

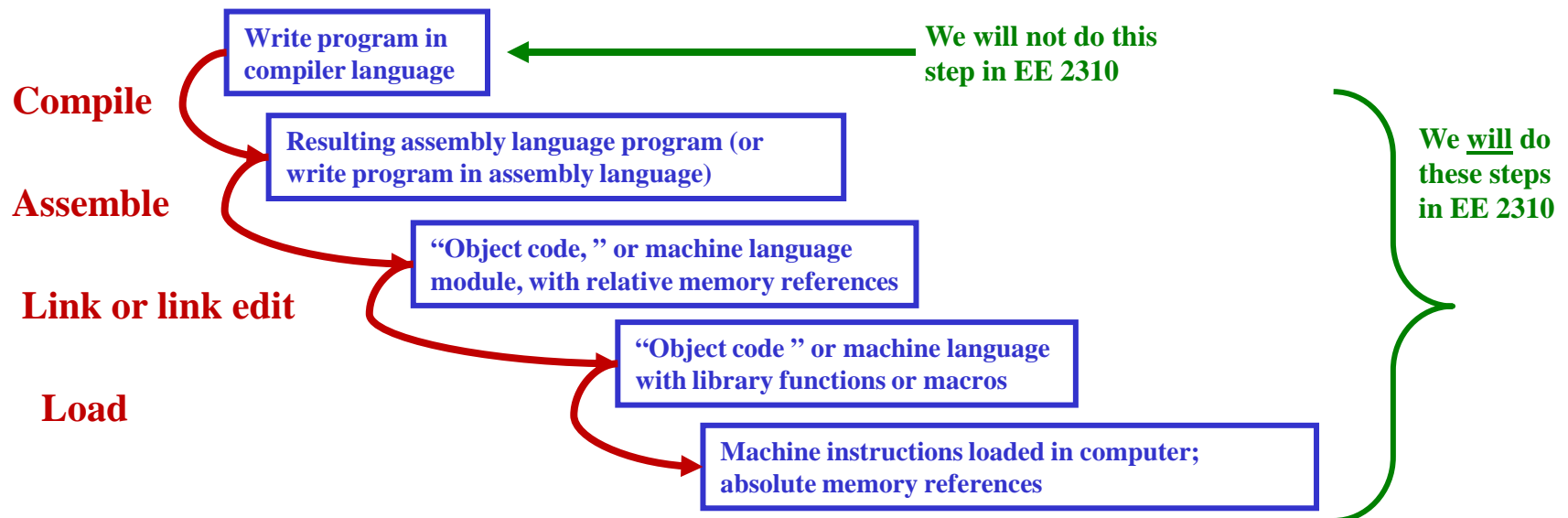


# Assembly Language Programming

- Assemblers were the first programs to assist in programming.
- The idea of the assembler is simple: represent each computer instruction with an acronym (group of letters). Eg: “add” for the computer add instruction.
- The programmer writes programs using the acronyms.
- The assembler converts acronyms → binary codes that the computer recognizes as instructions.
- Since most computer instructions are complex combinations of bits (as we will see in the next lecture), assembler programming is easier.
- Assemblers were succeeded by compilers, which are much more sophisticated programming tools.
- A compiler instruction can represent many computer instructions.

## Why Learn Assembly Language?

- Compilers ease the program development load.
- **However, compilers remove the visibility of computer operation.**
- Assembly language programs gives a better feel for computer operation.
- **Learning assembly language helps understand basics of computer design.**
- **Some programs need assembly language precision.**





