Subroutines and Control Abstraction

ICOM 4036

Lecture 8

Implementing Procedures

- Why procedures?
  - Abstraction
  - Modularity
  - Code re-use
- Initial Goal
  - Write segments of assembly code that can be re-used, or “called” from different points in the main program.
  - KISS: Keep It Simple Stupid:
    - no parameters, no recursion, no locals, no return values

Procedure Linkage Approach I

- Problem
  - procedure must determine where to return after servicing the call
- Solution: Architecture Support
  - Add a jump instruction that saves the return address in some place known to callee
    - MIPS: jal instruction saves return address in register $ra
  - Add an instruction that can jump to return address
    - MIPS: jr instruction jumps to the address contained in its argument register

Computing Integer Division (Procedure Version)

Iterative C++ Version

```
int a = 0;
int b = 0;
int res = 0;
main () {
  a = 12;
  b = 5;
  res = 0;
  div();
  printf(“Res = %c
}
void div(void) {
  while (a >= b) {
    a = a - b;
    res ++;
  }
}
```

C++

```
# div function
# PROBLEM: Must save args and registers before using them
d:                                              # void d(void) {
                                                #   // Allocate registers for globals
            la          $s0, x                  #   // x in $s1
            lw          $s1, 0($s0)
            la          $s0, y                  #   // y in $s2
            lw          $s2, 0($s0)
            la          $s0, res                #   // res in $s3
            lw          $s3, 0($s0)
while:      bgt         $s2, $s1, ewhile        #   while (x <= y) {
            sub         $s1, $s1, $s2           #     x = x - y
            addi        $s3, $s3, 1             #     res ++
            j           while                   #   }
ewhile:                                         #   // Update variables in memory
            la          $s0, x
            sw          $s1, 0($s0)
            la          $s0, y
            sw          $s2, 0($s0)
            la          $s0, res
            sw          $s3, 0($s0)
enddiv:     jr          $ra                     #   return;
                                                # }
```

MIPS
Assembly Language

Computing Integer Division (Procedure Version)

Iterative C++ Version

```
int a = 0;
int b = 0;
int res = 0;
main () {
  a = 12;
  b = 5;
  res = 0;
  div();
  printf(“Res
}
void div(voic
  while (a >=
    a = a - b
    res ++;
  }
}
```

C++

MIPS
Assembly Language

```
            .data
x:          .word   0
y:          .word   0
res:        .word   0
pf1:        .asciiz "Result = "
pf2:        .asciiz "Remainder = "
            .globl      main
            .text
main:                                           # int main() {
                                                #    // ... assumes registers sx unused
            la          $s0, x
            li          $s1, 12
            sw          $s1, 0($s0)
            la          $s0, y
            li          $s2, 5
            sw          $s2, 0($s0)
            la          $s0, res                #    res = 0;
            li          $s3, 0
            sw          $s3, 0($s0)
            jal         d                       #    div();
            lw          $s3, 0($s0)
            la          $a0, pf1                #    printf("Result = %d \n");
            li          $v0, 4                  #    //system call to print_str
            syscall
            move        $a0, $s3
            li          $v0, 1                  #    //system call to print_int
            syscall
            la          $a0, pf2                #    printf("Remainder = %d \n");
            li          $v0, 4                  #    //system call to print_str
            syscall
            move        $a0, $s1
            li          $v0, 1                  #    //system call to print_int
            syscall
            jr          $ra                     #    return // TO Operating System
```

