Computer System Principles and Practice

Examine how the architecture and how data is represented and manipulated in a microprocessor

Written by:

Name: Mohammed Ghalib Al-Hosni

Number: 1615

Class: 8A

Report aims

Through reading the following report, the reader is provided with two key aspects of computer science and engineering. The first being the way that digital circuits and how a computer system in general operates from a low level through the use the binary numerical system. This section would discuss what exactly is binary and its importance in computer software and system architecture, how the computer works with such a system from a mathematical perspective, and how binary is universally utilized throughout some of the varying application of computers, with further detail going into character writing and storage operations using standard formatting processor such as ASCII and Unicode.

The other significant segment of the report constitutes information relating to the core of any given computer system, the Central Processing Unit (CPU). Said segment would analyze the varying components found within the CPU from both their operation and theoretical working process as well as their circuitry and from an electrical point of view. Moreover, this section would go into how exactly the CPU utilized said components in order to operate through the use of the instruction cycle and what possible techniques could be utilized in order to enhance the processor's performance and throughput capabilities.

Table of contents

The binary system and data manipulation

The following section will discuss the one of the fundamental aspects of computer science and engineering in how computers – and the components within – transfer and convey data using low-level methods of communication understood only by said systems, with said method being the binary system.

In addition to that, the section is also going to go through how the aforementioned binary system is utilized in manipulating data found within a given computer system in order to make use of it in more functional and practical ways to accomplish the tasks required by the system. The final point of analysis is regarding the methodologies used in organizing and converting data in more recognizable and humanely understandable ways.

Binary fundamentals:

Binary – Hardware perspective:

Binary itself can be considered as the "core" language that the software within a given computer system uses in order to perform instruction and allow for communication between the software and hardware aspects of the system as well as between the internal and external components of the system as well.

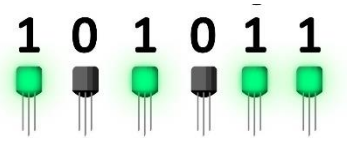


Figure 1 - Transistrs as binary digits

Now, from a hardware perspective, components are mainly made up of electrical transistors, that can either be on/high or off/low, with low meaning that a voltage of zero is going through the transistor, and high generally being a voltage of around five volts. Said voltage can oscillate and does not necessarily have to be a high or a low voltage, however, if the transistor is intended to provide a low signal, then the voltage its outputting would have to be below a given range, with said range being from 0 to 0.8 volts according to (Huang, 2022)*. And likewise, if the transistor is meant to provide a high signal, then it would have to output a range of 2.7 to 5 volts. Anything in between 0.8 and 2.7 volts would take on the state of the previous value, in that if high was the previous state, if voltage goes from 5 volts to 1.5 volts, then it would remain as high, and if it goes from 0.7 to 1.5 volts, then it would remain as low. Said decision on whether the transistor is low or high is primarily done by the computer's Central Processing Unit (CPU), which takes in the voltage and decides on whether the transistor is providing a high or a low value.*

From here, circuits within the computer system are used specifically to perform the "logic" of the binary values. From a very basic standpoint, these logics are mainly "And" and "Or" logics, but when used in large and complex configurations with in combination with the aforementioned transistors, this allows computers to practically perform any form of logic that the computer system requires, with this being done through the use of special diodes known as logic gates. An example of this is a part of the CPU known as the Control Unit is used specifically to decide what a certain binary values are meant to be used for and sends it to the relevant part of the CPU to do further work on it. Meaning when the CPU receives a given binary value, it would initially check the first couple of bits in the value that would represent the instruction on whether the CPU is meant to add or subtract the values, with said checking process being done through Boolean logic (i.e. 1100 is add while 0011 is subtract. These are just example values, not the real instruction set binary format). From there, the CPU would send the remaining binary values (not the shown instruction sets) to the relevant part of the CPU to do the rest of the work. This was just a brief example, and it would be further discussed and analyzed further on in a pertinent section of the report.

Binary – Mathematical perspective:

Now, the way that this all ties into the binary system is that the binary code language operates through values called "bits", with each bit being either equal to a value of one or zero, with each bit's value being determined by – as might be expected – the voltage range of the transistor. From a computational or mathematical perspective, each the number of bits given can account for a given number. Our day to day counting system utilize what is known as a "base 10" system, in that each time you increase the number's place value, it increases by ten. For example, 01 is 1 x 10^0, while 10 is 1 x 10^1, and 359 is 3 x 10^2 + 5 x 10^1 + 9 x 10^0, which gives 359. In binary, a base 2 system is utilized, and the number's value can only be a one or a zero. If we look at an example of that, this can be given as 00, which will be 0 x 2^0 + 0 x 2^1, while 1010 will be 0 x 2^0 + 1 x 2^1 + 0 x 2^2 + 1 x 2^3, which would equal to 10. To conclude this, the usual decimal system uses an iterating base 10 system that can be multiplied with a number range from 0 to 9, while a binary system uses an iterating exponent base 2 system that can only be multiplied by 0 or 1.

***Decimal values* - 359**		
3 → (third digit = 2)	5 → (second digit = 1)	9 → (first digit = 0)
3 x 10^2 = 300	5 x 10^1= 50	9 x 10^0 = 9
Total = 300 + 50 + 9 = **359**		

***Binary value* – 1101**			
1 → (fourth digit = 3)	1 → (third digit = 2)	1 → (second digit = 1)	1 → (first digit = 0)
1 x 2^3 = 8	1 x 2^2= 4	0 x 2^1 = 2	1 x 2^0 = 1
Total = 8 + 4 + 0 + 1 = **13**			

Additionally, the maximum number of bits that can be given simultaneously general found on most modern computers is between 32 to 64 bits, in that 64 copper lines are ran next to one another, with said lines being known as the "busses" of the computer system. These allow for faster transportation of data as more information can be provided simultaneously and therefore reducing time. This works in that instead of an 8 bit system where the maximum value is 11111111, the maximum value in a 64 bit system would be 64 bits in a single data transmission.

Binary – Software perspective and working with drivers:

Although binary values can range up to extremely high numbers, with 64-bit systems being able to provide 2^{64} values, which translates to 18 quintillion numbers. However, manufacturers of different produces and software cannot account for all those values, and different manufacturers using the same number for a specific operation would be inevitable unless a given standard is utilized, and even so, it is not guaranteed. As such, to account for the number of operations that different devices require, a software application is utilized to utilize standard binary formats for the device's specific needs, with said software being known as device drivers.

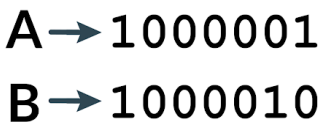


Figure 2 - Binary usage in characters

Basically, binary values themselves mean nothing to the computer system unless specifically organized and dedicated for their application and purpose. In that the same binary value could be used for outputting specific sound frequencies, a given RGB color on the monitor, or on the letter/character that should be outputted on the screen such as the letters being typed on a keyboard. In that the value 0011 could be used for all three purposes and more. This is due to the combined effort and working process of the CPU's core circuitry and Boolean logic working with the operating system as well as the device's specific driver software and the way it interprets instruction sets.

Why binary:

As mentioned by (Williams, 2019), *a binary – or a base 2 system – was initially utilized as transistors in older computers had to be relatively large, and so it was simply more efficient to use such a system in comparison to other numerical systems such as base three or the base ten that we use in our day to day lives. However, as computer engineering and architectures progressed, nowadays it is possible to store millions upon millions of transistors within a single CPU, however, binary systems are nonetheless dominant in the market.*

There are mainly two reasons for that, the first of which being that the binary system is simply easy for computers to understand and utilizing a base 3 system would further complicate matters from a mathematical standpoint, while binary systems are already established easy-to-understand forms of communication between computer systems. Furthermore, form an electrical hardware standpoint, transistors and Boolean logic work through true or false systems, which is how the binary systems functions, either one/true, or zero/false. This means that if a switch had to made to a more advanced system such as a base 3, a complete change in the hardware would have to be done, a shift as drastic as that would be very risky from a practical standpoint, especially in a competitive market.

With that being said, Ternary (base 3) systems do exist, and do have advantages over binary such as being able to store more data in a smaller number of "units" or "bits", which would significantly help in terms of efficiency and being able to access more instruction sets. Another important numerical system that is currently being experimented and worked on is the quantum or quaternary system, which acts as the fundamental working process for quantum computing, a system in which a value can both be true and false at the same time. The math and overall working process of both ternary systems and quantum computing is rather complex, and explaining it would cause the report to go off topic, however, it should be noted that both systems – especially quaternary ones – have shown significantly better performance compared to binary systems, with quantum computers being 158 million times faster than already existing "supercomputers" as stated by (Smith, 2022). *However, as of 2022, quantum computers are extremely expensive and reach costs of around 10 million USD. As such, until an efficient use and method of production has been acknowledged and realized for them by the industry, binary computing would remain the dominant methods in the market due to its simplicity and its substantial footprint in the computer science and engineering realms which is preventing drastic and sudden changes to the industry.*

Binary storage:

The methodologies utilized in order to store binary numerical value differs from one system of electronic storage to another. Said systems range from volatile and non-volatile, with non-volatile storage systems retaining memory even after the computer system has been shut down with the opposite being said about volatile memory storage methods. The stated systems would be further discussed later on in the report, especially throughout the registers section. Overall, binary Ais stored in memory as an electrical signal, and each single numerical value of binary (a 1 or a 0) is known as a "bit" as mentioned several times earlier.

Additionally, eight bits is then called a byte, with the reason for that being the fact that historically, 8 bits is the binary size utilized in order to contain the required identification number for given characters on the keyboard, which later on become the standard utilized for every day usage, not to mention that many other application also require 8 bits to operate such as the RGB system, therefore making it a very convenient naming structure. From there, data is iterated through standard prefixes using the byte such as kilo, -mega, giga, tera, and peta-, with a kilobyte being 1000 bytes or 8000 bits. Another one utilized based on the same concept as a "byte" is a "nibble", which represents 4 bits, however, this has not been as popular as a the byte and is not really utilized very much.

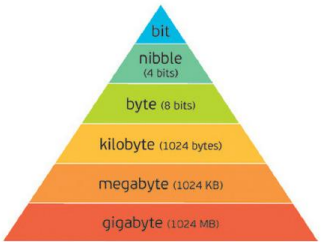


Figure 3 – Bit size interval pyramid

High- and low-level languages and their relation to binary:

Programming languages are at their core methods of transcribing – relatively – humanely understandable text-based formats of computer-control notations used in giving instruction to the computer systems, with said notations then translated into a binary format that the computer itself can understand and work with, with the translation being done through a compiler software application that takes in the aforementioned notations and directly translates into binary and gives it to the computer system. Now, said languages can be separated into a variety of different categories, however, one of the main categories that is important when talking about the translation process into binary would be whether the language is one that is considered low-level or a high-level language.

The difference between the two types of languages stems from how much translation would have to actually be done in order to fully translate the language into a usable binary format. High-level languages utilize a much more humanely understandable format of writing, and does all the memory management work automatically, while low-level languages use much more "obscure" syntax that requires very little to be compiled into binary, and would also necessitate that the user manually manages when data goes into or out of memory and how it moves throughout the memory architecture of the system, and although this might result in more relatively "incomprehensible" syntax, it is much easier on the computer system and instructions are almost fully done and require little to no automation during translation/compiling, therefore decreasing the overall compile time.

Overall, the "level" of a language can range from very low level languages such as "C", "C++", and on the very low end, "Assembly" which requires almost full manual management of the memory system and the instruction cycles, which would be further discussed later on in the report, with the lowest language utilized by the computer being binary itself. On the opposite end of the spectrum, high level languages such as C#, Java, or Python are much more comprehensible relative to human language standards and require very little to no memory or "garbage" management, but are also much slower, with Python having to go initially get translated into C, which in turn translated into binary, making it relatively extremely slow at compile. Basically, the main thing that separates the language levels is how much machine code (binary) is abstracted, with high levels languages have heavy abstractions, while low levels having to work a lot with binary itself and – although are humanely readable – are much less abstracted than high level languages.

A perfect example of the speed efficiency when it comes low level languages is a game by the name of "Rollercoaster Tycoon", a game that released in March of 1999 and was developed by single-handedly by Chris Sawyer, with the entire game being coded in Assembly. In 1999's standards, the game requires immense levels of computing power, however, since it was entirely coded in Assembly, Sawyer was able to efficiently manage memory to the point that even the weakest systems at the time were capable of running the game at relatively impressive speeds.

Figure 4 - Level of abstraction among programming languages

Mathematical operations:

Circuitry:

From an electrical and electronic standpoint, the way that any of the following mathematical operations would work is through the use of logic gates. Said gates would initially determine the actual mathematical procedure that is meant to be carried out with the values, and from there another logic gate system is added to determine how the ones and zeros are meant to interact with one another based on the required mathematical operation. An example could be a signal being generated based on the user's input would cause the operation to be addition, and if said signal was unavailable then it would cause the operation to be subtraction. The initial electrical signals that go into said logic gates are of course being generated from transistors.

The an instruction has been decided (e.g., addition or subtraction), it would then be stored in memory and be called upon and utilized as the system's user requires. What was explained is extremely brief and might seem vague at the moment, however, this would be discussed in much further detail later on in the report when talking about the "Arithmetic Logic Unit" or ALU.

Binary addition:

Since binary is simply the matter of adding ones and zeros to one another, binary addition is rather simplistic as the number of possible combinations that can occur are limited to only four combinations, with them being the following:

Operation	***Result***
0 + 0	0
1 + 0 (and vice versa)	1
1 + 1	0 with a carry of one
1 + 1 + 1 (during carries)	1 with a carry of one

To go into more detail, the way that binary addition operates is similar to how base 10 long addition works, in that it can be visualized as the two values being placed over one another, with the addition being from right to left. Adding zeros simply results in a zero, and adding a zero with a one will also result in a one. However, whenever two ones are added to each other, the resultant would actually be zero and a one would be carried to the adjacent value, and so resulting in three values being added, the initial two values plus the carried one. If a one is carried into zeros, then it would be one, if carried into a zero and a one, the number would be carried again. Now, if the one was carried into two ones, then it would result in a one and another one would be carried to the adjacent bit. Adding three numbers involves the same process, but a one is carried each time two ones are added regardless of current position.

Examples:

1001 +0100				
First value	1	0	0	1
Second value	0	1	0	0
Answer	1	1	1	1
Description	1 + 0 = 1	0 + 1 = 0	0 + 0 = 0	1 + 0 = 1

0011 + 0011				
Carry	N/A	1	1	N/A
First value	0	0	1	1
Second value	0	0	1	1
Answer	0	1	1	0
Description	0 + 0 = 0	1 + 0 + 0 = 1	1 + 1 + 1 = 1 (with a carry of one)	1 + 1 = 0 (with a carry of one)

Binary subtraction:

Just like in binary addition, the general method of binary subtraction utilized a limited number of "rules" that are followed in order to complete calculations, with the number of said rules being the same as binary addition, four. With them being the following:

Operation	***Result***
0 - 0	0
0 - 1	0 (with a borrow of one)
1 - 0	1
1-1	0

Three of the shown rules are very self-explanatory and follow the same basic principles as subtraction in base 10. However, when it comes to subtracting one from zero, the same process of "borrowing" that is used in base 10 is also utilized here. A useful way of thinking about it is if the number being borrowed from (the adjacent number) is a zero, said zero turns into "two", but if it was a one, then it turns into zero. The same thing is also utilized when changing the zero from (0 – 1), as it would (2 – 1 = 1). It can be thought of as whenever a zero is crossed, then it would turn into two. Moreover, if a number had to borrow from a zero, then the zero would look for the nearest one, borrow from it, from there it would turn into a two and become a one in order to be borrowed from.

Examples:

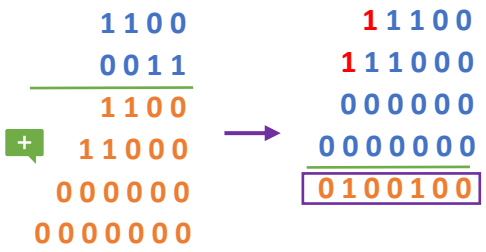
1100 + 0101				
Borrow	0 (gives to the right)	0 (gives to right) 2 (takes from left)	2 (takes from left)	2 (takes from left)
First value	1	1	0	0
Second value	0	1	0	1
Answer	0	1	1	1
Description	0 - 0 = 0	2 - 1 = 1	2 - 1 = 1	2 - 1 = 1

0110 - 0100				
First value	0	1	1	0
Second value	0	1	0	0
Answer	0	0	1	0
Description	1 + 0 = 1	1 - 1 = 0	1 - 0 = 1	0 - 0 = 0

Binary multiplication:

The process in which the binary multiplication process works is almost exactly the same as how long multiplication works in base ten, in that you would take the bits, multiply them by each value (1 x 1 = 1 | 0 x 1 = 0 | 0 x 0 = 0). Each time you increase a bit level, you add a zero the beginning of the line. And then you add the numbers as you would in binary addition.

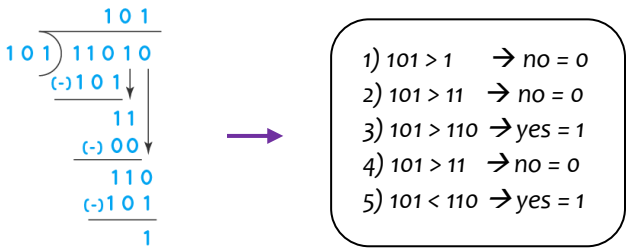
Example:



Binary division:

Similar to binary multiplication, the methodology utilized in order to divide binary values is the exact same as the one utilize in "long division" for values of base 10. This means that in order to divide binary values, the number dividing is going to be compared to the first digit/bit of the number its dividing, and if it is bigger, then the quotient would be zero, while if the dividing number is smaller, the quotient would be one. From there, the (quotient x dividend) would be subtracted from the binary value that it was just compared to. This is then repeated but the dividend is compared to the resultant received after subtraction, and with each iteration a bit from the original value is placed next to the resultant.

Example:



Binary arithmetics:

Two's compliment arithmetic:

Two's compliment is the arithmetic in computers responsible for working with any negative numbers in the system. The way that two's compliment works is very similar to how binary bits are normally represented, which is also known as one's compliment, however, the difference lies in that the last bit represents a negative number. For example, a binary value of 1010 in one's compliment is equal to 10 (0 + 2 + 0 + 8), however, in two's compliment, the last bit would be negative, and so the final value would be -6 (0 + 2 + 0 – 8).

A problem with two's compliment is that it limits the number of values that it decreases the overall efficiency of the system. In that normally, with 4 bits a system would be able to calculate from 0 to 15, while now, it's from -8 to 7. Although it is still a total of 15 possibilities, it nonetheless does limit the maximum possible value. From a circuitry standpoint, the way that two's complement works is that it the normal positive value of the number would be taken, all digits would be inverted, and a one would be added to the number. For example, to get -6, the value would first be 0110, which is positive six, from there it would get inverted to 1001, and one would be added to the value, which – if binary addition rules are to be followed – would provide a resultant of 1010 (0 + 2 + 0 – 8 = -6).

Example using: -7				
Stage 1	**Flip the values**			
One's compliment	0	1	1	1
Flipped value	1	0	0	0
Stage 2	**Add one**			
Flipped value	1	0	0	0
One in binary	0	0	0	1
Answer	1	0	0	1

The shown answer is -8 from the last bit and 1 from the first bit. -8 + 1 = -7

Floating point arithmetic:

Floating point arithmetic – as the name might suggest – is the mathematical manipulation that binary (or base 2) has to deal with in order to work with numbers that contain decimals within them. The reason that computers and binary in general struggles when it comes to floating points is due to the fact that certain decimals do not have an exact binary representation, and so the only possible way of displaying them is through the use of approximations. As such, some precision in the process can be lost due to this.

In base 10, fractional values can range anywhere from 1/1 to 1/9, with the numbers then getting iterated as 1/10, 1/100, 1/1000, and so on. However, in binary, since only two values are available, the way that that fractional values work are as such: 1/2, ¼, 1/8, 1/16, and so on, with each iteration being a power of two.