# The MIPS Computer and SPIM

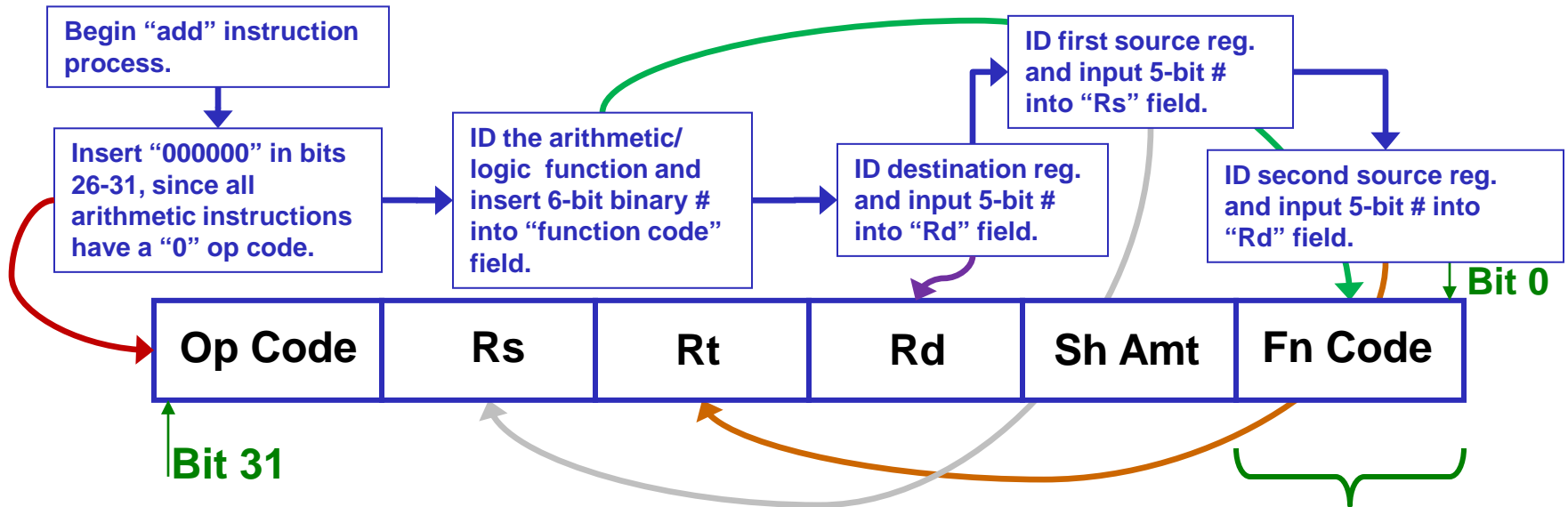
- We will study SPIM, the assembly language of the MIPS computer.
- The MIPS computer was developed by **Patterson and Hennessey**, who wrote our textbook.
- **The design dates from the 1980's. There were several MIPS models. The architecture was very influential – many electronic game systems today employ a descendant of the original MIPS.**
- **We will study the first MIPS model, the 32-bit MIPS R-2000.**
- **The assembler/simulator for the MIPS R-2000 is called SPIM.**
- We will program using the SPIM assembler/emulator, which can run on your laptop or home computer (the R-2000 no longer exists).
  - **Class instructions will primarily refer to QtSPIM, the PC SPIM assembler/simulator. SPIM is also available for Mac and Unix.**

# How Does an Assembler Work?

- An assembler is a **program** that aids **programming**.
- It is **much simpler than a compiler**.
- In an assembler, **1 acronym = 1 instruction**.
  - Example: “Add” = “Take the contents of 2 registers, add them together, and put the result (sum) in another register.”
- The program is written in assembly language, a sequence of acronyms which represent instructions:
  - The assembler converts assembly language instructions into patterns of binary numbers that make up the 32-bit instruction.
  - Registers ID’s are converted to binary numbers that are also inserted into appropriate fields in the 32-bit instruction.
  - The final 32-bit instruction consists of fields of binary numbers, each of which represents part of the instruction.

# Assembler Process Diagram

Instruction: add \$t5,\$t0,\$t1



The function code is the arithmetic/logical function to be performed.  
Example: Function code 10 0000 [0x20] means "add."

