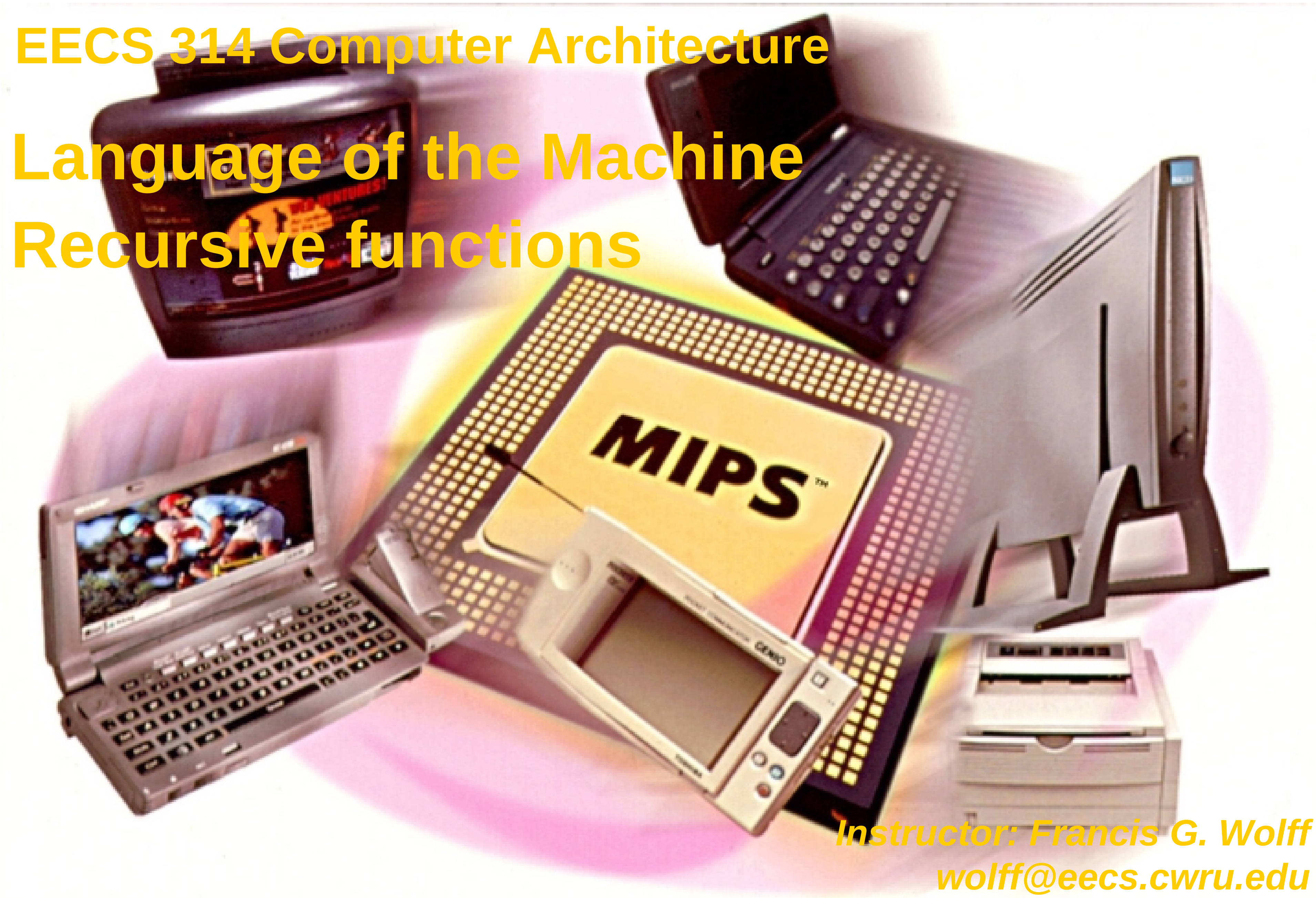
EECS 314 Computer Architecture

Language of the Machine

Recursive functions



Instructor: Francis G. Wolff
wolff@eecs.cwru.edu
Case Western Reserve University

This presentation uses powerpoint animation: please viewshow

Argument Passing greater than 4

- C code fragment

```
g=f(a, b, c, d, e, h);
```

- MIPS assembler

```
addi    $sp, $sp, -4
sw      $s5, 0($sp)     # push h
addi    $sp, $sp, -4
sw      $s4, 0($sp)     # push e
add     $a3, $s3, $0    # register push d
add     $a3, $s2, $0    # register push c
add     $a1, $s1, $0    # register push b
add     $a0, $s0, $0    # register push a
jal     f               # $ra = pc + 4
add     $s5, $v0, $0    # g=return value
```

Argument Passing Options

- 2 common choices
  - "Call by Value": pass a copy of the item to the function/procedure
  - "Call by Reference": pass a pointer to the item to the function/procedure
