- **Name:** Vivek Khimani (Freshmen)

- **Project Title:** Understanding the Apollo Guidance Computer (AGC).

- **Project Focus:** The primary focus of the project is to provide a detailed account of the AGC architecture and compare it with the system used in modern computers. The paper will explain the significant architectural aspects like- memory model, instruction set in the machine code and its interpretation, instruction encoding, distribution and functions of registers, I/O devices and their instruction sets, and some of the unusual features of AGC architecture (when compared to the modern computers).

- **School:** Drexel University

- **Degree:** BS in Computer Science (BSCS)

- **Course and Section:** CS 164, Section C (Honors)

- **Professor:** Dr Brian Stuart

- **Presentation Type:** Research Paper

# Table of Contents

# 1. General Introduction:

## 1.1. The Apollo Mission:

**Apollo** was the project conducted by United States National Aeronautics and Space Administration (NASA) between the 1960s and '70s, which was dedicated to **President John F. Kennedy's** national goal of ***landing first man on the Moon and returning him safely to the Earth.*** This was finally accomplished on **July 20, 1969** when **Neil Armstrong (**along with **Buzz Aldrin** and **Michael Collins**) became the first man to step on the moon. Undoubtedly, the team of scientists and Apollo astronauts are the real backbone of this success, this project report is based on the ***Apollo Guidance Computer (Designed and programmed at MIT)***, which was first used in the **Apollo 7** (first successful crewed mission after the Apollo 1 disaster). *Supporting Source: Britannica (Citation at the End).*

## 1.2. The Apollo Guidance Computer:

As soon as the mission was declared by President Kennedy, MIT was given a contract in 1961 to create a computer that would support such a mission. It took them a span of **5 years (1961-1966)** and a cost of **200,000 USD** each to build a total of **42 AGCs** that supported the Apollo moon landings between 1969-1972. In the mid-1960s it wasn't possible to put any computer on the aircraft because the first mini computers were of the size of a **small fridge**, **too heavy**, **too power hungry**, and **too slow** for the real-time scientific calculations. But, the AGC was **compacted to a size of suitcase (55*33*15cm)**, with a weight of **32 kg**, and consumed about **55W**. This was a huge achievement in that era (although it was huge compared to our computers). *Factual source: www.history.nasa.gov/computers/Ch2-1.html.*

# 2. Understanding the Architecture:

## 2.1. Introduction:

***Computer Architecture*** can be defined as a specification detailing how a set of software and hardware technology standards interact to form a computer system or platform (A definition by Techopedia). Before digging deep into the AGC architecture, let's compare some of the basic features that clearly distinguishes the capabilities of the AGC compared to the modern computer we use today:

- The **clock speed** (*put simply*, a rate at which processor can complete a processing cycle) of AGC was **1.024 MHz,** which is almost **2344 times** less than the processor (Intel Core i5-6200) I am using on my computer to type this paper. Also, the modern processors are way more efficient because of which they can get **more work done per clock cycle** compared to the **older processors.**
- All data on this computer is stored in form a **15-bit word.**

- Originally, it supported **2 KW of RAM** and **36 KW of ROM** which was increased as and when required by the software.

## 2.2. Instruction Set:

The **Instruction Set** can be basically defined as set of commands for the CPU in **Machine Language.** The AGC instruction set varies widely when compared to the modern-day computers. It can be compared in the following manner:

- The **armv8** instruction set for the modern computers is one of the most complex instruction sets designed for unbeatable performance. It comprises of **400 instructions**
- While **subleq** is considered as the **simplest instruction model** which shows that a **single instruction** can be enough to solve a problem.
- The **AGC instruction set** was in between these two which comprised of **36 instructions** for varied range of functions.
- In that era, **more complex instruction,** required **high code density,** and **higher complexity of CPU.**
- To **avoid this complexity** the **AGC Instruction set** was **tactfully** designed in a way that it was just enough for the mission and **used minimal memory.**

But, to precisely understand the logic behind instruction, it is extremely important to understand the **Memory Model** used in the **AGC**. As the instructions are chiefly **stored** and **written** in the memory, it is one of the most important **cornerstones** of not only the instruction set, but also the **Computer Architecture** as a whole.

