Roll Call
Requirements for Test #3 Exemption

1. No unexcused absences to end of semester (written excuse required—must be MD-verified illness or work-related, with written excuse from supervisor).
2. Turn in all required homeworks for rest of year. Miss one and you take the test.
3. Make grade of 100 on semester project.
4. Maintain your average at 90 or above.
We conclude our SPIM study with a more difficult example: a recursive loop. That is, a loop that calls itself during program execution.

You won’t have many recursive loops in EE 2310, but it is worthwhile to understand how to construct one.

When you understand today’s concepts, understanding regular assembly language loops will be easy.

The following a program is simple enough to compose fairly quickly, but challenging enough to be interesting.

The program alphabetizes a string of letters in random order.
Program Specification

• As before, we will do this program together in class.
• Our program will alphabetize a group of letters in arbitrary (in this case, “q w e r t y..”) order.
• The methodology will be to start at the first letter (q) and compare it with the next letter (w). If they are in alphabetical order, it will move to the second letter and compare it to the third.
• When letters are consecutively in order (e.g., qw), it will move one letter down and do the comparison again. If they are out of order (e.g., we), the program will invoke another routine (a recursive program) that backs the letter up until it finds its correct position.
• Once the letter is correctly placed (e.g., eqw), the recursive routine finds the next letter in the string to compare and continues.
Thoughts on the Program

- Will not need a counter in our program, since we will know when we get to the end of the loop by the null character (".asciiz").
- Reversing a letter (moving “upstream”), will not require a counter either. The jal/jr twins take care of helping us finding our way back to the proper spot.
- We do not have to have make sure the letters are lower-case, as we declare the string.
- Backing up one letter position requires a separate jal.
- Once the letter is properly placed, each step back “downstream” also requires a separate jr.

* “Upstream” means in the direction of the first letter in the string; “downstream” means toward the last letter.
Correctly Positioning a Letter

Letter string: q w e r t y u i o p a s d f....

Letter string: e q r t u w y i o p a s d f....

Letter string: e i q r t u w y o p a s d f....

Letter placement

Return to get Next letter
Basic Program Flow

- **Initialize program by setting up two character pointers.**
- **Enter character compare loop:**
  - Load first two characters in string to compare.
  - If characters are in order, compare next two characters.
  - If characters should be exchanged, go to placement procedure.
  - If newest character is 0, go to output routine.
- **Character placement routine (procedure call using jal):**
  - Store $ra on stack and decrement stack pointer.*
  - Exchange places of two characters.
  - Decrement the letter pointer and test for first position.
  - Load two characters one position “upstream,” and compare.
  - If in correct order, go to “find your place” routine (jr function).
  - If not, call placement procedure again (using another jal).
- **“Find your place” routine.**
  - Go through successive jr’s until we unspool back to current letter position.@
  - Go back into character compare loop.
- **“Done” = print-out routine and syscall 10.**

* Keeps track of numbers of jal’s.
@ Must make jr’s = jal’s.
The following variables are required:
- Pointer to the current position -- Use $t0
- “Upstream compare character” -- Use $t1
- Current character being analyzed -- Use $t2
- Pointer to the first character in the string -- Use $t7
Basic Flow Chart

1. Initialize the character alphabetization routine
2. Load next two characters in the list
3. Enter character compare loop
4. Enter character placement procedure
5. Exchange character positions
6. Beginning of list?
7. No
8. Yes
9. Increment character pointer by one
10. Move back downstream to last new character position
11. Load relocated character and next upstream character
12. Second char. = 0?
13. No
14. Yes
15. Exit loop
16. Characters in sequence?
17. No
18. Yes
19. Characters in sequence?
20. No
21. Yes
22. Print out alphabetized character string
We now know enough to start writing our program.
Let’s write the title and put in any additional information we think is desirable.
The following is what I did, but you don’t have to add so much info. However, make the data declaration just like mine.
My Program “Prolog” and Data Declaration

## Lecture 18 Demo Program 1, Character List Alphabetization
## Alphabetizes a sequence of ASCII lower-case letters and prints out the resulting alphabetized list.
## Note that the list is made in the data declaration and starts with all 26 letters of the alphabet in keyboard order.

## $t0 is the variable pointer.
## $t1 holds the “upstream compare character.”
## $t2 holds the current character being analyzed.
## $t7 is the fixed pointer (always points to the first character in the string).

Program name and description
Definition of register roles

.data
string: .asciiz "qwertyuiopasdfghjklzxcvbnm"  # Characters to be alphabetized.
Further Analysis

• We now need to expand on several activities within the program to completely define its operation.

