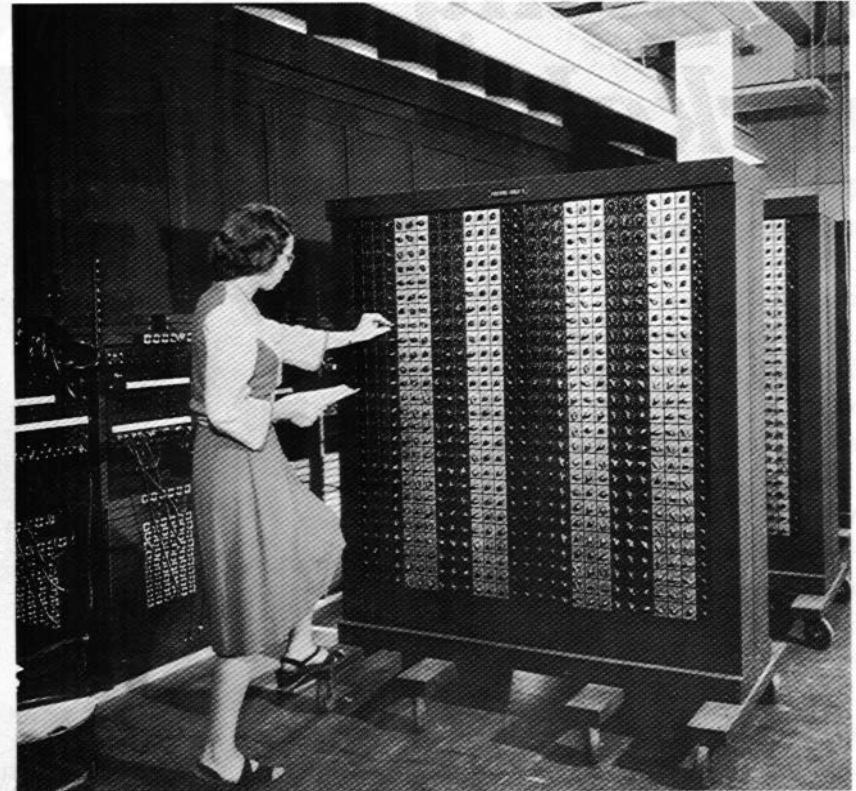


Assembly Language Programming

- Remember the “programmer” shown in our first lecture?
- Originally, computers were programmed **manually**.
- After a while, scientists began to consider ways to **accelerate and facilitate** programming.



TENDING ENIAC. Betty Holberton programs the first electronic computer.

