CSE 31
Computer Organization

Lecture 17 – MIPS Procedures

Announcements

- Labs
  - Lab 5 grace period ends this week
    - » No penalty for submission during grace period
    - » Demo is REQUIRED to receive full credit
  - Lab 6 due this week (with **14 days grace period** after due date)
    - » Demo is REQUIRED to receive full credit
  - Lab 7 out this week
    - » Due at 11:59pm on the same day of your lab after next (with 7 days grace period after due date)
    - » You must demo your submission to your TA within 21 days from posting of lab
    - » Demo is REQUIRED to receive full credit
- Reading assignments
  - Reading 05 (zyBooks 1.6 – 1.7, 6.1 – 6.3) due 03-APR
    - » Complete **Participation Activities** in each section to receive grade
    - » IMPORTANT: Make sure to submit score to CatCourses by using the link provided on CatCourses

Announcements

- Homework assignment
  - Homework 04 (zyBooks 4.1 – 4.9) due 03-APR
    - » Complete **Challenge Activities** in each section to receive grade
    - » IMPORTANT: Make sure to submit score to CatCourses by using the link provided on CatCourses
- Project 02
  - Due 05-MAY
  - Can work in teams of 2 students
    - » Each team member must identify teammate in “Comments…” text-box at the submission page
    - » If working in teams, each student must submit code (can be the same as teammate) and demo individually
    - » Grade can vary among teammates depending on demo
  - Demo required for project grade
    - » No partial credit for submission without demo
  - **No grace period**
    - » **Must complete submission and demo by due date.**

C functions

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

What information must compiler/programmer keep track of?

Flow of the program

```
/* really dumb mult function */
int mult (int mcand, int mlier){
  int product = 0;
  while (mlier > 0)  {
    product = product + mcand;
    mlier = mlier - 1; }
  return product;
}
```

What instructions can accomplish this?

Jump instructions

Function Call Bookkeeping

- Registers play a major role in keeping track of information for function calls.

- Register conventions:
  - Return address `$ra`
  - Arguments `$a0, $a1, $a2, $a3`
  - Return value `$v0, $v1`
  - Local variables `$s0, $s1, … , $s7`

- The stack is also used; more later.

Instruction Support for Functions (1/6)

```
... sum(a, b);... /* a, b: $s0, $s1 */

int sum(int x, int y) {
    return x + y;
}
```

C

MIPS

address (shown in Hexadecimal)

```
1000
1004
1008
100c
1010
…
2000
2004
```

In MIPS, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.

Instruction Support for Functions (2/6)

```
... sum(a, b);... /* a, b: $s0, $s1 */

int sum(int x, int y) {
   return x + y;
}
```

C

MIPS

address (shown in Hexadecimal)

```
1000 add  $a0,$s0,$zero      # x = a
1004 add  $a1,$s1,$zero      # y = b
1008 addi $ra,$zero,1010     # $ra = 1010
100c j    sum                # jump to sum
1010
...
2000 sum: add $v0,$a0,$a1
2004 jr   $ra             # new instruction
```

Return address

Instruction Support for Functions (3/6)

```
... sum(a, b);...  /* a, b: $s0, $s1 */

int sum(int x, int y) {
   return x + y;
}
```

C

MIPS

- Question: Why use jr here? Why not use j 1010?
- Answer: sum might be called at many places, so we can't return to a fixed place. The calling proc to sum must be able to say "return here" somehow.

```
…
2000 sum: add $v0,$a0,$a1
2004 jr   $ra              # new instruction
```

Instruction Support for Functions (4/6)

- Before:

```
1008 addi $ra,$zero,1010   # $ra = 1010
100c j sum                 # goto sum
```

- Single instruction to jump and save return address: jump and link (`jal`)

- After:

```
1008 jal sum  # $ra = 101c, goto sum
```

- Why have a **`jal`**?
  - Make the common case fast: function calls very common.
  - Don't have to know where code is in memory with **`jal`**!

Instruction Support for Functions (5/6)

- Syntax for **`jal`** (jump and link) is same as for **`j`** (jump):

  `jal label`

- **`jal`** should really be called **`laj`** for "link and jump":
  - Step 1 (link): Save address of next instruction into `$ra`
    - » Why next instruction? Why not current one?
  - Step 2 (jump): Jump to the given label

Instruction Support for Functions (6/6)

- Syntax for **`jr`** (jump register):

  `jr register`

- Instead of providing a label to jump to, the **`jr`** instruction provides a register which contains an address to jump to.

- Very useful for function calls:
  - **`jal`** stores return address in register (`$ra`)
  - **`jr $ra`** jumps back to that address

Nested Procedures (1/2)

```
int foo(int x, int y) {
   return bar(x, x + y) + y;
}
```

