RMIT UNIVERSITY

Programming Studio-2 COSC2804

Assignment 2

Assessment Type	Individual assignment. Submit online via GitHub Classroom. The last commit prior to the assignment deadline will be graded. Marks awarded for meeting requirements as closely as possible. Clarifications/updates may be made via announcements/relevant discussion forums.
Due Date	Sunday 1st October 2023, 11:59pm
Marks	45

1 Overview

This assignment will assess learning goals in computer architecture, focusing on assembly code programming. The tasks include:

a) Writing programs in LC-3 assembly language to demonstrate your understanding of loops, branches, subroutines, two's complement, and traps.
b) Converting between hexadecimal, binary and assembly representations of an LC-3 program.
c) Simulating a computer architecture via an existing Little Computer 3 (LC-3) virtual machine, written in C++.
d) Configuring all the required development tools (Minecraft Java Edition, Visual Studio Code + LC-3 extension, laser, Git, etc.)

The assignment is to be completed individually.

2 Learning Outcomes

This assessment relates to the following learning outcomes:

- **[CLO2]:** Apply fundamentals of computer architecture, operating systems, and system deployment to the design and development of medium-sized software applications.
- **[CLO4]:** Demonstrate skills for self-directed learning, reflection, and evaluation of your own and your peers work to improve professional practice.
- **[CLO5]:** Demonstrate adherence to appropriate standards and practice of Professionalism and Ethics.

3 Preliminaries

The aim of this assignment is to hone your LC-3 programming skills, using the game *Minecraft* as a testbed. This section will get you started with creating an LC-3 development environment and understanding how to write LC-3 programs that interact with Minecraft.

Setting up the tools required for carrying out this assignment.

Setup instructions can be found in the course Canvas shell. See the module Getting Started with LC-3, which is linked on the front page.

Communicating with Minecraft via LC-3.

LC-3 is a very simple language that offers no native way of communicating with Minecraft. To get around this, we have provided a modified LC-3 virtual machine that contains additional TRAP routines for this purpose. The additional TRAPs are summarised in the table below.

Trap Vector	Function	Description
0x30	postToChat(R0)	Outputs a null terminating string starting at the address contained in R0 to the Minecraft chat.
0x31	player.getTilePos() --> R0, R1, R2	Gets the position of the "tile" that the player is currently on. The x, y and z coordinates are output in registers R0, R1 and R2 respectively.
0x32	player.setTilePos(R0, R1, R2)	This function moves the player to the tile (x, y, z) = (R0, R1, R2).
0x33	getBlock(R0, R1, R2) --> R3	This function retrieves the ID of the block at tile (x, y, z) = (R0, R1, R2) and returns it to R3.
0x34	setBlock(R0, R1, R2, R3)	This function changes the ID of the block at tile (x, y, z) = (R0, R1, R2) to the value stored in R3.
0x35	getHeight(R0, R2) --> R1	This function calculates the y-position of the highest non-air block at (x, z) = (R0, R2) and returns the value to R1.
0x36	printRegisters()	Outputs the current register values to the console.

Notes:

- `TRAP 0x36 (printRegisters)` is provided for debugging purposes, since unlike the LC-3 web simulator shown in class, the virtual machine included with the starter code does not provide an easy way of inspecting the register values.

- Function arguments and return values are passed via the registers. For example, `TRAP 0x35 (getHeight)` assumes that the *x* and *z* arguments are passed via registers `R0` and `R2` respectively and outputs the return value to `R1`.
- **For this assignment, you must not modify the provided virtual machine in any way**. The only files in the starter repo that you should edit are the *.asm files, problem_15.txt, and README.md.

4 LC-3 Programming Challenges

Using traps, branches and logical operators (15 marks: 3 marks per problem)

1) Write an LC-3 assembler program that places a gold block (block ID #41) just above ground level, 4 units from the player in the *x* direction.
   a) Place your code in the assembly file "place_gold.asm".
   b) The z-coordinate of the glass block should match the player's z-coordinate.
   c) See Figure 1 for an illustration.