## 2.3. Memory Model:

As described earlier, the **Memory Model** is one of the most important features in a computer because the **architecture** is largely based on the **availability of memory**, which in turn affect the **software** and **hardware.** In 1960s, designing an efficient memory model was one of the greatest challenges faced by the makers of AGC because there was a limited access to memory. Ultimately, they ended up with the following model:

- Memory consisted of **4096 cells,** each numbered from **000** to **FFF** in **hexadecimal.**
- Each cell can contain a **maximum** of **15-bit value** which is from **0000** to **7FFF** in **hexadecimal.**
- Almost all the changes taking place in the memory had to go through the **15-bit accumulator (Register A)** which was a part of the **memory itself.**
- AGC had a **single address space** for **code** and **data** which made it a **Von Neumann Machine.**
- We can **COPY, ADD, SUBTRACT,** and **MULTIPLY** data **between** an **accumulator** and **AGC.**
- **Different data** in the **memory** can have **different meanings** based on the method in which they are **interpreted.**
- **PC (Program Counter)** was one of the **registers,** which used to **hold** the **address of the instruction** which was to be **executed next.** *Put simply,* it was responsible to **control** the **execution flow** in the AGC.

Let's further look at the **detailed instruction set** in the AGC and their functioning.

## 2.4.Load, Store, Arithmetic, and Logic Instructions:

Before jumping to the individual instructions and their meanings, it is important to understand the **syntax** used for the interpretation. The **same syntax** will be used throughout the paper to explain the different set of instructions:

- [k] -- are represented as the cardholders for the **memory address.**
- 'a' -- represents the **argument** (can be also interpreted as '**accumulator**')
- 'b' -- this will show up in the instructions which requires **double word values (**double the 15 bits available in the accumulator). It is a **register** which is **mostly** used **along with the accumulator** to **store** the **double word values.**

| | |
|---|---|
| **ld a, [k]** | **LOADS** the value **from [k] block in the memory** to **ACCUMULATOR**. (this is why it makes sense to consider 'a' as accumulator) |
| **add a, [k]** | **ADDS** the value **from [k] block in the memory** to the value in **ACCUMULATOR**. |
| **ld [k], a** | **STORES** the value **from accumulator** to the **[k] block in the memory.** |
| **xchg a, [k]** | **SWAPS** the **values between accumulator and [k] block in the memory.** |
| **sub a, [k]** | **SUBTRACTS** the **value from [k] from the value in accumulator and stores the result in accumulator.** (This can produce **negative results,** which are explained in the next section). |
| **ldc a, [k]** | **NEGATES (REVERSAL OF ALL THE BITS)** in **the given memory block [k]** and **LOADS it in the accumulator.** |
| **inc [k]** | **ADDS ONE to the value in [k] block of the memory WITHOUT assigning it in the accumulator.** |
| **aug [k]** | **ADDS ONE to the POSITIVE VALUE in [k]** and **SUBTRACTS ONE from the NEGATIVE VALUE in [k].** |
| **dim [k]** | **SUBTRACTS ONE from the POSITIVE VALUES in [k]** and **ADDS ONE to the NEGATIVE VALUES in [k]**. |
| **mul [k]** | **MULTIPLIES the number from [k] block with that in accumulator** and then **stores the result in the TWO registers 'a' (accumulator) and 'b'.** (This is where **'b' register** comes into the picture because **multiplication result** cannot be stored in **15-bit accumulator alone.** So, the **higher bits of the product** are stored in **'a'** and **lower bits of the product** are stored in **'b').** |
| **div [k]** | **DIVIDES** the **value from [k] block** by the **value in accumulator** and **stores the QUOTIENT** in **'a' register** and **REMAINDER** in **'b' register**. |