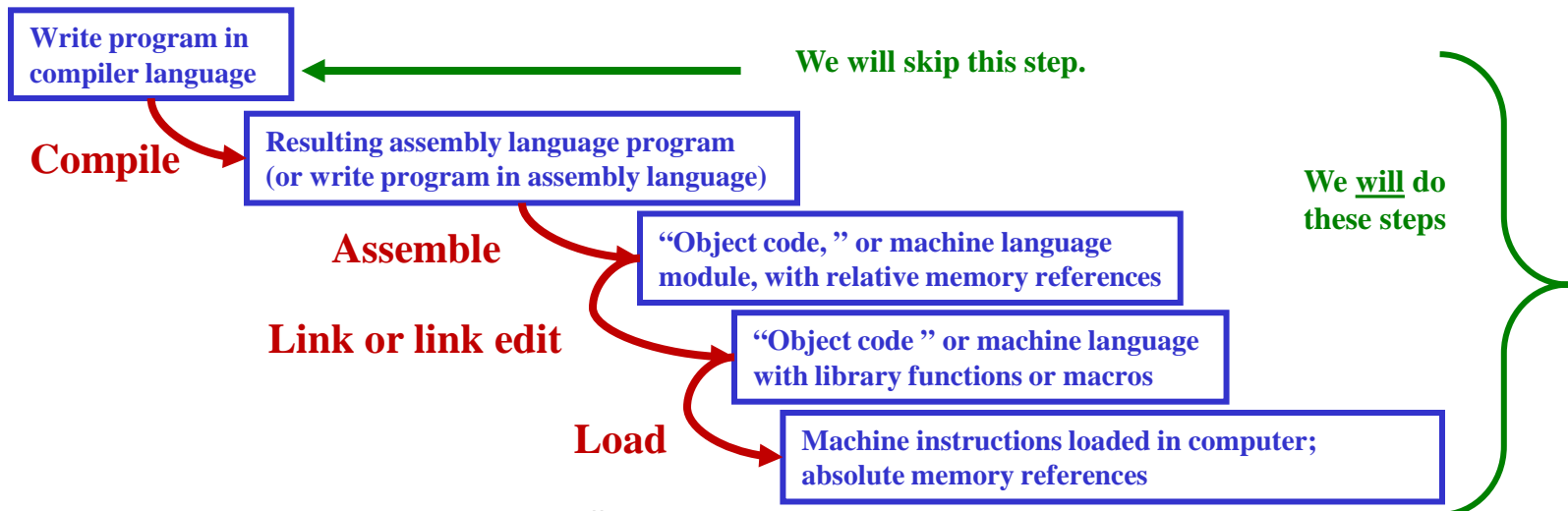


Assembly Language Programming

- 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 to 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 **greatly ease** the program development load. Assemblers are **much more primitive**. Then why learn assembly language?
- **First**, compilers remove the visibility of computer operation.
- **Second**, using assembly language gives a better feel for computer operation.
- **Third**, learning assembly language aids in understanding computer design.
- **Fourth**, assembly language improves precision (and reduces size) of programs.



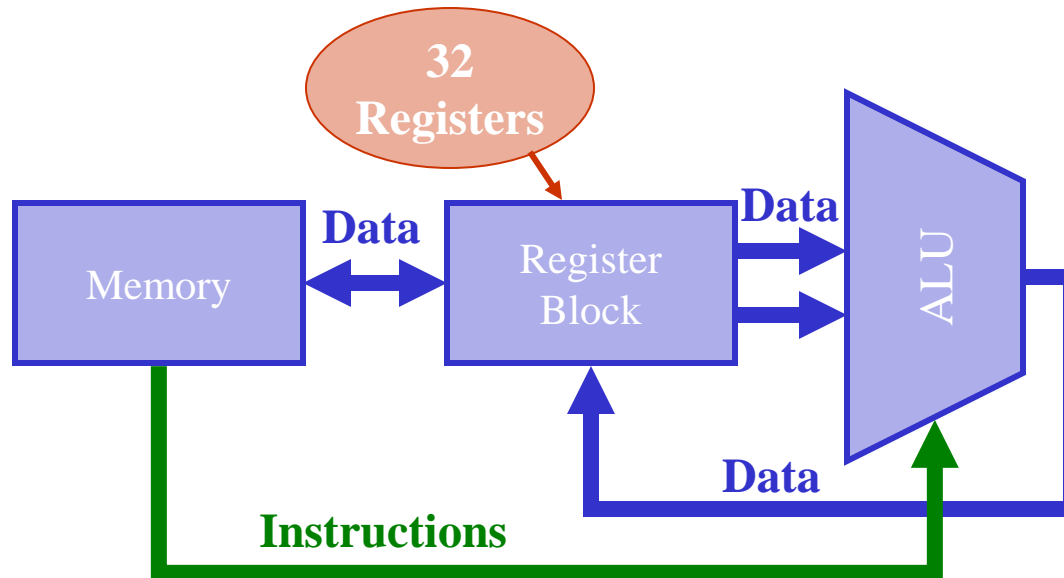


The MIPS Computer and SPIM

- We will study SPIM, the assembly language of the MIPS computer.
- **Patterson and Hennessey**, our textbook authors, helped design MIPS.
- 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.