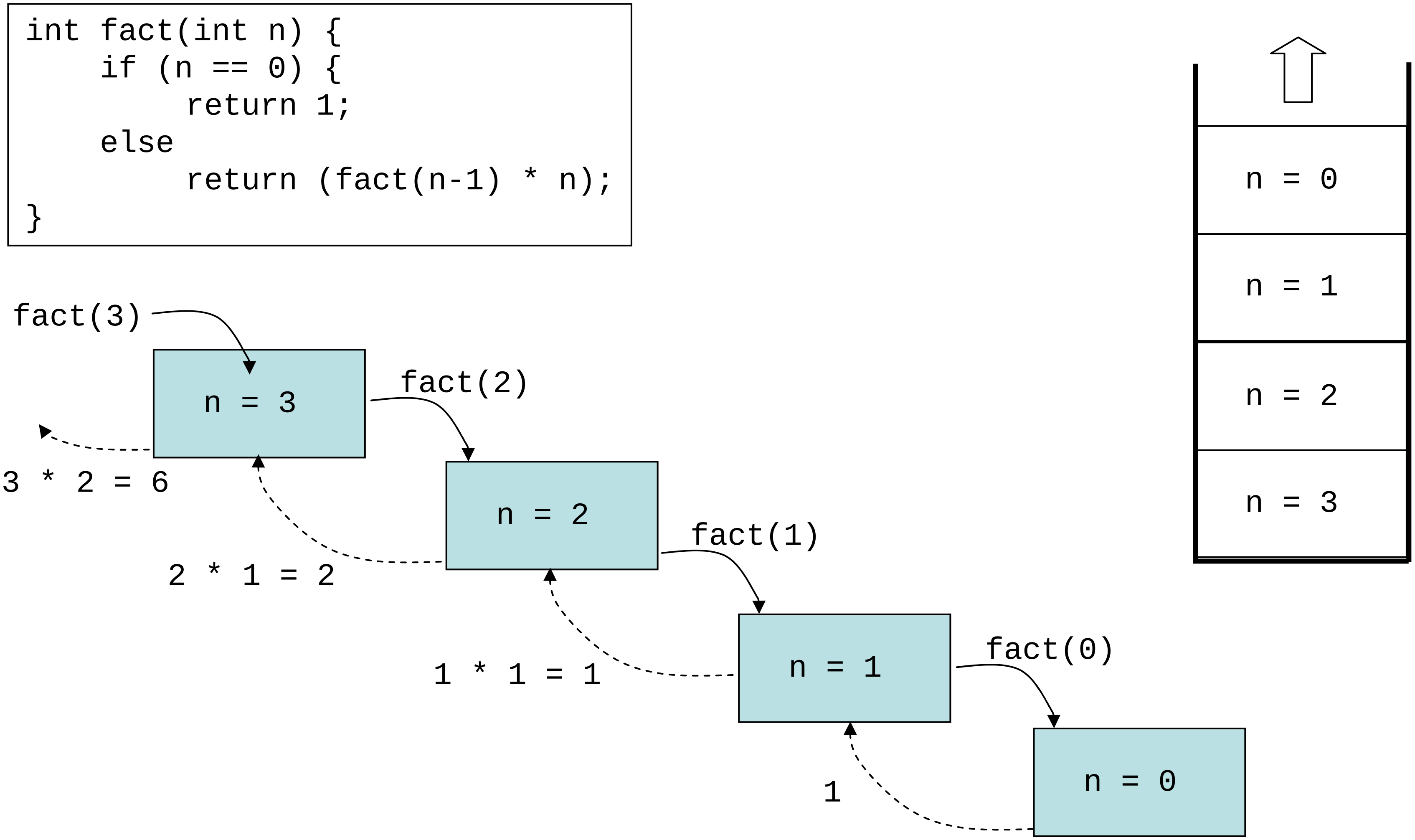
Function Call

Pending Problems With Linkage Approach I

- Registers shared by all procedures
  - procedures must save/restore registers (use stack)
- Procedures should be able to call other procedures
  - save multiple return addresses (use stack)
- Lack of parameters forces access to globals
  - pass parameters in registers
- Recursion requires multiple copies of local data
  - store multiple procedure activation records (use stack)
- Need a convention for returning function values
  - return values in registers

