CS 316:
Procedure Calls

Kavita Bala

Fall 2007

Computer Science

Cornell University

Announcements

- PA 2 due tonight
- PA 3 will be out later this week (Thu/Fri)
  - Due on the Friday after Fall break

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

Take 1: Use Jumps

```
main:
  j mult
Laftercall1:
  add $1,$2,$3

  j mult
Laftercall2:
  sub $3,$4,$5
```

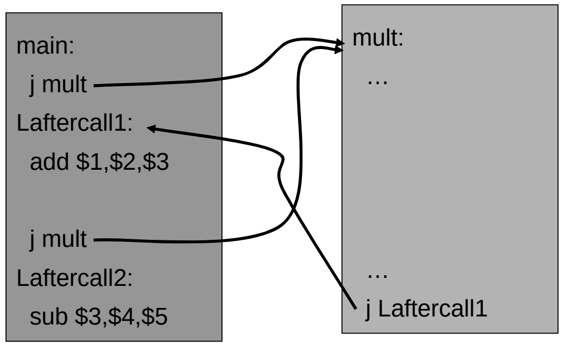
```
mult:
  …



  …
  j Laftercall1
```

- Jumps and branches can transfer control to the callee (called procedure)
- Jumps and branches can transfer control back

Take 1: Use Jumps



- Jumps and branches can transfer control to the callee
- 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

Take 2: JAL/JR

```
main:
  jal mult
Laftercall1:
  add $1,$2,$3

  jal mult
Laftercall2:
  sub $3,$4,$5
```

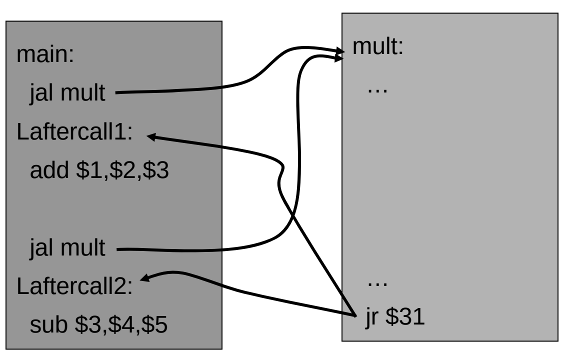
```
mult:
  …



  …
  jr $31
```

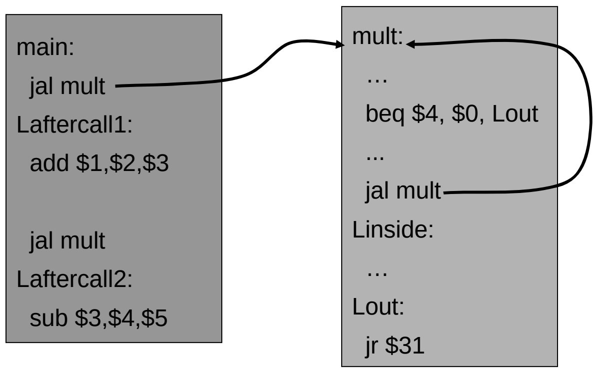
- JAL saves the PC in register $31
- Subroutine returns by jumping to $31

Take 2: JAL/JR



- JAL saves the PC in register $31
- Subroutine returns by jumping to $31
- What happens for recursive invocations?

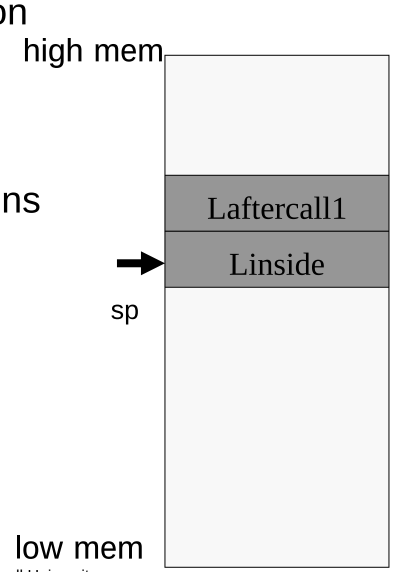
Take 2: JAL/JR



- Recursion overwrites contents of $31
- Need to save and restore the register contents

Call Stacks

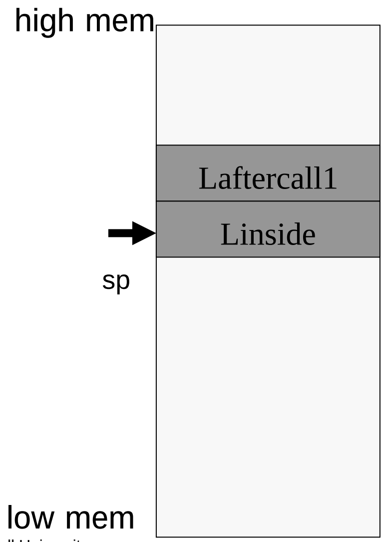
- A call stack contains activation records (aka stack frames)