- C Language: Single word variables passed by value
- Passing an array? e.g., a[100]
  - Pascal--call by value--copies 100 words of a[ ] onto the stack: inefficient
  - C--call by reference--passes a pointer (1 word) to the array a[ ] in a register

Memory Allocation

- int *sumarray(int x[ ], int y[ ])
- adds two arrays and puts sum in a third array
- 3 versions of array function that
  - Dynamic allocation (stack memory)
  - Static allocation (global memory)
  - Heap allocation (malloc, free)

- Purpose of example is to show interaction of C statements, pointers, and memory allocation

Dynamic Allocation

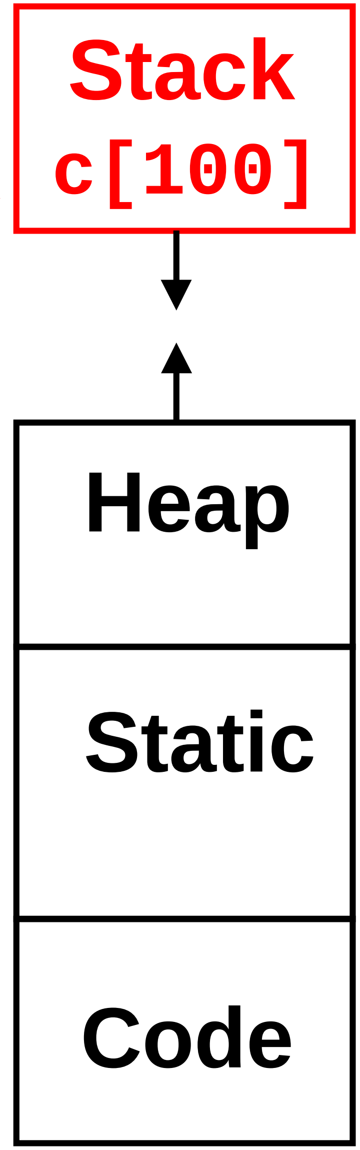
- **Caller provides temporary work space**

```
int f(int x[100], y[100]) {
    int c[100];
    sumarray(x, y, c);
    . . .
```

- **C calling convention means above the same as**

```
    sumarray(&x[0], &y[0], &c[0]);
```