Recursion Basics

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



fact(3)

n = 3

3 * 2 = 6

fact(2)

n = 2

2 * 1 = 2

fact(1)

n = 1

1 * 1 = 1

fact(0)

n = 0

1

n = 0

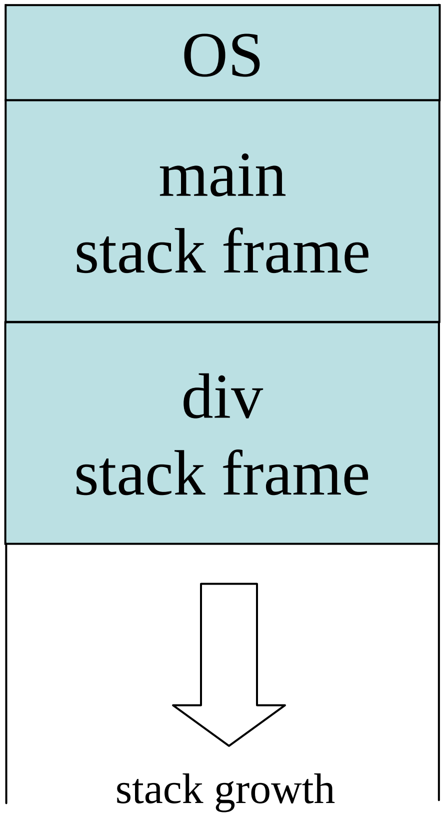
n = 1

n = 2

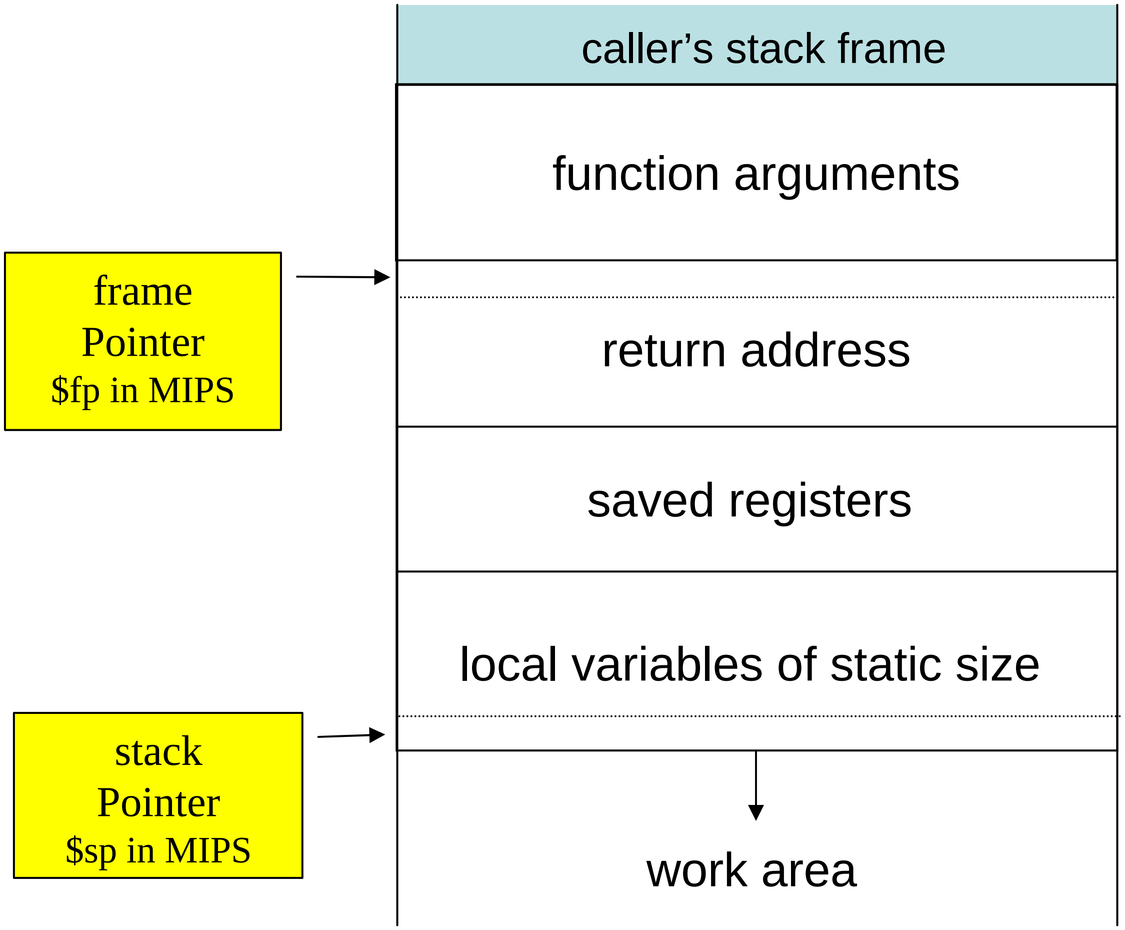
n = 3

Solution: Use Stacks of Procedure Frames

- Stack frame contains:
  - Saved arguments
  - Saved registers
  - Return address
  - Local variables



Anatomy of a Stack Frame



caller’s stack frame

function arguments

frame Pointer $fp in MIPS

return address

saved registers

local variables of static size

stack Pointer $sp in MIPS

work area

Contract: Every function must leave the stack the way it found it

Example: Function Linkage using Stack Frames

```
int x = 0;
int y = 0;
int res = 0;
main () {
  x = 12;
  y = 5;
  res = div(x,y);
  printf(“Res = %d”,res);
}
int div(int a,int b) {
  int res = 0;
  if (a >= b) {
    res = div(a-b,b) + 1;
  }
  else {
    res = 0;
  }
  return res;
}
```

- Add return values
- Add parameters
- Add recursion
- Add local variables

Example: Function Linkage using Stack Frames

```
div:        sub         $sp, $sp, 28            # Alloc space for 28 byte stack frame
            sw          $a0, 24($sp)            # Save argument registers
            sw          $a1, 20($sp)            # a in $a0
            sw          $ra, 16($sp)            # Save other registers as needed
            sw          $s1, 12($sp)            # Save callee saved registers ($sx)
            sw          $s2, 8($sp)
            sw          $s3, 4($sp)             # No need to save $s4, since not used
            li          $s3, 0
            sw          $s3, 0($sp)             # int res = 0;
                                                # Allocate registers for locals
            lw          $s1, 24($sp)            #   a in $s1
            lw          $s2, 20($sp)            #   b in $s2
            lw          $s3, 0($sp)             #   res in $s3