2^{-1}	2^{-2}	2^{-3}
$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$

Figure 5 - Binary bases with negative exponents

Additionally, base 10 notation of fractional numbers would be given as a mantissa and an exponent, with the exponent being of base 10. This would look something like the following 5.05 x 10^5, which would equal to 505000 as the floating point would be moved 5 times to the right. In binary, a very similar approach is utilized. According to (Computer Science, 2019)*, the way that this would be showcased is that the mantissa would be the first given number of bits, and then the remaining would represent the exponent, and these would be showcased in two's complement (i.e. the first digit represent whether the value is negative or positive). An important point to mention is that the floating point is initially placed right after the first bit* → 0.0011 x 2^3 = 1.1 *(the floating point moved three spaces to the right).* 1.1 = (1 x 2^0) + (1 x 2^{-1}) = 1 + 0.5 = 1.5 (in denary)

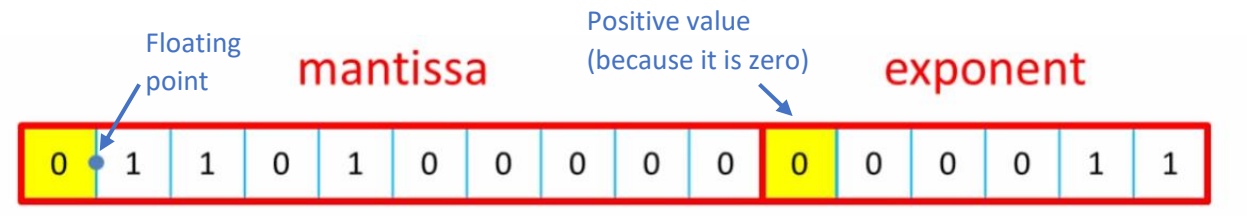


Figure 6 - How floating point are represented in binary

Floating point example					
Stage 1	**Before moving the floating point**				
Mantissa	0	1	1	1	0.111
Exponent	0	0	0	1	Equals $x2^1$
Stage 2	**After moving the floating point**				
Binary answer	0	1	1	1	01.11
Decimal answer	0×2^2	1×2^1	$1 \times 2^{1/2}$	$1 \times 2^{1/4}$	Total = 1.75

The total was gotten using: $(1 \times 2^1) + (1 \times 2^{1/2}) + (1 \times 2^{1/4})$ → 1 + 0.5 + 0.25 → **1.75**

Alternatives numerical systems:

What is the point of different bases:

The reason that various number systems are utilized is to be able to accommodate for various functions and requirements needed by the computer systems or us as humans in order to understand numerical representations easier. For example, as mentioned earlier, the binary system is used by computers due to its efficiency and ease of implementation from an electrical standpoint. In computer systems, there are 4 different number systems or bases that are heavily utilized, with them being denary, hexadecimal, octal, and – of course – binary.

Since the following section will discuss conversion from different numerical systems to binary, it should also be noted that computers never convert from any other base system to binary as a computer's native or default numerical system is in binary, and so in computers, the only conversion that would occur is from binary to denary/octal/hexadecimal and not vice versa.

Denary:

Utilized base: *Denary – also known as decimal or base 10 – is the most known numbering system utilized. As stated earlier, the way that this number system works is that it goes through numbers from zero to nine, and after exceeding said numbers, the number iterates by adding another digit next to the initial one. From a mathematical perspective, this can be seen as multiplying by a base 10 value with an exponent, with said exponent starting from zero for the first digit and iterating as the number of digits increase. An example that can be given is 158, which can be split into* $(8 \times 10^0) + (5 \times 10^1) + (1 \times 10^2) = 158$.

Application: *From a computer system stance, denary or decimal is almost entirely useless. With that being said, it is also one of the fundamental considerations when it comes to computer architecture and software as any binary numbers working within the computer would eventually have to be converted into decimal as that is the standard numerical system utilized for everyday life.*

Denary to binary: *A common method utilized when converting from denary to binary is to subtract the biggest possible power of two from the denary number without getting a negative value. Whenever the biggest possible number has been subtracted, the next iteration the second biggest number is subtracted from the resultant of the first subtraction. If the subtracting value would cause the number to become negative, then subtracting number would skipped. This process is repeated until the resultant number becomes zero. While this is being done, the numbers that are possible to subtract area noted down, said numbers are the "ones" in the binary value, and the numbers that resulted in a negative value would be the "zeros" of the final binary number.*

Converting 179 to binary								
Decimal values	179 - 128 = 51	51 - 64 = -13	51 - 32 = 19	19 - 16 = 3	3 - 8 = -5	3 - 4 = -1	3 - 2 = 1	1- 1 = 0
Answer	1	1	1	1	0	0	1	1

Binary to denary: *Converting to binary to decimal simply involves changing the "ones" in the binary number to their base 2 equivalent and then adding all the ones together. In that whenever a "one" is seen in the binary value, said one would be multiplied by a base of two with an exponent depending on the one's current position in the total number. This would be done for all the ones in the value, and the numbers would then simply be added together. This could also be done for the zeros in the value, however, multiplying by a zero is rather useless and would only be useful for showcasing purposes.*

Converting 10100111 to decimal								
Binary	1	0	1	0	0	1	1	1
Multiplying	1×2^7	0×2^6	1×2^5	0×2^4	0×2^3	1×2^2	1×2^1	1×2^0
Decimal value	128	0	32	0	0	4	2	1
Answer	1 + 2 + 4 + 32 + 128 = **167**							

Hexadecimal:

Utilized base: *Hexadecimal is a numerical system that utilized base 16 and relatively nonconventional methods of displaying values. That is the case since the first ten values are displays the same way that would in the denary number system (0 to 9), however, in order to accommodate for the remaining six, the English alphabet is utilized, and so ten would be A, eleven would be B, twelve would be C, and so on until F is reached, which represents sixteen. From there, the way that these numbers would be presented is the same way as in the denary system. Where the first value would be multiplied by sixteen with an exponent of zero, while the exponent iterates by one each time the number's position increases.*

Application: *The hexadecimal numerical system is utilized whenever a relatively large number is needed to be displayed as the system is much more compact and allow for the storage of larger numbers, not to mention that it is relatively much easier to read and understand compared to binary, which comes in extremely handy for large numbers. As such, it is seen in application such as displaying memory addresses, MAC addresses for devices (The unique address given to each computer system), and for displaying the Red-Green-Blue color ranges.*

Hexadecimal conversions to binary: *The simplest way in order to convert hexadecimal to binary it so simply convert it the value initially to denary by multiplying the base and exponent and then converting the denary value to binary. For example, 9EF would be converted through doing the following: $(F/15 \times 16^0) + (E/14 \times 16^1) + (9 \times 16^2) = 2543$. From there, 2543 would be converted into binary using the earlier shown method. What could also be done is to convert the individual values of the letters and numbers and then simply "attach them" to one another.*

Converting A9 to binary								
Hexadecimal	A (10)				9			
Binary value	1	0	1	0	1	0	0	1

Binary to hexadecimal: *Converting binary to decimal involves separating the binary value into section of four bits. From there, each denary equivalent would be calculated for each section. Said equivalent would then be simply be the hexadecimal value, in that if it was zero to nine, it would be written as is, and if it was from ten to fifteen, then it would be changed to its alphabetical equivalent in hexadecimal.*

Converting 0010 1111 0111 1011 to hex																
Hexadecimal	0	0	1	0	1	1	1	1	0	1	1	1	1	0	1	1
Multiplier	2^1				$2^0 + 2^1 + 2^2 + 2^3$				$2^0 + 2^1 + 2^2$				$2^0 + 2^1 + 2^3$			
Decimal	2				15				7				11			
Hexadecimal	2				F				7				B			

Octal:

Utilized base: *The name octal stems from the prefix "octa" in Greek, which directly translates into "eight". As such, the octal numerical system utilized eight as its base. This means that the numbers utilized in the number system range from zero seven (i.e., there is no number "8" or "9" in this system), and the first digit position is multiplied by eight with an exponent of zero, with each position subsequent to that gets its value iterated by one. For example, the value 765 would be:* $(5 \times 8^0) + (6 \times 8^1) + (7 \times 8^2) = 501$.

Application: *Just like hexadecimal, the use of octal comes from the need to simplify binary values into more humanely comprehensible digits. However, since the introduction of hexadecimal, octal became rather obsolete, and there are two different reasons for that. The first being the fact that hexadecimal is simply able to store a higher value in a shorter number of digits, therefore simplifying larger numbers even more so than that of octal or even denary. The second reason is that two digits of hexadecimal can exactly showcase 8 bits or a single byte as 4 bits add up to a total of 15, which is the entire range for hexadecimal (0 to 15) and placing two next to one another provides 8 bits or a byte, which – as mentioned earlier – is a significantly practical number of bits especially when it comes to the identification number of different characters. On the other hand, Octal has no real way of representing a byte, since in a single octal digit is represented by three bits since three bits provide the full range for an octal numerical digit (0 to 7).*

Octal to binary: *The method used to convert from octal to binary is exactly the same as that of hexadecimal, where the number could either be converted into denary initially and then into binary, which would be done by multiplying the values by their base and exponent based upon their position and then adding them together. An example of which being converting 536 to denary:* $(6 \times 8^0) + (3 \times 8^1) + (5 \times 8^2) = 350$. *The other method would be to convert each octal digit into three binary values and then "attach" them to one another, the same way it was done with hexadecimal, but instead of 4 bits, its now only 3, with the reason being the one stated in the applications section.*

Converting 156 (oct) to binary									
Octal	1			5			6		
Binary value	0	0	1	1	0	1	1	1	0

Binary to octal: *The same method that was utilized for converting binary into hexadecimal can be exactly replicated here for octal. Except this time, instead of separating the binary value into sections of 4 bits, it would be 3 bits, with said bits then simply being converted into their denary equivalent, which would also be their octal equivalent, and then simply "attaching" the numbers next to one another. To simplify the conversion, the same number that was used for hexadecimal would be used here, 1011110111011.*

Converting 010 111 101 111 011 to octal																
Hexadecimal	0	1	0	1	1	1	1	0	1	1	1	1	0	1	1	
Multiplier	2^1			$2^0+2^1+2^2$			2^0+2^2			$2^0+2^1+2^2$			2^0+2^1			
Octal	2			7			5			7			3			

There was no need to convert from denary to octal since the method utilized already does not allow for numbers to reach above base 8 as three bits have values ranging from 0-7, which is exactly the value range of base 8.

Characters and character sets:

What are characters:

A character from a computer perspective is simply a numerical value representing the identification number of the various unique characters found on the keyboard. That is the case since each character is represented in a binary number that – as stated by (Jenkov, 2022) – is known as the code point. Said code point allows the computer to identify which character is it meant to display. From a lower level's perspective, the code point allows the computer to identify the organized order of the LEDs that the computer is required to adjust the RGB values of as to make the characters more apparent on the screen.

Figure 7 - Characters on a typical English keyboard

What are character sets and encoding:

Character sets: *Character sets are the libraries that contain the various code point for each given character. Depending on the character set, this includes not only the characters found on the keyboard, but also characters from other languages and even emojis. As such, these libraries can end up with possible code points ranging from zero all the way to the millions.*

Encoding: *Encoding refers to the method utilized by the character sets in order to store the code points for the various unique characters, which in turn also the character sets to convert from the binary code point to the characters and vice versa.*

Examples of character sets and their encoding methods:

ASCII: *ASCII, the abbreviation to "American Standard Code for Information Interchange" is the first major character set that was utilized by computers. The character set originally used an encoding system that only utilized 7 bits or 128 different characters (including the binary value of zero). However, according to* (BBC, 2022)*, as computers became more popular, ASCII encoding was then changed into 8 bits, or 256 characters in total. With that being said, 256 is still rather little, and does not accommodate for the various different languages and icons/emojis that are utilized daily. As such, as of right now, ASCII – although continues to exist – is rather obsolete and has been largely replaced by Unicode.*

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Figure 8 - The ASCII characters

Unicode:

What is Unicode: *Unicode, which stands for universal code, is – as the name might suggest – the universal character set, in that it includes every character and symbol from any language or platform. As mentioned by* (Fyfe, 2019)*, Unicode was initially developed in 1991 to accommodate for the lack of a universal character set that could be utilized by computers around the world. The importance of Unicode stems from the fact that originally before Unicode was developed if a computer utilized a different character set were to send that character through the internet to another computer, the other computer might have issues recognizing the character as it is unavailable in its character set, which in turn could risk data potentially being corrupted. The way that Unicode encodes its character set is through the use of "Unicode Transformation Format", abbreviated as UTF. Said encoding method has three major versions, with them being UTF-32, UTF-16, and UTF-8.*

UTF-32: *UTF-32 encodes every single code in 32 bits or 4 bytes, and although this provides a significantly larger number of bits and possible code points that can range up to 4,294,967,296, the overall process is rather inefficient, as each character – common or uncommon – would require 32 bits, therefore making the storage of characters much harder and much more inefficient in that 1 byte encoding is not possible even for characters that are often used such as the English alphabet, therefore meaning each English letter being typed on the keyboard would have to be 4 bytes long in its binary format.*

UTF-16: *UTF-16 can work with 16 bit as well as 32 bit data, which is more efficient but could nonetheless be better since – as mentioned earlier – this still does not allow for the English alphabet and commonly used symbols to be stored in 1 byte.*

UTF-8: *Finally, and most importantly, UTF-8. UTF-8 can encode from 1 to bytes, meaning that commonly used characters are stored as one byte, while more obscure ones are stored in a higher byte range. This is extremely important for two different reasons, the first being storage efficiency for the aforementioned reasons, however, in addition to that, UTF-8 uses the same code points that ASCII uses for the first byte or 128 characters, therefore allowing it to be backwards compatible with any website or software that utilizes ASCII. As such, UTF-8 is almost solely used, with both UTF-16 and UTF-32 being basically unused except in very specific applications that do not worry about storage and memory usage.*

Bias towards the English language: *As can be noticed from both Unicode and ASCII, both heavily favor the English alphabet, with ASCII basically only containing English, and Unicode giving the English alphabet the smallest code points therefore decreasing the amount of storage required for them. The reason for which comes from two different areas, the first being that English is simply the most dominant language utilized on the internet. Secondly, historically the Americans and British were the ones that pioneered computers and computer science, and as such – understandably – they favored their own language.*

The Central Processing Unit

The CPU, an abbreviation for Central Processing Unit, is commonly referred to as the "brain" of the computer system as any functions that the computer needs to do such as the transfer of data to and from memory, calculations, commands, encoding and decoding, interpreting various input signals, and more have initially go through the CPU as it is the part of the computer system that actually decides how those signals need to actually be used and where they should be transmitted to, which is done through the use of various logic gates.

The aforementioned working process and logic gates of the CPU have been briefly touched upon throughout the Binary system part of the report. However, the following section will thoroughly discuss the entirety CPU from its component and the architecture in which they are placed and structured, its working process and cycle, and the methods and techniques utilizing in enhancing the overall speed and performance of the processor.

Memory hierarchy – prerequisite to main components:

What is memory hierarchy

Within a computer, a memory hierarchy that is based upon the speed and the size of the memory chip. At the bottom of the hierarchy is the slowest form of memory that can be accessed by the CPU but are generally able to store the most amount of information. While the ones at the top are generally the fastest but store the least amount of storage. The things that generally make a memory storage system or method fast relates to the distance between the CPU and the storage area, the size of the storage with smaller storage systems being generally faster, and finally, the method utilized in accessing said storage systems. In computers and especially computer architecture, the reason that someone would pay more for storage is to access the information at higher rates rather than being able to store more memory, as such, although memory systems higher on the hierarchy are generally significantly smaller, they do cost much more than the ones at the bottom of the hierarchy. This storage hierarchy generally goes as such: HDD → SSD → DRAM → L3 Cache → L2 Cache → L1 Cache → Registers.

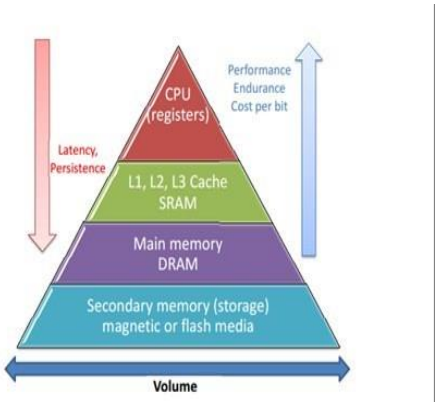


Figure 9 - Memory hierarchy pyramid

What affects access speed:

Distance from CPU to memory: *Although distance differences of a couple of centimeters might not seem significant, especially when talking about the extremely fast movement of electricity, however, in order to ensure a seamless and relatively smooth user experience, the CPU is required to operate at times measured in nano and even picoseconds, as such, the seemingly insignificant distances do actually cause a major impact on the aforementioned seamlessness.*

Size of memory: *When it comes to the size of the memory, generally, the larger the memory system means that more distance would have to be travelled searching and accessing the required data, which – as just mentioned – is an extremely important factor as seemingly insignificant as it might be. Another factor related to the size that should be mentioned is that in larger storage methods, more transistors and wires might be utilized, which in turns also increases the resistance and consequently decreasing the current and therefore the speed of access.*

Access method: *Finally, and most obviously, the method utilizing in actually retrieving said data is going to cause a major effect on the speed of access. The biggest example of this can be seen in HDDs when compared to SSD, where HDDs are limited by the mechanical factor that limits the speed of movement, therefore making it significantly slower than their flash memory counterpart the SSD.*

RAM access method – SRAM vs DRAM:

DRAM working process: *For a bit of background, the typical main memory is more technically known as the Dynamic RAM or DRAM. As mentioned by* (Basics Explained, H3Vtux, 2018)*, the way that DRAM is able to actually store binary values is by utilizing a single transistor-capacitor combination for each bit that is stored. The capacitor holds is the component that stores the actual value of the bit, while the transistor is the component that is utilized for transmitting the value and editing it. Said transistor-capacitor combination is extremely small, and allows for millions and even billions of them to be placed next to each other, and therefore increasing the overall possible space that can be held on the DRAM. However, a major issue with this method of holding or accessing data is that the capacitor is constantly discharging and can only contain the bit's value for a short duration amounting to milliseconds, as such, the transistor has to always recharge the capacitor. This causes a major issue when it comes to the accessing speed, in that whenever data has to be read, the CPU would have to wait until the capacitors have been recharged as to ensure that the bit values being read are accurate.*

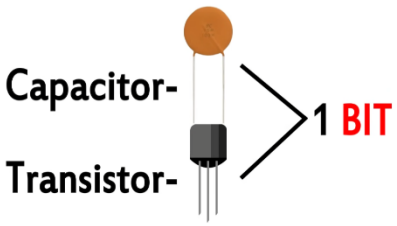


Figure 10 - Abstract DRAM assembly

SRAM working process: *There are two major reasons that the cache is significantly fasters than the main memory/DRAM. The first being simply that is closer to the CPU, while the second is that the cache is a special type of RAM known as SRAM or Static RAM. The difference between SRAM and DRAM stems from the fact that SRAM does not utilize the aforementioned transistor-capacitor combination, but rather uses a number of logic gates that – mentioned by –* (CrashCourse - PBS, 2017)*– create a flip-flop component and operate on Boolean logic in order to hold a single bit without having to be recharged which is done without having to be recharged, which – as mentioned in the DRAM section – is the main reason slowing down the main memory.*

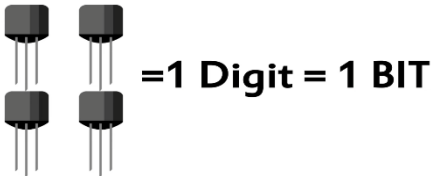


Figure 11 - SRAM abstract assembly

Now, a flip-flop is a component that is able to hold the high or low value constantly, hence the "static" RAM. The way that said flip-flops operate is that they consists of a logic gate circuit that has three main pins as shown in figure 11. Before getting into the logic gates, the way that this works is that whenever the Write Enable pin is high, nothing the output of the logic gate would copy that of the Data Input pin. Meaning that if the Data Input pin was either high or low, that would change nothing with the output pin if the Write Enable pin was low. However, if the Write Enable pin was high, that would "enable" the output from seeing the Data Input pin and copy it, in that if it was high, the output would be high, and if it was low, the output would be low.

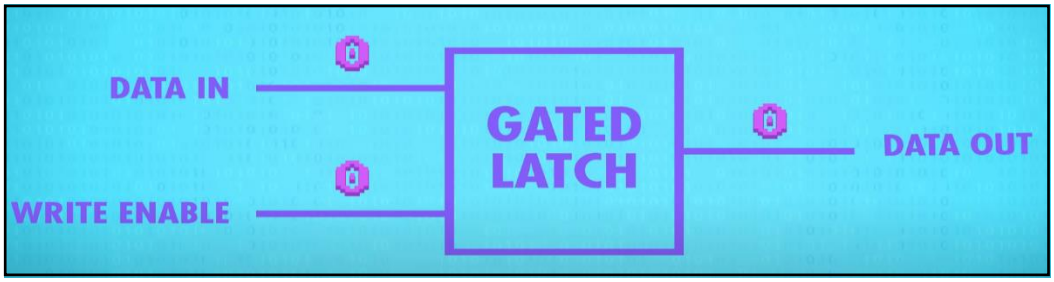


Figure 12 – Abstract flip flop diagram

For this section, it is recommended to use figure 13 as reference. In terms of the Boolean circuit that this process utilizes, what it does is if the Write Enable pin is high, that will send a single signal to both the uppermost and lowermost AND gates. With that, if the Data Input pin was then high, the uppermost AND gate would be on since the signal goes to its second leg/input. While if the Data Input was low, then the lowermost AND gate would be high since the NOT gate reverses the signal. Therefore, if the Data Input was low, the lowermost AND gate would be high, then the output would be low because of the NOT gate that comes after it, which prevents the final AND gate from sending out a high signal. On the other hand, if the Data Input is high, both the uppermost and lowermost AND gates would be high since the lower one's signal would be reversed by the final NOT gate. As such, the inputs of the final AND gate would be high, and the output would be high. Finally, the value is saved by having a pin that goes from the currently high output put to the OR gate, therefore keeping it always on since the OR gate would only need that signal to transmit a high output and as long as the lowermost AND gate does not become high, the output would also remain in a high state.

