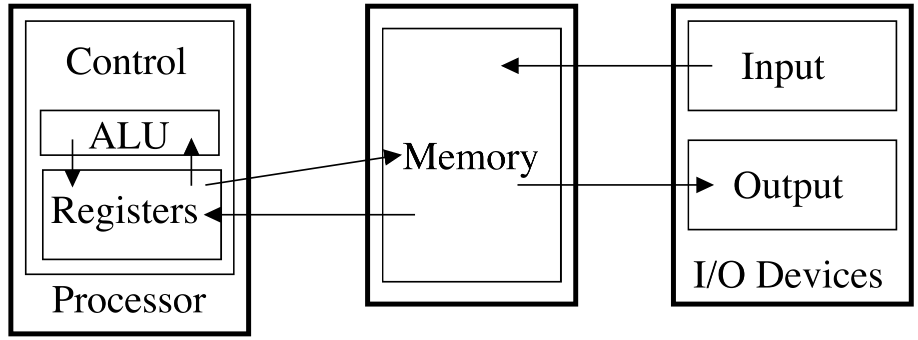
MIPS Introduction

Read: Chapter 2 and Appendix A (5th edition); Chapter 2 and Appendix B (4th edition)
The appendix is available as http://pages.cs.wisc.edu/~larus/HP_AppA.pdf from spim website.
Note: The on-line version is named Appendix A. It is the same as the 4th edition's Appendix B.

- Language of the Machine.
- More primitive than higher level languages.
  - E.g., no sophisticated control flow such as **`for`** and **`while`**.
  - Only simple branch, jump, and jump subroutine.
- Very restrictive.
  - E.g., MIPS Arithmetic Instructions have: two operands, one result.
    - Can do in one instruction: **`a = b + c`**
    - Cannot do in one instruction: **`a = b + c * d - e`**
- We will be working with the MIPS-32 instruction set architecture.
  - Similar to other architectures developed since the 1980's.
  - Used by (at various times) NEC, Nintendo, SGI (formerly Silicon Graphics Inc.), Sony
- Design goals of MIPS:
  - Maximize performance and minimize cost.
  - Reduce design time.

Basic CPU Organization:

- Simplified picture of a computer:



- Three components:
  - Processor (or Central Processing Unit or CPU or "core"); MIPS in our case. Intel, PowerPC, UltraSparc, …
  - Memory — contains the program instructions to execute and the data for the program.
  - I/O Devices — how the computer communicates to the outside world. Keyboard, mouse, monitor, printer, game controller, tablet, etc.
- CPU contains three components:
  - Registers — hold data values for the CPU to manipulate.
  - Arithmetic Logic Unit (ALU) — performs arithmetic and logic functions. Takes values from and returns values to the registers.
  - Control — Determines what operation to perform, directs data flow to/from memory, directs data flow between registers and ALU.
    - Actions are determined by the current Instruction.

Memory Organization:

- Viewed as a large, one-dimensional array, with an address for each element — byte — of the array.
- A memory address is an index into the array.
- "Byte addressing" — the index points to a byte, 8 bits in today's computers, of memory.
- MIPS addresses (up to) 4 Gigabytes of memory:
  - Bytes are numbered from 0 to $2^{32} - 1$; or 0 to 4,294,967,295.
- Bytes are nice, but most data items use larger "words".

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data
…	8 bits of data
4,294,967,293	8 bits of data
4,294,967,294	8 bits of data
4,294,967,295	8 bits of data

Memory Organization (continued):

- For MIPS, a "word" is 32 bits, or 4 bytes.
  - Each register in the CPU holds 32 bits.
  - Not just a coincidence!
- 2^{32} bytes with byte addresses from 0 to 2^{32} - 1.
- 2^{30} words. The words are at addresses:

  0, 4, 8, 12, 16, 20, 24, 28, ..., 2^{32} - 4 (in decimal)

  0, 4, 8, C, 10, 14, 18, 1C, 20, 24, 28, 2C, ... (in hexadecimal)
- Words are "aligned".
  - Each word starts on an address that is divisible by 4.
  - What are the least 2 significant bits of a word address in binary?
- Notes: If you have not already memorized these:
  - $2^{10} = 1{,}024_{ten}$ = 1 Kilo = 1K
  - 2^{20} = 1 Mega = 1M
  - 2^{30} = 1 Giga = 1G
  - 2^{40} = 1 Tera = 1T

0	32 bits, 4 bytes, of data
4	32 bits, 4 bytes, of data
8	32 bits, 4 bytes, of data
12	32 bits, 4 bytes, of data
...	32 bits, 4 bytes, of data
4,294,967,284	32 bits, 4 bytes, of data
4,294,967,288	32 bits, 4 bytes, of data
4,294,967,292	32 bits, 4 bytes, of data

0	32 bits, 4 bytes, of data
4	32 bits, 4 bytes, of data
8	32 bits, 4 bytes, of data
C	32 bits, 4 bytes, of data
...	32 bits, 4 bytes, of data
FF FF FF F4	32 bits, 4 bytes, of data
FF FF FF F8	32 bits, 4 bytes, of data
FF FF FF FC	32 bits, 4 bytes, of data

Registers vs. Memory:

- Registers can be thought of as a type of memory. They are inside the CPU; thus, they are the “closest” memory.
- Registers provide a place to hold values inside the CPU, and allow a large set of operations to be performed on their values. I.e., add, subtract, compare, etc.
- Principal advantages of registers vs. memory:
  - Fast access.
  - Fast access.
  - Fast access.
- Principal advantages of memory vs. registers:
  - Lower cost.
  - Lower cost.
  - Lower cost.
- An intermediate type of memory: Cache.
  - Different “flavors” depending on size and physical location.
  - Level 1 cache “closest” to the CPU.
    - Usually installed on the chip as part of the CPU.
    - Typically small: 32K, 64K
  - Level 2 cache between the CPU and the memory.
    - Not part of one processor core, but is present on the chip. Shared by all processor cores (or not).
    - Typically a few Megabytes.
  - We will come back to the topic of cache later in the semester.

Registers vs. Memory (continued):

- Register Organization. (See also the SPIM Appendix, page 24).

Name	Register number	Usage	Preserved on call?
`$zero`	0	the constant value 0	n.a.
`$at`	1	reserved for the assembler	n.a.
`$v0-$v1`	2-3	values for results & expression evaluation	no
`$a0-$a3`	4-7	arguments	yes
`$t0-$t7`	8-15	"temporary"; used for almost anything	no
`$s0-$s7`	16-23	"saved"; used for almost anything	yes
`$t8-$t9`	24-25	"temporary"; used for almost anything	no
`$gp`	28	global pointer	yes
`$sp`	29	stack pointer	yes
`$fp`	30	frame pointer	yes
`$ra`	31	return address	yes

- These are the "General Registers". MIPS also has:
  - PC (program counter) register, Status register.
  - Floating-point registers.

Registers vs. Memory (continued):

- For now (programs 1, 2, 3, and 4) we will use:
    - **`$zero`**, **`$s0-$s7`**, **`$t0-$t9`** for writing programs.
        - **`$zero`** always has the value **`0`**; you cannot change its contents.
        - **`$s0-$s7`** and **`$t0-$t9`** can be used interchangeably in programs.
            - This will change later when we start using functions.
    - **`$a0`** and **`$v0`** for printing.
        - Only for printing!
        - Both **`$a0`** and **`$v0`** have other uses that will show up in functions.
    - Do NOT use any of the other registers:
        - **`$at`**
        - **`$v1`**
        - **`$a1-$a3`**
        - **`$gp`**, **`$sp`**, **`$fp`**, **`$ra`**
- Some of the above will change when we start using functions.

MIPS Arithmetic:

- All arithmetic instructions have at most 3 operands.
- All arithmetic is done in registers!
  - Can not, for example, add a number directly to a value stored in memory. In MIPS, this requires 3 steps:
    - Load the value from memory into a register.
    - Add the number to the register.
    - Store the result in memory.
  - Thus, MIPS is a "load-store" architecture. All work other than loading and storing is done only in registers.
- Operand order is fixed.
  - Destination operand is first.
- Example:
  - C or Java code:

    ```
    xray = yoke + zebra;     $t0 = $t1 + $t2
    ```

  - MIPS code:

    ```
    add    $t0, $t1, $t2
    ```

    - adds contents of `$t1` and `$t2`, placing the result in `$t0`.

`$s1 has papa`

`$s2 has sierra`

`$s3 has tango`

`$s5 has foxtrot`

MIPS Arithmetic (continued):

- Longer expressions require more instructions:
  - C or Java code (assume all variables are of type `int`):

```
oscar = papa + sierra + tango;
foxtrot = foxtrot - oscar;
```

  - MIPS version:

Note: `sub` instruction for subtract.
Same format as `add` instruction.

```
add $t0, $s1, $s2  # $t0 = papa + sierra;  put result "temporarily" in $t0
add $s0, $t0, $s3  # oscar = $t0 + tango; use the "temporary" result from $t0
sub $s5, $s5, $s0  # $s5 = foxtrot - oscar
```

    - Note: use of `$t0` to hold "temporary" result.
    - Note: `#` marks the beginning of a comment that runs until the end of the current line
    - Note: operands must be registers.
- We did not use variable names in the MIPS version, just register names.
  - Programmer has to "remember" which registers hold which variables.
    - Useful to have comments that say this!
    - Essential if you want a good grade on the programs.