- **i.e. pass the pointer NOT the whole array**

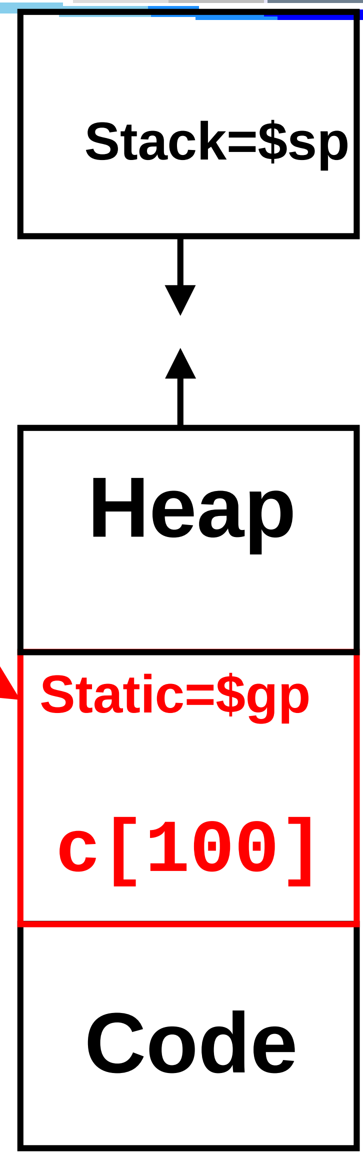


Static allocation (scope: private to function only)

- Static declaration

```
int *sumarray(int a[],int b[]) {
    int i;
    static int c[100];

    for(i=0;i<100;i=i+1)
          c[i] = a[i] + b[i];
    return c;
}
```



- Compiler allocates once for function, space is reused by function
  - On re-entry will still have old data
  - Can not be seen by outside functions

Alternate Static allocation (scope: public to everyone)

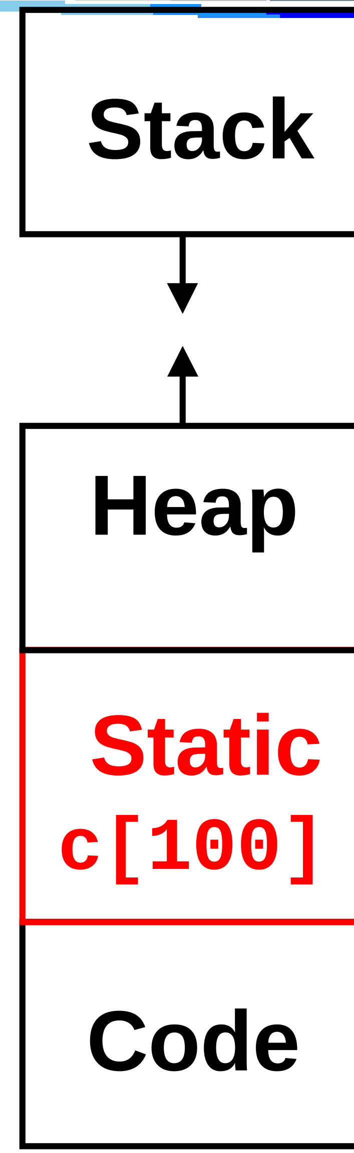
- Static declaration

```
int c[100];

int *sumarray(int a[],int b[]) {
    int i;
    static int c[100];

    for(i=0;i<100;i=i+1)
          c[i] = a[i] + b[i];
    return c;
}
```

- The variable scope of c is very public and is accessible to everyone outside the function



