Department of Computer Science

Faculty of Engineering, Built Environment & IT

University of Pretoria

COS132 - Imperative Programming

Assignment Specifications

Release Date: 22-04-2025 at 06:00

FitchFork Submissions Open: 12-05-2025 at 06:00

Due Date: 23-05-2025 at 23:59

Total Marks: 265

Read the entire specification before starting with the assignment.

Contents

1 General Instructions

- ***Read the entire assignment thoroughly before you begin coding.***
- This assignment should be completed individually.
- **Every submission will be inspected with the help of dedicated plagiarism detection software.**
- If your code does not compile, you will be awarded a mark of 0. The output of your program will be primarily considered for marks, although internal structure may also be tested (eg. the presence/absence of certain functions or structure).
- Failure of your program to successfully exit will result in a mark of 0.
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated, and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at `https://portal.cs.up.ac.za/files/departmental-guide/`.
- Unless otherwise stated, the usage of C++11 or additional libraries outside of those indicated in the assignment, will **not** be allowed. Some of the appropriate files that you have submit will be overwritten during marking to ensure compliance to these requirements. **Please ensure you use C++98**
- All functions should be implemented in the corresponding `cpp` file. No inline implementation in the header file apart from the provided functions.
- The usage of ChatGPT and other AI-Related software to generate submitted code is strictly forbidden and will be considered as plagiarism.
- Note the FitchFork submission will open on the **12-05-2025** at **06:00**.

2 Overview

In this assignment, you will be utilising all of the concepts taught in COS132 to create a simplistic programming language and interpreter.

3 Background

This section contains the background for the assignment.

3.1 Computer Architecture

Computer architecture refers to the design and organisation of a computer's core components and how they interact to execute programs. It is a blueprint that defines:

- The structure of the **central processing unit (CPU)**, including the control unit, arithmetic logic unit (ALU), and registers.
- The **instruction set architecture (ISA)**, which specifies the set of instructions the CPU can understand and execute.
- The design of **memory hierarchy**, including caches, main memory (RAM), and storage.
- The **input/output (I/O)** mechanisms and how data is transferred between the CPU and peripheral devices.
- The **data paths** and **control signals** that coordinate data flow and operations within the system.

In essence, computer architecture defines how hardware and software interact to perform computation efficiently.

3.2 Opcodes and Operands

In computer architecture, an **instruction** is a single operation that a processor can execute. Each instruction typically consists of two main parts:

- **Opcode** (short for *operation code*): This specifies the operation to be performed. For example, it could be an addition, subtraction, load, or store operation.
- **Operands**: These are the values or the addresses of the values that the operation will work on. Operands can be constants, registers, or memory addresses.

For example, in the instruction:

```
ADD R1, R2, R3
```

- The `ADD` is the **opcode**, indicating an addition operation.
- `R1`, `R2`, and `R3` are the **operands**. This instruction adds the contents of `R2` and `R3`, and stores the result in `R1`.

It is also possible to store **opcodes** as a number, for example, depending on the computer's architecture, `ADD` can be stored as 3.

3.3 Assignment Specific Details

This section details how the computer architecture, operands, and opcodes are used in this assignment specifically.

3.3.1 Computer Architecture

The basic computer/interpreter you will implement will have:

- Random Access Memory (RAM).
- Storage (which is persistent storage[1] like a hard drive),
- Terminal input and output.
- Operating system (which, amongst other operations, will perform the same functionality as the CPU in a traditional computer architecture).
- A program file that is written in binary, where each opcode and operand is 4 bits. Each instruction will be on a new line

The RAM and storage will be two memory buffer objects (a memory buffer object is a fancy name for an array with additional functionalities). The storage will be able to be saved to the hard drive (which represents saving it to a text file), and loaded from the hard drive (which represents loading the content of the text file into the memory buffer object). This means that the contents of the storage memory buffer object are only written to the "hard drive" once a certain command is called. This also means that if the command is not given to load the "hard drive", a blank hard drive will be used by the computer.

3.3.2 Opcodes and Operands

For this assignment, the following opcodes and operands are defined in Table 1.

