CS210 - Assessed Coursework Exercise #3

To be completed in your group

Available Marks: 50, plus 7 bonus marks

Submission deadline: 23:55 Sunday, 26th November, Week 10

Demo: Week 11, i.e. w/c 27th November

How to submit:

- MyPlace (*submission link will be made available on Monday, 20th November*).
- One submission per group!

General instructions:

- *You are advised to use Unix to compile & run your programs.*
- *All your programs must comply with the **C99** standard. Make sure that you use the appropriate gcc option to achieve this. Refer to the respective lecture slides if you do not remember how to do so.*
- ***Reminder regarding Labs:** Please remember that lab assistants are NOT there to help with your Assessed Coursework Exercise (ACE). You may ask a lab assistant technical questions regarding your ACE, e.g. "how can I print an integer on the screen?", "how can I read user input?" etc., but priority will be given to students working on the Practical. Do NOT ask a lab assistant to clarify parts of your ACE. This will be done in class (Q&A) and through the class forum.*

Objective

The aim of this Assessed Coursework Exercise is to implement & test a disassembler in C. A disassembler is a computer program that translates machine language into assembly language.

ACE#3 is to some extent open-ended; hence, you are expected to collect/refine requirements via the class forum dedicated to ACE#3 & during Q&A.

Background

Your disassembler should operate in the context of a conceptual computer (i.e. "ACE#3 Computer") that simulates many of the basic features found in modern electronic digital computers. "ACE#3 Computer" should have the following characteristics:

- Binary, two's complement representation
- Stored program, fixed word length
- Word (not byte) addressable
- 4K words of main memory (this implies 12 bits per address)
- 16-bit instructions, 4 for the opcode and 12 for the operand
- A 16-bit accumulator (AC)
- A 16-bit instruction register (IR)
- A 16-bit memory buffer register (MBR)
- A 12-bit program counter (PC)
- A 12-bit memory address register (MAR)
- An input register
- An output register

Primary Memory: 4K words of main memory. Each cell can hold machine instructions or data values.

The CPU: The CPU consists of a Control Unit, an Arithmetic Logic Unit (ALU), and a small set of temporary storage cells called registers:

- AC: The accumulator, which holds data values that the CPU needs to process.
- MAR: The memory address register, which holds the memory address of the data being referenced.
- MBR: The memory buffer register, which holds either the data just read from memory or the data ready to be written to memory.
- PC: The program counter, which holds the address of the next instruction to be executed in the program.
- IR: The instruction register, which holds the next instruction to be executed.
- InREG: The input register, which holds data from the input device.
- OutREG: The output register, which holds data for the output device.

The MAR, MBR, PC, and IR hold very specific information and cannot be used for anything other than their stated purposes. For example, you cannot store an arbitrary data value from memory in the PC. You must use the MBR or the AC to store this arbitrary value. In addition, there is a status or flag register that holds information indicating various conditions, such as an overflow in the ALU.

An "ACE#3 Computer" instruction is 16 bits wide and has two parts:

- The Operation Code/opcode (the most significant 4 bits) specifies what to do. All instructions have an opcode.
- The Operand (the least significant 12 bits) provides additional data for the opcode to operate on. Typically, the operand is a memory address where data to be operated on is stored or where the next instruction should be read, or it is an immediate (constant) numerical value to be operated on.

Assessed exercises

1. *Instruction Set Architecture (ISA) Design*

a. Complete the table below so that each mnemonic opcode has a unique binary representation.

Opcode		**Description**
Binary	**Mnemonic**	
	Load X	Load the contents of address X into AC.
	Store X	Store the contents of AC to address X.
	Subt X	Subtract the contents of address X from AC and store the result in AC.
	Add X	Add the contents of address X to AC and store the result in AC.
	Input	Input a value from the keyboard into AC.
	Output	Output the value in AC to the display.
	Halt	Terminate the program.
	Skipcond	Skip the next instruction on condition.
	Jump X	Load the value of X into PC.

(1 Mark)

b. Propose at least three more instructions. Provide an opcode (Binary & Mnemonic) and a description for each of the newly proposed instructions.

(2 Marks)

c. Provide a detailed description of how your Skipcond instruction should work.