`add $s3, $s3, $s3`

MIPS Arithmetic (continued):

- Registers vs. Memory:
  - Arithmetic instructions — operands must be registers.
  - There are only 32 registers available.
  - Compiler for a language, such as C, will associate variables with registers automatically.
- What about programs with more variables than there are registers?
  - This covers just about every program!
  - Must move the values of the variables between memory and registers.

<u>Load and Store Instructions</u>:

- MIPS uses load instructions to copy a value from memory to a register. There are three versions, depending on the amount of data being copied:
    - **lw** will copy a word (32-bits).
    - **lh** will copy half a word (16-bits).
    - **lb** will copy one byte (8-bits).
- MIPS uses store instructions to copy a value from a register to a memory location.
    - **sw** will copy a word (32-bits).
    - **sh** will copy half a word (16-bits).
    - **sb** will copy one byte (8-bits).

Load and Store Instructions (continued):

- MIPS allows us to use symbolic names for memory locations.
  - Saves having to use binary or hexadecimal addresses.
    - **bats** is easier to remember and understand than the memory address `0x1000 0010`, for example.
  - In a MIPS assembly program, we can assign symbolic names to memory locations using `.word`, `.half`, and `.byte`, as appropriate:

```
bats:   .word  17    # creates a 32-bit integer w/ the initial value 17, named bats
balls:  .word   3    # creates a 32-bit integer w/ the initial value  3, named balls
```

Load and Store Instructions (continued):

- We can create as many variables, with initial values, as we need for a program. I.e.,

```
bats:   .word  17
balls:  .word   3
gloves: .word  10
bases:  .word   4
```

- Suppose we want to perform some arithmetic on these values? To take a specific example:
    - Want to find the total of the **`bats + balls + gloves`**.
- To do arithmetic, we have to get the values from memory into registers. Thus, we need to put the values for **`bats`**, **`balls`**, and **`gloves`** into three registers.
- Where in memory?
    - The names **`bats`**, **`balls`**, etc. above are symbolic names that represent **locations** in memory.
        - Note: **`bats`** is a **location**, not a value.
    - To load a value from a location, we use the **`lw`** instruction (load word). **`lw`** has the form:

```
lw   destination-register, offset(register-with-address)
```

- The destination-register can be any of **`$s0-$s7`**, **`$t0-$t9`**.
- The register-with-address can also be any of these registers.
- How does the register-with-address get its value?

Load and Store Instructions (continued):

- Getting an address into a register:
  - The **la** instruction (load address) will put the address associated with a symbolic name into a register. I.e.,

```
la   $t0, bats   # puts the address of the bats memory location into register $t0
```

- Once the address is in a register, we can use that register for the register-with-address part of an **lw** instruction.

```
lw  $s0,  0($t0)      # gets value located at address stored in $t0 & puts it into $s0
```

  - These two instructions (**la** and **lw**) together will put the number stored at location **bats** into register **$s0**.
  - For now, we will use **0** for the offset every time.

- Now, we can write the code to find the sum: **bats + balls + gloves**:

```
la   $t0, bats       # $t0 now contains the address of the bats memory location
lw   $s0, 0($t0)     # $s0 now contains the number of bats

la   $t0, balls      # $t0 now contains the address of the balls memory location
lw   $s1, 0($t0)     # $s1 now contains the number of balls

la   $t0, gloves     # $t0 now contains the address of the gloves memory location
lw   $s2, 0($t0)     # $s2 now contains the number of gloves

add  $s3, $s0, $s1   # $s3 now contains the number of bats + balls
add  $s3, $s3, $s2   # $s3 now contains the number of bats + balls + gloves
```

- Our answer is now in register **$s3**.
- How do we get the answer from register **$s3** into a memory location?

Load and Store Instructions (continued):

- Copying a value from a register to a memory location is done with store (from slide #11):
  - **sw** will copy a word (32-bits).
  - **sh** will copy half a word (16-bits).
  - **sb** will copy one byte (8-bits).
- Concentrating on `sw` for now.
- `sw` has a similar format to the `lw` command:

```
sw   source-register, offset(register-with-address)
```

- Again, we need an address (a location) in memory. We get this using the `la` command.
- Need to declare a location to put our result. Will add this to the memory declarations for `bats`, `balls`, etc.
- Two step process to save a value:
  - Get the address of the location (using `la`).
  - Save the value into that location (using `sw`).

<u>Load and Store Instructions</u> (continued):

- A short (not yet complete) MIPS program, named *bats1.s*, to find: **`sum = bats + balls + gloves`**

```
.data

bats:    .word  17
balls:   .word   3
gloves:  .word  10
bases:   .word   4
sum:     .word   0    # Create a place to put our answer

.text

main:
        la   $t0, bats        # $t0 has the address of the bats memory location
        lw   $s0, 0($t0)      # $s0 now holds the number of bats
        la   $t0, balls       # $t0 has the address of the balls memory location
        lw   $s1, 0($t0)      # $s1 now holds the number of balls
        la   $t0, gloves      # $t0 has the address of the gloves memory location
        lw   $s2, 0($t0)      # $s2 now holds the number of gloves

        add  $s3, $s0, $s1    # $s3 now holds the number of bats+balls
        add  $s3, $s3, $s2    # $s3 now holds the number of bats+balls+gloves

        la   $t0, sum         # $t0 has the address of the sum memory location
        sw   $s3, 0($t0)      # sum now holds the number of bats+balls+gloves
```

`.data` marks a part of the program that defines memory locations.

`.text` marks a part of the program that contains assembly instructions (the program).

`main:` is a <u>required</u> label to tell the *spim* simulator where to begin executing.

Load and Store Instructions (continued):

- *bats2.s*: A complete version of the bats program. **`main`** is now a function that includes the proper way to start and end a function. For now, just put these lines into every **`main`** that you write, with the body of **`main`** in-between. We will cover later what these lines are actually doing, and why they are needed.

```
.data

bats:     .word  17
balls:    .word   3
gloves:   .word  10
bases:    .word   4
sum:      .word   0    # Create a place to put our answer

.text

main:     # Function prologue -- even main has one
          addiu $sp, $sp, -24   # allocate stack space -- default of 24 here
          sw    $fp, 0($sp)     # save caller's frame pointer
          sw    $ra, 4($sp)     # save return address
          addiu $fp, $sp, 20    # setup main's frame pointer

          la    $t0, bats       # $t0 has address of the bats memory location
          lw    $s0, 0($t0)     # $s0 now holds the number of bats
          la    $t0, balls      # $t0 has address of the balls memory location
          lw    $s1, 0($t0)     # $s1 now holds the number of balls
          la    $t0, gloves     # $t0 has address of the gloves memory location
          lw    $s2, 0($t0)     # $s2 now holds the number of gloves

          add   $s3, $s0, $s1   # $s3 now holds the number of bats+balls
          add   $s3, $s3, $s2   # $s3 now holds the number of bats+balls+gloves
```

Load and Store Instructions (continued):

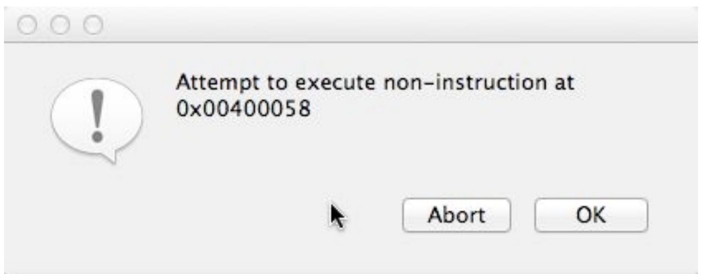
- **bats2.s**, continued:

```
        la    $t0, sum        # $t0 has address of the sum memory location
        sw    $s3, 0($t0)     # sum now holds the number of bats+balls+gloves

done:   # Epilogue for main -- restore stack & frame pointers and return
        lw    $ra, 4($sp)     # get return address from stack
        lw    $fp, 0($sp)     # restore the caller's frame pointer
        addiu $sp, $sp, 24    # restore the caller's stack pointer
        jr    $ra             # return to caller's code
```

Load and Store Instructions (continued):

- You can find the bats1.s and bats2.s examples from the previous 3 slides on D2L
  - We will be adding to this list of examples!
  - You can execute this program using:
    - QtSpim
      - Your own Windows/Linux/MacOSX machine.
      - lectura, Ubuntu machines in GS-930 and Mac's in GS-228.
- You will find, when you execute either program, that it runs to completion.
  - From **`bats1.s`**, you get an error:



  - Neither version prints anything!
    - We put the answer in a register and in memory.
    - We did not print it!

QtSpim and spim

- Reading: Textbook, Appendix B (4th edition), Appendix A (5th edition and from the Spim web page).
  - Appendix B in the 4th edition, and Appendix A from the spim web site are the same.
    - I refer to this as the "Spim Appendix".
  - Especially read section 9, pages 40 to 45.
  - Appendix A can also be found on the web at: http://www.cs.wisc.edu/~larus/HP_AppA.pdf
    - There is a link to this on-line version on the 252 class web page and in D2L.
- There are "Getting Started" guides for the command-line version of spim:
  - spim: http://www.cs.wisc.edu/~larus/spim.pdf
  - spim Command Line: http://www.cs.wisc.edu/~larus/SPIM_command-line.pdf
- You can also find these links on the 252 D2L page. Click on "Useful web links" under Content.