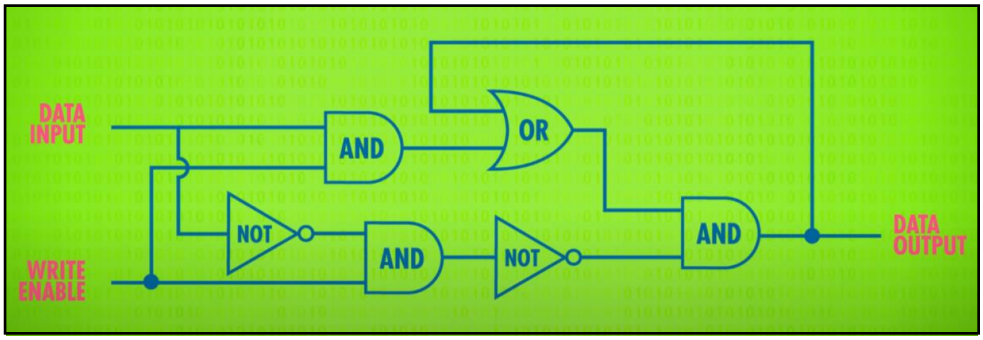


Figure 13 - Detailed diagram of flip flop

Conclusion: *Since the SRAM utilizes more components as each bit would need around four to six transistors, it can hold a significantly smaller number of information than the main memory as more components would have to go in, this is of course given that they both have similar physical sizes. This additionally means that the SRAM is more expensive as it requires more components and it utilized much more complex circuitry, meaning more would go into it its production.*

However, since it does not have to be constantly recharged like DRAM does, this makes much faster. Therefore, this the reason as to why the SRAM is utilized only for small yet essential and frequently used data for the CPU, such as the cache and registers, which would be further discussed in later sections within the report. On the other hand, this also means that the SRAM utilizes much more power than the DRAM as it requires charge to be constantly going through it.

SRAM	DRAM
1. SRAM has lower access time, so it is faster compared to DRAM.	1. DRAM has higher access time, so it is slower than SRAM.
2. SRAM is costlier than DRAM.	2. DRAM costs less compared to SRAM.
3. SRAM requires constant power supply, which means this type of memory consumes more power.	3. DRAM offers reduced power consumption, due to the fact that the information is stored in the capacitor.
4. Due to complex internal circuitry, less storage capacity is available compared to the same physical size of DRAM memory chip.	4. Due to the small internal circuitry in the one-bit memory cell of DRAM, the large storage capacity is available.
5. SRAM has low packaging density.	5. DRAM has high packaging density.

Figure 14 – SRAM vs DRAM conclusion table | Reference: (GeeksForGeeks, 2022)

Topic sentence: *Although different variation of the Central Processing Unit do have significant differences in their architecture and the overall layout and structure of component found within the CPU. With that being said, whatever CPU is being utilized, the main component that are utilized within the CPU's core operating logic do not change, and although different variations of said components could be used to as to provide better performance, the actual base concept of the components remains somewhat the same. The stated components are the ones that would be discussed in the upcoming sections of the report, and they are the Control Unit, Arithmetic Logic Unit, Registers, Cache, Clock, and Busses, which range from page () to page ().*

Control Unit (CU):

Definition and purpose:

What is the Control Unit: *The control unit is the main part of the CPU and is generally referred to as the "brain of the CPU" or the "brain of the brain of the computer". That is the causes due to its criticality in the CPU's overall operations as it is what actually directs electrical signals into their designated locations within the CPU's component and in turn into the rest of the component found within the computer. Overall, there are two different types of Control Unit, Hardwired Control Units and Micro-programmable Control units. However, since Hardwired Control units are generally faster and better than Micro-programmable ones according to* (GeeksForGeeks, 2022)*, they are the ones that would be discussed.*

Operations of the control unit: *The general operations of the control unit consists fetching instructions from memory, decoding instructions, sending instruction to a dedicated circuit that then relays said instructions to the hardware component responsible for executing them (e.g. the Arithmetic Logic Unit) and then repeating this process over and over again in a process known as the Fetch-Decode-Execute cycle.*

Working process – brief on the fetch decode execute (instruction) cycle:

Note: *Since the Control Unit is essentially the component that control the Fetch-Decode-Execute cycle, that would be referenced and discussed here, however, it would be done so from the Control Unit's circuitry standpoint. A more in-depth discussion can be seen in a dedicated discussion further in the report. It should also be noted that the reference to the following section is* (GeeksForGeeks, 2022)*. Furthermore, the main reference figure for the entirety of the following section is figure 15.*

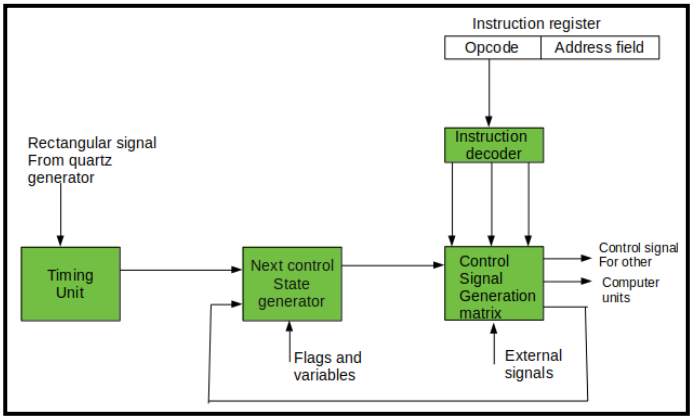


Figure 15 - Hardwired CU diagram

The timing unit: *The timing unit is the part of the control unit that actually supplies it with electrical signals. Said signals come in the form of direct current signals and are used in order to synchronize the working procedure so that all components are working in the same time pattern, consequently keeping operations in a predictable and timed manner.*

These signals are generated using a component external to the Control Unit known as the Clock which would be discussed further in the report. However, what needs to be known for now is that the Control Unit controls the timing and synchronization process of the timing unit and therefore also controls the speed of the instruction cycle.

Instruction fetching: *The fetching process involves taking instruction "keywords" or "identification number" known as the Machine Instruction Format, which is taken from a memory register at the top of the memory hierarchy which goes to show the criticality of the Control Unit.*

This fetching process is done in two parts, fetching the instruction from a part of the main memory and then storing it in a register known as the instruction register. From there, the instruction found within the instruction register is sent to be decoded.

Instruction decoder: *The decoding instruction stage of the Control Unit's working process involves actually understanding what the Machine Instruction Format. This is done so by sending the instruction into an instruction decoder circuit that takes in the various binary values of the instruction and splits the value into three sections, the Mode, Operation Code, and the Operand.*

The machine instruction format itself would be further discussed in the subsequent section. The resultant of the decoded instruction should be a control signal that is sent to a specific wire in the Control Unit which goes a matrix or grid-like circuit known as the Control Signal Generation circuit in order for the instruction to actually be executed.

Control signal generation – Execution: *The Control Signal Generation circuit is the one responsible for transferring the decoded control signal to the appropriate location in the CPU. The way that the Control Signal Generation circuit works is that it receives two signals, one responsible for deciding the column and one responsible for deciding the row. The first of the two signals is the decided control signal, with the second being a state signals that only changes in specific occasions.*

The combination of these two signals allows the circuit to decide on a specific "coordinate" on the matrix, which is the one that would then be sent out via the control bus/wire to the appropriate location. Said location could either be fetching another piece of information or sending a signal to the Arithmetic Logic Unit for adding to binary values with one another. This entire process back from the fetching stage is then repeated until a stop instruction occurs, in which an entirely new instruction set is fetched from the memory and the process is once again repeated.

The Next Control State Signal Generator – Flags: *The Next Control State Signal generate is a logic gate circuit that precedes the control signal matrix and is responsible for choosing one of the two mentioned coordinates (the row or column) in the case of any specific circumstances that require the output of the matrix to be changed. This circuit is inputted by both the timing unit which is simply used to time its output and consequently the output of the matrix, but more importantly it also inputs flags, which can be considered as the "states" of the CPU.*

Whenever a specific state occurs, say the result of a mathematical operation is negative, then the flag might cause a specific output in the Next Signal Control State Signal Generator that consequently changes the output of the matrix in order to accommodate for the flag. Said flags can be inputted from a variety of locations which could include the Arithmetic Logic Unit, I/O devices, or even the control signal generator/matrix itself.

Decoding process – Machine Instruction Format:

Format structure: *The way in which the Machine instruction format is placed is that it is a single bit value ranging anywhere from 8 bits to even 64 bits. Although it is a single bit value, it is split into three different sections, each of which representing critical information that the CPU needs to know in order to decode the instruction properly and transmit it to the appropriate lane in the control signal matrix. The aforementioned sections for the most part consist of the addressing mode, the opcode (short for operation code), and the operand. It should be noted that the main reference for the following section is* (Learn Computer Science Online, 2022).

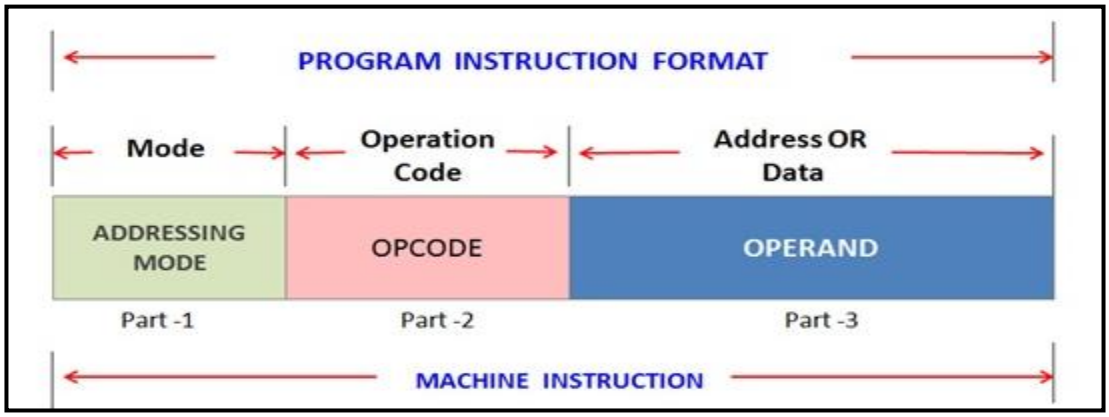


Figure 16 - General machine instruction format

Operand: *The operand is generally the largest section found in the machine instruction code and is stored at the starting bits of the binary value. This sections stores the actual data or address that is going to be worked with by whatever hardware component the control signal matrix is going to send it to.*

Opcode: *The operation code (opcode) section of the main instruction format is the second longest one behind the operand section. This section is used to hold the identification binary value of the type of operation that is meant to be done with the operand. In that if the operand is storing data and an arithmetic operation is meant to be utilized in an arithmetic operation, the opcode would state whether the data should be added or subtracted. Another example of an opcode often used is the store code, with which data is directly stored or sent to a particular part in the memory known as the accumulator.*

Addressing mode: *The addressing mode is generally the smallest section of the machine instruction format and is stored at the end of the binary value. This section of the instruction code used to indicate what type of value the operand will be holding. Depending on the systems, varying amounts of addressing modes can be available, however, as mentioned by* (Learn Computer Science Online, 2022)*, a basic example of an addressing mode would be if it is zero, then the operand would be holding data, while if it is one, then it would be holding an address. The variations of addressing modes generally stem from the mentioned "basic" types.*

For example, as stated by (Byjus, 2022)*, two of the many types of addressing modes that specify operand addresses are direct and indirect addressing, with direct addressing mode having the operand contain an address to the target data, while indirect addressing mode would have it contain an address that contains another address.*

Decoding circuit: *The actual circuitry that is utilized for decoding the machine instruction format varies significantly depending on the CPU, as each it depends on the type of opcodes that would be decoded by the processor, which generally varies from one manufacturer to another. However, in its core, the decoding circuit consists of a large number of logic gates arranged in a relatively complex circuit that take in the binary values from the instruction code and has a dedicated logic gate configuration that for that instruction, which in turn directs the signal to the appropriate location in the control signal matrix and therefore to the appropriate component.*

Arithmetic Logic Unit (ALU):

Definition and purpose:

What is the Arithmetic Logic Unit: *The Arithmetic Logic Unit, also known as the ALU, is the part of the CPU that is responsible for carrying out the arithmetic/mathematical operation for anything that might be required by the CPU. Overall, the ALU is a large number of logic gates connected in a specific manner that allows them to carry out said operations with the use of binary values.*

Said operations mainly include addition, subtraction, multiplication, and division, as said mathematical procedures can then be utilized for almost all other arithmetic applications, although some ALUs do utilizes dedicated logic circuitry for varying applications.

The adder: *The main logic circuit found within the ALU and the one that is going to be discussed is the adder circuit. Said adder is the one utilized for both the addition and subtraction of two given binary values and is the one universally found within any given CPU. The following section will discuss what the actual circuitry that causes the adder to operate the way it does and how through addition it is capable of both adding and subtracting values.*

The full adder – addition and subtraction circuitry:

Half adder: *The adder circuit, also known as a full adder, initially consists of two "half adder" circuits. As shown in figure 17, a half adder utilizes two input pins which shall be called A and, two outputs pins called the sum and the carry, and in terms of components, the circuit consists merely of two logic gates, an XOR gate and an AND gate. Both A and B pins go into the XOR gate, with said pins representing the two bits that are going to be added to one another. Now, the reason that an XOR gate is utilized is that when the A and B values go into it, if neither or* ***both*** *are high, the value would be zero, this is called an overflow, which is when the logic circuit is unable to hold the binary value due to it being too large. On the other hand, if only A were to be high or only B, then the output value would be high. This can be seen from the truth table of the XOR truth table shown below.*

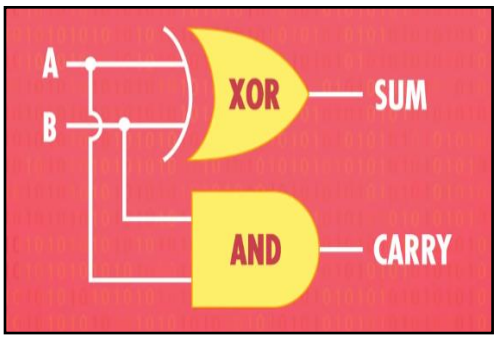


Figure 17 - Half adder assembly

AND truth table		*XOR truth table*	
Input	***Output***	***Input***	***Output***
A = 0 | B = 0	0	A = 0 | B = 0	0
A = 1 | B = 0	0	A = 1 | B = 0	1
A = 0 | B = 1	0	A = 0 | B = 1	1
A = 1 | B = 1	1	A = 1 | B = 1	0

However, there is still a problem, since although adding one to zero does result in one, adding one to one should not result in zero. This is where the AND gate comes in, which is also inputted also inputted by A and B. If both of their values are high, then the AND gate would activate. To conclude, if only A and B were to be high, XOR would output high and the sum would be high, but carry would be zero. While if both A and B are high, then the AND gate would be high and only the carry output pin would be high, which would take the carry or "overflown" bit value to the full circuit, the full adder. This is the same exact procedure that was discussed in the binary addition section, with the carry value being the one that is carried to the subsequent value (1 + 1 = 0 with a carry of one), while the zero being the sum value.

Full adder: *A full adder is simply two half adders placed one after the other with a few additions, hence two "half" adders making a full one. The purpose of the full adder is to address the overflow issue seen in the half adder. As seen in figure 18, the additional components that were mentioned are an input pin called pin C and an OR gate, the C input pin inputs a carry that might have been occurred in a previous full adder, this would be discussed further at the end of the section. The second half adder's input pins this time do not rely on A and B, but rather on the XOR output of the first half adder and input C.*

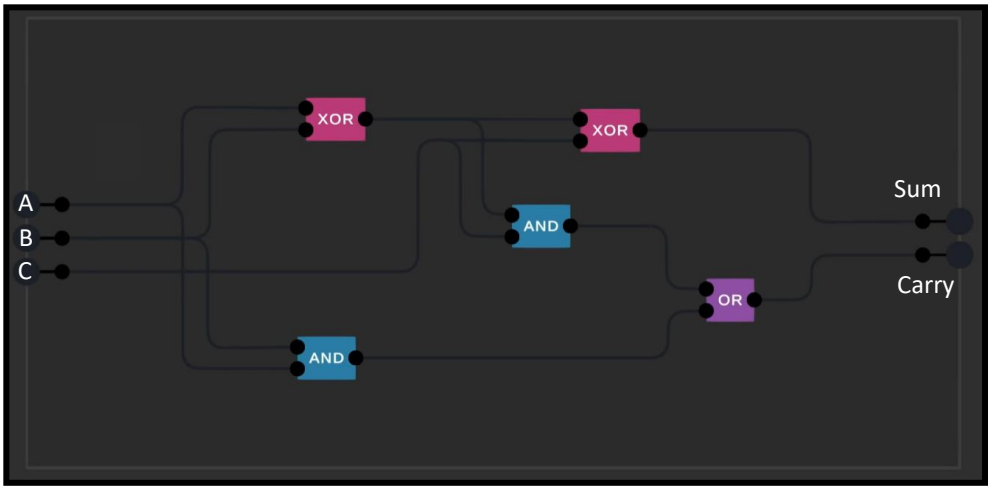


Figure 18 - Full adder assembly

All input possibilities		
Input	***Process***	***Output***
A alone	*If only A is high, then everything would work as it was in the half adder, the first half's XOR would be high and consequently the second half's XOR would be high which in turn results in a high sum.*	Sum
B alone	*Same exact process if as if A alone was high.*	Sum
C alone	*Same exact procedure as if A was high alone, however, this time, the first XOR is entirely ignored as input C does not work with the first XOR. As such, this would immediately go to the second half's XOR, and since it is the only input it has, the sum would be high.*	Sum
AB	*Same procedure if A and B were to be high in the half adder alone. Difference being that this time the output of the AND gate would have to go to an OR gate before going resulting in a high carry.*	Carry
AC	*Since A makes the first half's XOR high and input C is also high, which are both the inputs for the second half's XOR, the second XOR would be low. However, both inputs do go to the second half's AND gate, therefore it will be high and after going to the OR gate it would result in a high carry.*	Carry
BC	*Same exact procedure as if A and C were high.*	Carry
ABC	*If all pins were to be high, the first half's XOR would be low since A and B are high, and because of that, only input C would go to the XOR, therefore outputting high and making the sum high. However, in addition to that, since both A and B were, the AND gate of the first half would be high, which – after going through the OR gate – would also output a high carry.*	Both

Combining full adders: *As might be noticed, the carry or "overflow" issue is still available even in full adders. However, as mentioned earlier, input C actually inputs a carry value that might have occurred in a previous full adder. That is the case since it should be kept in mind that a single full adder is only capable of adding one bit (two bits with one another), and to accommodate for higher bit values, full adders are stacked or concatenated next to one another as to increase the number of possible bits added by two for each adder. This can be seen in in figure 19.*

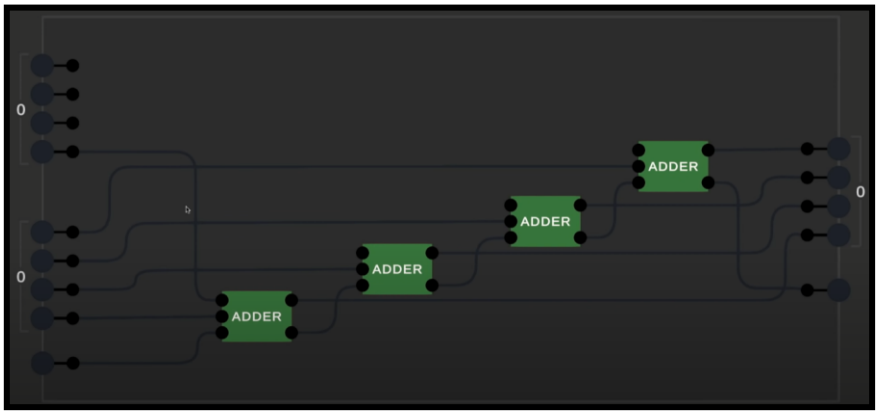


Figure 19 - Combining adders | 4-bit adder

Subtraction: *It has stated initially that the full adder is the circuit used for both adding and subtracting values, and the way that this works from a theoretical standpoint is exactly like two's complement, where the values are inverted and a one is added to the inverted value. From a practical or electrical standpoint, according to* (Lague, 2020)*, the way that this done by adding a subtraction input signal where this tells the circuit whether it is meant to add or subtract, basically an identification code for what the circuit is meant to do. If this signal is low, then the values would remain the same, but if the signal is high, then the values would be inverted.*

Subtract	Input Bit	Output Bit
0	0	0
0	1	1
1	0	1
1	1	0

Figure 20 - Subtraction truth table

As seen in figure 20, the truth table for this process is exactly like the one utilized for an XOR logic gate, and so the subtraction signal would be wired alongside all input signals through a series of XOR gates before they go into the full adder. However, two's complement also involves adding one to the binary value, this can simply be done by wiring the subtraction signal into input C (the carry input) of the very first full adder. A further and more dedicated circuit would be discussed about this in the following section. The circuit itself can be seen in figure 21.