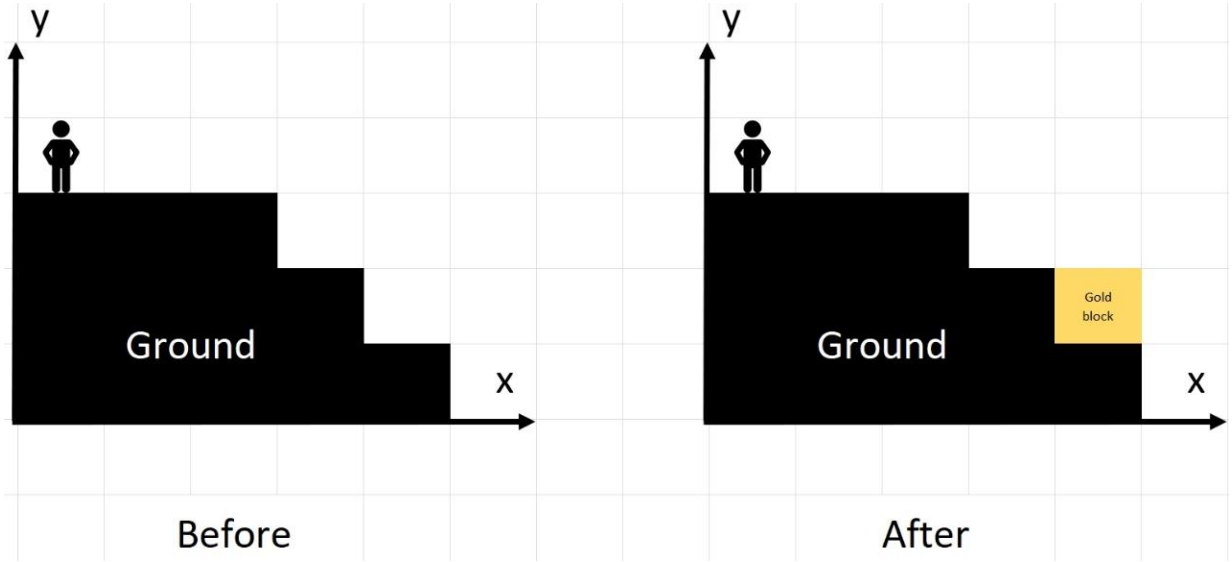

Figure 1: An example illustrating where the gold block should be placed for Problem 1.

2) Write an LC-3 assembler program that checks the ID of the block directly under the player's tile, then outputs a message to Minecraft chat depending on the block ID.
   a) Place your code in the assembly file "check_block.asm".
   b) If the ID of the block is even-numbered, e.g., air (ID #0), grass (ID #2), or cobblestone (ID #4), the program should output "The block beneath the player tile is even-numbered".
   c) If the block is odd-numbered and less than 10, e.g., stone (ID #1), the program should output "The block beneath the player tile is odd-numbered and less than 10".
   d) Otherwise, the program should output "The block beneath the player tile is odd-numbered and greater than 10".
   e) Note that the output messages should be sent to Minecraft chat, not the console!
   f) Please ensure that the text matches the specification above exactly, since your solution will be autograded!

3) Write an LC-3 assembler program that reads the player's current tile position:
(playerPos.x, playerPos.y, playerPos.z)
then teleports the player to:
(-playerPos.z, 3 * playerPos.y, |playerPos.x|)
where |playerPos.x| denotes the absolute value of playerPos.x.
   a) Place your code in the assembly file "teleport.asm".
   b) The player may end up somewhere "bad", e.g., in the middle of a mountain. You do not need to worry about handling this.

4) Write an LC-3 assembler program that calculates the bitwise OR between two blocks (based on their block IDs) then outputs a block representing the result.
   a) Place your code in the assembly file "bitwise_or.asm".
   b) The input block IDs should be read from:
      (playerPos.x + 1, playerPos.y – 1, playerPos.z)
      and
      (playerPos.x + 2, playerPos.y – 1, playerPos.z)
   c) The output block should be placed at:
      (playerPos.x + 3, playerPos.y – 1, playerPos.z)
   d) See Figure 2 for an illustration.

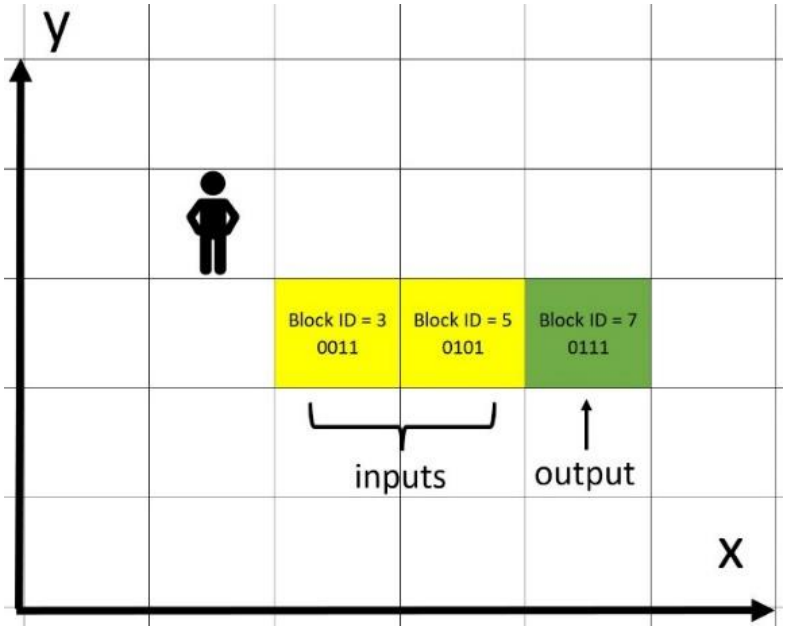

Figure 2: An example of how the bitwise OR program should behave.'