<u>QtSpim and spim</u> (continued):

- spim is a command-line tool usable on a UNIX system (Linux/OS X, among others).
- QtSpim is a GUI interface to spim that is usable on Windows/Linux/OSX.
- QtSpim displays:
  - Text, a scrollable window showing your code.
  - In Regs[16]: A scrollable window that shows:
    - Program Counter (PC) and status registers, among others.
    - General Registers: The 32 general-purpose registers.
  - Data: A scrollable window that shows:
    - User data segment (useful now).
    - User stack (will use this later).
  - FP Regs: A scrollable window that shows:
    - Floating Point Registers, both single- and double-precision registers

QtSpim and spim (continued):

- Register Display:
  - After *bats2.s* program has finished.
  - All register contents are displayed in hexadecimal.

$t0 has the hex address of sum

$s0 has bats

$s1 has balls

$s2 has gloves

$s3 has bats+balls+gloves

```
PC       = 400020
EPC      = 0
Cause    = 0
BadVAddr = 0
Status   = 3000ff10

HI       = 0
LO       = 0

R0  [r0] = 0
R1  [at] = 10010000
R2  [v0] = a
R3  [v1] = 0
R4  [a0] = 1
R5  [a1] = 7ffffe3c
R6  [a2] = 7ffffe44
R7  [a3] = 0
R8  [t0] = 10010010
R9  [t1] = 0
R10 [t2] = 0
R11 [t3] = 0
R12 [t4] = 0
R13 [t5] = 0
R14 [t6] = 0
R15 [t7] = 0
R16 [s0] = 11
R17 [s1] = 3
R18 [s2] = a
R19 [s3] = 1e
R20 [s4] = 0
R21 [s5] = 0
R22 [s6] = 0
```

QtSpim and spim (continued):

- Text Segment:
  - Displays the program code.

Address of instruction hexadecimal

MIPS machine language 32-bit, hexadecimal

Line number from your .s file

```
[00400024] 27bdffe8  addiu $29, $29, -24       ; 13: addiu $sp, $sp, -24 # allocate stack space -- default of 24 here
[00400028] afbe0000  sw $30, 0($29)            ; 14: sw $fp, 0($sp) # save caller's frame pointer
[0040002c] afbf0004  sw $31, 4($29)            ; 15: sw $ra, 4($sp) # save return address
[00400030] 27be0014  addiu $30, $29, 20        ; 16: addiu $fp, $sp, 20 # setup main's frame pointer
[00400034] 3c081001  lui $8, 4097 [bats]       ; 18: la $t0, bats # $t0 has address of the bats memory location
[00400038] 8d100000  lw $16, 0($8)             ; 19: lw $s0, 0($t0) # $s0 now holds the number of bats
[0040003c] 3c011001  lui $1, 4097 [balls]      ; 20: la $t0, balls # $t0 has address of the balls memory location
[00400040] 34280004  ori $8, $1, 4 [balls]
[00400044] 8d110000  lw $17, 0($8)             ; 21: lw $s1, 0($t0) # $s1 now holds the number of balls
[00400048] 3c011001  lui $1, 4097 [gloves]     ; 22: la $t0, gloves # $t0 has address of the gloves memory location
[0040004c] 34280008  ori $8, $1, 8 [gloves]
[00400050] 8d120000  lw $18, 0($8)             ; 23: lw $s2, 0($t0) # $s2 now holds the number of gloves
[00400054] 02119820  add $19, $16, $17         ; 24: add $s3, $s0, $s1 # $s3 now holds the number of bats+balls
[00400058] 02729820  add $19, $19, $18         ; 25: add $s3, $s3, $s2 # $s3 now holds the number of bats+balls+glove
[0040005c] 3c011001  lui $1, 4097 [sum]        ; 27: la $t0, sum # $t0 has address of the sum memory location
[00400060] 34280010  ori $8, $1, 16 [sum]
[00400064] ad130000  sw $19, 0($8)             ; 28: sw $s3, 0($t0) # sum now holds the number of bats+balls+gloves
[00400068] 8fbf0004  lw $31, 4($29)            ; 31: lw $ra, 4($sp) # get return address from stack
[0040006c] 8fbe0000  lw $30, 0($29)            ; 32: lw $fp, 0($sp) # restore the caller's frame pointer
[00400070] 27bd0018  addiu $29, $29, 24        ; 33: addiu $sp, $sp, 24 # restore the caller's stack pointer
[00400074] 03e00008  jr $31                    ; 34: jr $ra # return to caller's code
```

Assembly language instruction, after assembly

Assembly language instruction, from your .s file, before assembly

QtSpim and spim (continued):

- Data Segment:
    - Shows the contents of memory. This snapshot was taken after *bats2.s* stopped executing.

Contents of bytes, hexadecimal, grouped into words

bats

balls

gloves

bases

Addresses of bytes, hexadecimal

```
User data segment [10000000]..[10040000]
[10000000]..[10010000]  00000000[10010000]   00000011  00000003  0000000a  00000004
[10010010]    0000001e  00000000  00000000  00000000
[10010020]..[10040000]  00000000

User Stack [7ffffe38]..[80000000]
[7ffffe38]    00000001  7ffffe7c
[7ffffe40]    00000000  7fffffdd  7fffffa4  7fffff94
[7ffffe50]    7fffff80  7fffff73  7fffff63  7fffff40
[7ffffe60]    7fffff15  7ffffee0  7ffffeca  7ffffea8
```

sum

User Stack: ignore (for now)

<u>Printing in QtSpim and spim</u>:

- Need to connect to an outside device ("outside" the CPU).
- The simulator provides this for us via the System Call mechanism.
- There are 17 system calls. See Figure 9.1 on page 44 of the SPIM Appendix for the <u>complete list</u>.
- Here are <u>some</u> of the more useful ones (there are others you will need; look them up!):

Service	System call code	Arguments	Result
print_int	1	`$a0` = integer	
print_float	2	`$f12` = float	
print_double	3	`$f12` = double	
print_string	4	`$a0` = string	
read_int	5		integer (in `$v0`)
read_float	6		float (in `$f0`)
read_double	7		double (in `$f0`)
read_string	8	`$a0` = buffer, `$a1` = length	

Printing in QtSpim and spim (continued):

Printing Strings:

- We can declare character arrays using **.ascii**.
  - This is done in the **.data** section.
  - For example,

```
myChars:   .ascii  "abc7 d8-#$stuff8y wi"
gibberish: .ascii  "982(*&$junk(*&stuff"
words:     .ascii  "the of and or nor a an alphabet Zulu"
```

- A ***string*** is a character array with an additional character on the end. This last character is always the **nul** character (ascii value zero).
  - Declare a string using **.asciiz**. Examples:

```
myName:   .asciiz   "Patrick T. Homer"
morning:  .asciiz   "Good morning"
week:     .asciiz   "Su Mo Tu We Th Fr Sa"
```

Printing in QtSpim and spim (continued):

Printing Strings (continued):

- Declare one (or more) strings in the **.data** segment using **.asciiz**:

```
morning:  .asciiz   "Good morning"
myName:   .asciiz   "Patrick T. Homer"
week:     .asciiz   "Su Mo Tu We Th Fr Sa"
```

- Load the address of the string into **$a0**:

```
la   $a0, myName    # put the address of my name in register $a0
```

- Put the system call number for **print_string** into **$v0**.
  - Need a new instruction: add immediate:
    - Like **add**, but 3rd register is replaced with a positive or negative integer:

```
addi $v0, $zero, 4  # put system call number 4 into register $v0
```

- Call system:

```
syscall             # spim will now print the string that has my name
```

Printing in QtSpim and spim (continued):

- *HelloWorld1.s*

```
# Print the Hello World phrase.
# Here we load the base address of the string
# into register $a0, and use the print_string
# syscall to print the phrase.

.data

hello:  .asciiz "Hello World\n"

.text

main:    # Function prologue -- even main has one
         addiu $sp, $sp, -24   # allocate stack space -- default of 24 here
         sw    $fp, 0($sp)     # save caller's frame pointer
         sw    $ra, 4($sp)     # save return address
         addiu $fp, $sp, 20    # setup main's frame pointer

         # set up $a0 to hold address of the hello world string
         # then print the string
         la      $a0, hello    # Point to the string
         addi    $v0, $zero, 4 # syscall value for print_string
         syscall

done:    # Epilogue for main -- restore stack & frame pointers and return
         lw    $ra, 4($sp)     # get return address from stack
         lw    $fp, 0($sp)     # restore the caller's frame pointer
         addiu $sp, $sp, 24    # restore the caller's stack pointer
         jr    $ra             # return to caller's code
```

Printing in QtSpim and spim (continued):

- *HelloWorld1.s*: Use the step command to advance to the **`syscall`** on line 25.

```
PC       = 40003c
EPC      = 0
Cause    = 0
BadVAddr = 0
Status   = 3000ff10

HI       = 0
LO       = 0

R0  [r0] = 0
R1  [at] = 0
R2  [v0] = 4
R3  [v1] = 0
R4  [a0] = 10010000
R5  [a1] = 7ffffe3c
R6  [a2] = 7ffffe44
R7  [a3] = 0
R8  [t0] = 0
R9  [t1] = 0
R10 [t2] = 0
```

