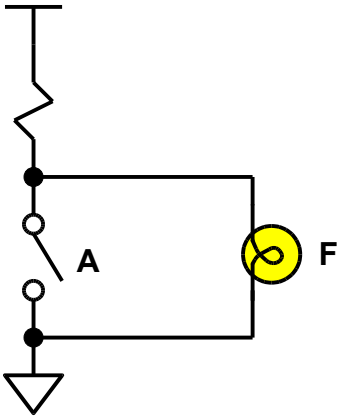


Figure 9 - NOT gate composed of a switch

A simple way to implement a NOT gate is by replacing the switch with an NPN transistor. The base of the transistor is the input to the gate. When the input is high, the current is rerouted through the transistor, thus bypassing the lamp.

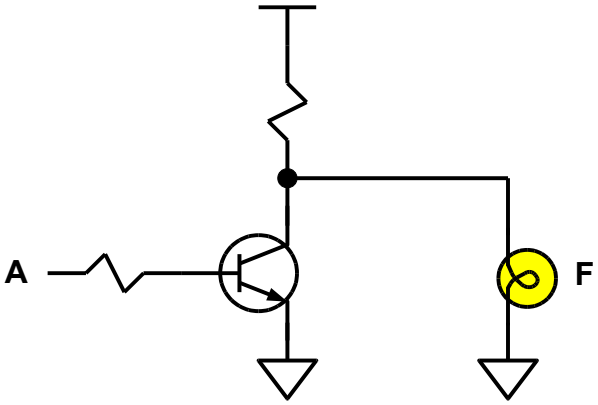


Figure 10 - NOT gate composed of a transistor

Combined Logic Gates

The most basic logic operations are AND, OR and NOT. These gates are fundamental. These gates can be combined to form other logical operations such as XOR, NAND, NOR and XNOR. Three of these gates (NAND, NOR and XNOR) have active low outputs. This means that their outputs are inverted. The XOR gate (as well as the AND and OR gates) are active high. Their outputs are not inverted.

The XOR Gate

The XOR gate implements the XOR function. The XOR logic function is represented as F = A (+) B, where F is the output and A and B are the inputs. The operator is represented by a plus sign with a circle around it (+). The output is a logic 1 if one of the inputs is a logic 1 and the other input is a logic 0. The following shows the symbol for an XOR gate and its associated truth table.

The XOR gate is composed of two AND gates, an OR gate and two NOT gates.

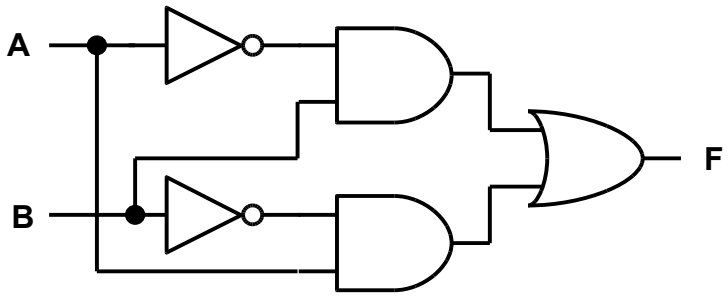
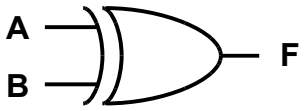


Figure 11 - XOR gate is composed of AND, OR and NOT gates



A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

Figure 12 - XOR Gate (F = A(+)B)

The NAND Gate

The NAND gate implements the NAND function. The NAND logic function is represented as F = (AB)', where F is the output and A and B are the inputs. The output is a logic 1 if either of the inputs is a logic 0. The following shows the symbol for a NAND gate and its associated truth table.

The NAND gate is composed of an AND gate followed by a NOT gate.

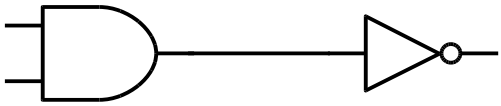
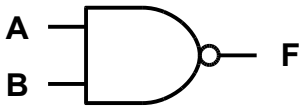


Figure 13 - NAND gate is composed of AND and NOT



A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

Figure 14 - NAND Gate (F = (AB)')

NAND gates have functional completeness. This means that any combinational logic function can be realized using only NAND logic (or using only NAND gates). NAND gates are universal gates that can be combined to form any other kind of logic gate (NOT, AND, OR, XOR, NOR, XNOR). A NAND gate is a universal gate which means that it can be used to implement any other Boolean function.

The NOR Gate

The NOR gate implements the NOR function. The NOR logic function is represented as F = (A + B)', where F is the output and A and B are the inputs. The output is a logic 1 if both of the inputs are a logic 0. The following shows the symbol for an NOR gate and its associated truth table.

The NOR gate is composed of an OR gate followed by an NOT gate.

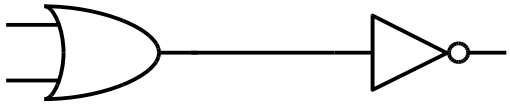
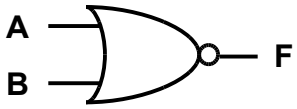


Figure 15 - NOR gate is composed of OR and NOT



A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

Figure 16 - NOR Gate (F = (A+B)')

The XNOR Gate

The XNOR gate implements the XNOR function. The XNOR logic function is represented as F = (A (+) B)', where F is the output and A and B are the inputs. The output is a logic 1 if both of the inputs are a logic 0 or if both of the inputs are a logic 1. The following shows the symbol for an XNOR gate and its associated truth table.

The XNOR gate is composed of an XOR gate followed by a NOT gate.

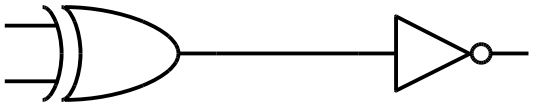
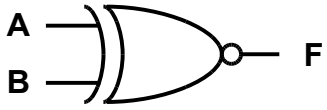


Figure 17 - XNOR is composed of XOR and NOT



A	B	F
0	0	1
0	1	0
1	0	0
1	1	1

Figure 18 - XNOR Gate (F = (A(+)B)')

Bit shifting operations move each bit in number's binary format left or right. There are two different types of shifting operations: logical shifts and arithmetic shifts. In a logical right shift, the least significant bit is lost and 0 is inserted in the most significant bit position. In a logical left shift, the most significant bit is lost and a 0 is inserted in the least significant bit position. In an arithmetic right shift, the least significant bit is lost, and the most significant bit is copied (thus preserving the sign bit in signed numbers). In an arithmetic left shift, the most significant bit is lost and a 0 is inserted in the least significant bit position.