5) Write an LC-3 assembler program that behaves identically to the program from Problem 4, except that it calculates the bitwise **XOR** (exclusive OR) between two blocks.
   a) See https://en.wikipedia.org/wiki/Bitwise_operation for an explanation of XOR.
   b) Place your code in the assembly file "bitwise_xor.asm".
   c) The input and output block positions remain the same as in Problem 4.

Loops and subroutines (12 marks: 3 marks per problem)

6) Write an LC-3 assembler program that creates a "totem pole" using all the Minecraft block IDs up to a certain number.
   a) Place your code in the assembly file "totem_pole.asm".
   b) The assembly file contains a predefined constant, `HEIGHT`, that specifies the height of the totem pole.
   c) The first block in the totem pole should be placed just above ground level at the location (x, z) = (playerPos.x, playerPos.z + 2). Each successive block should be placed one unit higher than the last.
   d) The first block in the totem pole should be block ID = 1 (stone). The next block should be block ID = 2 (grass), and so on.
   e) Some blocks (e.g., lava) will start spilling after you build the pole, but this is okay. There may also be some empty spaces in the pole due to block IDs that aren't used.
   f) See Figure 3 for a visual explanation. Note that the horizontal axis in the figure is the **z-axis**.

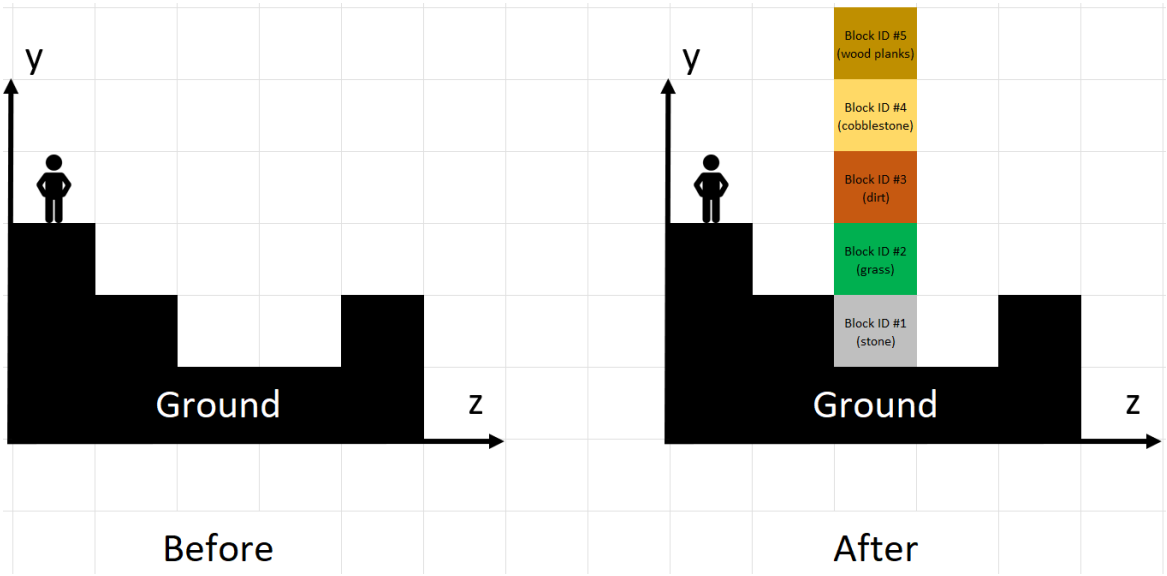

Figure 3: How the totem pole should be built for Problem 5 if `HEIGHT` = 5.

7) Write an LC-3 assembler program that reads a line of blocks extending from beneath the player and counts how far away the first air block (ID #0) is.
   a) Place your code in the assembly file "find_air.asm".
   b) The code should read a line of blocks starting beneath the player, i.e., at (playerPos.x, playerPos.y - 1, playerPos.z), and extending in the **positive x-direction**.
   c) The code should count the total number of non-air blocks before the first air block.
   d) The result should be stored in **R6**. Ensure that you use this specific register, since your solution will be autograded!
   e) See Figure 4 for a visual explanation.
      i. Green blocks represent grass, brown blocks represent dirt, white represents air.
      ii. The first block to be scanned is labelled "1", the second is labelled "2", etc.
      iii. In this example, there are 5 blocks before the first air block, so the value 5 should be stored in R6.

