MIPS Procedure Calls - Review

• When making a procedure or function call, it is necessary to:
  – Place parameters you wish to pass where they can be accessed by the procedure;
  – Transfer control to procedure;
  – Perform the desired task;
  – Place the result value(s) where the calling program can access them;
  – Return control to the point of origin.
• The procedure call should not clobber any registers other than those agreed upon by protocol.
• The MIPS architecture
  – Provides instructions to assist in procedure calls (jal) and returns (jr);
  – Uses software conventions to
    • Place procedure input and output values.
    • Control which registers are saved/restored by caller and callee.
  – Uses a software stack to save/restore values.
Stack

• A stack is a way of organizing data in memory. Data items are visualized as behaving like a stack of physical items. Often a stack is visualized as behaving like a stack of dinner plates. Data are added and removed from the stack data structure in a way analogous to placing plates on the stack of plates and removing them.

• Normally, with a stack of dinner plates, all operations take place at the top of the stack. If you need a plate, you take the one at the top of the stack. If you add a plate, you place it on top of the stack.

• Which plate is the last one added to the stack? Which plate will be the first one removed?

Upside Down MIPS Stack

• Stack-like behavior is sometimes called "LIFO" for Last In First Out.

• The data elements in our stacks will be 32-bit words. Stacks can be used for all types of data, but in this class, stacks contain only 32-bit MIPS full words.

• The picture shows a stack of MIPS full words. The stack pointer register $sp by convention points at the top item of the stack. The stack pointer is register $29 by software convention. The mnemonic register name $sp is used by the MIPS assembler.

• In the usual way of drawing memory, the stack is upside down. In the picture, the top item of the stack is 81. The bottom of the stack contains the integer -92.

• Before the operating system starts your program it ensures that there is a range of memory for a stack and puts a suitable address into $sp.
Push

- By software convention, $sp always points to the top of the stack. Also by convention, the stack grows downward (in terms of memory addresses). So, for our stack of 4-byte (full word) data, adding an item means subtracting 4 from $sp and storing the item at that address. This operation is called a push operation.
- To push an item onto the stack, first subtract 4 from the stack pointer, then store the item at the address in the stack pointer. Here is what that looks like in code. Say that the value to push on the stack is in register $t0.

```
# PUSH the item in $t0:
sub $sp,$sp,4  # point to the place for the new item
sw $t0, 0($sp)  # store the contents of $t0 as the new top.
```

Pop

- Removing an item from a stack is called a pop operation. In the real-world analogy an item is actually removed: a dish is physically moved from the stack. In a software stack, "removal" of an item means it is copied to another location and the stack pointer is adjusted.
- The picture shows a pop operation. The data is first copied from the top of stack to the new location and then the stack pointer is increased by four.
- To pop the top item from a stack, copy the item pointed at by the stack pointer, then add 4 to the stack pointer. Here is what that looks like in code. Say that we want the value to be popped into $t0.

```
# POP the item into $t0.
lw $t0, 0($sp)  # Copy the top item to $t0.
addu $sp,$sp,4  # Point to the item beneath the old top.
```
Example

- The stack is often used to hold temporary values when there are not enough registers. This is common in code that a compiler outputs for evaluating arithmetic expressions. Here is an example of this. The program evaluates the expression ab - 12a + 18b - 7. Pretend that only $t0 and $t1 are available, and that the stack pointer $sp has been initialized properly by an operating system. Terms of the expression are pushed onto the stack as they are evaluated. Then the sum is initialized to -7 and the terms on the stack are popped and added to the sum.

Example - Continued

# Evaluate the expression ab - 12a + 18b - 7
.text
.globl main
lw $t0,a # get a
lw $t1,b # get b
mult $t0,$t1 # a*b
mflo $t0
sub $sp,$sp,4 # push a*b onto stack
sw $t0,($sp)
lw $t0,a # get a
li $t1,-12 # -12a
mult $t0,$t1
mflo $t0
sub $sp,$sp,4 # push -12a onto stack
lw $t0,$sp # pop 18b
add $sp,$sp,4
add $t1,$t1,$t0 # 18b -7
………
Run-time Stack

- There is a finite amount of memory, even in the best computer systems. So it is possible to push more words than there are words of memory. Usually this would be the result of an infinite loop because when a program is first entered the operating system gives it space for a very large stack.
- The picture shows how a typical operating system arranges memory when a program starts. There are four gigabytes of (virtual) memory available in total. The section of memory from 0x10000000 to 0x7FFFFFFF is available for the data segment and the stack segment. This is 1.8 Gigabytes of space.
- When the program is started the stack pointer ($sp$) is initialized to 0x7FFFFFFF. As the program runs, the stack grows downward into the available space. The data segment grows upward as the program runs. Of course, in a dynamic program, the segments grow and shrink. If the combined size of the segments exceeds the available space, their boundaries will meet somewhere in the middle of the range. When this happens there is no memory left.

Reversing a String Example

- "Text" is traditionally what a segment of machine instructions is called. It becomes a "process" when it starts executing. (This is analogous to the phrase "text of a play" and "performance of a play").
- Here is an example program: the user enters a string. The program reverses the string and writes it out. To understand how the program works inspect the following diagram. The string "Hello" is pushed onto the stack, character by character. Then the characters are popped from the stack back into the original string buffer. This reverses the order of the characters.
Stack-based Calling Convention

• To return to the caller a subroutine must have the correct address in $ra when the jr instruction is performed. But this address does not have to remain in $ra for all the time the subroutine is running. It works fine to save the value of $ra and then restore it when needed.