Multiplication and division: *For basic ALUs, multiplication would be done through repeating the addition process, in that 5 x 3 would be five added to itself three times. However, in more complex ALUs utilized in personal computers, then a dedicated circuit would be utilized in order to increase operation speed, same thing with division. However, only the full adder would be discussed in this section.*

Logic unit of the ALU:

What is the logic unit and flags: *In addition to the full adder, another major part of the ALU is the logic unit. This unit is simply there to showcase the status of the calculated value and the various possibilities that it could have. Said flags are sent to other components in the operation such as the Next Control State Signal Generator found in the Control unit. This notifies CPU during "special calculations" which could lead to a different operation path than the one that would be usually utilized. Said statuses or states are generally called "flag". However, in addition to generating flags for other components, the logic unit's job is to also notify the ALU with its own set of flags such as the aforementioned subtraction signal, which would generally be sent to the ALU by the Control Unit.*

The number of flags that a given ALU is able to provide differs from one ALU to another as each flag requires its own logic circuit to operate, meaning more flags can be seen on higher end CPUs and ALUs. However, flags that are almost universal among all ALU are the overflow, negate, and zero flags.

Overflow: *The overflow showcases whenever the last full adder in a concatenated setup has a carry value. The way that this is done is simply by connecting the overflow output signal to the carry output of the last full adder. The importance of this stems from the fact that this showcases that the ALU is unable to provide the proper resultant as the number of bits required by the operation is too high.*

This might occur when trying to add 128 with 128 in an 8-bit full adder, which would not be able to since the largest possible value is 255. This can be seen in figure 21

Negative: *This simply showcases if the final resultant is negative or not, this can be checked by connecting the input of the negative pin to both the subtraction signal and the last bit of the binary value via an AND gate. That is the case since in two's complement, if said bit were to be high then the value would be negative. This is showcased in figure 21*

Zero: *The zero flag checks whether the value is zero or not. The way that this would be done is that a NOT gate would be connected to all output bits, and from there an AND gate would be connected to those NOT gates as shown in figure 21.*

What this would do is that it would check if any of the AND gates is going to turn on, and it does so, then that means that one of the a bit value is bigger one, and therefore the final value cannot possibly be zero. The NOT gates are utilized simply to turn on the AND gates in the case that all bits are indeed zero.

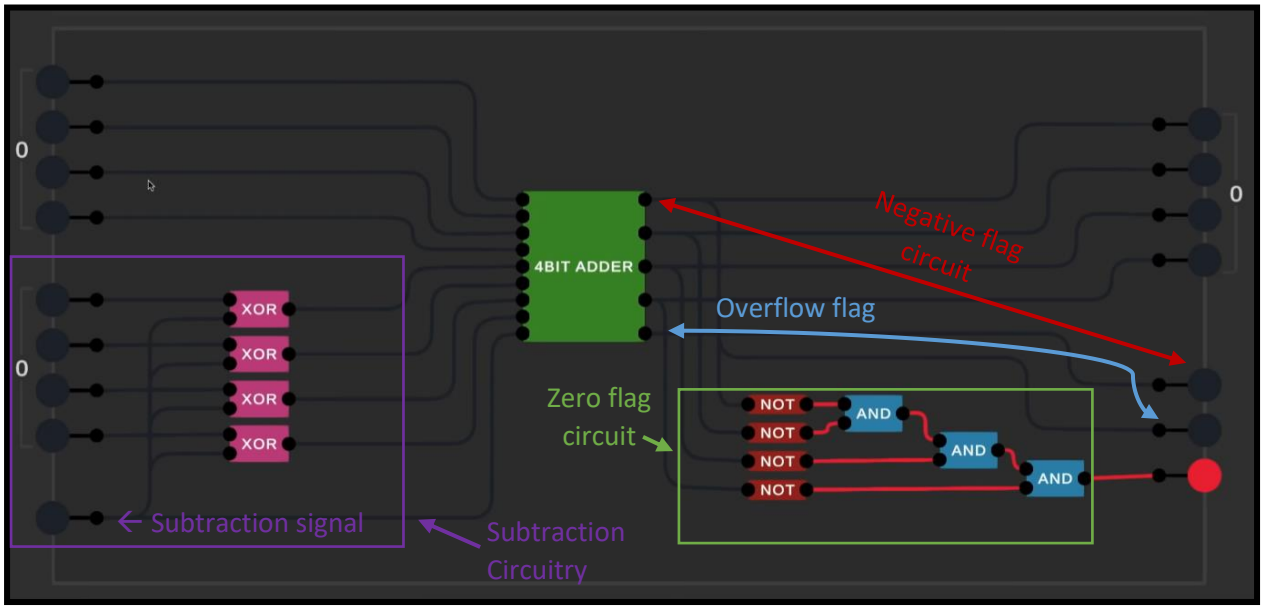


Figure 21 - Finalized adder circuit

Registers:

Definition and purpose:

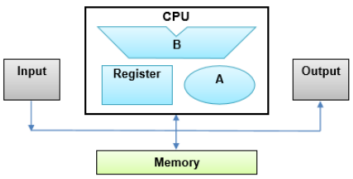


Figure 22 - Register's being the closest to the CPU

What are registers: *Registers are a form of very small yet quickly accessible RAM, or more specifically SRAM. These memory registers are used to store relatively basic yet very frequently called upon binary values that assist in the core operation of the CPU. As such, the registers must be extremely quick to access as they generally hold critical data. For that reason, the size of these registers in most modern computers generally range between either 32 or 64 bits only, which – as mentioned earlier – increases the speed at which the memory can be read from or edited. Moreover, these registers are extremely close to the CPU which furthers assist in shortening their call time. Overall, there are many different types of registers, each one dedicated specifically to hold certain information. This way the CPU has immediate access to the specific information required whenever needed. With that being said, register can be categorized into two different types, special purpose registers and general-purpose ones.*

Special purpose registers: *Special purpose registers are the ones that are utilized for dedicated applications and purposes by the CPU. These may include registers that can be accessed via the assembly programming language but either have limited accessibility or some that cannot be accessed at all.*

General purpose registers: *General purpose registers on the other hand are the registers that can be entirely and freely accessed by the programmer through the use of the assembly programming language. Unlike special purpose registers, general purposes ones can be used to either hold data or instructions and are not limited to a specific value. According to* (GeeksForGeeks, 2022)*, these registers are generally called R"number" and have a given range which varies depending on the CPU utilized (i.e., R0 – R7).*

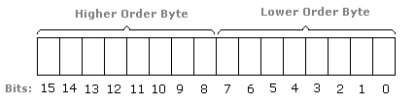


Figure 23 - High and low bytes in registers

High and Low byte sections: *Certain registers have been seen to be split into two different sections, one being utilized for "low bytes" while the other being categorized into "high bytes". In that for example a 16 bit register would a section to hold the first byte/8 bits (0-7) while the other section is being used to hold the remaining byte (8-15). This can be even further subdivided with 32 bit registers, which are split into two 16 bit sections that are each split into two 8 bit sections. Upon research, the reason for this is uncertain, however, a valid possibility is that it is to ensure backwards compatibility with older assembly applications that used to operate on smaller bit sizes.*

Types of special registers:

ACC: *The Accumulator (ACC) is – as stated by* (GeeksForGeeks, 2022)*– generally the most utilized register by the CPU in general as well within Assembly programming. The ACC register stores that data outputted by the arithmetic logic unit especially during multistep calculations were the calculation has more than one step that it needs to do perform. For example, in operations were multiplication and then addition have to be done subsequent to one another, the ACC would first store the multiplication value and then add on top of it the addition value, hence "accumulating" the value.*

MAR: *The Memory Address Register (MAR) holds the address that the CPU intends to read from or write to as mentioned by* (GeeksForGeeks, 2022)*. Said address is the one that would be transferred via the address bus (which would be discussed further on in the report).*

MDR: *The Memory Data Register (MDR) works in accordance with the MAR. The purpose of this register is to hold that data that would actually be sent to the address held on the MAR according to* (GeeksForGeeks, 2022)*. The data held on the MDR would be transferred via the data bus.*

PC: *The Program Counter (PC) register is used for keeping track of the memory address that holds instructions for the CPU whether that is in the RAM or in the cache. The PC can be considered a "pointer" to the memory address that contains instructions necessary for the CPU to execute. As stated by* (GeeksForGeeks, 2022)*, the way that the program counter works is that when the previous instruction has been successfully executed, it would increment by a given amount depending on the bit size of the memory in order to go to the subsequent memory address that would hold the next instruction that the CPU is to execute.*

IR: *The Instruction register (IR) holds the actual instruction that were contained in the PC after the CPU fetches them for them to then be subsequently relayed to the Control Unit.*

CCR: *The Condition Code Register (CCR) is a register checked by the CPU to check if there are any "flags" or special conditions that are occurring during the ongoing operation. According to* (GeeksForGeeks, 2022)*, these flags are determined by the ALU during calculation. Examples of such flags include:*

Name:	***Description:***
C – Carry	*Is set to high/one when a carry or a borrow operations during addition or subtraction (respectively) occurs.*
V – Overflow	*Set to high/one to ensure that the program avoids utilizing two's complement. For example, 60 + 80 in binary would result in 10001100, however, in two's complement, that can be considered as -116 due to the one at the beginning, the flag "V" would prevent that from occurring.*
Z – Zero	*Set to high/one whenever the result of a calculation is zero*
N – Negate	*Set to high/one whenever the result of a calculation is negative*

SP: *The Stack Pointer (SP) is – according to* (Sheldon, 2022)*– a register that holds the address for the last item added to the stack. The stack is a list which holds data in a similar fashion to that of a one-dimensional array. Said stack can be easily manipulated by adding ("pushing") or removing ("popping") elements/bits of data.*

Cache:

Definition and purpose:

The cache: *The cache is a form of Random Access Memory (RAM) known as Static RAM or SRAM that is found within the CPU. The cache's purpose is to be able to provide a dedicated RAM for the CPU that is not utilized by any other components in the system and is significantly closer and faster than the typical main memory in order to accommodate for the speed requirements that the CPU demands for ensuring that critical operations are done at an efficient manner and time in order to ensure a seamless user experience. The reason why the cache is significantly faster has been explained earlier in the storage hierarchy section.*

Type of data stored: *Unlike registers, the cache itself can be described a random pool of memory that can store or access any operations that the CPU might use frequently, unlike the registers that give a dedicated purpose for each register memory. For that reason, the type of data stored on the cache varies significantly and can range from web pages that have been opened that need to be accessed very frequently or instructions that command the CPU with varying tasks such as whether it should add or subtract a given binary value.*

Caching process:

The process: *The caching stage is the process in which the cache is actually loaded. The way that this process works is that when the computer system is initially launched the cache units are empty, a group of data within the main memory (DRAM) would be loaded into the cache as a block that would fill up the entire cache storage. This way, whenever the CPU required information from the main memory, it would initially check the cache for the same piece of information. If found, then the CPU would simply take that information from the cache rather than the main memory, which would be significantly faster as the cache is not only smaller but also much closer to the Control Unit.*

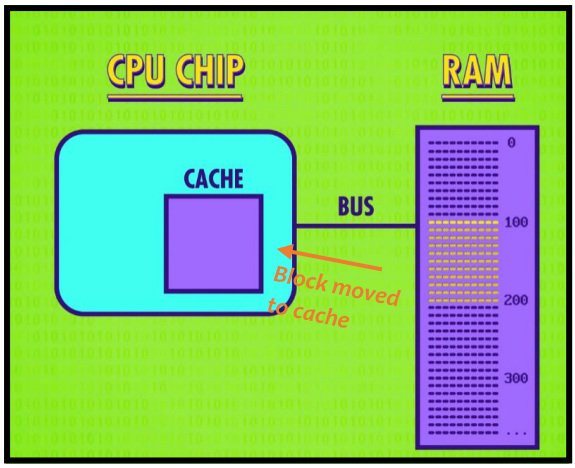


Figure 24 - Caching illustration

Dirty data: *As mentioned by* (PBS Studio - Crashcourse, 2017)*, dirty is the term utilized when referring to cache having data that does not match up with what is available on the main memory. This occurs whenever the main memory is updated with new information, but the cache still retains the old block of RAM that has been initially loaded into it. To address the issue, special flags are sent to the CPU which activate whenever a mismatch is recorded via special circuits utilized for comparing cache values with the main memories. If a dirty data flag is activated, then the CPU would unload the cache's old data and request an entirely new block of information from the main memory.*

Different levels of the cache:

What are cache levels: *Inside the cache, there are three different level to it which have been mentioned earlier when discussing the storage hierarchy. Said levels within the cache range from Level three or L3 to Level one, also called L1. Just like the hierarchy between the SRAM and DRAM, with the cache levels, the level one of cache is smallest yet fastest, while level two and three allow for memory to be stored yet are respectively slower in comparison the level one of the cache. The way that these levels are usually structured is that the level three cache is fed by the DRAM, the level two is fed by the level three, and the level one is fed by the level two.*

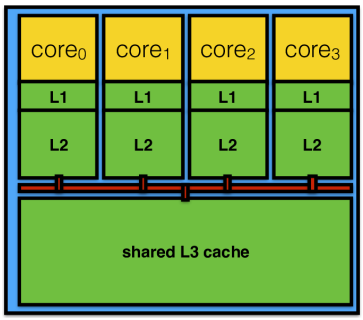


Figure 25 - Levels of cache

Level three: *As stated, level three cache is the slowest yet biggest type of cache memory. The reason for that is – as mentioned in the storage hierarchy section – simply due to it being large, that alone decreases its speed. Additionally, it is the furthest away from the CPU compared to level one and level two cache memory. As such, it is usually utilized for less frequently utilized data that is nonetheless required for faster access compared to the main memory. Additionally, as mentioned by* (Hruska, 2021)*, this level of cache is usually universal among all the CPU cores in a multi-core CPU setup, in that the memory pool on the level three cache can be utilized by all cores which consequently also makes it slower and so it is not used for critical information.*

Level two: *Level two cache memory is simply a level one cache but for data that is less sensitive due to it being slower. However, another major purpose for the level two cache is to feed the level one cache with the necessary data that it receives from the shared level three cache. Basically, it is a way of storing data dedicated to a specific core which can then be easily fed into the level one cache more so than if the shared (level three) cache were to feed the level one.*

Level one: *Finally, the most critical cache memory is the level one due to its size and distance from the CPU making it the fastest, and therefore it utilizing for holding critical information that are not quite as essential as the ones being stored on registers. Additionally, this cache level is split into two sections, one being used for data, while the other being utilized for instructions, with them being named (L1D) and (L I) respectively. As their names might suggest, the data level cache is used for holding binary values for data, while the instructions level one cache is used for storing the binary identification numbers for instructions and commands. This helps due to the fact that data and instructions usually go to different parts of the Control Unit (CU), and so it is more efficient to utilized separated caches for each.*

Cache hits: *The way that the CPU retrieves information from the cache is that it checks the levels sequentially from level one to level three, and if it does not find it in any of them, it then goes to the DRAM. Whenever a CPU is able to retrieve information directly from the level one cache, that is called a level one cache hit, and if it is retrieved from the second one it is a second level cache hit and same for level three. These are used to showcase t he efficiency of the architecture, since the more the CPU is able to retrieve information from the level one cache, that indicates that the essential information that the CPU requires is going to its required location (i.e., the level one cache). If the data is not found in any of the caches and the CPU is forced to go to the DRAM, then that is called a cache miss, indicating inefficacy.*

Cache eviction:

What is cache eviction: *A cache eviction is the process of clearing the cache of insignificant data in order to free up space for other data and information to be loaded into the cache from the main memory as discussed earlier. There are multiple algorithms utilized for deciding which data should be removed and for carrying out this process. Some of more popular methods – which are the ones that would be discussed – includes Least Recently Used, First In First Out, and Random Replacement.*

Least Recently Used (LRU): *This process involves keeping track of the last time a process has been used, and whenever the cache is deemed necessary to be cleared, the last used processes are removed from the cache. According to* (Simply Explained, 2020)*, the action of keeping track of the last time tasks have been used is somewhat intensive on the cache and can slow it down.*

First In First Out (FIFO): *As the name might suggest, this method of clearing the cache simply gets rid of the first item that was stored in the cache whether it is has been utilized recently or not.*

Random Replacement: *The random replacement process chooses any data stored in the cache at random and simply removed it. Although this might counterintuitive, according to* (Simply Explained, 2020)*, when implemented and utilized practically, this method is not impractical and actually provides similar results to LRU.*

Clock:

Definition and purpose:

Figure 26 - Clock pulses

Purpose: *The Clock is a circuit found within the CPU that outputs rapid electrical signals in digital format known as clock pulses, digital being that the signals are either high or low and do not have a state in which the voltage being outputted is between the maximum or minimum (just like binary signals). Said clock signals are used in addition to different logic gates in order to coordinate the timing at which different hardware components are supposed to work at.*

Synchronization process: *The way that the actual coordination process might take place is through the use of AND gates that have been placed before components that are connected to the clock component. With said AND gate being connected to both the control unit that sends the actual control signal and the clock which coordinates the timing at which the component should activate, this is needed to ensure that the component is actually needed (control signal) and is operating at the right time (clock pulse). The reason that synchronization is required within the computer is to fully ensure that process collision does not occur especially when operations within a computer are working billions of times per second, as such, if a process were to occur out of order or before one that is precedes it, especially in important parts of the CPU that run the computer, data could get corrupted and therefore the system would simply shut down and not work.*

Electrical standpoint:

The clock signal: *The way that the clock signal works is that initially the clock signal is at low, when it starts working, the signal increases and reaches its "rising edge" or the edge at which it reaches its maximum voltage, whenever the signal reaches its maximum voltage is when the clock signal is actually interpreted by components. From there, the signal might stay at its positive or high state for a short duration and then it would reach its "falling edge" where the signal goes from maximum to zero volts. During the falling edge nothing is interpreted by the system until the signal reaches its rising edge once again. Basically, the only time that the signal is interpreted is during the rising edge, not during the stable stage where the signal is at constant high or constant low nor is it interpreted at the falling edge.*

External oscillation and the clock multiplier: *An essential point that should be noted is that clock is not an actual component found within the CPU itself, but rather it is simply a frequency amplification circuit. For a bit of background, the actual part that produces that said signal is called an oscillator, which generally consists a quartz crystal that vibrates an extremely rapid rates and – as stated by* (Albiva, 2019) *– during said vibrations due to a phenomenon known as the Piezoelectric effect, electricity is generated. The aforementioned oscillator is placed within a component external to the CPU known as the chipset, which is found on the motherboard. Now, the oscillator is utilized for the all the components found on the motherboard itself and a part of that goes into the CPU, however, the CPU require much higher frequencies compared to the rest of the components on the motherboard due to the operations being done being much more critical. As such, the "CPU's clock" is essential a frequency amplification circuit that multiplies the motherboard's base oscillator frequency to one that might be more suitable for the CPU's operations, with the ratio between the base and CPU frequency being known the "Clock ratio".*

AC to DC: *When it comes to the oscillator crystal, the way that it functions in which electrical signals are gradually increase and decrease make it so it generated Alternating Current. This type of current does not follow the conventional states of binary, in that it can be anywhere between zero volts to the maximum voltage. However, binary values – and what the CPU utilizes – is square signals known as Direct Current, which is where the signal is either entirely off or at the maximum possible voltage, which is how binary operates. The reason for this is that Direct Current is simply much faster than alternating current as it switches form low to high almost immediately, while alternating current gradually does so.*

In order to address this issue, before going into the timing unit of the Control Unit, the current goes through what is known as an AC to DC converter as shown in figure 27, a circuit found within power supplies which turns Alternating Current to Direct current. The way that this circuit works is that the alternating current goes through a series of component that gradually "smoothen" out the signal into a straight/direct signal. This starts with going through a component known as a full bridge rectifier that prevents the signal from going to its second negative cycle by using diodes, from there the signal goes through a capacitor that rapidly captures and releases the electrical signal, The reason that the capacitor operates in such a manner is due to the resister connected to it creating an RC circuit, which causes the capacitor to automatically discharge. This creates what is known as a "ripple signal", which – although better – still is not stable enough as a direct current for it to be used within the CPU. As such, the signal lastly goes through a Zener diode which fully smoothens out the current, consequently outputting Direct Current signal.

