**Abstract**

This report serves as the fourth deliverable for the Platforms for Computing module and contains three parts to it: notes for the poster explaining how the main components and subsystems of a computer interact with each other, with particular emphasis on Von Neumann architecture, memory hierarchy, and Direct Memory Access; a tutorial for the Snake and Maze simulator, with screenshots and explanations of how it was built, optimized and how it works; a technical brief with full explained solutions to the Fetch-Decode-Execute Cycle, and CPU operations.

Reece alqotaibi | 19063069

4WCM0019-0105-2020 - Platforms for Computing (COM)

Assignment 1 Technical Report

2021

# Poster Notes

### Von Neumann Architecture

Designed by Mathematician and Physicist, Jon Von Neumann, in 1945 is the foundation of almost every modern-day computer architecture in use today. It is based on the concept of a stored-program computer.

**Architecture components**

There are three main components to the Von Neumann architecture.

1. A CPU, which is the ‘brain’ of the computer. This contains multiple high speed storage locations known as ‘Registers’. Any data being written to or read from memory must be stored in one of these before the data itself can be processed. All the processing and instructions that the computer performs are done by the CPU.

2. Main memory, which is one of the primary memory ranks within a computer. This stores the data of running programmes, each in their own memory location, and is accessible directly by the CPU.

3. Input/Output devices: these are the peripherals used to interact with the compute, such as; keyboard mouse, scanner; printer, monitor, et al, respectively.

[Author unknown n.d., *Von Neumann Architecture,* Computer Science GCSE Guru, viewed 01 March 2021, <https://www.computerscience.gcse.guru/theory/von-neumann-architecture>]

**Order of Operations**

The Fetch-Decode-Execute cycle, discussed in depth later, is the fundamental base for how the architecture works. At its core, the CPU has an in-built clock that ticks back and forth between each clock cycle. In each cycle, the CPU goes off the main memory with the memory address (location) that it needs to access, retrieves the value of that address, and then sends that to an output device.

[Author unknown n.d., *fetch-decode-execute cycle,* BBC Bitesize, viewed 01 March 2021, <https://www.bbc.co.uk/bitesize/guides/z2342hv/revision/5#:~:text=The%20main%20job%20of%20the,storage%20into%20the%20main%20memory>]

A high-level example of this is when click on a menu in Microsoft Word. When Word is opened, all the data for it is stored into main memory (RAM). If you click the ‘File’ menu, the CPU will go to the memory address(es) that point to the components of the menu, process them through the CPU registers, and then send the processed values to the output source (a monitor in this case).

**Advantages of the Von Neumann vs Harvard**

It is prudent to note that the Harvard architecture is the main alternative to Von Neumann, which is why this is the comparator.

1. Cheaper cost as does not require two separate platforms (Harvard uses an instruction memory platform, and a data memory platform)
2. Memory data is retrieved and written to from memory and devices on the same platform, and in the same way.
3. Especially true with lower-level programming languages, programmers have control over how they can reuse and organize memory.

Author unknown n.d., *Von-Neuman vs Harvard Architecture,* Teach Computer Science, viewed 01 March 2021, <https://teachcomputerscience.com/von-neumann-harvard-architecture/#Advantages_and_Disadvantages_of_von_Neumann_Architecture>]

### Memory Hierarchy

All computer memory is separated into classes, based on the response time of accessing that data, measured in nanoseconds (ns). The end goal is to share all the available memory in a computer system, to have large capacity, high speed and a low cost when accessing memory locations.

**Pyramid Tiers**

Registers – Static RAM (SRAM in the CPU). This is the fastest type memory available, and is present in every CPU today.

CPU Cache – Additional high speed, low-cost memory locations that are common in CPU’s. There are usually three levels, with each level containing less memory than the next, but also has a higher access speed. An Intel i7-9700K processor has 512KiB L1 cache, 2Mib L2 cache, and 12MB L3 cache.

