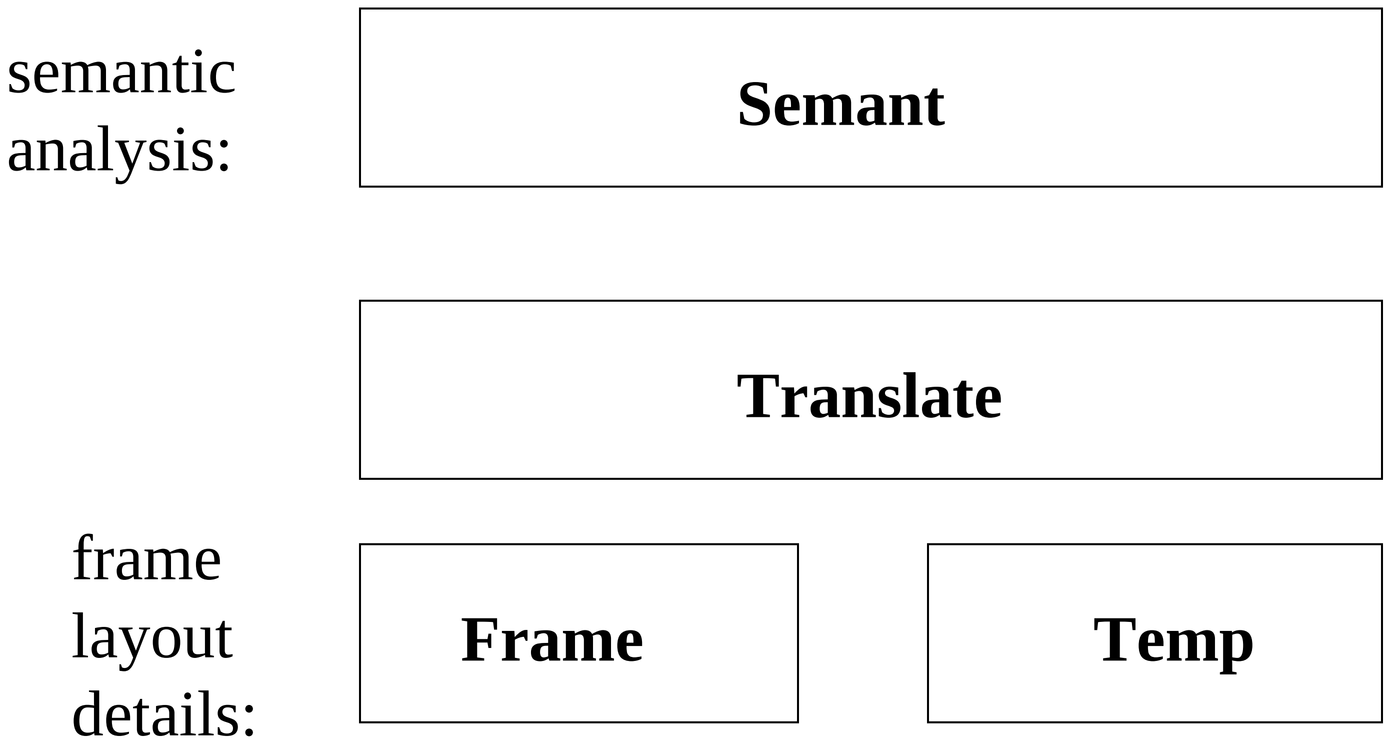
Runtime System - 3

- **Classes to implement frames in Tiger** **(Appel, Ch 6)**
  - **package Frame**
  - **package SPIM (like MIPS)**
  - **package Temp**
    - **Labels and temporaries**
- **Separating machine dependent details from machine independent details**
- **SPIM calling conventions and register usage**

Layers of Abstraction



package Frame

- **Holds information about formal parameters and local variables in machine independent manner**

```
public abstract class Access{…}
public abstract class AccessList{…head; …tail;…}
public abstract class Frame{
   abstract public Frame newFrame(Label name,
               Util.BoolList formals);
               public Label name;
               public AccessList formals;
   abstract public Access allocLocal(boolean escape);
```

Make a New Frame

Assume f is function of 2 parameters the first of which escapes (i.e., it has a memory address). To create its frame in Main.Main:

Frame.Frame frame = new Mips.Frame(…);

frame.newFrame(g, new BoolList(*true*,

newBoolList(*false*, null)))

name of function and list describing its arguments

Frame objects contain

- **Locations of all formals**
- **Instructions to implement "view shift"**
- **Number of locals allocated so far**
- ***label* at which the function's machine code will begin**

formals: 1	**InFrame(0)**	**necessary**	**sp ← sp - K**
2	**InReg(t_157)**	**code:**	**M[sp+K+0] ← r2**
			t_157 ← r4

Allocating New Locals

- **New locals allocated in a frame by call to *allocLocal(true)***
  - **Value of boolean variable tells if register or frame storage to be used**
  - **Could be used for *lets* in a function**

```
let var v:= 6
   in print(v); end;
let var v := 7
   in print(v); end;
```

Will allocate 2 local vars in frame here, unless optimizer sees opportunity to merge space.

