Calling Conventions

Hakim Weatherspoon
CS 3410, Spring 2012
Computer Science
Cornell University

See P&H 2.8 and 2.12
Goals for Today

Calling Convention for Procedure Calls
Enable code to be reused by allowing code snippets to be invoked

Will need a way to

• call the routine (i.e. transfer control to procedure)
• pass arguments
  – fixed length, variable length, recursively
• return to the caller
  – Putting results in a place where caller can find them
• Manage register
Jumps and branches can transfer control to the callee (called procedure)
Jumps and branches can transfer control back
What happens when there are multiple calls from different call sites?
**Jump And Link**

JAL (Jump And Link) instruction moves a new value into the PC, and simultaneously saves the old value in register $31.

Thus, can get back from the subroutine to the instruction immediately following the jump by transferring control back to PC in register $31.
Procedure Call Take 2: JAL/JR

main:
jal mult

Laftercall1:
add $1,$2,$3
jal mult
Laftercall2:
sub $3,$4,$5
jr $31

mult:
...
...

JAL saves the PC in register $31
Subroutine returns by jumping to $31
What happens for recursive invocations?
Recursion overwrites contents of $31
Need to save and restore the register contents
Call Stacks

Call stack

- contains activation records (aka stack frames)

Each activation record contains

- the return address for that invocation
- the local variables for that procedure

A stack pointer (sp) keeps track of the top of the stack

- dedicated register ($29) on the MIPS

Manipulated by push/pop operations

- push: move sp down, store
- pop: load, move sp up
Stack Growth

Stacks start at a high address in memory

Stacks grow down as frames are pushed on

- Recall that the data region starts at a low address and grows up
- The growth potential of stacks and data region are not artificially limited
Anatomy of an executing program

- 0xfffffffffc: system reserved (top)
- 0x80000000: stack
- 0x7fffffffffc: dynamic data (heap)
- 0x10000000: static data
- 0x00400000: code (text)
- 0x0000000000: system reserved (bottom)
Take 3: JAL/JR with Activation Records

main:
  jal mult
Laftercall1:
  add $1,$2,$3
  jal mult
Laftercall2:
  sub $3,$4,$5

mult:
  addiu $sp,$sp,-4
  sw $31, 0($sp)
  beq $4, $0, Lout
  ...
  jal mult
Linside:
  ...
Lout:
  lw $31, 0($sp)
  addiu $sp,$sp,4
  jr $31

Stack used to save and restore contents of $31
Take 3: JAL/JR with Activation Records

Stack used to save and restore contents of $31

How about arguments?

main:
jal mult
Laftercall1:
add $1,$2,$3
jal mult
Laftercall2:
sub $3,$4,$5

mult:
addiu $sp,$sp,-4
sw $31, 0($sp)
beq $4, $0, Lout
...
jal mult
Linside:
...
Lout:
lw $31, 0($sp)
addiu $sp,$sp,4
jr $31
Arguments & Return Values

Need consistent way of passing arguments and getting the result of a subroutine invocation

Given a procedure signature, need to know where arguments should be placed

- \texttt{int min(int a, int b);}
- \texttt{int subf(int a, int b, int c, int d, int e);}
- \texttt{int isalpha(char c);}
- \texttt{int treesort(struct Tree *root);}
- \texttt{struct Node *createNode();}
- \texttt{struct Node mynode();}

Too many combinations of char, short, int, void *, struct, etc.
- MIPS treats char, short, int and void * identically
Simple Argument Passing

First four arguments are passed in registers
• Specifically, $4, $5, $6 and $7, aka $a0, $a1, $a2, $a3

The returned result is passed back in a register
• Specifically, $2, aka $v0

main:
  li $a0, 6
  li $a1, 7
  jal min
  // result in $v0
Conventions so far:

- args passed in $a0, $a1, $a2, $a3
- return value (if any) in $v0, $v1
- stack frame at $sp
  - contains $ra (clobbered on JAL to sub-functions)

Q: What about argument lists?
Many Arguments

What if there are more than 4 arguments?

Use the stack for the additional arguments

• “spill”

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

What if there are more than 4 arguments?

