Computer Architecture

Chapter 2-2

Instructions: Language of the Computer

Procedures

- **A major program structuring mechanism**
- **Calling & returning from a procedure requires a protocol.**
- **The protocol is a sequence of conventions.**

```
…
RetVal = SomeProcedure(arg1, arg2, …, argN);
...
```

Caller

Callee

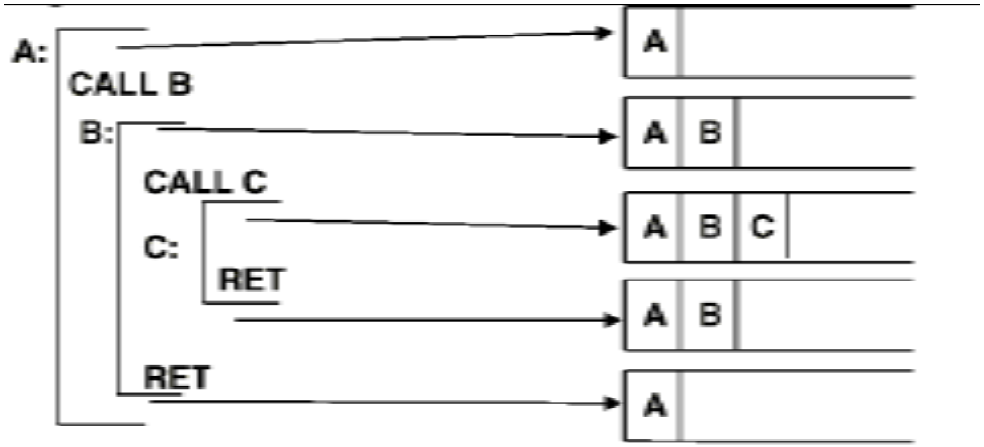
```
int SomeProcedure(int arg1, char arg2, …, float argN)
{
        int LocalVar, TempVar;

        …
        OtherProcedure(par1, …, parM);
        ...
        return RetVal;
}
```

Procedure Call Requirements (Caller/Callee)

- **Caller**
  1. must pass parameters to the callee.
  2. must pass to the callee the return address.
  3. must save whatever is volatile (i.e., registers that can be used by the callee).

- **Callee**
  1. must save its own return address.
  2. must provide local storage for its own use.
  3. should support recursive calls.

Program Stack



- Stacks are a natural structure to allocate dynamic data for procedures (as well as call/return linkage information) in a Last-In-First-Out fashion.
- $sp (register 29) is automatically loaded to point to the last used slot on top of the stack.
- By convention, the stack grows towards lower addresses.

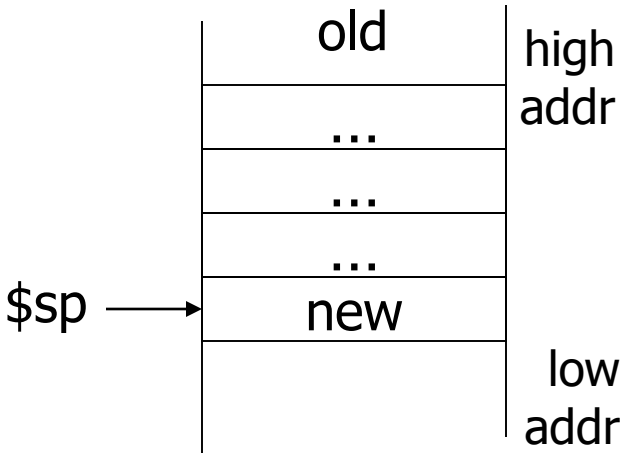
Stack

- **MIPS stack**
  - push

```
addi    $sp, $sp, -4
sw      $t0, 0($sp)
```

  - pop

```
lw      $t0, 0($sp)
addi    $sp, $sp, 4
```



Procedure Call Mechanisms

- **Registers are used for:**
  - passing a small number of parameters (up to 4, $a0 - $a3)
    - Q: more than 4 parameters?
  - Passing the return address ($ra)
    - jal ProcedureAddress
  - returning function values ($v0, $v1): Why two?
  - keeping track of stack ($sp)
- **Stack is used for:**
  - saving registers to be used by callee
  - saving information about the caller (i.e., return address)
  - passing extra parameters
  - allocating local data for the called procedure

Procedure Call Convention

```
0   $0   zero constant 0
1   $at reserved for assembler
2   $v0 expression evaluation &
3   $v1 function results
4   $a0 arguments (caller saves)
5   $a1
6   $a2
7   $a3
8   $t0 temporary: caller saves
...
15  $t7
```