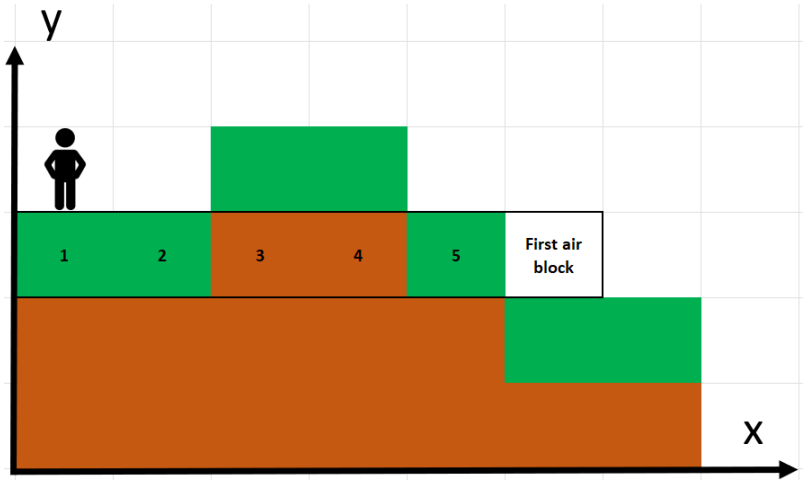

Figure 4: In this example, the program for Problem 7 should store the value 5 in R6.

8) Write an LC-3 assembler program that contains a subroutine for creating a "lamp" at a given location, then use it to create lamps at four locations, as specified below.
   a) Place your code in the assembly file "lamps.asm".
   b) For the purposes of this problem, a "lamp" at (x, y, z) is defined as:
      i. A stone block (block ID #1) at (x, y, z)
      ii. A stone block (block ID #1) at (x, y + 1, z)
      iii. A stone block (block ID #1) at (x, y + 2, z)
      iv. A glowstone (block ID #89) at (x, y + 3, z)
   c) The x, y and z arguments should be passed to the lamp-generating subroutine via designated registers (the choice of which registers is up to you).
   d) The subroutine should be used to place lamps at the following locations:
      i. (playerPos.x + 2, playerPos.y, playerPos.z + 2)
      ii. (playerPos.x + 2, playerPos.y, playerPos.z - 2)
      iii. (playerPos.x - 2, playerPos.y, playerPos.z + 2)
      iv. (playerPos.x - 2, playerPos.y, playerPos.z - 2)
   e) See Figure 5 for an illustration.

Figure 5: How the lamps should be placed around the player for Problem 7.

9) Write an LC-3 assembler program that places a rectangle of grass blocks underneath the player.
   a) Place your code in the assembly file "grass_rect.asm".
   b) The assembly file contains two predefined constants, `X_DIST` and `Z_DIST`, that specify the dimensions of the rectangle (see Figure 6 for an illustration). Your code should work for all non-negative values of `X_DIST` and `Z_DIST`, **including 0**.
   c) The rectangle should be made of grass blocks (block ID #2).
   d) The rectangle should be centred 1 unit under the player tile, so that after the rectangle is created, the player is standing on top of it.

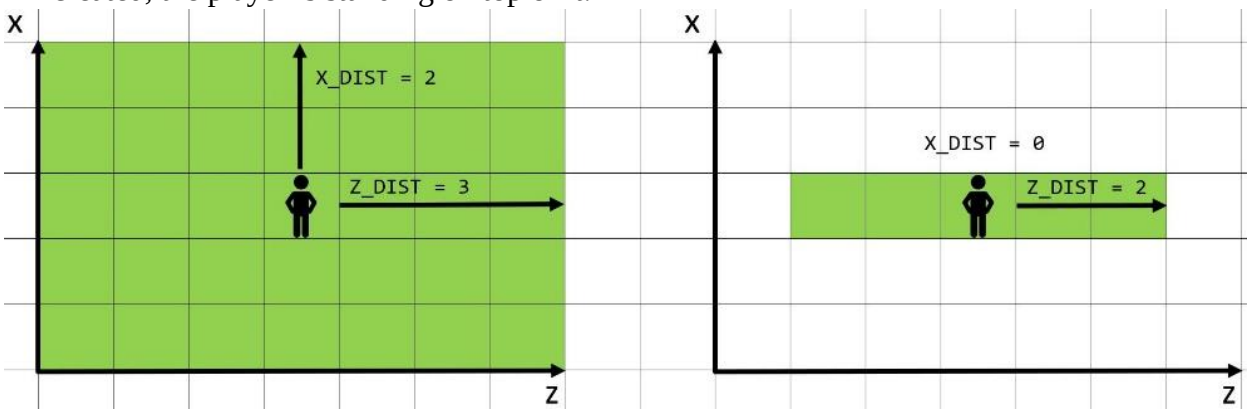

Figure 6: Illustration of the rectangles that should be built for `X_DIST` = 2, `Z_DIST` = 3 (on the left) and `X_DIST` = 0, `Z_DIST` = 2 (on the right).

