**Simulator**

The main job of the our software simulator is to simulate the SIMP processor and execute the fetch-decode-execute lo­­op of the binary code found in memin.txt, and update the memory and hardware linked to it accordingly.

The simulator receives 3 in­­­put files:

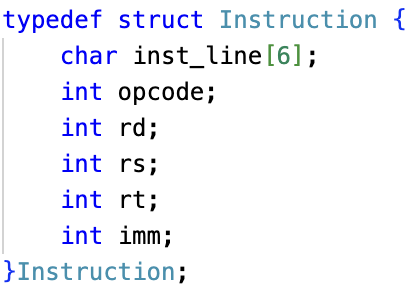
Memin.txt, diskin.txt, irq2in.txt.

The output of the simulator is:

Memout.txt, regout.txt, trace.txt, hwregtrace.txt, cycles.txt, leds.txt, display7seg.txt, diskout.txt, monitor.txt.

The role of every file is specified in the project assignment and thus not repeated in the documentation.

The core of the simulator functionality lies the Instruction struct as defined in the code:



This struct holds the ISA instruction line in hexadecimal as it is read from memin.txt, as well as the rest of the fields that comprise each instruction. Note that not each line in memin.txt is an instruction, but rather can be an immediate value. The way we handle it will be elaborated further in the documentation. Moreover, the struct also holds a condition in it that checks if there is a ‘1’ in the MSB of the immediate value, and treats it as a negative number.

**Definitions and globals**

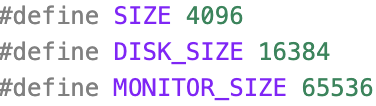
Now we will present the definitions and variables that may be in use in various places in the program and thus we want them to be accessible at all time, rather than being passed as arguments between functions.

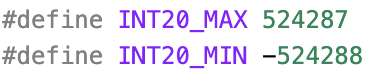
We will define SIZE 4096 as it is used as the maximum size of the main memory.

DISK\_SIZE equals 16384 that stand for 128 sectors times 128 lines that are in the hard disk.

MONITOR\_SIZE equals 65536 to represent it’s 256\*256 pixels.

In addition to that we define INT20\_MAX and INT20\_MIN that will help us detect overflow in int20 system (bit position 19 is the msb, and indicates that a number is negative in 2’s comp).





Next we declare:

* 

Which will be used to hold all of the 5 hexa digit (and one null character) lines that are fetched from memin.txt.

* 

This variable holds the number of instruction lines in practice that were passed from the assembler in memin.txt.

* 

This variable holds the current point counter value throughout the execution of the program.

* 

This array holds the values of all the registers with whom we perform our instructions.

* 

This array holds the values of the Input\Output registers as they are defined in the assignment. There are 23 registers, with max size of 32 bits represented by up to 8 hexadecimal digits.

* 

This variable is checked each cycle to determine if there is an interrupt.

* 

This variable indicates whether the program is ready to receive an interrupt.

* 

This variable indicates if we are in irq1.

* 

Since reading\writing operation take 1024 cycles to take place after the command was given, we would like to hold the command value that was in IOregister[14] at the time of the command until it actually takes place. Changing this variable in the simulator can be only done when the hard disk is free.

* 

This variable will be used to as counter of clock cycles (up to 1024) while we read or write to the hard disk.

* 

This array holds the clock cycle values in which we expect to get an irq2 interrupt. Those values are obtained from irq2in.txt file that is served as an input to the program.

* 

This variable will hold the index of the last irq2 interrupt that has occurred throughout the program. The index is used to access irq2\_interrupt\_pc array above.

* 

This array holds the values of the monitor pixels, total number of 256\*256 = 65536 pixels, with binary value ranges between 0-255 that is represented by 8 bits.