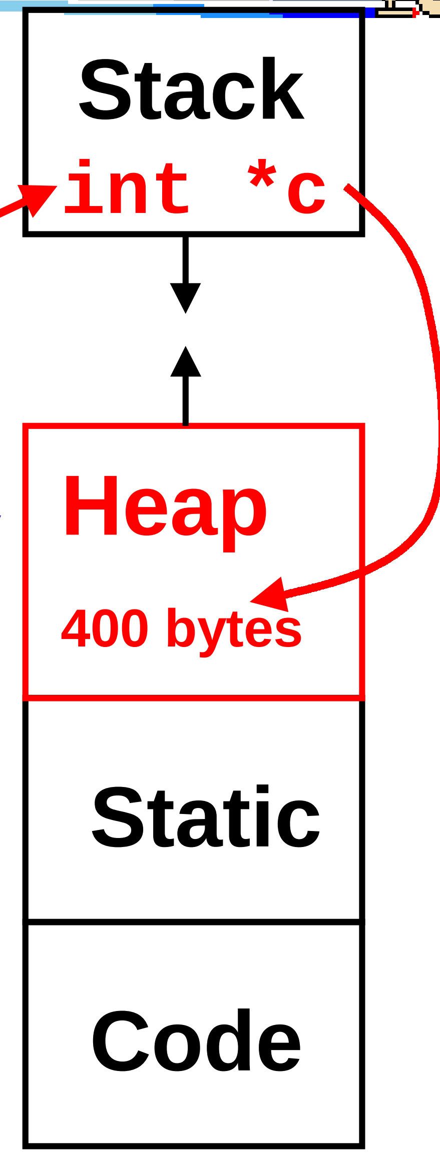
Heap allocation

- Solution: allocate c[100] on heap

```
int * sumarray(int a[],int b[]) {
    int i;
    int *c;  /* the pointer, c, is on stack */

    c=(int *) malloc(100*sizeof(int));
          /* what c is pointing to is in the heap */

    for(i=0;i<100;i=i+1)
          c[i] = a[i] + b[i];
    return c;
}
```



- Not reused unless freed
  - Can lead to memory leaks
  - Java, Scheme have garbage collectors to reclaim free space

Lifetime of storage & scope

- automatic (stack allocated)
  - typical local variables of a function
  - created upon call, released upon return
  - scope is the function
- heap allocated
  - created upon malloc, released upon free
  - referenced via pointers
- external / static
  - exist for entire program

Optimized Compiled Code

```
void sumarray(int a[],int b[],int c[]) {
  int i;
  for(i=0;i<100;i=i+1)
      c[i] = a[i] + b[i];
}
```

```
        addi  $t0,$a0,400    # =100*sizeof(int)=400
Loop:   beq   $a0,$t0,Exit   # if (i==400)
        lw    $t1, 0($a0)    # $t1=a[i]
        lw    $t2, 0($a1)    # $t2=b[i]
        add   $t1,$t1,$t2    # $t1=a[i] + b[i]
        sw    $t1, 0($a2)    # c[i]=a[i] + b[i]
        addi  $a0,$a0,4      # $a0++
        addi  $a1,$a1,4      # $a1++
        addi  $a2,$a2,4      # $a2++
        j     Loop
Exit:   jr    $ra
```

What about Structures?

- Scalars passed **by value** (i.e. int, float, char)
- Arrays passed **by reference** (pointers)
- Structures **by value** (struct { … })
- Pointers **by value**
- Can think of C passing everything **by value**, just that arrays are simply a notation for pointers

Review: Function calling

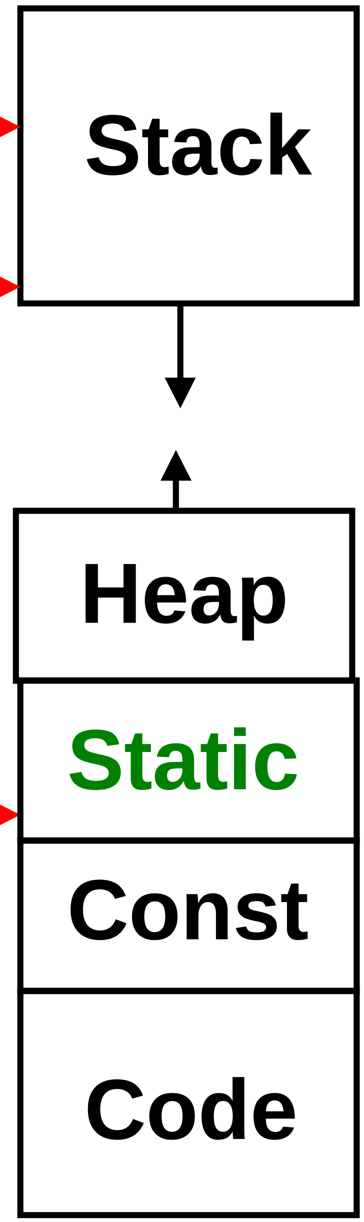
- Follow calling conventions & nobody gets hurt.

