CPS311 Lecture: Procedures Last revised 9/9/13

Objectives:

1. To introduce general issues that any architecture must address in terms of calling/returning from procedures, passing parameters (including the distinction between call by value and call by reference), getting return values, and preserving registers.
2. To introduce the notion of a procedure's stack frame
3. To discuss MIPS mechanisms for call/return, passing parameters, getting return values, and preserving registers.

Materials:

1. Example program
2. MIPS register conventions handout

I. Introduction

A. Any program of significant size is typically implemented as a collection of smaller units called by the name PROCEDURE or FUNCTION or METHOD. The idea, in any case, is to define a body of code that can be used from various other places in the program.

(Note: in keeping with accepted practice, we will use the term procedure for such a body of code - even though different HLL's prefer different terms - e.g. the C language documentation typically uses the term function, while OO languages typically use the term method.)

B. Consider the following fragment from a block of C++ code:

```
class Person
{
    private:

        ...
        int age;
        ...

    public:

        ...
        int getAge()
        {
            return age;
        }
};

int main()
{
    ...
    Person p;
    ...
    cout << p.getAge();
    ...
}
```

In particular, consider the machine language translation of the definition of the getAge() method, and its subsequent call later in the program. In order for this call to be successful, a number of issues must be addressed:

1. Saving a return point:

   The code in the main program will include a line that jumps to the code for getAge(), followed immediately by code to output the resulting integer value to cout - e.g.

   code to invoke getAge();
   code to output resulting integer value to cout

   This code must somehow let the code for getAge() know where execution is to resume once getAge() has done its job - i.e. at the code needed to output the value

   a. If a procedure were only used once in a program, this could be handled by having the _procedure_ "know" where it is to return.

   b. But since most procedures are used multiple times in a program, the caller must assume the burden of telling the procedure where to return to when it is done.

2. Return value:

   The value looked up by getAge() must somehow be made available to the output code in the main program.

3. Passing parameters:

   Even though the call to getAge() does not have any explicit parameters, it does have an implied parameter - the "this" parameter that points to the specific Person object whose age is being requested.

C. All architectures provide support for these tasks, and at least one other that we haven't considered yet. This support may take the form of special provisions in the hardware, or may take the form of commonly adhered-to software conventions.

   We will look at the MIPS mechanisms, but our goal is not simply to understand how MIPS does these things, but also to understand the task itself. Where appropriate, I will also mention alternative approaches that other architectures use.

II. Stack Frames

A. Before we discuss these issues in detail, we need to consider a concept that virtually all ISAs make some provision for - the stack frame for a procedure.

B. A common software convention is to set aside a special region in memory called the STACK.

   1. As you recall, a stack is a data structure that grows and shrinks like a stack of cafeteria trays. Stacks obey a LIFO (last-in first-out) discipline.

   2. Typically, when a procedure is called a special data structure called its stack frame (or call frame) is allocated space on the stack. When it returns, this space is releaed.

Note that procedure calls and returns obey a LIFO discipline - e.g. if A calls B and B calls C and C calls D, then D returns first, then C, then B.

3. An important characteristic of the stack in most ISA's is that it GROWS TOWARD LOW ADDRESSES.

   a. That is, when the stack is initially created, it starts at the high end of memory.

   b. As it grows, it grows toward low memory.

   c. The reason for this is that there are two ways that the memory allocated to a program can increase as the program is running:

      i. Frames can be pushed on the stack - if procedure A calls B and B calls C, then frames for A, B, and C will be on the stack while C is executing.

      ii. Memory can be allocated by new(). The region from which this memory is allocated is called the HEAP (nothing to do with the heap data structure - the same term but different meaning)

      iii. To avoid these two mechanisms coming into conflict, each starts from one end of memory and grows toward the middle:

```
Low memory addresses
--------------------
| PROGRAM CODE AND |
| STATIC DATA      |
--------------------
| HEAP (used for   |  | Direction of growth
| new() requests)  |  v
////////////////////
////////////////////  As-yet unallocated memory
////////////////////
| STACK            |  ^
|                  |  | Direction of growth
--------------------
High memory addresses
```