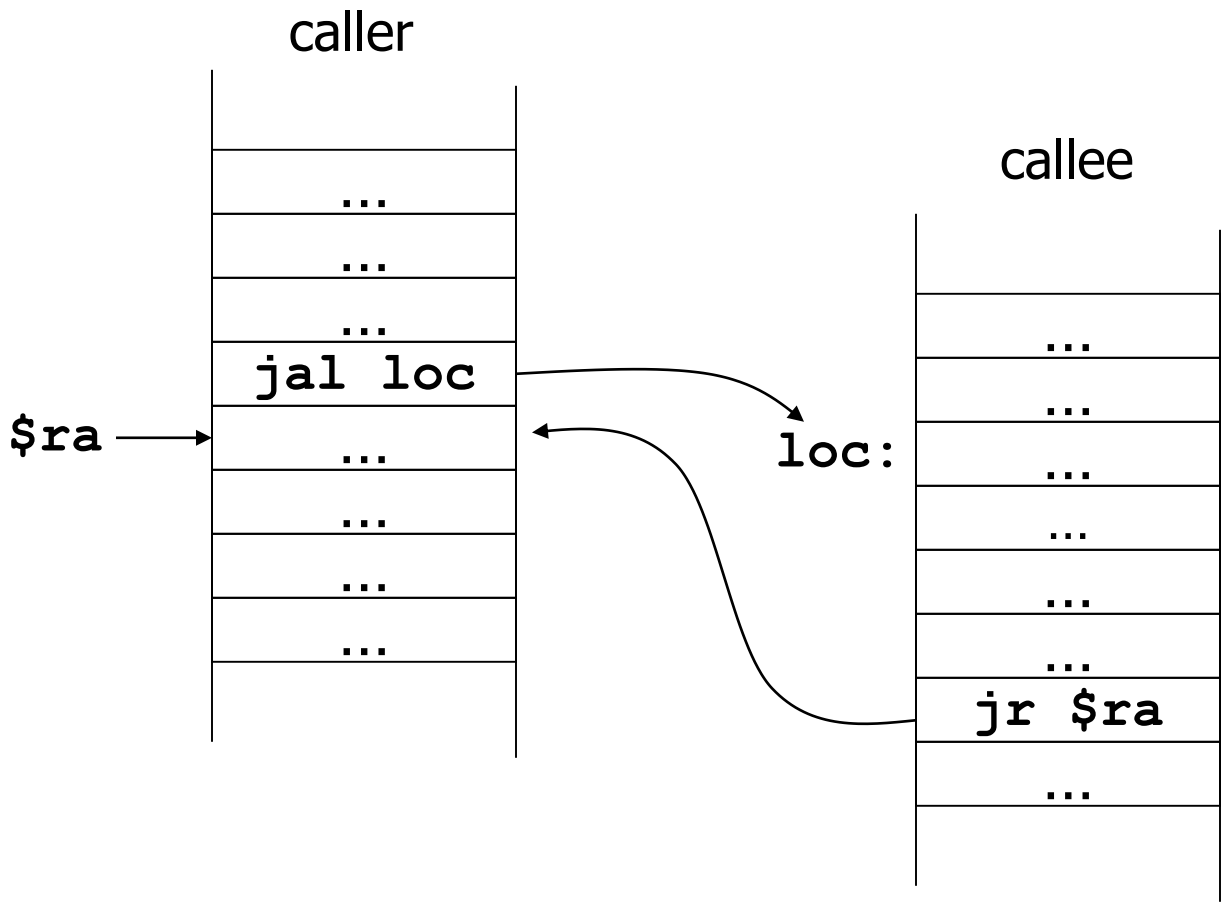
```
16 $s0   callee saves
...
23 $s7
24 $t8 temporary (cont'd)
25 $t9
26 $k0 reserved for OS kernel
27 $k1
28 $gp global pointer
29 $sp stack pointer
30 $fp frame pointer
31 $ra return address (HW)
```

Register Saving Convention

Preserved on Call	Not preserved on Call
Saved Registers: $s0-$s7	Argument Registers: $a0-$a3
Stack Pointer: $sp	Return Value Regs: $v0-$v1
Frame Pointer: $fp	Temporaries: $t0-$t9
Return Address: $ra	
Global Pointer: $gp (for static (vs. automatic) variables)	