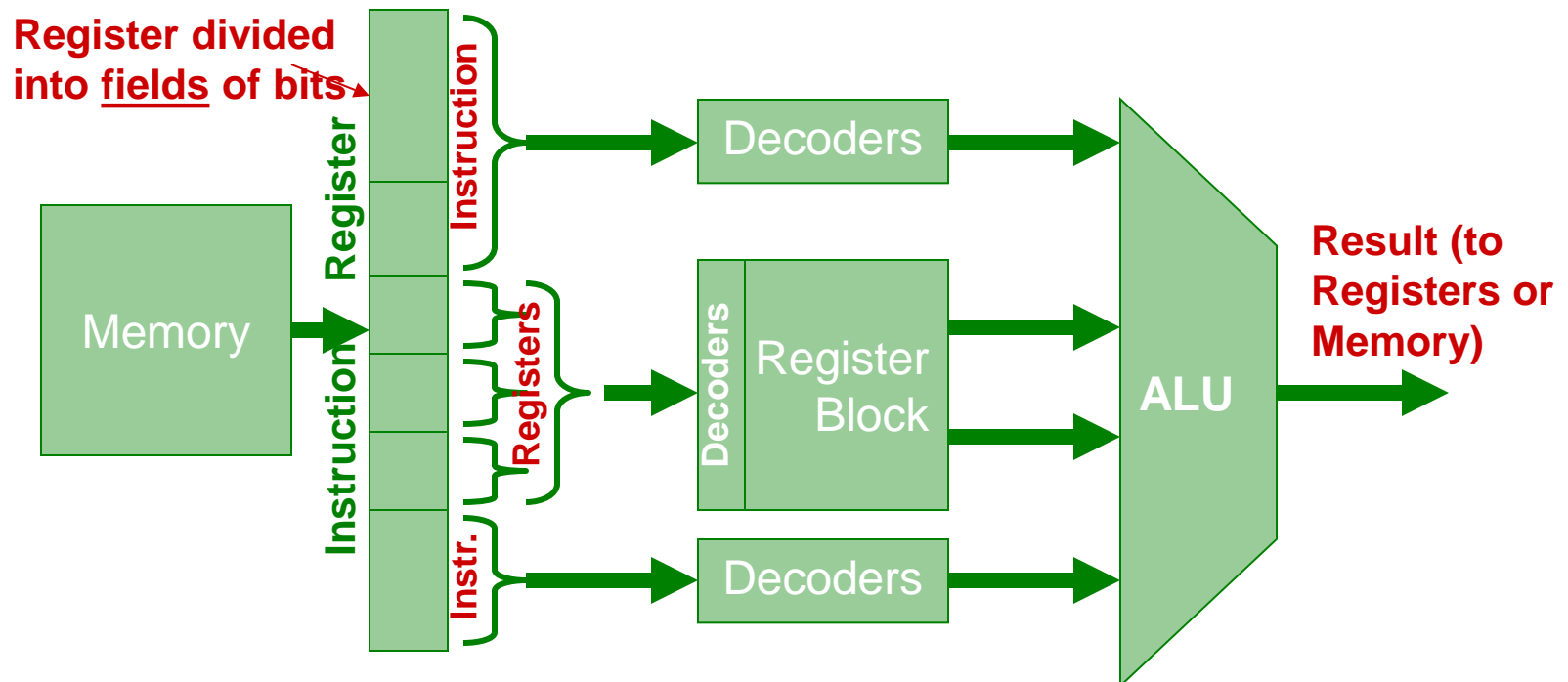
Instruction Execution on a MIPS Computer

- The MIPS computer (and in fact, most computers) execute instructions in the method shown below:
 - Obtain instruction and information to process (usually from data registers).
 - Interpret instruction and do processing in CPU arithmetic unit (ALU or datapath).
 - Put results in storage (in data registers).



Instruction Processing

- In all computers, an instruction is retrieved from memory, the fields of the instruction are decoded, and the decoded commands are processed.