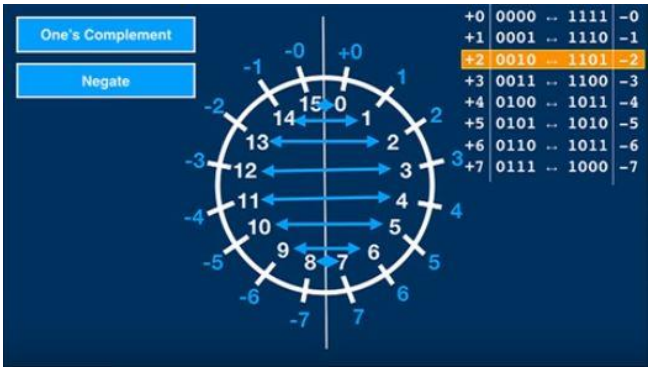
| | |
|---|---|
| **ld ab, [k]** | **LOADS** the **values from [k] block** and **ONE BLOCK BELOW THE [k] block** to **'a' and 'b' registers** as **DOUBLE WORD**. |
| **ldc ab, [k]** | **LOADS** the **NEGATED or COMPLEMENTED value from [k] block** and **ONE BLOCK BELOW THE [k] block** to **'a' and 'b' registers** as **DOUBLE WORD**. |
| **xchg ab, [k]** | **SWAPS** the **value from [k] block and ONE BLOCK BELOW THE [k] block** with **the values in 'a' and 'b' registers.** |
| **add ab, [k]** | **ADDS** the **value from [k] block and ONE BLOCK BELOW THE [k] block** TO **the values in 'a' and 'b' registers.** |
| **xchg b, [k]** | **SWAPS** the **value from [k] block in the memory** with the **value in 'b' register.** |
| **ld a, [k1+[k2]]** | **READS** the **value from [k2] block first**, **ADDS it to [k1] value**. **LOADS the value from the RESULTANT numbered block** in **the memory to A.** (This is how we can execute **"INDIRECT INDEXING"** in the AGC). |
| **ld a, [[k]]** | **READS THE VALUE FROM [k] block** and **adds it to 0**. **LOADS** the **value from the RESULTANT numbered block in the memory** to **'a' register.** (This is how we can shift our **Program Counter** to a specific location in the memory. This method can be indirectly used as a **"pointer").** |

## 2.5. Negative Number Representation:

AGC uses **ONE's COMPLEMENT** to represent the **NEGATIVE NUMBERS.** The representation in this format can be simply done by **reversing the bits** of a **given positive number.**

- However, these were some of the possible drawbacks,
  A. The **one's complement** resulted in **two values of zero (+0 and -0)** which practically doesn't make a lot of sense.
  B. Addition of various numbers can result into **positive** or **negative** overflow based on the values of numbers being added.
  C. It was only possible to use this method for **decimal numbers upto 7.**
- But, there was also a **brief logic** behind using **one's complement,**
  - It was possible to represent 1 to 15 **decimal numbers** using **4 bits.** So the numbers **0 to 7 represent +ve unsigned number,** while **8 to 15 represent -ve unsigned number.**
  - Therefore, this **simple reversal of bits** and the **concept of compliments** can be understood using a **circle (following diagram),** which in turns makes it **easier to understand** the **concept of carry and overflow.**

**NOTE:** The **concept of circle** (which will follow) is just used to explain the **concept of addition, carry, and overflows** in AGC. The **actual operations are done in the form of bits.** But, the same concepts apply. (Example: Incorrect bits are produced in overflow conditions which produces an unexpected result)




*Source: [https://www.youtube.com/watch?v=xx7Lfh5SKUQ&t=2480s](https://www.youtube.com/watch?v=xx7Lfh5SKUQ&t=2480s) (10:31)*

- Using this circle, let's understand an example of **SIMPLE ADDITION:**
  - (-5) + (2) #(-5) is represented as 10 using the complimentary value from the circle].
  - (10) + (2)
  - 12 == **(-3)** #[Again, use the circle!]
  - Therefore, **one's complement works perfectly for such simple additions.**

- Now, let's look at an example of **END-AROUND CARRY:**
  - (6) - (4)
  - (6) + (-4)
  - (6) + (11) #[Using the compliment from the circle.]
  - 17 #[This doesn't exist on the circle because it gets completed on 15. So, a **completion of circle represents CARRY]**
  - 1 + CARRY #[17 **corresponds** to 1 after completing the circle. And we include **carry,** which is **added** as 1]
  - **2** #[This gives us the correct result]

  **BUT, THESE RESULTS DON'T APPLY IN THE CASE OF NUMBERS WHICH RESULT INTO AN OVERFLOW!**

- **Positive Overflow**: The condition in which **the result of addition of two positive numbers does not fit into positive (unsigned) number space on the circle.**
  - (7) + (1)
  - (8) #This represents (-7) on the circle, which is INCORRECT.
  - This is called a **positive overflow.**

- **Negative Overflow**: The condition in which **the result of addition of two negative numbers does not fit into negative number space on the circle.**
  - (-7) + (-1)
  - (8) + (14) #When we consider the values from the circle.
  - 6 + CARRY #This represent (7) on the circle, which is INCORRECT.
  - This is called a **negative overflow.**

