FIT1047 S1 2016
Assignment 1

Submission guidelines

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

Deadline: September 11, 2016, 11:59pm

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. Just create a zip Archive containing all the files and upload this archive to Moodle.

Individualised exercises: Downloading the assignment tasks requires to input your student ID. Based on the student ID, you will get one of several versions of some of the exercises.

Late submission:

- By submitting a special consideration form, available from `http://www.monash.edu.au/exams/special-consideration.html`
- Or, 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 70 points, and count for 17.5% 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.

The faculty's Plagiarism Policy applies to all assessment:

`http://intranet.monash.edu.au/infotech/resources/students/assignments/policies.html`

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	0	1
0	0	0	1	1	0
0	0	1	0	0	1
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	1	1	1
0	1	1	0	1	0
0	1	1	1	0	1
1	0	0	0	1	1
1	0	0	1	1	0
1	0	1	0	1	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	0	1	1	1
1	1	1	0	1	0
1	1	1	1	1	1

The main result of this task will be two logical circuits, both 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 (8 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.

1.2 Step 2: Logical circuit in Logisim (8 points)

Model the resulting Boolean terms from Step 1 in a single Logisim circuit, using the basic gates AND, OR, NOT. You can use gates with more than 2 inputs.

Explain what you did for each step.

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

1.3 Step 3: Optimized circuit (9 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 should show the maps as well as the groups found in the maps and how they relate to terms in the optimized Boolean function.

Then use Logisim to create a minimal circuit. Don't use any other gates than AND, OR, and NOT.

Test your optimized circuit using values from the truth table.

2 A MARIE calculator

In this task you will develop a MARIE calculator application. We will break it down into small steps for you.

Most of the tasks require you to write code, test cases and some small analysis. The code must contain comments, and you submit it as `.mas` files together with the rest of your assignment. The test cases should also be working, self-contained MARIE assembly files. The analysis needs to be submitted as part of the main PDF file you submit for this assignment.

Note that all tasks below **only need to work for positive numbers**. If you want a challenge, you can try and make your calculator work for negative numbers, but it's not required to get full marks.

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 to the entire programming part of this assignment.

2.1 MARIE integer multiplication and division

2.1.1 Multiplication (5 points)

Implement a subroutine for multiplication (based on the multiplication code you wrote in the labs – you can use the sample solution as a guideline). Test your subroutine by writing a test program that calls the subroutine with different arguments, and then step through the programs in the MARIE simulator.

You need to submit at least one MARIE file that contains the subroutine and a test case.

2.1.2 Division (6 points)

The following code is a poor attempt at implementing a division subroutine in MARIE. It contains a number of errors (both in the assembly syntax and in the program logic). Find and fix these errors, and produce a list that documents for each error how you found it and how you fixed it.

Write a test program that calls the subroutine with different arguments, and then step through the programs in the MARIE simulator. Don't try to just re-implement division – the point of this task is to identify and fix errors in existing code!

```
/ Subroutine for division (with errors)
/ Quotient  = Dividend / Divisor

Divide,         Load 0          / entry point of subroutine, set AC to 0
                Store DivDvdnd  / Make dividend zero
Div1,           Load DivQt      / reduce quotient
                Subt DivDvsr
                Skipcond 800    / skip if result is negative
                JumpI Div2      / not negative: keep subtracting
                Add DivDvsr     / negative: add back
                Load DivQt      / save as remainder
                JnS Divide      / Return to calling code
Div2,           Store DivQt     / save quotient
                Load DivDivnd   / increment dividend
                Add 1
                Store DivDvdnd
                Jump Div1       / repeat iteration

DivDvsr,        DEC 0           / Divisor
DivDvdnd,       DEC 0           / Dividend
```

```
DivQt,          DEC 0           / Output: Quotient
DivRem,DEC 0 / Output: remainder

One,            DEC 1           / Constant 1

/ END subroutine for division
```

2.2 Reverse Polish Notation (RPN)

RPN is a notation for arithmetic expressions in which the operator follows the arguments. For example, instead of writing $5 + 7$, we would write 5 7 +. For more complex expressions, this has the advantage that no parentheses are needed: compare $5 \times (7 + 3)$ to the RPN notation 5 7 3 + $\times$.

A calculator for RPN can be implemented using a **stack**, one of the fundamental *data structures* we use in programming. A stack is just a pile of data: you can only *push* a new data value to the *top* of the stack, or *pop* the top-most value.