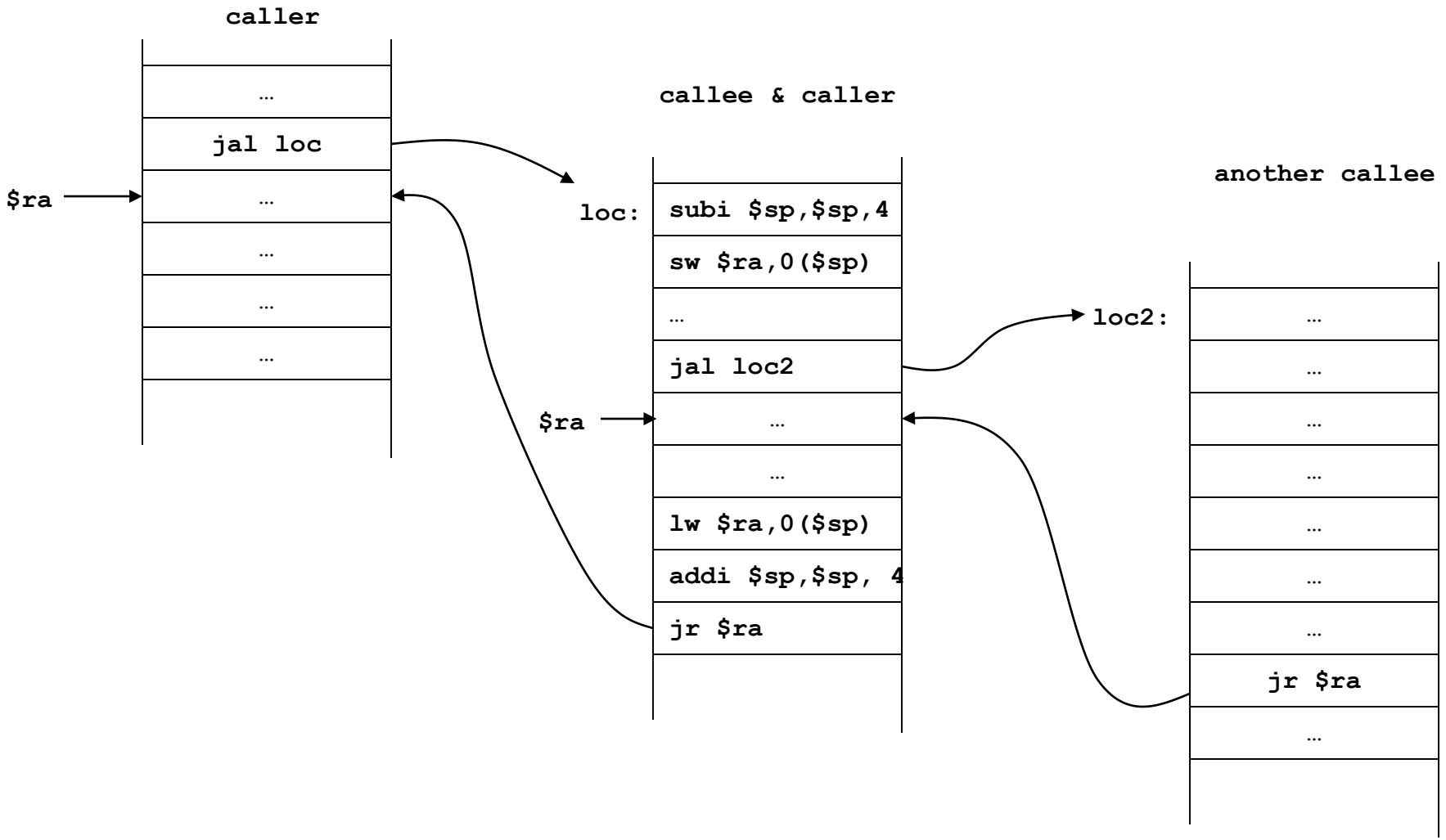
- **Preserved registers must be saved and restored by called procedure (callee) if modified.**
- **Unpreserved registers must be saved by caller if needed after call complete.**
- **Similar convention for Floating-Point registers**

MIPS Procedure Calls



What if the callee calls another procedure?