`print_str` system call

Address of hello string

```
User data segment [10000000]..[10040000]
[10000000]..[10010000]  00000000[10010000]    6c6c6548  6f57206f  0a646c72  00000000  H e l l o W o r l d
[10010010]..[10040000]  00000000
                                               l l e H   o W   o   \n d l r   nul

User Stack [7ffffe38]..[80000000]
[7ffffe38]    00000001  7ffffe76
[7ffffe40]    00000000  7fffffdd  7fffffa4  7fffff94
[7ffffe50]    7fffff80  7fffff73  7fffff63  7fffff40            s      c       @
```

Printing in QtSpim and spim (continued):

- *HelloWorld2.s*: Can break the string into multiple parts:

Note: The prologue and epilogue of main are missing here. They are present in the *HelloWorld2.s* example that is on D2L.

```
.data
hello:   .asciiz "Hello"
space:   .asciiz " "
world:   .asciiz "World"
newline: .asciiz "\n"
.text
         # set up and print the string "Hello"
         la      $a0, hello      # Point to the string
         addi    $v0, $zero, 4   # syscall value for print_string
         syscall
         # set up and print the string " "
         la      $a0, space      # Point to the string
         addi    $v0, $zero, 4   # syscall value for print_string
         syscall
         # set up and print the string "World"
         la      $a0, world      # Point to the string
         addi    $v0, $zero, 4   # syscall value for print_string
         syscall
         # set up and print the string "\n"
         la      $a0, newline    # Point to the string
         addi    $v0, $zero, 4   # syscall value for print_string
         syscall
```

Printing in QtSpim and spim (continued):

- *HelloWorld2.s*: Can break the string into multiple parts:

```
.data
hello:   .asciiz "Hello"
space:   .asciiz " "
world:   .asciiz "World"
newline: .asciiz "\n"
```

```
                                                  l l e H \0    \0 o l r o W  \0 \n \0 d
User data segment [10000000]..[10040000]
[10000000]..[10010000]  00000000[10010000]    6c6c6548  0020006f  6c726f57  000a0064  H e l l o   W o r l d
[10010010]..[10040000]  00000000

User Stack [7ffffe38]..[80000000]
[7ffffe38]      00000001  7ffffe76
[7ffffe40]      00000000  7fffffdd  7fffffa4  7fffff94
[7ffffe50]      7fffff80  7fffff73  7fffff63  7fffff40       s      c      @
```

Program Flow Control

- Assembly language supports a much smaller set of flow control instructions compared to high-level programming languages:
  - What assembly language does not have:
    - **`for`**, **`while`**, **`do...while`**, **`switch`**
  - What assembly language does have:
    - Unconditional branch.
    - Conditional branch.
    - Function calls.

<u>Unconditional branch</u>.

The jump instruction:

```
j    label
```

- Example:

```
            add  $t0, $s1, $s2
            j    toThere
            add  $t0, $t0, $s1  # This instruction is skipped
            add  $s1, $s2, $s2  # This instruction is skipped
toThere:    add  $s7, $s1, $s3
```

Conditional branch.

- Chooses between two control flows. Either
  - Execute the next instruction, or
  - Jump to a different instruction.
- MIPS has two conditional branch instructions:
  - Branch if equal:
    - **beq** register, register, Label
  - Branch if not equal:
    - **bne** register, register, Label
- Example:

```
if ( i == j )                        bne $s0, $s1, downThere  # if (i != j)
   h =  i + j;                       add $s3, $s0, $s1        #    h = i + j
z = h + h;            downThere: add $s4, $s3, $s3        # z = h + h
```

  - Note: reversal of the condition from equality to inequality!
    - This is a common technique.

Conditional branch (continued):

- Example, version 1:

```
 if ( i != j )              bne $s4, $s5, Lab1   # compare i, j
                            j Lab2               # skip true part
   h = i + j;         Lab1:
                            add $s3, $s4, $s5    # h = i + j
                            j Lab3               # skip false part
else                  Lab2:
   h = i - j;               sub $s3, $s4, $s5    # h = i - j

k = h + i;            Lab3:
                            add $s6, $s3, $4     # k = h + i
```

  - The statement labeled **`Lab1:`** is executed <u>only</u> when **`$s4`** is <u>not</u> equal to **`$s5`**.
  - The statement labeled **`Lab2:`** is executed only when **`$s4`** is equal to **`$s5`**.
  - The statement labeled **`Lab3:`** is executed <u>always</u>.
- Need to "skip the true part" if the **`bne`** is false; hence the need for **`j Lab2`**.
- Confusing structure, prone to errors.

Conditional branch (continued):

- Example, version 2:

```
if ( i != j )                beq $s4, $s5, Lab1   # compare i, j
   h = i + j;                add $s3, $s4, $s5    # h = i + j
                             j   Lab2             # skip false part

else                   Lab1:
   h = i - j;                sub $s3, $s4, $s5    # h = i - j

k = h + i;             Lab2:
                             add $s6, $s3, $s4    # k = h + i
```

- The reversed condition makes this easier to read, and less prone to errors.
  - Allows the assembly code to more closely follow the pattern from the higher-level language.
- The statement labeled **`Lab1:`** is executed only when **`$s4`** is equal to **`$s5`**.
- The statement labeled **`Lab2:`** is executed always.

Conditional branch (continued):

- Comparisons other than equal or not equal?
- New instruction: **Set Less Than**.
  - Compares two registers and puts the result in the destination register:

    ```
    slt   $t0, $s4, $s5
    ```

  - First operand, `$t0` in this case, is the destination of the result of the comparison.
    - Note: this follows the pattern of all the MIPS arithmetic instructions.
  - Second and third operands are compared: `$s4 < $s5`
  - Result is `1` if the comparison is true.
  - Result is `0` if the comparison is false.
- Example:

```
if ( x < y )                  slt $t0, $s2, $s3      # $t0 = (x < y)
                              beq $t0, $zero, yLess
   z = x;                     add $s5, $s2, $zero    # z = x
                              j   after
else                   yLess:
   z = y;                     add $s5, $s3, $zero    # z = y

w = z * 2;             after:
                              add $s6, $s5, $s5      # w = z * 2
```

```
# Basic idea:

# if (  $s3 ?? $s5 )  goto  Athos
slt   $t0, _____, _____
b__   $t0, $zero, Athos



# less than:
# if ($s5 < $t7) goto otherPlace

slt $t0, $s5, $t7
bne $t0, $zero, otherPlace



# greater than:
#    reverse the order
# if ( $s4 > $s1 ) goto Batman

# becomes
# if ( $s1 < $s4 ) goto Batman

slt  $t2, $s1, $s4
bne  $t2, $zero, Batman
```

```
# Basic idea:

# if (  $s3 ?? $s5 )  goto  Athos
slt   $t0, _____, _____
b__   $t0, $zero, Athos
```

```
# less than or equal to:
#    reverse the meaning, use greater than

# if ( $s3 <= $s2 ) goto Zorro
# becomes:
# if ( !($s3 > $s2) ) goto Zorro

# need to reverse the order to remove !

# if ( $s2 < $s3 ) goto Zorro
slt  $t1, $s2, $s3
bne  $t1, $zero, Zorro

# Can use a similar approach for >=
# But, this can be confusing(!!)
#      and prone to errors
```

```
# Alternative approach when ='s is present:

# if ( $s3 <= $s2 ) goto Zorro

# Two parts:

# First: if ( $s3 == $s2 ) goto Zorro
beq  $s3, $s2, Zorro

# Second: if ( $s3 < $s2 ) goto Zorro
slt $t2, $s3, $s2
bne $t2, $zero, Zorro
```

```
# if ( $s3 >= $s5 ) goto Porthos
# First part, take care of ='s
beq  $s3, $s5, Porthos

# Then, we have:
# if ( $s3 > $s5 ) goto Porthos
# reverse the order to get
# if ( $s5 < $s3 ) goto Porthos
slt  $t0, $s5, $s3
bne  $t0, $zero, Porthos
```

- MIPS Instruction summary to date:

Instruction	Meaning
`add $s1, $s2, $s3`	\$s1 = \$s2 + \$s3
`sub $s1, $s2, $s3`	\$s1 = \$s2 - \$s3
`addi $t0, $s3, Number`	add immediate: \$t0 = \$s3 + Number
`la $s2, label`	\$s2 = address of label, *pseudo-instruction*
`lw $s1, 0($s2)`	\$s1 = Memory[\$s2], load word in Memory at position \$s2 into \$s1
`sw $s1, 0($s2)`	Memory[\$s2] = \$s1, store word in \$s1 into Memory at position \$s2
`slt $t0, $s3, $s4`	\$t0 = \$s3 < \$s4, put 1 in \$t0 if true, else put 0 in \$t0
`bne $s4, $s5, aLabel`	Next instruction executed is at aLabel if \$s4 ≠ \$s5
`beq $s4, $s5, bLabel`	Next instruction executed is at bLabel if \$s4 == \$s5
`j cLabel`	Next instruction executed is at cLabel

For Loops.