[Author unknown, n.d., *WikiChip*, Core i7-9700k-Intel, <https://en.wikichip.org/wiki/intel/core_i7/i7-9700k>, viewed 3rd March 2021]

Main Memory – This memory unit is usually measured in GiB (Gibibytes) and is the main memory that is memory addresses and data are stored.

Auxiliary Memory – This is by far the cheapest memory in a computer system and is used to store persistent data. Hard drives are the most common type of auxiliary memory and are usually measured in GiB or TiB.

There are two types of memory in a computer system: volatile and non-volatile.

**Volatile Memory**

Split across the first three tiers in the pyramid, these are much higher in price than the fourth tier but come with them extremely fast access times. These types of memory are volatile, which means that the data stored in them will be lost when the system is powered off.

**Non-Volatile Memory**

All contained within the Auxiliary memory, devices such as hard drives, solid state drives, CD-ROMs, etc., are all considerably cheaper than the memory in the first three tiers. The benefit of these, however, is that data that is written to them is saved on them in a persistent state. When a computer is switched off and back on again later, the data is still available.

It is worth noting that due to the way memory hierarchy works, some of the data pertaining to non-volatile memory may be in stored in volatile memory at the time of the system being powered off may be being processed as part of an application or programme that is running. An unexpected loss of power does not give the volatile memory the chance to complete its processing and can lead to data corruption and data loss.

[Barbara Decares, 2018, *The Memory Hierarchy,* Medium.com*,* <https://medium.com/@barbaradecares/the-memory-hierarchy-f9c3016d7cba>, viewed 3rd March 2021)

**Characteristics**

1. High speed performance is the main design thought when utilizing memory hierarchy.
2. Having all the available memory to use in a system.
3. The amount of time it takes read and write to data in memory. When going down the tiers in the pyramid, the access time increases. The most dramatic speed increase is between main memory and auxiliary memory.

[Author unknown, n.d., *Elprocus,* Memory Hierarchy in Computer Architecture, <https://en.wikichip.org/wiki/intel/core_i7/i7-9700k>, viewed 3rd March 2021]

**Principle of Locality**

Memory hierarchy works using the Principle of Locality, which is the concept for CPUs to keep accessing the same memory locations is a short time scale. The two primary components to this are temporal locality and spatial locality. The former is the idea that a memory address recently accessed will likely be used again and so stores this location in the lowest cost area of memory. The latter is the idea that not just the location itself, but also of memory addresses around it will likely be accessed in a short timescale. It is common to guess a dynamic size of addresses close to the one currently being accessed and prepare them to be accessed too.

[William., Stallings (2010). *Computer organization and architecture: designing for performance (8th ed.),]*

### Direct Memory Access

Direct Memory Access provides a separate bus the computer system for peripherals to send and retrieve data to and from system memory, but without involving the CPU. The result of this is a high throughput of data.

The Direct Memory Access Controller is an integrated circuit that oversees the transfer of data between I/O devices and memory. A DMAC consists of several different channels, that can be used parallel to each other to allow multiple devices to read/transfer from memory.

[John Franco, n.d., *Introduction to Operating Systems,* University of Cincinnati, <http://gauss.ececs.uc.edu/Courses/c4029/lectures/dma.pdf>, viewed 4th March 2021]

**Modes**  
There are three modes in which DMA may use, and each one works differently: Burst Mode, Cycle Stealing Mode, and Transparent Mode.  
  
Burst Mode – The quickest mode and works by sending a full block of continuous data to the CPU. This allows all the device data to be sent at once but comes with the cost of locking the CPU functions until this has completed.

Cycle Stealing Mode – This is the second quickest mode of data transfer and works by using two system bus signals: Bus Request, which informs the system bus that a device is waiting to send or retrieve data; Bus Grant, where the CPU gives a bus to the device so that it can start DMA data transfers. This does not lock the CPU up for as long as Burst mode does.