Simple ALUs can shift an operand by one bit position. More complex ALUs can shift an operand by any number of bits in one operation by incorporating a barrel shifter. A barrel shifter is a digital circuit that can shift a data word by a specified number of bits without the use of any sequential logic, simply combinational logic. One implementation of a barrel shifter is to create a sequence of multiplexers where the output of one multiplexer is connected to the input of the next multiplexer in a way that depends on the shift distance.

Arithmetic Operations in ALUs

Arithmetic operations in an ALU are essential because they perform the fundamental mathematical calculations required to execute programs and process data. Arithmetic operations provide the means to perform data manipulation and control flow operations. Arithmetic operations such as addition, subtraction, multiplication, and division allow CPUs to perform essential numerical calculations required by most software applications, ranging from simple spreadsheets to complex scientific simulations. Arithmetic operations are used to compare numeric values, which is essential for program control flow. ALUs support integer arithmetic for operations such as multiplication and division.

Floating point operations are performed in a floating point unit (FPU) which is another component of a CPU.

Adder

An adder is a digital circuit that adds two numbers. An adder is sometimes called a summer (one that sums). An adder circuit is an integral part of a microprocessor's arithmetic logic unit (ALU). Adders operate on binary numbers. They add two numbers and can also subtract two numbers. Subtraction is accomplished by adding a number to a negative number. A negative number is created by taking the ones' complement or twos' complement of the number.

The simplest form of adder is a half adder. A half adder becomes a full adder when it takes a carry bit as an input. Several full adders can be cascaded in a row to add large numbers. This is sometimes called a ripple carry adder.

Half Adder

A half adder adds two binary digits A and B and produces two outputs, sum (S) and carry (C). The carry output represents an overflow condition adding the two digits A and B. The carry will be either a 0 or 1. The single digit half adder, or one-bit half adder, is very simple.

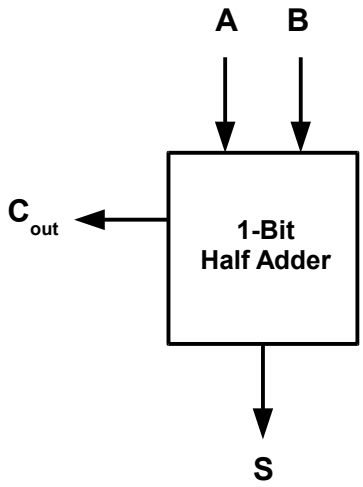


Figure 19 - One-bit half adder

Its truth table looks like this.

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

As you can see S is simply the result of A added to B, and C is high only when the result of A + B will not fit in one digit. From looking at the truth table the equations for S and C become apparent:

$$S = A \ (+) \ B$$

$$C = AB$$

So the logic diagram is

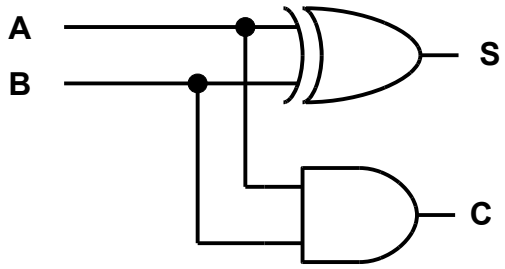


Figure 20 - Logic diagram of a half adder

Full Adder

A full adder is simply a half adder that takes in a carry bit from another section. A full adder has three inputs A, B and C_{in} and has two outputs S and C_{out}. A full adder is usually a single stage in a cascade of adders used to add large numbers.

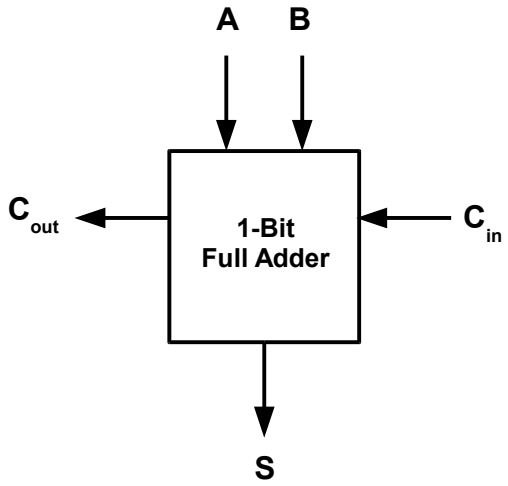


Figure 21 - One-bit full adder

The truth table of a full adder is

C_{in}	A	B	C_{out}	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

The equations for C_{out} and S are

$$S = C_{in}'A'B + C_{in}'AB' + C_{in}A'B' + C_{in}AB$$

$$= A \,(+)\, B \,(+)\, C_{in}$$

$$C_{out} = AB + C_{in}(A \,(+)\, B)$$

And the logic diagram is

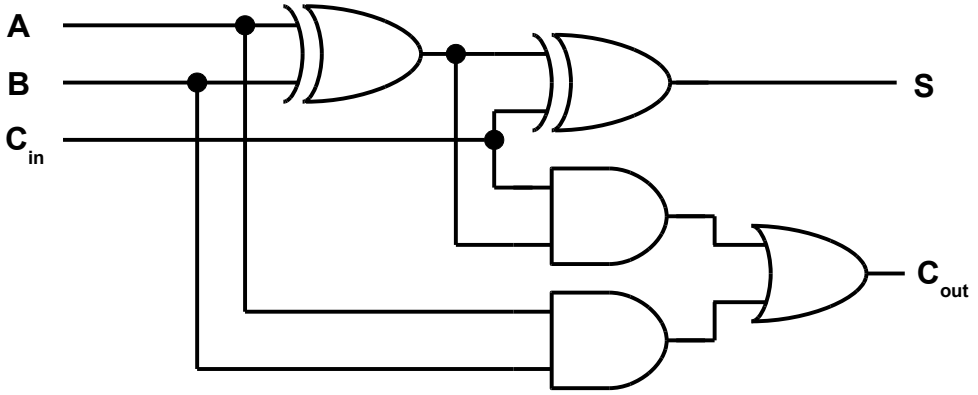


Figure 22 - Logic diagram of a full adder

In order to cascade the adders to add larger numbers the carry output of the least significant stage is connected to the carry input of the next significant stage.

