Lec 12: Register Calling

Kavita Bala
CS 3410, Fall 2008
Computer Science
Cornell University

Announcements

• PA 1 is due this Wed

• Ask us questions if in doubt
Program Layout

- Programs consist of segments used for different purposes
  - Text: holds instructions
  - Data: holds statically allocated program data such as variables, strings, etc.

When you run the program

- Stack segment
- Dynamic data
- Static data
- Data segment
- Text segment
- Reserved
Assembling Programs

• Programs consist of a mix of instructions, pseudo-ops and assembler directives

• Assembler lays out binary values in memory based on directives

Procedures

• Enable code to be reused by allowing code snippets to be invoked

• Will need a way to
  – call the routine
  – pass arguments to it
    ▪ fixed length
    ▪ variable length
    ▪ Recursive calls
  – return value to caller
  – manage registers
Call Stacks

- A call stack contains activation records (aka stack frames)

- Each activation record contains
  - the return address for that invocation
  - the local variables for that procedure

Take 3: JAL/JR with Activation Records

- Stack used to save and restore contents of $31
Many Arguments

• What if there are more than 4 arguments?
  
  • Use the stack for the additional arguments—“spill”

```assembly
main:
  li a0, 0
  li a1, 1
  li a2, 2
  li a3, 3
  addiu sp,sp,-8
  li $8, 4
  sw $8, 0(sp)
  li $8, 5
  sw $8, 4(sp)
  jal subf
// result in v0
```

Variable Length Arguments

• Best to use an (initially confusing but ultimately simpler) approach:
  – Pass the first four arguments in registers, as usual
  – Pass the rest on the stack
  – Reserve space on the stack for all arguments, including the first four

• Simplifies functions that use variable-length arguments
  – Store a0-a3 on the slots allocated on the stack, refer to all arguments through the stack
Register Layout on Stack

- First four arguments are in registers
- The rest are on the stack
- There is room on the stack for the first four arguments, just in case

 Globals and Locals

- Global variables are allocated in the “data” region of the program
  - Exist for all time, accessible to all routines

- Local variables are allocated within the stack frame
  - Exist solely for the duration of the stack frame

- Dangling pointers are pointers into a destroyed stack frame
  - C lets you create these, Java does not
  - int *foo() { int a; return &a; }
Frame Layout on Stack

blue() {
    pink(0, 1, 2, 3, 4, 5);
}

pink() {
    orange(10, 11, 12, 13, 14);
}

Buffer Overflows

orange() {
    char buf[100];
    gets(buf); // read string, no check
}
Frame Pointer

- It is sometimes cumbersome to keep track of location of data on the stack
  - The offsets change as new values are pushed onto and popped off of the stack

- Keep a pointer to the top of the stack frame
  - Simplifies the task of referring to items on the stack

- A frame pointer, $30$, aka fp
  - Value of sp upon procedure entry
  - Can be used to restore sp on exit
Register Usage

• Suppose a routine would like to store a value in a register

• Two options: caller-save and callee-save

• What is tradeoff?
  – If all caller save, could be waste
  – If all callee save, could be waste

• MIPS calling convention supports both
  – Callee-save regs: $16-$23 (s0-s7)
  – Caller-save regs: $8-$15,$24,$25 (t0-t9)

Register Usage

• Callee-save
  – Save it if you modify it
  – Assumes caller needs it
  – Save the previous contents of the register on procedure entry, restore just before procedure return
  – E.g. $31 (what is this?)

• Caller-save
  – Save it if you need it after the call
  – Assume callee can clobber any one of the registers
  – Save contents of the register before proc call
  – Restore after the call
Caller-Save

- Assume registers are free for the taking
- But other subroutines will do the same
  - must protect values that will be used later
  - save and restore them before and after subroutine invocations
- Pays off if a routine makes few calls to other routines with values that need to be preserved

```
main:
  ...
  [use $9 & $8]
  ...
  addiu sp,sp,-8
  sw $9, 4(sp)
  sw $8, 0(sp)
  jal mult
  lw $9, 4(sp)
  lw $8, 0(sp)
  addiu sp,sp,8
  ...
  [use $9 & $8]
```

Callee-Save

- Assume caller is using the registers
- Save on entry, restore on exit
- Pays off if caller is actually using the registers, else the save and restore are wasted

```
mult:
  addiu sp,sp,-12
  sw $31,8(sp)
  sw $17, 4(sp)
  sw $16, 0(sp)
  ...
  [use $17 and $16]
  ...
  lw $31,8(sp)
  lw $17, 4(sp)
  lw $16, 0(sp)
  addiu sp,sp,12
```
Leaf vs. non-leaf

• Leaf
  – Simple, fast
  – Don’t save registers

• int f(int x, int y) {return (x+y);}

• f:
  add $v0, $a0, $a1  # add x and y
  j $ra            # return
  nop

• Or
  j $ra
  add $v0, $a0, $a1

Example

f:  beq $a1, $zer0, Done
    nop
    addi $sp, $sp, -12
NotDone: sw $ra, 8($sp)
    sw $a0,4($sp)
    sw $a1,0($sp)
    move $a0, $a0
    subi $a1, $a1, 1
    jal f
    nop
    lw $a0,4(sp)
    lw $a1,0(sp)
    lw $ra,8(sp)
    addi $sp, $sp, 12
    add $v0, $a0, $v0
    j Exit
    nop
Done: move $v0, $zero
Exit: return $ra
Mult example