Use the stack for the additional arguments

• “spill”

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

```c
printf("Coordinates are: %d %d %d\n", 1, 2, 3);
```

Could just use the regular calling convention, placing first four arguments in registers, spilling the rest onto the stack

- Callee requires special-case code
- if(argno == 1) use a0, ... else if (argno == 4) use a3, else use stack offset

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

main:
    li $a0, 0
    li $a1, 1
    li $a2, 2
    li $a3, 3
    addiu $sp, $sp, -24
    li $8, 4
    sw $8, 16($sp)
    li $8, 5
    sw $8, 20($sp)
    jal subf
    // result in $v0
blue() {
    pink(0,1,2,3,4,5);
}
Frame Layout on Stack

```
return address
5
4
space for $a3
space for $a2
space for $a1
space for $a0
return address
```

```c
blue() {
    pink(0,1,2,3,4,5);
}
pink(int a, int b, int c, int d, int e, int f) {
    ... 
}
```
Conventions so far:

- **first four** arg words passed in $a0, $a1, $a2, $a3
- remaining arg words passed on the stack
- return value (if any) in $v0, $v1
- stack frame at $sp
  - contains $ra (clobbered on JAL to sub-functions)
  - contains extra arguments to sub-functions
  - contains **space** for first 4 arguments to sub-functions
<table>
<thead>
<tr>
<th>r0</th>
<th>$zero</th>
<th>zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>$at</td>
<td>assembler temp</td>
</tr>
<tr>
<td>r2</td>
<td>$v0</td>
<td>function</td>
</tr>
<tr>
<td>r3</td>
<td>$v1</td>
<td>return values</td>
</tr>
<tr>
<td>r4</td>
<td>$a0</td>
<td>function</td>
</tr>
<tr>
<td>r5</td>
<td>$a1</td>
<td>arguments</td>
</tr>
<tr>
<td>r6</td>
<td>$a2</td>
<td></td>
</tr>
<tr>
<td>r7</td>
<td>$a3</td>
<td></td>
</tr>
<tr>
<td>r8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r26</td>
<td>$k0</td>
<td>reserved</td>
</tr>
<tr>
<td>r27</td>
<td>$k1</td>
<td>for OS kernel</td>
</tr>
<tr>
<td>r28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r31</td>
<td>$ra</td>
<td>return address</td>
</tr>
</tbody>
</table>
Java vs C: Pointers and Structures

Pointers are 32-bits, treat just like ints

Pointers to structs are pointers

C allows passing whole structs

• int distance(struct Point p1, struct Point p2);

• Treat like a collection of consecutive 32-bit arguments, use registers for first 4 words, stack for rest

• Inefficient and to be avoided, better to use
int distance(struct Point *p1, struct Point *p2);

in all cases
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; }
Global and Locals

How does a function load global data?

- global variables are just above 0x10000000

Convention: *global pointer*

- r28 is $gp$ (pointer into *middle* of global data section)
  $gp = 0x10008000$
- Access most global data using LW at $gp +/-$ offset
  LW $v0, 0x8000($gp)
  LW $v1, 0x7FFF($gp)
Anatomy of an executing program

- Top:
  - System reserved

- Bottom:
  - Code (text)
  - Static data
  - Dynamic data (heap)
  - Stack
  - System reserved

Memory addresses:
- 0xfffffffffc
- 0x80000000
- 0x7fffffff
- 0x10000000
- 0x00400000
- 0x00000000

(top to bottom)
Frame Pointer

It is often 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
Suppose a routine would like to store a value in a register. Two options: callee-save and caller-save.

**Callee-save:**
- Assume that one of the callers is already using that register to hold a value of interest.
- Save the previous contents of the register on procedure entry, restore just before procedure return.
- E.g. $31

**Caller-save:**
- Assume that a caller can clobber any one of the registers.
- Save the previous contents of the register before proc call.
- Restore after the call.

MIPS calling convention supports both.
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

main:
  addiu $sp,$sp,-32
  sw $31,28($sp)
  sw $30, 24($sp)
  sw $17, 20($sp)
  sw $16, 16($sp)
  addiu $30, $sp, 28
  ...
  [use $16 and $17]
  ...
  lw $31,28($sp)
  lw $30,24($sp)
  lw $17, 20($sp)
  lw $16, 16($sp)
  addiu $sp,$sp,32
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

main:
addiu $sp,$sp,-32
sw $ra,28($sp)
sw $fp, 24($sp)
sw $s1, 20($sp)
sw $s0, 16($sp)
addiu $fp, $sp, 28
...
[use $s0 and $s1]
...
lw $ra,28($sp)
lw $fp,24($sp)
lw $s1, 20$sp)
lw $s0, 16($sp)
addiu $sp,$sp,32
Assume the registers are free for the taking, clobber them
But since other subroutines will do the same, must protect values that will be used later
By saving and restoring them before and after subroutine invocations
Pays off if a routine makes few calls to other routines with values that need to be preserved
Caller-Save

Assume the registers are free for the taking, clobber them

But since other subroutines will do the same, must protect values that will be used later

By saving and restoring 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 $t0 & $t1]
...
addiu $sp,$sp,-8
sw $t1, 4($sp)
sw $t0, 0($sp)
jal mult
lw $t1, 4($sp)
lw $t0, 0($sp)
addiu $sp,$sp,8
...
[use $t0 & $t1]
### Frame Layout on Stack