- Function Call Bookkeeping:
  - Caller:
    - Arguments $a0, $a1, $a2, $a3
    - Return address $ra
    - Call function jal label # $ra=pc+4;pc=label
  - Callee:
    - Not restored $t0 - $t9
    - Restore caller's $s0 - $s7, $sp, $fp
    - Return value $v0, $v1
    - Return jr $ra # pc = $ra

Review: Program memory layout

Address ∞

Memory Management: R:read only W:write only X:execute only



$fp → frame pointer

$sp → stack pointer

Stack — RW — Space for saved procedure information: return address, dynamic variables, >4 arguments

Heap — RW — Explicitly created space, e.g., malloc(); C pointers

Static — RW — (.data) Global static vars (.bss initialized to 0)

$gp → global pointer

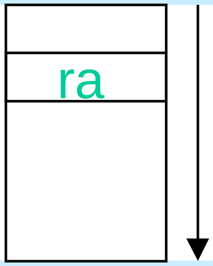
Const — R — const variables (strings)

Code — X — Program: machine instructions (.text)

Address 0

Basic Structure of a Function

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

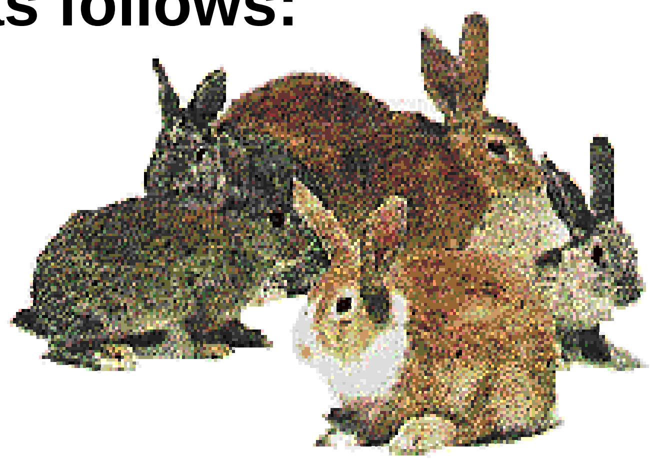



```
#Body            ....
```

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

Recursive functions: Fibonacci Numbers

- **How many pairs of rabbits can be produced from that pair in a year if it is supposed that every month each pair begets a new pair which from the 2nd month on becomes productive.** *Leonardo Pisano aka Fibonacci (1202, Pisa, Italy)*
- **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**



- **Re-writing this 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));
}
```

Prologue: Fibonacci Numbers

- Now, let's translate this to MIPS!
- Reserve 3 words on the stack: $ra, $s0, $a0
- The function will use one $s register, $s0
- Write the Prologue:

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

Epilogue: Fibonacci Numbers

° **Now write the Epilogue:**

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

Body: Fibonacci Numbers

- Finally, write the body. The C code is below. Start by translating the lines indicated in the comments

```
int fib(int n) {
    if(n == 0) { return 1; } /*Translate Me!*/
    if(n == 1) { return 1; } /*Translate Me!*/
    return fib(n - 1) + fib(n - 2);
}
```

```
addi    $v0,$zero,1     # $v0 = 1; return $v0
beq     $a0,$zero,fin   # if (n == 0) goto fin
addi    $t0,$zero,1     # $t0 = 1;
beq     $a0,$t0,fin     # if (n == $t0)goto fin
# Contiued on next slide.  .  .
```

return: Fibonacci Numbers

° Almost there, but be careful, this part is tricky!

