New York University Abu Dhabi Department of Engineering

Computer Organization and Architecture
ENGR-UH 3511

Lab 2

MIPS assembly, recursion and the SPIM simulator

Introduction

This lab will introduce basic concepts of programming in MIPS assembly. This will be accomplished using a program that emulates a MIPS CPU and offers runtime support and information, SPIM.

Setup

In order to get SPIM for your computer, you should download and install it with apt:

```
sudo apt install spim
```

SPIM basics

To invoke the SPIM simulator you just have to type `spim` in the terminal. After SPIM has been initialized you may load an assembly program. To accomplish that, you use the following command:

```
load "myprogram.s"
```

Try loading the ”hello.s” program you may find included in a the tarball attached at NYU classes Lab 2 assignment section.
In order to perform a full run of the program you should type:

```
run
```

SPIM also offers step-by-step execution of any program so you will be able to observe each executed instruction which will be a great help when trying to debug your program. This is accomplished by typing:

```
step
```

In addition to step-by-step execution, a SPIM user has also access to every register available on the simulated processor. This will further assist you in debugging your program, as well as gain additional information such as how negative numbers are expressed in a computer system. To get access to any register type :

```
print register
```

for example `print $v0` or `print $2`. Registers are identified either by name (e.g. v0) or by number. You may find a complete list of registers available in MIPS architectures with their names and numbers listed in this site :

```
https://en.wikibooks.org/wiki/MIPS_Assembly/Register_File
```

In addition to printing registers, the `print` command can also be used to output the content of memory addresses. This is very useful in order to have full control of all data in your program in conjunction with the print register command. The print address command is similar to the previous one:

```
print address
```

MIPS Assembly

Program Structure

An assembly program is just a simple text file with program code and data declarations (always save the program with the .s extension). Data declaration precedes program code.

- Data Declarations
  - Placed in the section of the program declared with the assembly directive `.data`
  - Used to hold variables used in the program; data is stored in the main memory (RAM)
- Code
  - Placed in the section of the program declared with the assembly directive `.text`
  - Contains instructions
  - Starting point of the program is denoted with the label `main:`
- Comments
  - Comments are preceded by the number sign (#)

Data Declarations

Format: `name: .storage_type value(s)` :Allocates memory space for variables of the specified type and value and ties their addresses to their name label, so they can be used by just referencing their name.

- Labels are always followed by colon
- Storage types are always preceded by a dot (.)
- Multiple values must be separated by commas

Examples can be found in the attached tarball.

Load/Store Instructions

The following commands provide access to the main memory (RAM) to write (store) or read (load).
Load:
`lw $Rd, RAM_source` Copies a word (4 bytes or 32 bits in a 32-bit computer) from a source location in RAM to $Rd (destination register)
`lb $Rd, RAM_source`: Copies a single byte from a source location in RAM to $Rd and extends the sign to higher-order bits.
Store:
`sw $Rs, RAM_destination`: Stores a word (4 bytes or 32 bits in a 32-bit computer) from $Rs (source register) to a destination location in RAM
`sb $Rs, RAM_destination`: Stores a single byte from $Rs to a destination location in RAM. A sample lw/sw program is included in the attached tarball.

Indirect and Base Addressing

Load Address:
`la $Rd, RAM_source`: Copies the RAM location of a variable stored in RAM (presumably defined in the program) into $Rd.

Indirect Addressing:
`lw $Rd, ($Rs)`: Loads a word at memory address contained in $Rs register into the $Rd.
`sw $Rs, ($Rt)`: Stores a word from $Rs to a memory location addressed by $Rt
Based or Indexed Addressing:
`lw $Rd, number($Rs)`: Loads a word at memory address ($Rs + number) into $Rd
`sw $Rs, number($Rt)`: Stores a word from $Rs to a memory location addressed by ($Rt + number). A sample indirect addressing program is included in the attached tarball.

Arithmetic Operations

Arithmetic operations are the "engine" of assembly programming. Arithmetic instructions deploy usually 3 operand, all of which are either registers or immediate numbers. No RAM or indirect addressing is possible in MIPS architecture.
Register operands:
`add $Rd, $Rs, $Rt`: Performs addition of the value stored in $Rs to $Rt and stores the result to $Rd. Other arithmetic operations include `sub`, `mult` and `div`. Logic operations include `and` `or` and `xor`.
Register/Immediate operands:
`addi $Rd, $Rs, Immediate_Operand`: Performs addition of the value stored in $Rs and the value denoted by the immediate operand (a simple number), storing the result in $Rd. Immediate logic operations include `andi`, `ori` etc.
Unsigned Arithmetic:
`addu $Rd, $Rs, $Rt`
`addiu $Rd, $Rs, Immediate_Operand`
The same operations as above, but the operands are unsigned. This means that the MSB is counted as part of the number and not the sign notation. In addition, if the number is less than 32-bit, 0s are used to extend the number rather than its sign value

Flow Control Structures

These instructions facilitate flow control, in other words changing the normal program flow (program counter increasing) to jumping backwards or forward, directly or based on a condition. These instructions are the building blocks of well known structurs such as `if...else`, `for` and `while` loops etc.