Remember, OVERFLOW condition doesn't mean that AGC will produce incorrect results in such cases. There's a solution for the overflow conditions too. When accumulator receives an overflowed value, it stores that value like any other value. But, it has an additional bit which stores the information about overflow condition. For example, (7) in the negative overflow example, will be stored with an EXTRA OVERFLOW BIT. So, while storing this value in a memory block, first the accumulator sends (7) which is stored. Then, it writes +1 or -1 in the memory block depending on the positive or negative overflow. This acts a signed carry and is stored in a higher order block (the block above). Therefore, these TWO BLOCKS TOGETHER store a negative value in form of a DOUBLE WORD result in the memory.

## 2.6.RAM and ROM:

- Original memory was comprised of **4096 words** in total (It was increased as per the needs of software).
- This original memory was composed of:
  - **1072 words of RAM.**
  - **3072 words of ROM.**
- However, the RAM and ROM were further divided in specific manner to provide efficient functionality of the machine.

- **DIVISION OF RAM:**
  a. **768 words** of **FIXED RAM**, **256 words** of **BANKED RAM.**
  b. The **7 words of the registers** are the **parts of FIXED RAM.**
  c. The **BANKED RAM has 8 (0 to 7) banks stacked over it.** The **register EB** is used to **switch between these banks**. When **EB is POINTED (using double index)** to a specific number from 0 to 7, that bank occupies the BANKED RAM area.
  d. The **FIXED RAM** in the **RAM** area is not occupied by anything else but the **banks 0,1, and 2** (these are from the same 0 to 7 banks in the banked RAM area).

- **DIVISION OF ROM:**
  a. **1024 words** of **BANKED ROM**, **2048 words** of **FIXED ROM**.
  b. The **BANKED ROM** has **32 banks** stacked over it. The **register FB is used to switch between these banks**. The **same pointing or indexing process as EB is to be used**.
  c. However, **more than 32 banks of ROM were added last minute which was called SUPERBANK**. This **SUPERBANK can be switched in the place of UPPER MOST 8 banks when all of them have been used** and **later can be used at the same locations** as those of the **existing 8 banks**. This makes a total of **40 banks of BANKED ROM.**
  d. The **FIXED ROM** in the ROM area is occupied by the **BANKS number 2 and 3** which cannot be changed (these are from the same 0 to 40 banks in the banked ROM area).
  **THIS FLEXIBILITY OF SWITCHING THE RAM AND ROM BANKS FORMS AN IMPORTANT PART OF THE <u>CONTROL FLOW INSTRUCTIONS!</u>**

## 2.7. Control Flow Instructions:

Normally, the instructions are executed in a **sequential manner.** The **"Program Counter"** (which is responsible for the execution of instruction), is incremented everytime an instruction is executed. This results in a **sequential execution of the code.** However, there are a few **control flow instructions** which can alter this normal flow of execution to provide the makers with **increased functionality.** The **syntax** is the same as **load, store, arithmetic, and logic** instructions.

| | |
|---|---|
| **jmp [k]** | **LOADS the ARGUEMENT [k]** in the **Program Counter (PC) register**, which means that **EXECUTION WILL CONTINUE AT THAT ADDRESS ([k]).** |
| **jz [k]** | **IMPLEMENTS the JUMP INSTRUCTION**, **ONLY IF register 'a' has the value zero**, otherwise **it continues with the next instruction**. |
| **jlez [k]** | **ONLY IMPLEMENTS the JUMP INSTRUCTION**, **ONLY IF register 'a' has -ve or zero**, otherwise **it continues the next instruction**. |
| **ccs a** | **COUNT, COMPARE, and SKIP: A set of FOUR commands** out of which **THREE are CONDITIONAL JUMP COMMANDS (a>+0; a=+0; a<-0)** and the the **FOURTH one is ld a, [k] command (with condition =-0)**. So if **any ONE** of the **FIRST THREE** conditions are **satisfied**, the **PC eliminates the next two from them** and **jumps to "ld a, [k]" instruction along with** the **JMP** (out of **FIRST THREE**) **instructions**. Therefore, these instructions are **used for LOOPS whose conditions are based on 'a'.** |
| **call [k]** | **Similar** to **JMP** but in this case, **PC is incremented first** (so it can move to the next instruction) and **the incremented PC is copied to LR (Link Register)**. And than **"call arguement"([k])** is **copied to Program Counter (PC)** so that **execution can continue from there**. |
| **ret** | It is used at the **end of the "call" instruction**. It **contains** the **address of LR** so that once it is issued, the **data from LR is copied to PC** and the **execution can be continued** from **wherever it was left (due to "call)**. Hence, **"ret"** is always used with **"call"**. |
| **xchg lr, [k]** | Used when user wants to **call something within a call instruction**. This stores the **existing LR information (for the main call)** in the **[k] memory block** and **performs ANOTHER CALL INSIDE (which overwrites the LR)**. Once the **LR from another call is "ret" to PC** and execution continues, the **LR is EXCHANGED again from the memory block** and **"ret" is used to return the main LR value to PC and overwrites it**. This is **similar to the principle of "nested if" in modern coding languages**. |
| **ld eb, a** | **EB is the register numbered 003** on the memory **which can be used to switch the RAM banks**. So, when you **LOAD** a **value in "a" from 000 to 007** into "eb", it **will LOAD that numbered bank into BANKED RAM section**. |
| **ld fb, a** | **FB is the register numbered 004** on the memory **which can be used to switch the ROM banks**. So, when you **LOAD a value in "a" from 000 to 007** into "fb", it **will LOAD that numbered bank into the BANKED ROM section.** |

