CSE 31
Computer Organization

Lecture 11 – MIPS: Procedures (2)

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 #6 this week
  - Due at 11:59pm on the same day of your next lab
  - You must demo your submission to your TA within 14 days
- HW #2 in CatCourses
  - Due Monday (3/11) at 11:59pm
- HW #3 in CatCourses
  - Due Monday (3/18) at 11:59pm
- Reading assignment
  - Chapter 4.1 – 4.9of zyBooks
    - Make sure to do the Participation Activities
    - Due Wednesday (3/20) at 11:59pm

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

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; }
```

“push”

“pop”

```
sumSquare:
        addi $sp,$sp,-8       # space on stack
        sw $ra, 4($sp)        # save ret addr
        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
        lw $ra, 4($sp)        # get ret addr
        addi $sp,$sp,8        # restore stack
        jr $ra
mult: ...
```

Steps for Making a Procedure Call

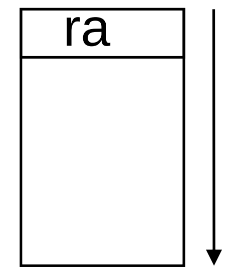
1. Save necessary values onto stack.
2. Assign argument(s), if any.
3. `jal` call
4. Restore values from stack.

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…)



memory

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 <u>names</u> 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

Register Conventions (1/4)

- CalleR: the calling function (where you call a function)
- CalleE: the function being called
- When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.
- Register Conventions: A set of generally accepted rules as to which registers will be unchanged after a procedure call (`jal`) and which may be changed.

Register Conventions (2/4) – saved

- **`$0`**: No Change. Always `0`.
- **`$s0`-`$s7`**: Restore if you change. Very important, that's why they're called saved registers. If the callee changes these in any way, it must restore the original values before returning. **`$sp`**: Restore if you change. The stack pointer must point to the same place before and after the `jal` call, or else the caller won't be able to restore values from the stack.
- HINT -- All saved registers start with S!

It's callee's job to restore!

Register Conventions (3/4) – volatile

- **$ra**: Can Change.
  - The jal call itself will change this register. <u>Caller</u> needs to save on stack before next call (nested call).
- **$v0-$v1**: Can Change.
  - These will contain the new returned values.
- **$a0-$a3**: Can change.
  - These are volatile argument registers. <u>Caller</u> needs to save if they are needed after the call.
- **$t0-$t9**: Can change.
  - That's why they're called temporary: any procedure may change them at any time. <u>Caller</u> needs to save if they'll need them afterwards.

It's caller's job to backup!

Register Conventions (4/4)

- What do these conventions mean?
  - If function **R** calls function **E**, then function **R** must save any **V (volatile) registers** that it may be using onto the stack **before making** a **`jal`** call.
  - Function **E** must save any **S (saved) registers** it intends to use before garbling up their values. It must restore any modified S registers **before returning** back to **R**
- Remember: caller/callee need to save only volatile/saved registers they are using, not all registers.

Example: Fibonacci Numbers 1/7

- The Fibonacci numbers are defined as follows: F(n) = F(n – 1) + F(n – 2),
  F(0) and F(1) are defined to be 1
- In C we have:

```
int fib(int n) {
  if(n == 0)
    return 1;
  if(n == 1)
    return 1;
  return (fib(n - 1) + fib(n - 2));
```

Example: Fibonacci Numbers 2/7

- Now, let's translate this to MIPS!
- You will need space for **three words** on the stack
- The function will use one `$s` register, `$s0`
- Write the Prologue:

```
fib:
addi $sp, $sp, -12  # Space for 3 words
sw $ra, 8($sp)      # Save return address
sw $s0, 4($sp)      # Save s0
```

Example: Fibonacci Numbers 3/7

- Now write the Epilogue:

```
fin:
lw $s0, 4($sp)          # Restore $s0
lw $ra, 8($sp)          # Restore return address
addi $sp, $sp, 12       # Pop the stack frame
jr $ra                  # Return to caller
```

Example: Fibonacci Numbers 4/7

