FIT1047 S2 2018
Assignment 1

Submission guidelines

This is an individual assignment, **group work is not permitted.**

We use **tools to check for plagiarism and collusion.**

Deadline: September 7th, 2018, 11:55pm

Submission format: PDF for the written tasks, LogiSim circuit files for task 1, MARIE assembly files for task 2. All files must be uploaded electronically via Moodle.

Individualised exercises: Some exercises require you to pick one of several options based on your student ID.

Late submission:

- By submitting a special consideration form available from `http://www.monash.edu.au/exams/special-consideration.html`
- Without special consideration, you lose 5% of your mark per day that you submit late (including weekends). Submissions will not be accepted more than 5 days late.

  This means that if you got x marks, only $0.95^n \times x$ will be counted where n is the number of days you submit late.

In-class interviews: See instructions for Task 2 for details.

Marks: This assignment will be marked out of 90 points. It is worth 22.5% of your total unit marks.

Plagiarism: It is an academic requirement that the work you submit be original. **Zero marks** will be awarded for the whole assignment if there is any evidence of copying (including from online sources without proper attribution), collaboration, pasting from websites or textbooks. This includes both the student who copies, and the student who lets them copy!

Further Note: When you are asked to use internet resources to answer a question, this **does not mean copy-pasting text** from websites. Write answers in your own words such that your understanding of the answer is evident. Acknowledge any sources by citing them.

1 Boolean Algebra and Logisim Task

The following truth table describes a Boolean function with four input values $X1, X2, X3, X4$ and two output values $Z1, Z2$.

X1	X2	X3	X4	Z1	Z2
0	0	0	0	1	0
0	0	0	1	1	1
0	0	1	0	0	1
0	0	1	1	1	1
0	1	0	0	1	1
0	1	0	1	0	0
0	1	1	0	1	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	1	1	1
1	0	1	0	1	0
1	0	1	1	0	1
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	0	0

The main result of this task will be a logical circuit correctly implementing this Boolean function in the Logisim simulator. Each step as defined in the following sub-tasks needs to be documented and explained.

1.1 Step 1: Boolean Algebra Expressions (10 points)

Write the Boolean function as Boolean algebra terms. First, think about how to deal with the two outputs. Then, describe each single row in terms of Boolean algebra. Finally, combine the terms for single rows into larger terms.

Briefly explain these steps for your particular truth table (e.g., explain for one particular row how you come up with the Boolean terms for that row, and then explain how you combine all rows). This explanation should be no more than a few sentences.

Notation: use the following symbols and notation for writing Boolean algebra expressions. Variables are upper-case (e.g., $X1$, $Z2$). Boolean AND is written without a symbol, e.g. $X1X2$. Boolean OR is written with the $+$ symbol, e.g. $X1 + X2$. Negation is written using an overline, e.g. $\overline{X1}$. **Important:** when writing terms like NOT $X1$ AND NOT $X2$, there must be a clear gap in the overlines, e.g. $\overline{X1}\ \overline{X2}$.

1.2 Step 2: Logical circuit in Logisim (10 points)

Model the resulting Boolean terms from Step 1 in a single Logisim circuit, using only the basic gates AND, OR, NOT. You can use gates with more than two inputs.

Briefly explain your construction (as for Step 1, a short explanation is enough).

Test your circuit using values from the truth table and **document the tests**.

1.3 Step 3: Optimized circuit (10 points)

The goal of this task is to find a minimal circuit using only AND, OR, and NOT gates. Based on the truth table and Boolean algebra terms from Step 1, **optimize the function using Karnaugh maps**.

You will need to create two Karnaugh maps, one for each output. Your documentation must show (a) the maps, (b) the groups found in the maps, and (c) how they relate to terms in the optimized Boolean function.

Then use Logisim to create a minimal circuit, using only AND, OR, and NOT gates. Test your optimized circuit using values from the truth table and document your tests.

2 Development of a Database for Unit Prerequisites in MARIE

In this task you will develop a MARIE program that implements a database with information about unit prerequisites. We will break it down into small steps for you.