Ripple Carry Adders

Full adders can be cascaded (or connected together) to form an adder that can add numbers larger than one bit. A half adder is used at the least significant bit and full adders are used for each additional bit. The carry out output of the previous stage is connected to the carry in input of the next stage. These interconnections continue until all stages are connected. The sum output for each stage forms the multiple bit outputs, and the A and B inputs for each stage form the multiple bit inputs. The A and B inputs are $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$. The output for the circuit above is a 5-digit binary number formed by all of the sum outputs and the last carry out output: $C_{out3}S_3S_2S_1S_0$.

In order to add two four-digit binary numbers, four stages are required:

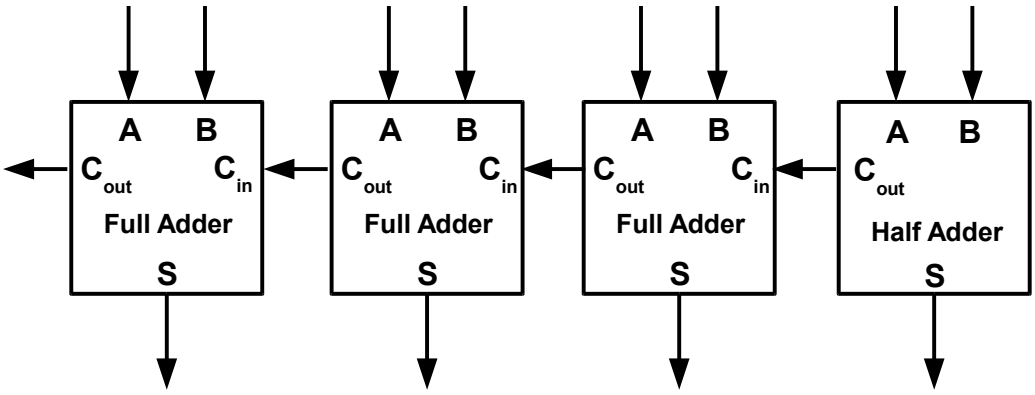


Figure 23 - 4-bit adder

Arithmetic Logic Unit

All microcontrollers contain an arithmetic logic unit (ALU). It is a fundamental building block of the CPU. The ALU is a digital circuit that performs arithmetic and logical operations. The arithmetic operations include addition, subtraction, multiplication and division. These are integer operations, i.e., the inputs (or operands) are integers as well as the output. Some of the more complex processors will contain a floating point unit that handles fractional numbers in addition to integers. The logic operations include AND, OR, NOT and XOR as well as logical comparisons and bit shifting left and right.

The control bits on an ALU is called the opcode. It is short for operation code. It controls the operation of the ALU. The size of the opcode (width in bits) will determine the number of operations that the ALU can perform. The opcode will instruct the ALU to perform either an add, subtract, AND, OR, etc. operation. There are two data inputs to the ALU. These are the operands. The operands are what are operated on and the result appears at the output of the ALU.

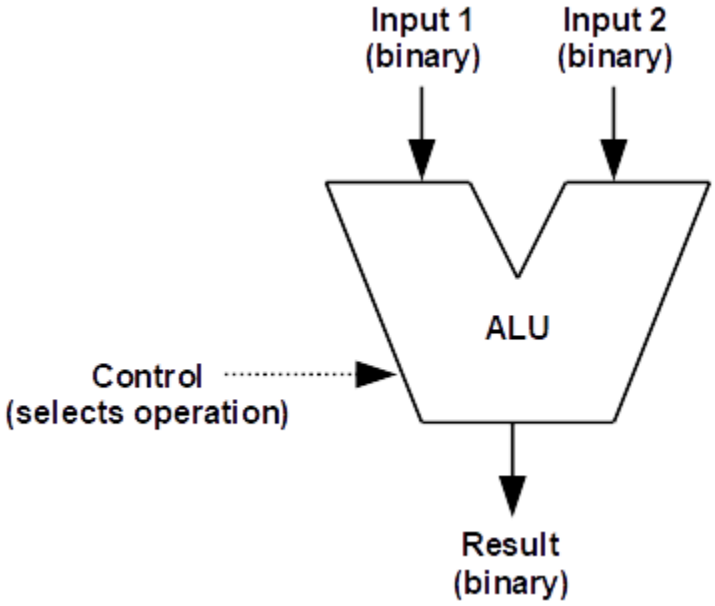


Figure 24 - Arithmetic Logic Unit

The ALU performs arithmetic and logic operations on two binary numbers resulting in another binary number. The numbers (the operands and the result) are represented using twos' complement format. Twos' complement allows subtraction to be performed by adding the complement of a number instead of subtracting the

number. The utilization of twos' complement greatly simplifies the ALU circuit since addition is a fundamental operation.

Design and Structure of an ALU

A simple Arithmetic Logic Unit (ALU) can be described by a truth table. An ALU performs a variety of arithmetic and logic operations, and a truth table can represent the output of these operations based on different input conditions.

For instance, a 1-bit ALU that supports operations like AND, OR, addition, and subtraction might have a truth table with columns for each input (A and B) and control signals that select the operation (Op) and the resulting output. Here's an example of a truth table for a simple 1-bit ALU:

Op	A	B	Result
00	0	0	0
00	0	1	0
00	1	0	0
00	1	1	1
01	0	0	0
01	0	1	1
01	1	0	1
01	1	1	1
10	0	0	0
10	0	1	1
10	1	0	1
10	1	1	0
11	0	0	0
11	0	1	1
11	1	0	1
11	1	1	0

In this table:

- **A** and **B** are the inputs
- **Op** is the opcode to select the operation to perform
    - 00 for AND

  - 01 for OR
  - 10 for addition
  - 11 subtraction
- **Result** is the output of the ALU

The schematic for the 1-bit ALU is shown in Figure 25. A and B are the 1-bit operands. The opcode is 2-bits x and y. The result is presented at output F.