**ALERT:** If my **PC LOADS some BANK** into the **BANKED RAM** or the **BANKED ROM section** due to the above command, and **then if the next instruction** (function call for example) **refers to an EXISTING bank**, **the function call will not work on the updated bank** and the **remaining instructions for the existing bank would not work**. So, the

**makers included "callf" function which helps us to change FB and PC structure simultaneously and solve this issue.**

| | |
|---|---|
| **callf** | It is **similar to existing "xchg ab, [k]" instruction** because what it has to do is to **CHANGE FB and PC together**. So, for this to happen, we will have to **LOAD the values for FB and PC into "A" and "B" first**, and **then "callf" will load these values in FB and PC from "A" and "B"**. |
| **retf** | This function is **used with "callf" to GET BACK the existing values of "FB" and "PC"**. |

**BB (Both Banks) register:** Two **banks register (FB and Eb)** hold **5 bits (32 banks)** and **3 bits (8 banks)** respectively. Other **bits are ZERO**. There is **another register BB (Both Banks)** which has the **information combined from BOTH THE BANK REGISTERS**.

| | |
|---|---|
| **callfbb** | **DOUBLE WORD INSTRUCTION** that can **UPDATE PC and BOTH BANKS (FB and EB) altogether**. It **works** on the same **"xchg ab" principle**, but **LOADS PC and BB into A and B.** This function gives us **access to the subroutines whose variables are often stored on different banks**. |
| **retfbb** | **Used with "callfbb"** to **get back the existing values of "BB" and "PC".** |

**NOTE:** This instruction explains us the **UNUSUAL ORDERING FOR THE BANK REGISTERS**. They were ordered in the following order in the memory. EB, FB, PC, BB. Therefore, **when "callf" or "callfbb" instructions are given** the **DOUBLE WORD INSTRUCTION** can be **LOADED** into **FB and PC (for callf**) and **PC and BB (for callfbb)**. As **BOTH THE RAM banks can be switched** with **ROM bank and PC using "callfbb"**, so **"EB" register is not used to store any DOUBLE WORD VALUE.**

## 2.8. Primary Registers:

There are **eight primary registers** which are assigned **first eight blocks in the memory (000 to 007).** This feature allows **greater flexibility** in the instruction set. For example: While LOADING some value in register 'a' from 'b', we can treat 'b' as the memory block by using the NORMAL LOAD INSTRUCTION. However, the registers have **specific functions** and a **specific order (one of the reasons behind this was seen in CONTROL FLOW INSTRUCTIONS).**

| **Register** | **Memory Location** | **Function** |
|---|---|---|
| A | 000 | Used as a **Primary Accumulator.** |
| B | 001 | Used with 'a' to **store Double Word Values.** |
| LR | 002 | Link Register- **Stores the Tem.** |
| EB | 003 | Used to **switch between RAM BANKS** in the banked RAM area. |
| FB | 004 | Used to **switch between ROM BANKS** in the banked ROM area. |
| PC | 005 | **Controls** and **keeps a track** of **execution** of the instructions. |

| | | |
|---|---|---|
| BB | 006 | **Both Banks-** Stores **combined information of FB and EB.** Used in **"callfbb" instruction** to switch **EB, FB, and PC altogether.** |
| 0000 | 007 | **Zero Register-** When we **READ FROM IT** we get **ZERO,** and when we **WRITE TO IT** the value is **DISCARDED.** |