(2 Marks)

2. *Auxiliary C functions*

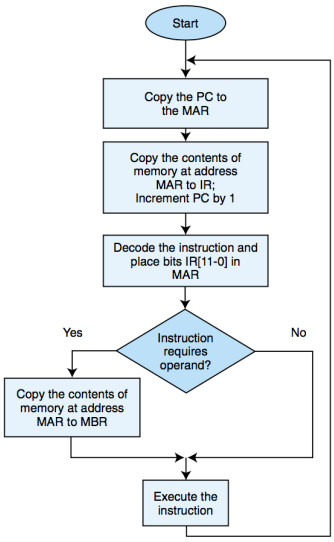
Write a C functions that implement the following functionality:

- Load some default content into the memory.
- Convert an integer from decimal representation to binary.
- Convert an integer from binary representation to decimal.
- Display the memory contents on the screen. *Your function must also display the respective memory locations.*
- Convert the memory contents to assembly language and display the assembly language on the screen. *Your function must display each assembly instruction on a new line.*
- Prompt the user to input the memory contents from the console (i.e. command line), read the input in and store it into the memory. *Your function must be able to stop reading user input in, e.g. you may require that the user types "stop" or "exit" etc.*
- Load the instructions from a file (i.e. rather than the console) and store its contents into the memory.

(15 Marks)

3. *The Fetch-Decode-Execute Cycle*

Write a C function that runs the program (i.e. instructions) stored in memory.



(10 Marks)

4. *Running your disassembler*

Write a `main` function that uses the above functions (and any additional functions you have implemented) to simulate the disassembler. The function should take one argument.

If no argument or a wrong argument is given, the `main` function should print an informative message about the expected arguments and terminate. If the argument is `-d`, the `main` function should load the default content into the memory. If the argument is `-c`, the `main` function should read user input from the console into the memory. If the argument is `-f`, the `main` function should read the contents of a file into the memory.

The `main` function should then execute the program stored in memory, starting at address location 0, and finally print the contents of the memory and the value of the Accumulator after execution on the screen. Depending on your implementation, the latter can be done in the function from Exercise 3.

(3 Marks)

Optional Task for a total of 7 bonus marks

Enhance your implementation so that it works for a "real-world" MIPS assembly language, i.e. as described in Chapter 2 of the following text:

> *"Computer organization and design: the hardware/software interface"*, Fifth edition.
> David A. Patterson, John L. Hennessy

The textbook is available as a free electronic resource through the University Library (you will need your DS username & password).

Minimum requirements:

- Implementation of all registers.
- Implementation of at least 15 instructions.

Partial marks are available.

(7 Marks)

Marking Scheme

- **Completeness** (has all required functionality been implemented?), and **correctness** (does everything work as specified?)

  As specified by the marks below each exercise

  [33 Marks]

- **Commenting** (i.e. is everything commented as it should?)

  Minimum requirements:

  - Description for each function
  - Description at the top of each C file (*template is provided on MyPlace*)
  - Consistency between descriptions and implemented functionality

  [1 Mark]

- **Style**

  Minimum requirements:

  - Code nicely laid out and formatted
  - Well-chosen variable/function names
  - Source code modularity
  - Use of constants instead of literal values
  - Quality of output: new lines, values separated from text, displayed info is accurate, meaningful error messages

  [1 Mark]

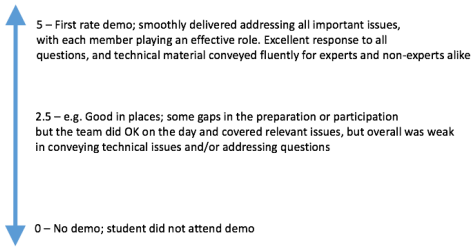
- **Demo**

  Demos will take place in Week 11 and will involve running your systems. Furthermore, you will be asked a number of questions regarding your work.

  Notes:

  1. Your demo will also be used as evidence to evaluate the quality of other deliverables

  2. This can be an individual mark if a student does not attend the demo



  [5 Marks]

- **Peer/Self Assessment (Individual mark)**

  Peer/Self Assessment form regarding the contribution of each group member. More details in Week 10.

  [10 Marks]

- **Optional Task**

  Completeness (has all required functionality been implemented?), and **correctness** (does everything work as specified?)

  As specified by the marks below the exercise

  Partial marks available

  [7 Marks]

What to submit:

A single compressed (e.g. .zip) file that contains:

1. All your source code (i.e. C files). NO executable files!
2. A document containing your answers to Exercise 1.

Penalties will be applied for late submission of assessed coursework according to the Departmental Policy.

Good luck!