• The picture shows main calling subA (step 1), which calls subB (step 2), which calls subC (step 3). Just before each call the value in $ra is pushed onto the stack. Just before each return, that value is popped back into $ra. Then the return is made with jr $ra.

• For this to work, each subroutine must restore the stack to the same state it was in just as the subroutine started.

Pushing the Return Address

• When main starts execution it has been passed control by the operating system. As far as the OS is concerned, main is a subroutine. So it has a return address (in $ra) to use when it finishes execution (and returns control to the OS).

• As another example, consider subB. When it starts execution the top of the stack is subA's return address ($ra for returning to main). When subB calls subC, subB pushes its return address on the stack ($ra for returning to subA). Later on it pops this address when it returns to subA.

• The push and pop operations are done in the usual way. There are no explicit push and pop operations as the diagram might imply.
Register Problem

- In the Simple Linkage convention, registers $s0$—$s7$ must not be altered by a subroutine. This restriction creates a problem if subroutines are to call other subroutines.
- Say that main calls subA and that subA calls subB. subA can't save any values in $s0$—$s7$ (because it is not allowed to alter them). But any values it saves in $t0$—$t9$ might be clobbered by subB (because subB is allowed to alter them). In effect, subA can't use any registers! Not good.
- The solution is to allow subA to use $s0$—$s7$. However, subA must restore these registers to their initial state (to what was in them on entry) when it returns to its caller.

Pushing and Popping Registers

- Here is a rule: if a subroutine alters any of the "S" registers, it must first push their values onto the stack. Just before returning to the caller it must pop these values from the stack back into the registers they came from. Of course, this must be done in the correct order.
- The registers are popped in the **opposite order** that they are pushed.

```
subC:
    sub $sp,$sp,4    # push $ra
    sw $ra,($sp)
    . . .
    jal subD        # call subD
    . . .
    lw $ra,($sp)    # pop return address
    add $sp,$sp,4   # return to caller
    jr $ra

# subD expects to use $s0 and $s1
subD:
    sub $sp,$sp,4   # push $s0
    sw $s0,($sp)
    sub $sp,$sp,4   # push $s1
    sw $s1,($sp)
    . . .
    # statements using $s0 and $s1
    lw $s1,($sp)    # pop s1
    add $sp,$sp,4   # pop s0
    lw $s0,($sp)
    add $sp,$sp,4   # return to subC
    jr $ra
```
Stack-based Calling Convention

- The Simple Linkage Convention can be extended into a Stack-based Calling Convention. This is not an official convention. However it is what you and a group of programmers might agree to use because it is not very complicated and does nearly everything you need. If you want to link assembly language routines to "C" or "C++" you need to use the full, official, linkage rules. Here are the simpler rules:

  • Calling a Subroutine (done by the caller):
    1. The caller pushes onto the stack any registers $t0-$t9 that contain values that must be saved. The subroutine may change these registers.
    2. The caller puts argument values into $a0-$a3.
    3. Call the subroutine using jal.

  • Subroutine Prolog (at the start of execution of the subroutine):
    1. If this subroutine calls other subroutines, push $ra onto the stack.
    2. Push onto the stack any registers $s0-$s7 that this subroutine might alter.

  • Subroutine Body:
    1. The subroutine may alter any "T" or "A" register, or any "S" register that it saved in the prolog.
    2. If the subroutine calls another subroutine, then it does so by following these rules.

  • Subroutine Epilog (at the end of execution of the subroutine):
    1. Return values are put in $v0-$v1
    2. Pop from the stack (in reverse order) any registers $s0-$s7 that were pushed in the prolog.
    3. If it was pushed in the prolog, pop $ra from the stack.
    4. Return to the caller using jr $ra.

  • Return from a Subroutine (done by the caller):
    1. The caller pops from the stack (in reverse order) any registers $t0-$t9 that it previously pushed.
Stack-based Calling Convention

- Those rules are somewhat complicated. Here is a picture. It shows the four sections of subroutine linkage. The basic tasks of each section are:

- **Subroutine Call:** Push all "T" registers that contain values that will be needed after the call. Put arguments in "A" registers. jal to the subroutine.

- **Prolog:** If this subroutine calls other subroutines, push $ra. Push all "S" registers that the subroutine alters.

- **Body:** Normal code, except that it must follow these conventions. "T" and "A" registers can be used freely, as can any "S" registers that were saved in the prolog.

- **Epilog:** Put return values in "V" registers. Pop any "S" registers. Pop $ra if it was pushed in the prolog. jr $ra back to the caller.

- **Return:** Pop any "T" registers that were previously pushed.

Programming Language History

- Well... this is not easy stuff. As proof of that statement, look at computer history. It took several decades before modern high level languages were established. The first one of any importance was Algol, created about 1960. Algol established stack-based subroutine linking. But Algol never quite caught on.

- Pascal (created about 1974) was a milestone. It became highly popular and used this stack-based idea of subroutines (which it called procedures and functions). Programming languages can be classified as "Before Pascal" and "After Pascal."
Summary

- Introduced the concept of Stacks;
- Illustrated with the use of push and pop concepts;
- Discussed a stack-based calling convention.