```
int fib(int n) {
    . . .
    return (fib(n - 1) + fib(n - 2));
}
```

```
sw   $a0,0($sp)        # Need $a0 after jal
addi $a0,$a0, -1       # $a0 = n - 1
jal  fib               # fib($a0)
add  $s0,$v0,$zero     # $s0 = fib(n-1)
lw   $a0,0($sp)        # Restore original $a0 = n
addi $a0,$a0, -2       # $a0 = n – 2
jal  fib               # fib($a0)
add  $v0,$s0,$v0       # fib(n-1) + fib(n-2)
```

return: $s1 improvement?

- Can we replace the `sw   $a0,0($sp)`
- with `add $s1,$a0,$zero`
- in order to avoid using the stack?

```
add  $s1,$a0,$zero  # was sw  $a0,0($sp)
addi $a0,$a0, -1    # $a0 = n - 1
jal  fib            # fib($a0)
add  $s0,$v0,$zero  # $s0 = fib(n-1)
                    # was lw  $a0,0($sp)
addi $a0,$s1, -2    # $a0 = n – 2
jal  fib            # fib($a0)
add  $v0,$s0,$v0    # fib(n-1) + fib(n-2)
```

return: $t1 improvement?

- Can we replace the `sw   $a0,0($sp)`
- with `add $t1,$a0,$zero`
- in order to avoid using the stack?

- We did save one instruction so far, a plus!
- By convention, all $t registers are not preserved for the caller.
- and therefore we would have to add another lw and sw for $t1 to the stack.

Here's the complete code: Fibonacci Numbers

```
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
sw   $a0, 0($sp)
addi $a0, $a0, -1
jal  fib
add $s0, $v0, $zero
```

```
lw   $a0, 0($sp)
addi $a0, $a0, -2
jal  fib
add  $v0, $v0, $s0

fin: # epilog
lw   $s0, 4($sp)
lw   $ra, 8($sp)
addi $sp, $sp, 12
jr   $ra
```

Sun Microsystems SPARC Architecture

- In 1987, Sun Microsystems introduced a 32-bit RISC architecture called SPARC.
- Sun's UltraSparc workstations use this architecture.
- The general purpose registers are 32 bits, as are memory addresses.
- Thus 2^{32} bytes can be addressed.
- In addition, instructions are all 32 bits long.
- SPARC instructions support a variety of integer data types from single bytes to double words (eight bytes) and a variety of different precision floating-point types.

SPARC Registers

•The SPARC provides access to 32 registers

- regs 0 %g0 ! global constant 0 (MIPS $zero, $0)
- regs 1-7 %g1-%g7 ! global registers
- regs 8-15 %o0-%o7 ! out (MIPS $a0-$a3,$v0-$v1,$ra)
- regs 16-23 %L0-%L7 ! local (MIPS $s0-$s7)
- regs 24-31 %i0-%i7 ! in registers (caller's out regs)

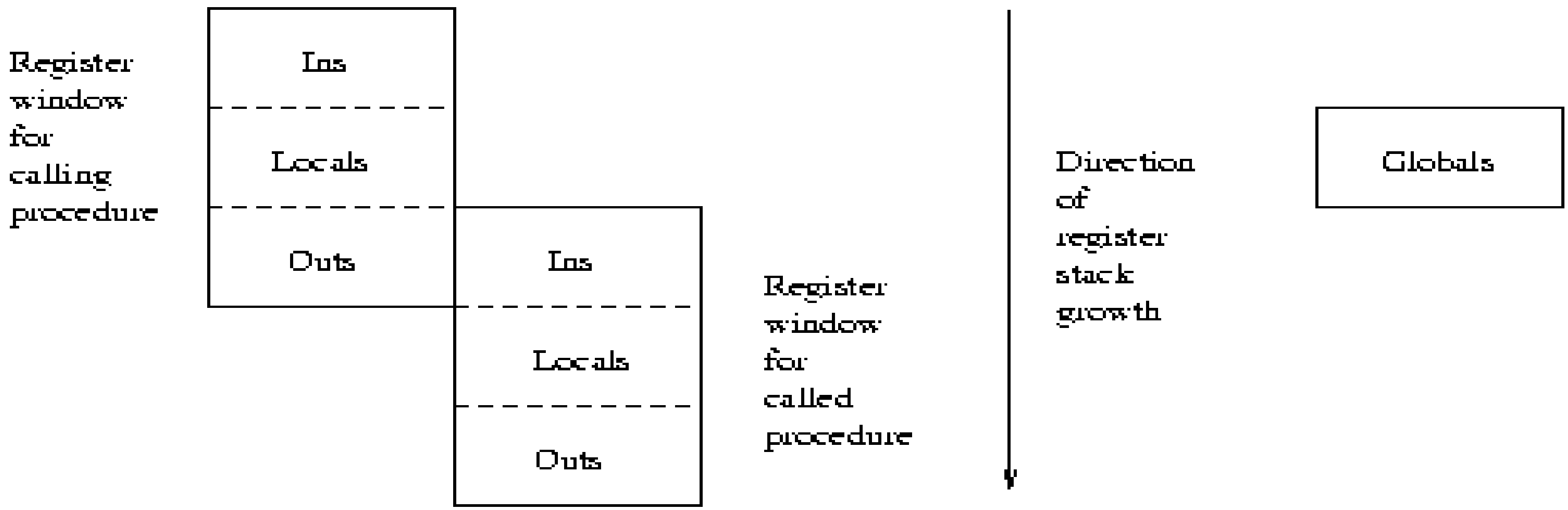
• The global registers refer to the same set of physical registers in all procedures.

• Register 15 (%o7) is used by the call instruction to hold the return address during procedure calls (MIPS ($ra)).

• The other registers are stored in a register stack that provides the ability to manipulate register windows.

• The local registers are only accessible to the current procedure.

SPARC Register windows



- When a procedure is called, parameters are passed in the out registers and the register window is shifted 16 registers further into the register stack.
- This makes the in registers of the called procedure the same as the out registers of the calling procedure.
- in registers: arguments from caller (MIPS %a0-$a3)
- out registers: When the procedure returns the caller can access the returned values in its out registers (MIPS $v0-%v1).

SPARC instructions

Arithmetic