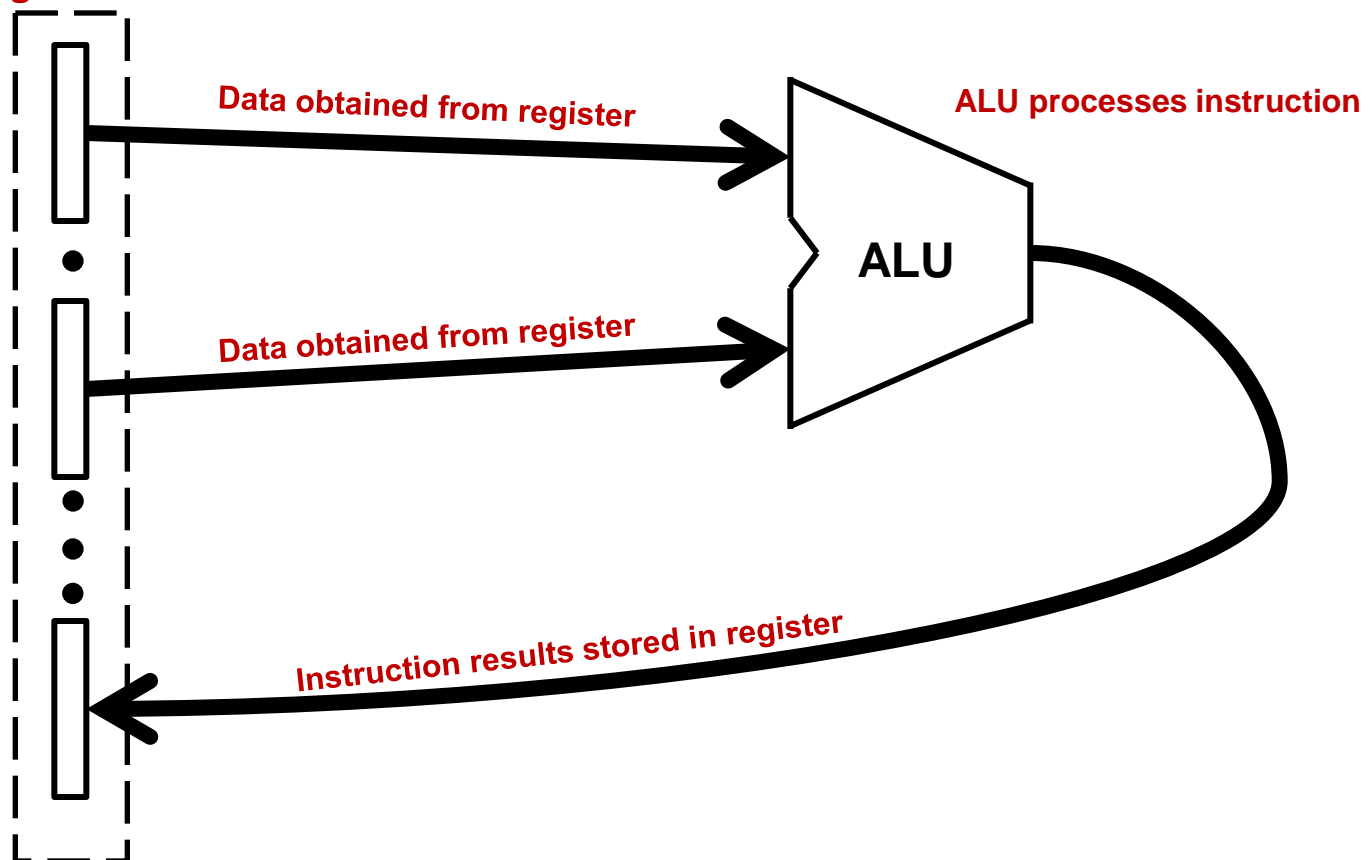
# SPIM Register-Register Instructions

- Today, we will study a few instructions that allow us to write a simple program. We will also check the SPIM set up on your computer.
- “Register-to-register” instructions are the MIPS instructions that do all the calculations and data manipulations.
- Register-to-register instructions are usually of the form:  
**Operation\_Dest. Reg.,Source Reg.,2nd Source Reg.**

**For example: add \$t2,\$t0,\$t1**

## Register-Register Instruction Process

Register Block



# Registers

- **There are 32 registers in the MIPS computer.**
- Remember that registers are just collections of D flip-flops.
- The number of register bits (width) depends on the computer.
- **The MIPS R-2000 computer has 32-bit data words, so each register has 32 flip-flops.**
- The MIPS computer has 32 fixed-point (whole number) registers.
- There are rules (actually suggestions or policies) for register use.
- **The rules may be violated; the assembler has no “Register Police.”**
- **However, to do so is very bad practice.**
- **In EE 2310, you must use registers as recommended in the rules.**



# MIPS Special-Purpose Registers

- These registers are generally used only for the purposes noted:

<u>Registers</u>	<u>Name or Value</u>	<u>Purpose</u>
\$0	Always = 0	Comparisons; clearing registers
\$1	\$at	Assembler temporary register <sup>1</sup>
\$2	\$v0	Value to be passed to function <sup>2</sup>
\$3	\$v1	Value to be passed to function
\$4	\$a0	Argument passing to procedures <sup>3</sup>
\$5	\$a1	Argument passing to procedures <sup>4,5</sup>
\$6	\$a2	Argument passing to procedures <sup>4</sup>
\$7	\$a3	Argument passing to procedures <sup>4</sup>

In MIPS, the dollar sign (“\$”) signifies “register.”

