CSC B58 - Lab 8

Introduction to Assembly

1 Learning Objectives

The purpose of this lab is to learn how to write some assembly language program and also to use the MARS simulator.

2 Marking Scheme

This lab worth 4% of your final grade, and also includes a bonus part that worth 1% of your final grade. You will be graded out of 8 marks for this lab, as follows:

- Prelab (including Simulations and Diagrams)
- Part I : 2 marks
- Part II : 3 marks
- Part III: 3 marks

Preparation Before the Lab

You are required to complete all Parts of the lab by writing and testing your assembly code in MARS.

You are required to code and test all the Parts of the lab and demonstrate them to the TAs.

3 Introduction

From this week, we are leaving Logisim behind and starting something new – programming in assembly. We will be learning about a specific architecture called MIPS. MIPS is a RISC (Reduced Instruction Set Computer) family of processors originally introduced in the early 1980's. The MIPS assembly language is well-known for being concise and logically structured, and because of their simplicity and energy efficiency, MIPS processors are still in use as embedded processors in devices like cell phones and portable game players.

Since the computers we use today are not MIPS machines, we will need to simulate one. The simulator we are using is called MARS, and it is written in JAVA. You can easily download it from the MARS web page: `http://courses.missouristate.edu/kenvollmar/mars/download.htm`.

Note that some students encountered a bug that sometimes causes MARS to freeze when debugging code. You can download a fixed version from Quercus under Modules - Assembly Project - Updated MARS jar.

Note: Take your time on this lab and future lab to get used to MARS, since we'll be using it in all of the remaining labs as well as the project. Play with the code, and also run the lecture and review code available on Quercus. As always, you will need to submit your solution code (as `.s` or `.asm` files) to Quercus before the start of your practical session.

An Assembly Program

Every assembly program we write will look very similar. Here is an outline of a typical program:

```
# Anything after # is a comment. Document your code!
# Write your name and UTorID at the top of your lab submissions!

.text   # This tells the assembler that the following are instructions.

.globl main
main:   # This is a label ()which is simply a name for an address).
        # The main label tells the OS where to start execution.

        # Write your main program code here...

        # This tells the OS when your program ends.
        li $v0, 10  # Uses system call 10 to exit program
        syscall     # More on this in future weeks

# Write functions here...
```

Every program is separated into two parts: a data section and a code section indicated by `.text`. We will learn about the data section in future weeks. Your code should be placed in the main block. The `main` label specifies where the program's main function starts (where MARS should start executing code). You may create other labels as you like; each label is used to name a specific line of code so that you can branch or jump to it. We'll use labels a lot when we implement control flow like branches, loops, and function calls.

The keyword *syscall* asks the operating system (or the simulator, in this case) to intervene to run a privileged instruction. We will learn about it in future weeks. Here we are using it to tell the OS to end the program

Note: Make sure to always document your assembly code using comments.

MIPS Reference

Before you become an assembly programming guru, you almost always need a reference card at hand, since some instructions implicitly work with specific registers (e.g., *syscall* implicitly uses *$v0* and *$a0*); and some register values are assigned to specific operations (like different system calls), which we don't want to memorize.

Download the **MIPS reference card from Qerucus** – it has the basic information that you need while programming MIPS assembly. You will find the reference card, as well as other useful Assembly resources such as detailed MIPS assembly references on Quercus under Modules - Assembly Project - MIPS Assembly Resources.

MARS Basics

Copy or type out one of the lecture MIPS assembly programs from the lecture slides. Load the file into MARS and then "Assemble" it (under the "Run" menu, on the toolbar, or using the F3 key). You cannot run your code until it assembles correctly. To test this, add a few random characters to the code in the "Edit" window, then try to assemble again. You will see an error message printed in the "MARS Messages" window at the bottom of the screen.

Now, take a moment to familiarize yourself with the layout of MARS. Fix the code, and then re-assemble it. You'll be taken to an "Execute" series of windows.

- The top left window contains the text segment – the code in your assembly program. That window provides you with the addresses of the various instructions, the machine-code value that is stored at that address, the assembly equivalent, and finally the line of code in the source file that generated that assembly instruction. You'll see that some lines of source code generate multiple lines of assembled code. In some cases, the original assembly instruction is a pseudo-instruction. In other cases, extra operations are required because of the simulated machine's architecture (hardware).
- The middle left window (above the message window) contains a window into memory. This will be more useful in future weeks when we start defining variables. Initially, the memory window is set to shopw the data segment – the variables defined in your assembly program. However, the pulldown bar lets you select other segments including the heap or stack. Note that you can also scroll through memory and select how the values are interpreted. ASCII mode is particularly useful for checking strings.
- The pane on the right contains information about all of the registers in the machine. By default, it shows the registers of the processor: register file, lo, hi, and PC. "Coproc 1" (co-processor 1) is the floating point unit, and we will not use it in this course. "Coproc 2" supports the execution of interrupts, which we may briefly touch on in future weeks.

