CSE 31
Computer Organization

Lecture 10 – MIPS: Procedures

Announcement

- Project #1
  - Due at 11:59pm on 3/18, Monday
  - You must demo your submission to your TA within 7 days after due date
- Lab #5
- HW #2 in CatCourses
  - Due Monday (3/11) at 11:59pm
- Reading assignment
  - Chapter 3.1 – 3.7, 3.9 of zyBooks
    - Make sure to do the Participation Activities
    - Due Wednesday (3/13) at 11:59pm

Announcement

- Midterm exam Wednesday (3/6, not 2/27)
  - Lectures 1 – 7
  - Lab #1 - #4
  - HW #1
  - Closed book
  - 1 sheet of note (8.5” x 11”), both sides
  - Sample exam in CatCourses

Review

- To help the conditional branches make decisions concerning inequalities, we introduce: “Set on Less Than” called
  `slt`, **`slti`**, **`sltu`**, **`sltiu`**
- One can store and load (signed and unsigned) bytes as well as words with `lb`, `lbu`
- Unsigned add/sub don’t cause overflow
- New MIPS Instructions:

```
sll, srl, lb, lbu
slt, slti, sltu, sltiu
addu, addiu, subu
```

Example: The C Switch Statement (1/3)

- Choose among four alternatives depending on whether `k` has the value 0, 1, 2 or 3. Compile this C code:

```
switch (k) {
 case 0: f=i+j; break; /* k=0 */
 case 1: f=g+h; break; /* k=1 */
 case 2: f=g-h; break; /* k=2 */
 case 3: f=i-j; break; /* k=3 */
}
```

Example: The C Switch Statement (2/3)

- This is complicated, so simplify.
- Rewrite it as a chain of if-else statements, which we already know how to compile:

```
if(k==0) f=i+j;
   else if(k==1) f=g+h;
      else if(k==2) f=g-h;
         else if(k==3) f=i-j;
```

- Use this mapping:

```
f:$s0, g:$s1, h:$s2,
i:$s3, j:$s4, k:$s5
```

Example: The C Switch Statement (3/3)

- Final compiled MIPS code:

```
      bne  $s5,$0,L1       # branch k!=0
      add  $s0,$s3,$s4     #k==0 so f=i+j
      j    Exit            # end of case so Exit
L1:   addi $t0,$s5,-1      # $t0=k-1
      bne  $t0,$0,L2       # branch k!=1
      add  $s0,$s1,$s2     #k==1 so f=g+h
      j    Exit            # end of case so Exit
L2:   addi $t0,$s5,-2      # $t0=k-2
      bne  $t0,$0,L3       # branch k!=2
      sub  $s0,$s1,$s2     #k==2 so f=g-h
      j    Exit            # end of case so Exit
L3:   addi $t0,$s5,-3      # $t0=k-3
      bne  $t0,$0,Exit     # branch k!=3
      sub  $s0,$s3,$s4     # k==3 so f=i-j
Exit:
```

Always compared with $0!

Quiz

```
Loop:addi $s0,$s0,-1    # i = i - 1
     slti $t0,$s1,2     # $t0 = (j < 2)
     beq  $t0,$0 ,Loop  # goto Loop if $t0 == 0
     slt  $t0,$s1,$s0   # $t0 = (j < i)
     bne  $t0,$0 ,Loop  # goto Loop if $t0 != 0
```

($s0=i, $s1=j)

What C code properly fills in the blank in loop below?

do {i--;} while(__);

```
1)  j < 2 && j < i
2)  j ≥ 2 && j < i
3)  j < 2 && j ≥ i
4)  j ≥ 2 && j ≥ i
5)  j > 2 && j < i
6)  j < 2 || j < i
7)  j ≥ 2 || j < i
8)  j < 2 || j ≥ i
9)  j ≥ 2 || j ≥ i
10) j > 2 || j < i
```

Quiz

```
Loop:addi $s0,$s0,-1    # i = i - 1
     slti $t0,$s1,2     # $t0 = (j < 2)
     beq  $t0,$0 ,Loop  # goto Loop if $t0 == 0
     slt  $t0,$s1,$s0   # $t0 = (j < i)
     bne  $t0,$0 ,Loop  # goto Loop if $t0 != 0
```

($s0=i, $s1=j)