**1 – Don’t ever use this register!** 2 – Used in all system calls to load the syscall number; also specifically used in syscalls 5 & 12 for input data. 3 – Specifically used in syscalls 1, 4, 8, and 11. 4 – Argument passing past a total of four uses the stack. 5 – Also used in syscall 8.

## Special-Purpose Registers (2)

- These registers are also generally used only for the purposes noted.

<u>Register</u>	<u>Name</u>	<u>Purpose</u>
<b>\$26</b>	<b>\$k0</b>	<b>Reserved for use by operating system.<sup>1</sup></b>
<b>\$27</b>	<b>\$k1</b>	<b>Reserved for use by operating system.<sup>1</sup></b>
\$28	\$gp	Pointer to global area <sup>2</sup>
\$29	\$sp	Points to current top of stack
\$30	\$fp	Points to current top of frame
\$31	\$ra	Return address to exit procedure

**1 – Don't ever use these registers! 2 – We will not use this register.**

## MIPS Temporary Registers

- The t-registers hold data in programs. Use as you wish.

<u>Register</u>	<u>Name</u>	<u>Purpose</u>
\$8	\$t0	Holds a local (temp) variable
\$9	\$t1	Holds a local (temp) variable
\$10	\$t2	Holds a local (temp) variable
\$11	\$t3	Holds a local (temp) variable
\$12	\$t4	Holds a local (temp) variable
\$13	\$t5	Holds a local (temp) variable
\$14	\$t6	Holds a local (temp) variable
\$15	\$t7	Holds a local (temp) variable
\$24	\$t8	Holds a local (temp) variable
\$25	\$t9	Holds a local (temp) variable


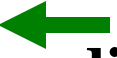
- NOTE: Think of t-registers as “**scratchpad registers.**”

## MIPS Saved Temporary Registers

- **S-registers also hold data in programs. In general, use as you wish. Some restrictions on s registers will be discussed later.**

<u>Register</u>	<u>Name</u>	<u>Purpose</u>
\$16	\$s0	Holds a saved temporary variable
\$17	\$s1	Holds a saved temporary variable
\$18	\$s2	Holds a saved temporary variable
\$19	\$s3	Holds a saved temporary variable
\$20	\$s4	Holds a saved temporary variable
\$21	\$s5	Holds a saved temporary variable
\$22	\$s6	Holds a saved temporary variable
\$23	\$s7	Holds a saved temporary variable

## Register-Register Instructions: Add

- The format of add is: `add_rd,rs,rt`. The contents of registers `rs` and `rt` (the “source registers”) are added, and the result is put in `rd`. Examples:
  - `add $s1, $t2, $t6: [$t2] + [$t6] → [$s1]`.  [ ] = “Contents of”
  - `add $t7, $t3, $s4: [$t3] + [$s4] → [$t7]`.  “\$” = “Register”
- In addi, the second source is an “immediate,” i.e., a number coded in the instruction.\* Examples:
  - `addi $s1, $t2, 27: [$t2] + 27 → [$s1]`.
  - `addi $t7, $t3, 0x15: [$t3] + 0x15 → [$t7]`.
- Adding the “u” at the end (i.e., `addu` or `addiu`) simply instructs the computer to ignore overflow indication.