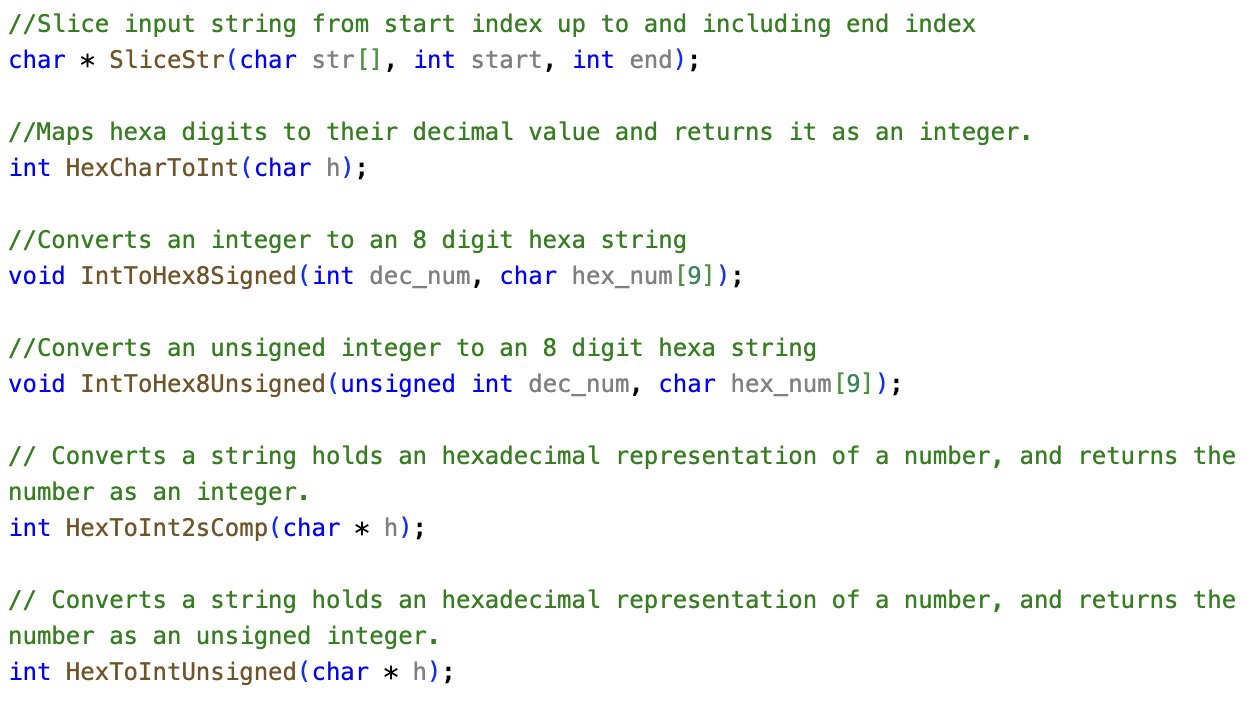
* 

The hard disk can contain 16384 lines in total (128 sectors times 128 lines in each), each hold 20 bits in the form of 5 hexa digits. One more place is reserved for the null character.

**Functions**

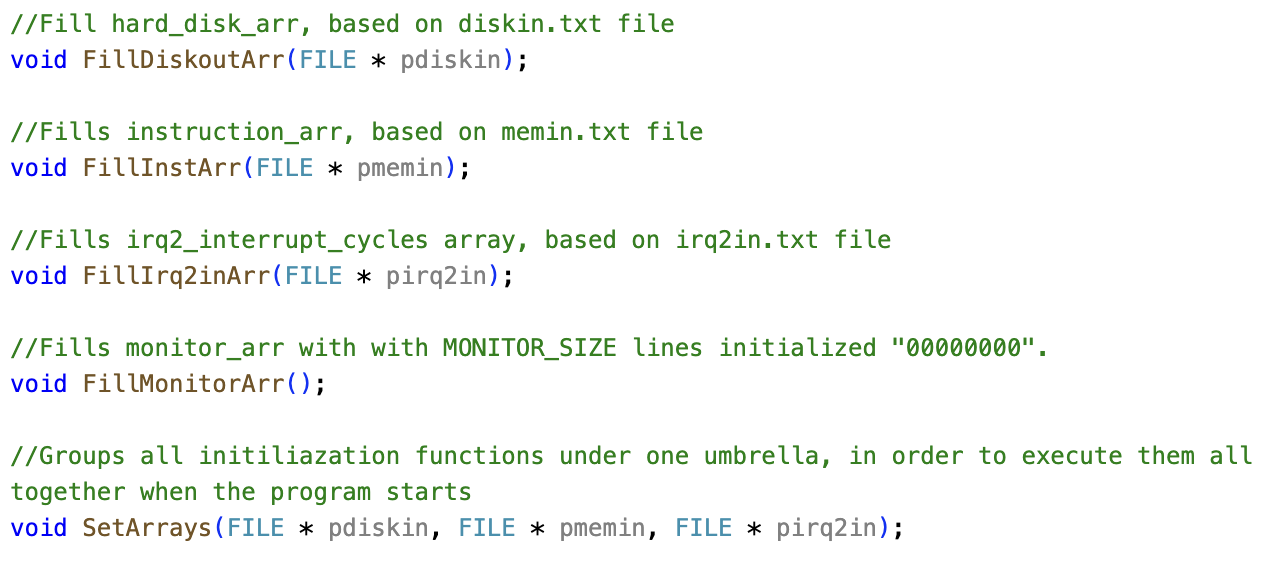
Auxiliary Functions

In the simulator we will work with define several auxiliary functions that will help us handle format and type conversions and adaptions. For example, converting a string of 8 hexadecimal digits into it’s integer value in 2’s complement, slicing strings etc.