- C (or Java) code:

```
sum = 0;
for (i = 0; i < y; i++)
   sum = sum + x;
```

- MIPS:
  - Assume: **$s0** has **x**, **$s1** has **y**, and **$s2** (will have) **sum**.
  - Will use **$t0** for **i**.

initialize loop index

condition

increment

```
        add  $s2, $zero, $zero    # sum = 0
        addi $t0, $zero, 0        # i = 0, initial value set before loop begins
LoopBegin:
        # for loop does comparison at beginning of each iteration
        slt  $t2, $t0, $s1        # is i < y  ??
        beq  $t2, $zero, LoopEnd  # branch below end of loop if done
        # loop body
        add  $s2, $s2, $s0        # sum = sum + x
        # increment loop index
        addi $t0, $t0, 1          # i++
        j    LoopBegin
LoopEnd:
        # rest of program goes here...
```

For Loops (continued):

- Complete MIPS program: for loop example. Available as *for1.s* on D2L.

```
.data

x:        .word  42
y:        .word   8
sum:      .word   0

answer:   .asciiz  "The sum is "
newline: .asciiz "\n"

.text

main:     # Function prologue -- even main has one
          addiu $sp, $sp, -24        # allocate stack space -- default of 24 here
          sw    $fp, 0($sp)          # save caller's frame pointer
          sw    $ra, 4($sp)          # save return address
          addiu $fp, $sp, 20         # setup main's frame pointer

          # Put x into $s0
          la    $t0, x
          lw    $s0, 0($t0)

          # Put y into $s1
          la    $t0, y
          lw    $s1, 0($t0)

          add   $s2, $zero, $zero    # sum = 0
          add   $t0, $zero, $zero    # i = 0
```

For Loops (continued):

```
LoopBegin:
          # for loop does comparison at beginning of each iteration
          slt   $t2, $t0, $s1       # is i < y ??
          beq   $t2, $zero, LoopEnd # branch below end of loop if done

          # loop body
          add   $s2, $s2, $s0       # sum = sum + x

          # increment loop index
          addi  $t0, $t0, 1         # i++
          j     LoopBegin

LoopEnd:
          # Print message
          la    $a0, answer
          addi  $v0, $zero, 4
          syscall

          # Print the sum
          add   $a0, $s2, $zero
          addi  $v0, $zero, 1
          syscall

          # Print newline
          la    $a0, newline
          addi  $v0, $zero, 4
          syscall
```

For Loops (continued):

```
done:    # Epilogue for main -- restore stack & frame pointers and return
         lw    $ra, 4($sp)          # get return address from stack
         lw    $fp, 0($sp)          # restore the caller's frame pointer
         addiu $sp, $sp, 24         # restore the caller's stack pointer
         jr    $ra                  # return to caller's code
```

For Loops (continued):

- Print the integers in an array where the number of elements in the array is known.
- First problem to understand: How to represent an array in MIPS?
  - Handled by declaring memory in the **.data** section.
  - The label **numElements** tells us how many values are stored in the array.
  - The array has only one label, which is the name of the array.
    - The label identifies the beginning of the array.
  - Example:

```
.data
numElements:
          .word  7

elements:
          .word 55
          .word 66
          .word 77
          .word 0
          .word -16
          .word -19
          .word 82
```

- Arrays in assembly differ from arrays in Java in two very important ways:
  - Nothing marks the end of an array.
  - Nothing indicates how many elements are in the array.

- Sets up an array:
  - Named **elements**.
  - 7 locations, each of size 4 bytes (a **word**).
  - Each position has an initial value.

For Loops (continued):

- We need:
  - A register to hold the loop index, **`i`**.
  - A register to hold the number of elements in the array.
  - A register to hold the starting address of the array.

```
addi    $s1, $zero, 0   # $s1 = i = 0

la      $t0, numElements
lw      $s2, 0($t0)     # $s2 = numElements

la      $t0, elements   # $t0 = address of elements[0]
```

- To get one element of the array:
  - Compute the offset (number of bytes) from the beginning of the array to the desired integer.
  - Each integer is 4 bytes (one word).
  - Can multiply **`i`** by **`4`**, then add to the starting address of the array:

```
add     $t1, $s1, $s1
add     $t1, $t1, $t1   # $t1 = 4 * i
add     $t2, $t0, $t1   # $t2 = address of elements[i]
lw      $a0, 0($t2)     # $a0 = elements[i]
```

  - Note: it is faster to do two **`add`**'s than to do one multiply.

For Loops (continued):

- Complete MIPS program: array print example. Available as *for2.s* on D2L.

```
# Print the values of an array using a for loop.
.data
numElements:
         .word  7

elements:    #   la   $t0, elements
         .word 55     #  lw  $t7, 0($t0)
         .word 66     #  lw  $t7, 4($t0)
         .word 77     #  lw  $t6, 8($t0)
         .word 0
         .word -16
         .word -19
         .word 82

newline:
         .asciiz "\n"

.text
main:
        # Function prologue -- even main has one
        subu  $sp, $sp, 24    # allocate stack space -- default of 24 here
        sw    $fp, 0($sp)     # save caller's frame pointer
        sw    $ra, 4($sp)     # save return address
        addiu $fp, $sp, 20    # setup main's frame pointer
```

For Loops (continued):

- Complete MIPS program: array print example. Available as *for2.s* on examples from class web page and D2L.

```
        # for ( i = 0; i < numElements; i++ )
        #     print elements[i]
        addi    $s1, $zero, 0    # i = 0
        la      $t0, numElements
        lw      $s2, 0($t0)      # $s2 = numElements
        la      $t0, elements    # $t0 = address of elements[0]
loopBegin:
        # test if for loop is done
        slt     $t1, $s1, $s2    # $t1 = i < numElements
        beq     $t1, $zero, loopEnd
        # Compute address of elements[i]
        add     $t1, $s1, $s1
        add     $t1, $t1, $t1    # $t1 = 4 * i
        add     $t2, $t0, $t1    # $t2 = address of elements[i]
        lw      $a0, 0($t2)      # $a0 = elements[i]
        addi    $v0, $zero, 1
        syscall
        # print newline
        la      $a0, newline
        addi    $v0, $zero, 4
        syscall
```

Write a pseudo-code version of your algorithm.
Then, put it in your code as comments.
We will look for these in your programs.

For Loops (continued):

- *for2.s* continued:

```
            addi    $s1, $s1, 1     # i++
            j       loopBegin

    loopEnd:

    done:   # Epilogue for main -- restore stack & frame pointers and return
            lw    $ra, 4($sp)      # get return address from stack
            lw    $fp, 0($sp)      # restore the caller's frame pointer
            addiu $sp, $sp, 24     # restore the caller's stack pointer
            jr    $ra              # return to caller's code
```

While Loops:

- Consider the while loop:

```
while ( save[i] != -2 )      Stop the loop when we find a -2
   if ( save[i] == k )       How many elements in the array save are equal to k?
      count += 1;
   i = i + 1;
```

- Here is a complete MIPS program to solve this problem. Available as *while1.s* on the class web page and D2L.

```
.data

save:    .word   19          # save[0] = 19
         .word   42          # save[1] = 42
         .word   42          # save[2] = 42
         .word   42          # save[3] = 42
         .word   42          # save[4] = 42
         .word   42          # save[5] = 42
         .word   42          # save[6] = 42
         .word   93          # save[7] = 93
         .word   -2          # save[8] = -2
k:       .word   42          # number within save that we are looking for

str:     .asciiz "The final value of count = "
newline: .asciiz "\n"
```

While loops (continued):

```
.text
main:    # Function prologue -- even main has one
         addiu $sp, $sp, -24     # allocate stack space -- default of 24 here
         sw    $fp, 0($sp)       # save caller's frame pointer
         sw    $ra, 4($sp)       # save return address
         addiu $fp, $sp, 20      # setup main's frame pointer

         la    $s6, save         # $s6 = address of save[0], beginning of array

         add   $s3, $zero, $zero # initial value of i is 0

         la    $t0, k
         lw    $s5, 0($t0)       # $s5 = value of k

         addi  $s1, $zero, -2    # Put ending value of -2 into $s1
         add   $s2, $zero, $zero # $s2 = count = 0;  start count at zero

LoopBegin:
         # Loop Test, stop loop when we find a -2 in save[i]
         add   $t1, $s3, $s3     # quadruple i to get offset for save[i]
         add   $t1, $t1, $t1
         add   $t1, $t1, $s6     # compute address of save[i]
         lw    $t0, 0($t1)       # $t0 = value stored at save[i]
         beq   $t0, $s1, LoopEnd # end loop if save[i] == -2
```

Note: The code for the if statement is <u>entirely</u> contained within the body of the loop. Do **<u>not</u>** jump outside the loop!