MIPS Procedure Call (2)



Example: Call Sequence

- **Two arguments in $t0 and $t3**
- **Want to save $t6 and $t7**

```
add $a0,$t0,$zero
```

↗

```
move  $a0, $t0   # first arg in $a0 (move is a pseudoinstruction)
move  $a1, $t3           # second arg in $a1
addi  $sp, $sp, -8       # adjust stack pointer
sw    $t6, 4($sp)        # save $t6 on stack
sw    $t7, 0($sp)        # save $t7 on stack
jal   Function           # call function
```

- **Callee will do:**

```
Function:
            addi $sp, $sp, -4      # adjust stack pointer
            sw   $ra, 0($sp)       # save return address
```

Example: Return from Call

- **Before returning the callee will put results in $v0-$v1 if needed then:**

```
lw   $ra, 0($sp)        # restore the return address
addi $sp, $sp, 4        # adjust stack
jr   $ra                # return
```

- **The caller will restore $t6-$t7:**

```
lw   $t6, 4($sp)        # restore $t6
lw   $t7, 0($sp)        # restore $t7
addi $sp, $sp, 8        # adjust stack pointer
```

Leaf Procedure Example

- **C code:**

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Arguments g, …, j in $a0, …, $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Result in $v0

-

```
leaf_example:
addi $sp, $sp, -4
sw   $s0, 0($sp)
add  $t0, $a0, $a1
add  $t1, $a2, $a3
sub  $s0, $t0, $t1
add  $v0, $s0, $zero
lw   $s0, 0($sp)
addi $sp, $sp, 4
jr   $ra
```