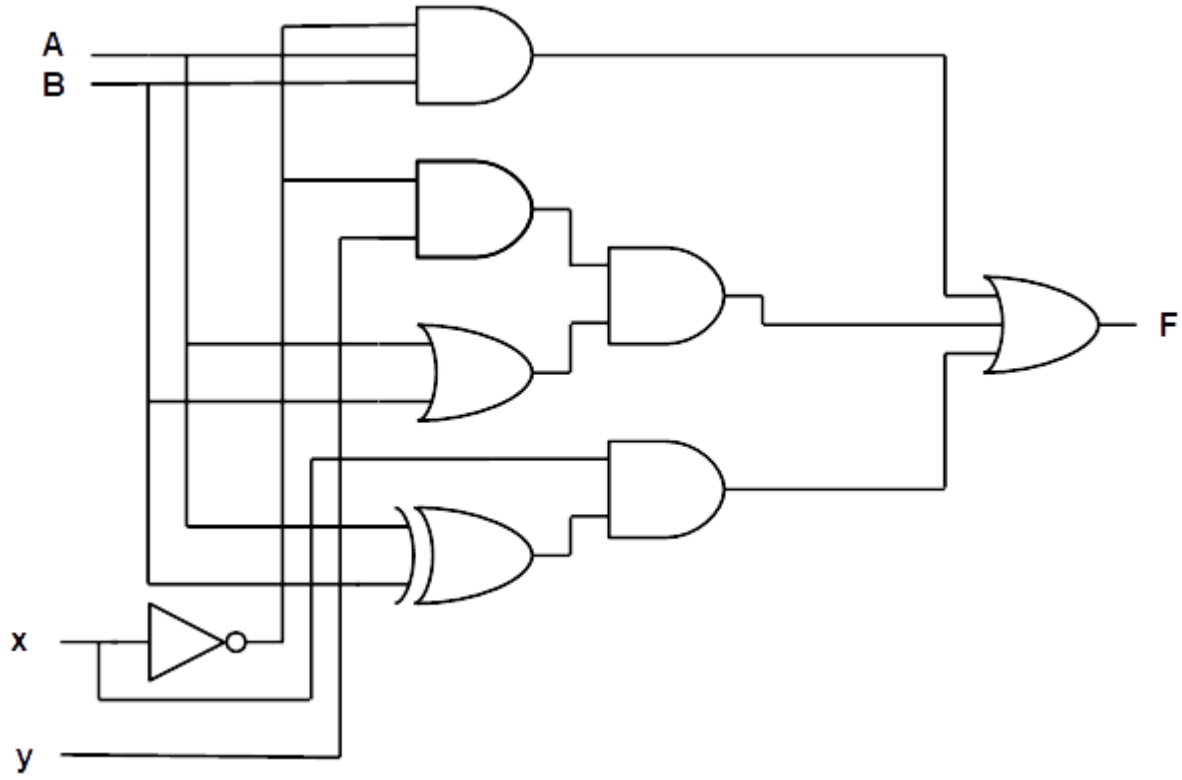


Figure 25 – 1-Bit Arithmetic Logic Unit

This is a simplified example; a real ALU truth table would be larger and more complex, especially for multi-bit ALUs or those supporting a wider range of operations.

Two-Bit ALU

A 2-bit ALU processes 2-bit inputs and can perform various operations like AND, OR, addition, and subtraction. Below is a simple example truth table for a 2-bit ALU with operations like AND, OR, addition, and subtraction.

Here is the truth table for the 2-bit ALU:

Op	A	B	Result	Carry/Borrow
00	00	00	00	0
00	00	01	00	0
00	00	10	00	0
00	00	11	00	0
00	01	00	00	0
00	01	01	01	0
00	01	10	00	0
00	01	11	01	0
00	10	00	00	0
00	10	01	00	0
00	10	10	10	0
00	10	11	10	0
00	11	00	00	0
00	11	01	01	0
00	11	10	10	0
00	11	11	11	0
01	00	00	00	0
01	00	01	01	0
01	00	10	10	0
01	00	11	11	0
01	01	00	01	0
01	01	01	01	0
01	01	10	11	0
01	01	11	11	0
01	10	00	10	0
01	10	01	11	0
01	10	10	10	0
01	10	11	11	0
01	11	00	11	0
01	11	01	11	0
01	11	10	11	0
01	11	11	11	0

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

In this table:

- **A** and **B** are the inputs
- **Op** is the opcode to select the operation to perform

  - 00 for AND
  - 01 for OR
  - 10 for addition
  - 11 subtraction
- **Result** is the output of the ALU
- **Carry/Borrow** indicates if there was a carry (for addition) or a borrow (for subtraction).

Instruction Set and Opcodes

Microcontrollers and microprocessors run a series of sequential instructions called a program. The program is usually written in a high level language like C or C++. The high level language is compiled or translated by a software tool called a compiler. The compiler translates the series of instructions into machine code. The program may also be written in a lower level assembly language. A software tool called an assembler translates or assembles the series of assembly instructions into machine code. The machine code is stored in the processor's program memory.

Machine code is simply a file containing a series of ones and zeros. Some programs may be very short, for example, an infinite loop controlling a couple of output pins flashing some LEDs in a child's toy. Other programs may be very long and complicated to the computer controller in a car's engine to the guidance system on a missile. No matter how simple or complex all computer programs are translated into a file containing a stream of ones and zeros called machine code.

Machine code is interpreted by the processor's CPU. Depending on the processor's complexity, the instruction width can be 16, 32 or 64 bits wide. A processor with 16-bit wide instructions will, for example, fetch instructions 16 bits at a time. The instruction will be stored in the instruction register. It will be decoded by the CPU in which the control unit will configure the ALU for the particular operation (ADD two registers, for example) and configure memory and/or some registers for a read operation. The CPU will then execute the instruction (ADD two registers) and store the result in another register or in data memory. The following examples demonstrate how an instruction for the Atmel AVR family of microcontrollers is translated into machine-readable machine code [43].