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

Call Stacks

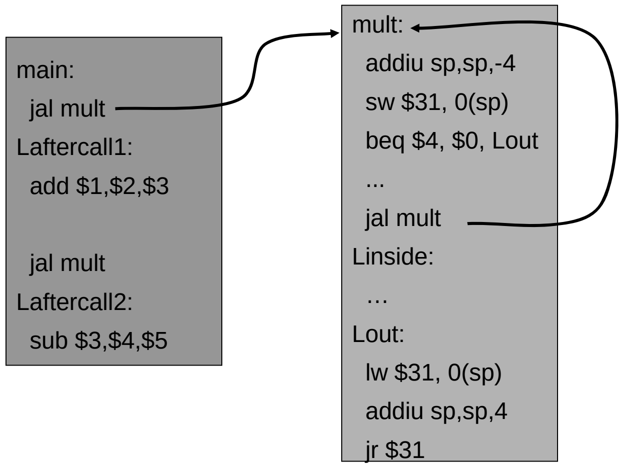
- A stack pointer (sp) keeps track of the top of the stack
  - dedicated register ($29) on the MIPS
- Manipulated by push/pop operations
  - push: move sp down, store
  - pop: load, move sp up



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

Take 3: JAL/JR with Activation Records



- Stack used to save and restore contents of $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
  - `int min(int a, int b);`
  - `int subf(int a, int b, int c, int d, int e);`
  - `int isalpha(char c);`
  - `int treesort(struct Tree *root);`
  - `struct Node *createNode();`
  - `struct Node mynode();`
- Too many combinations of char, short, int, void *, struct, etc.
  - MIPS treats char, short, int and void * identically

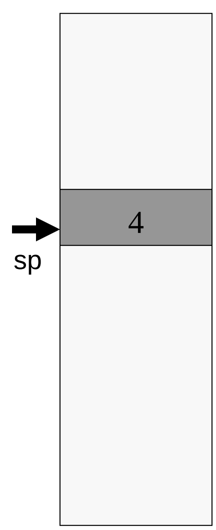
Simple Argument Passing

```
main:
  li a0, 6
  li a1, 7
  jal min
  // result in v0
```

- 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

Many Arguments

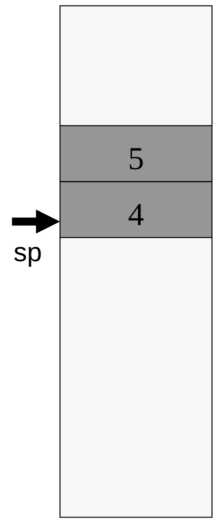
```
main:
  li a0, 0
  li a1, 1
  li a2, 2
  li a3, 3
  li $8, 4
  addiu sp,sp,-4
  sw $8, 0(sp)
  jal subf
  // result in v0
```



- What if there are more than 4 arguments?
- Use the stack for the additional arguments
  - "spill"

Many Arguments

```
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
```



- What if there are more than 4 arguments?
- Use the stack for the additional arguments
  - "spill"

Variable Length Arguments

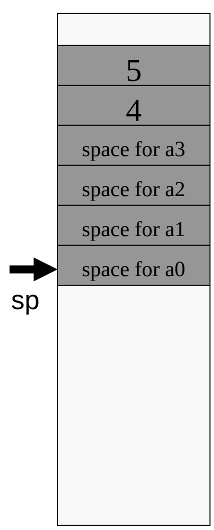
- `printf("Coordinates are: %d %d %d\n", 1, 2, 3);`
- Could just use the regular calling convention, placing first four arguments in registers, spilling the rest onto the stack
  - Callee requires special-case code
  - if(argno == 1) use a0, … else if (argno == 4) use a3, else use stack offset

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

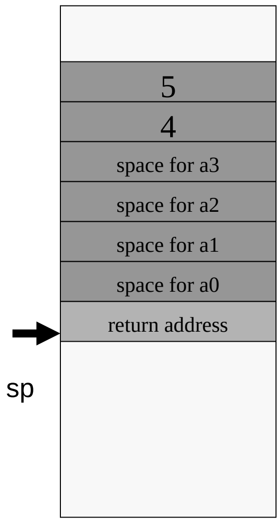
```
main:
  li a0, 0
  li a1, 1
  li a2, 2
  li a3, 3
  addiu sp,sp,-24
  li $8, 4
  sw $8, 16(sp)
  li $8, 5
  sw $8, 20(sp)
  jal subf
  // result in v0
```



- 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

Kavita Bala, Computer Science, Cornell University

Frame Layout on Stack




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


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);`

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 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