CS61C
Review of Procedures; MIPS Argument Conventions and Variable Number Arguments

Lecture 24
April 23, 1999

Mark D. Spiller

(www-cad.eecs.berkeley.edu/~mds)
(replacing Dave for a day...)
Review 1/1 - Performance

° Response Time v. Throughput

° “X times faster than Y” is Time(Y)/Time(X) or Performance(X)/Performance(Y)

° Iron triangle: need product of all terms to have meaningful comparison (execution time): Instruction Count, Clocks Per Instruction, Clock Rate

° “For Better or Worse, Benchmarks Shape a Field”
  • Benchmark selection important as well as proper metrics
Overview

° Review - Procedures & Registers
° What happens when you have many arguments?
° Administrivia, QUIZ!, “Computers in the news”
° Variable Numbers of Arguments
° Project 6 (sprintf)
° Recursion
° Conclusion (1 minute)
Review - Procedures

```c
main() {
    int i, j, k, m;
    i = mult(j, k); ... ;
    m = mult(i, i); ...
}

int mult (int mcand, int mlier){
    int product;

    product = 0;
    while (mlier > 0) {
        product = product + mcand;
        mlier = mlier - 1;
    }
    return product;
}
```

What information must compiler/programmer keep track of?
Review - Registers in Procedures

main() {
    int i, j, k, m;

    i = mult(j, k); ...;
    m = mult(i, i); ...;
}

int mult (int mcand, int mlier) {
    int product;
    product = 0;
    while (mlier > 0) {
        product += mcand;
        mlier = mlier - 1;
    }
    return product;
}
Review: Instruction Support

```c
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

**M**

<table>
<thead>
<tr>
<th>Address</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>move $a0,$s0  # x = a</td>
</tr>
<tr>
<td>1004</td>
<td>move $a1,$s1  # y = b</td>
</tr>
<tr>
<td>1008</td>
<td>jal sum      # jump to sum</td>
</tr>
<tr>
<td>1012</td>
<td>...</td>
</tr>
</tbody>
</table>

**S**

<table>
<thead>
<tr>
<th>Address</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>sum: add $v0,$a0,$a1</td>
</tr>
<tr>
<td>2004</td>
<td>jr $ra</td>
</tr>
</tbody>
</table>
**Review: Exceeding Limits of Registers**

° **Temporary registers: $t0, ..., $t9**
  - What if you need more than 10 words of local variables? Stack! ($sp)

° **Callee-Saved:**
  - When A calls B, if A uses any of the $s registers, it must first save their values on the stack, and then restore them to the registers before returning.

° **Caller-Saved:**
  - A must store on the stack any values in the $t registers that it will need after the call. (also true for $a registers)
Arguments

° Have limited number of registers for passing arguments ($a0-$a3, $f12,$f14)

° What happens if you need to pass more than 4 words worth of data?

Foo(int val1, int val2, int val3,
    int val4, int val5, int val6);

Bar(double val1, double val2,
    double val3);

Baz(char char1, int val1, double val2,
    char *str1);

RdCache(struct SimulatedComputer sim);
Argument Conventions

° For performance, you want to use registers as much as possible, since they are fast.
  
  • No matter how many registers you have for arguments, there will always be cases where you need more! Where do you go?
  
  • Stack!

° Part of designing an architecture and its compiler is the definition of the argument-passing convention.

° We’ll take a look at the MIPS convention.
The MIPS Argument Convention

° The first 4 words of data will always go in registers, when possible.
  • Even though we have 8 words worth of argument registers ($a0-$a3, $f12, $f14), only 4 words are used!

° Additional arguments/data go onto the stack

° What kind of mapping might you see?
  • (double, int, int) -> $f12, $a2, $a3
  • (single, int, int, int) -> $f12, $a1, $a2, $a3
  • (char *, char, int, char) -> $a0-$a3
The MIPS Argument Convention (cont.)

° But it can get tricky!
  • Characters get “promoted” to int
  • If a float is not an initial argument, it may go into an integer register!!!
    - (int, int, double) -> $a0, $a1, ($a2,$a3) !!!
  • Space is still allocated on the stack for the first four arguments, even if not used...

° How do you figure this out? The manual!
  • Read “Mips RISC Architecture”, by G. Kane

° (Fear not, project 6 is NOT this tricky!)
A Simple Example

SetInfo(char initial, char *name
       int age, char *company,
       char *title, double salary);

° How many of these arguments will fit into registers?

.. SetInfo(‘S’, “Mark”, 16, “UCB”
          “Gradual Student”, 1e-15)
Example

SetInfo( ‘S’, “Mark”, 16, “UCB”, “gradual student”, 1e-15);

° $a0 - (char) ‘S’
° $a1 - (char *) -> “Mark”
° $a2 - (int) 16
° $a3 - (char *) -> “UCB”
° arg5 - (char *) -> “g...”
° arg6 - (double) 1e-15
Example: Caller Setup