Operation	**Opcode**	**Operands**
Exit	0	
Load	1	MemoryLocation, RAMLocation
Save	2	RAMLocation, MemoryLocation
Add	3	ResultLocation, InputA, InputB
Minus	4	ResultLocation, InputA, InputB
Greater	5	ResultLocation, InputA, InputB
Less	6	ResultLocation, InputA, InputB
Equal	7	ResultLocation, InputA, InputB
If	8	BooleanLocation, TrueProgramLine, FalseProgramLine
Input	9	RAMLocation
Output	10	RAMLocation
Goto	11	ProgramLine
Persist	12	
Reload	13	
Const	14	ConstantValue, RAMLocation
Move	15	SourceLocation, DestinationLocation

Table 1: Instruction Set Table

Table 2 contains a brief description of the operations, which will be expanded on later in the specification.

[1]Persistent storage means that the data will be available after the computer has been restarted, i.e., it is not in RAM.

Operation	Description
Exit	Terminates the program.
Load	Loads data from a memory location into RAM.
Save	Saves data from RAM into a memory location.
Add	Adds the values stored at InputA and InputB together, and stores the result.
Minus	Subtracts the value stored at InputB from the value stored at InputA, and stores the result.
Greater	Stores 1 if the value stored at InputA is greater that the value stored at InputB, else stores 0.
Less	Stores 1 if the value stored at InputA is less than the value stored at InputB, else stores 0.
Equal	Stores 1 if the value stored at InputA is equal to the value stored at InputB, else stores 0.
If	Conditional jump to TrueProgramLine or FalseProgramLine based on the value stored at BooleanLocation.
Input	Receives user input and stores it in the given RAM location.
Output	Outputs the value from the specified RAM location.
Goto	Jumps unconditionally to a specified program line.
Persist	Writes the data that is stored in the storage memory buffer object to the predefined text file.
Reload	Reload the data that is stored in the predefined text file into the storage memory buffer object.
Const	Stores a constant value in RAM.
Move	Copies value from SourceLocation to DestinationLocation.

Table 2: Instruction Descriptions

3.4 Understanding `*&` in C++

In C++, a function parameter declared as `*&` is a **reference to a pointer**. This means:

- The function receives an alias to an existing pointer.
- Changes to the pointer itself (not just the value it points to) will be visible outside the function.

3.4.1 Function Declaration Example

To illustrate this point, consider the following example:

```
void updatePointer(int*& ptr) {
    ptr = new int(42);
}
```

- `ptr` is a reference to a pointer to an `int`.
- The function assigns a new address to the pointer.
- Because `ptr` is a reference, the original pointer (outside the function) now points to the new memory location.

We can then use the `updatePointer` function as follows:

```
int main(){                                  1
    int* p = NULL;                           2
    cout << p << endl; //prints out 0        3
    updatePointer(p);                        4
    cout << p << endl; //prints out an address   5
    cout << *p << endl; //prints out 42      6
}                                            7
```

In this example, we can see that `updatePointer` was able to populate the variable `p`, which was originally `NULL`, with a new memory location.

3.4.2 Memory Layout Illustration

Using the two code examples in the previous section, we can draw the following illustrations:

Figure 1 illustrates the state of variable `p` as it stands on line 2 of the example `main` program.

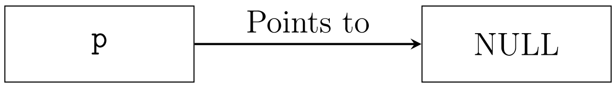

Figure 1: Line 2 of the example `main` function

Figure 2 shows the state of variable `p` after the `updatePointer` function call.

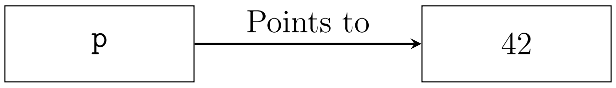

Figure 2: State of variable `p` on line 4.

So effectively, what is happening is illustrated in Figure 3 and 4. Figure 3 shows that both `p` and `ptr` initially point to the `NULL` memory address due to `ptr` being an alias (reference) for variable `p` (This is seen on line 1 of the `updatePointer` function). When `ptr` is then assigned a new memory location (after line 2 of the `updatePointer` function is executed), this change is propagated back to `p`. This is seen in Figure 4. This technique is useful if a function needs to change the actual address a variable is pointing to, and can be used for construction and destruction operations (discussed later in the specification).