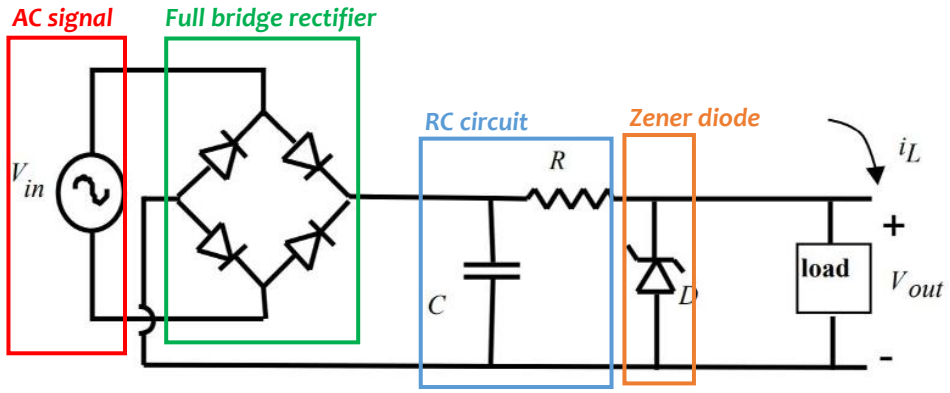
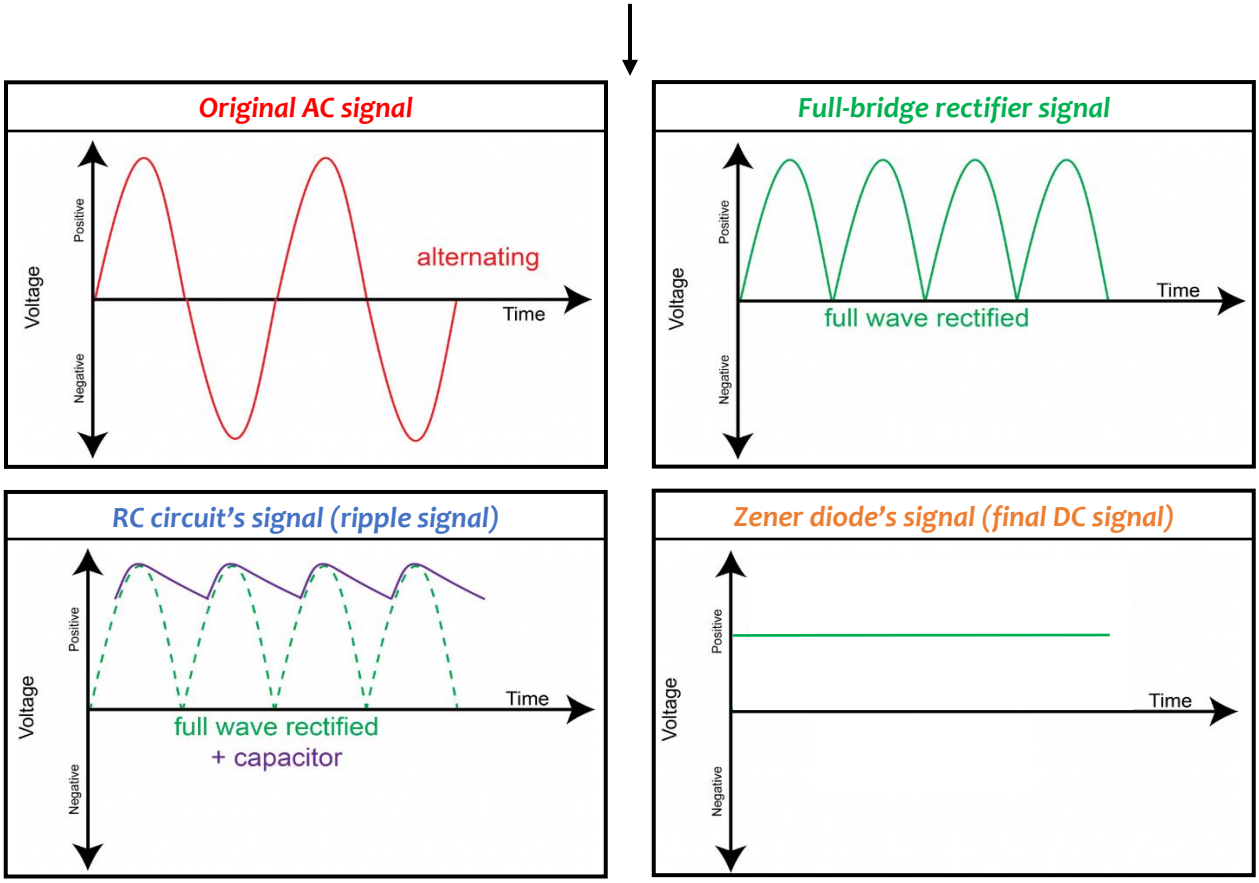


Figure 27 - AC to DC converter



Buses:

The bus system:

System bus: *The term "system bus" refers to the entirety of the bus system found within a given computer. Now, the busses themselves refer to the main form of communication for the CPU with the rest of the component inside the system in general as well inside of the CPU itself. These bus system consists of conductive electrical lines that are placed parallel to one another. Said line can be seen on the motherboard and branch out of the CPU's socket.*

The way that said communication method is established can be seen in two different ways, the first of which being in a unidirectional path, which is where the data is only being transferred from the CPU to a given component. The other method has the copper lines or busses in a bidirectional path, in that data gets transferred both from the CPU to the component and vice versa, meaning data goes both ways and hence bidirectional. Overall, there are three different elements or types of busses that are found within the system bus or computer architecture, each of which having their own dedicated purpose, with said busses being the Address bus, Data bus, and Control bus, with an two additional connection related to peripheral devices such as the Direct Memory Access and Peripheral Component Interconnect busses.

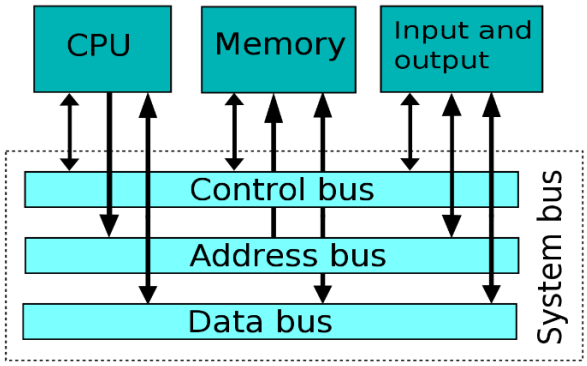


Figure 28 - Main busses in the system bus

Address Bus: *The address bus is a unidirectional bus that is dedicated for communication between the CPU and any form of memory whether it is the cache or main memory. The purpose of the address bus is to carry CPU signals to the memory in order to identify available addresses on the memory. This process is done in order for the CPU to know what address it is going read from or which ones it is able to store new information in. Additionally, the address bus could be used by the CPU to identify an available input/output device which is also done in a unidirectional manner.*

Data Bus: *The data bus is bidirectional in its path, with its purpose being to allow for the CPU to carry data to other components such as data to be stored into the RAM or rendered files to be sent to the output devices. Basically it allows for binary data to be transferred in the form of electrical signals. However, due to the data bus being bidirectional, it also allows for data to be carried from other components like the RAM or input devices into the CPU. It should also be noted that the data carried by the data bus is in the form of binary.*

Control Bus: *The control bus is the second bidirectional connection found on the CPU and computer system. Its main purpose is to by the CPU it to transfer the control signals that were decoded by the Control Unit from the CPU to other components, therefore making it responsible for coordinating components within and out of the CPU with one another. Additionally, for that that reason the control bus is also responsible for carrying signals transmitted by the clock as it is also responsible for coordinating the timing for when components are to operate. This is not to be confused with the data bus, which transfers data, in that the control bus transfers the HIGH and LOW clock pulses*

which tell the other components when they are meant to run, while data refers to the transfer of binary signals that are meant to be further interpreted. The main reason that the control bus exist is to synchronize operations as to avoid data collision, which is when multiple signals are trying to be sent causing one of them to interfere with the operation of the other.

The aforementioned operation was the control bus's job as it is transferring signals from the CPU, however, since the bus is bidirectional it also has another purpose while transferring data into the CPU. Said purpose is to carry status messages from other devices to the CPU, with said status messages containing prioritization signals that could force the CPU to cease the ongoing operation in order to operate the prioritized one. Examples of such signals could be to back up data in the case of an emergency shutdown or something as seemingly simply as a mouse click, which is extremely important in order to provide a seamless and user experience.

Peripheral Component Interconnect Express: *The Peripheral Component Interconnect Express, abbreviated as PCIe is a bidirectional connection port used for peripheral devices that is found on the motherboard and connects to the CPU as to provide the user with the option of upgrading and enhancing their motherboard and CPU connections. This port or bus can be used by various peripheral devices such as network cards, sounds cards, GPUs, and more.*

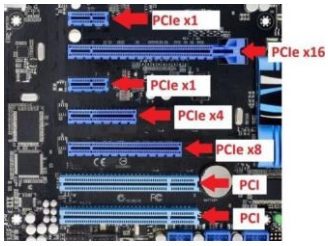


Figure 29 - PCI slots

However, different peripheral devices have different requirements for the amount of data that should go into and out of them, as such, in order to accommodate for that, different PCIe slots are available with larger number of bus lanes found on them. Each bus lane – as mentioned by (Glawion, 2022) *– consists of four copper/bus lines, two of which being for data transmission with the other two being for receiving data, hence being a bidirectional bus. Said PCIe slots can range from PCIe x1 to something like PCIe x16, which offers 16 different lanes and therefore enhance performance. For example, something that is process intensive like the GPU would use a PCIe x16, while other devices that might not be used in as much of a rapid manner such as a network card might use PCIe x1.*

Direct Memory Access: *Direct Access Memory, also called DMA, is a method employed by most modern motherboards that allows certain devices to directly access the memory by going through a dedicated bus rather than having to initially go through the CPU. This process allows for memory sensitive devices to quickly access the RAM, making their operations faster and safer. Examples of such devices are the disk drive, graphics card, network card, sound card, and other peripheral devices. Although this has nothing to do with the CPU itself, it nonetheless is important to mentioned as it showcases some devices – when necessary – can bypass the CPU and go directly into memory if required. Although this is a bus, it is controlled through a special component found on the motherboard called the DMA controller.*

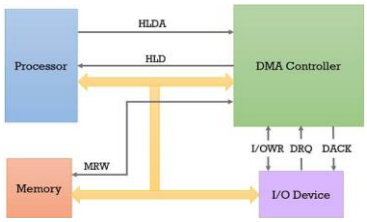


Figure 30 - DMA bus illustration

Bus specifications and properties:

Bandwidth: *Bandwidth refers to the number of bits that can be transferred from one end the bus to the other in a given amount of time, with the said time generally being within a second. In simper terms, this basically refers to the data transfer speed of speed of the bus and is usually measured in Hertz (amount per second).*

Serial and Parallel communication: *There are two types of methods of implementations that busses can utilize, with them being parallel or serial connection. In parallel connection, a bus connection is implemented by having multiple copper lines placed – as the name might suggest – in a parallel manner, with each bus transmitting one part of the binary value that is meant to be send. For example, 11110000 would be transmitted in the following manner, the first four parallel lines would contain the ones, and the last four parallel lines would contain the zeros. A major problem*

with parallel connections is that they are half-duplex, in that they either send or receive data and cannot do both operations simultaneously.

On the other hand, serial communication is where a binary value is entirely transmitted on a single line, where the aforementioned 11110000 would be start the transmission at the rightmost 0 and end the leftmost one. Compared to parallel connections, this is much more economical and cheaper as less wires are utilized, not to mention that electrical strain that can be caused when power is being sent to 8 wire at the same instant. Serial communication is also relatively much simpler to implement, but at the same time is also slower in comparison to parallel as it would take 8 clock cycles to transfer 8 bits of data, while parallel connection would take only a single clock cycle since all 8 bits would be transferred simultaneously. Additionally, component generally work with parallel signals, in that each pin within the component be specified for a given bit. As such, a serial to parallel converter would have to be utilized in order to actually execute serial communication. This can be seen in figure 32.

With that being said, although it was just stated that serial connections are slower, if a serial connection is able to transfer 800 bits/second, and eight parallel lines are able to transfer 100 bits/second for each line, translating to 800 bits/second, the overall speed would be identical. As such, if possible, serial communication is initially pushed to its furthest, and if the difference in speed between parallel and serial is still significant, then parallel would be utilized, however, if the difference in speed is deemed insignificant, serial communication would be preferred due to the stated economic benefits of serial connections.

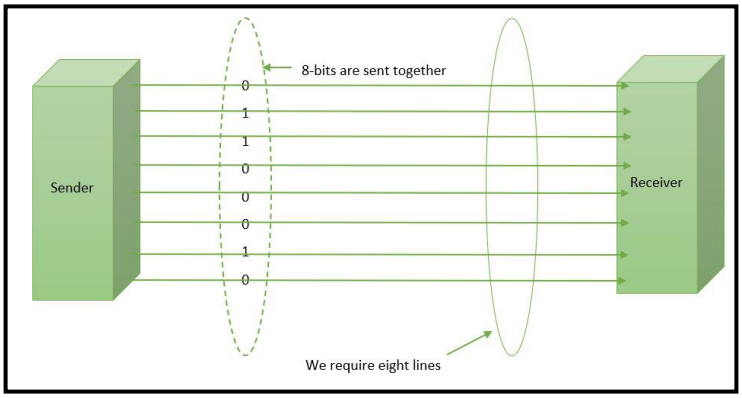


Figure 31 - Parallel communication

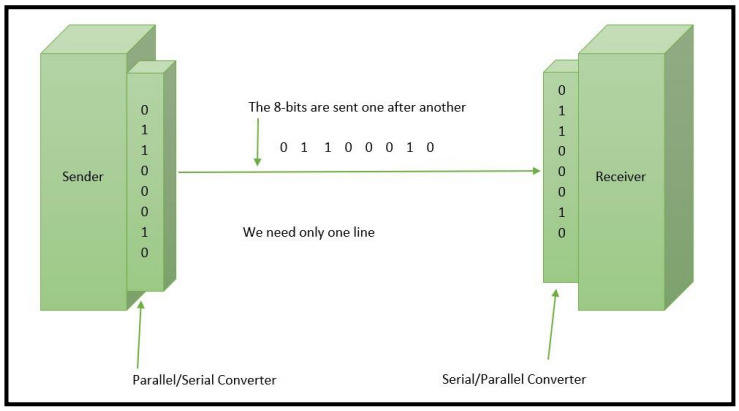


Figure 32 - Serial communication

The instruction cycle – The combined working process of the CPU's components:

What is the instruction cycle:

By definition, the fetch-decode-execute cycle is the series of operations that the CPU does in order to complete an instruction or a command given to it by the operating system. It is the fundamental method that is utilized in almost all CPU in order for CPU to be able to complete its job of organizing components within the computer systems and completing arithmetic calculations and tasks. As the name might suggest, this process is done in the form of a continuous that each time involves initially fetching instructions, decoding them, which – as seen earlier – come in the form of a special machine instruction format, and finally executing the instruction which is by sending the decoded value into a special instruction matrix. An additional step to this would be to store the resultant of the execution, and as such the cycle might sometimes be called as the fetch-decode-execute-store cycle.

The entire purpose of the CPU is to be able to carry out the instruction cycle, basically all components found within it are responsible for this operation. As such, the following section will basically be showcasing the combined effort of the CPU's components. With that being said, the main component responsible for the bigger part of the instruction cycle is the Control Unit as has been shown earlier. However, all other components nonetheless play a necessary role in ensuring that the cycle is accomplished accordingly.

The Instruction Set Architecture (ISA):

Overview:

Assembly language: *As mentioned earlier, the assembly language is the lowest level programming language utilized by computer systems. It is the one that is able to immediately control the binary output and instruction sets built into the CPU, unlike higher level languages which have to initially be compiled assembly and then into binary. With that being said, since assembly can be said to work directly with the CPU, different types of CPU's and the instruction sets available on them does change the way that the assembly code is written.*

What is the Instruction Set Architectures (ISA): *The ISA is the actual code from an assembly level perspective that is utilized in order to connect the hardware with software. As show by* (Chen, Novick, & Shimano, 2000)*, the two main ISAs utilized in assembly include RISC and CISC. These two are – as just mentioned – different ways in which the assembly code can be written. A programmer uses one of them based on the CPU that they are writing the program for.*

Complex Instruction Set Computer (CISC):

Objective of CISC: *As mentioned by reference, the CISC methodology was initially developed in 1970 and its goal was to decrease the number of lines of code that an assembly programmer had to write, essentially compiling multiple low-level steps such as storing and loading from memory into a single line of code that can be executed using a single instruction. This led to CISC assembly programming being very similar to what modern high level programming languages look like, at least when it came to its syntax. The reason for this was to essentially make the programming process much easier.*

Implications on hardware: *Although CISC programming was relatively easier, it did come at the cost of hardware complexity. That is the case since – according to accomplished – reducing instructions into one command meant that the control unit architecture and instruction decoder had to be much more complex in order to have the right logic gate configuration to be able to translate the decoded instruction due since that single instruction would require multiple steps to accomplish and therefore the architecture needed to have a single logic gate setup for the entire process. Additionally, this also meant that single instruction would require more than one clock cycle in order for them to be accomplished.*

Example: *In the CISC methodology, multiplying two variables could be done using a single instruction and can be done in a singe line of code. As mentioned by* (Chen, Novick, & Shimano, 2000)*, this would be done using the command "MUL":*

```
MULT R1, R3
```

The shown snippet of code is utilized in order to multiply two values, with the values being the data found in registers R1 and R3. This can be very much resembled to multiplication in higher level languages where the syntax would be a = b * c.

Reduced Instruction Set Computer (RISC):

Objective of RISC: *On the other hand, the RISC methodology was initially developed in 1980 and aimed to be the successor of the of the CISC methodology. This approach solely utilized "reduced instruction" or instructions that do relatively very small tasks. This translates to having to write very long lines of codes in order to perform relatively simple instructions, at least that is the case when compared to CISC. This consequently means that commands seen in CISC such as the MULT command would have to subdivided into separate operations that involve loading the data into the appropriate registers initially and then multiplying the values found on said registers.*

Implications on hardware: *The advantage of RISC of CISC comes from a hardware point of view. Unlike CISC which has to accommodate for large logic gate configurations and multiple cycles per instructions, the RISC approach aims to ensure that every instruction can be done in a single clock cycle, which in turns means that although the code is longer, the time taken for the instruction to be done is around the same time taken by the CISC approach. Overall, this means that the overall size CPU chips that utilize the RISC approach would be much smaller since the logic gate circuitry would be much more individuated for smaller approaches that can be reused and combined in order to accomplish the same tasks that the larger yet more dedicated circuits found in the CISC methodologies are capable of doing.*

Example: *The same example shown earlier of multiplying two variables would be done here but using the RISC methodology of assembly programming:*

```
LOAD R1, A

LOAD R2, B

MUL A, B

STORE R3, A
```

The shown code does the following: Loads contents of register R1 into address A, loads contents of register R2 into address B, multiply contents of register A with contents of register B, and finally store the contents of that answer into register A. Obviously a much longer process compared to the code snippet shown for development in CISC assembly.