Now, step through the code line by line. You can also execute the entire program, if you just wish to see the result, or step backward, if you wish to investigate a particular instruction more carefully. Take a few moments to familiarize yourself with how MARS uses highlighting to indicate the currently executing instruction and the registers that are being accessed. You can reset the simulator through the "Run" menu or by clicking on the "rewind" button in the toolbar.

Part I - Compute the Number of Solutions to a Quadratic Equation

Let a, b, c and x be some integers. Consider the quadratic equation: $ax^2 + bx + c = 0$. The number of real solutions for this equation can be found using the discriminant Δ:

$$\Delta = b^2 - 4ac\,.$$

If $\Delta < 0$ there are zero real solutions for x, if $\Delta = 0$ there is one solution, if and $\Delta > 0$ there are two.

Write an assembly program that receives the coefficients a, b, c in registers `$t0`, `$t1`, `$t2` (respectively) and computes the number of solutions. At the beginning of your program, initialize these registers to some values. When your program ends, `$t5` should contain the number of solutions: 0, 1, or 2. You can assume results fit in 32 bits. **Test your program well!**

Submit a file `lab08a.asm` that implements the above. Make sure your code is well-documented. **(PRELAB)**

Write your name and UTorID at the top of the file as a comment! **(PRELAB)**

Part II - Use Euclid's Algorithm to Compute Greatest Common Divisor

Let a and be b be two positive, non-zero integers. Their *greatest common divisor* is the largest positive, nonzero integer that divides both a and b without leaving a remainder. For example if $a = 46$ and $b = 20$ their greatest common divisor is $gcd(46, 20) = 2$ since $46/2 = 23$ and $20/2 = 10$ but no larger integer divides both. Other examples: $gcd(60, 15) = 15$, $gcd(16, 60) = 4$, $gcd(60, 48) = 12$, $gcd(25, 64) = 1$.

Euclid provided a simple iterative algorithm to compute the GCD, given here in Python:

```
while a != b:
    if a > b:
        a = a - b
    else:
        b = b - a
```

At the end of the run, a will contain the GCD.

Implement Euclid's algorithm in assembly. Your program will first initialize `$t0` and `$t1` to two positive numbers, and then implement the algorithm above. When your program ends, the GCD of `$t0` and `$t1` should be stored in `$t9`.

Submit a file `lab08b.asm` that implements the above. Make sure your code is well-documented. **(PRELAB)**

Write your name and UTorID at the top of the file as a comment! **(PRELAB)**

Hints:

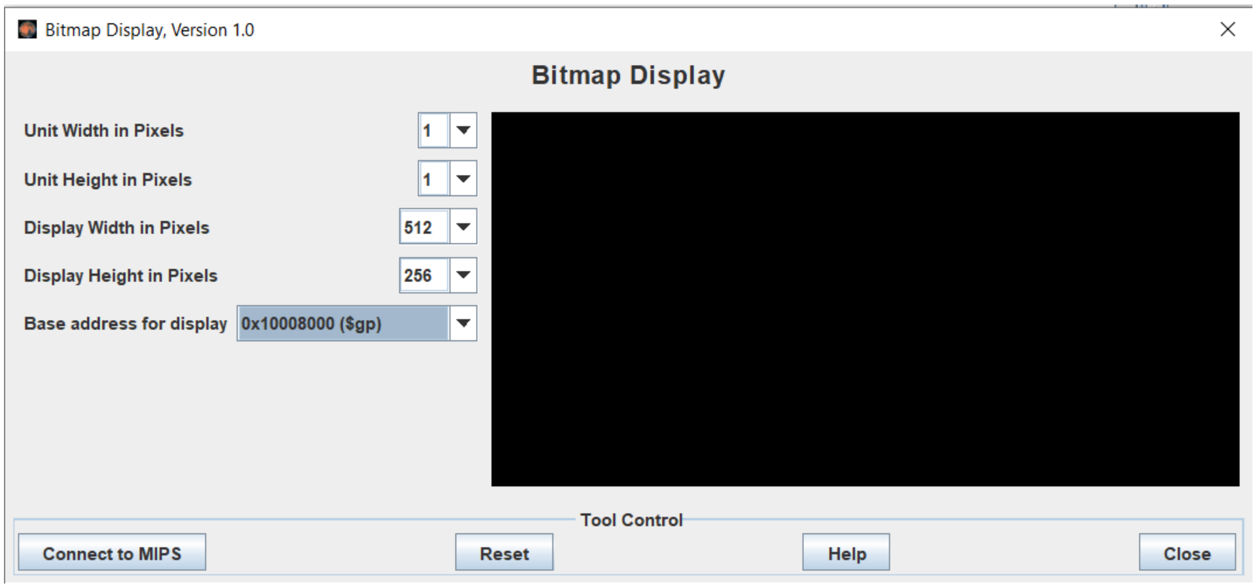
- First test the algorithm in a high level language (e.g., Python) so you can understand how it works.
- You will need a loop, and also some conditionals. This means several branches, jumps, and labels.