<u>While loops</u> (continued):

```
          # Loop body
          # if ( save[i] == k )
          #    count++
          bne   $t0, $s5, afterIncrement  # if ( save[i] != k )
          addi  $s2, $s2, 1        # count += 1

afterIncrement:
          addi  $s3, $s3, 1        # i++

          j     LoopBegin          # back to start of loop

LoopEnd:
          # Print count with a label
          la    $a0, str           # $a0 = address of start of string
          addi  $v0, $zero, 4
          syscall

          add   $a0, $s2, $zero    # $a0 = value of count
          addi  $v0, $zero, 1
          syscall

          la    $a0, newline       # $a0 = address of newline string
          addi  $v0, $zero, 4
          syscall
```

While loops (continued): The WRONG way to do an if statement that is inside a loop

```
        # Loop body
        # if ( save[i] == k )
        #   count++
        beq   $t0, $s5, increment  # if ( save[i] != k )

afterIncrement:
        addi  $s3, $s3, 1        # i++

        j     LoopBegin          # back to start of loop

LoopEnd:
        # Print count with a label
        la    $a0, str           # $a0 = address of start of string
        addi  $v0, $zero, 4
        syscall

        add   $a0, $s2, $zero    # $a0 = value of count
        addi  $v0, $zero, 1
        syscall

        la    $a0, newline       # $a0 = address of newline string
        addi  $v0, $zero, 4
        syscall
        j   done

increment:   addi  $s2, $s2, 1        # count += 1
        j   afterIncrement
```

```
if ( x .lt. y ) goto 53

if ( x ) 42, 93, 16
```

done:
on next
page

While loops (continued):

```
done:    # Epilogue for main -- restore stack & frame pointers and return
         lw    $ra, 4($sp)       # get return address from stack
         lw    $fp, 0($sp)       # restore the caller's frame pointer
         addiu $sp, $sp, 24      # restore the caller's stack pointer
         jr    $ra               # return to caller's code
```

<u>Other conditions for branching</u>:

- So far, we have **`beq`** and **`bne`** for branching, and we have **`slt`** for comparing (without branching).
- What about other possibilities? Branch on less than, branch on greater than, branch on less than or equal, etc?
- Can have an assembly language that provides all of these options.
  - MIPS does not! (I am not counting pseudo-instructions.)
- Can use **`slt`** with **`bne`** or **`beq`** to build the missing branch instructions.
- Example:
  - Branch if greater than (assume **`x`** is in **`$s1`**, and **`y`** is in **`$s2`**):

```
if ( x > y ) then                     slt  $t0, $s2, $s1   # $t0 = $s2 < $s1?  $t0 = y < x?
                                      beq  $t0, $zero, toElse
   z = x - y                          sub  $s4, $s1, $s2   # z = x - y
                                      j    afterIf
else                          toElse:
   z = y - x                          sub  $s4, $s2, $s1  # z = y - x
                              afterIf:
                                      # continue program here
```

  - Note: The order of the comparison was reversed.
    - The question becomes: Is **`not(y < x)`**? If so, the branch goes to the else clause.
    - Reversing the meaning of the comparison is a common technique!

<u>Constant or Immediate Operands</u>:

- Many times, one operand of an arithmetic instruction is a small constant integer.
  - Can occur in 50% or more of the MIPS instructions in some programs!
- Possible solutions:
  - Put typical constants in memory and load them into a register when needed.
    - Can be quite slow to perform the `lw` operation so many times in a program!
  - Hard-wire some of the registers to hold the most commonly used constants.
    - The `$zero` register in MIPS.
    - Difficult in general to decide which constants (other than zero) are most needed.

```
.data
twenty: .word 20
.text
la   $t0, twenty
lw   $s3, 0($t0)
slt  $t0, $s2, $s3
```

- MIPS Solution (and something similar is done in many other assembly languages):
  - Add "a few" instructions that allow one operand (of the three) to be stored in the instruction itself.
    - `addi`
    - `slti`
  - Have already been using `addi`.

- `add  $zero, $s7, $s2`
- `addi $s7, $s2, -17`
- `slti $t0, $s2, 20`
- `-32,768 to 32,767`
- `slti $t0, 20, $s2  # does NOT work!!`

Constant or Immediate Operands (continued):

- Examples:

```
addi  $t1, $s3, 4       # $t1 = $s3 + 4
addi  $t7, $s2, -27     # $t7 = $s2 + (-27)
slti  $t2, $s4, -8      # $t2 = $s4 < -8, $t2 == 0 if false,  $t2 == 1 if true
```

- The immediate operand in all these cases is a 16-bit, signed, two's complement integer.
  - Thus, the range for the immediate operand is: -32,768 to +32,767.
- Note: there is no **`subi`** instruction. Why?

- **`la`** is a *pseudo-instruction*. It will accept an argument as large as a 32-bit, signed, two's complement integer.
  - The argument is assumed to be a label.
  - The address associated with the label is put in the specified register.
  - The assembler (spim in our case) will substitute one or two instructions for the **`la`**.
  - How many instructions are substituted depends on the address of the label.

```
la $t3, numbers
```

- MIPS Instruction summary to date:

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add  $s1, $s2, $s3`	`$s1 = $s2 + $s3`	3 operands; data in reg
	add immediate	`addi $s1, $s2, 17`	`$s1 = $s2 + 17`	3 operands; data in reg
	subtract	`sub  $s1, $s2, $s3`	`$s1 = $s2 - $s3`	3 operands; data in reg
Data transfer	load word	`lw   $s1, 0($s2)`	`$s1 = Memory[$s2]`	Data fm memory to reg
	store word	`sw   $s3, 0($t5)`	`Memory[$t5] = $s3`	Data fm reg to memory
Conditional branch	branch on equal	`beq  $s1,$s2,Label`	`if ($s1 == $s2) goto Label`	Equal test and branch
	branch on not equal	`bne  $s1,$s2,Label`	`if ($s1 ≠ $s2) goto Label`	Not equal test & branch
	set on less than	`slt  $s1, $s2, $s3`	`if ($s2 < $s3) $s1 = 1; else $s1 = 0`	Compare less than
	set on less than imm	`slti $s1, $s2, num`	`if ($s2 < num) $s1 = 1; else $s1 = 0`	Compare less than w/ immediate operand
Unconditional branch	jump	`j    Label`	`goto Label`	Branch to target address

Machine Language

Reading: Section 2.5 (4th edition).

- Instructions have to be stored in memory. It is a design choice as to how to represent the instructions.
- MIPS makes the choice to use a 32-bit value to represent instructions.
  - This choice could be different.
  - Some instruction sets use different sizes for different instructions.
    - Can result in multiple reads from memory to get one instruction into the CPU.
    - Example: The Motorola 68000 chip uses instruction sizes that vary in length from 16 to 80 bits.

Machine Language (continued):

- Repeating Slide #23:
- Text Segment:
  - Displays the program code.

Address of instruction hexadecimal

MIPS machine language 32-bit, hexadecimal

Line number from your .s file

```
[00400024] 27bdffe8  addiu $29, $29, -24        ; 13: addiu $sp, $sp, -24 # allocate stack space -- default o
[00400028] afbe0000  sw $30, 0($29)             ; 14: sw $fp, 0($sp) # save caller's frame pointer
[0040002c] afbf0004  sw $31, 4($29)             ; 15: sw $ra, 4($sp) # save return address
[00400030] 27be0014  addiu $30, $29, 20         ; 16: addiu $fp, $sp, 20 # setup main's frame pointer
[00400034] 3c081001  lui $8, 4097 [bats]        ; 18: la $t0, bats # $t0 has address of the bats memory locat
[00400038] 8d100000  lw $16, 0($8)              ; 19: lw $s0, 0($t0) # $s0 now holds the number of bats
[0040003c] 3c011001  lui $1, 4097 [balls]       ; 20: la $t0, balls # $t0 has address of the balls memory loc
[00400040] 34280004  ori $8, $1, 4 [balls]
[00400044] 8d110000  lw $17, 0($8)              ; 21: lw $s1, 0($t0) # $s1 now holds the number of balls
[00400048] 3c011001  lui $1, 4097 [gloves]      ; 22: la $t0, gloves # $t0 has address of the gloves memory l
[0040004c] 34280008  ori $8, $1, 8 [gloves]
[00400050] 8d120000  lw $18, 0($8)              ; 23: lw $s2, 0($t0) # $s2 now holds the number of gloves
[00400054] 02119820  add $19, $16, $17          ; 24: add $s3, $s0, $s1 # $s3 now holds the number of bats+ba
[00400058] 02729820  add $19, $19, $18          ; 25: add $s3, $s3, $s2 # $s3 now holds the number of bats+ba
[0040005c] 3c011001  lui $1, 4097 [sum]         ; 27: la $t0, sum # $t0 has address of the sum memory locatio
[00400060] 34280010  ori $8, $1, 16 [sum]
[00400064] ad130000  sw $19, 0($8)              ; 28: sw $s3, 0($t0) # sum now holds the number of bats+balls
[00400068] 8fbf0004  lw $31, 4($29)             ; 31: lw $ra, 4($sp) # get return address from stack
[0040006c] 8fbe0000  lw $30, 0($29)             ; 32: lw $fp, 0($sp) # restore the caller's frame pointer
[00400070] 27bd0018  addiu $29, $29, 24         ; 33: addiu $sp, $sp, 24 # restore the caller's stack pointer
[00400074] 03e00008  jr $31                     ; 34: jr $ra # return to caller's code
```

Assembly language instruction, after assembly

Assembly language instruction, from your .s file, before assembly

Machine Language (continued):

- For example:

  `add $t1, $s1, $s2`

  - Registers are represented in the machine instruction as numbers in the range 0..31.
    - **`$t1`** is register 9; **`$s1`** is register 17; **`$s2`** is register 18

32 bits

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct
000000	10001	10010	01001	00000	100000
0_{ten}	17_{ten}	18_{ten}	9_{ten}	0_{ten}	32_{ten}

  - Field abbreviations: SPIM Appendix section 10.2, page 50.
    - **op** — basic operation of the instruction, the *operation code* or *opcode*.
    - **rs** — first register source operand.
    - **rt** — second register source operand.
    - **rd** — register destination operand; the result of the operation goes into this register.
    - **shamt** — shift amount (we will see the use of this field later, for now it will always be zero).
    - **funct** — function. Selects the specific variant of the opcode. (Figure 10.2, page 50 in SPIM Appendix).