## 2.9.Editing Registers:

After the **primary register (000 to 007)**, memory contains **eight more "shadow registers" (will be understood after the INTERRUPTS). Thereafter, four "editing registers" (010 to 013)** make up for different **shift and rotate functions.**

| **Register** | **Memory Location** | **Function** |
|---|---|---|
| ROR | 010 | When reading a 15-bit value **cause the bits to move right by one space** and **the last bit (from the right) will be rotated and become the left most bit**. |
| SHR | 011 | **Shifts all bits by one place to the right** and **discards the last bit (from the right)**. And **replaces the left most bits with a 0 bit.** |
| ROL | 012 | When reading a 15-bit value, this will **cause the bits to move left by one space** and the **last bit (from the left) will be rotated and become the right most bit.** |
| SHR7 | 013 | **Shifts all bits by SEVEN places to the right** and **discards the last SEVEN bits (from the right)**. And it **replaces the SEVEN LEFTMOST BITS** with **SEVEN ZERO BITS**. |

## 2.10. I/O Channels:

A computer has **CPU** at the centre which is connected to **memory (already discussed)** at one end and other **peripherals which form the Input/Output (I/O Channel).** The I/O channels are **not a part of memory**, but they have a **different address space.** There are **512 I/O channels** labelled from **000 to 1FF.** Each of them is **15-bit in size,** similar to **memory. "in"** and **"out"** instructions can **read from** and can **write words to** I/O channels. These channels contain **15 individual control bits from different devices.** These control bits are used to **control the devices.**

## 2.11. I/O Channels Instructions:

| | |
|---|---|
| **in a, [kc]** | Provides an **INPUT to the I/O channels**. |
| **out [kc], a** | Generates an **output from the I/O channels.** |
| **out\| [kc], a** | **SENDS INDIVIDUAL bits** for **the control of devices.** |

| | |
|---|---|
| **out& [kc], a** | **CLEARS INDIVIDUAL bits** to **turn off the devices or reversal**. |
| **in\| a, [kc]** | **RECEIVES INDIVIDUAL bits** from **other devices (which in turn receives bits from users).** |
| **in& a, [kc]** | **CLEARS the received individual bits**. |
| **in^ a, [kc]** | **DOES not take an input** if **two inputs are same**. |

**NOTE:** All the above instructions are **ONLY RESTRICTED TO I/O channels.** The **ONLY BOOLEAN OPERATION** for the **memory and accumulator** is represented below:

1. **and a, [k]**

## 2.12. Shadow Registers:

These are the registers in the memory which are placed **after the primary registers**.

| Register | Memory Location | Function |
|---|---|---|
| A' | 008 | Used by **software if needed.** |
| B' | 009 | Used by **software if needed.** |
| LR' | 00A | Used by **software for specific needs**. |
| 00B | 00B | There is no shadow register like **EB'.** |
| 00C | 000C | There is no shadow register like **FB'.** |
| PC' | 00D | **Temporarily stores PC** in case of an **interrupt (will be explained in the next section).** |
| BB' | 00E | Used by **software if needed.** |
| IR' | 00F | **Temporarily stores IR** in case of an **interrupt (will be explained in the next section).** |

## 2.13. Interrupts:

When **I/O devices** need **attention of the CPU** they can **INTERRUPT normal execution.** There is a **register called IR**, which **stores the value (or OPCODE) in the given PC**. When an **interrupt is caused**, the **PC at that point is copied into shadowed memory location PC'** and **IR is copied to IR'**. The **interrupt causes the PC to jump to some specified location** (which **depends on the type of an interrupt**). **"iret" instruction** is **used after the interrupt to copy back the PC' and IR'** into the **respective PC and IR registers** so the **normal execution can continue**. So, the **PC' and IR'** from the **shadowed registers** are **automatically used by computer in case of an interrupt.** However, **OVERFLOW CONDITION FLAG cannot be saved or restored**. Therefore, **during an OVERFLOW,** the **interrupts will be DISABLED until the next STORE INSTRUCTION (which terminates the overflow).**