Conclusion:

Comparison:

Reduced Instruction Set Computer (RISC)	***Complex Instruction Set Computer (CISC)***
Software dependent (hardware is simpler)	*Hardware dependent (more complex)*
Longer lines of code	*Shorter lines of code*
One cycle per instruction	*Multiple cycles per instruction*
A lot of RAM is require to store the code	*Very little RAM is require to store the code*

Popularity: *Between both CISC and RISC, CISC is definitely more utilized due to it being seen in microchips and ISA's such as the x86, which have an extremely prevalent market share according to* (Kukunas, 2015)*, while the RISC method is seen on ARM processors. However, RISC devices are generally considered to be an improvement upon CISC. With that being said, each chip does have its own use, for example, chips utilizing the RISC methodology are mostly seen in mobile phones as stated by* (PC mag, 2022)*, with the obvious reason being that RISC requires less components and therefore more hardware can be fit into a tight space such as that of a mobile phone.*

Ties to the instruction cycle: *The way that this all ties into the instructions cycle are that depending on the approach utilized, whether it is RISC or CISC, the way the cycle is executed differs drastically. Especially when it to the number of clock cycles it takes for a command to executed and the way the circuitry utilized for the cycle is built. As such, the architectural paradigms (ISA) are a major point of consideration as the instruction cycle is a major part of everything occurring in the CPU and consequently the computer system as a whole, making it a critical point of discussion.*

Instruction sets:

What are instruction sets:

What are they: *Instructions sets are the library that contains the actual machine instruction format codes. Said sets are manufacturer specific as each manufacturer would design their own unique decoding circuits for their CPUs. As such, instruction sets differ from one manufacturer to another as the instruction binary values utilized are entirely based upon how the decode circuit is going to interact with said value, therefore utilizing different decoding circuits means different instructions sets would be required.*

What are microarchitectures: *A microarchitecture refers to the circuitry that is utilized in order to interpret and actually implement the ISA (RISC or CISC code) seen in the previous section. It is used to understand he varying instructions and operation codes that the CPU is supposed to accomplish. The type of microarchitecture that is utilized indicates the limits of operations that the CPU is capable of understanding since utilizing operations that are not inherently available on the CPU's ISA means that there is not appropriate combination of logic gates to accomplish the required task. A consequence of this is that depending on the ISA the assembly code is written changes. This leads to certain microarchitectures having special instructions sets and some utilizing RISC while others utilizing CISC, with ones even applying a combination of both methodologies.*

Some of the more popular instructions sets found on CPUs currently available on the market include intel's x86, Arm's ARM64, and DEC's Alpha. As mentioned earlier, these instructions sets are basically binary values, so going into them would be unnecessary. However, the following section contains some of the most frequently seen opcodes throughout most ISA's and can generally be seen in both the RISC and CISC methodologies/paradigms.

Frequently seen opcodes in the instruction set:

Name	Description
Add "address"	Adds the number on the memory address to the number stored on the accumulator
Load "address"	Loads the contents of the given address on the accumulator
Jump "address"	Changes the Program Counter (PC) to the specified address
Store "address"	Loads the data in the accumulator to the specified address
XOR "operand 1" "operand 2"	Checks the XOR state for data found in two addresses and stores the result (1 or 0) in the accumulator

Working process:

Process: → Reference to (Baptiste, 2022)

Process	Description	Component
Fetching	Go to the memory address specified on the program counter	PC
	Copy program counter address into the MAR	MAR
	Fetch and place instruction from the address into the MDR	MDR
	Copy instruction from the Memory Data Register to the IR	IR
	Send the instruction from the Instruction Register to the instruction decoder	IR
	Increment PC	PC
Decoding	Check the opcode found for a matching instruction set	Instruction decoder
	Based upon operation, the instruction decoder chooses the appropriate path for the control signal	Instruction decoder
	Send signal to Control Signal Generator Matrix	Instruction decoder
Execute	Check for any flags or interrupts	Next Control State Signal Generator
	Send operand to ALU for calculation	Control Signal Generator Matrix
	Send state/flag of the matrix's output to the Next Control State Signal Generator	Control Signal Generator Matrix
Store	Store output value from ALU in accumulator	ACC
Repeat	Check value of incremented program counter and copy it to the Memory Address Register (process repeats itself)	PC

Purpose of MAR: *The MAR holds the address value any time an address is utilized in the process, whether it is by the PC or if an address is called in the instruction.*

Purpose of MDR: *Holds the contents of whatever is in the fetched address, whether its instructions, integers. It is also used to feed the MAR with the address if one is found on the instruction operand if an instruction was stored in the MDR. This is done while the initial instruction is held in the IR to remember the instruction in case of interrupts.*

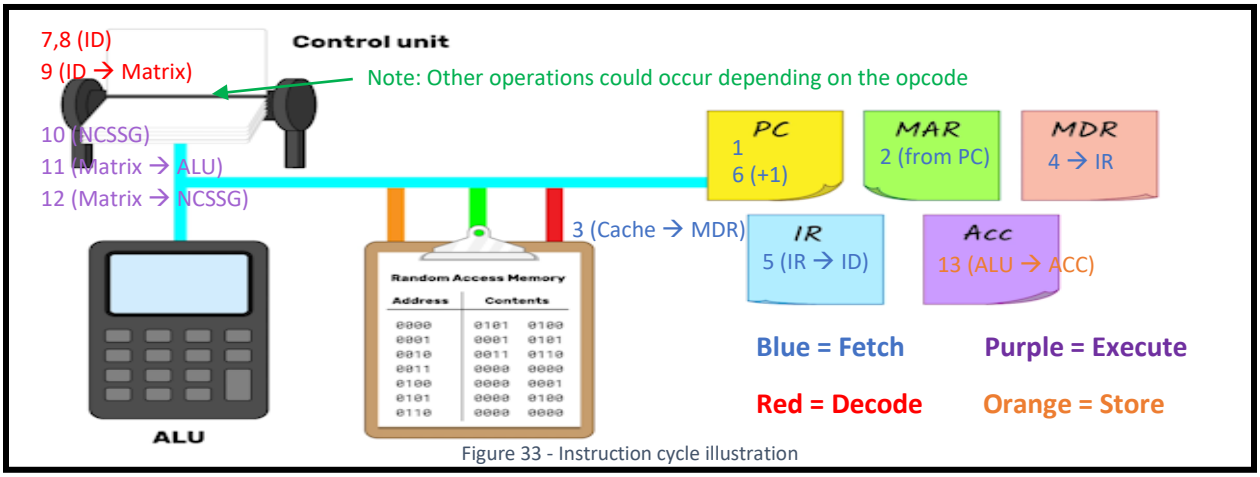


Figure 33 - Instruction cycle illustration

Interrupt handling:

What are interrupts: *As discussed earlier, interrupts are any flags or signals sent to the CU in order to stop (interrupt) ongoing operations to prioritize a more important task, usually one related to user experience.*

What could cause interrupts: *The most prominent causes for interrupts are mouse movements, keyboard strokes, non-volatile memory in the case of an emergency shutdown such as a power shortage for quick backup of RAM. With that being said, even applications – depending on their criticality – can alert flags which cause major CU interrupts.*

Fetch-Decode-Execute example:

The following process is the instruction cycle for fetching a value and placing into the accumulator, adding it with another value, and then storing the resultant in part of the memory.

Referenced memory locations (ONLY initial values)	
Memory address	***Contents***
1	*LOAD 20*
2	*ADD 21*
3	*STORE 22*
20	*50*
21	*28*
22	*0*

First operation		
Process	***Description***	***Component***
Fetching	*PC is given memory address 1*	*PC*
	MAR copies PC (MAR = 1)	*MAR*
	Place instruction found in address one into the MDR (MDR = LOAD 20)	*MDR*
	Copy instruction from the Memory Data Register to the IR (IR = Load 20)	*IR*
	Send the instruction from the Instruction Register to the instruction decoder	*IR*
	Increment PC (PC = 2)	*PC*
Decoding	*Decode operation code (operation = LOAD)*	*Instruction decoder*
	Decode address and load it to the MAR (MAR = 20)	*MAR*
	Send signal to Control Signal Generator Matrix	*Instruction decoder*
Execute	*Check for any flags or interrupts*	*Next Control State Signal Generator*
	Send control signal to fetch data found in address 20	*Control bus*
	Place data found in the address in the MDR (MDR = 50)	*MDR*
	Send state/flag of the matrix's output to the Next Control State Signal Generator	*Control Signal Generator Matrix*
Store	*Load the value of the MDR to the ACC using the data bus (ACC = 50)*	*ACC*

Second operation		
Process	**Description**	**Component**
Fetching	Check PC address (PC = 2)	PC
	MAR copies PC (MAR = 2)	MAR
	Place instruction found in address one into the MDR (MDR = ADD 21)	MDR
	Copy instruction from the Memory Data Register to the IR (IR = ADD 21)	IR
	Send the instruction from the Instruction Register to the instruction decoder	IR
	Increment PC (PC = 3)	PC
Decoding	Decode operation code (operation = ADD)	Instruction decoder
	Decode address and load it to the MAR (MAR = 21)	MAR
	Send signal to Control Signal Generator Matrix	Instruction decoder
Execute	Check for any flags or interrupts	Next Control State Signal Generator
	Send data bus to fetch contents of address 21	Data bus
	Place data found in the address in the MDR (MDR = 28)	MDR
	Send state/flag of the matrix's output to the Next Control State Signal Generator	Control Signal Generator Matrix
	Load the value of the MDR to the ACC using the data bus (ACC = 50 and 28)	ACC
	Send values in the accumulator to the ALU for addition using control signals and the data bus	ALU
Store	Store output of ALU in the ACC (ACC = 50 + 28 = 78)	ACC

Third operation (final)		
Process	**Description**	**Component**
Fetching	Check PC address (PC = 3)	PC
	MAR copies PC (MAR = 3)	MAR
	Place instruction found in address one into the MDR (MDR = STORE 22)	MDR
	Copy instruction from the Memory Data Register to the IR (IR = STORE 22)	IR
	Send the instruction from the Instruction Register to the instruction decoder	IR
	Increment PC (PC = 4)	PC
Decoding	Decode operation code (operation = STORE)	Instruction decoder
	Decode address and load it to the MAR (MAR = 22)	MAR
	Send signal to Control Signal Generator Matrix	Instruction decoder
Execute	Check for any flags or interrupts	Next Control State Signal Generator
	Send control signal and data bus to fetch data found in ACC and load it into the MDR (MDR = 78)	MDR
	Send address bus to check for address 22, send control signal and data bus to store the data found in the MDR into address 22 (address 22 = 78)	Data bus

CPU architectures:

The CPU architecture and what it constitutes:

The Instruction Set Architecture (ISA): *As mentioned in the software perspective section of the instruction cycle, the ISA can be defined as the part or bridge between the software and hardware. In that it is the assembly format code that directly communicates with the circuitry and microarchitecture of the system. Although more are available (e.g., VLIW and ILW), the main types usually utilized for the ISA are RISC and CISC which also were discussed in detail in the aforementioned section.*

The microarchitecture: *The microarchitecture is the hardware section of the CPU's system and is the way in which the circuit has been implemented. This includes the positioning, or the actual working methodologies of the components found within the CPU. This generally constitutes of the CPU's major components such as the CU, ALU, the varying memory storages, clock, and even the buses. However, some microarchitectures do also incorporate specialized components that could be used for specific application requirements. This also means that based on the microarchitecture, different ISAs would be utilized, and although – as stated by* (Michael, 2019) *– different types ISAs can be utilized on different microarchitecture, this comes at the cost of functionality as well as performance.*

The CPU architecture: *The CPU architecture is the working process combination of both the ISA as well as the microarchitecture and is usually expressed as the "Contract between the software and the hardware." That is the case as through the architecture – the ISA and circuit implementation – can the operating system interact with the CPU and consequently with the rest of the components found on the computer system.*

On that account, the software working with the CPU architecture on a low enough level has to be dedicated for said architecture, this includes the ISA as well as other low-level processes. Moreover, the CPU architecture contributes to the overall effectiveness of the system as through is the hardware and specifically the instruction cycle's efficiency is determined.

The two types of CPU architectures that have been utilized throughout history and still retain their dominance in the modern market share are the Von Neuman and Harvard architecture, with both of which usually being utilized for different applications. As such, the following section will specifically discuss the stated architectures.

Von Neuman architecture:

History: *The Von Neuman architecture, also known as the Princeton architecture, was created in 1945 by American-Hungarian mathematician and physicist John Von Neuman.*

Working process: *The main characteristic that distinguishes the Von Neuman architecture is the fact that it utilizes the same memory space for both instructions and data, in that a single main memory would store both of them. This was the majority of what was discussed throughout almost all of the prior section, in that both the data and instructions are called upon by the CPU during the instruction cycle and depending on what the program counter's memory address holds, the fetched information could either be a data value or an instruction.*

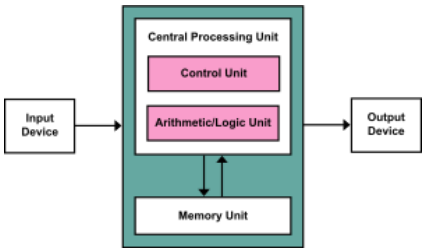


Figure 34 - Von Neuman architecture layout | One memory unit

Advantages:

Simpler circuitry: *A major benefit of utilizing this process as mentioned by* (Wreggit, 2022) *is that it decreases the number of instructions that need to be constructed by the manufacturer in order to operate the CPU from a circuitry standpoint. Since both the data and instructions resides within the same memory unit and either one of them could be fetched at any given chance, this means that the circuitry built should be able to accommodate for either one of them, meaning the buses utilized for fetching, transmitting, and addressing data is the same as the one used for instructions, with the difference later being distinguished within the instruction decoder. This consequently simplifies the ISA and the microarchitecture and therefore frees up more space in the CPU for the addition performance enhancing components. Not to mention that having to use more than one memory space for instructions and data would significantly increase the space taken up.*

Better application flexibility: *Another major advantage of this architecture is the ability for applications to rewrite code segments in order to incorporate specialized properties dedicated specifically for the application's requirements. Since both instructions and data are in the same memory space, this also means that both of them have read-write eligibility, and therefore both of them can be edited. According to* (Mitton, 2015)*, this enhances the range of applications and possibilities found on a given computer system which consequently could improve application efficiency and features potential, since without this freedom of editing the code, then developers would be limited by the boundaries given by the manufacturer.*

The Von Neuman bottleneck: *With that being said, one a major disadvantage to the Von Neuman architecture is the fact that it is unable to call instructions and data during the same cycle, consequently slowing down the CPU. As stated by* (Rambus Press, 2016)*, the reason for that is since both of them are found in a single memory area, they also utilize the same buses, therefore the bus can only be occupied by either the instruction or the data during fetching and transfer of data, consequently decreasing the instruction cycle's throughput.*

Harvard architecture:

History: *The Harvard architecture was one of the first ones major CPU architecture to ever be developed. As the name suggests, it initially developed at Harvard University to be utilized in their Mark One computer. A computer that – according to* (Computer Science, 2021) *– aided in the development process of the first atomic bomb as a part of the Manhattan project during World War Two.*

Working process: *In the Harvard architecture, both instructions of the computer system and data are placed in separate memory units as shown in figure 35. This means that any instructions that the computer system would utilize would all be in one area while any data would be in a totally different memory unit and they would not be treated equally as seen in the Von Neuman architecture. This also means that the process utilizes an entirely different bus systems for both addressing and actually sending the information from either memory unit, with the data memory having its own data bus (transfer) and own addressing bus, with the same with the instructions memory unit, which has its instruction bus (transfer) and addressing bus. Another major point that should be mentioned that due to the distinction between data and instructions, manufacturers generally tend to lock the instruction memory into read-only form. This consequently improves security as no user can change in a way that is outside of what the developer's expect from the system, while on the other hand this limits the user drastically as it does not allow for instruction rewrites, which – as mentioned in the Von Neuman architecture section – are a major advantage if found on a given system.*

Figure 35 - Harvard architecture | Uses two memory units

Advantages:

Parallel information fetching: *The major advantage of such an architecture is that it is able to call on both the instruction and data simultaneously during a single cycle, therefore decreasing the fetching speed basically by half, since in the Von Neuman architecture this would have to be done in two different cycles, one for the instruction and the other for the data.*

Dedicated memory units: *Moreover, as mentioned by* (Tech Computer Science, 2022)*, this separation can lead to the memory units being more dedicated towards the information they store, whether it is instructions or data. This means that if a larger number of instructions are needed for a given application than the amount of data for the utilized application, then a larger bandwidth and memory space could be implemented solely for the instruction memory unit.*

Better security: *They are generally more secure due to the instruction's memory having the possibility of being locked into a read-only mode.*

Disadvantages:

More expensive to develop: *On the other hand, developing Harvard architecture CPUs required much more time and money due to the complexity that these systems could have as they utilize a much larger number of buses compared to the ones seen in the Von Neuman architecture.*

Wasted memory space: *Additionally, any unutilized data memory addresses cannot be made use of by instruction and vice versa for unutilized instructions memory addresses.*

Lower flexibility: *This architecture is also less flexible due instructions memory usually being locked into a read-only mode by the manufacturer.*

Conclusion:

Applications of each architecture:

Von Neuman architecture: *The Von Neuman architecture it utilized in less dedicated and more general-purpose applications that do not require a dedicated and "hardwired" instruction set. The reason for that stems from its ability to rewrite code segments and having applications write their own programs, making it a more "open-source" CPU than the Harvard architecture, especially when it comes to developers. Therefore, the Von Neuman architecture tends to be seen more often in personal computers as they do not have a dedicated function and therefore allow for wider flexibility.*

Harvard architecture: *Harvard architectures, due to them being relatively narrow, are mostly utilized for dedicated CPU applications that require a specific instruction set. In that the manufacturer knows exactly what the application that the processor would be used for. The Harvard architecture shines in such applications as the manufacturer is able to specifically decide on the size of the bandwidth suitable for the application as well as the required sizes for the instruction and data memory units. This leads to the processing speed being significantly faster as not only is the Harvard architecture already more efficient than Von Neuman due to it being able to fetch in parallel, but now the CPU is dedicated exactly for the required application. For that reason, as mentioned by* (GeeksForGeeks, 2021)*, the Harvard architecture is mostly seen in microcontrollers and signal processors, both of which are devices with relatively minimal capabilities.*

Final comparison table:

Von Neuman architecture	***Harvard architecture***
Uses a single memory unit for data and instruction	*Divided instructions and data into two memory units*
Common bus for data and instruction transfer	*Separate buses for data and instructions*
Cheaper to produce	*More expensive to produce*
Relatively simple to construct	*Complex construction*
Requires two cycles to fetch data and instruction	*Requires one cycle to fetch data and instructions*

The modified Harvard architecture:

What is it: *The modified Harvard architecture is the one seen in most modern-day computers as mentioned by* (Wreggit, 2022)*. This computer system architecture is a combination of both the Von Neuman and Harvard systems and uses both methodologies to receive "the best of both worlds." This specific architecture is in fact what was being discussed in the previous sections talking about the computer systems.*

The Von Neuman part: *The modified Harvard architecture utilizes the Von Neuman architectural system on a larger scale, in that the main memory is in fact a single memory unit that is used to store both instructions and data, with both of them being treated the same way, consequently giving freedom for developers creating applications that require them writing their own programs.*

The Harvard part: *On the other hand, on a smaller scale, the cache architecture utilized follows the Harvard architecture system. Although level three and two caches are the same in that they contain both data and memory, the most crucial cache, the level one, is a Harvard system, in that it is split into both an instructions level one cache and a data level one cache, therefore increasing the fetching speed near the Control Unit.*

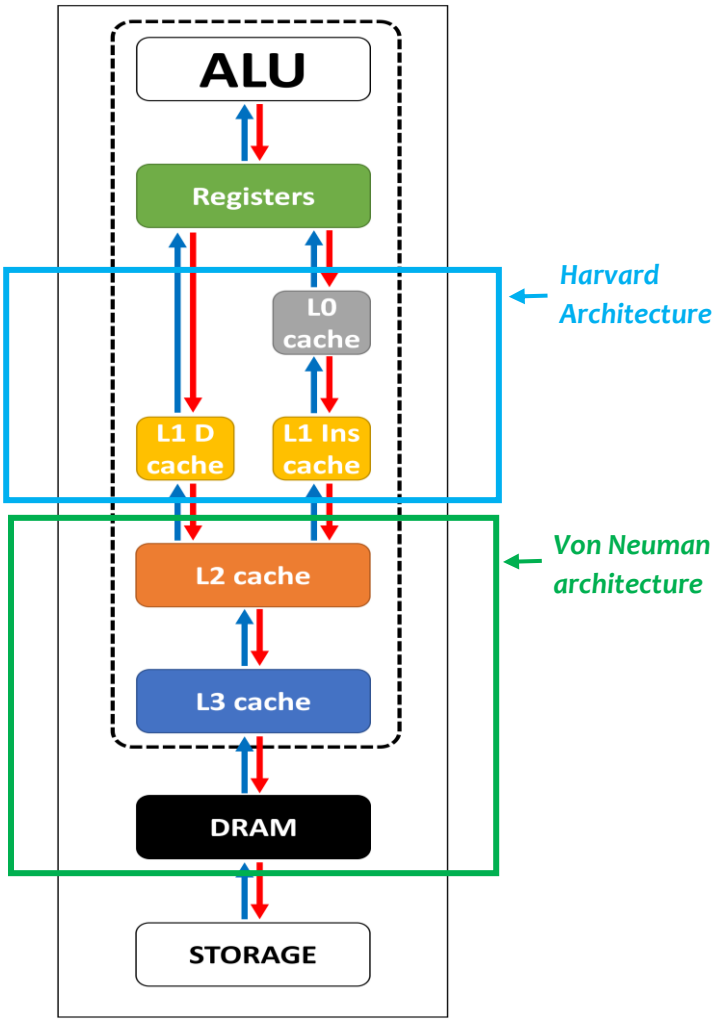


Figure 36 - Modified Harvard architecture

Advanced CPU designs:

Brief:

Overall, there are a number of different design and architectural adjustments that have been implemented into CPUs throughout the years that majorly impacted its performance capabilities. However, the ones that would be specifically focused on are multitasking through the use of multicore CPUs and pipelining as they potentially have had the biggest impact. However, in addition to those, other design methodologies should also be discussed briefly such as dedicated ALU circuitry and caches.