\* **Immediates may be any size up to 32-bits (that is  $\sim \pm 2,000,000,000$ ).**

## Subtract

- **Subtract** has exactly the same form as add: `sub rd, rs, rt`. Examples:
  - `sub $t3, $t4, $t5`:  $[\$t4] - [\$t5] \rightarrow [\$t3]$ .
  - `sub $s3, $s6, $t0`:  $[\$s6] - [\$t0] \rightarrow [\$s3]$ .
- Although there is not a formal “subi,” a coded number in the instruction can be substituted for `rt`. Thus:
  - `sub $t3, $t4, 67`:  $[\$t4] - 67 \rightarrow [\$t3]$ .
  - `sub $s3, $s6, 0xdf8`:  $[\$s6] - 0xdf8 \rightarrow [\$s3]$ .
- Like “addu,” `subu` prevents an overflow indication from being given when the instruction is executed, if overflow occurs.

## Multiply/Divide

- **Multiply and divide are real MIPS instructions, but SPIM makes them into pseudo instructions in the way that it handles the product and quotient results. We will discuss the details of this later.**
- **Multiply is written as `mul $rd,$rs,$rt`. An example:  
`mul $t2, $t0, $t1`**
- **Similarly, divide is written as:  
`div $t4,$t2,$t3`**
- **Note that the form of multiply and divide is identical to that for add and subtract.**

## Logical Instructions

- Logical instructions perform logic functions on the arguments on which they operate. For example:
  - **and \$t0, \$t1, \$t2:**  $[\$t1] \cdot [\$t2] \rightarrow [\$t0]$ .
- Logical operations include AND, OR, NOT, NOR, and XOR.
- Logical instructions are performed on the arguments on a bitwise basis, as shown below (**and \$t0,\$t1,\$t2**).



- Thus in the above example, bit 0 of register t1 is ANDed with bit 0 of register t2 and the result stored in bit 0 of \$t0; bit 1 of \$t1 is ANDed with bit 1 of \$t2, and the result stored in bit 1 of \$t0, etc.



## Logical Instructions (2)

- **AND function:**
  - **and \$s0, \$t3, \$t4:**  $[\$t3] \cdot [\$t4] \rightarrow [\$s0]$ , on a bitwise basis.
- **OR is performed likewise:**
  - **or \$s0, \$t3, \$t4:**  $[\$t3] + [\$t4] \rightarrow [\$s0]$ , on a bitwise basis.
- **andi/ori – These instructions are similar to addi, thus:**
  - **ori \$t4, \$t5, 0xff1:**  $[\$t5] + 0xff1 \rightarrow [\$t4]$ , on a bitwise basis.
  - **andi \$t4, \$t5, 587:**  $[\$t5] \cdot 587 \rightarrow [\$t4]$ , on a bitwise basis.
- **NOR and XOR are functionally identical to “AND” and “OR.”**
  - **nor \$t1, \$t2, \$t6:**  $\overline{[\$t2] + [\$t6]} \rightarrow [\$t1]$ , on a bitwise basis.
  - **xor \$s1, \$s2, \$s3:**  $[\$s2] \oplus [\$s3] \rightarrow [\$s1]$ , on a bitwise basis.
- **NOT is a simpler instruction, with only one operand:**
  - **not \$t7, \$s5:**  $\overline{[\$s5]} \rightarrow [\$t7]$ , on a bitwise basis.

## Other Register-Register Instructions

- **Load immediate:** `li $rd, #:` “load immediate.” Load immediate allows a constant whole number to be loaded into a register. Note: `li` is a pseudo instruction in some cases. This will be covered later.
  - `li $t0, 2405` – the number 2405 is loaded into register `$t0`.
  - `li $s4, 16523` – the number 16,523 is loaded into register `$s4`.
- **move:** **move** transfers the contents of one register to another. Example:
  - `move $s5, $t3:` [`$t3`] → [`$s5`] (`$t3` contents are not changed).
  - **move** is also a pseudo instruction.

## Putting SPIM on Your Computer

- The most up-to-date version of the SPIM simulator is maintained by James Larus, formerly of the University of Wisconsin at Madison. It is “freeware,” and is maintained by a web site called “SourceForge.”
- Go to the SourceForge website:  
<http://sourceforge.net/projects/spimsimulator/files/>
- You will see the choice of a number of SPIM downloads. Select: [QtSpim\\_9.1.12\\_Windows.zip](#) and click on that link. Note that there are Linux and Mac versions as well.
- Follow instructions to load onto your computer (next page).