What C code properly fills in the blank in loop below?

do {i--;} while(__);

1) j < 2 && j < i
2) j ≥ 2 && j < i
3) j < 2 && j ≥ i
4) j ≥ 2 && j ≥ i
5) j > 2 && j < i
6) j < 2 || j < i
7) j ≥ 2 || j < i
8) j < 2 || j ≥ i
9) j ≥ 2 || j ≥ i
10) j > 2 || j < i

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 */
```

C

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

---

address (shown in Hexadecimal)

MIPS

```
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)

C

```
... sum(a,b);...    /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

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)

C

```
... sum(a,b);...    /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

M
I
P
S

- Question: Why use `jr` here? Why not use `j`?
- Answer: `sum` might be called by 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:

```
100c jal sum  # $ra=1010,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)

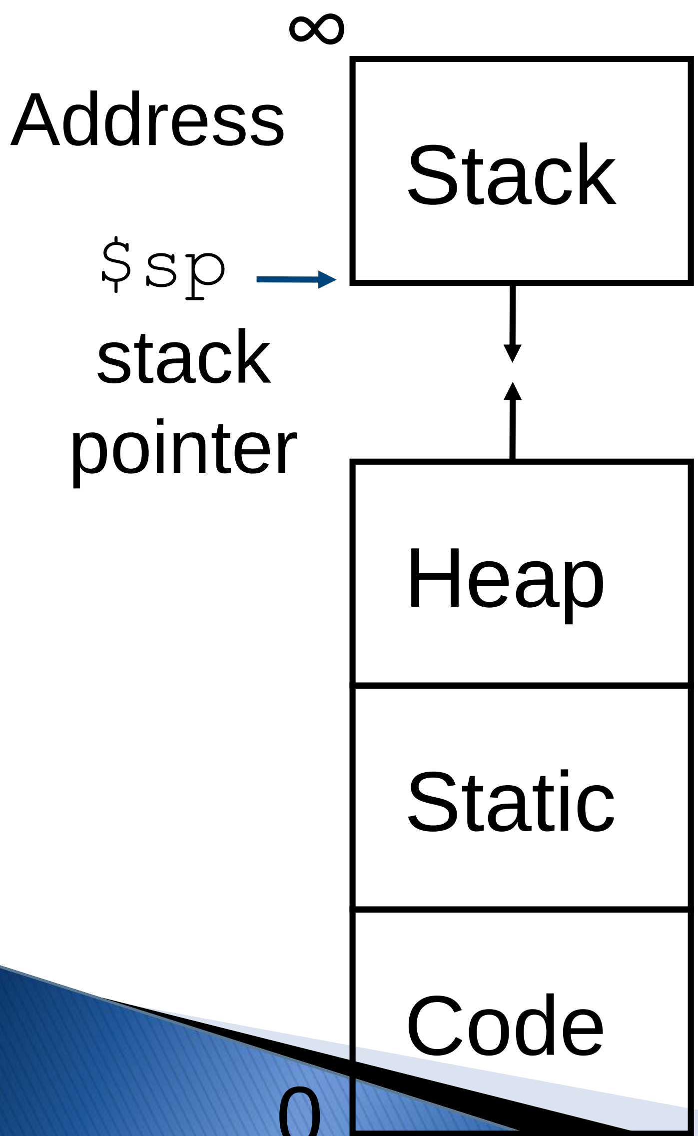
```
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}
```

- Something called **`sumSquare`**, now **`numSquare`** is calling **`mult`**.
- So there's a value in `$ra` that **`sumSquare`** wants to jump back to, but this will be overwritten by the call to **`mult`**.
- Need to save **`sumSquare`** return address before call to **`mult`**.
  - 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 sumSquare(int x, int y) {
    return mult(x,x)+ y;
}
```

Using the Stack (2/2)

- Hand-compile

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

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

Steps for Making a Procedure Call

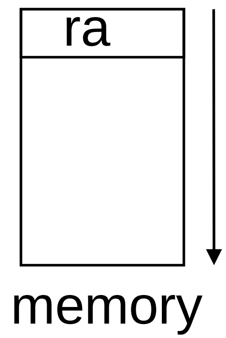
1. Save necessary values onto stack.
2. Assign argument(s), if any.
3. `jal` call
4. Restore values from stack.

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
  - What are they?

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