Dedicated ALU circuitry: *This topic has been discussed earlier when going through the ALU, and it basically refers to having dedicated logic gate configuration circuitry for multiplication, division, and any other mathematical operation that might be frequently utilized by the CPU. That is the case since normally, multiplying two values with one another in basic CPU involves simply adding a value the number of times it is being multiplied (I.e., 5 x 3 = 5 + 5 + 5). However, creating dedicated logic gate circuitry specific for doing actual multiplication rather than cycling addition would increase the performance of the CPU as multiplication operations could be done significantly faster.*

The downside to this is that said circuitry would have to obviously be placed somewhere in the CPU, and therefore it would be taking space that could be utilized for other components such as general-purpose registers. As such, this is only done for crucial and frequently utilized operation such as multiplication or division.

Caches: *Another crucial CPU designs is of course the use of caches in their architectures. This evolved from initially utilizing a single level cache, which was simply meant to be a closer version for the RAM to the three-level cache discussed earlier. The simple of fact of having information that could potentially be correct in a location closer to the CPU than the main memory boosted CPU performance drastically and is one of the more significant architectural design implementations found on processors. Moreover, due to the quicker eviction times and larger caches found in most modern CPUs, the likelihood of cache this is relatively high, which further amplified the effectiveness of caching.*

Multitasking and multicore systems:

Multicore CPUs:

What is it: *A multicore processing setup is essentially multiple processing units placed within a single CPU casing. A technology that was initially introduced in 2001 with the purpose of enhancing and even doubling a normal CPU's processing potential as mentioned by* (IBM, 2022). *The way that such a design works is that the multiple cores are able to share the processes and tasks that are being sent to it by the operating system and the user, and therefore unlock a greater level of multitasking capabilities. According to* (Mugerwa, 2022), *each of the aforementioned CPU cores contains its unique Control Unit, Arithmetic Logic Unit, and special registers, with some even having dedicated caches for level one and level two caches. Most modern CPUs nowadays utilize a dual core system; however, more performance sided CPUs do have four, eight, or even eighteen cores within them as seen within intel's i9 10980XE processor.*

Shared resources: *Although a multicore processing setup is generally referred to as having multiple CPU, that is not really the case; the reason for that being that although the varying cores contain the essential what helps the CPU process, they do not contain the main memory, meaning that the cores do nonetheless share the same bus architecture, DRAM storage, and the same being said about the level three cache as discussed previously in the cache section. With that reason for that being that the aforementioned components are centralized within the computer system, and therefore to the CPU, having two of them is essentially like having two different computers, leading do the varying cores being severely unsynchronized.*

The obvious consequence of this is a decrease in the potential performance as the fetching process for the cores would nonetheless be the same. As such, having a dual core system does not necessarily mean double the performance, a widely believed misconception among many consumers.

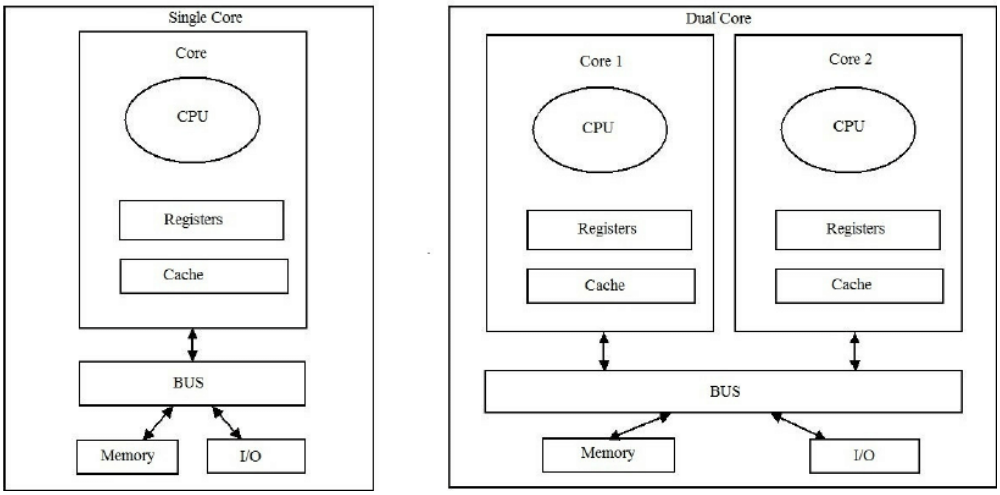


Figure 37 - Shared resources in a multicore CPU setup

Compatibility: *One of the issues when it comes to multicore systems especially for software engineers and developers is the fact that their piece of software needs to be purposefully designed in order to be able to make use of multicore systems, especially when developing systems or utility software. The reason for that is the an application would work using only one core by default as mentioned by* (Bigelow, 2022)*. This is a major problem when the CPU as a whole is weak, meaning without the additional cores that would further amplify the issue. This problem is even greater when the application being ran is CPU intensive and requires a lot of calculations to be made.*

Limitations: *As just discussed, increasing the number of cores generally leads to a drastic improvement in performance, however, the reason that the number of cores within a CPU is not simply increased for every processor is due to the limitations that come with such a setup. One of those main limitations obviously includes the cost of materials and production. Additionally, the size of the processor is another factor, in that having multiple cores takes up a lot of space within the CPU which could be deemed more efficient for other components. Finally, increasing the number of cores leads to greater heat dissipation, this consequently limits the performance capability of the CPU as it cannot be "boosted" as much as a single core CPU would since it would reach temperatures potentially considered dangerous for the processor that could lead to permanent damage.*

Homogenous vs. heterogeneous multicore CPUs: *Overall, there are two types of architectures and systems when it comes to multicore systems, and they are the homogenous and heterogenous setups. These two terms simply denote to whether the multicore system utilizes the same type of core for all of its available cores or different ones. Homogenous systems utilize the same core and are the more popular of the two as this can be seen in the widely utilized x86 CPU architecture as stated by* (Bigelow, 2022)*. On the other hand, heterogeneous setups utilize different cores. The reason for this is to give each core its dedicated function and there is no setup that is strictly better but rather it depends on the requirements from the CPU.*

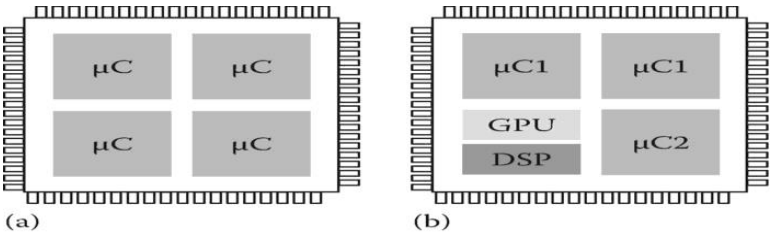


Figure 38 - (a) Homogeneous | (b) Heterogeneous

Multiprocessing:

What is a process: *In computers, a process is a standalone application running on the system that has to be managed and operated by the CPU. For example, a web browser and messaging applications such as WhatsApp running on system would be considered two different processes.*

What is multiprocessing: *Multiprocessing is a capability and technique that can be utilized in multicore systems. The way that it works is that is that the operating systems sends the instructions and every required operation for a given process into one core, while the other is sent to the other. This way, both processes are being ran simultaneously rather than in single core processes where the CPU alternates instructions for both of them. Overall, there are two different types of methodologies for carrying out this process, with them being Asymmetrical Multiprocessing and Symmetrical multiprocessing, abbreviated as AMP and SMP respectively.*

Asymmetric multiprocessing (AMP): *According to* (Neso Academy, 2018)*, in AMP, the way that the multiprocessing architecture is setup is that one core would be a "master" core while the remaining would be "slave" cores. The master core is utilized in order to fetch the instructions and processes that are meant to be executed from the memory and sends them to the slave cores, in that it acts as a transmissions core that regulates and monitors that transfer of processes. On the other hand, the slave cores would receive the processes from the master core and would execute them. This way, whenever a slave core fails, the master processor would direct the task to another slave core. This also means that an AMP architecture uses a heterogeneous core setup as the master core differs from the slave cores.*

These architectures are generally easy to design and less complex as there is only one way for the memory to be accessed which is through the master core, this consequently means they are also cheaper. However, this comes at the cost of processing ability, as this does nonetheless mean one less core would be utilized for actual processing.

Symmetric multiprocessing (SMP): *On the other hand, as mentioned by* (Neso Academy, 2018)*, an SMP architecture would have each processor core would work individually, in that – unlike the AMP architecture – there is no central supervisor that provides the processes to the cores, but rather each core work as "peers" and fetch the instruction from the memory individually.*

Opposed to AMP architectures, an SMP setup uses homogeneous cores, it is much more complex and expensive, but it generally outshines AMP in terms of performance – if the number of cores is identical – as all cores would be working on processing instructions. The reason that SMP architectures are more complex is that CPU infrastructure would have to be built in order to synchronize tasks between the cores and ensure that data collision does not occur and that a preceding instruction is not done processing after the subsequent one.

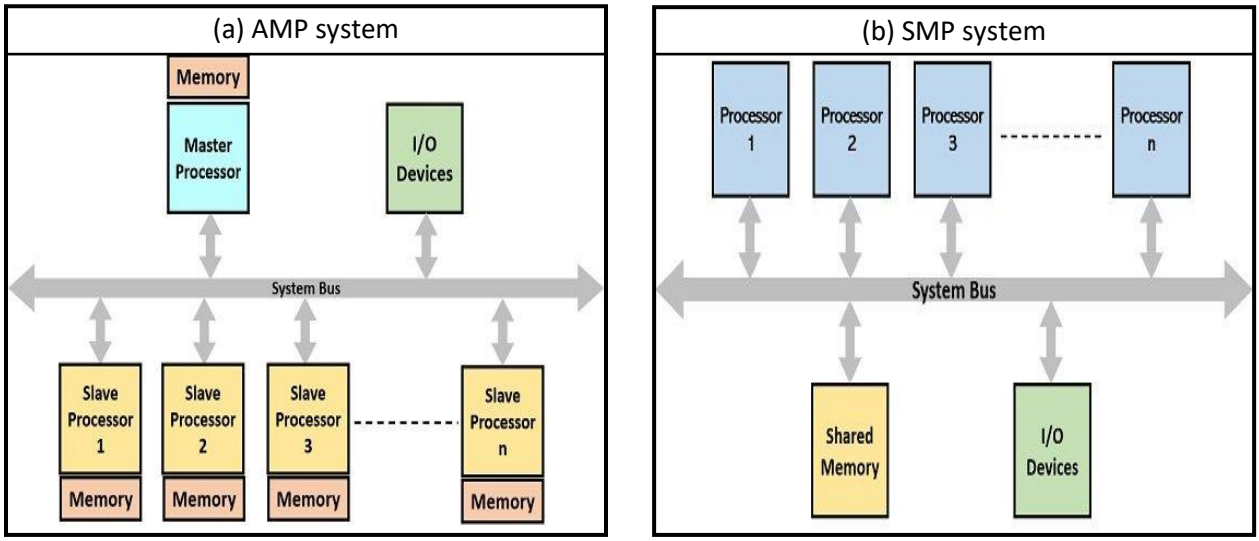


Figure 39 - (a) AMP illustration | (b) SMP illustration

Multithreading:

What is a thread: *Where a process was the entire application, a thread is the subprocesses and instructions found within said application. As an example, if a browser is to be consider a process, a thread would be the tabs opened within the browser, or as* (Khandelwal, 2020) *describes it the "Smallest set of independent commands executed in a program".*

What is multithreading: *Just like in multiprocessing, multithreading is the process of splitting the workload and having it being processed by different cores simultaneously. However, instead of splitting the workload by giving the entire process to various cores, multithreading splits the workload for the subtasks found within the individual processes. However, multithreading can in theory by done in both single and multicore setups although it does so in varying methods. Said methods are known as concurrent threading and parallel threading. Both of which are not determined by the computer's architecture, but rather by the programing methodology of the software application.*

Concurrency in threading: *Concurrent threading is – as mentioned by* (Myrianthous, 2021) *– the process of alternating between the given threads sent to a CPU, in that the CPU would execute a portion of the threads required for a given process and it would do so for a given amount of time, it would then pause that process, work on executing thread portion of another process, and repeat. This way, although the threads are not technically being done simultaneously, what matters is that they are both making progress in an overlapping manner, therefore giving the "illusion" of running simultaneously. This can occur in single core systems; however, it can also happen in multicore setups where concurrency occurs in each of the given cores.*

Parallelism in threading: *Parallel threading on the other hand is when threads are literally being executed in a simultaneous manner. This occurs by sending threads to each core found in a multicore system individually. This basically denotes to executing many tasks at once while concurrent threading is dealing with many tasks at once. Overall, parallel processing is much faster however it is also much more complex to program and actually implement.*

Parallel-concurrent threading: *It also is possible to combine both of the aforementioned processes by using a system in which the threading execution process occurs on more than one CPU core simultaneously while said cores also alternate the processes between one another. In that the core one would work on process one and alternate to process two, while on the other hand core two would work on process two and alternate to process one. Therefore two processes are being done simultaneously and in a sequential and ordered manner.*

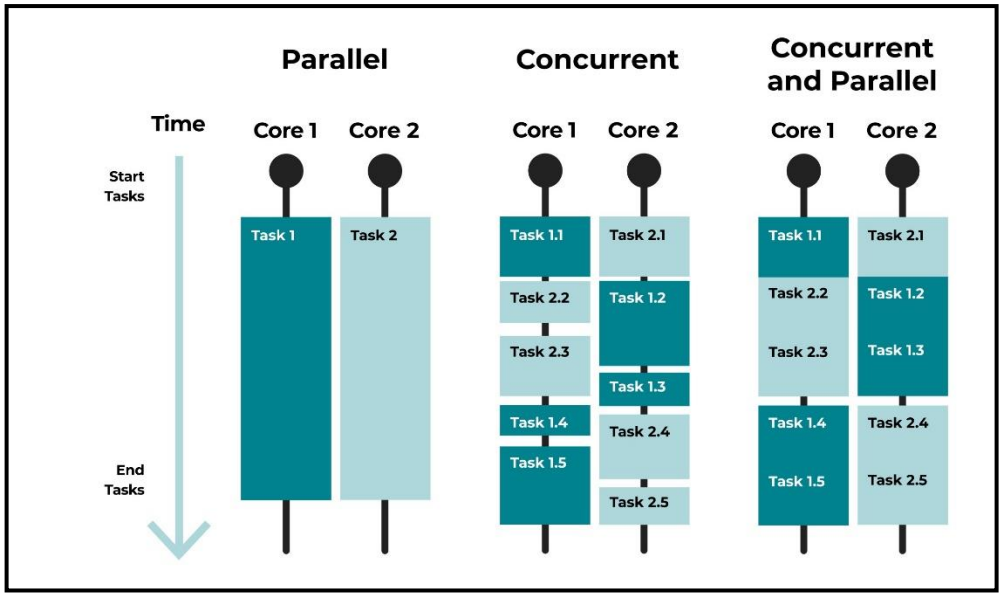


Figure 40 - Parallel, concurrent, and combined threading illustration

Pipelining:

Definition and purpose:

What is pipelining: *Pipelining is a fetch-decode-execute cycle methodology that aims to execute two processes of the instruction cycle (e.g. fetch and decode) simultaneously. Without pipelining, the way that the instruction cycle work is that when the computer system is launched and the very first fetch command is executed, then it decodes, and finally it executes, and only when its done with all three commands of the cycle does system start working on a second instruction. What pipelining does is that when the first fetch command is done and the system moves on to decoding, at the same time the fetch command for the second instruction is initiated, consequently having two instruction running at the same time. A perfect representation of this process can be seen in figure 41.*

Hardware perspective: *The reason that this process even works is due to the fact that the components being utilized during the fetch command are not the same ones as those being used for decoding, and the same as the one being used for executing. What this means is that without pipelining when the fetch command is done, the component responsible for fetching would remain idle until the second instruction initiates. However, pipelining makes use of the components at all times. An important aspect of this is that this process does not require multicore systems as no new components would be needed to execute the commands.*

Speed of execution with pipelining: *The speed that pipelining provides to a computer can be seen as almost tripling its normal processing speed. The reason for that is on the first instruction, three clock cycles would be required in order to fully execute it. However, for every instruction after the initial one only one clock cycle would be required. That is the case since the as the first instruction is being done, the second one is "catching up" to it. As when instruction one is at execute, instruction two is at decode, and it would only need one more cycle to be done. Again, a perfect representation of this can be seen in figure 41.*

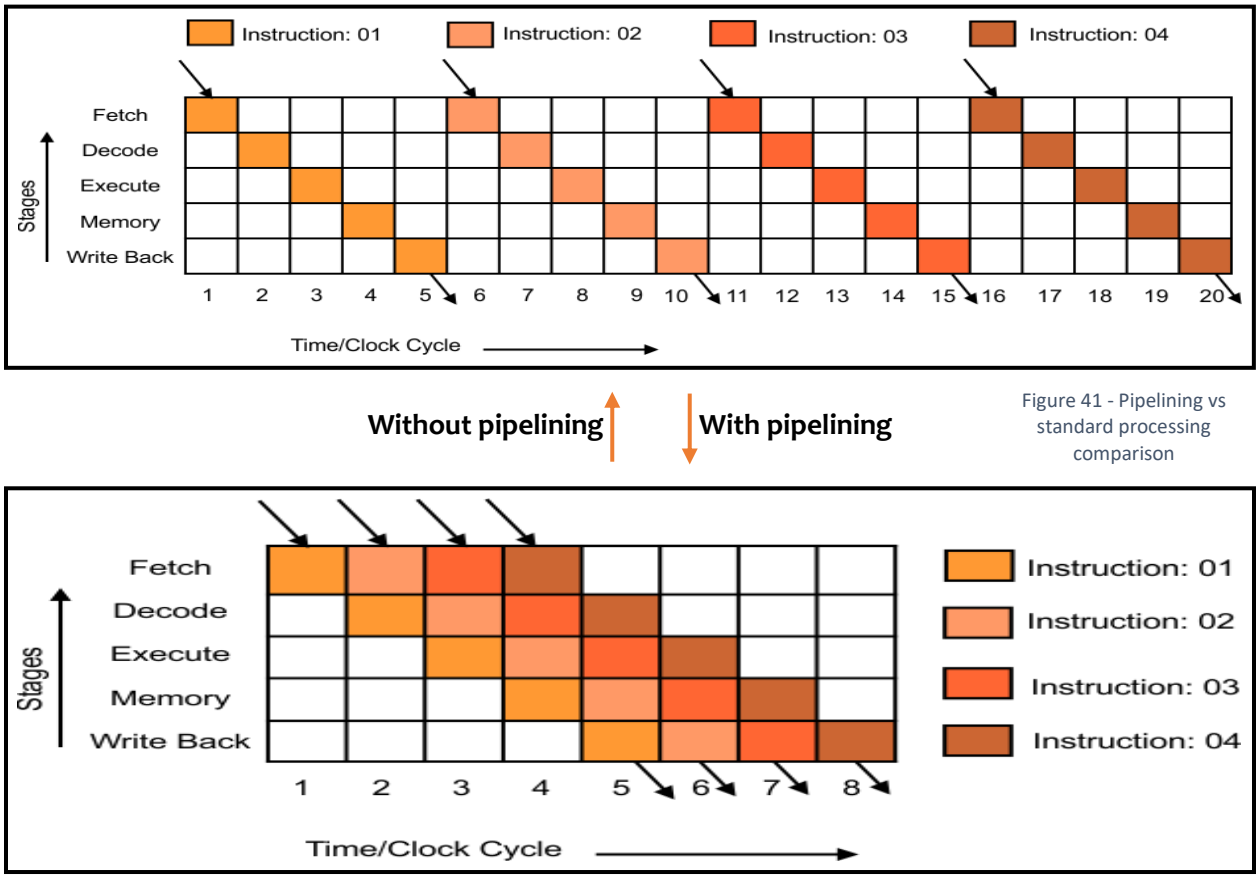


Figure 41 - Pipelining vs standard processing comparison

Pipeline hazards:

