Homework 3: Assembly
Due Nov. 7th 2022, 23:59

25 points

1 Objectives and Materials

In this assignment you will learn how to program low-level assembly code and how to interface that from C.

The matrerials you need for this lab are:

- Any computer with an ssh client installed.
- OR own computer running linux and ability to install software.

Complete the following tasks. Provide all source code, including appropriate inline comments. In the report explain how the code works and describe how you tested the validity of your code.

1.1 Preliminaries

Before starting with this lab please verify you have performed these preliminary steps:

- Log-in to the course server at `cs2ga3.cas.mcmaster.ca`. You have several options on how to run the emulation packages, but in case you go with the GTK visual interface, you need to make sure to ssh using X forwarding, as in

  ```
  ssh -X macid@cs2ga3.cas.mcmaster.ca
  ```

  where macid is your macmaster id. Note that you need to be on the university network or VPN in order to access the server. If you have not done so, you need to first reset your CAS password by visiting `https://www.cas.mcmaster.ca/reset`.

- Open a terminal and copy the folder with qemu virtual machine settings to your home folder:

  ```
  cp -r /usr/share/qemu_vms ~/
  ```

- Start qemu by executing:

  ```
  cd ~/qemu_vms
  ./start_qemu.sh
  ```

- Note: if you want to run the graphical interface use the script `start_qemu_gui.sh` instead.

- If you are running the GUI, then a window with an emulation of a Raspberry Pi ARM computer will show up. Note that this takes some time to boot up and that depending on your connection the forwarding of graphical interface may not be very responsive.
- When prompted log in with username `pi` and password `raspberry`
- In order to copy files from the emulated machine, you should use scp. The ssh service should already be running on the emulated machie. You can test that by trying to log in through another terminal window (NB: Make sure you are on the cs2ga3 server first! It will not work from your local terminal)

  ```
  ssh -p5022 pi@127.0.0.1
  ```

  To copy a file type:

  ```
  scp -P 5022 pi@127.0.0.1:/home/pi/YOUR_FILE DESTINATON
  ```

- NOTE: many of you will be running on the same server. To avoid accessing someone else's emulated machine, change the port 5022 from the line above **and** inside the file `start_qemu.sh` to a different suitably chosen number. Try something relatively unique, like your birthday month and day for example. You can see what ports are currently in use by:

  ```
  netstat --tcp --listen -n
  ```

  This is now your emulated machine, so feel free to change the password for the default udser `pi` (use the passwd command).
- In case the GUI is too slow or too cumbersome you may want to run in graphics-free mode. Do that by executing the script `start_qemu.sh` and then ssh into the emulated machine. Note that in order to kill the running emulation you have to follow a slightly arcane procedure: in the terminal running qemu press `Ctrl+B+C` to drop into the qemu shell. Then type quit and hit Enter.
- The rest of the lab assumes you are working within emulated machine.
- The command-line text editors `nano` and `vi` are pre-installed on the emulated machine. If you prefer to edit on your own machine you can do so and only copy the files when you are going to compile and test.
- The following links give some useful references for ARM assembly programming:
  - `https://azeria-labs.com/writing-arm-assembly-part-1/`
  - `https://thinkingeek.com/arm-assembler-raspberry-pi/`
- Note that you only have 32MB of free disk space on the emulated machine, so use it wisely and sparingly.

- Finally, in case you want to work entirely on your own machine, head over to `https://www.qemu.org/` and follow the instructions to installing qemu on your own computer. Then use scp to copy the files from inside the folder `qemu_vms` to your local computer and proceed from there.

1.2 Example

In order to create executable assembly programs, your code needs to return control of the processor back to the OS in a correct fashion. Similarly, there are a number of setup tricks needed when linking your main function. We will clear these up with an example.

```
@ ------------------------------------------------
@   Data Section
@ ------------------------------------------------
.data
string: .asciz "\nHello World!\n"
@ ------------------------------------------------
@   Code Section
@ ------------------------------------------------
.text
.global main
.extern printf

main:
  @ push the return address (lr) and a
  @ dummy register (ip) to the stack
  push {ip, lr}

  @ load the address of variable string into r0
  ldr r0, =string
  @ branch to printf, passing r0 as argument
  bl printf

  @ pop the return address into the program counter
  pop {ip, pc}
```

To compile the program above you can use the asembler `as` to generate an object file. You would then use the linker `ld` to link the object file to the standard C library that implements `printf`. We will instead rely on `gcc` to do the linking for us, which lets us abstract away what happens between the main function and the entrypoint of the program in `_start`. You can either look at the example scripts `compile.sh` and `link.sh`, or directly compile an executable using gcc:

```
gcc hellow_world.S -o hello_world
```

```
./hello_world
```

Try out the example code and verify that you can compile and execute the program.

2 Task 1: Shifting Integers (10 points)

In this task we will practice mixing C and assembly and see how to shift integer numbers. Submit the files `int_out.c main.c problem_1.s` as well as a description of what you did and what outputs you obtained as a PDF report. Follow these steps to complete the task:

- Write a C function `void int_out(int a)` which takes an integer argument a and prints the value in hexadecimal notation. Put the function in a file called `int_out.c` and compile it into an object file `int_out.o`.
- Write a separate C program `main.c` that contains a `main` function. In the `main` function call `int_out` on a range of inputs and verify that it produces the expected output (by manually comparing it to what the function should print).
- In a separeate file `problem1.s` write an assembly program which loads an immediate value of 4 into a register and then calls the external function `int_out` to print it.
- Compile the assembly program to an object file and link the two object files (assembly and C) into an executable. Verify that the output is as expected.
- Next, modify `problem1.s` to also load the integer `0xBD5B7DDE` and print it using `int_out`. Verify the output is as expected. *Note:* the value `0xBD5B7DDE` is too large to be specified as an immediate value. You need to define a new variable in memory in the data section of the program and load the value into a register.
- The value `0xBD5B7DDE` is a signed integer constant. Calculate in theory what would be the result of shifting this constant 1 bit to the right. Assume we are using two's complement integers and that sign extension is implemented correctly. Verify your theoretical result by modifying `problem_1.s` to shift the integer one bit to the right and print out the result. *Hint:* check the documentation of the assembly instruction `asr`.

3 Task 2: Assembly in C (10 points)

In this task we will practice calling assembly routines from within C. Submit the files `xor.c axor.s` and describe the steps and results in the PDF report. Follow these steps:

- Write a C funxtion `int xor(int a, int b)` which returns the bit-wise exclusive or of two integer arguments.

- Write a main function to test your `xor` implementation and verify that the output is as expected.
- In a new file `axor.s` write an assembly version of the `xor` function. Your function should accept two arguments in registers $r0$ and $r1$ and return the result in register $r0$. Code the function from scratch, you are not allowed to use gcc to generate the assembly code for you. Compile the function into an object file `axor.o`.
- Declare `axor` as an external function in your C file `xor.c`. Call `axor` from inside the `main` function and compare the results to the C code `xor` implementation to verify that the assembly code works as expected. Test your implementation over a range of random inputs. *Note:* you will need o link your C code to the object file `axor.o`.

4 Task 3: Theory (5 points)

The remaining questions **do not** refer to the ARM architecture above, but are general.

a: What is the difference between horizontal and vertical microcode?

b: Explain the different ways through which operands can be addressed in a computer architecture. Describe which modes offer the fastest access.

c: Explain how a two stage assembler processes code to produce a binary executable program. Provide an example.