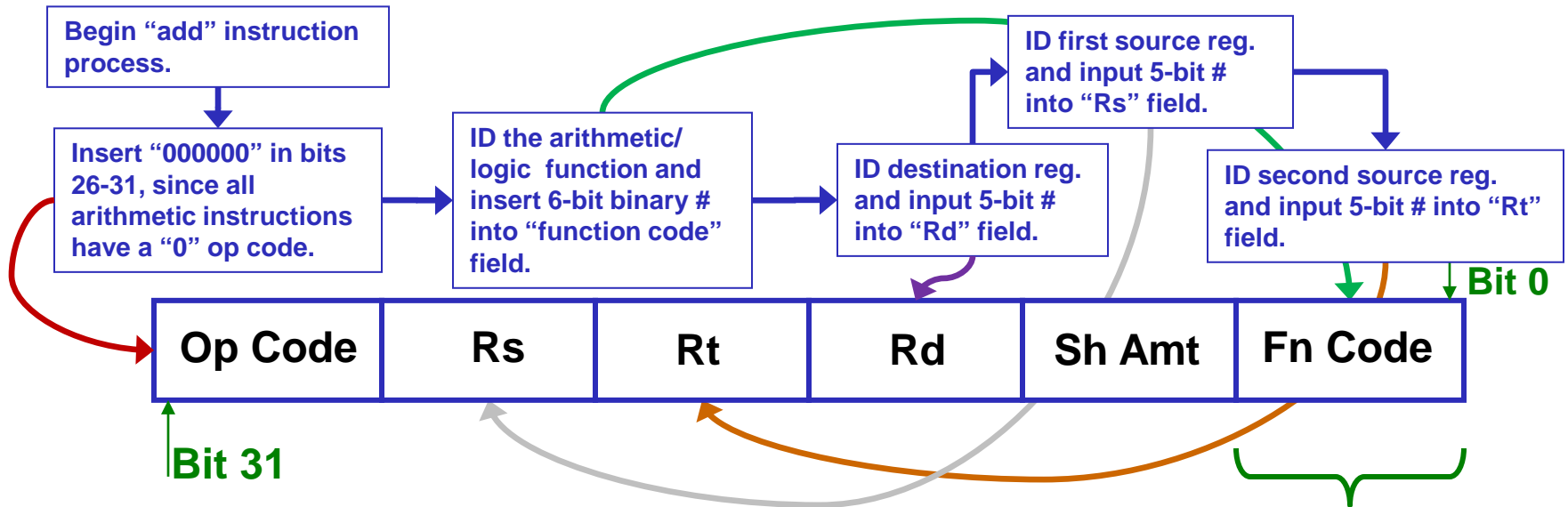


How Does an Assembler Work?

- An assembler substitutes acronyms for binary instructions to **simplify programming**.
- 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 \$t2,\$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

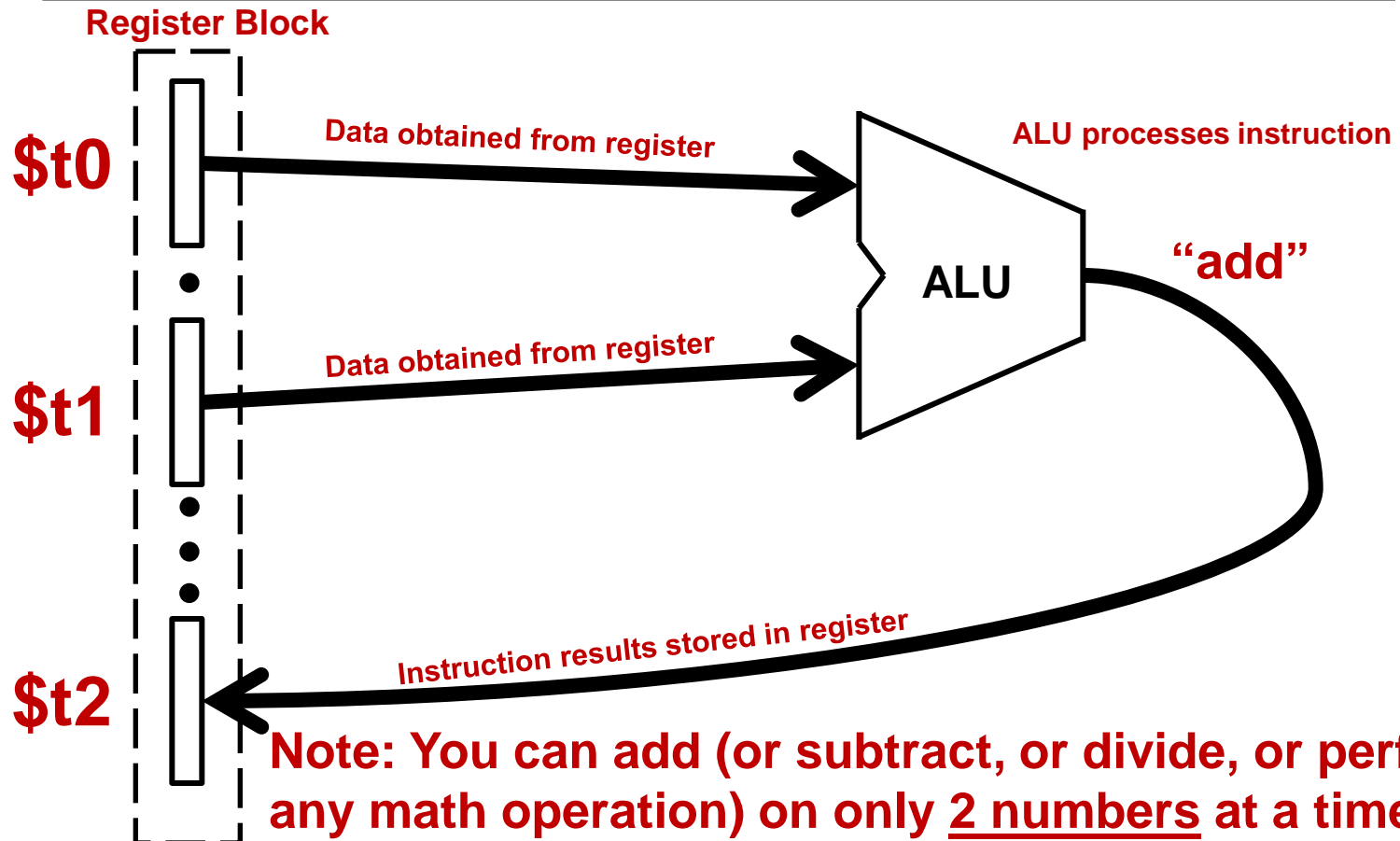
- Today, we will study just a few instructions that will allow us to write a simple program.
- “Register-to-register” instructions do all the calculations and data manipulations.
- Register-to-register instructions are usually of the form:
Operation_Dest. Reg.,Source Reg.,2nd Source Reg.