• There are two loops in the program:
  – A non-recursive loop that is a sort of “outer loop” that takes us down the list of characters, gathering a new one to compare after we have finished with the last character.
  – An inner, recursive loop that, when a character is discovered out of order, proceeds back “upstream” in the character list until it places the character in its proper alphabetical position.
  – The inner loop is made recursive to assist us in finding our way back to the proper position in the list to get the next letter to compare.
Outer Loop

- The outer or “get letter” loop loads two adjacent letters in the unalphabetized list into registers $t1$ and $t2$.
- The letter in $t1$ is one space “upstream” from the one in $t2$. The letter in $t2$ is the letter being evaluated.
- ASCII codes for the two letters are then compared.
- If $t1$ has the smaller number, (smaller ASCII code), then the numbers are in alphabetical order, and the outer loop simply increments the pointer ($t0$) by 1 and repeats itself.
- If $t2$ is smaller, that letter must be moved “upstream” to be alphabetized, so that the program enters the inner, recursive loop.
Basic Flow Chart of Outer Loop

1. Initialize the character alphabetization routine
2. Enter character placement procedure
3. Enter character compare loop
4. Load next two characters in the list
5. Second char. = 0?
6. Characters in sequence?
   a. Yes
   b. No
      - Increment character pointer by one
      - Print out alphabetized character string
5. Yes
   - Print out alphabetized character string
   - Exit loop
.text

main:    la $t0,string # Load address of string in $t0.
la $t7,string # Load address of string in $t7.

comp:   lb $t1,0($t0)  # Load first two characters to be compared.
       lb $t2,1($t0)

beqz $t2,done   # If the new character is 0, we are done.
ble $t1,$t2,count # If characters are in the correct order, get
                 #    the next character.
       jal rev  # Characters not in correct sequence;
                #    go to rearrangement routine.
       j comp  # When character is placed in correct
                #    position, get next character.

Sets up fixed and variable pointers

Outer loop; gets next char. in list and compares it to previous char.
Outer Loop (Concluded) and End Routine

```
count:    addi $t0,$t0,1   # Increment current character address.
j comp   # Compare next two characters.
done:    la $a0,string    # When finished, print out the alphabetized
         li $v0,4         # string in correct order.
syscall
li $v0,10  # Done; end program.
syscall

End of outer loop (pointer increment)
End routine which prints the alphabetized string and halts the program
```
Review of jal/jr

- We call a procedure with a `jal`, which directs program to the procedure and stores the address of the next instruction after the `jal` in `$ra`.
- At the end of the procedure, the very last instruction is `jr $ra`.
- This instruction returns the program to the next address after the `jal`.
- If the program `jals` many times (a procedure may call itself), earlier values of `$ra` will be erased.
- Thus, after a `jal`, the value of `$ra` is stored on the stack, to be retrieved in subsequent `jr`'s.
Inner Loop

- The inner loop is called as a recursive procedure (via jal).
- After storing the two letters in reverse order in the string, the pointer is decremented, and two adjacent letters are loaded again.
- Note that the letter being compared will still be in $t2.
- Another comparison is made. If $t1<$t2, the letters are now in alphabetical order. The inner loop is exited.
- If $t1>$t2, the letter in $t2 is still not properly placed. The inner loop procedure is called again via a jal.
Correctly Positioning a Letter

Letter string:  q w e r t y u i o p a s d f....

Letter placement

Letter string:  e q r t u w y i o p a s d f....

Return to get
Next letter

Letter string:  e i q r t u w y o p a s d f....
Inner Loop Details

- We use a “jal” to enter the inner loop each time.
  - Inside the loop, we save the contents of $ra (next instruction after the jal) by pushing it onto the stack, since the procedure will probably call itself more than once. Then we decrement the stack pointer (remember, it doesn’t decrement itself).
  - Each time we find the letter still out of position, we recall the loop again (jal) and move one character further upstream.
- When the letter is properly placed, we use a series of jr’s to retrace our steps to where we started.
  - Each time we execute a jr, we increment $t0 (move pointer back toward the spot of the current letter under analysis), pop the stack (putting the latest stack entry back in $ra), increment the stack pointer, and do jr $ra.
  - When jr’s = jal’s, we are back where we started the inner loop.
Inner Loop Details (2)

• Thus, for procedure entry:
  – Do jal “label” (“label” is start of inner letter placement loop).