Advanced challenges (15 marks: 3 marks per problem)

10) Write an LC-3 assembler program that checks whether the player is within a certain Euclidean distance of a specified "goal" point.
   a) Place your code in the assembly file "euclidean_dist.asm".
   b) The assembly file contains predefined constants, `G_X`, `G_Y` and `G_Z`, that specify the position of the goal point.
   c) An additional predefined constant, `GOAL_DIST`, specifies the distance bound to be checked. You may assume that `GOAL_DIST` > 0.
   d) The Euclidean distance between the player and the goal point is given by:

$$d_{euclidean} = \sqrt{(\text{playerPos.x} - \text{G\_X})^2 + (\text{playerPos.y} - \text{G\_Y})^2 + (\text{playerPos.z} - \text{G\_Z})^2}$$

However, calculating this quantity is tricky in LC-3, because there is no square root function. Therefore, instead of checking whether $d_{euclidean} <$ `GOAL_DIST`, you should check the squared distances, i.e.:

$$(\text{playerPos.x} - \text{G\_X})^2 + (\text{playerPos.y} - \text{G\_Y})^2 + (\text{playerPos.z} - \text{G\_Z})^2 < \texttt{GOAL\_DIST}^2$$

If this inequality is met, the program should output "The player is within distance of the goal" to Minecraft chat. Otherwise, it should output "The player is outside the goal bounds" to Minecraft chat.

11) Write an LC-3 assembler program that converts a number stored in memory into its binary representation in the Minecraft world.
   a) Place your code in the assembly file "write_binary.asm".
   b) The number to convert is specified in the LC-3 starter file as `NUMBER_TO_CONVERT`.
   c) You can assume that the number to convert will always be non-negative.
   d) Use air (block ID #0) to represent zeroes and stone blocks (block ID #1) to represent 1s.
      i) The **least** significant bit should be written to (playerPos.x + 1, playerPos.y, playerPos.z).
      ii) The next bit should be written to (playerPos.x + 2, playerPos.y, playerPos.z).
      iii) And so on… See Figure 7 for a visual explanation.
      iv) Since the word size in LC-3 is 16 bits, your program should always output 16 blocks, writing extra zeroes as air blocks if necessary.

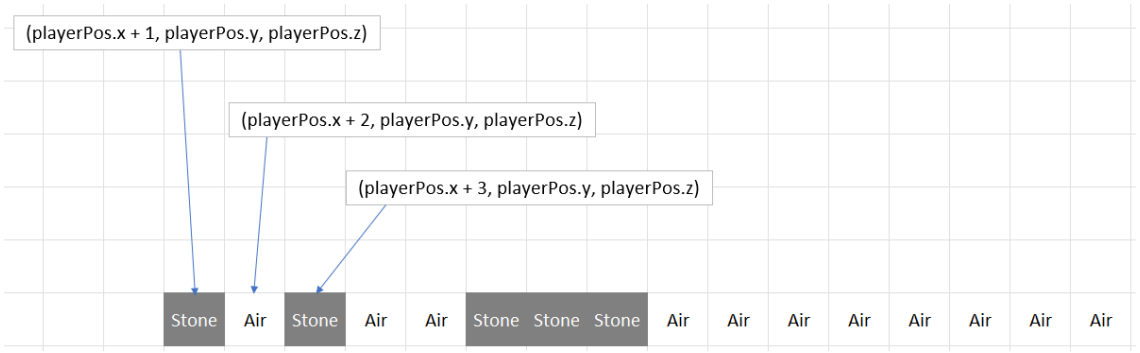

Figure 7: Illustration of how the binary representation should be output in Minecraft. The number being represented here is 229 = 0000000011100101. Note that the least significant bit is written closest to the player.

12) Write an LC-3 assembler program that reads two 16-bit binary words from the Minecraft world (formatted as in Problem 11 above) and adds them together, writing the answer to the Minecraft world in binary.
   a) Place your code in the assembly file "binary_addition.asm".
   b) The first number should be read from a line of 16 blocks starting at:
      (playerPos.x + 1, playerPos.y, playerPos.z)
      and extending to:
      (playerPos.x + 16, playerPos.y, playerPos.z)
   c) The second number should be read from a line of 16 blocks starting at:
      (playerPos.x + 1, playerPos.y, playerPos.z + 1)
      and extending to:
      (playerPos.x + 16, playerPos.y, playerPos.z + 1)
   d) The result should be written to a line of 16 blocks starting at:
      (playerPos.x + 1, playerPos.y, playerPos.z + 2)
      and extending to:
      (playerPos.x + 16, playerPos.y, playerPos.z + 2)

See Figure 8 for a visual explanation.