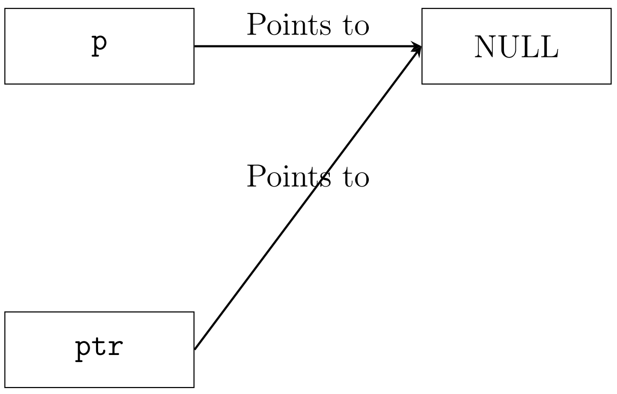

Figure 3: Initial state of `ptr` and `p` when the `updatePointer` function is called, i.e., line 1 of `updatePointer`.

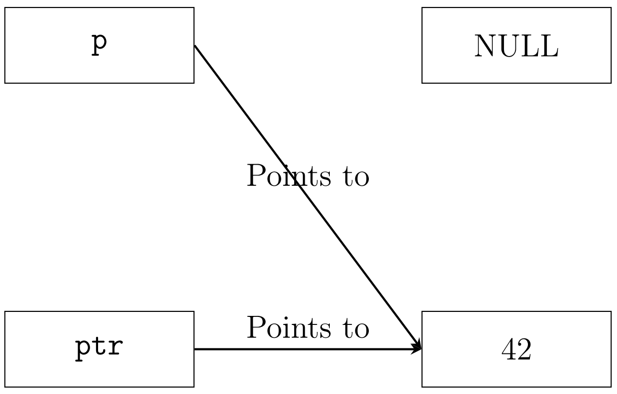

Figure 4: Final state of `ptr` and `p` when the `updatePointer` function just before the function terminates, i.e., line 3 of `updatePointer`.

3.4.3 Key Takeaway

Using `int*&` allows you to:

- Change where a pointer points.
- Allocate new memory for a pointer inside a function and have the caller see the change.

4 Your Task:

You are required to implement all of the functions laid out in this section. The functions you are required to implement are contained in the following files:

- `IO` - which contains any terminal input and output operations.
- `Memory` - which contains all the operations that are related to the memory buffer objects.
- `Instructions` - which contains all the operations that are related to the instructions discussed in Tables 1 and 2.

- `OS` - which contains all the operations that are related to the computer/operating system.

Reminder, if you want to access a variable or function in a namespace, use `NamespaceName::` followed by the variable/function you want to access.

Task	Mark
IO	25
Memory	44
InstrUtils	20
Instructions	26
ExeInstructions	48
OS	25
ExecuteProgram	33
Testing	29
Program Task (Will open at a later stage)	15

4.1 IO

4.1.1 Functions

- `strToInt`
  - This function needs to convert the passed-in string to an int.
- `intToBin`
  - This function needs to convert the passed in `value` to a binary string.
  - If the binary string is longer than the passed in `size`, drop the leftmost bit until the string is of the correct length.
  - If the binary string is shorter than the passed in `size`, left pad the string with “0” until the string is of the correct length.
  - Example:
    - If the string is `1100010` and the size is 3, the result should be: `010`.
    - If the string is `01` and the size is 5, the result should be: `00001`
- `writeToFile`
  - This function needs to write the contents of the passed in array to the text file with the passed-in name.
  - If the array is null or the size of the array is not valid, the function should not modify the text file.

- `readFromFile`
  - This function needs to populate the passed-in array with the contents of the text file with the passed-in name.
  - If the array is null or the size of the array is not valid, the function should terminate.
  - If the text file contains more data than there is space for in the array, the surplus data should be ignored.
  - If the text file does not contain enough data, the indexes in the array that do not contain data should be filled with 0.
- `printError`
  - This function needs to print a specific error followed by a newline.
  - The error to print is represented by the passed-in `errorCode`.
  - Table 3 represents the error that corresponds to the specific code.

Error Code	**Error Message**
0	Error: Buffer Is Null
2	Error: Segfault
3	Error: Invalid buffer size
4	Error: Unknown Instruction

Table 3: Mapping of Error Codes to Error Messages

  - Note: 1 was omitted on purpose.
  - If an invalid `errorCode` is passed in, the function should not print out anything.
- `obtainInput`
  - This function is used to ask the user for input and return the input.
  - The function should print out: “Please enter an input:” followed by a newline.
  - The function should then obtain the integer input from the user and return the input.
- `printOut`
  - This function is used to print out the passed-in parameter.
  - The printout should be structured as follows:

```
Printout: <<value>>$
```

    Where
    * `«value»` represents the value being printed out
    * `$` represents a newline.