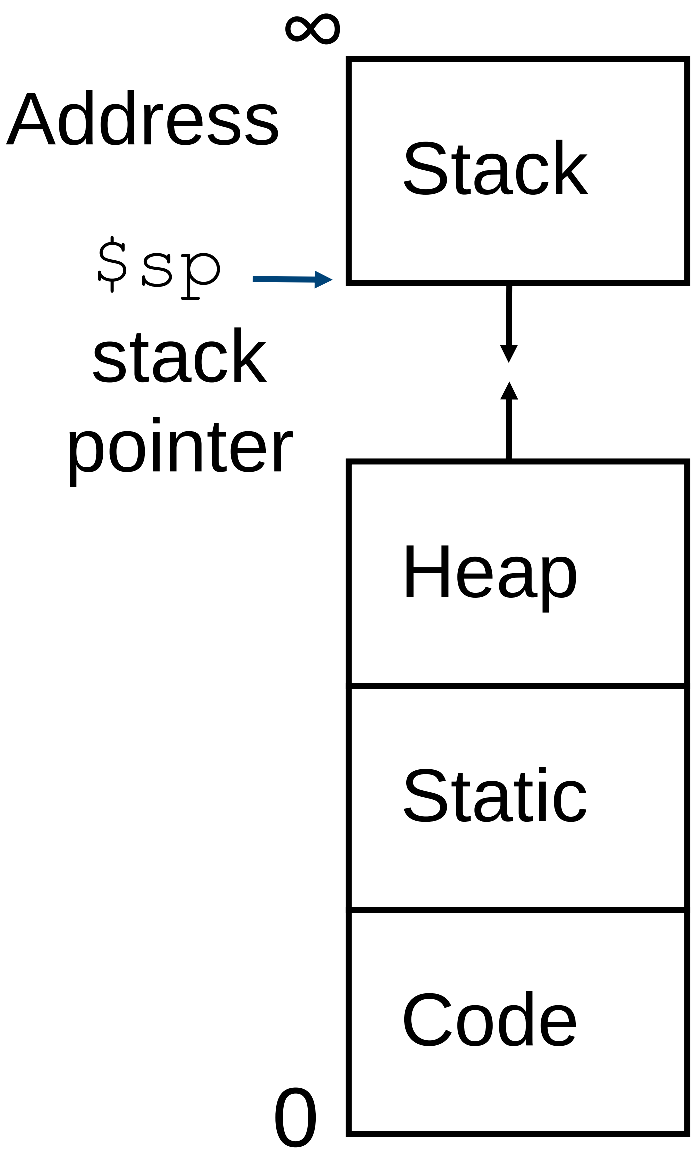
- Something called **foo**, now **foo** is calling **bar**.
- So there's a value in `$ra` that **foo** wants to jump back to, but this will be overwritten by the call to **bar**.
- Need to save **foo** return address before call to **bar**.
  - How to prevent the return address from being over-written?

Nested Procedures (2/2)

- In general, may need to save some other info in addition to `$ra`.

- When a C program is run, there are 3 important memory areas allocated:
  - **Static**: Variables declared once per program, cease to exist only after execution completes. E.g., C globals
  - **Heap**: Variables declared dynamically via `malloc`
  - **Stack**: Space to be used by procedure during execution; this is where we can save register values

C memory Allocation review



Space for saved procedure/function information

Explicitly created space, i.e., `malloc()`

Variables declared once per program; e.g., globals

Program

Using the Stack (1/2)

- We have a register `$sp` which always points to the last used space in the stack (top of stack).
- To use stack, we decrement this pointer by the amount of space we need and then fill it with info.
- So, how do we compile this?

```
int foo(int x, int y) {
    return bar(x, x + y) + y;
}
```

Using the Stack (2/2)

```
           $a0         $a1
int foo(int x, int y) {
    return bar(x, x + y) + y; }
                $a0     $a1
```

- Hand-compile

```
foo:
         addi $sp,$sp,-8     # space on stack
"push"   sw $ra, 4($sp)      # save ret addr
         sw $a1, 0($sp)      # save y
         add $a1,$a0,$a1     # bar(x, x + y)
         jal bar             # call bar
         lw $a1, 0($sp)      # restore y
"pop"    add $v0,$v0,$a1     # bar()+y
         lw $ra, 4($sp)      # get ret addr
         addi $sp,$sp,8      # restore stack
         jr $ra
bar: ...
```

Steps for Making a Procedure Call

1. Save necessary values onto stack.

2. Assign new argument(s), if any.

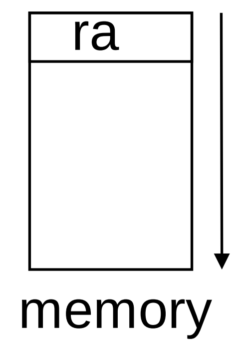
3. **`jal`** call

4. Restore values from stack.

Basic Structure of a Function

Prologue

```
entry_label:
addi $sp, $sp, -framesize
sw $ra, framesize-4($sp)   # save $ra
save other regs if need be
```



Body ...   (call other functions…)

Epilogue

```
restore other regs if need be
lw $ra, framesize-4($sp)   # restore $ra
addi $sp, $sp, framesize
jr $ra
```

Rules for Procedures

- Called with a `jal` instruction
  - returns with a `jr $ra`
- Accepts up to 4 arguments in `$a0`, `$a1`, `$a2` and `$a3`
- Return value is always in `$v0` (and if necessary, in `$v1`)
- Must follow register conventions
  - Covered later

MIPS Registers

The constant 0	\$0	\$zero
Reserved for Assembler	\$1	\$at
Return Values	\$2-\$3	\$v0-\$v1
Arguments	\$4-\$7	\$a0-\$a3
Temporary	\$8-\$15	\$t0-\$t7
Saved	\$16-\$23	\$s0-\$s7
More Temporary	\$24-\$25	\$t8-\$t9
Used by Kernel	\$26-27	\$k0-\$k1
Global Pointer	\$28	\$gp
Stack Pointer	\$29	\$sp
Frame Pointer	\$30	\$fp
Return Address	\$31	\$ra

Use names for registers -- code is clearer!

Other Registers

- **`$at`**: may be used by the assembler at any time; unsafe to use

- **`$k0-$k1`**: may be used by the OS at any time; unsafe to use

- **`$gp`**, **`$fp`**: don't worry about them