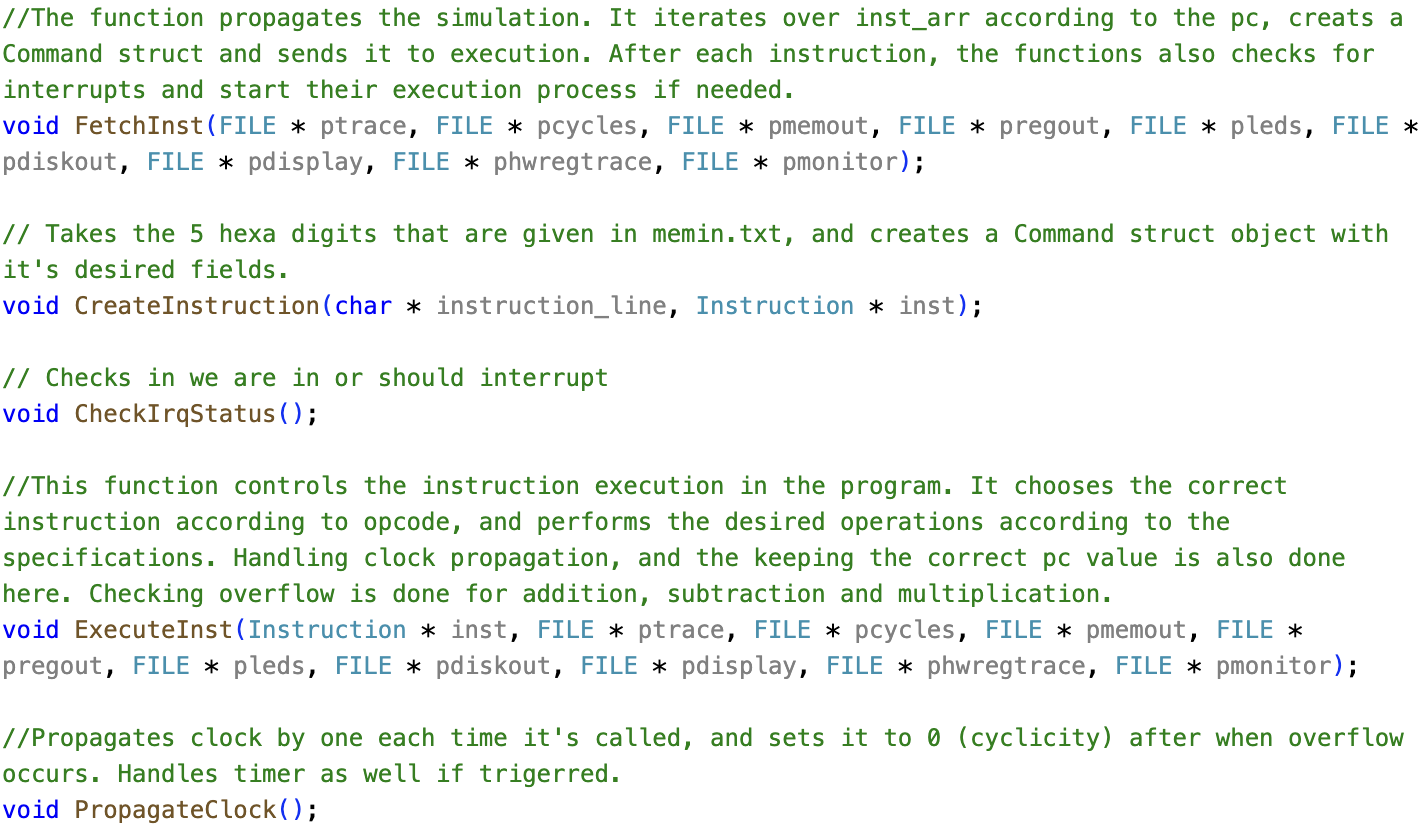
Setup functions

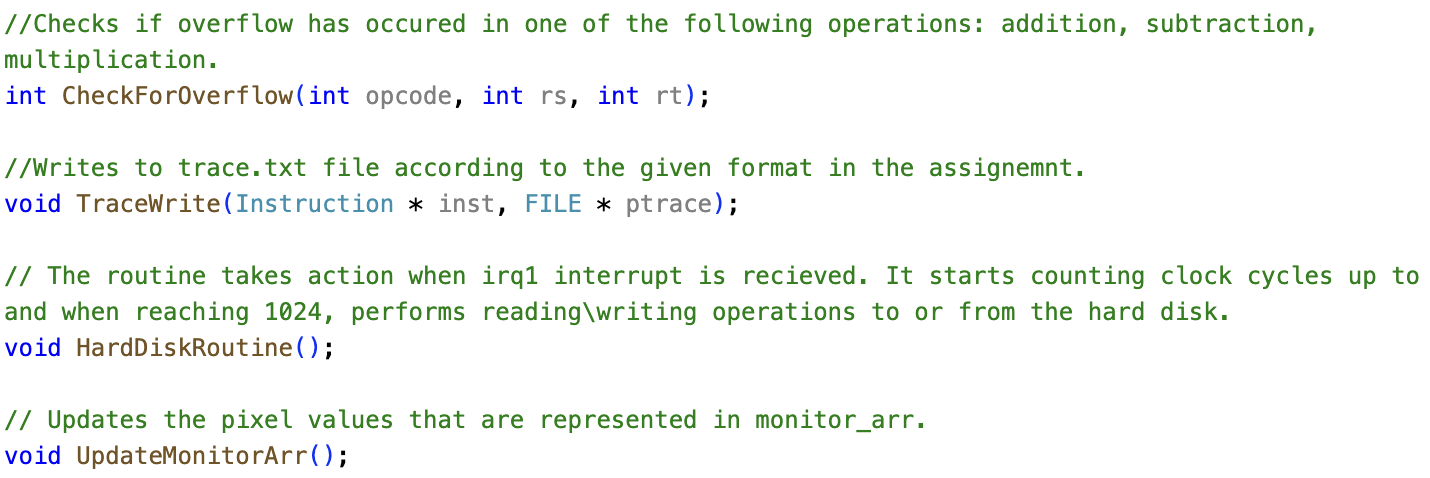
These functions are called at once upon program start, right after we opened all the program files in their correct mode. The role of these functions is to prepare all the arrays that are based on input files, by reading their data into the arrays and closing the files