Part III - Create your Assembly Game Avatar

The bitmap display is a 2D array of "units", where each "unit" is a small box of pixels of a single colour. These units are stored in memory, starting at an address called the "base address of display". The Bitmap Display window allows you to specify the width and height of these units, the dimension of the overall display and the base address for the image in hexadecimal (see the Bitmap Display window in Figure 1 below).

The 2D array of units for this bitmap display are stored in a section of memory, starting at the "base address of display" (memory location `0x10008000` in the example above). To colour a single unit in the bitmap, you must write a 4-byte colour value into the corresponding location in memory.

- The unit at the top-left corner of the bitmap is located at the base address in memory, followed by the rest of the top row in the subsequent locations in memory. This is followed by the units of the second row, the third row and so on (referred to as row major order).
- The size of the array in memory is equal to the total number of units in the display, multiplied by 4 (each colour value is 4 bytes long).



Unit Width in Pixels: 1
Unit Height in Pixels: 1
Display Width in Pixels: 512
Display Height in Pixels: 256
Base address for display: 0x10008000 ($gp)

Figure 1: Bitmap Display Window

- For example, in the configuration in the above image, each unit is 8 pixels x 8 pixels and there are 32x32 = 1024 units on the display (since 256/8 = 32).
- This means that the unit in the top-left corner is at address `0x10008000`, the first unit in the second row is at address `0x10008080` and the unit in the bottom-right corner is at address `0x10008ffc`.

• Each 4-byte value stored in memory represents a unit's colour code, similar to the encoding used for pixels in Lab 7. In this case, the first 8 bits aren't used, but the next 8 bits store the red component, the 8 bits after that store the green component and the final 8 bits store the blue component.

  - For example, `0x000000` is black `0xff0000` is red and `0x00ff00` is green. To paint a specific spot on the display with a specific colour, you need to calculate the correct colour code and store it at the right memory address (perhaps using the sw instruction).

When setting up the Bitmap Display dialog above, you must change the "base location of display" field. If you set it to the default value (static data) provided by the Bitmap Display dialog, this will refer to the "`.data`" section of memory and may cause the display data you write to overlap with instructions that define your program, leading to unexpected bugs.

Bitmap Display Starter Code

To get you started, the code below provides a short demo, painting three units at different locations with different colours. Understand this demo and make it work in MARS.

```
# Demo for painting
#
# Bitmap Display Configuration:
# - Unit width in pixels: 8
# - Unit height in pixels: 8
# - Display width in pixels: 256
# - Display height in pixels: 256
# - Base Address for Display: 0x10008000 ($gp)
#
.data
displayAddress: .word 0x10008000
.text
lw $t0, displayAddress # $t0 stores the base address for display
li $t1, 0xff0000 # $t1 stores the red colour code
li $t2, 0x00ff00 # $t2 stores the green colour code
li $t3, 0x0000ff # $t3 stores the blue colour code

sw $t1, 0($t0)  # paint the first (top-left) unit red.
sw $t2, 4($t0)  # paint the second unit on the first row green. Why $t0+4?
sw $t3, 128($t0) # paint the first unit on the second row blue. Why +128?
Exit:
li $v0, 10 # terminate the program gracefully
syscall
```

1. Implement the given starter code in MARS and paint the three units. **(PRELAB)**
2. Draw an avatar for your game's player on the Bitmap Display. Your game avatar should look like an aircraft. Some samples are shown in Figure 2. (You are allowed to change your game avatar for the project at a later date.) **(PRELAB)**

Summary of tasks

1. Write a program `lab08a.asm` that computes the number of solutions to a quadratic equation.
2. Write a program `lab08b.asm` that computes the GCD using Euclid's algorithm.
3. Write a program `lab08c.asm` that drawes .
4. Make sure to document your assembly code using comments, and to write your name at the top.
5. Submit the programs to Quercus.
6. Demonstrate your solutions to your TA.

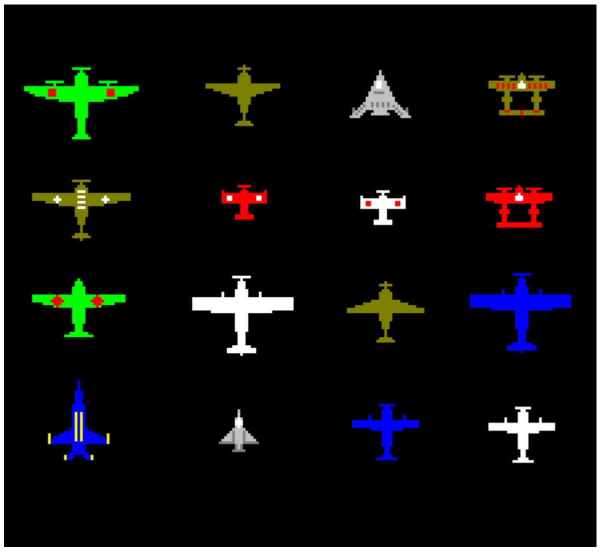


Figure 2: Game Avatars - Planes