## 2.14. Interrupt Instructions:

There are various instructions in AGC which can be used to **control the interrupts.**

| | |
|---|---|
| **iret** | Used **after an interrupt** to **copy back** the **PC' and IR'** into **PC and IR** so **that normal execution of the code can continue**. |
| **int k** | **PAUSE interrupts** in **SOFTWARE**. |
| **cli** | **Enables INTERRUPT globally.** |
| **sti** | **Disables INTERRUPT globally**. |

## 2.15. Instruction Encoding:

***Instruction Encoding*** means representing entire INSTRUCTION as a BINARY VALUE. For example: In **ld a, [k] instruction,** the **FIRST THREE BITS** are **OPCODE** and represent ld a; and the **remaining bits (12 bits)** represent the **memory address [k] in binary**. Following this, we can ONLY have a total of EIGHT primary instructions as follows:

| OPCODE | ADDRESS | INSTRUCTION |
|---|---|---|
| 000 | remaining memory address (12 bits) | call k |
| 001 | remaining memory address (12 bits) | ccs [k] |
| 010 | remaining memory address (12 bits) | inc [k] |
| 011 | remaining memory address (12 bits) | ld a, [k] |
| 100 | remaining memory address (12 bits) | ldc a, [k] |
| 101 | remaining memory address (12 bits) | ld [k], a |
| 110 | remaining memory address (12 bits) | add a, [k] |
| 111 | remaining memory address (12 bits) | and a, [k] |

**These are the only possible set of instructions if we use FIRST THREE BITS for OPCODE. The ADDRESS ENCODING helps in representing the remaining instructions:**

- **RAM address always starts with 00.**
- **ROM address** start with **anything except 00.**

Going back to the table mentioned above,

1. The STORE INSTRUCTION (ld [k], a) is possible only on **RAM.** So the **next two bits after specified opcode** are filled with **00 (for RAM).** And it has to be filled with the **remaining TEN address bits instead of TWELVE.** This makes room for another **THREE INSTRUCTIONS USING STORE OPCODE (101) in the beginning.** A set of **four instructions** can be represented in the following manner:

| **OPCODE** | **ADDRESS** | **INSTRUCTION** |
|---|---|---|
| 101 | 00 + 10 bits memory address. | INDEX (this will be explained later) |
| 101 | 01 + 10 bits memory address. | xchg ab, [k] |
| 101 | 10 + 10 bits memory address. | ld [k], a |
| 101 | 11 + 10 bits memory address. | xchg a, [k] |

2. Similarly, the INCREMENT instruction is also the **RAM only instruction.** This helps us to form **one more set of instructions in following manner:**

| **OPCODE** | **ADDRESS** | **INSTRUCTION** |
|---|---|---|
| 010 | 00 + 10 bits memory address. | add k, [ab] |
| 010 | 01 + 10 bits memory address. | xchg b, [k] |
| 010 | 10 + 10 bits memory address. | inc [k] |
| 010 | 11 + 10 bits memory address. | add [k], a |

3. Similarly, **CCS instruction** shares an **OPCODE with JUMP which is ROM only instruction:**

| **OPCODE** | **ADDRESS** | **INSTRUCTION** |
|---|---|---|
| 001 | 00 + 10 bits memory address. | ccs k |
| 001 | Any except 00 + 10 bits memory address. | jmp |

4. Further, **jumps** to the bank registers aren't practically possible, so the **call opcode** is shared with **sti, cli, and EXTEND** instructions in following manner:

| **OPCODE** | **ADDRESS** | **INSTRUCTION** |
|---|---|---|
| 000 | remaining memory address (12 bits) | call k |
| 000 | 000000000010 (calls LR-002 on memory) | return |
| 000 | 000000000011 (calls EB-003 on memory) | sti |
| 000 | 000000000100 (calls FB-004 on memory) | cli |
| 000 | 000000000110 (calls BB-006 on memory) | EXTEND |

**HERE, EXTEND IS THE PREFIX WHICH IS CAPABLE TO CHANGE THE MEANING OF THE OPCODE OF NEXT INSTRUCTION, WHICH GIVES RISE TO A WHOLE NEW SET OF INSTRUCTIONS THAT CAN BE ENCODED.**