Save $s0 on stack

Procedure body

Result

Restore $s0

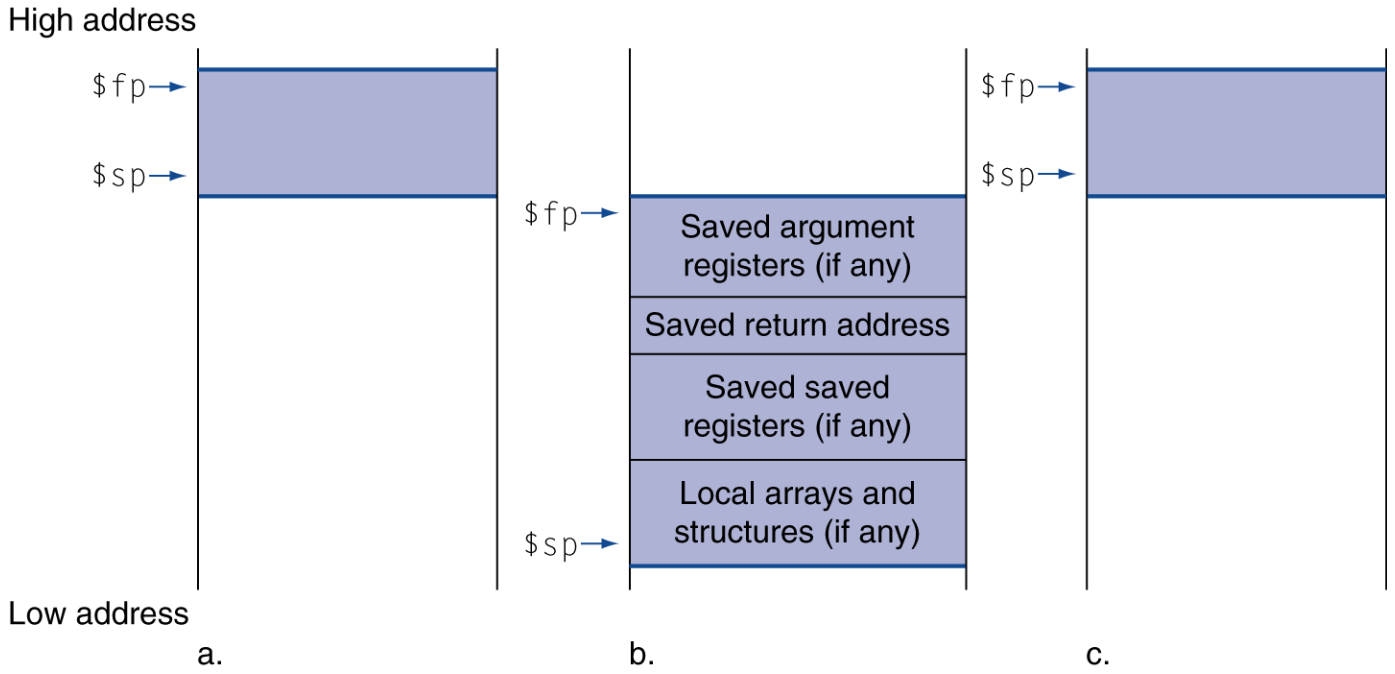
Return

Non-Leaf Procedure: Factorial Example

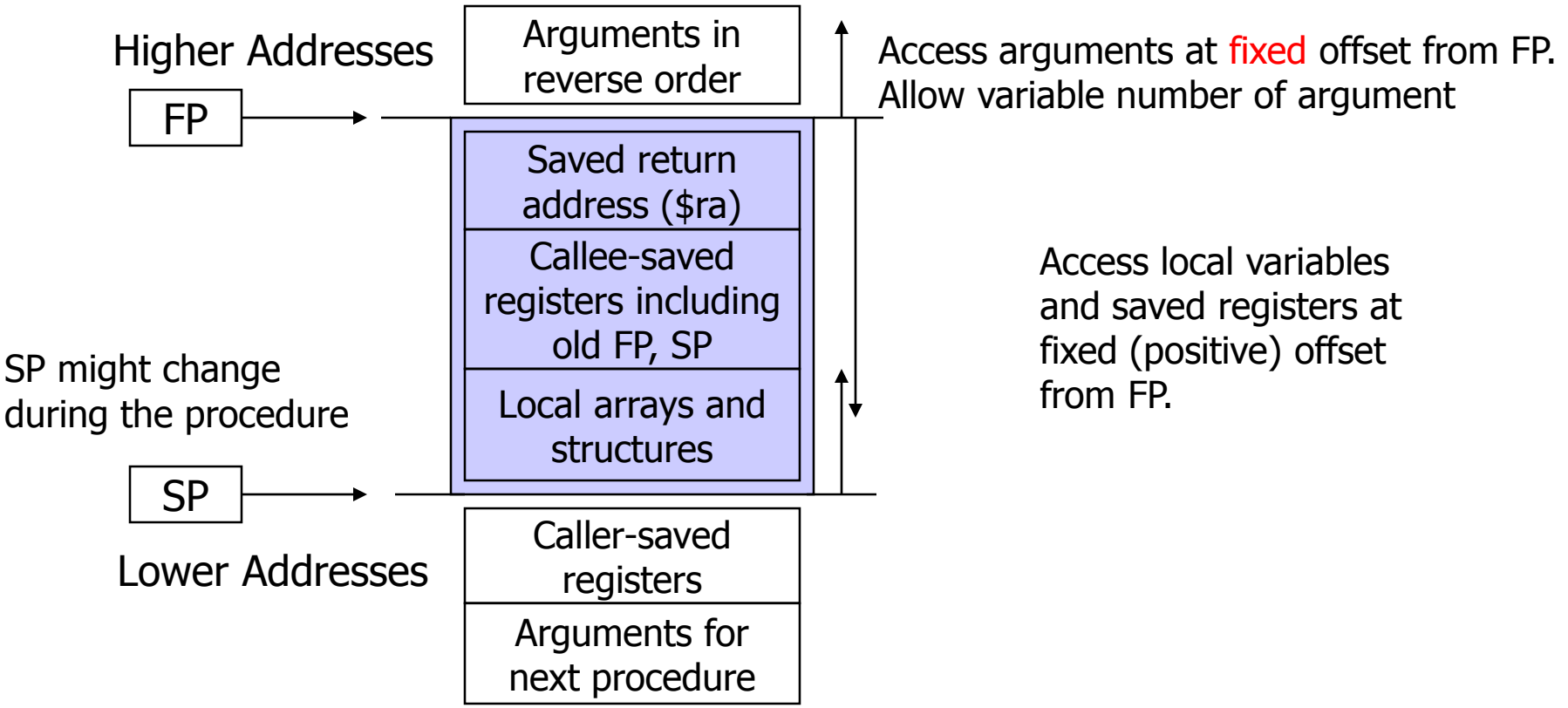
```
fact:
# procedure entry
        addi $sp, $sp, -8       # adjust sp
        sw   $ra, 4($sp)        # save ret. addr.
        sw   $a0, 0($sp)        # save arg.
# procedure body
        slti $t0, $a0, 1        # test for n < 1
        beq  $t0, $zero, L1     # if n≥1, goto L1
        addi $v0, $zero, 1      # return 1
        addi $sp, $sp, 8        # pop 2 items
        jr   $ra                # return to after jal
L1:     addi $a0, $a0, -1       # if n≥1: gets (n-1)
        jal  fact               # call fact w/ (n-1)
        lw   $a0, 0($sp)        # restore n
        lw   $ra, 4($sp)        # restore ret. addr.
        addi $sp, $sp, 8        # pop 2 items
        mul  $v0, $a0, $v0      # return n*fact(n-1)
        jr   $ra                # return to caller
```