<table>
<thead>
<tr>
<th>fp</th>
<th>saved ra</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>saved fp</td>
</tr>
<tr>
<td></td>
<td>saved regs ($s0 ... $s7)</td>
</tr>
<tr>
<td></td>
<td>locals</td>
</tr>
<tr>
<td></td>
<td>outgoing args</td>
</tr>
</tbody>
</table>

```c
# allocate frame
# save $ra
# save old $fp
# save ...
# set new frame pointer
...
...
# restore ...
# restore ...
# restore old $fp
# restore $ra
# dealloc frame
```
Frame Layout on Stack

```plaintext
blue() {
    pink(0,1,2,3,4,5);
}
```
blue() {
    pink(0, 1, 2, 3, 4, 5);
}
pink(int a, int b, int c, int d, int e, int f) {
    orange(10, 11, 12, 13, 14);
}
Frame Layout on Stack

```
blue() {
    pink(0,1,2,3,4,5);
}
pink(int a, int b, int c, int d, int e, int f) {
    orange(10,11,12,13,14);
}
orange(int a, int b, int c, int, d, int e) {
    char buf[100];
    gets(buf); // read string, no check!
}
```
Buffer Overflow

```c
blue() {
    pink(0,1,2,3,4,5);
}
pink(int a, int b, int c, int d, int e, int f) {
    orange(10,11,12,13,14);
}
orange(int a, int b, int c, int, d, int e) {
    char buf[100];
    gets(buf); // read string, no check!
}
```
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)
### MIPS Register Conventions

<table>
<thead>
<tr>
<th>r0</th>
<th>$zero</th>
<th>zero</th>
<th>r16</th>
<th>$s0</th>
<th>saved (callee save)</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>$at</td>
<td>assembler temp</td>
<td>r17</td>
<td>$s1</td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td>$v0</td>
<td>function return values</td>
<td>r18</td>
<td>$s2</td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td>$v1</td>
<td></td>
<td>r19</td>
<td>$s3</td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td>$a0</td>
<td></td>
<td>r20</td>
<td>$s4</td>
<td></td>
</tr>
<tr>
<td>r5</td>
<td>$a1</td>
<td></td>
<td>r21</td>
<td>$s5</td>
<td></td>
</tr>
<tr>
<td>r6</td>
<td>$a2</td>
<td></td>
<td>r22</td>
<td>$s6</td>
<td></td>
</tr>
<tr>
<td>r7</td>
<td>$a3</td>
<td></td>
<td>r23</td>
<td>$s7</td>
<td></td>
</tr>
<tr>
<td>r8</td>
<td>$t0</td>
<td></td>
<td>r24</td>
<td>$t8</td>
<td>more temps (caller save)</td>
</tr>
<tr>
<td>r9</td>
<td>$t1</td>
<td></td>
<td>r25</td>
<td>$t9</td>
<td></td>
</tr>
<tr>
<td>r10</td>
<td>$t2</td>
<td></td>
<td>r26</td>
<td>$k0</td>
<td>reserved for kernel</td>
</tr>
<tr>
<td>r11</td>
<td>$t3</td>
<td></td>
<td>r27</td>
<td>$k1</td>
<td></td>
</tr>
<tr>
<td>r12</td>
<td>$t4</td>
<td></td>
<td>r28</td>
<td>$gp</td>
<td>global data pointer</td>
</tr>
<tr>
<td>r13</td>
<td>$t5</td>
<td></td>
<td>r29</td>
<td>$sp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>r14</td>
<td>$t6</td>
<td></td>
<td>r30</td>
<td>$fp</td>
<td>frame pointer</td>
</tr>
<tr>
<td>r15</td>
<td>$t7</td>
<td></td>
<td>r31</td>
<td>$ra</td>
<td>return address</td>
</tr>
</tbody>
</table>

- **r0** is used for the zero value.
- **r1** is used for an assembler temporary variable.
- **r2** and **r3** are used for function return values.
- **r4** to **r7** are used for function arguments.
- **r8** to **r15** are used as temporary registers (caller save).
- **r16** to **r19** are saved registers (callee save).
- **r20** to **r31** are more temporary registers (caller save).
- **r32** to **r38** are reserved for the kernel.
- Additional registers are used for global data pointers, stack pointers, and frame pointers.
Recap: Conventions so far

- first four arg words passed in $a0, $a1, $a2, $a3
- remaining arg words passed in parent’s stack frame
- return value (if any) in $v0, $v1
- stack frame at $sp
  - contains $ra (clobbered on JAL to sub-functions)
  - contains local vars (possibly clobbered by sub-functions)
  - contains extra arguments to sub-functions
  - contains space for first 4 arguments to sub-functions
- callee save regs are preserved
- caller save regs are not
- Global data accessed via $gp