right afterwards. Moreover the function initializes th­e monitor\_arr to serve us at the next stage of the simulation.

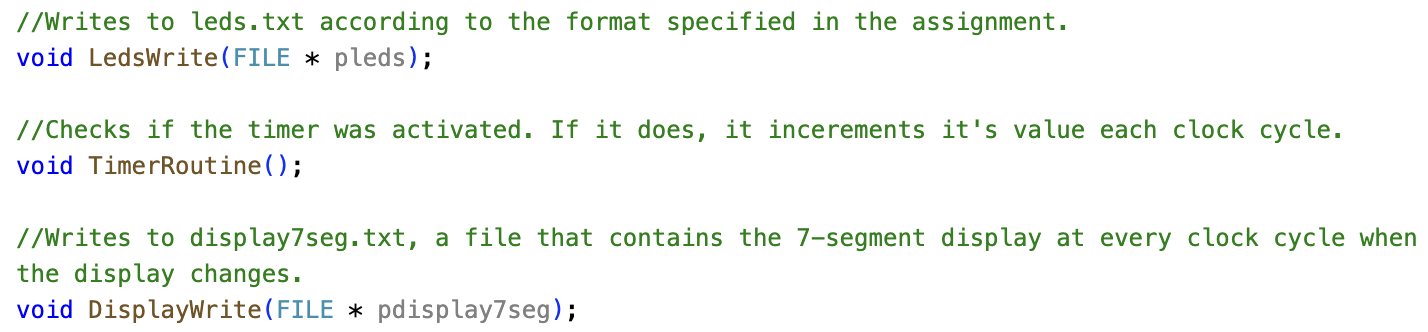
­ 

Main Runtime Functions:

The functions in this section are the heart of the program and contain most of the simulation functionality – fetching and executing the instructions, handling interrupts etc.