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

Local Data on the Stack



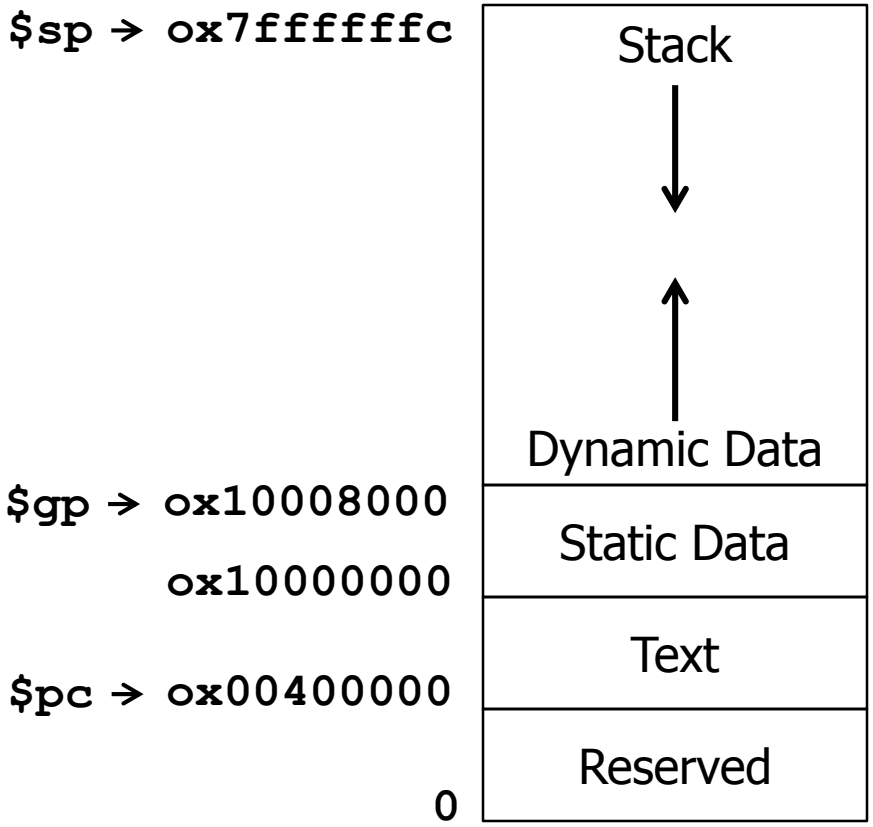
Procedure Frame/Stack Frame/Activation Record



- **Frame is the memory between $fp and $sp.**
- **Can be built in many different ways.**
- **We describe the most common convention on most MIPS machines.**

MIPS Memory Layout

- **Text: program code**
- **Static data: global variables**
  - e.g., static variables in C, constant arrays and strings
  - $gp
- **Dynamic data: heap**
  - E.g., malloc in C, new in Java
- **Stack: automatic storage**



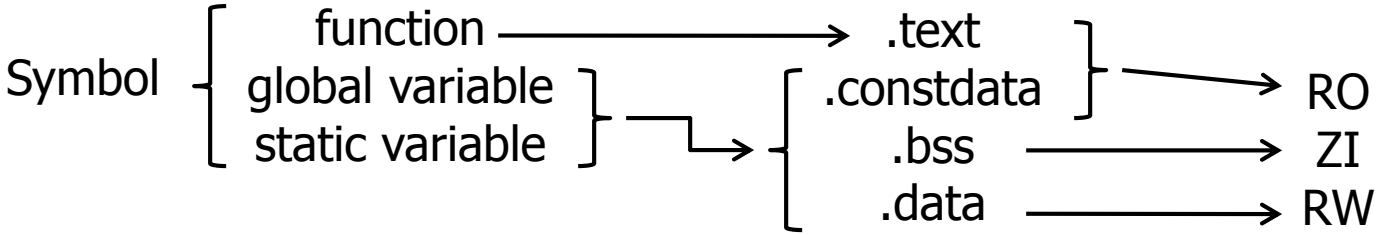
Variables

- **local**
  - valid within the function or block {}
- **global**
  - valid within a file
  - can be used by other files
- **static**
  - local static: retain its value
  - global static: protected from other files
- **volatile**