4.2 Memory

The `Memory` files contain a `MemoryBuffer` namespace, which contains the majority of the functionality, but it also contains two standalone variables.

4.2.1 MemoryBuffer Namespace

The `MemoryBuffer` namespace contains a struct which is used to represent a `MemoryBufferObject` and a series of functions that can use a `MemoryBufferObject`.

4.2.1.1 MemoryBufferObject

The `MemoryBufferObject` struct is used to encapsulate all the information for a `MemoryBufferObject` together. It has the following variables:

- `buffer`
  - It is a 1D dynamic array containing integers which represent the memory cells.
- `bufferSize`
  - This determines the size of the `buffer`.

4.2.1.2 Functions

This section discusses the functions used in the `MemoryBuffer` namespace.

- `printOutContents`
  - This function is used to print out the contents of the passed-in parameter's `buffer`.
  - If the passed-in parameter is `NULL` or the `buffer` of the passed-in parameter is `NULL`, the function should print an error, using an error code of 0[2].
  - The format the output for each item in the buffer should be:

    ```
    [i]: <<value_i>>$
    ```

    where:
    - `i` is the index of the item in the buffer.
    - `«value_i»` is the value of the i^{th} index in the buffer.
    - $ is a new line.

[2] *Hint: this requires the use of a function discussed earlier in the specification.*

  – Example:
    If the buffer contains the following data:

5
4
2
1

    The printout should be:

```
[0]: 5
[1]: 4
[2]: 2
[3]: 1
```

- `dereference`
  – This function should return a pointer to the memory address of the item in the buffer at the index specified by the passed-in `memoryAddress`.
  – If the passed-in `memoryBufferObject` is `NULL` or the `buffer` of the passed-in `memoryBufferObject` is `NULL` print out an error using an error code of 0, and return `NULL`.
  – If the passed in `memoryAddress` is invalid (i.e., outside the scope of the buffer), the function should print out an error using an error code of 2, and return `NULL`.
  – Example:
    If the buffer contains the following data:

5
4
2
1

    A function call like: `dereference(obj, 1)` should return a pointer pointing to the memory location of 4. A function call like: `dereference(obj, 1000)` will print out an error and return `NULL`.

- `store`
  - This function needs to store the passed-in `value` as the index specified by the passed-in `memoryAddress` in the passed-in `memoryBufferObject`'s `buffer`.
  - If there is already a value in that index in the `buffer`, the old value should be overwritten with the passed-in `value`.
  - If the passed-in `memoryBufferObject` is `NULL` or the `buffer` of the passed-in `memoryBufferObject` is `NULL`, print out an error using an error code of 0, and exit out of the function.
  - If the passed in `memoryAddress` is invalid (i.e., outside the scope of the buffer), the function should print out an error using an error code of 2, and exit out of the function.
  - Example: a function call like: `store(obj, 2, 10)` will store the value of 10 at the 2^{nd} index in the `buffer` of `obj`.
- `read`
  - This function needs to return the value stored at the index of the passed-in `memoryBufferObject`'s `buffer`.
  - The index is specified by the passed-in `memoryAddress`.
  - If the passed-in `memoryBufferObject` is `NULL` or the `buffer` of the passed-in `memoryBufferObject` is `NULL`, print out an error using an error code of 0, and return `0`.
  - If the passed in `memoryAddress` is invalid (i.e., outside the scope of the buffer), the function should print out an error using an error code of 2, and return `0`.
- `createBuffer`
  - This function is used to create a new `MemoryBufferObject` in the passed-in reference to a pointer of type `MemoryBufferObject`.
  - *Hint: See Section 3.4.*
  - The function also needs to initialise the newly created `MemoryBufferObject`'s `buffer` to be of passed-in `bufferSize` size.
  - If the passed-in `bufferSize` is negative, the function should print an error using error code 3, and return without creating or modifying the passed-in `memoryBufferObject`.
  - If the passed-in `memoryBufferObject` has already been created (i.e., is not `NULL`), the function should first destroy the existing buffer then create a new one.
  - *Hint: Remember to populate all of the members of the newly created `MemoryBufferObject`.*
- `destroyBuffer`
  - This function needs to deallocate all the memory allocated to the passed-in `memoryBufferObject`.
  - If the passed-in `memoryBufferObject` is `NULL`, the function should simply return.
  - After all the memory has been deallocated, the passed-in `memoryBufferObject` should be set to `NULL`.