```
add %l1, %i2, %l4     ! local %l4 = %l1 + i2
add %l4, 4, %l4       ! Increment %l4 by four.
mov 5, %l1            ! %l1 = 5
```

Data Transfer

```
ld [%l0], %l1         ! %l1 = Mem[%l0]
ld [%l0+4], %l1       ! %l1 = Mem[%l0+4]
st %l1, [%l0+12]      ! Mem[%l0+l2]= %l1
```

Conditional

```
cmp %l1, %l4    ! Compare and set condition codes.
bg  L2          ! Branch to label L2 if %l1 > %l4
nop             ! Do nothing in the delay slot.
```

SPARC functions

Calling functions

```
      mov %l1, %o0           ! first parameter = %l1
      mov %l2, %o1           ! second parameter = %l2
      call fib               ! %o0=.fib(%o0,%o1,…%o7)
      nop                    ! delay slot: no op
      mov %o0, %l3           ! %i3 = return value
```

Assembler

```
gcc hello.s                ! executable file=a.out
gcc hello.s -o hello       ! executable file=hello
gdb hello                  ! GNU debugger
```

SPARC Hello, World.

```
        .data
hmes:.asciz Hello, World\n"
        .text
        .global main    ! visible outside
main:
        add   %r0,1,%%o0      ! %r8 is %o0, first arg
        sethi %hi(hmes),%o1 ! %r9, (%o1) second arg
        or    %o1, %lo(hmes),%o1
        or    %r0,14,%o2      ! count in third arg
        add   %r0,4,%g1       ! system call number 4
        ta 0                  ! call the kernal

        add   %r0,%r0,%o0
        add   %r0,1,%g1       ! %r1, system call
        ta 0                  ! call the system exit
```