Branches

`beq $Rs, $Rt, target`: Branches to (pc+target) if \$Rs and \$Rt registers are equal.
`bne $Rs, $Rt, target`: Branches to (pc+target) if \$Rs and \$Rt are not equal.
`blt $Rs, $Rt, target`: Branches to (pc+address) if \$Rs is less than the \$Rt.
`ble $Rs, $Rt, target`: Branches to (pc+target) if \$Rs is less or equal to \$Rt.
`bgt $Rs, $Rt, target`: Branches to (pc+target) if \$Rs is greater than \$Rt.
`bge $Rs, $Rt, target`: Branches to (pc+target) if \$Rs is equal or greater than \$Rt.

Jumps

`j target`: Jumps unconditionally to the target address
`jr $Rs`: Jumps unconditionally to the address contained in \$Rs.

Miscellaneous Instructions

Move Instruction

`move $Rt, $Rs`: Copies the contents of register \$Rs to \$Rt

Load Immediate

`li $Rd, immediate`: Loads an immediate value to register \$Rd.

There is a full manual of MIPS assembly commands in your textbook appendix.

Syscalls

System calls are the means for a computer program to make a request to the system program (usually the OS) such as read a file, produce an output, terminate the current process etc. Essentially they are the interface between the program and the system. In MIPS such an invocation to the system is being performed with the instruction `syscall`. When `syscall` is invoked, the operating system will read the `$v0` register and based on the value within it will determine the nature of the system call (e.g. print, read, terminate etc)

At the resources folder of NYU Classes under Lab Manuals folder you may find `SPIM_instruction_set` PDF file which contains a list of system services along with their corresponding system call code and the specific register that will be used to pass an argument (e.g. the string to be printed) or return the result (e.g. the character that was read from the keyboard) that are available for the SPIM simulator [1].

Functions

Functions in MIPS assembly are segments of code, conventionally implementing a specific function, that are preceded by a specific label and are located either before or after the main function. Function *call* is performed with the instruction `jal function_name` which jumps to the label `function_name` while saving the value of the program counter pointing to the next instruction after the `jal` (PC + 4) to the reserved register `$ra`.

Conventionally, values that are held in registers which will be used in the function as well (e.g. using \$a0 in the main program as well as the function) are pushed into the *stack*. The *stack* is a LIFO (Last In First Out) struct, the beginning of which is held in a specific register called *stack pointer* (`$sp in MIPS`). In order to push values in the stack, space must first be allocated, which is performed by decreasing the stack pointer by the number of stored words multiplied by 4 (e.g. `addi $sp, $sp, - 12` for 3 values). Then the values can be *pushed* (stored) into the stack in successive addresses through `sw` and then *popped* (retrieved) through `lw`. After the stored values are popped, the stack pointer must return to its original value, effectively freeing the previously allocated space.

In case of nested functions, that is calling a function within a function, the return address register must also be pushed into the stack before performing `jal`. This is necessary since `jal` will overwrite the previous return address with the new one, so the original return point (e.g. to the main function) will be lost. This is especially crucial to recursive functions.

[1]SPIM is an emulator for the MIPS architecture that also includes a peudo-OS that will handle a subset of the available system calls

C to MIPS Assembly

In this part we will use GCC to compile a C program for the MIPS architecture. To be able to accomplish that, GCC must be upgraded to support this compilation for another architecture (also called cross-compilation). This can be achieved with this command:

```
sudo apt install gcc-mips-linux-gnu
```

Following the installation everything is set. To compile a C program into MIPS assembly, the procedure is mostly the same, instead of `gcc`, `mips-linux-gnu-gcc` must be used. In addition to that, instead of compiling a C program into an executable, we will compile it into assembly source code that we then can load to the SPIM simulator. This can be achieved with the flag `-S`. A sample use of this is as follows:

```
mips-linux-gnu-gcc -S myprogram.c
```

This command will produce a file named `myprogram.s` which will contain MIPS assembly source code that implements the same program as the original C source code.

In the attached tarball you will find a sample C program to compile.

In Lab Exercise

Implement Hamming distance algorithm.

`https://en.wikipedia.org/wiki/Hamming_distance`

Assignment

Deliverables

For this Lab homework you will have to write a program implementing the following algorithms in MIPS assembly:

1. An assembly program for Matrix Multiplication between two 2x2 matrices (separate source file).
2. An assembly program implementing a recursive algorithm that calculates a power of three ($3^n, 10 \leq n < 20$) (separate source file).
3. A report which should include screenshots, code snippets etc in order to back up your source file implementation.

Assignment guidelines

- The input number (your chosen power of 3) will be given to the program as an input from the keyboard and the final result will be printed on the screen.
- You may find the algorithm in pseudo code or a high-level language online.
- Make sure your code is well commented.
- Your code will be graded based on functionality, elegant use of control structures and overall size.
- For the 1st deliverable make sure you are using a loop to print out the resulting matrix. The values of the input matrices can be hard-coded in the .data section of your assembly file.
- For the 2nd deliverable make sure you are implementing a **recursive** version of the algorithm. An iterative solution is going to be graded much less.