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

Op Code	Rs	Rt	Rd	Sh Amt	Fn Code
000000	01000	01001	01010	00000	100000
Reg./Reg.	\$t0	\$t1	\$t2	Shft.=0	add

Computer instruction as loaded: 00000001000010010101000000100000

Register-Register Instruction Process



A Few Things We Need to Know

- To write a program in QtSPIM, we need to know how to start, end and format the instructions of a program.
- We'll learn more about **directives** in the next lecture. All you need to know today is that **all programs are started with the directive .text.**
- Beneath that directive, start writing the program instructions. **ONLY 1 INSTRUCTION PER LINE.**
- **Labels** are important in QtSPIM. ANY instruction may be labeled or not as you choose, except for the 1st instruction in the program. **It MUST be labeled main:**

A Few Things We Need to Know (2)

- The program must be in **one contiguous string of instructions** (1 per line).
- The **.text** directive **ALWAYS** starts the program.
- At the end of the program, you must use **syscall 10** to end the program. It says “halt!”
- There are a number of system calls that allow the user to communicate with the program. They all have the same form. Load register \$v0 with the number of the syscall, then execute the syscall instruction.
- Thus: **li \$v0,10** stops the program.
syscall

A Few Things We Need to Know (3)

- The MIPS R-2000 has a lot of registers to store information. Many of them are available to the user.
- We will learn more about MIPS registers in a few moments. Right now, **here are the identities of a few of the registers you can use in a program (note that the \$ sign is the QtSPIM designation meaning “register”):**

—\$t0

—\$t3

—\$t1

—\$t4

—\$t2

—\$v0 (for syscalls)

Structure of a Simple Program

- Suppose we want to square the numbers 648 and 1329 and add them together.
- How might we do that in a program?
- First, remember that **ALL data used in calculations MUST be in one of the CPU registers.**
- Thus we must **load the registers with the numbers to operate on them.**
- We can do that with an “**li**” instruction, which puts any number up to 32 bits in the designated register.
- Thus: **li \$t0, 957** puts that decimal number in **\$t0**.

Structure of a Simple Program (2)

- “.text” = “Program follows”
 - Program starting instruction always labeled “**main:**”
 - Label always ends with colon (:)
 - One instruction per line.
 - Syscall 10 ends the program.
 - All program instructions must be grouped together under the **.text** label.
 - Check \$t4 in the register window for the answer.
-
- The diagram illustrates the structure of a simple program. Red arrows point from the list items to specific parts of the code: from “.text” to the **.text** label; from “Program starting instruction always labeled ‘main:’” to the **main:** label; from “Label always ends with colon (:)” to the colon in **main:**; from “One instruction per line.” to each of the four instruction lines; from “Syscall 10 ends the program.” to the **li \$v0,10** instruction; and from “All program instructions must be grouped together under the **.text** label.” to the entire code block.
- ```
.text
main: li $t0,648
 li $t1,1329
 mul $t2,$t0,$t0
 mul $t3,$t1,$t1
 add $t4,$t2,$t3
 li $v0,10
 syscall
```

## Let's Do a First Program

- To write a simply program, remember:
- Write your program in Notepad and store in the SPIM folder.
  - Each program must start with the “.text” directive on the first line (without the quotes).
  - On the **second line**, put the first instruction. Note: It must be labeled “main:” in the margin (again, no quotes).
  - **Remember: the instruction “li” puts a number in any register, e.g., li \$t1,273. Remember to put a space after the “li.”**
  - A few of the registers we can use are **\$t0, \$t1, \$t2, \$t3.**
  - And remember that:  
**li \$v0,10**  
**syscall**  
tells your program to **“stop!” Use them last.**
- See your program outline to **check your formats.**