4.2.2 Standalone Variables

This file also contains two standalone variables, representing the assignment's RAM and storage.

4.3 Instructions

This namespace contains a struct named `Instruction`, functions and a variable which is discussed below.

4.3.1 Instruction

This struct is used to store the opcode and operands of an instruction. The struct has the following members:

- `opcode`
  - Which is a decimal representation of the opcode
- `operands`
  - Which is a 1D dynamic array which contains the decimal representations of the values of the operands.

4.3.2 Functions

The following functions need to be implemented. *Hint: for interacting with `ram` and `storage`, use the functions in the `MemoryBuffer` namespace.*

- `binaryToInt`
  - This function needs to convert an arbitrary-length binary string to the correct decimal/integer representation.
  - It can be assumed that an unsigned binary representation is used, and that only valid binary strings are passed in.
- `determineNumberOfOperands`
  - This function needs to determine and return the number of operands the passed-in `opcode` has using Table 1.
  - If an unknown `opcode` is passed-in, the function should return 0.
- `createInstruction`
  - This function needs to convert the passed-in `strInstruction` to an `Instruction` object, and return the newly created `Instruction`.
  - The passed-in `strInstruction` is a binary string.
  - The first four bits in the passed-in `strInstruction` specify the opcode.
  - The remaining bits specify the operands, where each four bits specify a single operand.

  - *Hint: remember to correctly populate the newly created `Instruction` object.*

- `destroyInstruction`
  - This function needs to deallocate all memory allocated to the passed-in `instruction`.
  - If the passed-in `instruction` is `NULL`, the function should just return and not deallocate any memory.
  - Lastly, the function should set the passed-in `instruction` to `NULL` after deallocation.

- `exitOp`
  - This function needs to set the `halt` variable to true.

- `loadOp`
  - This function needs to **read** the value stored at the index specified by `memoryLocation` in `storage`, and **store** it at the index specified by `RAMLocation` in `ram`.

- `saveOp`
  - This function needs to **read** the value stored at the index specified by `RAMLocation` in `ram`, and **store** it at the index specified by `memoryLocation` in `storage`.

- `addOp, minusOp, greateOp, lessOp, equalOp`
  - These functions need to perform their own specific functionality (as discussed in Table 2) as follows.
  - **Read** the value stored at the index specified by `inputA` in `ram`.
  - **Read** the value stored at the index specified by `inputB` in `ram`.
  - Perform the operation and **store** the result at the index specified by `resultLocation` in `ram`.

- `ifOp`
  - This function mimics an if-selection statement.
  - If the value that is **read** from the `booleanLocation` in `ram` is 0 (false), set the passed-in `currentInstructionNumber` to the passed-in `falseInstructionNumber`.
  - If the value that is **read** from the `booleanLocation` in `ram` is not 0 (i.e, true), set the passed-in `currentInstructionNumber` to the passed-in `trueInstructionNumber`.
  - Essentially, what is happening is that the `currentInstructionNumber` is the current instruction's index in the loaded program. If the condition is true, the program will "jump" to the index specified by `trueInstructionNumber`; else it will "jump" to the index specified by `falseInstructionNumber`.

- `inputOp`
  - Using the appropriate function in the `IO` file, obtain input from the user and **store** it in the index specified by `RAMLocation` in `ram`.
- `outputOp`
  - Using the appropriate function in the `IO` file, print out the value stored at the index specified by `RAMLocation` in `ram`.
- `goto`
  - This function is used to unconditionally "jump" around the program.
  - This is done by assigning the passed-in `targetInstructionNumber` to the passed-in `currentInstructionNumber`.
- `persistOp`
  - This function needs to save the contents of the `storage` to a text file named `memory.txt`.
  - Again, use an appropriate function from the `IO` file.
- `reloadOp`
  - This function needs to load the contents of the text file named `memory.txt` into `storage`.
  - Again, use an appropriate function from the `IO` file.
- `constOp`
  - This function is used to **store** a constant value specified by `constValue` into `ram` at the index specified by `RAMLocation`.
- `moveOp`
  - This function is used to make a copy of the value **read** from the index specified by `sourceLocation` in `ram`.
  - This copy of the value then needs to be **stored** at the index specified by `destinationLocation` in `ram`.
- `debugPrintout`
  - This function is used to help you debug your programs.
  - This function needs to print out the instruction as follows:

    ```
    [<<opcode>>]: <<operands>>$
    ```

    where:
    - «`opcode`» specifies the opcode of the passed in `instruction`

    * `«operands»` specifies the operands of the passed in `instructions` as a space deliminted list.
    * `$` indicates a newline.
  - If the passed-in `instruction` is `NULL`, the function needs to printout:

    ```
    NULL Instruction$
    ```

    where:

    * `$` indicates a newline

    and return.
  - Example:
    * Consider the following instruction:

      ```
      111000000010
      ```

    * The function should print out:

      ```
      [14]: 0 2
      ```

- `executeInstruction`
  - This function needs to invoke the appropriate instruction's function based on the `opcode` of the passed-in `instruction`.
  - The function should also pass any of the required parameters to the instruction's function.
  - The `currentInstructionNumber` parameter dictates the current instruction number the program is currently at.
  - If the passed-in `instruction` or the `operands` of the passed-in `instruction` is `null`, the function should simply return without invoking any instruction's function.
  - Example:
    * If the instruction with an opcode of 1, and operands of 0, and 5 is passed into the `executeInstruction` function, the function should call the `load` function, passing 0, and 5 as parameters to the `load` function.
  - If an instruction with an invalid opcode is passed into the function, the function should print an error with an error code of 4.

4.3.3 Variable

The sole variable of the `Instructions` namespace is the `halt` variable, which indicates that the program should halt execution.

4.4 OS

This function contains the `OperatingSystem` namespace, which includes a series of variables and functions.

4.4.1 Variables

- `programInstructions`
  - This is a dynamic 1D array of dynamic `Instruction` objects.
  - This variable simulates the “program” that the computer/operating system is going to execute.
- `numberOfInstructions`
  - This variable stores the number of instructions that the program consists of.
- `currentInstructionNumber`
  - This variable stores the index of the current instruction that is being executed by the program.

4.4.2 Functions

- `bootComputer`
  - This function is used to setup the computer such that programs can be run.
  - This function needs to initialise the `ram` and `storage` `MemoryBufferObjects` using the passed-in parameters.
  - *Hint: use the appropriate function in the `MemoryBuffer` namespace.*
  - After initialising the `ram` and `storage`, ensure that each index in both `buffer`s are set to 0.
  - Lastly, initialise the `currentInstructionNumber` to 0.
- `shutDownComputer`
  - This function needs to deallocate any memory allocated to the `ram`, `storage`, and `programInstructions`.
  - *Hint: remember to use the appropriate destruction function.*
  - Lastly, set the `programInstructions` to `NULL`.

- `loadHardDrive`
  - This function is used to load the contents of the hard drive (specified by the text file named `hardDriveDiskName`) into `storage`.
  - *Hint: use the function in `IO` file.*
  - If the `storage` is `NULL` or the `storage`'s `buffer` is `NULL`, the function should simply return without reading the file.
- `determineNumberOfInstructions`
  - This function needs to determine the number of instructions that are in the file with the name specified by `programFile`.
  - The function needs to return this count.
  - Each instruction is specified in binary on a separate line.
- `loadInstructions`
  - This function needs to load the instructions from the file specified by `programFile`, into the `programInstruction`.
  - Each instruction is specified in binary on a separate line.
  - Remember to also set the `numberOfInstructions`.
- `executeProgram`
  - This function is used to simulate the program being executed.
  - This function is represented by the following pseudo code:

Algorithm 1 Pseudocode for `executeProgram`

```
while currentInstructionNumber is less than the numberOfInstructions AND halt is false do
    executes the instruction specified by currentInstructionNumber
    if currentInstructionNumber was not changed by currentInstructionNumber then
        increment currentInstructionNumber by 1.
    end if
end while
```

- `printProgram`
  - This function is used to debug your program.
  - This function needs to iterate through all the `Instruction` objects in `programInstructions` and using the `debugPrintout` function print out the instruction.