**NOTE:**

1. **INDEX ADDRESSING** is achieved by using INDEX PREFIX (in the 101-opcode group mentioned above). This consists of two instruction words as follows:
   A. INDEX
   B. ld a, [k]
   Represented in the following manner:

| OPCODE | ADDRESS | INSTRUCTION |
|---|---|---|
| 101 | 00 (only RAM) | INDEX |
| 011 | RAM or ROM | ld a, [k] |

Hence, **INDEX** is an actual instruction. When the above-mentioned instructions are **executed together**, they are e**quivalent to ld a, [k1 + [k2]]** instruction, where **k1- index address and k2- memory address.**

- If an **INTERRUPT** is generated while **indexing**, the **IR' contains effective instruction code** which can be **restored at the end of an interrupt handler.**

2. **"iret"- INDEX ENCODING WITH SPECIAL MEANING:**

- **"iret"** is used to **return to the original PC with the same OPCODE value when an INTERRUPT is generated**. This is done by **restoring the original OPCODE value from IR' which uses the principle of INDEX instruction**. Therefore, **both use the same OPCODE.**

| OPCODE | ADDRESS | INSTRUCTION |
|---|---|---|
| 101 | 00 (only RAM) | INDEX |
| 101 | 000000001111 (for IR'-00F on memory) | iret |

## 2.16. Unusual Features of AGC Architecture:

Considering the **memory constraints, limitations, and requirements** the architecture used in the AGC was undoubtedly one of the best in 1960s. However, there were some **unusual features** which makes the architecture questionable.

1. **One's complement** was used for the representation of negative numbers instead of **two's complement.** This was one of the main reasons behind the **overflows** caused while adding few numbers. Nevertheless, there was an extra bit in the accumulator to store an overflow flag which could be later resolved by adding or subtracting 1 based on the type of the overflow. But, the fact that overflow flag cannot be saved or disabled branches to various other loopholes.

2. It could have been possible to **save, disable, or interrupt** the overflow flag if the makers would have added a **static register for this function.**

3. **Store instruction** can skip a word under certain circumstances like an overflow.

4. **CCS instruction** can skip several words altogether and this can be dangerous when the instruction following the ccs is a **prefix.** (For example: **EXTEND** is one of the most important prefixes which is capable of changing the interpretation or meaning of the opcode while encoding another set of instructions except the first few primary instructions).

5. There are **special memory cells (four cells)** allocated for **shift and rotate function** which can be operated only while writing into them. There could have been **instructions for shift and rotate** which could have saved some memory (which was too limited).

6. Most of the **Boolean instruction** worked only on the **I/O channels** except the **and instruction** (which worked between the memory and accumulator).

7. **INDEX prefix or INDEXING in general** is done by adding the value to IR register. This causes an interrupt in the normal flow of execution because the **altered IR value** does not match the value of memory block which is pointed by PC. (Remember, IR register contains the value (opcode) of the memory cell which is represented by PC) But, indexing causes addition of different values in IR register which causes an interrupt. However, this was resolved by **restoring the primary value of IR** which was stored in **IR' register.** This primary value is stored back in IR after the indexing, so the normal flow of execution is continued.

8. **Stack** isn't provided in the architecture because of which **indexing** is used as and when needed. Moreover, **indexing** is also done by using a set of instructions.

## 2.17. Citation:


- O'Brien, Frank. "The Executive and Interpreter." *The Apollo Guidance Computer: Architecture and Operation*, Springer-Praxis, 2010.
- "Digital Apollo: Human and Machine in Six Lunar Landings." *Digital Apollo: Human and Machine in Six Lunar Landings*, by David A. Mindell, The MIT Press, 2008.
- "Chapter 10 Next-Generation---Block 2 (Architecture)." *Journey to the Moon: the History of the Apollo Guidance Computer*, by Eldon C. Hall, American Institute of Aeronautics and Astronautics, 1996, pp. 117–128.
- Tomayko, James. "The Apollo Guidance Computer." *NASA*, NASA, www.history.nasa.gov/computers/Ch2-1.html.
- Britannica, The Editors of Encyclopaedia. "Apollo." *Encyclopædia Britannica*, Encyclopædia Britannica, Inc., 3 Aug. 2018, www.britannica.com/science/Apollo-space-program.
- Media.ccc.de, director. *The Ultimate Apollo Guidance Computer Talk. YouTube*, YouTube, 28 Dec. 2017, www.youtube.com/watch?v=xx7Lfh5SKUQ. From 5:50 to 28:10.