Here is an example of an ADD instruction:

Instruction:

ADD (add without carry)

Operation:

Rd ← Rd + Rr (Rr – source register; Rd – destination register)

Syntax:

ADD Rd, Rr

Machine Code:

15 8 7 0			
0000	11rd	dddd	rrrr

For example, the instruction to add register R3 to register R2

ADD R2, R3

translates to the following binary machine code (note: d = 00010, r = 00011):

0000 1100 0010 0011

or in hexadecimal:

0C23

The code "0C23" is the machine code for the ADD R2, R3 instruction in the Atmel AVR instruction set [43]. Bits 10 through 15 (000011 in this case ADD without carry) is the opcode that would be fed directly into the control bits of the ALU. This instructs the ALU that this will be an ADD operation without carry.

Here is an example of an AND instruction:

Instruction:

AND

Operation:

Rd ← Rd ● Rr (Rr – source register; Rd – destination register)

Syntax:

AND Rd, Rr

Machine Code:

15 8 7 0			
0010	00rd	dddd	rrrr

For example, the instruction to AND register R3 to register R2

AND R2, R3

translates to the following binary machine code (note: d = 00010, r = 00011):

0010 0000 0010 0011

or in hexadecimal:

2023

The code "2023" is the machine code for the AND R2, R3 instruction in the Atmel AVR instruction set [43]. Bits 10 through 15 (001000 in this case AND) is the opcode that would be fed directly into the control bits of the ALU. This instructs the ALU that this will be an AND operation.

Here is an example of copy register (MOV or move) instruction:

Instruction:

Copy Register

Operation:

Rd ← Rr (Rr – source register; Rd – destination register)

Syntax:

MOV Rd, Rr

Machine Code:

15 8 7 0			
0010	11rd	dddd	rrrr

For example, the instruction to copy register R3 into register R2

MOV R2, R3

translates to the following binary machine code (note: d = 00010, r = 00011):

0010 1100 0010 0011

or in hexadecimal:

2C23

The code “2C23” is the machine code for the MOV R2, R3 instruction in the Atmel AVR instruction set [43]. Bits 10 through 15 (001011 in this case a move operation or MOV) is the opcode that would be fed directly into the control bits of the ALU. This instructs the ALU that this will be an move operation.

ALU Integration in the CPU

The ALU is tightly integrated with the CPU’s control unit, registers, and memory. The control unit sends instructions to the ALU, and the registers provide the operands. The result from the ALU is often stored back in a register or memory.

Cycles of the Central Processing Unit

The CPU performs all of the calculations that occur and is the primary component carrying out the microcontroller’s or microprocessor’s functions. The main building blocks of the CPU include the ALU, the control unit (instruction fetch and instruction decoder) and registers. Figure 26 shows a block diagram of the brains of a microcontroller: the CPU.

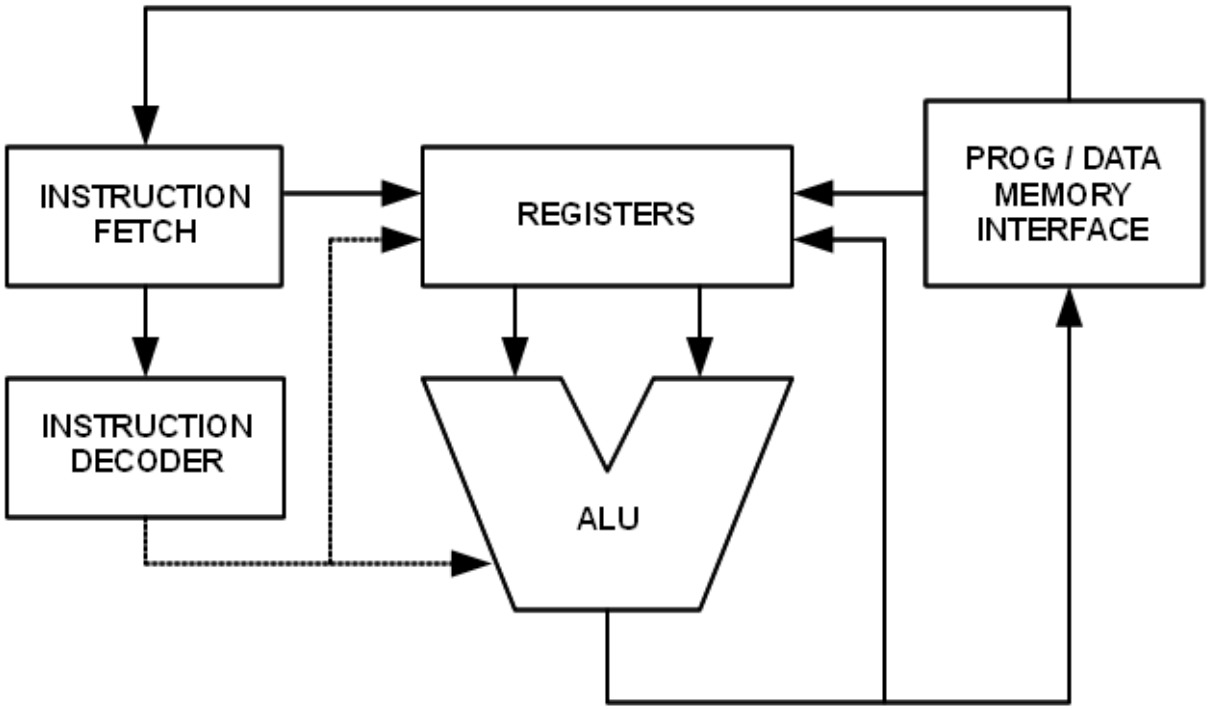


Figure 26 - Block Diagram of CPU Core

The control unit is a complete computational engine; it is a digital circuit that extracts instructions from memory, decodes the instructions and executes them. The CPU is the portion of the microcontroller or microprocessor that carries out the instructions of the stored computer program. The CPU contains a limited set of discrete states. The states are switched by millions of transistors on the processor chip.

The CPU's operation is governed by what is called the fetch-decode-execute cycle. The CPU executes a binary file that contains instructions in a particular sequence so that the processor steps through the instructions and executes (or runs) the program. The fetch-decode-execute cycle is analogous to loading, cocking and firing a gun.

The Fetch-Decode-Execute Cycle

The fetch-decode-execute cycle is the fundamental operational process of the CPU. Each stage of the cycle plays a critical role in instruction execution. This cycle

ensures the correct and efficient execution of programs. The fetch-decode-execute cycle of a CPU is analogous to loading, cocking, and firing a gun.

