CSE 31
Computer Organization

Lecture 18 – MIPS Procedures (wrap up)

Announcements

- Labs
  - Lab 6 grace period ends this week
    - » No penalty for submission during grace period
    - » Demo is REQUIRED to receive full credit
  - Lab 7 due this week (with **7 days grace period** after due date)
    - » Demo is REQUIRED to receive full credit
  - Lab 8 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 14 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 **tonight**, 03-APR and Reading 06 (zyBooks 6.4 – 6.7) due 10-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

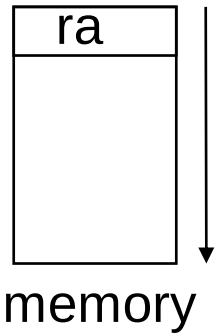
- Homework assignment
  - Homework 04 (zyBooks 4.1 – 4.9) due **tonight**, 03-APR and Homework 05 (zyBooks 1.6 – 1.7) due 10-APR
    - » Complete **Challenge Activities** in each section to receive grade
    - » IMPORTANT: Make sure to submit score to CatCourses by using the link provided on CatCourses
- 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.**

Basic Structure of a Function (review)

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 (review)

- 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

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. Caller 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. Caller 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. Caller 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.

Register Conventions Summary

- 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.
  - Volatile registers: `$ra`, `$v0-$v1`, `$a0-$a3`, `$t0-$t9`

- 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**
  - Saved registers: `$0`, `$s0-$s7`, `$sp`

- 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 $a0, 0($sp)          # restore $a0 (= n – 1)
addi $a0, $a0, -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…

We can implement it using iterations

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

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!