4. Most ISA's designate one or two registers to hold the address of the stack frame in memory.

   a. A STACK POINTER register to point to the current top of the stack.

   b. A FRAME POINTER register (not always used) to point to the base of the current stack frame. (This allows other items to be pushed on the stack after the frame is pushed, which will change the stack pointer but not the frame pointer.

   c. This leads to the following convention when a procedure is executing

```
                                    Low memory addresses
        ---------------------
        | Stack frame for   | $sp - points to "top" of stack
        | current procedure |
        | (saved registers  |
        | and other values) | $fp - points to base of frame
        ---------------------
                                    High memory addresses
```

d. The MIPS convention is to use register $29 as the stack pointer (commonly referred to by the special name $sp) and to use register $30 as the frame pointer (commonly referred to by the special name $fp).

   i. Note that, in the MIPS ISA, these are software conventions. Actually, any general registers could be used for these purposes, along as all code agreed on the conventions. (Since we will be making use of the operating system and system libraries, we will need to adhere to these conventions!)

   ii. In contrast, some ISA's use special registers for the stack pointer and/or frame pointer, and these conventions are "wired in" to the hardware.

C. The stack frame for a procedure may store some or all of the following:

1. The return address in the caller to which the procedure is to return.

2. The procedure's parameters.

3. The values of other registers that the procedure uses, that need to be preserved if they are used elsewhere.

4. Local variables:

   a. For efficiency reasons, local variables of a procedure are kept in registers where possible. However, there are a number of cases where a local variable will be allocated space in the procedure's stack frame instead:

      i. The local is too big to fit in a register (e.g. it's an object or an array.)

      ii. The number of locals exceeds the number of available registers.

      iii. The local will be passed as a by-reference parameter to some other procedure (meaning it must have a memory address to pass.)

   b. When a local variable "lives" in the procedure's stack frame, its address can always calculated as some offset relative to the base address of the stack frame. Given that stack growth is toward low memory, this offset is generally negative.

      Example: Suppose we have a procedure which declares a local variable foo that is allocated space in its stack frame. Suppose, further, that other elements of the frame occupy a total of 16 bytes ahead of it. Then while the procedure is executing the frame may look like this

```
                                                Low memory addresses
              ---------------------
              | ................. |
              | foo               |
      16 bytes | ................. | $fp
              ---------------------
                                                High memory addresses
```

The address of foo will be $fp-16, or - in mips notation - -16($fp)

III. Mechanisms for Supporting Procedure Calls

A. Basic Procedure Calling Mechanisms

1. All ISAs support some sort of procedure call instruction, which does two things:

   a. Save the return address - the address of the next instruction after it - in some ISA-defined location. Often, this is on the stack (i.e. the return address is pushed on the stack)

   b. Begin executing the first instruction of the procedure.

2. In the MIPS architecture, the basic instruction for invoking a procedure is the JUMP AND LINK (jal) instruction.

   a. It looks very much like the ordinary jump instruction; in fact, it uses the same J-Format:

```
# of bits      6       26

field name     op      target address

contents       op = 3
               for jal
```

   b. The fundamental distinction between this instruction and the ordinary jump instruction (j) is that the CPU saves the address of the next instruction to be executed in register 31.

      i. That is, in the getAge() example we used earlier, the following code will occur in main:

```
jal start of getAge()
code to output resulting integer value to cout
```

      ii. When getAge() is being executed register 31 will contain the address of the code to output the integer value to cout. To return to this code upon completion, getAge() will end with the following instruction:

```
jr     $31
```

      which - as you recall - does a jump to the instruction whose address is contained in register 31. (This is, in fact, the chief use of the jr instruction.)

c. Note that the use of register 31 for this purpose is a HARDWARE provision - the jal instruction always results in the address of the next instruction being placed in register 31. (The designers of MIPS made this choice to allow the maximum possible number of bits in the instruction format to be available for specifying the target address. If they had decided to let any register be used, not just register 31, they would have needed to set aside 5 bits to specify a register number.)

d. What MIPS does in this regard is actually a bit unusual - most ISAs save the return address on the stack. MIPS uses the convention it does for hardware simplicity, and relies on the called procedure to actually push the return address on the stack if this needs to be done (which isn't always the case).