Instruction Fetch in a CPU

The fetch operation retrieves an instruction from program memory. The CPU stores the instruction in the instruction register. The location of the next instruction in memory to be executed is stored in a register called the program counter. The gun is now loaded.

The instruction fetch phase in a CPU is the first step of the fetch-decode-execute cycle, which is fundamental to the operation of a processor. During this phase, the CPU retrieves an instruction from memory to be executed. Here is a detailed explanation of what happens during the instruction fetch phase:

Key Functions of Instruction Fetch

Address Read

The program counter (PC) holds the address of the next instruction to be fetched from memory. At the beginning of the fetch phase, the CPU reads the address stored in the program counter. The program counter provides the address of the instruction to be fetched. This address is placed on the address bus, which connects the CPU to the memory.

Example:

- The PC value `0x00400000` is placed on the address bus.

Memory Read

The address from the program counter is sent to the memory unit to fetch the instruction stored at that location. The memory unit responds by sending the binary-encoded instruction back to the CPU. The CPU sends a read signal to the memory unit, indicating that it wants to read data from the specified address. The memory unit places the instruction from the specified address onto the data bus, which carries the data back to the CPU.

Example:

- The CPU sends a read request to the memory for address `0x00400000`.
- The memory retrieves the instruction stored at `0x00400000` and places it on the data bus.

Instruction Transfer

The fetched instruction is loaded into the instruction register (IR), a special register within the CPU that temporarily holds the instruction to be decoded and executed. The fetched instruction travels from the memory to the CPU via the data bus. The instruction is then loaded into the instruction register for processing in subsequent stages.

Example:

- The instruction (e.g., `0x20100004`, which might correspond to an "add immediate" operation) is transferred to the IR.

Program Counter Increment

After fetching the instruction, the CPU increments the program counter to point to the address of the next instruction in the sequence. This is typically done by adding the size of the instruction (e.g., 4 bytes for a 32-bit instruction) to the current program counter value. This ensures that the CPU will fetch the next instruction in the program during the next fetch phase. The CPU increments the PC to the address of the next instruction. This prepares the CPU for the next instruction fetch cycle.

Example: The program counter is incremented to `0x00400004` (assuming a 4-byte instruction length), ready to fetch the next instruction.

The instruction fetch phase is essential for retrieving and preparing instructions for execution, maintaining the correct sequence of operations, and ensuring the CPU can execute programs correctly and efficiently. The fetch phase ensures that instructions are fetched in the correct sequence, maintaining the flow of the program. By updating the PC, the CPU can follow the program's control flow, including handling jumps, branches, and calls. Efficient fetching is crucial for overall CPU performance, as delays in fetching can stall the entire instruction pipeline.

Instruction Decode in a CPU

After an instruction is fetched from memory it must be decoded or interpreted. The decoding operation determines how to configure the ALU for the particular operation. The gun is now cocked. The decode operation breaks up the instruction into pieces. The meaning of the different chucks of binary bits depends upon the processor's instruction set. The instruction set can be thought of as the processor's vocabulary. Part of the instruction to be decoded is called the opcode (or operation code). This indicates which instruction to perform, for example, an AND operation

and is the control signal that is fed into the ALU. The remaining parts include the operands for the operation. These are the inputs to the command. They can be a constant value, a register or a memory location.

An instruction decoder in a CPU is a crucial component responsible for interpreting and converting binary-encoded instructions fetched from memory into signals that control other parts of the CPU. Here's a detailed breakdown of its functions:

Key Functions of an Instruction Decoder

Fetch Instruction

The instruction decoder receives the instruction fetched from memory.

Example:

- The CPU fetches an instruction from memory, say `ADD R1, R2, R3`, which means "add the contents of registers R2 and R3 and store the result in register R1".

Decode Opcode

The instruction decoder examines the opcode (operation code) portion of the instruction, which specifies the operation to be performed (e.g., add, subtract, load, store).

Example: The instruction decoder identifies the opcode for ADD.

- It generates control signals to:
    - Configure the ALU for addition.
    - Select the contents of registers R2 and R3 as inputs to the ALU.
    - Direct the ALU's output to register R1.

Generate Control Signals

Based on the opcode, the instruction decoder generates specific control signals. These signals direct various components of the CPU, such as the ALU, registers, and memory, to carry out the specified operation. Control signals may include activating specific registers, selecting the ALU operation, and controlling data paths.

Determine Operand Locations

The instruction decoder identifies the locations of the operands (data to be operated on). Operands can be in registers, memory, or included as immediate values within the instruction.

Control ALU Operations

The instruction decoder sends control signals to the ALU to specify the type of arithmetic or logical operation to be performed.

Coordinate Data Paths

The instruction decoder manages the data paths by activating multiplexers and buses that direct data flow between the CPU's components. It ensures that data is routed correctly from registers or memory to the ALU and back.

Handle Instruction Variants

The instruction decoder can handle different instruction formats and lengths, especially in complex instruction set computers (CISC) or reduced instruction set computers (RISC). It ensures that all parts of multi-step instructions are properly decoded and executed.

Instruction Execute Cycle in a CPU

After an instruction is fetched and decoded it is then executed. The gun is now fired. During the execute phase different parts of the CPU are connected to perform the desired operation. For example, during an AND operation, the ALU will be configured to perform a logical AND. The two operands (a constant, a register location, or memory location) will be presented to the inputs of the ALU. The control unit will present the proper binary code to the ALU’s control section to prepare it for an AND operation and the output of the ALU will contain the result. The result is then written to an internal CPU register or some memory location.

At this phase the program counter is incremented and the CPU is reconfigured to fetch the next instruction and complete the next fetch-decode-execute cycle. In our analogy, the gun ejects its spent cartridge and readies itself to fire again.

History

The 74181 integrated circuit is a 4-bit ALU implemented by Texas Instruments in 1970. It was the first complete ALU on a single chip and used as the ALU in the CPUs of many minicomputers in the 1970s. Before chips like the 74181, CPUs in the 1960s were constructed using discrete logic gates on circuit boards. Chips like the 74181 were used when CPUs were built using a combination of many discrete chips, before CPUs were single chips like they are today.

Early computers contained ALUs that were built out of a large number of simple gates. In 1970, Texas Instruments built the 74181 integrated circuit, which put a full 4-bit ALU into one fast TTL chip. This chip provided 32 arithmetic and logic functions on two 4-bit words.