```c
..SetInfo('S','Mark',16,"UCB"
   "Gradual Student", 1e-15)
```

---

**address**

```
1000 move $a0, $s0  # 'S'
1004 move $a1, $s1  # char*
1008 move $a2, $s2  # 16
1012 move $a3, $s3  # char*
1016 add $sp, $sp, -28  # (stack)
1020 sw  $s4, 16($sp)  # char*
1024 sw  $s5, 20($sp)  # dbl-1
1028 sw  $s6, 24($sp)  # dbl-2
1032 jal SetInfo
1036 add $sp, $sp, 28
1040 ...
```
Crossing Stack Frames

° The caller puts the arguments onto the end of its stack frame before calling the callee.
  • The function being called must know how many arguments it takes, so that it can pick them off the caller’s stack frame if necessary.

° Frame pointer points to the beginning (top) of the callee stack frame.
  • Can be used to access the stack args
  • If the frame pointer is not used, can also use the stack pointer, but more complex
Example

setInfo(‘S’, “Mark”, 16, “UCB”, “gradual student”, 1e-15);

° $a0 - (char) ‘S’
° $a1 - (char *) -> “Mark”
° $a2 - (int) 16
° $a3 - (char *) -> “UCB”
° arg5 - (char *) -> “g…”
° arg6 - (float) 1e-15
Example: Callee Getting Arguments

```
..SetInfo(‘S’, “Mark”, 16, “UCB”
    “Gradual Student”, 5)
```

```
SetInfo:

<table>
<thead>
<tr>
<th>C</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>add $sp,$sp,-4 # save $ra</td>
</tr>
<tr>
<td>2004</td>
<td>sw $ra,0($sp) #</td>
</tr>
<tr>
<td>2016</td>
<td>lw $t0,16($fp) # ld arg5</td>
</tr>
<tr>
<td>2020</td>
<td>lw $t1,20($fp) # ld arg6-1</td>
</tr>
<tr>
<td>2024</td>
<td>lw $t1,24($fp) # ld arg6-2</td>
</tr>
<tr>
<td>2040</td>
<td>lw $ra, 0(sp) # get $ra</td>
</tr>
<tr>
<td>2044</td>
<td>jr $ra # return</td>
</tr>
</tbody>
</table>
```
**Administrivia: Rest of 61C**

**W 4/28** Processor Pipelining; section 6.1
Project 6 due

**F 4/30** Review: Caches/TLB/VM; section 7.5
Last homework due

**M 5/3** Deadline to correct your grade record

**W 5/5** Review: Interrupts/Polling
Final 61C Survey in lab

**F 5/7** 61C Summary / Your Cal heritage

**Sun 5/9** Final Review starting 2PM (1 Pimintel)

**W 5/12** Final (5PM 1 Pimintel)

• Need Alternative Final? Contact mds@cory
Pop Quiz!!!

° How many of you read the newsgroup?
° How many of you know what (who?) this is?:

M’Piero!!

Artistic credits: Alex Fabrikant
Summer Mishcherghi

° If you don’t get this joke, it just goes to show that if you don’t read the newsgroup, you may be missing the best parts of the class!
“Computers in the News”

° Phones -> Cell phones -> Satellite Phones
How long until wires are obsolete?
Until you see more satellites than stars in the sky?

Halo
(www.angeltechnologies.com)

Iridium
(www.iridium.com)

Teledesic
(www.teledesic.com)

Qualcomm announces wireless payphone...
Procedures with Variable #s of Args

° What happens if you need a function to handle many different types of arguments? Why do you need this?

° Imagine that you want to print to the screen different types of data:
  • “Values are: “ (int) (int) (int)
  • Could do: myPrint(char*, int, int, int)
  • “Values are:” (int) (double)(int)
  • Function2: myPrint(char*, int, double, int)

° A function for every variation of arg type is clearly a waste of time!
Variable Args (cont.)

° How about:
  • “Values are: “ (int) (double) (int)
  • printStr(“Values are:’’);
    printInt(int);
    printDouble(double);
    printInt(int);
  • Also not very pretty/convenient

° Different ways of doing this, Java for instance
  • System.out.println(“String” + int + double)
  • Basic types are objects, have the ability to convert themselves to strings.
C “...” Argument

- C uses the “...” argument to represent a variable number of arguments.
  - Used in printf, sprintf, scanf, error msgs
    - printf(char *fmt, ...);

- When writing the function code, there has to be a way to find:
  - The number of arguments - there is no indicator in the stack frame specifying where the arguments end!
  - The types of the arguments - a value in memory could be anything! (int, pointer, char,...)
Variable Arguments - printf

° The first argument in printf indicates the number of arguments and types
  • “%” is a special marker that indicates that something will be expanded at that point
    • `printf("%d days left in %s\n", 22, "61C");`
° The function definition of printf has go through the first argument and determine the rest of the arguments and what to do with them.
Where Do Variable Arguments Go?

◦ Required arguments follow the normal convention
  • The first four arguments go in the registers
  • Any additional arguments get passed on the stack

◦ Variable arguments go into the integer registers if possible, stack otherwise
Compiling Procedures Using “…”

° Compilers have to be more lax with variable argument procedures:

- `int foo(int x, int y)`
  - `foo(1,2,3);` will NOT compile

- `int printf(char *fmt, ...)`
  - `printf("Value is %d");`
  - `printf("Values are %d, %d", 1, 2, 3);`

° Could this cause problems?

- For `printf`, probably not
  - Printing out, not modifying = print garbage

- Other functions using “…” - it might!
#include <stdarg.h>

void printf(char *fmt, ...) {
    va_list ap;     #allocate an arg list
    va_start(ap, fmt);   #initialize it
    ...

    for(p = fmt; *p; p++)  #go through fmt, find
    { ...  #a ‘%’
        case ‘d’:
            intval = va_arg(argptr, int);
            /*convert and print out the integer*/
            ...
    }

    va_end(ap) #clean-up
}
Project 6 - sprintf

° Hands-on experience with variable args, basically doing what the C program just shown does!

° But MUCH simpler than the real thing!
  • All args are on the stack! (You don’t have to worry about what goes into registers...)
  • Only have to deal with %{d,x,c,s,%} - How many words does each of these use?

° Basically, just need to parse through char *fmt, find the right argument, convert it to a string if necessary, and put it into char *outbuf...
Thought Exercise: Recursion

What happens when a program calls itself? For instance:

```c
int Factorial(int intVal) {
    if (intVal == 1)
        return 1;
    else
        return (intVal * Factorial(intVal - 1));
}
```

If it follows all of the register and stack conventions, it should be just like calling any other procedure, and will “just work”!
“And in Conclusion …” 1/1

° Every machine has a convention for how arguments are passed.

° In MIPS, where do the arguments go if you are passing more than 4 words? Stack!

° It is sometimes useful to have a variable number of arguments.
  • The C convention is to use “…”
  • *fmt is used to determine the number of variables and their types.

° Easy, right? No more new projects, labs! You’re in the home stretch!