2. Most architectures specify that procedures that return a value to the caller place the value in a specific register. In the case of MIPS, the convention is to use register 2 for this purpose - if the return value is 32 bits or less - and to use registers 2 and 3 together for a 64 bit return value.

   a. Thus, in the earlier example, getAge() will contain code like this:

```
put age field of the person object in $2
jr $31
```

   b. And the code in main will look like this

```
jal start of code for getAge()
output the value in $2 to cout
```

   c. Note that this is a SOFTWARE convention. Any register(s) could be used to return a value from procedure - but the common MIPS convention is to use $2 (and $3 if needed) for this purpose.

      (Each ISA typically uses a convention that is appropriate for the underlying register set).

3. Parameter-passing actually involves two basic issues: WHAT is passed, and WHERE it is passed.

   a. In terms of WHAT, there are two basic answers that all ISAs support PASS BY VALUE and PASS BY REFERENCE.

      i. In pass by value, what is passed is the ACTUAL VALUE of the argument.

      ii. In pass by reference, what is passed is the ADDRESS IN MEMORY where the argument resides.

      Example: Given the following declaration:

```
void foo(int a, int & b)
...
```

      And call

```
int x;
foo(3, x);
```

And assuming that x is stored in memory location 1000, what is passed (on any ISA) is 3 and 1000, since a is passed by value, and b is passed by reference.

b. In terms of WHERE parameters are passed, ISAs vary quite widely.

   i. Many ISAs pass parameters using the stack - i.e. the caller pushes the parameters on the stack before calling the procedure.

   ii. In the case of MIPS, a common convention is to use registers beginning with 4 for passing parameters. Some MIPS-based systems reserve 4 registers for this purpose (4 .. 7); others reserve 8 (4..11). We will discuss below what happens if some procedure requires more parameters than this - but this is relatively rare - procedures have relatively few parameters.)

      (a) Thus, in the getAge() example, the code in main will look like this:

```
put p (address of object p refers to to) in $4
jal start of code for getAge()
output the value in $2 to cout
```

      (b) The code in getAge() will look like this

```
lw $2, age field of person object pointed to by $4
```

      (c) As was true with the return address and return value, this is a SOFTWARE CONVENTION. Any registers could have been used for passing parameters - or some other strategy could] have been used, like pushing them on the stack. But since MIPS system software uses $4, $5, $6 and $7 for this purpose, the convention is to use these for user software as well (which often calls library routines anyway)

   iii. What does MIPS do if a procedure needs more parameters than there are registers reserved for this purpose (4 or 8)? (We've already noted that this will be rare.)

      In brief, the answer is that the extra parameters are pushed on the stack by the caller.

   iv. One more MIPS example: Given the following declaration:

```
void foo(int a, int & b)
...
```

      And call

```
int x;
foo(3, x);
```

      And assuming that x is stored in memory location 1000, then the MIPS parameter registers would be as follows:

```
$4: 3                  <- value of a - passed by value
$5: 1000               <- address of b - passed by reference
```

B. The Issue of Register Preservation

1. One issue we haven't faced in our examples yet is that of PRESERVING REGISTER VALUES ACROSS CALLS. But this is important in many cases.

2. Procedures are of two general types:

   a. "Leaf" procedures do not, themselves, call other procedures. (In our example, getAge() is a leaf procedure.)

   b. "Non-leaf" procedures do call other procedures. (In our example, getAge() might need to be a non-leaf procedure if instead of storing the person's age in the object, we stored the person's date of birth. Then getAge() would have to COMPUTE the age by getting the current date, and then calling a method of class Date that gets the difference in years between two Dates.)

3. A non-leaf procedure has to deal with the saving of information in registers that might be changed when calling another procedure.