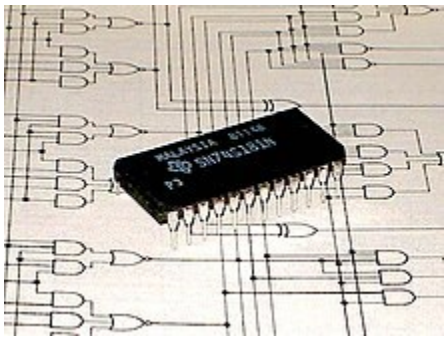


Figure 27 – 74181 4-bit ALU from Texas Instruments

The 74181 is a 7400 series medium-scale integration (MSI) transistor-transistor logic (TTL) integrated circuit. It contains the equivalent of 75 logic gates and packaged as a 24-pin DIP package. The 74181 is a 4-bit wide ALU that can perform add, subtract, decrement operations with or without carry. It can perform logic operations such as AND, NAND, OR, NOR, XOR, and bit shifting operations. It can perform a total of 16 arithmetic and 16 logical operations on two four-bit words. Multiply and divide are not possible directly. Figure 28 show a complete schematic of the 74181.

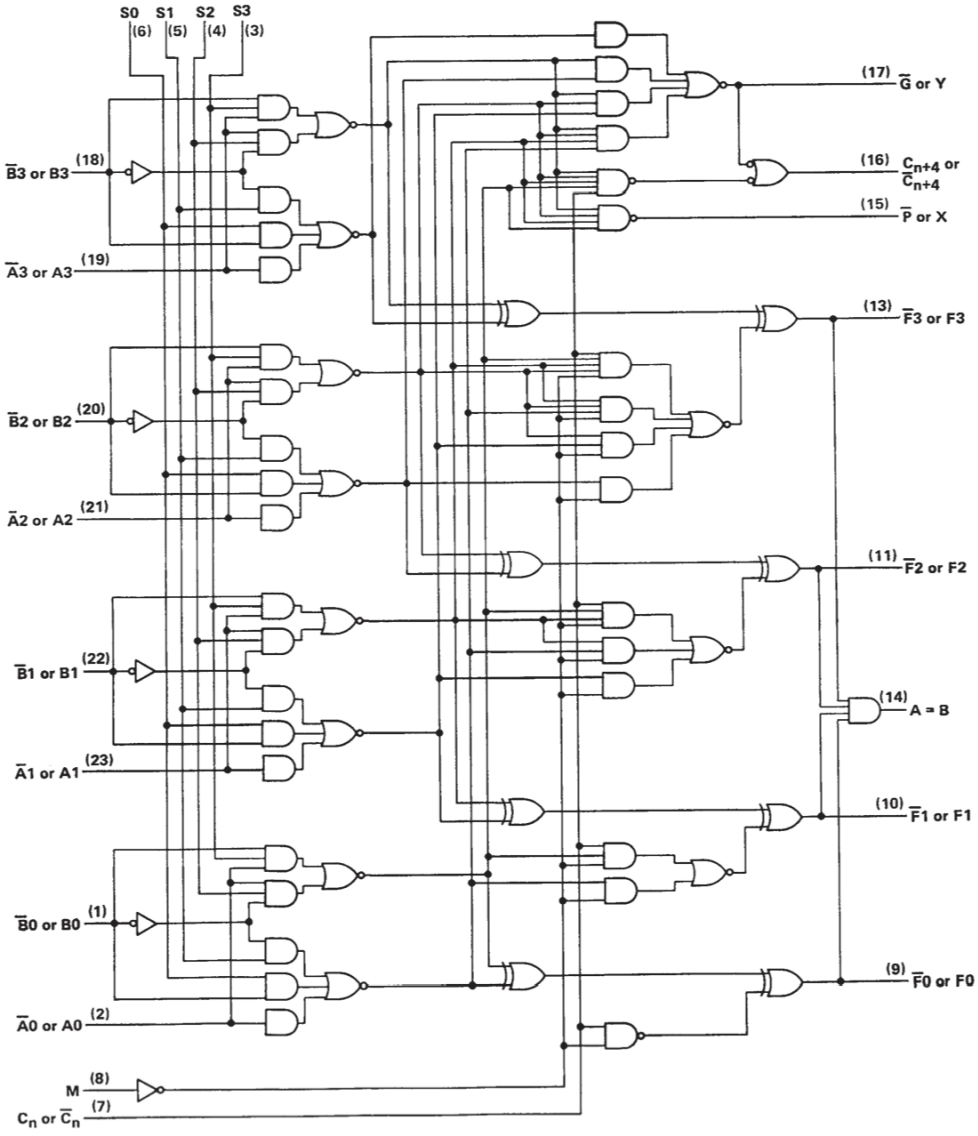


Figure 28 – 74181 4-bit ALU Schematic (from the TI datasheet)

The arithmetic and logic operations are selected by the four function select lines (S0, S1, S2, S3) as well as the mode control input (M) and the carry input line (Cn). This is shown in the truth table in Figure 29. Together, the function select lines and the mode control input make up the opcode that would appear in the machine code that was compiled (or assembled) for a computer that contains this ALU.

SELECTION				ACTIVE-LOW DATA		
				M = H	M = L; ARITHMETIC OPERATIONS	
S3	S2	S1	S0	LOGIC FUNCTIONS	Cn = L (no carry)	Cn = H (with carry)
L	L	L	L	F = $\overline{A}$	F = A MINUS 1	F = A
L	L	L	H	F = $\overline{AB}$	F = AB MINUS 1	F = AB
L	L	H	L	F = $\overline{A}$ + B	F = A$\overline{B}$ MINUS 1	F = A$\overline{B}$
L	L	H	H	F = 1	F = MINUS 1 (2's COMP)	F = ZERO
L	H	L	L	F = $\overline{A + B}$	F = A PLUS (A + $\overline{B}$)	F = A PLUS (A + $\overline{B}$) PLUS 1
L	H	L	H	F = $\overline{B}$	F = AB PLUS (A + $\overline{B}$)	F = AB PLUS (A + $\overline{B}$) PLUS 1
L	H	H	L	F = $\overline{A \oplus B}$	F = A MINUS B MINUS 1	F = A MINUS B
L	H	H	H	F = A + $\overline{B}$	F = A + $\overline{B}$	F = (A + $\overline{B}$) PLUS 1
H	L	L	L	F = $\overline{A}$B	F = A PLUS (A + B)	F = A PLUS (A + B) PLUS 1
H	L	L	H	F = A $\oplus$ B	F = A PLUS B	F = A PLUS B PLUS 1
H	L	H	L	F = B	F = A$\overline{B}$ PLUS (A + B)	F = A$\overline{B}$ PLUS (A + B) PLUS 1
H	L	H	H	F = A + B	F = (A + B)	F = (A + B) PLUS 1
H	H	L	L	F = 0	F = A PLUS A‡	F = A PLUS A PLUS 1
H	H	L	H	F = A$\overline{B}$	F = AB PLUS A	F = AB PLUS A PLUS 1
H	H	H	L	F = AB	F = A$\overline{B}$ PLUS A	F = A$\overline{B}$ PLUS A PLUS 1
H	H	H	H	F = A	F = A	F = A PLUS 1