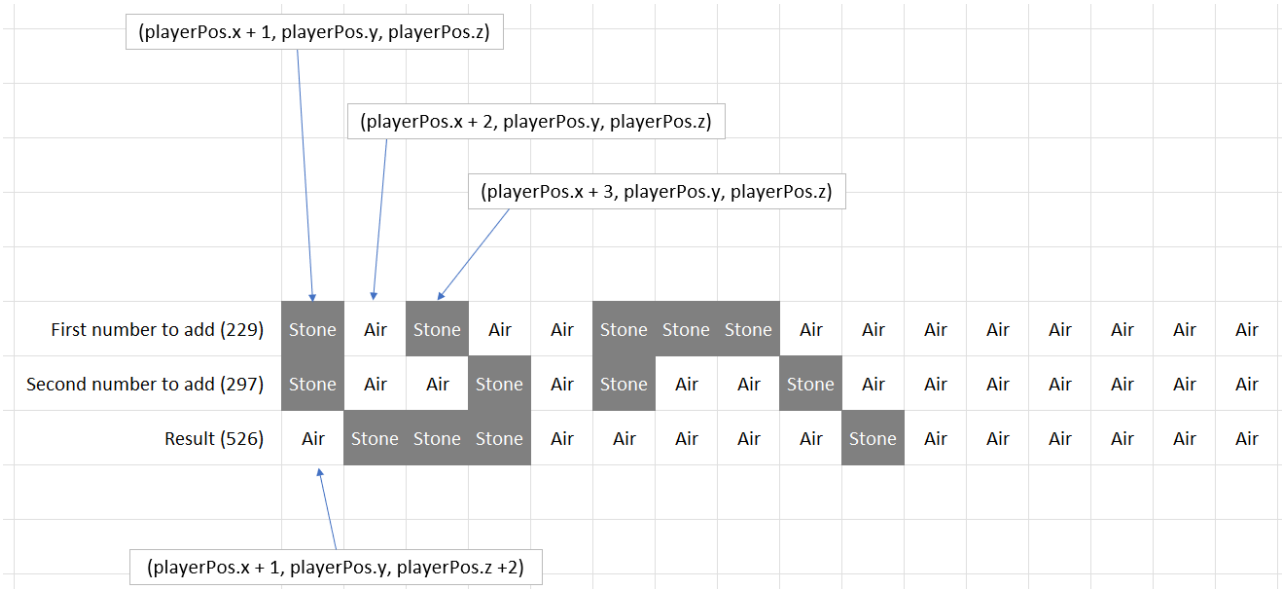

Figure 8: Illustration of how the sum 229 + 297 = 526 should appear in the Minecraft world for Problem 12.

13) Write an LC-3 assembler program that builds a stairway next to the player. You must abide by the following constraints:
   a) Place your code in the assembly file "stairway.asm".
   b) The stairway should be built from stone blocks (block ID #1).
   c) The assembly file contains three predefined constants, `WIDTH`, `LENGTH` and `HEIGHT`, that specify the dimensions of the stairway.
   d) You can assume that the dimensions are strictly positive and that `LENGTH` >= `HEIGHT`.
   e) One way to visualise the construction of the stairway is to imagine multiple "layers" of blocks stacked on top of each another, as shown in Figure 9. Following this terminology:

      The bottom layer of the stairway should have one corner at
      (playerPos.x + 1, playerPos.y, playerPos.z + 1)
      and another corner at
      (playerPos.x + `WIDTH`, playerPos.y, playerPos.z + `LENGTH`).

      The next layer up should have one corner at
      (playerPos.x + 1, playerPos.y + 1, playerPos.z + 2)
      and another corner at
      (playerPos.x + `WIDTH`, playerPos.y + 1, playerPos.z + `LENGTH`).
      and so on…

      The top layer should have one corner at
      (playerPos.x + 1, playerPos.y + `HEIGHT` - 1, playerPos.z + `HEIGHT`)
      and another corner at
      (playerPos.x + `WIDTH`, playerPos.y + `HEIGHT` - 1, playerPos.z + `LENGTH`).

