CS 237 Meeting 17 — 10/19/12

Announcements

1. Midterm: New date: Oct 29th. In class open book/notes.

Robust Function Call Conventions for MIPS

1. Last class, I discussed how MIPS programmers could use the stack to ensure that called functions would preserve elements of the computer's state (mainly register values) that a calling function might depend on.

2. I commented on my surprise that the MIPS architecture did not seem to have a widely accepted set of standards/conventions for how the information a particular function put on the stack should be formatted. All of the other architectures with which I am familiar have such conventions.

3. After class, I concluded that it would have been better to pick some standard and treat it as universal for the purpose of explaining how the stack is used. Basically, I fear that by exploring the flexibility provided by MIPS lack of standards (i.e., by sometimes saved $a register values on the stack and sometimes storing them in $s registers), I just made things more confusing.

4. So, today I want to finish our discussion of function stack frames on MIPS by pretending there is only one way that all functions must use the stack!

5. The "standard" I am going to discuss is the result of merging standards I found in various texts on on other architecture course web sites.

6. It is based on the existing standard for MIPS register usage:

   Saved Registers Registers $s0 through $s7 (otherwise know as registers 16 through 23) must contain the same values when a function returns as they did when it was called.

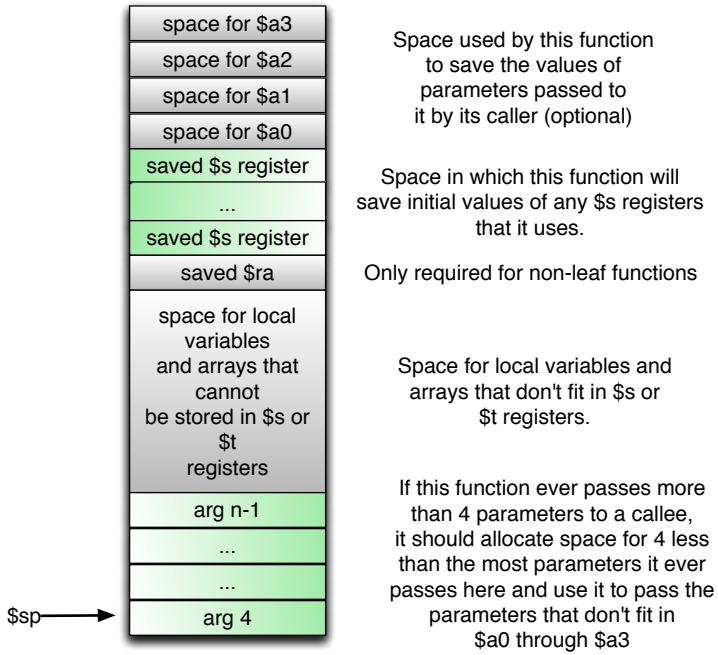
   Temporary Registers Registers $t0 through $t9 may have their values changed during a function call. The calling function must not depend on values left in these registers before a call.

   Function Linkage Registers Registers $a0 through $a3 are used to pass the first four arguments to a function. The called function may change the values in these registers before it returns. Registers $v0 and $v1 are used to hold any value a function is expected to return, so a calling function should assume that the values in these registers may be changed during a call.

   Stack Pointer Register $sp holds the address of the lowest byte used on the function stack. This will also be the lowest byte of the currently executing function. A called function must modify $sp to hold the address of its own frame and then restore $sp to refer to its caller's frame before it returns.

   Return Address Register $ra holds the current function's return address. A calling function must save this value before executing a jal instruction.

7. The layout of our standard frame is described by the following diagram (and the text that follows):



Starting from the bottom of the diagram:

- If a function passes more than 4 parameters to a callee, it is expected to put the first four parameter values in $a0 through $a3 and to place the remaining parameter values on the stack just above the callee's frame with the last parameter at the highest address and parameter 4 (assuming we number the parameters starting at 0) just above the callee's frame. As a result, if a function makes any such calls, it should allocate enough space in its frame for the most parameters required by any call.
- If a function has so many local variables that they cannot all be kept in registers, or it has a local variable whose address is used (i.e., the C & (address of) operator is applied to the variable), or has arrays declared as local variables, space for variables should be allocated in the stack frame.
- If the function calls any other function, a word should be allocated to save this functions return address. It should be saved in the function's prologue. A function that does not call other functions is called a *leaf* function.
- If the function uses any $s registers, it should allocate one word for each register used, save the values of these registers in its prologue and restore them in its epilogue.
- If the function chooses to save the values of its parameters on the stack, it should make room for them at the top of its frame. This way, in the case that a method takes more than 4 parameters, the saved values together with the values passed by the caller on the stack will form an array containing all of the parameter values.