Transparent Mode – The slowest of the three methods, but generally considered as the best approach for transferring data. The DMAC continuously waits for free cycles in the CPU operations where a system bus is not required. This is safer than the two other methods as it is the least likely to have a noticeable negative impact on system performance. The author states “*while the disadvantage is that the hardware needs to determine when the CPU is not using the system buses, which can be complicated.”*

[Author Unknown, n.d., *MiniTool,* What Is Direct Memory Access (DMA) and How Does It Work?, viewed 04th March 2021)

# Snake and Maze Tutorial

### Controlling the snake

The Snake and Maze solution uses a binary byte, later converted to hexadecimal to control how the snake moves through the maze. The byte is split into two parts, each containing four bits (also called a nibble). Binary is read from right to left.

The first nibble is used to control the distance in which the snake travels, as seen below:



We control the distance by assigning ‘1’ to each bit we want to use:

Movement nibble 1 1 1 1

Binary representation 8 4 2 1

Currently using only, the first nibble as above, the result of this (1111) in binary is 15, which when converted to hexadecimal (remembering that the snakes movement value must be a hex value) is **F**.

The second nibble is used to control the direction in which the snake moves, using the following image and table below:

|  |  |
| --- | --- |
| 1000 | Moves the snake up |
| 0100 | Moves the snake down |
| 0010 | Moves the snake left |
| 0001 | Moves the snake right |



Using what we know, we can combine both nibbles to form a binary byte, which will then be assigned as a hexadecimal value. For example, if we wanted to move the snake down at a movement distance of 15, we would end up with the binary result **01001111**, which becomes **4F** in hexadecimal.

4 F

0100 1111

### Writing the solution

The simulator uses Assembly language at its core, so we must provide instructions in assembly language for it to do what we want. There are only three types of instructions that we utilise in the first solution and these are:

|  |  |
| --- | --- |
| MOV AL, XX | This instruction moves the value specified (XX is the example placeholder of the hexadecimal conversion) into one of the CPU’s general-purpose registers and store it in memory. **Note: FF is used to reset the snake.** |
| OUT 04 | This instruction sends the value of the AL memory address to output 04, which is the Snake and Maze user interface |
| END | This instruction ends the programme. |

Using what we have learned about controlling the snake, we can start to begin programming the snake to traverse the maze it is in. Looking at the output, we can clearly see that the first thing the snake must do is move down by a distance of X.

Having learned that, to move down in binary we use 0100, we can then test a movement distance for the snake. Let us start with the maximum distance available, which is 15 (8 + 4 + 2 + 1) as we can always reduce this if the snake moves too far.

Now we have a complete binary byte, 01001111 (0100 to move down and 1111 to move a distance of 15) we need to convert this to hexadecimal. The result is **4F.**

In the source code we would write the following:

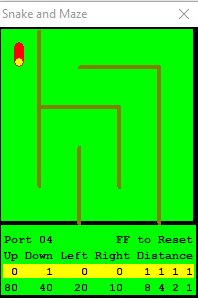
MOV AL,4F

OUT 04

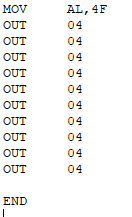
END

As explained earlier in this section, all this is telling the compiler to do is to move 4F into the AL, and then send to value of AL to the Snake and Maze interface.

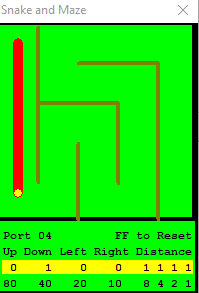
Running the solution will produce the following output:



As we can see from the output, the maximum distance value of 15 is fine to use, as the snake has not travelled too far. This means we can keep sending the ALs value to the output as many times as required which will eventually look like:



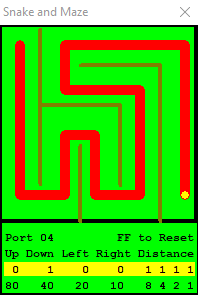
Resulting in:



Now we can see that the snake has moved down 10 times.