   Example: If we stored the person's date of birth instead of current age, then getAge() would look something like this - assuming the field in the object that holds the person's birth date is called dob, and that there is a Date class with suitable methods:

```
int getAge()
{
    Date today = Date.getToday();
    return today.yearsSince(dob);
}
```

   This could translate into assembly language code like the following:

```
#       Note: upon entry "this" is in $4
#       and return address is in $31
jal     code for getToday() method of Date
put returned value in $4
lw      $5, dob field offset($4)
jal     code for yearsSince() method of date
# return value from yearsSince() method is already in $2
jr      $31     # Return to original caller
```

   Unfortunately, though, this won't do what we want it to. Why?

   ASK

   a. The lw $5 that sets up the second parameter to yearsSince() assumes that $4 holds the "this" pointer for the method. That was correct upon entry, but the previous instruction put a different value in $4. Even if this were not so, the getToday() method code might have changed $4 somehow.

b. The final jr $31 assumes that $31 contains the address to return to in the caller. But the two intervening jal's will have changed this.

   This illustrates the necessity of non-leaf procedures preserving registers that it needs.

4. In general, there are two approaches one might take to addressing this issue:

   a. The "caller save" approach: when a non-leaf procedure calls another procedure, it must assume that the procedure it calls may change any of the registers. Therefore, the caller must save the values of any registers it needs before it calls the other procedure, and restore the values after the called procedure returns.

      Example: in the above, the getAge() method needs to make use of values that are in $4 and $31 when it is called. Therefore, it would need to save these two values in a safe place before calling getToday(). It would need to restore $4 before loading the dob argument for the second call, and would need to restore $31 just before it returns to its caller.

      Under this convention, a caller can make NO assumptions about what values are in the registers when a call to another procedure finishes. This might be called the "defensive driving" approach to register preservation.

   b. The "callee save" approach: a procedure that alters a value stored in any register must save the value that was in that register when it is entered, and restore it just before it returns.

      Example: In the above, getAge() would be responsible for saving and restoring the values in $4, $5, and $31.

      Under this convention, a caller can assume that a called procedure does not make any visible changes to any registers - except, of course, the register holding a return value if there is one. This might be called the "clean up your own mess" approach to register preservation.

   c. Note that the choice of approach to take is really a matter of software convention, not hardware design (though some ISA's include mechanisms to facilitate one or the other of the two approaches - e.g. the DEC VAX has a mechanism that allows a procedure to specify that certain registers be saved by any CALL instruction that calls it and restored by the RET instruction that it uses to return - i.e. the hardware implements a callee save facility.)

5. Software conventions on MIPS use a hybrid approach - some registers are designated as callee save registers, while the rest are caller save.

   HANDOUT - MIPS conventions

a. Callee save - a procedure that modifies any of these registers MUST save them and restore them:

   i. Registers $16 .. $23 - known by the special names s0 .. s7 (where s stands for "saved")

   ii. Registers $28 .. $$30 - known by the special names $gp, $sp, $fp ($sp and $fp are discussed below).

   iii. Register $31 - known by the special name ra (return address - as already discussed)

      - A leaf procedure does not need to save its return address, since it doesn't call other procedures. Thus, the MIPS convention of putting the return address in a register rather than pushing it on the stack avoids a stack push in this case.

      - A non-leaf procedure must always save its return address, since $31 is used for the procedures it calls.

b. Caller save - a procedure that needs any of these registers after a call to another procedure must save them before the call and restore them afterward:

   i. Registers $2 .. $3 - known by the special names v0 .. v1 (where v stands for "value" - these are used for return values of procedures - of course, when calling a function, it is often desired to get a return value in these registers anyway, so saving/restoring them is rarely desirable.)

   ii. Registers $4 .. $7 - known by the special names a0 .. a3 (where a stands for "argument")

   iii. Registers $8 .. $15, $24 .. $25 - known by the special names t0 .. t9 (where t stands for "temporary")

c. The remaining registers are special in some way, and are not ordinarily altered by user-written procedures

   i. Register $0 - known by the special name zero - always contains zero (hardwired)

   ii. Register $1 - known by the special name at (where at stands for assembler temporary) reserved for use by the assembler when constructing 32-bit immediate values or addresses that require two instructions to create

   iii. Registers $26 .. $27 - known by the special names k0, k1 (where k stands for kernel) reserved for use by the kernel of operating system - may change at any time outside the control of the user program.