Most of the tasks require you to write **code** and **test cases**. On Moodle, you will find a **template** for the code. It contains some subroutines that you should use, as well as a number of predefined test cases. **Your submission must be based on this template**, i.e., you must add implementations of your own subroutines into this template. The code must contain comments, and you submit the `.mas` file together with the rest of your assignment.

In-class interviews: You will be required to demonstrate your code to your tutor after the submission deadline. **Failure to demonstrate will lead to zero marks** being awarded for the entire programming part of this assignment.

Code similarity: We use tools to check for collaboration and copying between students. If you copy your code from other students, or you let them copy your code, **you will receive 0 marks for the entire assignment**.

2.1 Strings

This section doesn't contain any tasks for you yet, it only introduces a concept you need for the rest of the assignment.

A *string* is a sequence of characters. It's the basic data structure for storing text in a computer. There are several different ways of representing a string in memory – e.g. you need to decide which character set to use (Unicode or ASCII), and how to deal with strings of arbitrary length.

For this assignment, we will use the following string representation:

- A string is represented in a contiguous block of memory, with each address containing one character.
- The characters are encoded using the ASCII encoding.
- The end of the string is marked by the value 0.

As an example, this is how the string FIT1047 would be represented in memory (written as hexadecimal numbers):

```
046 049 054 031 030 034 037 000
```


Note that for a string with n characters, we need $n + 1$ words of memory in order to store the additional 0 that marks the end of the string.

In MARIE assembly, we can use the `HEX` or `DEC` keywords to initialise the MARIE memory with a fixed string:


```
FIT1047,    HEX 046
            HEX 049
            HEX 054
            HEX 031
            HEX 030
            HEX 034
            HEX 037
            HEX 000
```

After assembling this code, you can verify that the string has been loaded into the MARIE memory.

The assignment template contains two subroutines. The `PrintString` subroutine outputs a string one character at a time using the `Output` instruction. The `InputString` subroutine reads a string one character at a time using the `Input` instruction.

Tip: you can switch the type of input in the MARIE simulator. For the actual string, use the unicode input method so you can enter real characters. For the final 0, switch to the decimal input method.

2.2 Your name as a MARIE string (4 points)

This is the first task you need to submit.

Similar to the FIT1047 example above, encode your name using ASCII characters. You should encode at least 10 characters – if your name is longer, you can shorten it if you want, if it's shorter, you need to add some characters (such as !?! or ..., or invent a middle name).

In the assignment template, you will find a label `Name`*, which is initialised to the value* `HEX 0`*. You need change this part of the code so that after assembling, the MARIE memory contains the string with your name starting at label* `Name`*.*

2.3 Trimming a string (14 points)

This is your second task.

Trimming a string means to remove any space characters from the beginning and the end of the string. For example, given the string ␣␣`FIT`␣`1047`␣␣␣ as input, trimming it would result in the string `FIT`␣`1047`. The ␣ symbolises a space character (its ASCII value is decimal 32). Trimming is often used to clean up user input.

You have to implement a subroutine `TrimString` that trims a string. Its argument, which is passed in the label `TrimStringAddr`, is the address in memory where the string starts. When the subroutine finishes, the string at that address must not contain any leading or trailing space.

We will trim the string in two phases.

Phase 1 (5 points). The first phase removes any trailing space (ie, spaces at the end of the string). Your code should follow these steps to achieve the result:

1. Iterate through the string, one character at a time, until you reach the end of the string (the 0 character). This is similar to how the `PrintString` subroutine works, just without actually printing the characters.
2. In step 1, you found the address of the final 0 in the string. Now iterate *backwards*, and replace any space character (decimal 32) with a 0, until you reach a character that is not a space character, or the beginning of the string.

The second phase removes leading space characters. There are two approaches, an easy one and a more complicated one. You can choose to implement the easy approach for 2 points, or the complex one for 5 points.

Easy phase 2 (2 points) This approach modifies the start address of the string, updating it to the first non-space character.

1. Iterate through the string, one character at a time, until you reach a character that is not the space character (decimal 32).
2. Update the start address of the string (at label `TrimStringAddr`) to the address you found in step 1 (the address of the first non-space character).