if:         bgt         $s2, $s1, else          # if (a >= b) {
            sub         $a0, $s1, $s2           #
            move        $a1, $s2
            jal         div                     #
            addi        $s3, $v0, 1             #   res = div(a-b, b) + 1;
            j           endif                   # }
else:       li          $s3, 0                  # else { res = 0; }
endif:
            sw          $s1, 32($sp)            #   deallocate a from $s1
            sw          $s2, 28($sp)            #   deallocate b from $s2
            sw          $s3, 0($sp)             #   deallocate res from $s3
            move        $v0, $s3                # return res

            lw          $a0, 24($sp)            # Restore saved registers
            lw          $a1, 20($sp)            # a in $a0
            lw          $ra, 16($sp)            # Save other registers as needed
            lw          $s1, 12($sp)            # Save callee saved registers ($sx)
            lw          $s2, 8($sp)
            lw          $s3, 4($sp)             # No need to save $s4, since not used
            addu        $sp, $sp, 28            # pop stack frame
enddiv:     jr          $ra                     # return;
#
```

Run Div Example in SPIM

MIPS: Procedure Linkage Summary

- First 4 arguments passed in $a0-$a3
- Other arguments passed on the stack
- Return address passed in $ra
- Return value(s) returned in $v0-$v1
- Sx registers saved by callee
- Tx registers saved by caller

Blackboard Exercise

- Implement recursive gcd in MIPS

```
int gcd(int a, int b)
{
    if (a % b == 0)
            return b;
    return gcd(b, a % b);
}
```

Discuss Impact of Other Procedure Features on Implementation

- Reference parameters
- Functional parameters
- Complex object parameters
- Variable number of parameters
- Functional return values
- Named parameters

Which phases of the compiler will be affected?