8. Given this guidance, we can revisit the question of how we might use the stack while translating the functions in the quicksort program into MIPS assembly code.

   - First, consider the code for the quicksort function itself:

```
// Sort an array 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 function clearly never calls another function with more than 4 parameters, so we don't need space in its frame for parameter passing.
   - The function has only one local variable and it is not an array, so we don't need space for local variables in the stack frame.
   - The function does call other functions, so we will need a slot to save $ra in the stack frame. This will be at displacement 0 from $sp.
   - The local variable mid needs to retain its value across calls to other functions, so we will keep it in a saved register, $s0. We will use the word at 4($sp) to save any value held in $s0 by the function that called quicksort.
   - We could move the $a registers to some $s registers, or we could save them in the stack frame and restore them as needed. Since there is no loop in the function, the number of restores required is quite limited. Therefore, we will opt to keep these values in $a registers and need 3 words in the frame for this.
   - The overall frame will therefore look like:

9. The code for this function's prologue will make room on the stack for this frame and place the values to be saved in the appropriate slots:

```
quicksort:
# REGISTER USAGE                STACK FRAME
#       list = a0               saved at 16($sp)
#       low  = a1               saved at 12($sp)
#       high = a2               saved at 8($sp)
#       mid = s0                saved at 4($sp)
#       $ra                     saved at 0($sp)
#
# PROLOGUE:
     addiu   $sp,$sp,-20
     sw      $a0,16($sp)
     sw      $a1,12($sp)
     sw      $a2,8($sp)
     sw      $s0,4($sp)
     sw      $ra,0($sp)
```

10. The code in the epilogue will not need to restore the $a registers, but it does need to restore $s0, $ra, and $sp:

```
# EPILOGUE
    lw          $s0,4($sp)
    lw          $ra,0($sp)
    addiu       $sp,$sp,20
    jr          $ra
```

11. After any call that might change the values of the $a registers, any necessary values must be restored from the stack:

```
#     // 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    $a0,16($sp)
           lw    $a1,12($sp)
           addi  $a2,$s0,-1
           jal quicksort
```

12. We would perform a similar analysis of each function to be translated. The partition function:

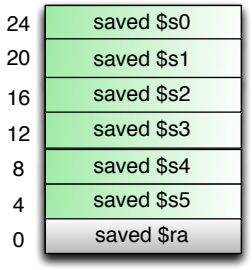
```
// Split the array into two sections so that all
// values in the first section are less than those
// in the second
int partition( int list[], int low, int high ) {
  int left, right, mid, pivot;

  mid = (low + high )/2;
  swap(&list[low],&list[mid]);
  pivot = list[low];
  left = low+1;
  right = high;
  while(left <= right) {
      while((left <= high) && (list[left] <= pivot)) {
       left++;
      }
      while((right >= low) && (list[right] > pivot)) {
       right--;
      }
      if( left < right) {
       swap(&list[left],&list[right]);
      }
  }
  // swap two elements
  swap(&list[low],&list[right]);
  return right;
}
```

requires no space for extra arguments or local variables at the end of the stack frame.

13. It does, however, need to use 3 saved registers for its local variables (the fourth local variable does not need to be preserved over any call so it can be kept in a temporary register).

14. In addition, since the argument values might be damaged by the call to swap that appears in a loop, it is better to move the three arguments into saved registers rather than to restore them each time around the loop.

15. This leads to a stack frame that looks like:



a prologue that both saves many values in the stack frame and moves argument values to saved registers:

```
# int partition( int list[], int low, int high ) {
#   int left, right, mid, pivot;
partition:
#
# REGISTER USAGE/FRAME LAYOUT:
#
# $s0 = list (from a0)      saved at 24($sp)
#  $s1 = low  (from a1)     saved at 20($sp)
#  $s2 = high (from a2)     saved at 16($sp)
#  $s3 = left               saved at 12($sp)
#  $s4 = right              saved at 8($sp)
#  $s5 = pivot              saved at 4($sp)
#  $ra                      saved at 0($sp)
#  $t0 = mid

subi $sp,$sp,28
sw $ra,0($sp)
sw $s5,4($sp)
sw $s4,8($sp)
sw $s3,12($sp)
sw $s2,16($sp)
sw $s1,20($sp)
sw $s0,24($sp)

move $s0,$a0
move $s1,$a1
move $s2,$a2
```

and an epilogue that restores all the saved values:

```
# EPILOGUE
lw $ra,0($sp)
lw $s5,4($sp)
lw $s4,8($sp)
lw $s3,12($sp)
lw $s2,16($sp)
lw $s1,20($sp)
lw $s0,24($sp)
addi $sp,$sp,28
jr $ra
```

Structures in C

1. Programming in an object-oriented language like Java depends heavily on the definition of classes that collect a group of variables and functions (method) that manipulate them.

2. C provides a lower-level facility that allows the programmer to describe a collection of related variables called a structure. Using structures, one can simulate much of the functionality of a class by defining appropriate functions.

3. Consider a simple example of a Java class:

```
public class Point {

    private double x;
    private double y;

    public Point( double xCoord, double yCoord ) {
        x = xCoord;
        y = yCoord;
    }

    public void move( double xDiff, double yDiff ) {
        x = x + xDiff;
```

```
        y = y + yDiff;
    }

    public double distanceTo( Point otherPoint ) {
        double xDiff = x - otherPoint.x;
        double yDiff = y - otherPoint.y;
        return Math.sqrt( xDiff*xDiff + yDiff*yDiff );
    }

. . .
}
```

4. This defines a new type of object with two components (x and y) together with methods to manipulate these objects.

5. In C, the notation

```
struct point {
    double x;
    double y;
}
```

   describes a new type of values composed of pairs of values of the double type. This new type is called a struct or structure. Unlike the Java class, a structure is just a collection of variables. There are no functions.

6. Just as other type names like int and char can be used to declare variables, we can define a variable using a structure type by placing the type description before the name we wish to declare and ending the declaration with a semicolon. this, we could declare a variable `somePoint` as a structure by saying:

```
struct point {
    double x;
    double y;
} somePoint;
```

7. Better yet, once we have describe a struct we can use the keyword struct followed by the name we used to define other variables:

```
struct point someOtherPoint;
```

   Basically, we don't have to repeat the components of the struct.

8. Give a variable of a struct type, we can access its components using periods and component names. For example, we could print the coordinates of a point using the statement

```
printf( "( %f, %f )\n" , somePoint.x, somePoint.y );
```

9. We can also pass struct values as parameter and return them as the results of a function. Here are three examples of functions (two of which mimic methods from our Java Point class) that take point parameters:

```
void move( struct point p, double xDiff, double yDiff ) {
  p.x = p.x + xDiff;
  p.y = p.y + yDiff;
}

double distanceTo( struct point p, struct point otherPoint ) {
  double xDiff = p.x - otherPoint.x;
  double yDiff = p.y - otherPoint.y;
  return sqrt( xDiff*xDiff + yDiff*yDiff );
}

void printPoint( struct point p ) {
  printf( "( %f, %f )\n" , p.x, p.y );
}
```

typedefs

1. Structure definitions are often used together with another C mechanism that allows the programmer to define new type names.

2. For example, we have seen that C does not provide a boolean type. When we want to work with boolean values, we typically include the #defines:

```
#define TRUE 1
#define FALSE 0
```

   in our code and define "bool" variables like:

```
int matchFound;
```

3. The declaration

```
typedef int bool;
```

   tells C that we want to use bool as another name for the type int. Therefore, if we include such a declaration we can then say

```
bool matchFound;
```

4. In general, a typedef looks like a variable declaration preceded by the word typedef. Its effect is to associate the name that would have been a variable with the type with which that variable would be associated if typedef had been omitted. For example,

```
typedef int numberList[10];
```

   define numberList to be a name for the type of arrays of 10 integers. After this declaration, saying

```
numberList scores;
```

   would define scores as an array.

5. As mentioned above, it is common to use a typedef to associate a name with a structure that can be used without first saying struct. For example, if we said:

```
typedef
    struct point {
           double x;
           double y;
        } point;
```

   we could then define variables as

```
point somePoint;

point someOtherPoint;
```

   and also use the new name in parameter declarations as in:

```
void move( point p, double xDiff, double yDiff ) {
  p.x = p.x + xDiff;
  p.y = p.y + yDiff;
}

double distanceTo( point p, point otherPoint ) {
  double xDiff = p.x - otherPoint.x;
  double yDiff = p.y - otherPoint.y;
  return sqrt( xDiff*xDiff + yDiff*yDiff );
}
```