Complicated phase 2 (5 points, alternative to easy phase 2) The easy approach above has a disadvantage in some programs, because you have to make sure that you don't use the old string address any more. The following steps will modify the original string and remove leading spaces (without changing its start address).

1. Keep track of two addresses, the current read address `CurRead` and the current write address `CurWrite`. Both are initialised with the start address of the string.
2. Iterate through the string by incrementing `CurRead` until you find the first non-space character.
3. Iterate through the remaining string, reading the character from `CurRead` and storing it at `CurWrite` at each step. `CurWrite` is now incremented together with `CurRead` in each step.

There's a short animation on Moodle to help you understand this algorithm.

Test cases (4 points) *You need to implement this subroutine in the assignment template file. The template already contains one test case. Add the following additional test cases, and briefly document them in your submission PDF:*

1. Trimming a string that does not contain any space characters
2. Trimming a string that does not contain any leading or trailing space characters, but does contain at least one space character
3. Trimming a string with at least two leading space characters
4. Trimming a string with at least two trailing space characters

2.4 String comparison (15 points)

Now we are going to implement a *string comparison subroutine*. We will use *lexicographic ordering*, i.e., the ordering you would use e.g. when sorting names alphabetically.

The subroutine takes two (addresses of) strings as arguments and returns -1 if the first string is lexicographically smaller than the second, 0 if they are equal, and 1 if the first string is lexicographically greater than the second string. For example, ABCD is smaller than BCDEF (because the first character is smaller), ABC is smaller than ABCDE (because it is a proper prefix), and BCDF is greater than BCDE (because the first character that is different, the F, is greater).

The subroutine will communicate the result back to the caller (the code that called `JnS`) in the address with the label `CmpStringResult`, i.e., after returning from the subroutine, `CmpStringResult` will be equal to -1 if the first string is smaller, 0 if the strings are equal, and 1 if the first string is greater.

Subroutine (10 points) In order to achieve this, your code needs to loop through both strings simultaneously, one character at a time:

1. If the current character in the first string is 0 (i.e., the end of the string), check if the character in the second string is also 0. If it is, set `CmpStringResult` to 0 ("equal") and return, otherwise set it to -1 ("less") and return.
2. If the current character in the first string is not 0:
   a) Check if the current character in the second string is 0. If it is, then the first string is larger, so set `CmpStringResult` to 1 and return.
   b) If the current character in the second string is not 0, you now have to compare the two characters. If they are equal, move to the next character in both strings and go back step 1 in order to compare the next characters.
   c) If they are not equal, set `CmpStringResult` according to the ordering of the characters (i.e., if the character in the first string is less than the character in the second string, set it to -1, otherwise to 1).

You need to implement the subroutine in the assignment template file.

Test cases (5 points) *The template already contains one test case, using your name as both arguments, for which your subroutine should return that the two strings are equal. Add the following additional test cases and document them in your PDF submission:*

1. A test case where the two strings are empty

2. A test case where the first string is shorter and a prefix of the second string (i.e., it's shorter, but all characters are the same as in the second string; e.g. "abcde" is a prefix of "abcdefgh")
3. A test case where the second string is shorter and a prefix of the first string
4. A test case where the first two characters of the strings are the same, but the first string is lexicographically smaller than the second
5. A test case where the first two characters of the strings are the same, but the first string is lexicographically larger than the second

2.5 Database lookup

We will now explain the database format. This is just an explanation, the first task you need to implement is in Sect. 2.5.2. Let's assume we have the following information about unit prerequisites:

- for FIT2093, the prerequisites are FIT1045 and FIT1047
- for FIT2100, the prerequisite is FIT1047

We now have to store this information in our MARIE memory in a way that lets us query it. For this task, we will use the following encoding into MARIE:

2.5.1 Representing the data in MARIE

For each unit, we store a string that lists the prerequisites. In this case, since our database has two entries (for FIT2093 and FIT2100), we store two strings:


```
FIT2093, DEC 70  / F
         DEC 73  / I
         DEC 84  / T
         DEC 49  / 1
         DEC 48  / 0
         DEC 52  / 4
         DEC 55  / 7
         DEC 44  / ,
         DEC 32  /
         DEC 70  / F
         DEC 73  / I
         DEC 84  / T
         DEC 49  / 1
         DEC 48  / 0
         DEC 52  / 4
```

```
          DEC 53  / 5
          DEC 0

FIT2100,  DEC 70  / F
          DEC 73  / I
          DEC 84  / T
          DEC 49  / 1
          DEC 48  / 0
          DEC 52  / 4
          DEC 55  / 7
          DEC 0
```