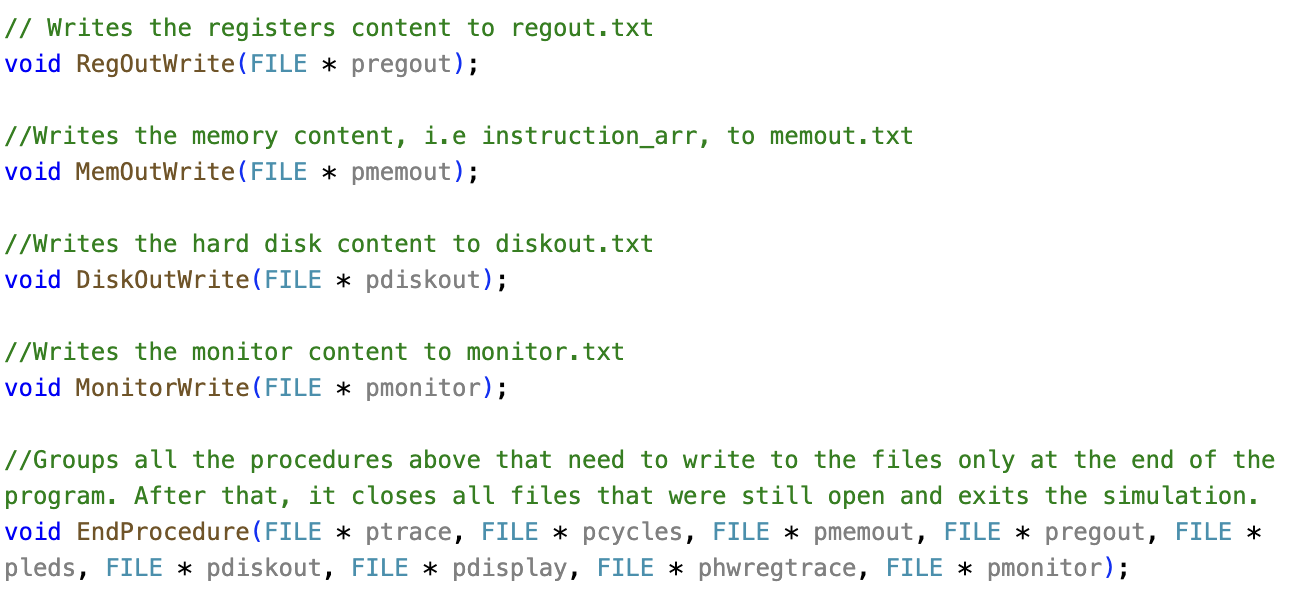






End of simulation functions

The role of these functions gets in practice when we finish the simulation (due to halt, overflow or finishing going over all of the instructions given to us in memint.txt). These functions write to the files that their output is determined only at the end of the program, and close all files that were in use during it.

­ 

Main Function and Simulation Flow Description

The main function opens all the files that were given to it in the command line in the right mode. If an error with opening one of the file occurs, an error message will be printed and the simulation will stop.

The main process starts by calling to SetArrays function in order to set up all the instruction, hard disk, irq2 and monitor arrays that will be in use throughout the whole simulation.

Right after that FetchInst function is called which triggers the main simulation phase. FetchInst main functionality resides in a while loop that gets the instruction that matches the current pc, creates it as a Command structure and sends it to execution. In order to create the correct command, the function first checks for interrupts. If there are any, we will have the pc updated accordingly, particularly if we are we get an irq1 interrupt we call the HardDiskRoutine to handle it and perform the read\write command from the disk and it will take 1024 cycles.  
Only after that CreateCommand function is called. CreateCommand knows how to distinguish between an instruction that uses the $imm register (I-type) and those that don’t (R-type) and assigns its fields accordingly.

Execution process:

First of all we write to trace.txt, that essentially serves as the log of the program. Right after that we perform the operation that matches the opcode of the command built.

Inside the switch we will note the following key operations:

* We propagate the clock by 1 and increment the pc by 1 whenever we detect a usage in the $imm register.
* We propagate the clock whenever we use lw\sw instructions.
* We will check if overflow will occur before performing addition\subtraction\multiplication. If it will, we will call the end procedure that will safely stop the program. The simulator as wev’e been instructed does not support overflowing int values.
* Timer propagation is checked (and executed if trigerred) whenever clock propagation is made.
* After in\out operations we will write to hwregtrace.txt.
* In the out operation we have a switch that detects operations are needed to be done with:
  + Leds
  + Display7seg
  + Reading\ writing from\to the hard disk.
  + Monitor

After the switch, the clock and pc will be incremented by 1 once more, indicating that the operation is done.

End procedure as was described above is being called in one of three cases – getting a halt command, in case of overflow or in case we finish going over all of the instructions array. After this procedure, all the files are closed and the program can finish safely.