COMP 30080: Computer Systems

2021 / 2022

Assignment 3

Pavel Gladyshev

UCD School of Computer Science,
University College Dublin,
Belfield, Dublin 4.

Introduction

There are five assignments in total. Assignments should be submitted using Moodle via the 'Assignment submission' links. The assignment deadlines are provided in Moodle. Roughly speaking, one assignment is to be submitted every two weeks of term. Moodle will allow submissions up to 2 weeks late. However, late submissions incur penalties according to UCD policy unless a medical certificate or similar is submitted to the lecturer or the UCD Science Programme Office. There is a link to the UCD policy document in Moodle.

Assignments do not have to be done during lab times. However, Demonstrators are only available during lab times. A record of attendance is taken during lab times.

Assignment 1 is not assessed. Students should compare their solutions with the model solution provided. All assignment marks and feedback will be available in Moodle.

For all assignments:

- Submit the assembly programs that you design as .asm files. Use your student number and question number as file names, e.g. 12345678_a1q1.asm
- Submit the circuits you design as .circ files. Use your student number and question number as file names, e.g. 12345678_a1q1.circ
- For each assignment, submit an accompanying report (.doc or .pdf) which includes screenshots showing verification of the submitted program, any workings and the answers for written work questions. Use your student number and assignment number as the file name, e.g. 12345678_ass1.doc
- Make sure that your name and student number are visible in the assembly files, circuits, and report.
- If necessary, scan (or photograph) any handwritten work into a file for submission. Submit the graphics file (.gif or similar) or include the scan in the report.
- Submission is via Moodle. Moodle requires a single file that should be a .zip of all individual files. Use your student number and assignment number as the file name, e.g. 12345678_a1.zip

PLEASE NOTE, THE REPORTS MUST BE IN WORD OR PDF FORMAT, THE ASSEMBLY PROGRAMS MUST BE IN ASM (TEXT) FORMAT AND THE SUBMISSION MUST BE IN ZIP FORMAT. OTHER FORMATS OR CORRUPT FILES WILL NOT BE MARKED

The marking scheme for most questions is: 0=nothing done; 1-3=partial working; 4=working; 5=working and elegant.

Assignment 3: Mixing C & Assembly

In this assignment we will study the use of GNU C compiler and binary utilities for MIPS32 Little Endian (MIPSEL) CPU to write programs that combine C and Assembly programming. To perform this assignment you will need a virtual machine (or a physical PC) running recent version of Ubuntu Linux. It is possible to use Windows Subsystem for Linux, if you are using a Windows PC running Windows 10 or 11. All tasks specified in this document had been performed on the **mips-tools** virtual appliance.

1) Downloading and installing GNU C compiler and related utilities for MIPS little endian architecture

In this assignment you will need to install GNU C compiler for MIPS32 Little Endian CPU, QEMU userland emulator, and multi-architecture GNU debugger. Start a terminal window and run the following commands:

```
sudo apt-get install qemu-user qemu-system-mips
```

```
sudo apt-get install gcc-mipsel-linux-gnu gdb-multiarch
```

Please study the sample C and assembly programs provided on CSMoodle alongside this assignment.

For additional information about interfacing C and Assembly please refer to Chapters 7 and 8 of MIPS Assembly Language Programmer's Guide (https://csmoodle.ucd.ie/moodle/mod/url/view.php?id=1617) .

To use GNU debugger, open two terminal windows and navigate each of them to the sub-directory containing your program

1. Start MIPS simulator in one window using command:

   `qemu-mipsel -g 1235 -L /usr/mipsel-linux-gnu/ yourExecutableProgram`

   here -g 1235 option specifies TCP port waiting for GNU debugger connection.

2. Start GNU Debugger in the other window using command:

   `gdb-multiarch yourExecutableProgram -ex "target remote :1235"`

3. Once GDB starts switch to the interactive "Text User Interface" mode by entering command

   ```
   tui enable
   ```

   If the source code and debugging information is available, enable display of program source code and CPU registers by entering commands:

   ```
   layout regs
   ```

   alternatively, if the source code and/or debugging information is not available, display program disassembly and CPU registers by entering commands:

   ```
   layout asm
   layout regs
   ```

4. Step through the program **s**ingle **i**nstruction at a time using command

   ```
   si
   ```

   or **s**ingle line (of the source code) at a time using command

   ```
   s
   ```

   to run (continue) program until completion (or the next break point) use command

   ```
   c
   ```

You can set breakpoints using command 'b' please refer to this and other commands' description in the official GDB user manual (https://sourceware.org/gdb/current/onlinedocs/gdb/) and various online tutorials.