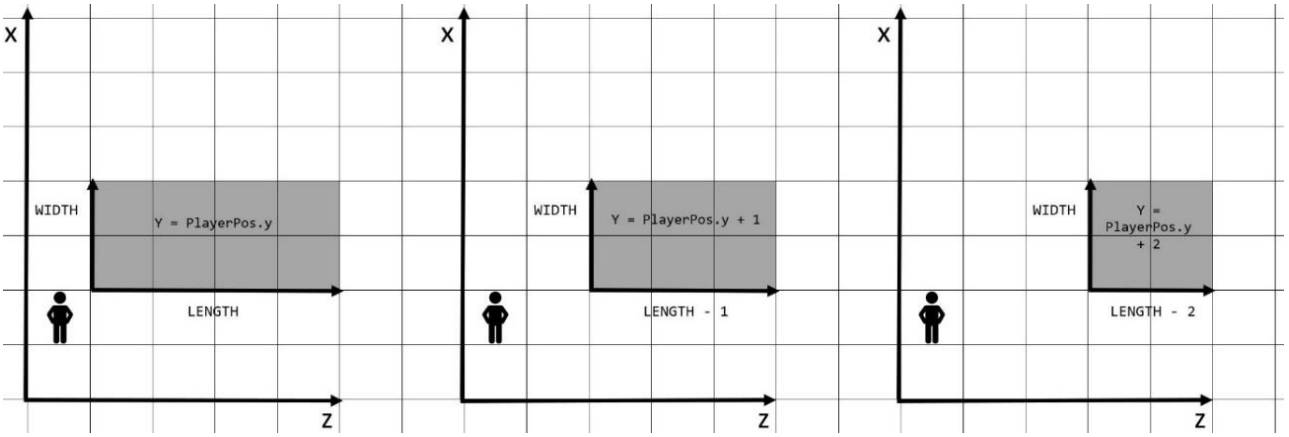

Figure 9: Illustration of how a stairway with `WIDTH` = 2, `LENGTH` = 4 and `HEIGHT` = 3, can be built from three layers of blocks stacked on top of each other. The bottom layer is shown to the left; the top layer is shown to the right.

f) See Figure 10 for an illustration of how the stairway should appear in-game.

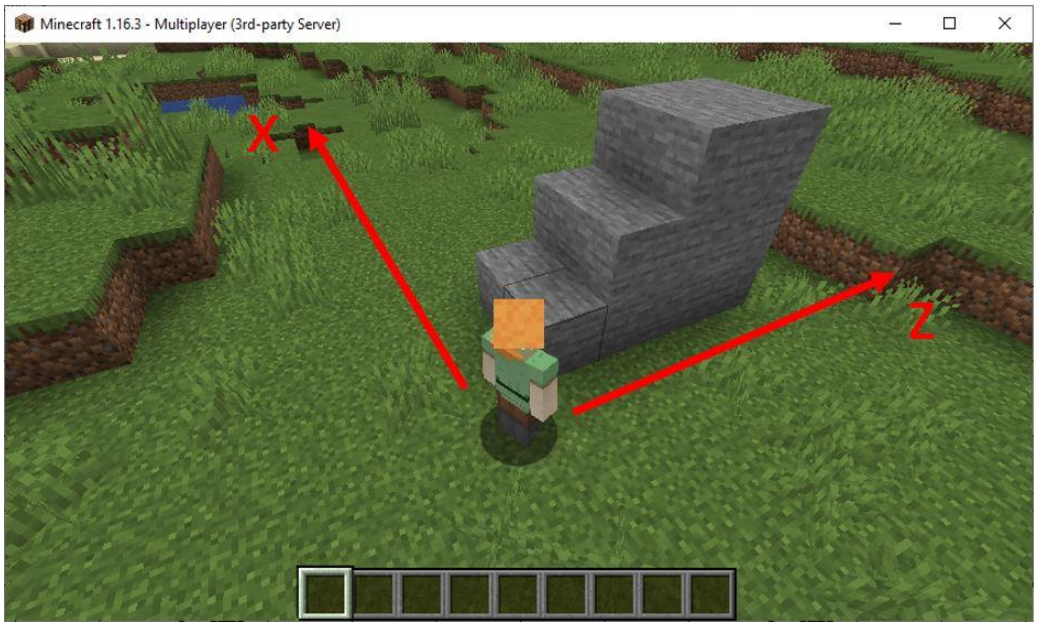

Figure 10: In-game illustration of a stairway with `WIDTH` = 2, `LENGTH` = 4 and `HEIGHT` = 3.

14) Write an LC-3 program that flattens a 3x3 area centred underneath the player.
   a) Place your code in the assembly file "terraform.asm".
   b) Let *h* denote the height of the land at the (x, z) location of the player. The height of the 3x3 area centred underneath the player should be levelled so that it all has a height of *h*.
   c) Locations with a land height of less than *h* should be built up to the required height by placing grass blocks (block ID #2). Be careful not to replace any blocks unnecessarily; only air blocks should be replaced with grass.
   d) Locations with a land height of more than *h* should be cut down to the required height by replacing non-air blocks with air blocks (block ID #0).

e) See Figure 11 for a visual explanation.