## Let's Do a First Program (2)

- Here's what your program should do:
  - Put 10 in \$t0 and 5 in \$t1 (using the “li” instruction).
  - Add the contents of \$t0 and \$t1 and put it in \$t2 (use “add”).
  - Add the contents of \$t0 and \$t2 and put it in \$t3 (ditto).
  - Stop the program.
  - The “add” instruction looks like: add \$rs, \$rt, \$rd -- \$rs is the first source register, \$rt is the second source, and \$rd is the destination. Note that either number can be in either register.
- After the program runs, look at the register display (left side panel of MIPS display). Find \$t3.
- Contents of \$t3 should be 0x19. When you see this number in \$t3, you have completed your first program.
- The complete program is on the next slide.

# 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`. This means  $[\$rs] + [\$rt] = [\$rd]$ , where `[ ]` = “contents of.” Examples:
  - `add $s1, $t2, $t6`:  $[\$t2] + [\$t6] \rightarrow [\$s1]$ .  `[ ]` = “Contents of”
  - `add $t7, $t3, $s4`:  $[\$t3] + [\$s4] \rightarrow [\$t7]$ .  “\$” = “Register”
- In addi, the second source is an “immediate,” i.e., a number coded in the instruction.\* Examples:
  - `addi $s1, $t2, 27`:  $[\$t2] + 27 \rightarrow [\$s1]$ .
  - `addi $t7, $t3, 0x15`:  $[\$t3] + 0x15 \rightarrow [\$t7]$ .
- Adding the “u” at the end (i.e., `addu` or `addiu`) simply instructs the computer to ignore overflow indication.
- **Note: add (and ALL MIPS instructions) can process only two (2) numbers.**

\* **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.

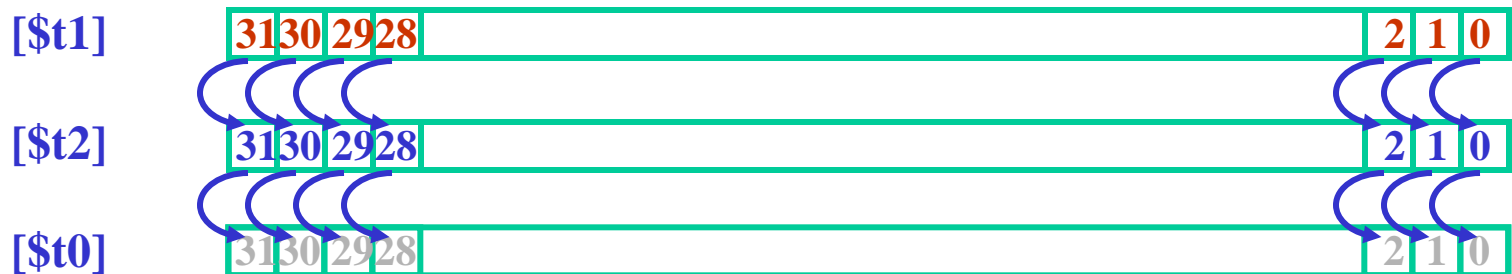


## Multiply/Divide

- **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.
  - **The ONLY way to transfer data from one register to another is via “move!” NOT load, NOT load immediate!**

## 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\\_9.1.17\\_Windows.exe](#) and click on that link. There are Linux and Mac versions as well.
- Follow instructions to load onto your computer (next page). **Note that this used to be a zip file, but as far as I can tell, it no longer is—it is simply a binary file.**

## James Larus Version of SPIM (2)

- The download software will ask if you want to open or save. Click “save,” and save it as a download file. **Note that the file is an “exe” file.**
  - At 34 Mbytes, the download takes a bit even on 3G or broadband.
  - **Open the download file and it will begin to install.**
  - **The program will ask to create a folder: “C:/Program Files/QtSpim.” Click “OK” and the file folder will be created.**  
**Note: All of the programs you create will be saved to this file.**
  - The setup program will also install a QtSpim.exe icon on your desktop.
  - **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.**

## Reminder about “.text” and “main:”

- 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:”.
  - Example:

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

- The last instruction in a program is always 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 folder.
- 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)?**

## Bonus Program\*

- 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?
- Try this program on your own—we will review it later.

\* Optional: Work at home if you wish for more practice.