- `translateProgram`
  - This function acts like a "compiler" in the sense that it takes a "high-level" program and converts it to a "low-level" program.
  - The "high-level" program can be found in the text file with the name `highLevelFile`.
  - The "low-level" will be stored in a text file with the name `lowLevelFile`.
  - The "high-level" program is structured as each instruction is on a separate line.
  - Each instruction is broken up into a comma-delimited list of integers, and the first number indicates the opcode, and the remaining numbers represent the operands.
  - The function then needs to convert each instruction to its equivalent binary representation and then save it in the "low-level" file.
  - Example: Consider the following "high-level" file:

    ```
    9,0
    9,1
    3,2,0,1
    10,2
    ```

    The function should then convert it to the following file:

    ```
    10010000
    10010001
    0011001000000001
    10100010
    ```

  - *Hint: remember each instruction is composed of groupings of four bits.*

5 Program Task

Apart from the above functions you need to implement you also need to implement a program using the instructions specified in Table 1. The program needs to be functionally equivalent to the following C++ program:

```cpp
int main(){
    int a,b;
    cin >> a;
    cin >> b;
    cout << a % b << endl;
}
```

Store the code in a file called: `programTask.txt`. The program will be verified using instructor-supplied files.

6 Memory Management

As this practical has a heavy memory management focus, specialised tools will be used to evaluate you on your memory management skills.

The tool that will be used is `valgrind`. To determine the effectiveness of your memory management skills, the following command can be used:

```
valgrind --leak-check=full --keep-stacktraces=alloc-and-free ./main
```

This command assumes that the code was compiled to an executable `main`, and that the debugging flag was used during compilation.

7 Testing

As testing is a vital skill that all software developers need to know and be able to perform daily, 10% of the assignment marks will be allocated to your testing skills. To do this, you will need to submit a testing main (inside the `main.cpp` file) that will be used to test an Instructor Provided solution. You may add any helper functions to the main.cpp file to aid your testing. In order to determine the coverage of your testing the gcov [3] tool, specifically the following version *gcov (Debian 8.3.0-6) 8.3.0*, will be used. The following set of commands will be used to run gcov:

```
g++ --coverage *.cpp -o main
./main
gcov -f -m -r -j ${files}
```

This will generate output which we will use to determine your testing coverage. The following coverage ratio will be used:

$$\frac{\text{number of lines executed}}{\text{number of source code lines}}$$

We will scale this ration based on class size.

The mark you will receive for the testing coverage task is determined using Table 4:

Coverage ratio range	% of testing mark
0%-5%	0%
5%-20%	20%
20%-40%	40%
40%-60%	60%
60%-80%	80%
80%-100%	100%

Table 4: Mark assignment for testing

[3]For more information on gcov please see `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

Note the top boundary for the Coverage ratio range is not inclusive except for 100%. Also, note that only the functions stipulated in this specification will be considered to determine your testing mark. Remember that your main will be testing the Instructor Provided code and as such, it can only be assumed that the functions outlined in this specification are defined and implemented.

As you will be receiving marks for your testing main, we will also be doing plagiarism checks on your testing main.

8 Implementation Details

- You must implement the functions in the header files exactly as stipulated in this specification. Failure to do so will result in compilation errors on FitchFork.
- You may only use **C++98**.
- You may only utilise the specified libraries. Failure to do so will result in compilation errors on FitchFork.
- Do not include using `namespace std` in any of the files.
- You may only use the following libraries:
  - `<string>`
  - `<fstream>`
  - `<iostream>`
  - `<sstream>`
  - `<cmath>`
- You are supplied with a **trivial** main demonstrating the basic functionality of this assessment.

9 Upload Checklist

The following C++ files should be in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number:

- `Instructions.cpp`
- `IO.cpp`
- `Memory.cpp`
- `OS.cpp`
- `main.cpp`
- Any textfiles used by your `main.cpp`

- `programTask.txt`

The files should be in the root directory of your zip file. In other words, when you open your zip file you should immediately see your files. They should not be inside another folder.

10 Submission

You need to submit your source files on the FitchFork website (`https://ff.cs.up.ac.za/`). All methods need to be implemented (or at least stubbed) before submission. Your code should be able to be compiled with the following command:

```
g++ -std=C++98 -g *.cpp -o main
```

and run with the following command:

```
./main
```

Remember your `h` file will be overwritten, so ensure you do not alter the provided `h` files.
You have 5 submissions and your final submission's mark will be your final mark. Upload your archive to the Assignment slot on the FitchFork website. Submit your work before the deadline. **No late submissions will be accepted!**

11 Change Log

Date	**Change**
22/04/2025 @ 08:09	Due date changed to 23/05/2025
26/04/2025 @ 18:31	Added description for the `executeInstruction` function.
12/05/2025 @ 00:10	Added the mark breakdown and a section of memory management