6. One issue we have not yet discussed is the question of WHERE we save registers.

a. At first glance, it might seem that setting aside a special area in memory for each procedure would suffice - but this will not always work. Why? ASK

   Recursive procedures

b. The convention used by most ISA's including MIPS - is to make use of the procedure's stack frame.

   i. To save a register, it can be pushed on the stack.

   ii. To restore a register, it can be popped from the stack. Note that, due to the LIFO property of stacks, restoring pops would have to be done in the reverse order of saving pushes.

   iii. On some ISA's - including MIPS - it is more efficient to push a whole stack frame, and then store the register values into the various slots, rather than to push each register individually. We'll see how this works out in the handout.

C. Go over handout example program

IV. More About Memory Allocation

A. At this point, it may be worthwhile to pull together various comments we've made about how MIPS systems (and, indeed, most systems) handle the allocation of physical memory for various purposes.

B. Recall that most systems divide memory into a user portion ("user space") and a system portion ("kernel space"). We are only concerned with user space here. How kernel space is divided up is a bit more system-dependent.

C. Typically, user space is divided into four regions - often with large unallocated spaces between them. The following is a typical structure:

```
--------------------------- Lowest address in user space
|                          |
| CODE                     |
|                          |
---------------------------
|                          |
| GLOBAL DATA              |
|                          |
---------------------------
|                          |
| HEAP  | (growth)         |
|       v                  |
---------------------------
| ////                     |
| //// Unused              |
| ////                     |
---------------------------
|        ^                 |
|  STACK | (growth)        |
|                          |
--------------------------- Highest address in user space
```

1. The code region contains the machine-language version of the program. This is typically loaded at program startup, and is not changed during program execution. (Often, this region of memory is mapped "read-only" during program execution.

2. The global data region contains variables and arrays that exist throughout the execution of the program. This includes:

   a. Global variables declared outside of any function or class in a language like C/C++.

   b. Static variables in the various classes for languages like C++.

   Note that, while the values of these variables can certainly change during program execution, the fact that a particular variable is found at a particular address does not change during program execution.

3. The heap region contains variables that are created dynamically while the program runs - e.g. objects created by new in languages like C++, or by routines like malloc() in C.

   a. Because new variables can be created in this region in any time, it can grow during program execution. This is managed by the sytem maintaining a "fence" value that represents the largest valid address in the heap - any address <= this is part of the heap; > this is unused. By increasing the value of the fence, the heap can be expanded.

   b. Of course, variables that are created in the heap can also be destroyed - e.g. by delete in C++ or free() in C. Typically, what happens in this case is the system uses the memory that they formerly occupied to satisfy a new request - i.e. the heap usually does not shrink during program execution, but space can be recycled.

4. The stack is used for allocating temporary memory for procedures. Whenever a procedure is called, a certain amount of memory (called its "stack frame" or "activation record" is allocated on the stack). When the procedure terminates, this memory is released for reallocation. Since procedures obey a "last in first out" discipline for calls and returns, this space grows and shrinks constantly during program execution.

D. Example: consider the following class

```
class SomeClass
{
       int someInstanceVariable;
       static int someStaticVariable;

       void someMethod()
       {
           int someLocalVariable;
           ...
       }
};
```

```
main()
{
  SomeClass c = new SomeClass();
  c.someMethod();
  ...
}
```

When someMethod is being executed, the various variables are allocated space as follows:

```
--------------------------- Lowest address in user space
|                          |
| CODE                     |
|                          |
---------------------------
| someStaticVariable       |
| GLOBAL DATA              |
|                          |
---------------------------
| c.someInstanceVariable   |
| HEAP  | (growth)         |
|       v                  |
---------------------------
| ////                     |
| //// Unused              |
| ////                     |
---------------------------
|        ^                 |
|  STACK | (growth)        |
|  someLoclaVariable       |
---------------------------  Highest address in user space
```