2) Assignment questions

In this assignment you will need to provide solutions for the following three questions:

Q1. Write a pure assembly language program **lowcase.s** that keeps reading bytes from standard input one by one using Linux read() system call[1] ($v0 = 4003) until the standard input is exhausted. Assume that each byte read from the standard input represents an ASCII character code. If the code is between 65 ('A') and 90 ('Z') inclusively, **lowcase.s** must convert it to the lower case by adding 32. The character code – lower-cased or not – must then be sent to the standard output using Linux write() system call ($v0 = 4004). Once the standard input is exhausted, the program must terminate gracefully using Linux exit() system call ($v0 = 4001).

It must be possible to assemble lowcase.s using the following commands:

```
mipsel-linux-gnu-as -o lowcase.o lowcase.s
```

```
mipsel-linux-gnu-ld -o lowcase lowcase.o
```

and run it using the following command

```
qemu-mipsel -L /usr/mipsel-linux-gnu/ lowcase
```

[5 Marks]

Q2. Write an assembly language implementation of the function `int rotStr(char *str, int n)`, which takes a NULL terminated string `str` and **rotates 8-bit code of each character in place** by abs(`n`) bits to the left or to the right[2]. The direction of rotation is determined by the sign of `n` : if `n` is positive, the rotation is to the left, if `n` is negative the rotation is to the right. The examples of rotations are given below. The function must return the value of abs(`n`).

You can use program `rotate.c` provided with this assignment to experiment with your implementation. Please save your assembly language version of `rotStr()` into the file `rotstr.s` , then you should be able to compile both files using the following command:

```
mipsel-linux-gnu-gcc -g3 -static -o rotate rotate.c rotstr.s
```

It will produce executable file `rotate` that can be run using the following command

```
qemu-mipsel -L /usr/mipsel-linux-gnu/ rotate
```

or debugged with GDB as explained in the previous sub-section.

Hint: Consider using **lbu** instruction instead of lb to load individual character codes.

Some examples of rotations:

1. **Example 1: "Hi" rotated two bits to the left**

   NULL terminated string: **"Hi"**

   Hexadecimal codes of "Hi": **0x48 0x69**

   Same in binary:
   01001000 01101001

   `Str` **with each byte rotated** to the **left** by two bits (`n`=2):
   00100001 10100101

   The resultant content of str in hexadecimal (again, assuming little-endian representation of multi-byte integers) : **0x21 0xA5**

   The text of the resultant NULL terminated string after rotation (assuming ISO 8859-1 encoding for character codes 128-255): **"!¥"**

[1] A description of read() system call can be found at https://man7.org/linux/man-pages/man2/read.2.html
[2] for an explanation of binary rotation see https://en.wikipedia.org/wiki/Bitwise_operation#Rotate

2. **Example 2: “covid” rotated one bit to the right**

   NULL terminated string: **“covid”**

   Hexadecimal codes of “covid”: **0x63 0x6F 0x76 0x69 0x64**

   Same in binary:
   01100011 01101111 01110110 01101001 01100100

   `str` **with each byte rotated** to the **right** by 1 bit (`n`=-1):
   10110001 10110111 00111011 10110100 00110010

   The resultant content of str in hexadecimal (again, assuming little-endian representation of multi-byte integers) : **0xB1 0xB7 0x3B 0xB4 0x32**

3. **Example 3: “” rotated 40000000 bits to the right**

   The resultant string should still be empty.

4. **Example 4: “” rotated 50000000 bits to the left**

   The resultant string should still be empty.

5. **Example 5: “madam” rotated 0 bits to the left**

   The resultant string should still be **“madam”** or **0x6D 0x61 0x64 0x61 0x6D**

[15 Marks]

(20 marks in total)