Machine Language (continued):

- Other points to notice about the layout:

← 32 bits →

31 26	25 21	20 16	15 11	10 6	5 0
op	rs	rt	rd	shamt	funct

← 16 bits → ← 16 bits →

  - It is no accident that there are 16 bits with three fields in each half of the format.
  - The Most Significant Bit (MSB) is on the **opcode** end of the instruction.
  - The Least Significant Bit (LSB) is on the **funct** end of the instruction.
  - This instruction format is called R-type.
  - This format is used for most of the arithmetic instructions.
    - The main exception are the arithmetic instructions that take immediate operands.

Machine Language (continued):

- More R-format instruction examples:
- Set-Less-Than:

```
slt   $t0, $t1, $t2
```

  - Register `$t0` is `$8`; Register `$t1` is `$9`; Register `$t2` is `$10`.
  - The opcode is 0 (again); the funct field is different from the **add**.
    - This is the only way to tell it is an **slt** rather than an **add**.

op	rs	rt	rd	shamt	funct
000000	01001	01010	01000	00000	101010
0_{ten}	9_{ten}	10_{ten}	8_{ten}	0_{ten}	42_{ten}

Machine Language (continued):

- Subtract:
  - There are two versions of subtract:
    - **sub** assumes the values in the registers are signed numbers.
    - **subu** assumes the values in the registers are unsigned numbers.

```
sub  $s1, $t3, $t4            # $s1 is $17; $t3 is $11; $t4 is $12
```

op	rs	rt	rd	shamt	funct
000000	01011	01100	10001	00000	100010
0_{ten}	11_{ten}	12_{ten}	17_{ten}	0_{ten}	34_{ten}

```
subu $s1, $t3, $t4            # $s1 is $17; $t3 is $11; $t4 is $12
```

op	rs	rt	rd	shamt	funct
000000	01011	01100	10001	00000	100011
0_{ten}	11_{ten}	12_{ten}	17_{ten}	0_{ten}	35_{ten}

Machine Language (continued):

Arithmetic instructions with Immediate operands:

- Major difference: Each instruction has two registers and an immediate value (signed, 16-bit, 2's complement integer).
- Need 16 bits (out of a total of 32 bits) for the immediate operand:
    - We can make use of the shamt field: 5 bits.
    - We only need two registers, not three. This gains: 5 bits.
    - Total so far: 10 bits. But we need 16…
- Solution: Use a different opcode for each immediate instruction.
    - Gains the funct field: 6 bits.
    - Total is now: 16 bits.
- This is the I-format instruction.
- Puts the immediate operand in the second half (right half) of the instruction:

```
addi  $s7, $t0, 321    # $s7 is $23; $t0 is $8
```

op	rs	rt	16 bit number
001000	01000	10111	0000 0001 0100 0001
8_{ten}	8_{ten}	23_{ten}	321_{ten}

Machine Language (continued):

More I-format instructions:

- Conditional branch statements need to include an address.
- Addresses are 32-bit values in MIPS.
- Dilemma: How to fit the opcode, two registers, and an address into a 32-bit MIPS instruction?
  - Use PC-relative addressing.
    - Address specified in the conditional branch is relative to the current value of the Program Counter.
    - Use the specified address as a 16-bit, two's complement value. Add this to the current value of the program counter.
    - Allows forward or backward branching.
    - Can not branch to locations that are "too far away."
    - Optimization:
      - All MIPS instructions are 32-bit. They are all word-aligned (start on an address divisible by 4).
      - Let the offset in the conditional branch instruction be an "instruction" offset, rather than a byte offset. That is, the offset is multiplied by 4 before being added to the Program Counter.

Machine Language (continued):

More I-format instructions:

- Example of conditional branch in I-format:

```
            beq   $t0, $s1, toThere    # $t0 is $8; $s1 is $17
            add   $t0, $s7, $s7        # $t0 = 2 * $s7
            j     below
toThere:    sub   $t0, $s7, $s5  # $t0 = $s7 - $s5
below:      add   $t1, $t1, $t0  # $t1 = $t1 + $t0
```

op	rs	rt	16 bit number
000100	01000	10001	0000 0000 0000 0010
4_{ten}	8_{ten}	17_{ten}	2_{ten}

- The Program Counter contains the address of the next instruction.
- Thus, to branch to the label **toThere** we add two instructions: PC + 2 * (size of a MIPS instruction).

Machine Language (continued):

More I-format instructions — **`lw`** and **`sw`**:

- The **`lw`** and **`sw`** instructions use only two registers (similar to **`bne`** and **`beq`** in this respect).
- They also use a 16-bit, two's complement number; the offset.
- Unlike **`beq`**, the offset for **`lw`** and **`sw`** is a count of bytes, not words.

```
lw   $s7, 0($s2)     # $s7 is $23; $s2 is $18
```

op	rs	rt	16 bit number
100011	10010	10111	0000 0000 0000 0000
35_{ten}	18_{ten}	23_{ten}	0_{ten}

```
sw   $s3, 168($s1)   # $s3 is $19; $s1 is $17
```

op	rs	rt	16 bit number
101011	10001	10011	0000 0000 1010 1000
43_{ten}	17_{ten}	19_{ten}	168_{ten}

Machine Language (continued):

Unconditional branch and the J-Format:

- The jump instruction takes a single argument, which is the address of an instruction.
- The instruction has to fit within 32-bits (as do all MIPS instructions).
    - Need an opcode: 6 bits.
    - Need an address: 32 bits.
    - That's 38 bits(!!)
    - Can not use a 32-bit address.
- The address of an instruction is always a multiple of 4.
    - Addresses that are multiplies of 4 always have two binary zeroes on the right.
    - Don't store the two zeroes.
- Now:
    - Need an opcode: 6 bits.
    - Need an address: 30 bits.
    - That's 36 bits; closer, but not quite there yet.

Machine Language (continued):

Unconditional branch and the J-Format (continued):

- So far:

```
j      0x0040 0240

0x0040 0240 = 0000 0000 0100 0000 0000 0010 0100 0000two
```

- Take the left-most 4 bits from the PC:

Dropped

```
j      0x0040 0240

0x0040 0240 = 0000 0000 0100 0000 0000 0010 0100 0000two
```

From the PC+4

op	26 bit, unsigned number
000010	00 0001 0000 0000 0000 1001 0000
2_{ten}	0x010 0090 = $1{,}048{,}720_{ten}$

- Allows a **j** instruction to go to any instruction that has the same high-order 4 bits as the PC.
  - Divides the 4GB address space into 16 segments, each containing (how many?) bytes.

Logical Operations

Reading: Section 2.6 (4th edition), 2.5 (3rd edition).

- AND:

```
and   $t0, $s1, $s2
```

  - Performs a *bit-wise* AND operation:

```
            $s1 = 0010 0110 0110 0110 1111 1111 0000 1010
            $s2 = 1111 0000 0000 0000 1111 0000 1111 1111
    $s1 AND $s2 = 0010 0000 0000 0000 1111 0000 0000 1010
```

  - For Java and C, this operation is done with `&`
    - Example:

```
x = y & z;
```

op	rs	rt	rd	shamt	funct
000000	10001	10010	01000	00000	100100
0_{ten}	17_{ten}	18_{ten}	8_{ten}	0_{ten}	36_{ten}

In Java:

```
short x, y, z;
x = 19;     0000 0000 0001 0011
y = 21;     0000 0000 0001 0101
z = x & y;  0000 0000 0001 0001
System.out.println(z);  17
```

a	b	c = a AND b
0	0	0
0	1	0
1	0	0
1	1	1

Logical Operations (continued):

- OR:

```
or    $t0, $s1, $s2
```

  - Performs a *bit-wise* OR operation:

```
           $s1 = 0010 0110 0110 0110 1111 1111 0000 1010
           $s2 = 1111 0000 0000 0000 1111 0000 1111 1111
    $s1 OR $s2 = 1111 0110 0110 0110 1111 1111 1111 1111
```

  - For Java and C, this operation is done with `|`
    - Example:

```
x = y | z;
```

op	rs	rt	rd	shamt	funct
000000	10001	10010	01000	00000	100101
0_{ten}	17_{ten}	18_{ten}	8_{ten}	0_{ten}	37_{ten}

a	b	c = a OR b
0	0	0
0	1	1
1	0	1
1	1	1

Logical Operations (continued):

- NOR, Not OR:

```
nor    $t0, $s1, $s2
```

  - Performs a *bit-wise* NOT OR operation.
  - The NOT OR operation takes the OR of the two operands, then reverses (NOT's) the result:

```
         $s1 = 0010 0110 0110 0110 1111 1111 0000 1010
         $s2 = 1111 0000 0000 0000 1111 0000 1111 1111
  $s1 OR $s2 = 1111 0110 0110 0110 1111 1111 1111 1111
 $s1 NOR $s2 = 0000 1001 1001 1001 0000 0000 0000 0000
```

  - There is not an operator for this operation in Java or C

op	rs	rt	rd	shamt	funct
000000	10001	10010	01000	00000	100111
0_{ten}	17_{ten}	18_{ten}	8_{ten}	0_{ten}	39_{ten}

a	b	c = a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

Logical Operations (continued):

