Worksheet 1

An echo of Assembler

Marking scheme

- Task 1 - 20%
- Task 2 - 20%
- Task 3 - 20%
- Task 4 - 40%

All code should be submitted via a gitlab.uwe.ac.uk repo.

Each task should be documented in a README.md in the main repo, including screen shots of the code running, and details of how your implementation works.

After completing this worksheet you should be familiar with:

- Basic assembler programming
- Call C from assembler
- Translating C loops into assembler
- Using `make` to build programs

It is assumed that you are already confident in basic C programming and have completed worksheet 0. We will begin to look at how to program the machine from a lower level, moving on to implementing a very simple operating system in worksheet 2.

The deadline for completing this worksheet is 5th December, 2024, by 2pm.

> IMPORANT: All code must be submitted via gitlab, failure to provide a link to Gitlap repo will result in a mark of 0. No email or other form of submission is acceptable. There will be in class vivas week 17 and 18, i.e. weeks stating the 9th and 16th of December, 2024.

Before continuing please insure that you have watched the week 8 lecture, on introduction to assembler. You will also need to have the book PC Assembly Language on hand to help gain a full understanding of what needs to be done to complete the tasks.

To proceed

Before continuting with the worksheet it is recommended that you create a new repo on Gitlab and clone this on to the remote server, i.e. `csctcloud.uwe.ac.uk`. This will be the repo that you use for this worksheet and all work must be commited to Gitlab and then the URL submitted to Blackboard.

> IMPORTANT: Due to two-factor authentication you will need to create a Gitlab (SSH) token that is associated with your repo, and then use this to clone to the remote server. Please see the video for an example of how this works.

Once cloned on the server create a directory structure that is similar to the following:

```
repo-name
   README.md
   src
```

The `src` is where you will place your assembler and C files for this worksheet, and the README.md is where you will documented your work.

> The `.md` extension stands for Markdown and is a simple markup language for creating formatted text using a plain-text editor. An overview can be found on Blackboard here.

To complete this workshop your directory structure should look similar to the following:

```
repo-name
   README.md
   src
     asm_io.inc
     asm_io.asm
     XXX
   Makefile
```

Where XXX are the files that you will create as you proceed with the worksheet. The file `asm_io.inc` and `asm_io.asm` can be found on Blackboard alongside this worksheet. The `Makefile` will be created in the last task.

> To upload the `.asm` and `.inc` files to the remote server you will need to use a command line tool, such as sftp, or an FTP transfer program.

Task 1

In this task you are expected to implement the programs described in the lecture material for week 8. For the first program you should implement an assembler program that contains a function `asm_main`, as seen on slide 18 of the lecture slides, which adds two integers stored in global

memory, and then outputs the results using the provided function `print_int`. Again this is the program as descibed on slide 18 of the lecture material.

For this to work a C program with `main` function that calls the assembler function `asm_main` is required, you can use the following and place in `driver.c`:

```
int __attribute__((cdecl)) asm_main( void );

int main() {
  int ret_status;
  ret_status = asm_main();
  return ret_status;
}
```

Assuming that you have put the `.asm` and `.inc`, along with `driver.c` and `task1.asm`, in the src directory on the remote server, you can compile these different elements with the commands:

```
nasm -f elf task1.asm -o task1.o
nasm -f elf asm_io.asm -o asm_io.o
gcc -m32 -c driver.c -o driver.o
```

These three commands assembly, and in the case of `driver.c` compile and then assembly, the inputs files to object files. Object files are an intermidate representation of each file as x86 binary instructions. At this point we need to conbine them to produce an executable program that can be run. This can be achived with the following command:

```
gcc -m32 driver.o task1.o asm_io.o -o task1
```

> Note that as we are working on 64-bit machine, but producing 32-bit binaries, the argument `-m32` is required.

If everything ran correctly you should be able to see `task1`, by running the `ls` command, in that src directory. (The option `-o` specfices the name of the resulting output file.) Run the executable with the following command:

```
./task1
```

Ok, now take a look at the program on slide 22 and have a go at the same process from above, but this time adding the code to `task2.asm`. Build and run this program, documenting the results in your `README.md`.

Task 2

In this task we will consider loops and conditionals. Using the PC Assembly Language book for reference, sections 2.2 and 2.3, translate the following C programs to assembler, using the `main` function from `driver.c`.

Once you have spent a bit of time understanding how loops and conditionals can be implemented in assembler, write an assembler program that asks the user for their name and the number of times to print a welcome message. Test that the value is less than 100 and greater than 50 and then finally print out a welcome string that many times. Pretty an error message if the number if two large or small.

Write an assembler program that defines an array of 100 elements, initialize the array to the numbers 1 to 100, and then sum the that array, outputing the result.

Finally, extend the previous program so that it asks the user to enter a range to sum, checking the range is valid, and then display the sum of that range.

Task 3

Up until now each program has been built individually, even if the program source has not changed, and so on. If you have used tools such as Visual Studio or Xcode for Windows and MacOS, respectively, then you will already know that a more robust way of building programs is to use a tool to manage the process. An old but common tool for this is `make`, which uses a file, `Makefile`, to tell it what to do. Most often, the makefile tells `make` how to compile and link a program.

A simple makefile consists of “rules” with the following shape:

```
target … : prerequisites …
        recipe
        …
        …
```

A target is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as ‘clean’ (see Phony Targets).

A prerequisite is a file that is used as input to create the target. A target often depends on several files. This is most often a dependency, such as a source file when creating an object file and so on.

A recipe is an action that make carries out. A recipe may have more than one command, either on the same line or each on its own line.

As a simple example consider the case when we want to build a hello world program, with a single input file `hello.c`. The following makefile does this:

```
hello: hello.c
  gcc -o hello hello.c
```

Has a single rule that descibes how to make or build the executable, i.e. target, `hello``, given the input file, i.e. dependency,`hello.c. In general, `make rules are added to a file called`Makefile“‘ and can be run with the command:

`make`

This looks for `Makefile` and trys to build the first rule it comes across, alternatively the target can be given, e.g.:

```
make hello
```

It is possible to have a makefile have a rule to build more than one target at a time, for example, the following defines a rule `all` to build two targets:

```
all: hello goodbye

hello: hello.c
  gcc -o hello hello.c

goodbye: goodbye.c
  gcc -o goodbye goodbye.c
```

By putting the `all` first, it then becomes the default when `make` is run without any explict target.

To complete this task create a makefile, in the root directory of your repo, and add targets to build each of the executable programs from the previous two tasks. Have a default rule, ‘`all`, that builds all targets.

Task 4

Document your work in your README.md and submit to Gitlab. The readme should contain code snippets, demostrating understanding, screen shots, and so on. It should not contain complete files of code, as these will be in the repo itself.