## James Larus Version of SPIM (2)

- The download software will ask if you want to open or save. Click “save.” If you are using Firefox (which I STRONGLY recommend), simply save it as a Firefox download file. Note that the file is “pcspim.zip” file.
- If you do not have a download file, a good approach is to save this program in the Windows folder called “Program Files.” Use the “save in” selection at the top of this pop-up window to select “Program Files.” Click “OK.”
- The file will be saved to the selected folder. It is about 21 Mbytes, which takes a short while on 3G or broadband.
- Now either open the download file or use Windows Explorer to find the zipped PCSPIM. Double-click on it to unzip and install (ignore the occasional sales pitch on “Winzip” and simply unzip).

## James Larus Version of SPIM (3)

- The program will unzip and start to install. Follow directions and exit other programs on your computer.
- The program will give you the option of storing PCSPIM under “C:/Program Files/QtSpim.” Click “OK” and the file folder will be created.
- The setup program will also install a QtSpim.exe icon on your desktop so that you can easily start up this version of SPIM.
- Once QtSpim is created, you may open it to see if you get the register, text, and data windows. If you do, it is correctly installed.

## Two Other Things You Need to Know

- Directives are special instructions that tell the assembler important information. We will study most of them later.
  - All programs are started with the “.text” directive.
  - When programming, put only one SPIM instruction per line.
  - The first line of a program must be labeled “main:” . That is, the first instruction in the program has the label **main:** to its left, e.g.:

```
main:  .text
       li $t3,356
       li $t7,44
       add $t2,$t7,$t3
       etc., .....
```

- The last instruction in a program is system call 10 (“stop!”).

```
li $v0,10
syscall
```



## How to Construct and Run a Program in SPIM

- Write the program in NotePad. Save it in the SPIM file.
- To execute the program:
  - Open QtSPIM.
  - In QtSPIM, open your NotePad program as a file (“Load file”).
  - You may not see the program in the open file list, until you select “All Files” in the program file type select window.
  - After selecting the program, it will assemble automatically, link, and load.
  - Mouse click on “Simulator” tab. Select “Run/Continue.”
  - The program should immediately run.
  - Instead of running the program, you can single step using F10.
- Debug is simple; for program errors (lines that will not properly assemble), SPIM merely stops with a note indicating the line that failed (more on next slide).

## Fixing Program Bugs

1. Launch QtSpim or PCSPIM and open your program.
2. It will assemble until encountering an error, then stop. The assembler will indicate a line where the error exists.
2. Open the text editor, edit the offending instruction, and re-save.
3. Restart SPIM and reopen the program. The program will assemble until another error is encountered.
4. If another error is encountered, repeat above procedure.
5. If the program assembles, it will show up in the text window, and no error message will appear.
6. Click “Go” (under “Simulator” tab) and click “OK” to run, per previous slide.
7. If your program writes to the console, check it for desired result.
8. Next, you can single step the program to watch instructions execute if you wish.



# Program 1

- **Write a program on your computer to do the following:**
  - **Enter 47 into \$t0 (“x”)**
  - **Put 253 into \$t1 (“y”)**
  - **Load 23 in \$t2 (“z”)**
  - **Compute  $x^2 + 3y + 10z$**
  - **Store the result in \$t3**
  - **Stop the program by using system call 10.**
  - **What is the number in \$t3?**

**Remember: \$ = “register.”**

## Program 2

- **Write a program to do the following:**
  - **Load 25 in \$t4 and 126 into \$t7.**
  - **Add the contents of registers \$t4 and \$t7, storing in \$t3.**
  - **AND this result with \$t4, storing in \$s2.**
  - **Multiply \$s2 by the contents of \$t7, storing in \$t9. Also store the result in \$t9 in \$s5 and \$s7.**
  - **Use syscall 10 to end the program.**
  - **What is the value of the number in \$s7 (state in hex)?**

## Program 3

- Write a program to solve the equation:  $w = 4x^2 + 8y + z$ .
- Variable  $x$  is 0x123,  $y$  is 0x12b7, and  $z$  is 0x34af7.
- Use  $\$t0$  for  $x$ ,  $\$t1$  for  $y$ , and  $\$t2$  for  $z$ .
- Remember to start your program with `.text` and label the first instruction as “main:”.
- Store  $w$  in  $\$s0$  and end your program with `syscall 10`.  
What is the value of  $w$  in hex?