Now that we have the actual data in memory, what is missing is an *index*, i.e., a way of finding the right information when the user queries the database. The index is represented as a "list" of strings and addresses:

```
IdxAddr,  ADR Idx
Idx,      DEC 70  / F
          DEC 73  / I
          DEC 84  / T
          DEC 50  / 2
          DEC 48  / 0
          DEC 57  / 9
          DEC 51  / 3
          DEC 0
          ADR FIT2093
          DEC 70  / F
          DEC 73  / I
          DEC 84  / T
          DEC 50  / 2
          DEC 49  / 1
          DEC 48  / 0
          DEC 48  / 0
          DEC 0
          ADR FIT2100
          DEC 0             / End of the index
```


The index starts at address `Idx`, with the string `FIT2093` (including final `0`), followed by the address where the prerequisites for FIT2093 are stored (`ADR FIT2093`). Then we have the next index entry, which is the string `FIT2100` followed by the address where the data for FIT2100 is stored. Finally, the end of the index is marked by a 0.

The assignment template contains a database with a few extra entries.

2.5.2 Finding an entry (12 points)

You will now write a subroutine `FindEntry` that finds an entry in the index and outputs the corresponding data. The argument to the subroutine is the address of the string to look up, passed using the label `FindEntryString`.

Subroutine (10 points) The subroutine should work as follows:

1. Introduce a label `CurrentEntry`, and load the address of the first index entry into `CurrentEntry`.
2. If the current entry contains an empty string (i.e., there is a 0 stored at that entry, as for the `End of index` marker above), your subroutine should output "not found" and then return.
3. Otherwise, compare the string in the current entry to the query string using the `CmpString` subroutine.
   a) If the strings are equal, load the address of the data (which is stored at the address after the index string), output the data using the `PrintString` subroutine and return.
   b) If they are not equal, skip to the next index entry by searching for the 0 that marks the end of the current string and skip the next address (which contains the address of the prerequisites), and continue with step 2.

Test cases (2 points) *You need to add your search subroutine into the assignment template. Your search subroutine must use your comparison subroutine from 2.4. The template already contains one test case. You need to add two more test cases as follows:*

1. A test case that finds the first entry in the database
2. A test case that finds the last entry in the database

2.5.3 Cleaning up user input (15 points)

The code above now works if the user enters the unit code exactly as it is stored in the database. We want to make the system more flexible by automatically stripping white space, converting the user's input into upper case, and adding the letters FIT if necessary. E.g., if the user enters "␣Fit2100" or "2100" they should still get the correct answer.

Subroutine (12 points) Write a subroutine `CleanInput` that takes the address of a string as input using label `CleanInputString` and modifies the string in three steps:

1. Trim white space using the subroutine developed earlier. (1 point)
2. Convert all characters into upper case (and leave other characters such as numbers unchanged). (5 points)
3. If the string starts with a number, your subroutine will add the characters "FIT" to the beginning of the string. (6 points)

Hint: to convert an ASCII character from lower case to upper case, you just have to subtract a fixed number from the integer that encodes the ASCII character. For example, "A" has ASCII code 65, and "a" has code 97, so subtracting 32 from a lower case character converts it into upper case.

Test cases (3 points) *You need to add your flexible input subroutine into the assignment template. The template already contains one test case. You need to add three more test cases as follows:*

1. A test case that does not require any changes (i.e., all characters are already upper case)
2. A test case that contains some lower case and some upper case characters
3. A test case that starts with a number (i.e., your subroutine should add the "FIT")

2.5.4 Putting everything together

The template contains a section (commented out, at the very start) that calls all your subroutines in the right order to implement a simple application. When you run that code, it will

1. Ask the user to input a unit they want to look up
2. Clean up the user input
3. Look up the prerequisites in the database and print them if found
4. Go back to step 1