10/6: Lecture Topics

- Procedure call
  - Calling conventions
  - The stack
  - Preservation conventions
  - Nested procedure call

Calling Conventions

- Sequence of steps to follow when calling a procedure
- Makes sure:
  - arguments are passed in
  - flow of control from *caller* to *callee* and back
  - return values passed back out
  - no unexpected side effects

Calling Conventions

- Mostly governed by the compiler
- We'll see a MIPS calling convention
  - Not the only way to do it, even on MIPS
  - Most important: be consistent
- Procedure call is one of the most unpleasant things about writing assembly for RISC architectures

A MIPS Calling Convention

1. Place parameters where the procedure can get them
2. Transfer control to the procedure
3. Get the storage needed for the procedure
4. Do the work
5. Place the return value where the calling code can get it
6. Return control to the point of origin

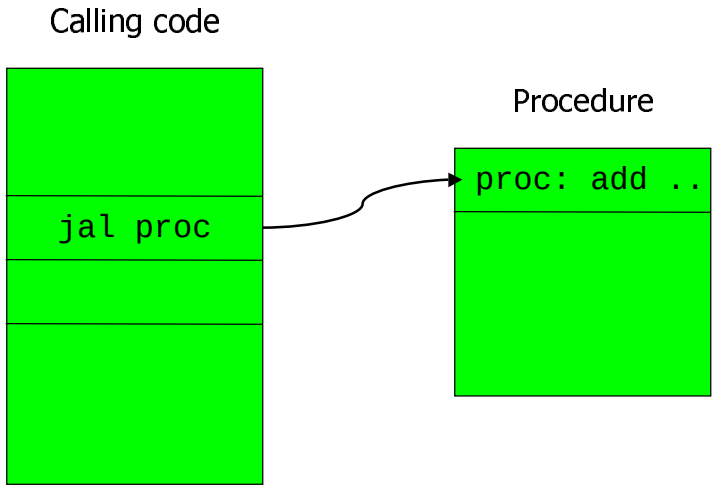
Step 1: Parameter Passing

- The first four parameters are easy - use registers $a0, $a1, $a2, and $a3
- You've seen this already
- What if there are more than four parameters?

Step 2: Transfer Control

- Getting from caller to callee is easy -- just jump to the address of the procedure
- Need to leave a way to get back again
- Special register: $ra (for *return address*)
- Special instruction: `jal`