```
int fib(int n) {
  if(n == 0)      /* Base case 0*/
     return 1;
  if(n == 1)      /* Base case 1 */
     return 1;
  return (fib(n - 1) + fib(n - 2));
```

```
addi $v0, $zero, 1        # $v0 = 1
beq  $a0, $zero, fin      # Base case 0
addi $t0, $zero, 1        # $t0 = 1
beq  $a0, $t0, fin        # Base case 1
```

Continued on next slide. . .

Example: Fibonacci Numbers 5/7

```
int fib(int n) {
  if(n == 0)     /* Base case 0*/
     return 1;
  if(n == 1)     /* Base case 1 */
     return 1;
  return (fib(n - 1) + fib(n - 2));
```

Write fib(n-1):

```
addi $a0, $a0, -1        # $a0 = n - 1
sw $a0, 0($sp)           #  Need $a0 after jal
jal fib                  #  fib(n - 1)
```

Example: Fibonacci Numbers 6/7

```
int fib(int n) {
  if(n == 0)     /* Base case 0*/
     return 1;
  if(n == 1)     /* Base case 1 */
     return 1;
  return (fib(n - 1) + fib(n - 2));
```

Write fib(n-2) and + :

```
lw $t0, 0($sp)          # restore $a0
addi $a0, $t0, -1       # $a0 = n - 2
add $s0, $v0, $zero     # Place fib(n – 1) somewhere
jal fib                 # fib(n - 2)
add $v0, $v0, $s0       # $v0 = fib(n-1) + fib(n-2)
```

Example: Fibonacci Numbers 7/7

- Here's the complete code for reference:

```
fib:  addi $sp, $sp, -12
      sw $ra, 8($sp)
      sw $s0, 4($sp)
      addi $v0, $zero, 1
      beq $a0, $zero, fin
      addi $t0, $zero, 1
      beq $a0, $t0, fin
      addi $a0, $a0, -1
      sw $a0, 0($sp)
      jal fib
```

```
      lw $a0, 0($sp)
      addi $a0, $a0, -1
      add $s0, $v0, $zero
      jal fib
      add $v0, $v0, $s0
fin:  lw $s0, 4($sp)
      lw $ra, 8($sp)
      addi $sp, $sp, 12
      jr $ra
```

Quiz

```
int fact(int n){
     if(n == 0) return 1; else return(n*fact(n-
1));
}
```

When translating this to MIPS…

1) We COULD copy `$a0` to `$a1` (and then not store `$a0` or `$a1` on the stack) to store `n` across recursive calls.

2) We MUST save `$a0` on the stack since it gets changed.

3) We MUST save `$ra` on the stack since we need to know where to return to…

	123
a)	FFF
b)	FFT
c)	FTF
d)	FTT
e)	TFF
f)	TFT
g)	TTF
h)	TTT

Quiz

```
int fact(int n){
    if(n == 0) return 1; else return(n*fact(n-1));
}
```

When translating this to MIPS…

1) We COULD copy `$a0` to `$a1` (and then not store `$a0` or `$a1` on the stack) to store `n` across recursive calls.

2) We MUST save `$a0` on the stack since it gets changed.

3) We MUST save `$ra` on the stack since we need to know where to return to…

	123
a)	**FFF**
b)	FFT
c)	FTF
d)	FTT
e)	TFF
f)	TFT
g)	TTF
h)	TTT

We can implement it using iterations

Summary

- Functions called with `jal`, return with `jr $ra`.
- The stack is your friend: Use it to save anything you need. Just leave it the way you found it!
- Instructions we know so far…

  Arithmetic: `add, addi, sub, addu, addiu, subu`

  Memory: `lw, sw, lb, sb, lbu`

  Decision: `beq, bne, slt, slti, sltu, sltiu`

  Unconditional Branches (Jumps): `j, jal, jr`
- Registers we know so far
  - All of them!
  - There are CONVENTIONS when calling procedures!