• Immediately inside the loop:
  – sub $sp,$sp,4
  – sw $ra,0(sp)
  – Proceed with loop (do not need to store any of the s-registers)

• For the procedure exit:
  – addi $t0,$t0,1 (move pointer back toward starting position)
  – lw $ra,0($sp)
  – addi $sp,$sp,4
  – jr $ra

• We jr at exit as many times as we did a jal.
Program Flow with Additional Detail

Put first character address in $t0 and $t7

Enter outer compare loop

lb $t1, 0($t0);
lb $t2, 1($t0)

$t2 = 0?

Yes

No

$jal$ to character placement procedure

sb $t1, 1($t0);
sb $t2, 0($t0)

$t0 = $t7?

Yes

No

Sub 1 from $t0;
lb $t1, 0($t0);
lb $t2, 1($t0)

$t1 < $t2?

Yes

No

Increment $t0 by one

Do (add 1 to $t0 and $jr) as many times as there were jal’s

Exit loop

Print out alphabetized character string

No
Revision of Procedure Exit on Diagram

• The previous flow chart does not accurately convey that the jr loop must be done as many times as the recursive procedure was called.
• As our inner loop calls itself, it builds up a series of “nested loops” from which it must extricate itself.
• This jr “unwind” procedure lets the program find its way back to the current position in the string of letters.
• The “unwind” pops the stack to $ra, increments $sp, and performs a jr until there has been a jr for each jal.
• The jr loop looks like this:
Thus a more accurate flow chart would be:
Lecture #17: Final Session on SPIM; Bonus Set

Final Loop Flow Chart

Put first character address in $t0 and $t7

Enter outer compare loop

lb $t1 0($t0);
lb $t2, 1($t0)

$t2 = 0?

Yes

No

$j1 < $t2?

Yes

No

Sub 1 from $t0;
lb $t1 0($t0);
lb $t2, 1($t0)

$t0 = $t7?

Yes

No

Increment $t0 by one

addi $t0,$t0,1
lw $ra,0($sp)
addi $sp,$sp,4
jr $ra

jr’s = jal’s?

Yes

No

Exit loop

Print out alphabetized character string

jal to character placement procedure

sb $t1 1($t0);
sb $t2, 0($t0)

$t0 = $t7?

Yes

No
Writing the Inner Loop Program

- We have a good inner loop flow chart, so we can proceed to compose the recursive routine at this point.
- We start the recursive loop (or so-called “backup” routine) by storing the contents of $ra on the stack, then proceeding:
  - Store $ra contents on the stack and decrement stack pointer.
  - Reverse the positions of the two letters.
  - Decrement the pointer and load the letter we just re-positioned and the next letter “upstream.”
- Do a compare to see if those two letters are in order.
- If not, do another jal and start above again.
Inner (Positioning) Loop

rev:

- sub $sp,$sp,4  # Decrement stack pointer.
- sw $ra,($sp)  # Store contents of $ra on the stack.
- sb $t1,1($t0)  # Exchange positions of the two characters in the string.
- sb $t2,0($t0)  # Move letter back one position
- beq $t0,$t7,goback  # If position is first in the string, we are done -- get next letter.

- sub $t0,$t0,1  # Decrement the letter pointer.
- lb $t1,0($t0)  # Compare current letter in placement to the next letter "upstream."
- lb $t2,1($t0)  # Compare letter to the next "upstream" letter to see if it needs to be moved further up.
- ble $t1,$t2,goback  # If letter properly placed, return to current outer loop position

- jal rev  # Not done yet; go back "upstream" one character and do next compare.
“Rewind” Loop Using jr

```
goback:  addi $t0,$t0,1   # Placement done; move the
lw $ra,($sp)           #    pointer "downstream" until
addi $sp,$sp,4        #    we find our correct
jr $ra                 #    position to get next letter.
```

This loop repeats as many times as necessary, until the number of jr’s equals the number of jal’s. Note that once in the inner loop, since the instruction before the line labeled “goback” was the jal, each jr will return the program to “goback” until the last jr, which sends the program back to the outer loop (“comp”).

Note: each jr takes the loop back to the first memory address after the jal. Therefore, the instruction labeled “goback,” above, must be the first instruction after the jal.
We have completed the alphabetization program as specified.

On the following two pages, the program is printed, so that you can check your version against my final version.

If you have a problem, please see me.

But please, please, please, continue to work on and “debug” your program until you get the right answer! It will be a massive “learning experience.”
# Lecture 17 Demo Program 1, Character List Alphabetization
# Alphabetizes a sequence of ASCII lower-case letters and prints out the resulting
# alphabetized list.
# The list of letters is in the data declaration and contains all 26 alphabet letters in
# random order.