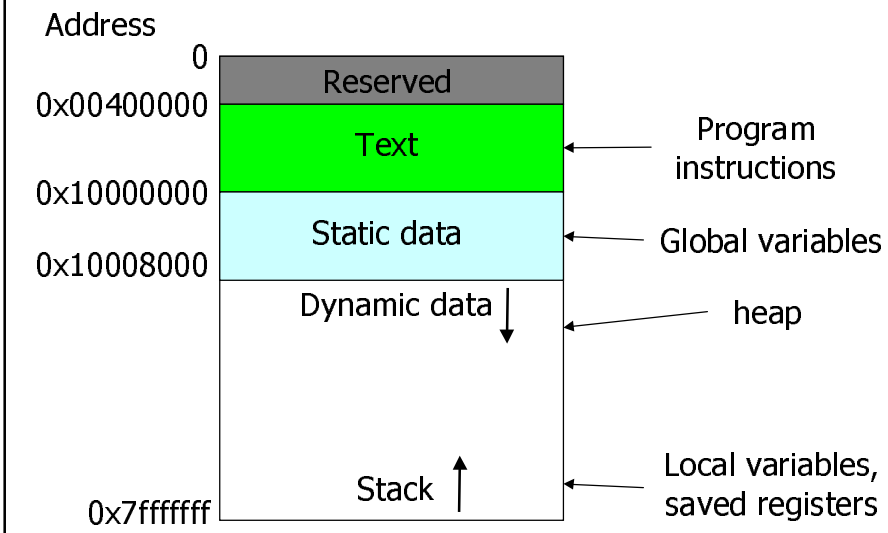
Jump and Link



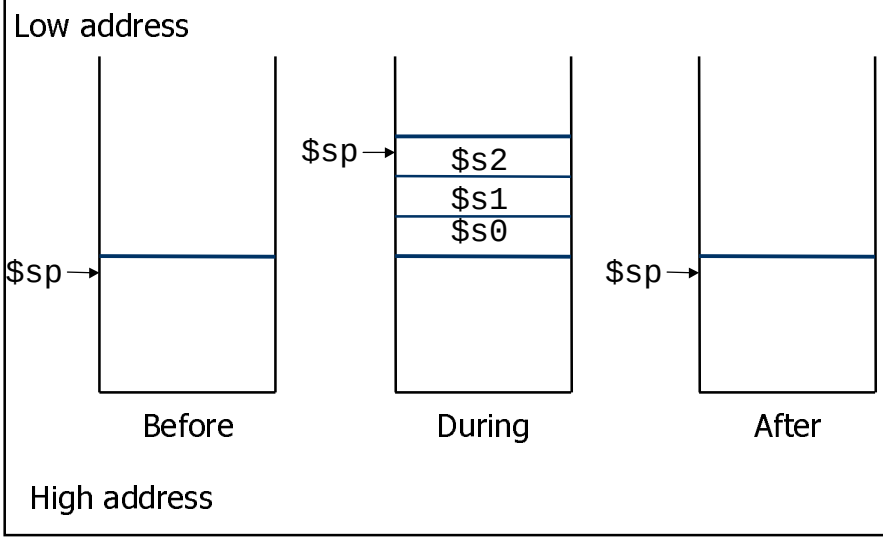
Step 3: Acquire Storage

- What storage do we need?
  - Registers
  - Other local variables
- Where do we get the storage?
  - From the stack

Refining Program Layout



Saving Registers on the Stack



Assembly for Saving Registers

- We want to save $s0, $s1, and $s2 on the stack

```
sub $sp, $sp, 12  # make room for 3 words
sw  $s0,          # store $s0
sw  $s1,          # store $s1
sw  $s2,          # store $s2
```

Step 4: Do the work

- We called the procedure so that it could do some work for us
- Now is the time for it to do that work
- Resources available:
  - Registers freed up by Step 3
  - All temporary registers ($t0-$t9)

Callee-saved vs. Caller-saved

- Some registers are the responsibility of the callee
  - *callee-saved* registers
  - $s0-$s7
- Other registers are the responsibility of the caller
  - *caller-saved* registers
  - $t0-$t9

Step 5: Return values

- MIPS allows for two return values
- Place the results in $v0 and $v1
- You've seen this too
- What if there are more than two return values?

Step 6: Return control

- Because we laid the groundwork in step 2, this is easy
- Address of the point of origin + 4 is in register $ra
- Just use `jr $ra` to return

An Example

```
int leaf(int g, int h, int i, int j) {
    int f;

    f = (g + h) - (i + j);
    return f;
}
```

Let g, h, `i`, `j` be passed in `$a0`, `$a1`, `$a2`, `$a3`, respectively

Let the local variable `f` be stored in `$s0`

Compiling the Example

```
leaf: sub $sp, $sp, 4   # room for 1 word
      sw  $s0, 0($sp)   # store $s0
      add $t0, $a0, $a1 # $t0 = g + h
      add $t1, $a2, $a3 # $t1 = i + j
      sub $s0, $t0, $t1 # $s0 = f
      add $v0, $s0, $zero  # copy result
      lw  $s0, 0($sp)   # restore $s0
      add $sp, $sp, 4   # put $sp back
      jr  $ra           # jump back
```

Nested Procedures

- Suppose we have code like this:
- Potential problem: the return address gets overwritten

```
main() {
  foo();
}

int foo() {
  return bar();
}

int bar() {
  return 6;
}
```

A Trail of Bread Crumbs

- The registers $s0-$s7 are not the only ones we save on the stack
- What can the caller expect to have preserved across procedure calls?
- What can the caller expect to have overwritten during procedure calls?

Preservation Conventions

Preserved	Not Preserved
Saved registers: $s0-$s7	Temporary registers: $t0-$t9
Stack pointer register: $sp	Argument registers: $a0-$a3
Return address register: $ra	Return value registers: $v0-$v1
Stack above the stack pointer	Stack below the stack pointer

A Brainteaser in C

- What does this program print? Why?

```
#include <stdio.h>

int* foo() {
  int b = 6;
  return &b;
}

void bar() {
  int c = 7;
}

main() {
  int *a = foo();
  bar();
  printf("The value at a is %d\n", *a);
}
```