The process for traversing the maze is the same concept, with simple adjustments to the direction and distance as required, keeping the snake within the boundaries of the maze.

The output result for a fully completed source code will look like:



This is the output of my Snake1.ASM

This concludes the tutorial for the Snake and Maze.

### Optimisation

As part of Task 2 of the assignment, I used a loop to make the program more efficient.

The following is the loop I have used, and an explanation of what it is:



This loop is essentially a modern language while loop, and a high-level overview is:



Using the following code from the Snake2.ASM we can interpret the loop further:

**ASM Instruction:**

****

**Interpretation:**

****

This will keep on repeating itself for each declaration of MOV AL, MOV BL and CALL until it hits the end command.

### Observation and Review

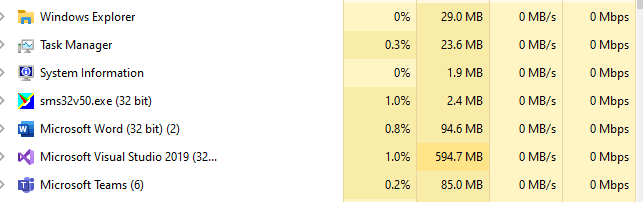
After both solutions had been created, I decided to do some testing on each one. The results, although not what I was expecting, made sense. I initially thought that the second solution would be more efficient on both the CPU and the memory, but this was not to be the case:  
  
**Test Environment:**

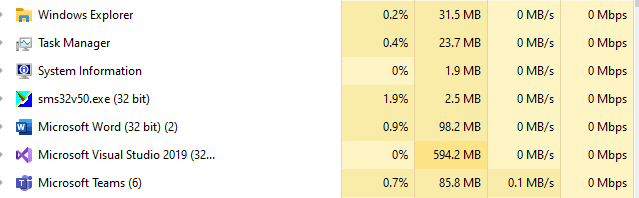
|  |  |
| --- | --- |
| **Processor** | x64 based Intel Core i7 |
| **Operating System (not relevant in test)** | Windows 10 |
| **Memory** | 8GB DDR4 SDRAM |
| **CPU Speed Increment** | 3 Clicks of the ‘Faster’ button (increments of x2) |
| **Distance Moved** | 15 Hexadecimal |

**Solution One – CPU**

Running the first solution under the normal CPU speed took 37.5 seconds to complete.

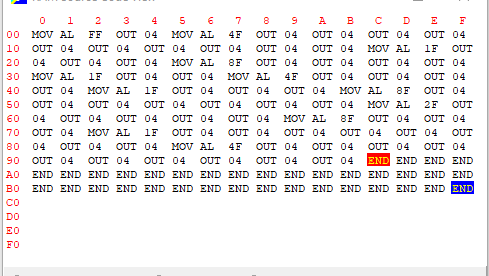
The CPU Usage fluctuation throughout but had an overall average of 1.0% utilization of the CPU.



Running the test again at 3x speed, I observed that the completion time was 6.02 seconds and the CPU usage averaged at 1.9%.  


**Solution One – Memory**

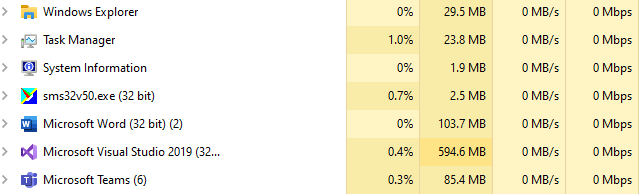
The memory usage should remain constant regardless of the CPU speed and tests show this to be true; the memory usage was over half of the memory available to the simulator.

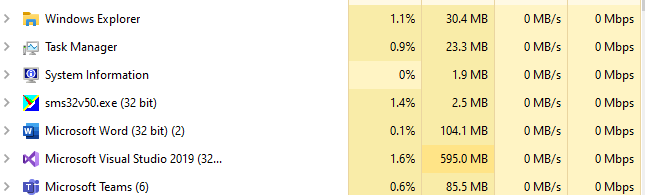


**Solution Two – CPU**

Running the second solution under the normal CPU speed took 134.2 seconds to complete.