To evaluate RPN, we just need to go through the RPN expression from left to right. When we find a number, we push it onto the stack. When we find an operator, we pop the top-most two numbers from the stack, compute the result of the operation, and push the result back onto the stack. Here is a step-by-step evaluation of 5 7 3 + $\times$:

current symbol	*operation*	*stack after operation*
5 7 3 + $\times$	push 5	5
5 **7** 3 + $\times$	push 7	5, 7
5 7 **3** + $\times$	push 3	5, 7, 3
5 7 3 **+** $\times$	pop 3 and 7, push $7 + 3$	5, 10
5 7 3 + **$\times$**	pop 10 and 5, push 5×10	50

Note that we read the stack from left to right here, so e.g. after reading the 7, the stack contains the values 5 and 7, with 7 at the top of the stack.

After all operations are finished, the final result is the only value left on the stack.

2.2.1 A stack in MARIE assembly (10 points)

A stack can be implemented in memory by keeping track of the address of the current top of the stack. This address is called the `StackPointer`, and we use a label to access it easily:

```
StackPointer,       HEX FFF
```

In this example, we set the initial stack pointer to address `FFF`, the highest address in MARIE.

A *push* operation writes the new value into the address pointed to by `StackPointer`, and then decrements the stack pointer. A *pop* operation increments the `StackPointer` and then returns the value at the address pointed to by the `StackPointer`. This means that when we push a value, the stack "grows downwards" from the highest address towards 0.

1. Write a sequence of instructions that pushes the values 5, 4, 3 onto the stack and then pops them again, printing each value using the `Output` instruction. Step through your code to make sure that the `StackPointer` is decremented and incremented correctly, and that the values end up in the right memory locations.
2. Write a program that implements `Push` and `Pop` subroutines. Test the subroutines by pushing and popping a few values, stepping through the code to make sure the stack works as expected.

You need to submit two source files for this task!

2.2.2 A simple RPN calculator (12 points)

We will now implement a simple RPN calculator that can only perform a single operation: addition.

We will use the `Input` instruction to let the user input a sequence of numbers and operators. To simplify the implementation, we will switch MARIE's input field to `Dec` mode, meaning that we can directly type in decimal numbers. But how can we input operators?

We will simply only allow **positive numbers** as input values, and use **negative numbers** as operators. For this first version, we will use -1 to mean addition.

So to compute $10 + (20 + 30)$ we would have to enter 10 20 30 -1 -1. Of course we could also enter 10 20 -1 30 -1 (which would correspond to $(10 + 20) + 30$).

The pseudo code for this simple calculator could look like this:

```
Loop forever:
  AC = Input          // read user input
  if AC >= 0:         // normal number?
    push AC
  else if AC == -1:   // code for addition?
    X = Pop
    Y = Pop
    Result = X+Y
```

```
    Output Result     // output intermediate result AND
    Push Result       // push onto stack for further calculations
```

Note that your calculator doesn't need to do any error checking, e.g. if you only enter a single number and then use the addition operator, or if you enter any negative number other than -1.

Implement this calculator in MARIE assembly. Use the Push and Pop subroutines from the previous task to implement the stack. It is a requirement that your calculator can handle *any* valid RPN expression, no matter how many operands and operators, and no matter in what order (up to the size of the available memory). I.e., the following expressions should all work and deliver the same result:

10 20 30 40 50 -1 -1 -1 -1

10 20 -1 30 40 -1 50 -1 -1

10 20 30 -1 -1 40 -1 50 -1

10 20 -1 30 -1 40 -1 50 -1

Document a set of test cases and the outputs they produce.

2.2.3 More RPN operations (12 points)

Of course the previous calculator is quite useless – if there is only one operation, we don't need to use RPN at all. Therefore, the next step is to extend your calculator with support for additional operations:

- Multiplication (code -2)
- Integer division (code -3)
- Square, i.e., compute x^2 for a value x (code -4)

The extended pseudo code would look like this:

```
Loop forever:
  AC = Input          // read user input
  if AC >= 0:         // normal number?
    push AC
  else if AC == -1:   // code for addition?
    X = Pop
    Y = Pop
    Result = X+Y
    Output Result
    Push Result
```

```
else if AC == -2:  // code for multiplication?
  X = Pop
  Y = Pop
  Result = X*Y
  Output Result
  Push Result
else if ...        // remaining cases follow the same structure
```

Note that by convention, $XY-$ means $X-Y$ in RPN, and similarly $XY/$ means X/Y.

As above, test your calculator using different test cases, which you should document in your written submission.

You can submit just one file for the entire extended version.