# $t0 -- Pointer to current spot in letters
# $t1 -- Holds the “upstream compare character”
# $t2 -- Holds the current character being analyzed
# $t7 -- Pointer to the first character in string

.text
main:   la $t0,string # Load the string address into $t0
         la $t7,string # Load the string address into $t7
comp:   lb $t1,0($t0)    # Load first two characters to be compared
         lb $t2,1($t0)
         beqz $t2,done # If the new character = 0, done
         ble $t1,$t2,count # If characters in correct order, get next character
         jal rev # Characters not in correct order; go to reverse
         j comp # Character in correct position; get next character
count:  addi $t0,$t0,1 # Increment current character address
         j comp # Return to next character compare
done:      la $a0,string    # Print out alphabetized string + CR
         li $v0,4
         syscall
         li $v0,10    # Done; end program.
         syscall
# Character reverse routine follows

rev:       sub $sp,$sp,4  # Store contents of $ra on the stack
         sw $ra,($sp) # Decrement stack pointer.
         sb $t1,1($t0) # Exchange two character positions
         sb $t2,0($t0)
         beq $t0,$t7,goback # If at first position in the string, done
         sub $t0,$t0,1 # Decrement the letter pointer.
         lb $t1,0($t0) # Compare letter to next "upstream" letter
         lb $t2,1($t0)
         ble $t1,$t2,goback # If letter is properly placed, done
jal rev     # Not done yet; move back another position

goback:    addi $t0,$t0,1  # Reverse done; move back to current position
         lw $ra,($sp)
         addi $sp,$sp,4
         jr $ra

.data
string:    .asciiz "qwertyuiopasdfghjklzxcvbnm"    # Character list

# End of Lecture 17 Demo Program 1.
Summary

• Once again, we have gone through the “thought process” of defining, flow-charting, coding, and checking out a MIPS assembly-language program (and this one was the most complicated by far).

• When you can understand the preceding program and compose similar software in SPIM, you have completed the learning process we began in Lecture 10.

• If you still have questions about this program or about how to write a recursive loop, now is the time to visit office hours, complete the homework and lab assignments, and polish your assembly language skills.
Program 2

• This problem is not a recursive loop, but it will give you more loop practice.
• The text and data declarations for this program are given.
• Write a brief loop program to compare the numerical value of the ASCII bytes that represent the characters in “Hello, world!\n” and output the largest one as a decimal number.
• This is a short program – mine took 20 instructions in all.

.text
main:

.data
str: .asciiz “Hello, world!\n”
Final Practice Programming Problem

- Copy the example program into Notepad and single-step it, watching especially, in the inner loop, how recursion is used to track the position in the list.

- Change the program so that the list of letters is input from the keyboard (using system call 8) and then alphabetized.
• What follows is some miscellaneous material on how computers start up their operation and how operating systems work in general.
• We do not have time for this material in class, but as it is a worthwhile set of information to be aware of, it is offered as a second bonus homework (as in Lecture 11).
• Turning in this homework is worth 100 points, but it does not count as a homework when the overall homework grade is calculated. It therefore adds 100 to the numerator of your homework average but nothing to the denominator.
In this bonus material on computers in general, we discuss several miscellaneous topics:

- Operating systems
- Starting up or “booting” a computer
- How a computer operating system (OS) manages the execution of user programs in a multi-tasking environment
- Multiprogramming and virtual memory
- Processes; launching a process; process switching
- Exceptions
An operating system (OS) is a program that supervises computer system functions.

The OS runs application programs (e.g., Word) and input/output (I/O) like printing.

Examples are Unix®, Linux®, Windows®, and MacOS®.

The OS relieves users from operational details:

- Controls all process operations (i.e., it is “the boss”).
- Protects each process from damage by another.
- Activates processes and supervises peripheral access (printer, CRT, disk memory, DVD-ROM, etc.).
- Prioritizes process operation and access to peripherals.
- Sets up and monitors communication with other processes, network input/output, etc.
• Since the OS handles fundamental operations, **users do not have to deal with these details**.
  – Example: printing a Word® file under Windows® -- When the user “prints” a Word file, Word converts the document to a standard data format and passes it to Windows, which activates a “print driver” (printer software) to output data to the printer.