Main () { int res = mult (a, b);}

int Mult (int a, int b) {
  if (b == 0) {return 0;}
  else {
    res = a + mult (a, b-1);
    return res;
  }
}

Translates to
Main:
  move a0, a
  move a1, b
  jal mult

Preserved vs. Not preserved

• Preserved (Callee Save)
  – $s0-$s7
  – Save prior to use, restore before return
  – $sp$, $fp$, $gp$, $ra$

• Not preserved (Caller Save)
  – $t0$-$t9$, $a0$-$a3$, $v0$, $v1$
  – Saved by caller if needed after proc call
MIPS Register Recap

- Return address: $31 (ra)
- Stack pointer: $29 (sp)
- Frame pointer: $30 (fp)
- First four arguments: $4-$7  (a0-a3)
- Return result: $2-$3  (v0-v1)
- Callee-save free regs: $16-$23  (s0-s7)
- Caller-save free regs: $8-$15,$24,$25 (t0-t9)
- Reserved: $26, $27
- Global pointer: $28 (gp)
- Assembler temporary: $1 (at)

What happens on a call?

- **Caller**
  - Save caller-saved registers $a0-$a3, $t0-$t9
  - Load arguments in $a0-$a3, rest passed on stack
  - Execute jal
- **Callee Setup**
  - Allocate memory for new frame ($sp = $sp-frame)
  - Save callee-saved registers $s0-$s7, $fp, $ra
  - Set frame pointer ($fp = $sp-frame-4)
- **Callee Return**
  - Place return value in $v0 and $v1
  - Restore any callee-saved registers
  - Pop stack ($sp = $sp + frame size)
  - Return by jr $ra
Example

f:  slti $t0, $a0, 2
    beq $t0,$zero, skip
    ori $v0, $zero, 1
    jr $ra
skip:  addiu $sp, $sp, -32
       sw $ra, 28($sp)
       sw $fp, 24($sp)
       addiu $fp, $sp, 28
       sw $a0, 32($sp)
       addui $a0, $a0, -1
    jal f
link:  lw $a0, 32($sp)
       mul $v0, $v0, $a0
       lw $ra, 28($sp)
       lw $fp, 24($sp)
       addiu $sp, $sp, 32
    jr $ra  #return
Factorial

```c
int fact(int n) {
    if (n <= 1) return 1;
    return n * fact(n-1);
}
```

```asm
fact: slti $t0, $a0, 2          # a0 < 2
    beq $t0,$zero, skip   # goto skip
    ori  $v0, $zero, 1     # return 1
    jr  $ra
skip: addiu $sp, $sp, -32   # $sp down 32
    sw  $ra, 28($sp)        # save $ra
    sw  $fp, 24($sp)        # save $fp
    addiu $fp, $sp, 28     # set up $fp
    sw  $a0, 32($sp)        # save n
    addui $a0, $a0, -1     # n = n-1
    jal fact
link: lw  $a0, 32($sp)        # restore n
    mul  $v0, $v0, $a0      # n * fact (n-1)
    lw  $ra, 28($sp)        # load $ra
    lw  $fp, 24($sp)        # load $fp
    addiu $sp, $sp, 32     #pop stack
    jr  $ra                #return
```

Foo and Bar

```c
int foo(int num) {
    return bar(num+1);
}
```

```c
int bar(int num) {
    return num + 1;
}
```

```asm
foo: addiu $sp, $sp, -32 #push frame
    sw  $ra, 28($sp)        #store $ra
    sw  $fp, 24($sp)        #store $fp
    addiu $fp, $sp, 28     #set new fp
    addiu $a0, $a0, 1      #num + 1
    jal  bar
    lw  $fp, 24($sp)        #load $fp
    lw  $ra, 28($sp)        #load $ra
    addiu $sp, $sp, 32     #pop frame
    jr  $ra
bar: addiu $v0,$a0,1     #leaf procedure
     jr  $ra            #with no frame
```

© Kavita Bala, Computer Science, Cornell University
From Assembly to Running

Big Picture

- Assembler output is obj files
  - Not executable
  - May refer to external symbols
  - Each object file has its own address space

- Linker joins these object files into one executable

- Loader brings it into memory and executes
Object File Generation

- A program is made up of code and data from several object files
- Each object file is generated independently
- Assembler starts at some PC address, e.g. 0, in each object file, generates code as if the program were laid out starting out at location 0x0
- It also generates a symbol table, and a relocation table
  - In case the segments need to be moved

Object file

- Header
  - Size and position of pieces of file
- Text Segment
  - instructions
- Data Segment
  - Static data
- Relocation Information
  - Instructions and data that depend on absolute addresses
- Symbol Table
  - External and unresolved references
- Debugging Information