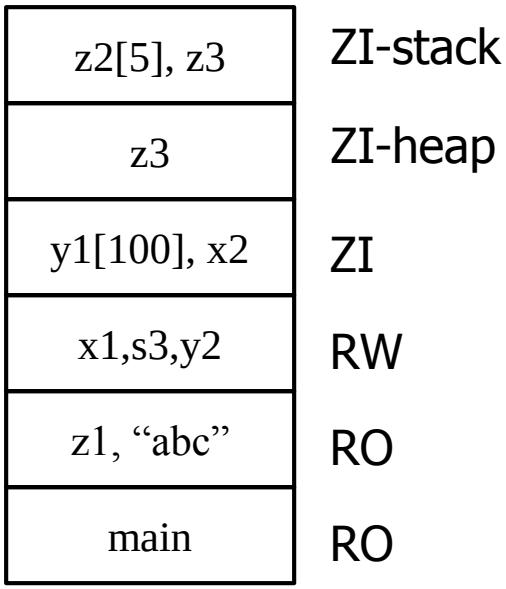
```
int a = 0x10;
void funcA (void)
{
  int a1 = 3;
  int b1 = 4;
  ......
  return;
}

char b = 0x20;
void funcB (void)
{
  .......
  return;
}
```

```
static int d = 0x10;

void func(void)
{
  static int a = 3;
  static int b = 4;

  a++;
  b++;

  return;
}
```

Symbol




```
int x1 = 5;                          // in .data
int y1[100];                         // in .bss
int const z1[3] = {1,2,3};           // in .constdata
char *s3 = "abc";                    // s3 in .data, "abc" in .constdata

int main(int x)                       // in .text
{
   static int x2;                    // in .bss
   static int y2 = 10;               // in .data
   char z2[5];                       // stack
   char z3;                          // stack

   z3 = (char *)malloc(suzeof(char)*200);       // heap
}
```



Byte and Halfword

```
lb $t0, 0($sp)          # read byte
sb $t0, 0($gp)          # write byte

lh $t0, 0($sp)          # read halfword
sh $t0, 0($gp)          # write halfword
```

```
while ((x[i]=y[i])!='\0') i++;
```

x[]:$a0, y[]:$a1, i:$s0

```
L1:     add  $t1,$s0,$a1     # $t1 = addr. of y[i]
        lb   $t2,0($t1)      # $t2 = y[i]
        add  $t3,$s0,$a0     # $t3 = addr. Of x[i]
        sb   $t2,0($t3)      # x[i] = y[i]
        beq  $t2,$zero,L2    # if y[i]==0, goto L2
        addi $s0,$s0,1       # i++
        j    L1
L2:
```

Assembly Language vs. Machine Language

- **Assembly provides convenient symbolic representation**
  - much easier than writing down numbers
  - e.g., destination first
- **Machine language is the underlying reality**
  - e.g., destination is no longer first
- **Assembly can provide 'pseudoinstructions'**
  - e.g., “move $t0, $t1” exists only in Assembly
  - would be implemented using “add $t0,$t1,$zero”
- **When considering performance you should count real instructions**

To summarize:

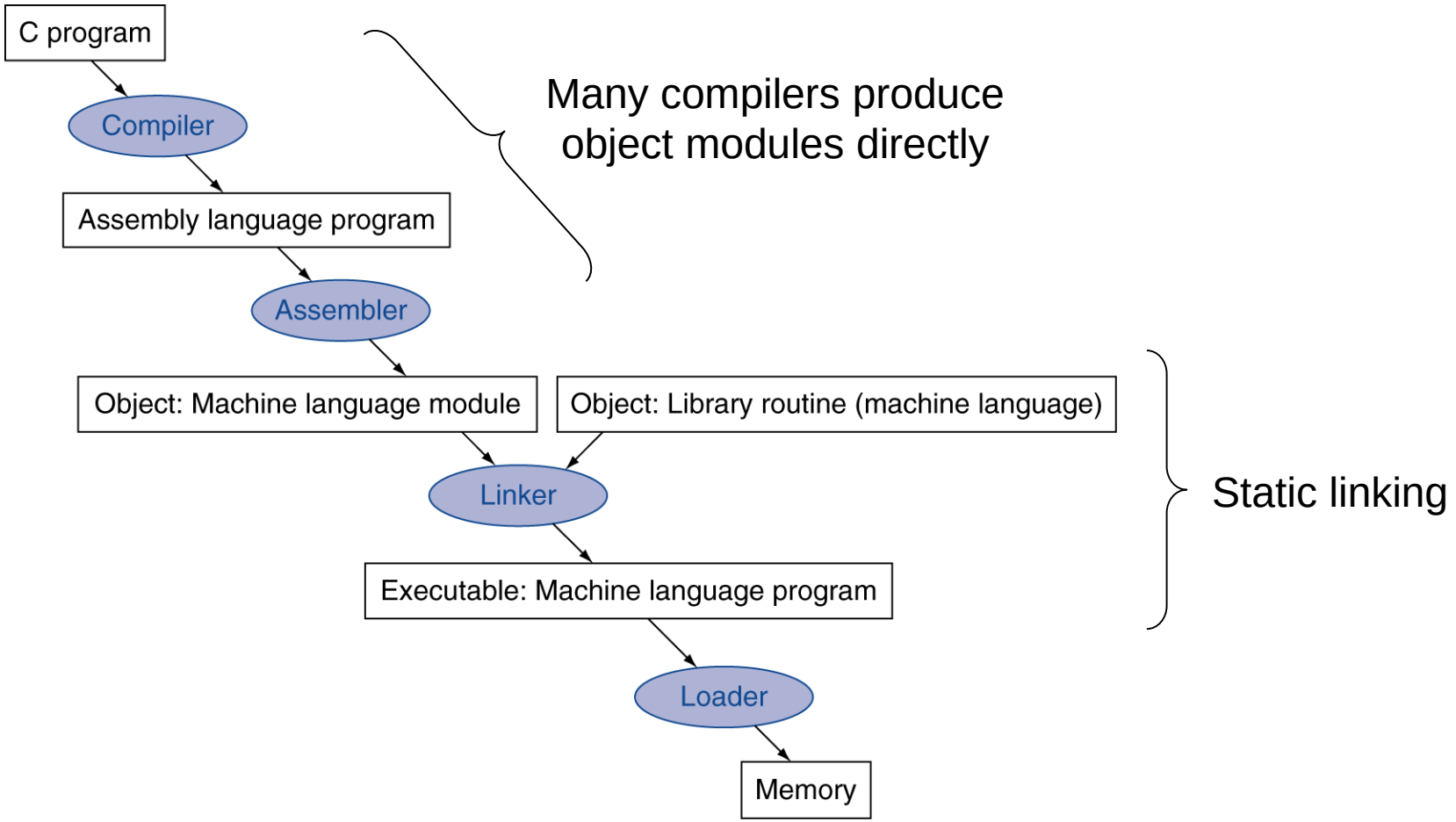
MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	`$s1 =` Memory[`$s2` + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1 =` Memory[`$s2` + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

Translation Hierarchy



Alternative Architectures

- **Design alternative:**
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI
    - ***"The path toward operation complexity is thus fraught with peril. To avoid these problems, designers have moved toward simpler instructions"***

The Intel x86 ISA

- **Evolution with backward compatibility**
  - 8080 (1974): 8-bit microprocessor, Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080, Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor, Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU, Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations, Paged memory and segments
  - i486 (1989): pipelined, on-chip caches and FPU
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture
  - Pentium III (1999): Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001): New microarchitecture, Added SSE2 instructions
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements), Added SSE3 instructions
  - Intel Core (2006): Added SSE4 instructions, virtual machine support
  - AMD64 (2007): SSE5 instructions
  - Advanced Vector Extension (2008): Longer SSE registers, more instructions

IA-32 Overview

- **Complexity:**
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., “base or scaled index with 8 or 32 bit displacement”
- **Saving grace:**
  - the most frequently used instructions are not too difficult to build
  - compilers avoid complex instructions

“what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective”

Fallacies

- **Powerful instruction ⇒ higher performance**
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions

- **Use assembly code for high performance**
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

What is Common Case?

- **Measure MIPS instruction executions in benchmark programs**
  - Consider making the common case fast
  - Consider compromises

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	`add, sub, addi`	16%	48%
Data transfer	`lw, sw, lb, lbu, lh, lhu, sb, lui`	35%	36%
Logical	`and, or, nor, andi, ori, sll, srl`	12%	4%
Cond. Branch	`beq, bne, slt, slti, sltiu`	34%	8%
Jump	`j, jr, jal`	2%	0%

Summary

- **Instruction complexity is only one variable**
  - lower instruction count vs. higher CPI / lower clock rate
- **Design Principles:**
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast
- **Instruction set architecture**
  - a very important abstraction indeed!
- **MIPS: typical of RISC ISAs**
  - c.f. x86