CS 237 Meeting 16 — 10/17/12

Announcements

1. Midterm: Oct 26th. In class open book/notes.

Robust Function Call Conventions for MIPS

- The heap sort code I gave you for lab 4 included an instructive mistake.
  - The heapify function invoked the swap function.
  - If you implemented both functions using the simple approach I have presented, the invocation of swap would overwrite the contents of $ra making it impossible for heapify to return.
- In this case, it is relatively easy to get around the problem.
  - The call to swap requires setting $a0, $a1, and $ra, so if any valuable information is in these registers, the caller must save that information before the call and restore it afterwards.
  - In the case of heapify calling swap, the needed values can be saved in other registers.

```
#  swap( &values[rootIndex], &values[smallestChild] );
# SAVE REGISTERS CHANGED BY CALL
    move  $t2,$a0
    move $t3,$a1
    move $t4,$ra

    sll $a0,$t3,2
    add $a0,$a0,$t2
    sll $a1,$t0,2
    add $a1,$a1,$t2
    jal swap

# RESTORE REGISTERS CHANGED BY CALL
    move $a0,$t2
    move $a1,$t3
    move $ra,$t4
```

- This approach would clearly get difficult if there were just one or two levels of calls. If swap called another function or heapify was called by another function, it would be quite difficult to keep track of which registers each function was using and/or there would simply not be enough registers.
- Worse yet, if a recursive function is involved, the number of values that might need to be saved is unbounded.
  - While my code defined heapify using a loop, it is arguably clearer to define the function recursively:

```
void heapify( int values[], int rootIndex, int heapSize ) {

  int smallestChild = 2*rootIndex + 1;

  if ( smallestChild < heapSize ) {
    int sibling = smallestChild + 1;
    if ( sibling < heapSize &&
         values[ sibling ] < values[ smallestChild ] ) {
         smallestChild = sibling;
    }

    if ( values[rootIndex] > values[smallestChild] ) {
      swap( &values[rootIndex], &values[smallestChild] );
      heapify( values, smallestChild, heapSize );
    }
  }
}
```

  - This version of heapify is an example of a relatively simple form or recursion known as *tail recursion*. The recursive call is the last thing the method does.
  - A tail recursive method can be easily rewritten (as I did) using a loop instead of recursion. A good compiler will do this automatically!
  - There are more interesting examples of recursion where saving the values of $ra and other registers is unavoidable.

- Quicksort is a nice example of this "harder" form of recursion.

```
// Sort an array of integers using the quicksort algorithm
void quicksort(int list[], int low, int high ) {
  int mid;
  if( low < high) {
    // partition the list into two sublists
    mid = partition( list, low, high );

    // recursively sort the lesser list
   quicksort(list, low, mid-1);
   quicksort(list, mid+1, high);
  }
}
```

- The first recursive call of quick sort will use and probably change all of the registers that the function is using. When this first call returns, the original values of the variables mid and high will be needed to make the second recursive call.
- To implement any deeply nested sequence of function calls, it is very useful to store information in memory in addition to in registers.
- To implement a recursive function, it is essential to use memory rather than just registers.

The Function Call Stack

1. Function calls and returns have the Last-in-First-out property. The last function called must return before the function that called it can return.
2. As a result, if we allocate memory to hold information for functions when they are called and release the memory when a function returns, a stack can be used to hold the parcels of memory provided for each function. These parcels are called call frames.
3. On the MIPS machine, one of the 32 registers, $sp, is dedicated to keeping track of the stack of call frames.
   - The MIPS stack grows toward smaller memory addresses.
   - The value in $sp at any point is the address of word with the smallest address that is in use on the stack.
   - When a new function is invoked, its code should grow the stack by subtracting the number of bytes it needs from $sp.
   - When a function returns, it should release its frame on the stack by adding the number of bytes it was using to $sp.
4. Recall that the MIPS registers include a number of registers whose names start with s. These are the "saved" registers. There are also temporary registers whose names start with t.
5. These register names reflect a standard/convention that MIPS programmers follow when deciding which registers to use and what to save on the stack.
   - A function that is using saved registers can assume that these registers will not be changed when another function is called.
   - A function that uses saved registers must make sure that the values that were in those registers are restored to their original values before it returns.
   - A function should not assume that the values in any other registers (particularly the t and a registers) will be preserved if another function is called.
   - A function that calls other functions must save its return address before such calls.
6. To see how this applies in practice, consider the quick sort code shown above:
   - The method includes 4 variable/parameter names (list, low, high, and mid).
     - The local variable mid must be preserved across the first recursive call, so it is best held in a saved register (like $s0).
     - The parameters, list, low, and high could be handled in one of two ways. We could move them to s registers at the very start.

This would require saving and later restoring the previous values of the s registers. Alternately, we can save the initial values of the a registers on the stack and restore them as needed.

- The method also needs to save its own return address.
- Altogether, there are 5 values quick sort needs to save, so the first instruction in the translation of the method will decrement $sp by 5×4.
- The code for the method should begin with comments clearly explaining the plan for using registers and stack space followed by code to allocate space on the stack and to save the necessary values.

```
# void quicksort(int list[], int low, int high ) {
#   int mid;

quicksort:
# register usage:                stack frame:
#   $a0 points to list               saved at 0($sp)
#   $a1 = low                        saved at 4($sp)
#   $a2 = high                       saved at 8($sp)
#   $s0 = mid                        saved at 12($sp)
#    return point = $ra              saved at 16($sp)
#
# PROLOGUE:
     addiu   $sp,$sp,-20
     sw      $a0,0($sp)
     sw      $a1,4($sp)
     sw      $a2,8($sp)
     sw      $s0,12($sp)
     sw      $ra,16($sp)
```

- It is good practice to immediately place code at the end of the function to restore necessary values. Note, that the function is responsible for restoring the s registers but not the a registers.

```
lw $s0,12($sp)
lw $ra,16($sp)
addiu $sp,$sp,20
jr $ra
```

7. The code for the method body will include load instructions to restore the initial values of the a registers when they are needed:

```
#   if( low < high) {
     slt $t0,$a1,$a2
     beq $t0,$0,skipIf

#     // partition the list into two sublists
#     mid = partition( list, low, high );
     jal partition
     move $s0,$v0
#
#     // recursively sort the lesser list
#    quicksort(list, low, mid-1);
     lw $a0,0($sp)
     lw $a1,4($sp)
     addi $a2,$s0,-1
     jal quicksort

#     quicksort(list, mid+1, high);
     lw $a0,0($sp)
     addi $a1,$s0,1
     lw $a2,8($sp)
     jal quicksort
#   }
skipIf:
```