• Programs issue **system calls** to request OS service.
• The OS call handler uses information passed in the system call to decide what action to take.
• After servicing a system call, the OS resumes, suspends, or terminates the process that called it. It can schedule other tasks while a process awaits the results of a system call (e.g., an application awaiting data from the disk).
A modern operating system provides other important services:

- Allocates memory segments to programs (processes) that are active.
- Supervises starting, running and termination of all programs.
- Determines priority of all operations.
- Provides interprocess communication and access to I/O devices (CRT screen, mouse, keyboard, printer, disk, etc.) as noted previously.
- Detects and handles events that interrupt program execution (interrupts, exceptions, protection faults – covered in detail below).
- On a large system, can provide accounting functions (usage or time running for a customer when computer time is “bought”).
Starting or “Booting” a Computer

- At computer power-up (think PC), there is a short (~1-3 sec) “power settle” delay, then an all-system reset.
- OS load is then initiated.
  - Long ago (circa 1950-1970), a “bootstrap” load might be put into memory using switches on the computer front panel.
  - Later, it was input from paper tape, magnetic tape, or disk.
- Today, an OS load usually starts with a small (“micro”) OS, which may be in read-only memory (ROM).
- The micro OS (PC “Bios” or “CMOS”) checks system hardware, then initiates an OS loader.
  - For Windows™, this is a program called “command.com,” which is on the lowest sector of the hard drive.
  - Once in operation, the loader loads the OS and starts it.
How the OS Runs Programs

• Single-task operating systems were the first OS’s (1950’s IBM systems; 1960 mini-computers, 1970 PC-DOS).
• These primitive OS’s ran one program at a time.*
  – User programs issued commands directly to the hardware (!).
  – Users had “free reign” within the computer, and could do just about anything (“FORMAT C:” for example).
  – User programs could overwrite and destroy the OS.
• Today, a multi-tasking OS is the norm (Large IBM systems, MacOS, Windows).
  – Many programs reside in main memory at the same time.
  – The OS protects itself and user program vigorously.
  – Programs obtain services only via system calls.

* Anybody remember printing a document in DOS?
Processes

• A program is an executable file (e.g., a SPIM NotePad file). A process is an instance of a program in execution (e.g., that NotePad file assembled and running).

• In a modern OS, processes execute “simultaneously.”

• “Simultaneously” is in quotes since only one process can be executed by a single CPU at a time.
  – Although processes execute serially, many can be in memory at the same time. Since in a given second, the OS runs parts of many processes, we say that the execution is “simultaneous,” since they are simultaneous in our perception.
  – The OS runs a process until it no longer can run, due to:
    • Process completion.
    • Process suspension (awaiting disk load or print completion).
    • Process abort (stopped by the OS due to a problem).
Launching a Process

- To launch a user process or application, the OS:
  - Reserves space in operating memory for the process.
  - Copies the executable text module into the address space.
  - Pushes “command-line and environment arguments” onto the stack (process identification and priority or scheduling information, relationship to other processes, process communication requirements) in a “frame” assigned to it.
  - Allocates process space for data and text (program) segments (as in PCSPIM), and expansion room for storage of data generated within the process.
  - Initiates the process (e.g., calls “main” in SPIM).
- A process is protected within system memory from other “predatory” processes.
• When a process terminates or is suspended (perhaps awaiting disk I/O), the OS makes a process switch.

• In making the switch, the OS will:
  – Push the contents of registers and other information about the curtailed process onto the stack
  – Use a “scheduling algorithm” (process priority according to relative importance) to find the highest pending process.
  – Retrieve (pop) the context of this process (register contents, etc.) from its stack frame.
  – Restore the context (registers, stack pointer, mode) to the CPU.
  – Re-initiate the process.
Exceptions

- An **exception** is an event that causes the OS to halt or suspend normal operation.
- Many exceptions are benign – most are **interrupts**.
  - Interrupts are often caused by **external events**, such as a signal from a peripheral that it needs to communicate with the CPU.
  - A good example is the disk drive on a computer. It is a mechanical device that must be serviced immediately.
  - The current process will be halted for interrupt service.
- Some exceptions may be system problems:
  - Bus (data I/O) error, Syscall timeout, bad memory address.
  - Protection exception (memory violation).
  - CPU error.
Exceptions (2)

- In the R2000, exceptions are handled by Coprocessor 0.
- For most exceptions:
  - The processor exits user mode and enters executive mode.
  - A subroutine is initiated (usually via memory location 0x80000080 in OS memory).
  - This subroutine, called an exception handler, takes control, determines the exception, processes it (possibly by executing I/O), and resumes normal operation.
- For more serious exceptions, such as application errors, the operating system usually deletes the offending process (with proper information to the user).
1. What is the difference between an exception and an interrupt?

2. What is the difference between a process and a program?

3. What is the real definition of “simultaneous processing” in a multi-tasking system?

4. In a multi-process system, what happens when the current process suspends?

5. How does an application program request service from the OS?

6. To facilitate process switching, where is the process context kept?

7. When a computer is turned on, what is the first software that activates?

8. What is a one-sentence description of an operating system?