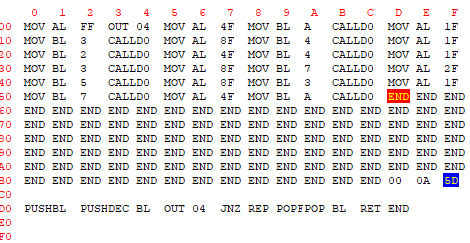
The CPU Usage fluctuation throughout but had an overall average of 0.7% utilization of the CPU.



Running the test again at 3x speed, I observed that the completion time was 22.3 seconds and the CPU usage averaged at 1.4%.  


**Solution Two – Memory**

This solution used almost less than half of the memory that the first solution used.



**Test Comparison**

Each solution had three tests performed on it for a fair analysis. The results above reflect the average over the three tests.

It is fair to say that ‘efficiency’ may not always be without drawbacks, and it is about finding a balance between the two. For example, results show that the second solution, whilst consuming less memory and utilizing less of the CPU, took substantially longer to run and complete. One can argue that if this were an application designed for a smooth user experience, the ‘optimization’ will not have been suitable to use in production.

**Simulator vs My/Modern CPU’s**

The simulator uses an Intel 8080, released in 1974. This is an 8-bit CPU with 1 core, 1 thread and a maximum clock rate of 3.125 Mhz. The simulator memory contained 256 bytes of RAM.

My own CPU is and Intel i7-4785T, with four cores, 8 threads and a clock rate of 2.20GHz. Clearly the difference between them is exponential and can perhaps explain why the application took so long to complete.

The i7 CPU in my machine has 6 ALU’s per core, so a total of 24 ALUs available in total. Considering this, it is almost certain that running the same programme on the i7 would have completed much, much quicker than the simulator.

Multi-core processing clearly is not a feature of the simulator CPU, so even with only 6 cores, it means that the i7 can perform all the arithmetic and logic procedures six times quicker than that of the 8080.

The available memory in my system is also exponentially more than the in the 8080 and far superior in terms of throughput, so it is clear that I would have also been able to store more addresses, and read/write to and from them quicker also.

# Fetch-Decode-Execute Cycle

### Brief

Fetch-Decode-Execute refers to the primary process that a CPU performs to execute programmes. This cycle begins as soon as a computer is switched on by loading the programme into the main memory (RAM)

### Terms

**Program Counter (PC)**

A register in a computer processor that contains the memory locations (memory addresses) of the next instruction being executed for a particular programme. When a programme is loaded, the first memory address is passed into the program counter, which in turn passes this to the **MAR**. The program counter then increments the value stored in its counter by one.

**Memory Address Register (MAR)**

This register contains the memory address of the current instruction that is to be fetched from memory and is connected only to the CPUs address bus - the MAR is the only way a CPU can communicate with this bus. The MAR contains two different types of addresses: the address of the current instruction; the address of a piece of data that is transferred to the MDR/MBR. Typically, the **MAR** works with the **MDR.** First the MAR sends the memory address across the address bus into memory, decodes the instruction in that address and then sends it down the data bus to the **MDR.**

**Memory Data Register (MDR)**

Also known as the Memory Buffer Register, it is comparable to the **MAR**, but uses the data bus instead of the address bus. Another difference is that the **MDR** can not only load data from the data bus, but also store it as well, both using the address stored in the **MAR**.

**Current Instruction Register (MDR)**

This register holds the instruction that is currently being executed or decoded by the CPU. The value held in here is obtained from the **MDR.**

**Arithmetic Logic Unit (ALU)**

This is where the CPU performs all its mathematical operations, hence the name ‘Arithmetic Logic’. By nature, the ALU does not always need to be used. For example, storing a value in memory does not explicitly perform a mathematical or logical process.