What are hazards: *A pipeline hazards is when two instructions in the pipeline sequential to one another are working with the same registers or variables. This means that if the first instruction edits the a register that is meant to be used in the second instruction, then the second instruction would have false data. For example, say the first instruction adds variable "X" by five, and the second instruction stores variable "X" into variable "num". By the time the variable "X" has been increased by five the second instruction is already on its decode stage and cannot fetch X's new value, and so it would store "X" without the added five into "num". This can be seen in the table below. There are multiple ways in which engineers have solved this issue with varying levels of efficiency, with them being pipeline stalling, out-of-order-execution, and speculative execution, and more, however, only the mentioned ones would be discussed.*

Without pipeline stalling					
Process	***Instruction one***				
Fetch	*Fetch X*				
Decode		*Add function*			
Execute			*X = X + 5*		
Store				*Store X*	
Process	***Instruction two***				
Fetch		*Fetch X*			
Decode			*Add X & num*		
Execute				*X + num*	
Store					*Store result*

Resultant of X = X + 5 is needed here

Pipeline stalls: *Pipeline stalling is simply when two registers or variables are being edited after one another and so using a specific logic gate configuration the control unit realizes that and "stalls" (pauses) the second ongoing instruction and consequently the instructions after it. This is done in order to wait for the result of the first instruction and then continue with the process. This is somewhat counterproductive since the whole reason for pipelining is to remove idle time from the process, however, this creates. With that being said, as mentioned by* (0612 TV w/ NERDfirst, 2022)*, the benefits of pipeline outweigh the disadvantages of pipeline stalls, as not having pipelining inherently has more idle time than stalling.*

With pipeline stalling						
Process		***Instruction one***				
Fetch	*Fetch X*					
Decode		*Add function*				
Execute			*X = X + 5*			
Store				*Store result*		
Process		***Instruction two***				
Fetch		*Fetch X*				
Decode			*Add X & num*	*Stalling*		
Execute					*X + num*	
Store						*Store result*

Out-of-order-execution: *Another process for avoiding hazards is out-of-order-execution. This process works by rearranging instructions in a manner so that processes which involve the same variables or registers are not sequential to one another. For example, if two instructions with the same register were after one another, then the third instruction – given that it does not work with the same data – takes place instruction two, with instruction two going to instruction three's location in the pipeline. This way, the instruction with the same variable would be delayed while at the same time another instruction is being executed. This can be considered as an improvement to stalling the operation as this replaces the stalling process with real work. However, due to the huge variety of possible instructions, this process is also much more complicated to implement than pipeline stalling.*

Speculative execution: *Speculative execution is when the pipeline "speculates" the result of a given instruction in the pipeline before it actually finishes, therefore being able to start executing the second instruction immediately without having to stall and wait for the result. Opposite to what the name might suggest, the pipeline does not speculate or guess the resultant or path in which the preceding instruction would go, but rather it is a series of hardwired instructions (logic gate configurations) that force the pipeline to follow a given path based upon the chances that it goes in said direction. This is usually a guess made by the engineers creating the circuitry, hence the name.*

To elucidate further, a major example of this can be seen in for or while loops or found within many programming languages. What happens normally is that the program is stalled and waits for the result of the Boolean expression of the for/while loop. It then proceeding either with the code within the loop in the circumstance that the expression was true or ignoring the loop and moving on if the expression was false. However, as stated by (0612 TV w/ NERDfirst, 2022)*, what occurs in speculative execution is that the circuit "speculates" that the Boolean expression would be true and therefore it would load the instructions found within loop. This is done since most of the time the loop would indeed continue, however, if it was wired in so that it loads in the code after the loop, then in most circumstances then guess would be false since most of the time a loop occurs more than once.*

Now, what this changes is that if the guess was indeed correct, then the instruction would move on perfectly without having to stall. However, if the instruction was false, then the entire pipeline would have to be "flushed" (removed) and start over again as if a pipeline was not even implemented in the first place. That is why when guessing the path a smart guess should be implemented similar to the guess utilized for loops (guess that the expression would be true as to decrease the chances of pipeline flush or a squash.

Overall, a pipeline speculation is a very large series of hardwired "guesses" that can be seen on a number of instructions and not only on loops. If implemented correctly and accurately, this can significantly decrease the chances of a pipeline hazard occurring.

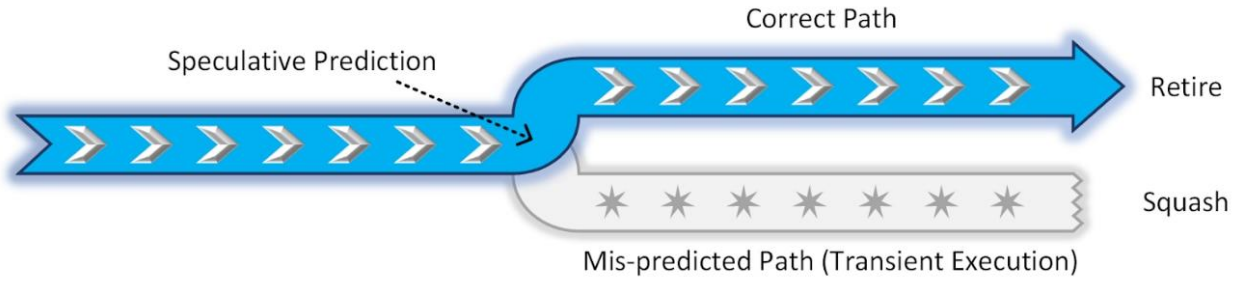


Figure 42 - Abstract illustration of speculative execution

Processing speed:

What limits processing speed:

Speed of electrical transmission: *The overall speed limit of electrical transmission and movement is a factor that can be considered when talking about processing speed, especially since processor nowadays works in speeds reaching timed in picoseconds. With that being said, the speed of electricity is almost as fast as the speed of light itself, as according to* (UCSB ScieneLine, 2022)*, it is only 100 times slower, when talking about such speeds, being a hundred times slower is not that significant of a difference. As such, although it is a factor, it is not something to be concerned about. However, this does nonetheless pose a literal physical limit that cannot possibly be exceeded when it comes to speed even as speed-related technological advancements are made on newer computers.*

Heat: *Heat is possibly the biggest factor when it comes to limiting down the speed of electrical flow. That is the case since the main method of increasing speed that does not involve the use of techniques is increasing the voltage, with the theory behind this being further explained further on. However, as voltage increases and frequency increases, the amount of energy being dissipated into other forms, especially heat, becomes significantly larger. The problem with this is that if the heat within the CPU reaches a given limit, that could very well cause certain pins or parts of the CPU to just simply melt, therefore rendering the CPU inoperable. Although modern CPU have a measure for such an issue known as thermal throttling where the CPU decreases the clock speed in order to reduce heat dissipation, this does also result in slowing down the processor.*

Propagation delays: *The second major factor behind heat that affects the CPU's speed is propagation/gate delays. Said gate delay refers to the time it takes a logic gate to output an appropriate signal based upon the input. In other words, it is the time between when the input signal of a logic gate goes from one stable (constant) state to the other (high to low or vice versa) and the time that the output is able to reach a stable yet valid state based upon the input. The reason that said delay occurs is – as mentioned by* (Altium, 2021)*– simply due to the capacitance found within logic gates that prevent it from charging and discharging in an instantaneous manner, just like capacitors which have a charging and discharging delay.*

Speed enhancing specifications:

Number of cores: *The number of cores found within the CPU determines its capability of multitasking different processes and therefore this inherently increase the speed of the CPU. That is the case since although having multiple cores cannot alleviate the workload on the CPU as a whole, it can decrease the stress being applied by sharing it among multiple cores and therefore ensure that tasks are done in a simultaneous fashion, which is especially important when multiple processes are running at the same time. This is amplified if the CPU has multithreading capabilities per core.*

Cache size: *The cache size is also another important factor when it comes to increase the speed of the CPU. That is the case since the bigger the cache is, the more information it can store and therefore the more likely that a cache hit will occur rather than the CPU having to go the main memory in order to retrieve a piece of information. Retrieving from the cache (cache hit) is significantly faster since – as mentioned earlier – it is much closer to the CPU and is smaller than the main memory, not to mention that it utilized SRAM technology rather than the main memory which uses DRAM.*

Clock speed: *The most important factor when it comes to enhancing the speed of the CPU is the clock speed that it has and its tolerance for overclocking (which would be discussed in the following section). That is the case since as the clock speed increases, the faster the CPU will be done with an instruction, and therefore the faster it can move on to the following instruction. This, alongside the number of cores is possibly the biggest factor affecting CPU's speed.*

IPC: *IPC, short for Instructions Per Cycle, is not necessarily a physical improvement to the CPU, but it is a metric that informs that user on how efficient the CPU is. That is the case since if two CPU have the same clock speed, but CPU one is able to perform more instruction per cycle via the use of different processing techniques such as pipelining or utilizing more complex circuitry for things like multiplication or addition, then CPU one would definitely be more efficient and therefore faster than CPU two.*

Improvement method – overclocking:

What is clock ratio adjustment:

Definition: *As seen in the Clock section, the CPU ratio is one of the more essential properties or specifications about a given CPU and can determine the clock rate and therefore determining the speed and overall performance of the processor. In addition to that, it is also one of the more easily adjustable and editable properties. As such, many users adjust their CPU's clock ratio – and therefore their clock rate – in order for their CPU to suit their own personal needs and applications they demand from their computers in a process known as either overclocking if the ratio is increased and underclocking if it is decreased.*

Method: *his process can simply be done by launching the system's BIOS menu while it is launching and then editing the CPU ratio. It should also be mentioned that in addition to changing the CPU ratio, the voltage going to the CPU should also be adjusted accordingly. While overclocking, a CPU with a higher frequency would require more voltage in order to increase the maximum possible frequency of the processor and vice versa for underclocking. This is required since increasing the maximum frequency is dependent on the logic gate's capability to turn on and turn off the voltage through clock cycle faster or at the frequency given, and more specifically the transistor's found within the logic gates. In that the transistor should be able to turn on before the clock cycle ends and vice versa for when it should turn off. Increasing voltage forces the transistor to increase its speed and decrease its transmission time, which is the time it takes for the transistor to go from a low to a high state. This consequently gives the transistors and the CPU as a whole greater stability, consequently allowing it to run higher frequencies.*

Advantages and disadvantages:

Advantages of clock adjustments: *Over or under clocking a CPU can greatly enhance the user experience based on their specific requirements and demands from their computer. When it comes to overclocking, it is used to speed up the CPU and therefore enhance its performance, which is utilizing whenever the user is using relatively demanding processes and wants to ensure a seamless and lag free experience. This is mostly utilizing when running video games on maximum graphical quality, however, it could nonetheless be seen in other applications such as video editing or CAD modeling as extremely CPU and require a lot of processing power. On the other hand, underclocking can be used to enhance the economics of the CPU, as it decreases power consumption, and therefore improving the battery (on portable computers such as laptops), decreasing cooling noises, decreasing heat emission, and a consequence of that is – most importantly – increase in life span.*

Risks of clock adjustments: *When it comes to underclocking, since the only thing being done is actually decreasing the electrical and thermal strain on the CPU, there is no physical risk that could occur to the CPU due to it. However, on the other hand, overclocking increases the frequency and potential difference, and in consequence increases the power consumption of the CPU. These operations combined can result in extremely high CPU temperature that – if not done in a slow and careful manner – could cause certain pins on the CPU to melt. Most modern system nowadays have an in-built protection system that realizes when the CPU is in a critical state thermally and does what is known as "thermal throttling", where the CPU would automatically decrease the clock its clock, the opposite to what is required from overclocking. As such, an essential prerequisite to overclocking the CPU is to utilize a sufficient cooling system that is capable of keeping the CPU at appropriate working temperatures.*

References

0612 TV w/ NERDfirst. (2022, April 5). CPU Pipelining - The cool way your CPU avoids idle time! Retrieved from https://youtu.be/cZIPxra_apA

Albiva. (2019, August 23). *The Power of Crystals – Do They Work, and How?* Retrieved from albiva: https://www.albiva.com/blogs/news/the-power-of-crystals-do-they-work-and-how#:~:text=The%20property%20that%20makes%20quartz,change%20its%20shape%20very%20slightly.

Altium. (2021, January 25). *Compensating Propagation Delay in Digital Electronics Logic Gates: Keep Your Pulse Trains On Time*. Retrieved from altium: https://resources.altium.com/p/propagation-delay-adjustments-keep-your-pulse-trains-time

Baptiste, C. (2022). *What is the Fetch Decode and Execute cycle of a computer system?* Retrieved from gradeacomputerscience.

Basics Explained, H3Vtux. (2018, August 23). Static RAM and Dynamic RAM Explained. Retrieved from https://www.youtube.com/watch?v=0rNEtAz3wJQ

BBC. (2022). *Data representation*. Retrieved from bbc: https://www.bbc.co.uk/bitesize/guides/zsnbr82/revision/5

Bigelow, S. J. (2022, March). *multicore processor*. Retrieved from techtarget: https://www.techtarget.com/searchdatacenter/definition/multi-core-processor

Byjus. (2022). *Direct Vs. Indirect Addressing Modes: Explore the Difference Between Direct and Indirect Addressing Modes*. Retrieved from byjus: https://byjus.com/gate/difference-between-direct-and-indirect-addressing-modes/#:~:text=The%20direct%20addressing%20mode%20contains,address%20field%20of%20any%20instruction.&text=It%20requires%20no%20memory%20references%20for%20accessing%20the%20data.

Chen, C., Novick, G., & Shimano, K. (2000). *RISC vs. CISC - Stanford Computer Science*. Retrieved from stanford: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/#:~:text=The%20CISC%20approach%20attempts%20to,number%20of%20instructions%20per%20program.

Computer Science. (2019, August 4). Binary 4 – Floating Point Binary Fractions 1. Retrieved from https://youtu.be/L8OYx1I8qNg

Computer Science. (2021, October 19). Harvard Architecture versus Von Neumann Architecture. Retrieved from https://www.youtube.com/watch?v=4nY7mNHLrLk

CrashCourse - PBS. (2017, March 30). Registers and RAM: Crash Course Computer Science #6. Retrieved from https://www.youtube.com/watch?v=fpnE6UAfbtU

Fyfe, D. (2019, April 16). *History. A very brief introduction to Unicode*. Retrieved from medium: https://medium.com/@danfyfe/unicode-7626c437e62b

GeeksForGeeks. (2021, August 4). *Difference between Von Neumann and Harvard Architecture*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/difference-between-von-neumann-and-harvard-architecture/

GeeksForGeeks. (2022, August 24). *Computer Organization | Hardwired v/s Micro-programmed Control Unit*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/computer-organization-hardwired-vs-micro-programmed-control-unit/#:~:text=Hardwired%20control%20is%20faster%20than,can%20operate%20at%20high%20speed.

GeeksForGeeks. (2022, August 16). *Difference between Asymmetric and Symmetric Multiprocessing*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/difference-between-asymmetric-and-symmetric-multiprocessing/#:~:text=In%20asymmetric%20multiprocessing%2C%20the%20processors,the%20processors%20are%20treated%20equally.

GeeksForGeeks. (2022). *Different Classes of CPU Registers*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/different-classes-of-cpu-registers/amp/

GeeksForGeeks. (2022, November 10). *Different Types of RAM (Random Access Memory)*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/different-types-ram-random-access-memory/

GeeksForGeeks. (2022, June 16). *Introduction of Control Unit and its Design*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/introduction-of-control-unit-and-its-design/

Glawion, A. (2022, February 15). *Guide to PCIe Lanes: How many do you need for your workload?* Retrieved from cgdirector: https://www.cgdirector.com/guide-to-pcie-lanes/#:~:text=PCIe%20lanes%20are%20the%20physical,the%20processor%20or%20motherboard%20chipset.

Hruska, J. (2021, February 9). *L2 vs. L3 cache: What's the Difference?* Retrieved from extremetech: https://www.extremetech.com/computing/55662-top-tip-difference-between-l2-and-l3-cache

Huang, B. (2022). *https://learn.sparkfun.com/tutorials/logic-levels/ttl-logic-levels#:~:text=Likewise%2C%20the%20maximum%20output%20LOW,when%20read%20into%20the%20device.* Retrieved from sparkfun: https://learn.sparkfun.com/tutorials/logic-levels/ttl-logic-levels#:~:text=Likewise%2C%20the%20maximum%20output%20LOW,when%20read%20into%20the%20device.

IBM. (2022). *The First Multi-Core, 1GHz Processor*. Retrieved from ibm.

Jenkov, J. (2022, 8 6). *Unicode*. Retrieved from jenkov: https://jenkov.com/tutorials/unicode/index.html

Khandelwal, R. (2020, August 12). *Multi-Threading and MultiProcessing in Python*. Retrieved from gitconnected: https://levelup.gitconnected.com/multi-threading-and-multiprocessing-in-python-3d5662f4a528

Kukunas, J. (2015). *Power and Performance.* Elsevier Inc. .

Lague, S. (2020, November 16). Exploring How Computers Work. Retrieved from https://www.youtube.com/watch?v=QZwneRb-zqA

Learn Computer Science Online. (2022). *Instruction Format*. Retrieved from learncomputerscienceonline: https://www.learncomputerscienceonline.com/instruction-format/#:~:text=The%20instruction%20format%20is%20simply,are%20grouped%20together%20called%20fields.

Michael, S. S. (2019, February 7). *What Is a Microarchitecture? Understanding Processors and Register Files in an ARM Core*. Retrieved from allaboutcircuits: https://www.allaboutcircuits.com/technical-articles/what-is-a-microarchitecture-processor-register-files-ARM-core/

Mitton, R. (2015, June 10). *The Death Of The Von Neumann Architecture*. Retrieved from codersnotes: http://www.codersnotes.com/notes/the-death-of-the-von-neumann-architecture/#:~:text=What%20this%20means%20is%20that,convert%20its%20data%20into%20code.

Mugerwa, S. (2022, September 12). *The difference between CPU and Cores*. Retrieved from dignited: https://www.dignited.com/36727/the-difference-between-cpu-vs-cores/#:~:text=A%20core%20is%20the%20most,Unit%20and%20set%20of%20registers.

Myrianthous, G. (2021, January). *Multi-threading and Multi-processing in Python*. Retrieved from towardsdatascience: https://towardsdatascience.com/multithreading-multiprocessing-python-180d0975ab29#:~:text=In%20fact%2C%20multiprocessing%20module%20lets,literally%20at%20the%20same%20time.

Neso Academy. (2018, February 21). Computer System Architecture. Retrieved from https://www.youtube.com/watch?v=So9SR3qpWsM

Newth, A. (2022, November 24). *What Is a Self-Modifying Code?* Retrieved from easytechjunkie: https://www.easytechjunkie.com/what-is-a-self-modifying-code.htm

PBS Studio - Crashcourse. (2017, April 27). Advanced CPU Designs: Crash Course Computer Science #9. Retrieved from https://www.youtube.com/watch?v=rtAlC5J1U40

PC mag. (2022). *RISC*. Retrieved from pcmag: https://www.pcmag.com/encyclopedia/term/risc#:~:text=The%20most%20widely%20used%20RISC,smartphone%20and%20countless%20electronic%20devices.

Rambus Press. (2016, October 5). *Maximizing Von Neumann architecture*. Retrieved from rambus: https://www.rambus.com/blogs/maximizing-von-neumann-architecture-2/

Sheldon, R. (2022). *stack pointer*. Retrieved from techtarget: https://www.techtarget.com/whatis/definition/stack-pointer

Simply Explained. (2020, November 25). Caching - Simply Explained. Retrieved from https://www.youtube.com/watch?v=6FyXURRVmR0

Smith, M. (2022, March 18). *Quantum computing: Definition, facts & uses*. Retrieved from livescience: https://www.livescience.com/quantum-computing#:~:text=Quantum%20computing%20is%20a%20new%20generation%20of%20technology%20that%20involves,supercomputer%2010%2C000%20years%20to%20accomplish.

Tech Computer Science. (2022). *Harvard Architecture*. Retrieved from teachcomputerscience: https://teachcomputerscience.com/harvard-architecture/

UCSB ScieneLine. (2022). *We want to know which is faster: electricity or light?* Retrieved from ucsb: http://scienceline.ucsb.edu/getkey.php?key=2910#:~:text=Light%20travels%20through%20empty%20space,th%20the%20speed%20of%20light.

Williams, T. (2019, 10 10). *Why Do Computers use Binary Numbers?* Retrieved from geek-computer: https://www.geek-computer.com/wiki/why-do-computers-use-binary-numbers

Wreggit, N. (2022). *John Von Neumann and Computer Architecture*. Retrieved from washington: https://courses.cs.washington.edu/courses/cse490h1/19wi/exhibit/john-von-neumann-1.html#:~:text=Harvard%20Architecture%20(1939),which%20allow%20for%20simultaneous%20access.

"*A system of logical instructions that an automaton can carry out and which causes the automaton to perform some organized task is called a code.*"

- John Von Neuman

Inventor of the Von Neuman architecture