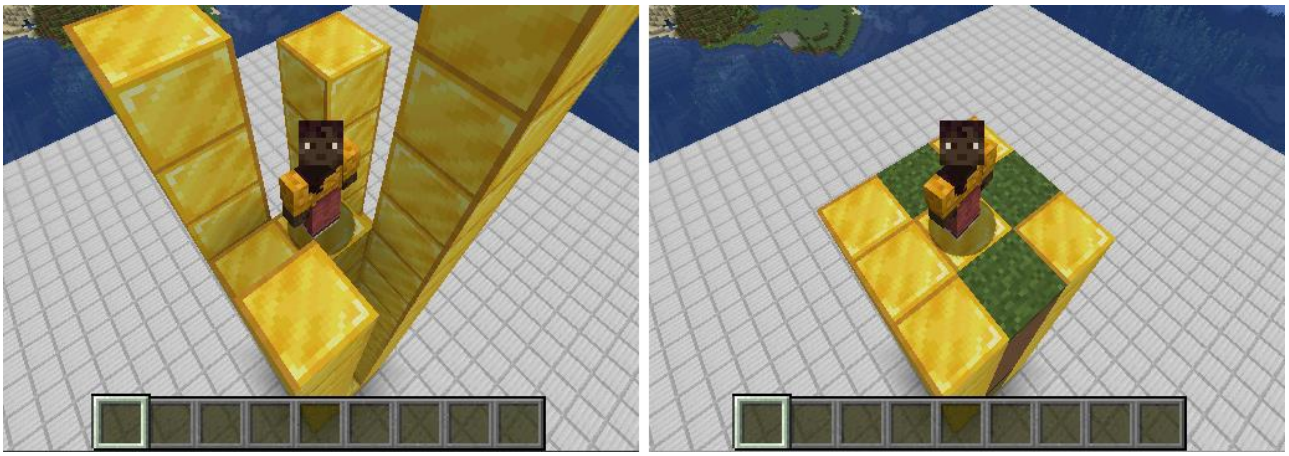

Figure 11: An example that shows how the terraform program for Problem 14 should behave.

Understanding the LC-3 instruction set (3 marks)

For the below problem, please place your solutions in the file "problem_15.txt".

15) Below is an excerpt from an LC-3 program, where the instructions highlighted red are expressed in hexadecimal:

```
.ORIG x3000
x5020
ADD R0, R0, #7
x923F
x1261
x1240
x1240
x103F
x03FC
HALT
.END
```

a) Rewrite the program in a more readable format.
   i. First, convert the hexadecimal instructions into binary.
   ii. Now, write out the full program in LC-3 assembler notation. (You are allowed add labels before any of the instructions if you wish.)

b) What value will `R1` hold by the end of the program if we replace the third line (`ADD R0, R0, #7`) with:
   i. `ADD R0, R0, #4`
   ii. `ADD R0, R0, #9`

c) Write a succinct, plain English summary of what the program does.
Note: Don't explain each line of code; explain what the overall program does.

5 Academic integrity and plagiarism (standard warning)

Academic integrity is about honest presentation of your academic work. It means acknowledging the work of others while developing your own insights, knowledge and ideas. You should take extreme care that you have:

- Acknowledged words, data, diagrams, models, frameworks and/or ideas of others you have quoted (i.e. directly copied), summarised, paraphrased, discussed or mentioned in your assessment through the appropriate referencing methods.
- Provided a reference list of the publication details so your reader can locate the source if necessary. This includes material taken from Internet sites. If you do not acknowledge the sources of your material, you may be accused of plagiarism because you have passed off the work and ideas of another person without appropriate referencing, as if they were your own.

RMIT University treats plagiarism as a very serious offence constituting misconduct. Plagiarism covers a variety of inappropriate behaviours, including:

- Failure to properly document a source.
- Copyright material from the internet or databases.
- Collusion between students.

For further information on our policies and procedures, please refer to the following: https://www.rmit.edu.au/students/student-essentials/rights-and-responsibilities/academic-integrity.

We will run code similarity checks.

6 Getting Help

There are multiple venues for getting help. The first places to look are the Canvas modules and discussion forum. You are also encouraged to discuss any issues you have in class with your tutors. Please refrain from posting solutions (full or partial) to the discussion forum.

7 Marking Guide

The rubric that will be used to grade the assignment is viewable on Canvas. (See the bottom of the Assignment 2 page.)

Important note: Please do not rename any of the predefined constants in the *.asm files. Most components of the assignment will be autograded, and changing the names of the constants will mess up the autograding scripts!