**Arithmetic**

Mathematical operations such as addition and subtraction, multiplication, and division.

**Logic**

Logical comparisons such as OR, AND, and XOR.

The ALU works with multiple circuits that do mathematical and bitwise operations on binary numbers and contains two inputs: the first is an operand, which is essentially the piece of date that is going the have the operations performed on; the second is the actual operation that is going to be performed on the operand.

In the ALU are signal receivers for each of the inputs. The calculations are performed when a signal is sent to one of these inputs.

**Accumulator (ACC)**

This is a register inside of the ALU that stores the intermediate results of the operations performed by the ALU.

### Assignment Task

Below is the tri-cycle task in which the three cycles for the Fetch-Decode-Execute Cycle are performed using the data provided.

**Cycle One**

1. The programme begins at the first memory address for it, and the PC initially contains the memory address for the next instruction which is going to be executed. In this case it is the address 67 (start address).
2. The PC then increments by one (68) and places the current value (address location 67) into the MAR.
3. As mentioned earlier, the MAR then sends the memory address stored in it across the address bus and finds the corresponding instruction in that location, which here is ‘LOAD 71’.
4. This is then returned to the MDR via the data bus.

ACC

CIR

MDR

**MDR <- [Memory] PC <-[PC]+1**

**MAR address;**

PC

MAR

68 [+1]

67

LOAD 71



1. The MDR then passes the instruction to the CIR.

ACC

CIR

MDR

**CIR <- [MDR]**

PC

MAR

68

67

LOAD 71

LOAD 71



1. The CIR then decodes and executes this instruction by loading the value of memory address 71 and placing it in the ACC.

ACC

CIR

MDR

**[CIR] Executes**

PC

MAR

68

67

LOAD 71

LOAD 71

92



**Cycle Two**

1. Still in the same programme, the cycle starts again, and the PC contains the memory address for the next instruction to be loaded (68).
2. The PC then increments by one (69) and places the current value (address location 68) into the MAR.
3. The MAR then sends the memory address stored in it across the address bus and finds the corresponding instruction in that location, which this time is ‘ADD 72’
4. This is then returned to the MDR via the data bus.

ACC

CIR

MDR

**MDR <- [Memory] PC <-[PC]+1**

**MAR address;**

PC

MAR

69

68

ADD 72

92



1. The MDR then passes the instruction to the CIR.

ACC

CIR

MDR

**CIR <- [MDR]**

PC

MAR

69

68

ADD 72

ADD 72

92



1. The CIR then decodes and executes this instruction by first loading the memory address, decoding it, and then executing the instruction. This time the instruction is sent to the ALU which then performs: ADD 5 to ACC (92), which becomes 97. This is then stored in the ACC.

ACC

CIR

MDR

**[CIR] Executes**

PC

MAR

69

68

ADD 72

ADD 72

97



**Cycle Three**

1. Again, in the same programme, the cycle starts again, and the PC contains the memory address for the next instruction to be loaded (69).
2. The PC then increments by one (70) and places the current value (address location 68) into the MAR.
3. The MAR then sends the memory address stored in it across the address bus and finds the corresponding instruction in that location, which this time is ‘STORE 72’.
4. This is then returned to the MDR via the data bus.

ACC

CIR

MDR

**MBR <- [Memory] PC <-[PC]+1**

**MAR address;**

PC

MAR

70

69

STORE 72

97



1. The MDR then passes the instruction to the CIR.

ACC

CIR

MDR

**CIR <- [MDR]**

PC

MAR

70

69

STORE 72

STORE 72

97



1. The CIR then decodes and executes this instruction by first loading the memory address, decoding it, and then executing the instruction. This time the instruction is ‘STORE in memory address 72’, so the CIR sends the value in the ACC back down the data bus to the MDR. The MDR then stores the value into memory address location 72.

ACC

CIR

MDR

**[CIR] Executes**

PC

MAR

70

69

STORE 72

STORE 72

97