- NOT:
  - The bit-wise NOT operation simply reverses each bit: 0 becomes 1; 1 becomes 0.

```
    $s1 = 0010 0110 0110 0110 1111 1111 0000 1010
NOT $s1 = 1101 1001 1001 1001 0000 0000 1111 0101
```

  - MIPS does not have a NOT operation.
  - Instead, we can use NOR with $zero as one of the registers. For example, to apply the NOT operation to $s1, do the following:

```
NOR  $t0, $s1, $zero
```

  - There is a NOT operator in C and Java for doing a bit-wise NOT operation:
    - The tilde operator:

```
x = ~ y;
```

a	NOT a
0	1
1	0

Logical Operations (continued):

- XOR, Exclusive-OR:

```
xor    $t0, $s1, $s2
```

  - Performs a *bit-wise* XOR operation. Exclusive-OR has the answer 0 when both operands are 1.
  - XOR

```
        $s1 = 0010 0110 0110 0110 1111 1111 0000 1010
        $s2 = 1111 0000 0000 0000 1111 0000 1111 1111
$s1 XOR $s2 = 1101 0110 0110 0110 0000 1111 1111 0101
```

  - For Java and C, this operation is done with ^
    - Example:

```
x = y ^ z;
```

op	rs	rt	rd	shamt	funct
000000	10001	10010	01000	00000	100110
0_{ten}	17_{ten}	18_{ten}	8_{ten}	0_{ten}	38_{ten}

a	b	c = a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

Logical Operations (continued):

- XOR, Exclusive-OR:

```
xor    $t0, $s1, $s2
```

  - Performs a *bit-wise* XOR operation. Exclusive-OR has the answer 0 when both operands are 1.
  - XOR

```
          $s1 = 0010 0110 0110 0110 1111 1111 0000 1010    the key
          $s2 = 1111 0000 0000 0000 1111 0000 1111 1111    the message
$s1 XOR $s2 = 1101 0110 0110 0110 0000 1111 1111 0101    the encrypted message

$s1 XOR $s2 = 1101 0110 0110 0110 0000 1111 1111 0101    the encrypted message
          $s1 = 0010 0110 0110 0110 1111 1111 0000 1010    the key
                1111 0000 0000 0000 1111 0000 1111 1111    the message
```

- `and  $t1, $s5, $zero`
- `or   $t2, $s5, $zero`

Logical Operations (continued):

- Immediate versions of AND, OR, XOR. These are named: **andi**, **ori**, **xori**.
  - They each have a 16-bit, signed, two's complement number for the immediate operand.
  - The immediate value is *zero-extended* to create a 32-bit number. Then the operation is applied.

```
andi  $t0, $s1, 0x74 A2       andi  $t0, $s1, 0x74A2    andi $t3, $s4, 17
        $s1 = 0010 0110 0110 0110 1111 1111 0000 1010
and  0x74A2 = 0000 0000 0000 0000 0111 0100 1010 0010
              0000 0000 0000 0000 0111 0100 0000 0010
```

op	rs	rt	16 bit number
001100	10001	10010	0111 0100 1010 0010
12_{ten}	17_{ten}	18_{ten}	29858_{ten}

```
ori  $t0, $s1, 0x94 A2
        $s1 = 0010 0110 0110 0110 1111 1111 0000 1010
or   0x94A2 = 0000 0000 0000 0000 1001 0100 1010 0010
              0010 0110 0110 0110 1111 1111 1010 1010
```

op	rs	rt	16 bit number
001101	10001	10010	1001 0100 1010 0010
13_{ten}	17_{ten}	18_{ten}	-27486_{ten}

Logical Operations (continued):

Shifting:

- Logical shifts:
  - Move the bits in the register.
  - Bits "fall off" the end of the register.

```
sll   $t0, $s1, 31       # $t0 = $s0 shifted left 31 bits
$s1 = 1001 1100 0100 0000 1111 0101 1100 0011
$t0 = 1000 0000 0000 0000 0000 0000 0000 0000
sll   $t0, $s1, 0        # $t0 = $s0 shifted left 4 bits
$s1 = 1001 1100 0100 0000 1111 0101 1100 0011
$t0 = 1001 1100 0100 0000 1111 0101 1100 0011
```

- Shift Left Logical: **sll**

```
sll   $t0, $s1, 4       # $t0 = $s0 shifted left 4 bits
$s1 = 1001 1100 0100 0000 1111 0101 1100 0011
$t0 = 1100 0100 0000 1111 0101 1100 0011 0000
```

  - For Java and C, this operation is done with **<<**
    - Example:

```
x = y << 4;     x <<= 5;
```

  - **rs** field is ignored (not used) since we only have two registers in the instruction.
  - **shamt** field is now being used.

op	rs	rt	rd	shamt	funct
000000	00000	10001	01000	00100	000000
0_{ten}	0_{ten}	17_{ten}	8_{ten}	4_{ten}	0_{ten}

Logical Operations (continued):

- Shift Right Logical: **`srl`**

```
srl   $t0, $s1, 4        # $t0 = $s0 shifted right 4 bits
$s1 = 1001 1100 0100 0000 1111 0101 1100 0011
$t0 = 0000 1001 1100 0100 0000 1111 0101 1100
```

  - For C, this operation is done with **>>**
    - Example: **`x = y >> 4;`**
  - For Java, this operation is done with >>>
    - Example: **`x = y >>> 4;`**

op	rs	rt	rd	shamt	funct
000000	00000	10001	01000	00100	000010
0_{ten}	0_{ten}	17_{ten}	8_{ten}	4_{ten}	2_{ten}

- Notes on using Logical shifts:
  - The **`shamt`** field has 5 bits; enough to shift from 0 to 31 positions.
  - We can use a logical shift left by two places to multiply by 4:
    - Useful in computing array offsets. Use one **`sll`** instead of using two **`add`** instructions.
  - Can think of shift left logical as a multiply by a power of two.

Logical Operations (continued):

- Shift Right Arithmetic

```
sra   $t0, $s1, 6       # $t0 = $s0 shifted right 6 bits
```

```
$s1 = 1001 1100 0100 0000 1111 0101 1100 0011
$t0 = 1111 1110 0111 0001 0000 0011 1101 0111
```

- The arithmetic shift will copy the sign bit on the left.
  - If the number is positive, 0's are shifted into the left side.
  - If the number is negative, 1's are shifted into the left side (as in the example above).
- For Java, this operation is done with **>>**
  - Example:

```
x = y >> 6;
```

op	rs	rt	rd	shamt	funct
000000	00000	10001	01000	00110	000011
0_{ten}	0_{ten}	17_{ten}	8_{ten}	6_{ten}	3_{ten}

- For C, the definition of >> is an implementation-dependent feature. That is, a compiler may choose to implement >> as a logical shift, or as an arithmetic shift.
  - Example:

```
x = y >> 6;
```

  - For an **unsigned int**, this gives shift right logical.
  - For a signed **int**, the result is implementation-dependent!

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add  $s1, $s2, $s3`	`$s1 = $s2 + $s3`	3 operands; data in reg
	add immediate	`addi $s1, $s2, 17`	`$s1 = $s2 + 17`	3 operands; data in reg
	subtract	`sub  $s1, $s2, $s3`	`$s1 = $s2 - $s3`	3 operands; data in reg
Data transfer	load word	`lw   $s1, 0($s2)`	`$s1 = Memory[$s2]`	Data fm memory to reg
	store word	`sw   $s3, 0($t5)`	`Memory[$t5] = $s3`	Data fm reg to memory
Logical	and	`and  $t0, $s1, $s2`	`$t0 = $s1 & $s2`	bit-wise AND
	and immediate	`andi $t0,$s1,0x00A3`	`$t0 = $s1 & 0x00A3`	bit-wise AND immediate
	or	`or   $t0, $s1, $s2`	`$t0 = $s1 | $s2`	bit-wise OR
	or immediate	`ori  $t0,$s1,0x00A3`	`$t0 = $s1 | 0x00A3`	bit-wise OR immediate
	nor	`nor  $t0, $s1, $s2`	`$t0 = $s1 NOR $s2`	bit-wise NOR
	xor	`xor  $t0, $s1, $s2`	`$t0 = $s1 ^ $s2`	bit-wise XOR
	xor immediate	`xori $t0,$s1,0x00A3`	`$t0 = $s1 ^ 0x00A3`	bit-wise XOR immediate
	shift left logical	`sll  $t0, $s1, 3`	`$t0 = $s1 << 3`	shift $s1 left 3 bits
	shift right logical	`srl  $t0, $s1, 3`	`$t0 = $s1 >>> 3`	shift $s1 right 3 bits, fill on left with 0’s
	shift right arithmetic	`sra  $t0, $s1, 3`	`$t0 = $s1 >> 3`	shift $s1 right 3 bits, fill on left with sign bit

Category	Instruction	Example	Meaning	Comments
Conditional branch	branch on equal	`beq  $s1,$s2,Label`	`if ($s1 == $s2) goto Label`	Equal test and branch
	branch on not equal	`bne  $s1,$s2,Label`	`if ($s1 ≠ $s2) goto Label`	Not equal test & branch
	set on less than	`slt  $s1, $s2, $s3`	`if ($s2 < $s3) $s1 = 1; else $s1 = 0`	Compare less than
Unconditional branch	jump	`j    Label`	`goto Label`	Branch to target address