‡Each bit is shifted to the next more significant position.

Figure 29 – 74181 ALU Truth Table (from the TI datasheet)

The inputs and outputs of the 74181 chip can be overlayed on the ALU symbol shown in Figure 30. The Input 1 is the 4-bit word A and the Input 2 is the 4-bit word B. The Control signal is the combination of the function select lines (S0, S1, S2, S3) as well as the mode control input (M) and the carry in input (Cn). The Result is the 4-bit function word F as well as the carry output (C_{n+4}).

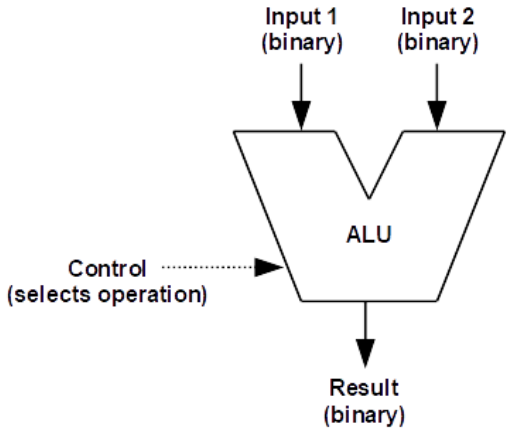


Figure 30 – ALU Symbol

Summary

The arithmetic logic unit is the central core of a central processing unit. Digital systems such as the computer central processing unit use a numbering system based on the number two called the binary system. The ALU is a digital circuit that performs arithmetic and logical operations on binary numbers. All microcontrollers contain an arithmetic logic unit (ALU). It is a fundamental building block of the CPU.

The ALU performs arithmetic and logic operations on two binary numbers resulting in another binary number. The numbers (the operands and the result) are represented using two's complement format. Twos' complement allows subtraction to be performed by adding the complement of a number instead of subtracting the number. Arithmetic operations such as addition, subtraction, multiplication, and division allow CPUs to perform essential numerical calculations required by most software applications, ranging from simple spreadsheets to complex scientific simulations. Arithmetic operations are used to compare numeric values, which is essential for program control flow. Addition and subtraction (adding the twos' complement of another number) is performed by an adder circuit. An adder is a digital circuit (AND, OR, and XOR gates) that adds two numbers. An adder is sometimes called a summer (one that sums), and is an integral part of a microprocessor's arithmetic logic unit (ALU).

Logic operations allow for the manipulation of data, such as setting, clearing, bit flipping, and masking bits. These operations are fundamental for tasks like data encoding, data decoding, encryption, error detection, and error correction. Logic operations allow for control flow mechanisms made possible by conditional statements and loops to determine the flow of program execution. Branching operations by comparing two values to decide which branch of code to execute involves logic operations.

The control bits on an ALU is called the opcode. It is short for operation code. It controls the operation of the ALU. The opcode will instruct the ALU to perform either an add, subtract, AND, OR, etc. operation. There are two data inputs to the ALU, called the operands. The operands are what are operated on and the result appears at the output of the ALU. Machine code is interpreted by the processor's CPU. Depending on the processor's complexity, the instruction width can be 16, 32 or 64 bits wide. A processor with 16-bit wide instructions will, for example, fetch instructions 16 bits at a time. The instruction will be stored in the instruction register. It will be decoded by the CPU in which the control unit will configure the

ALU for the particular operation (ADD two registers, for example) and configure memory and/or some registers for a read operation.

In the early days of computers, pre-1970, the ALU (part of the CPU) would be constructed of discrete integrated circuits on a circuit board. In 1970 Texas Instruments implemented the first complete ALU on a single chip. The 74181 integrated circuit is a 4-bit ALU used in the CPUs of many minicomputers in the 1970s. Before chips like the 74181, CPUs in the 1960s were constructed using discrete logic gates. The 74181 is a 4-bit wide ALU that can perform add, subtract, decrement operations with or without carry. It can perform logic operations such as AND, NAND, OR, NOR, XOR, and bit shifting operations. It can perform a total of 16 arithmetic and 16 logical operations on two four-bit words. In modern microcontrollers and microprocessors, ALUs are integrated as a complete circuit on the same die as the CPU.

References

1. "Arithmetic logic unit – Wikipedia, the Free Encyclopedia." Visited 8 July 2024. <https://en.wikipedia.org/wiki/Arithmetic_logic_unit>

2. "Atmel AVR Instruction Set Manual." November 2016. <https://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf>

3. ChatGPT, GPT-4, OpenAI, knowledge cutoff October 2023, chat.openai.com.

4. "IC 741181 (4-bit slice arithmetic logic unit)." Visited 6 September 2024. <https://www.igelectronics.com/products/a13d88f3c5/17268100000000211015>

5. "SN54LS181, SN54S181, SN74LS181, SN74S181 Arithmetic Logic Units/Function Generators." Visited 9 September 2024. <https://www.ti.com/lit/ds/symlink/sn54ls181.pdf?ts=1725050679943&ref_url=https%253A%252F%252Fwww.google.com%252F>

6. Strain, Mark, P.E., "Digital Logic Design: Combinational Logic." 2016.

7. Strain, Mark, P.E., "Microcontrollers - An Introduction." 2010.