Escaping Variables

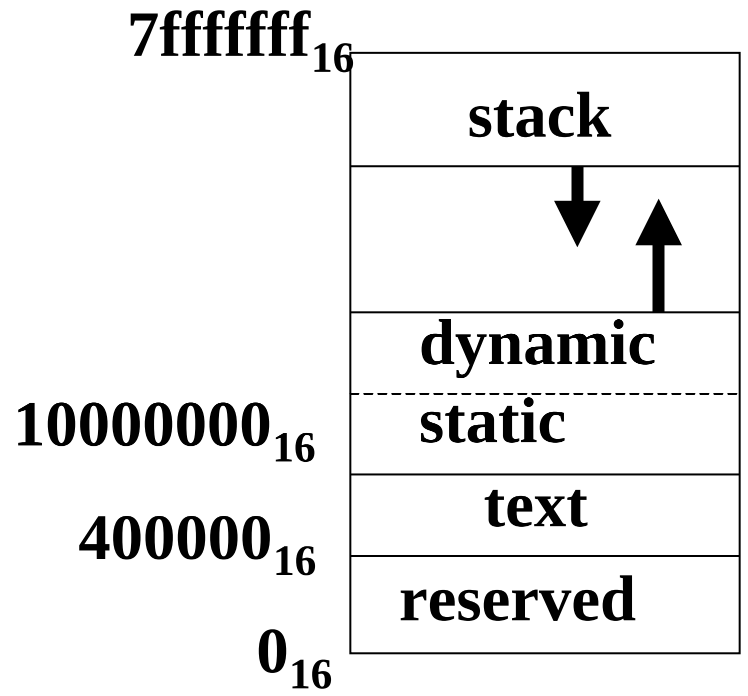
- **A variable that is passed by reference, has its address taken or is accessed from a nested function**
- **Can’t tell this property about variables at their declarations**
- **Requires an independent analysis with an environment much like type checking; examine *escaping* uses**

package Temp

- ***Temp*’s are virtual registers**
  - **May not be enough registers available to store all temporaries in a register**
  - **Delay decision until later**
- ***Label*’s are like labels in assembler, a location of a machine language instruction**
- **Classes *Temp* and *Label* in *package Temp***
- **Packages *Frame* and *Tem*p provide machine independent views of variables**

SPIM

- **A simulator that executes MIPS programs**
- **Target machine for our Ocelot compiler project**
- **Memory model**
  - **Text - program code**
  - **Data**
    - **Static (globals)**
    - **Dynamic (heap)**
  - **Stack (runtime stack)**



SPIM Calling Conventions

- **MIPS CPU has 32 general purpose registers numbered 0-31**
  - **Register $0 always contains the value 0**
  - **Registers $at, $k0, $k1 are reserved, cannot be used by users**
  - **Registers $a0-$a3 used to pass first 4 arguments to routines**
  - **Registers $v0, $v1 are used to return values from functions**

SPIM Calling Conventions

- Registers $t0-$t9 are caller-save registers used to used to hold temporaries not preserved across calls
- Registers $s0-$s7 are callee-save registers, used to hold long-lived values across calls
- Register $gp is used to point to middle of 64K block of memory in the static data segment
- Register $sp is stack pointer, which points to first free location in stack; $fp is frame pointer; jai instruction writes $ra, the return address from a procedure call

MIPS Register Usage

Register	Number	Use
$zero	0	constant 0
$at	1	reserved for assembler
$v0-$v1	2-3	expr eval and return values
$a0-$a3	4-7	arguments 1-4
$t0-$t7	8-15	temporary (not preserved)
$s0-$s7	16-23	saved temporary across calls
$t8-$t9	24-25	temporary (not preserved)
$k0-$k1	26-27	reserved for os
$gp	28	pointer to global data area
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

MIPS Calling Context Switch-1

- **Caller on executing the call**
  - **Pass first 4 arguments in registers \$a0-\$a3; put rest of arguments at beginning of callee's frame**
  - **Save caller-save registers (\$a0-\$a3,\$t0-\$t9)**
  - **Execute *jai* which jumps to callee's first instruction and saves return address in \$ra**

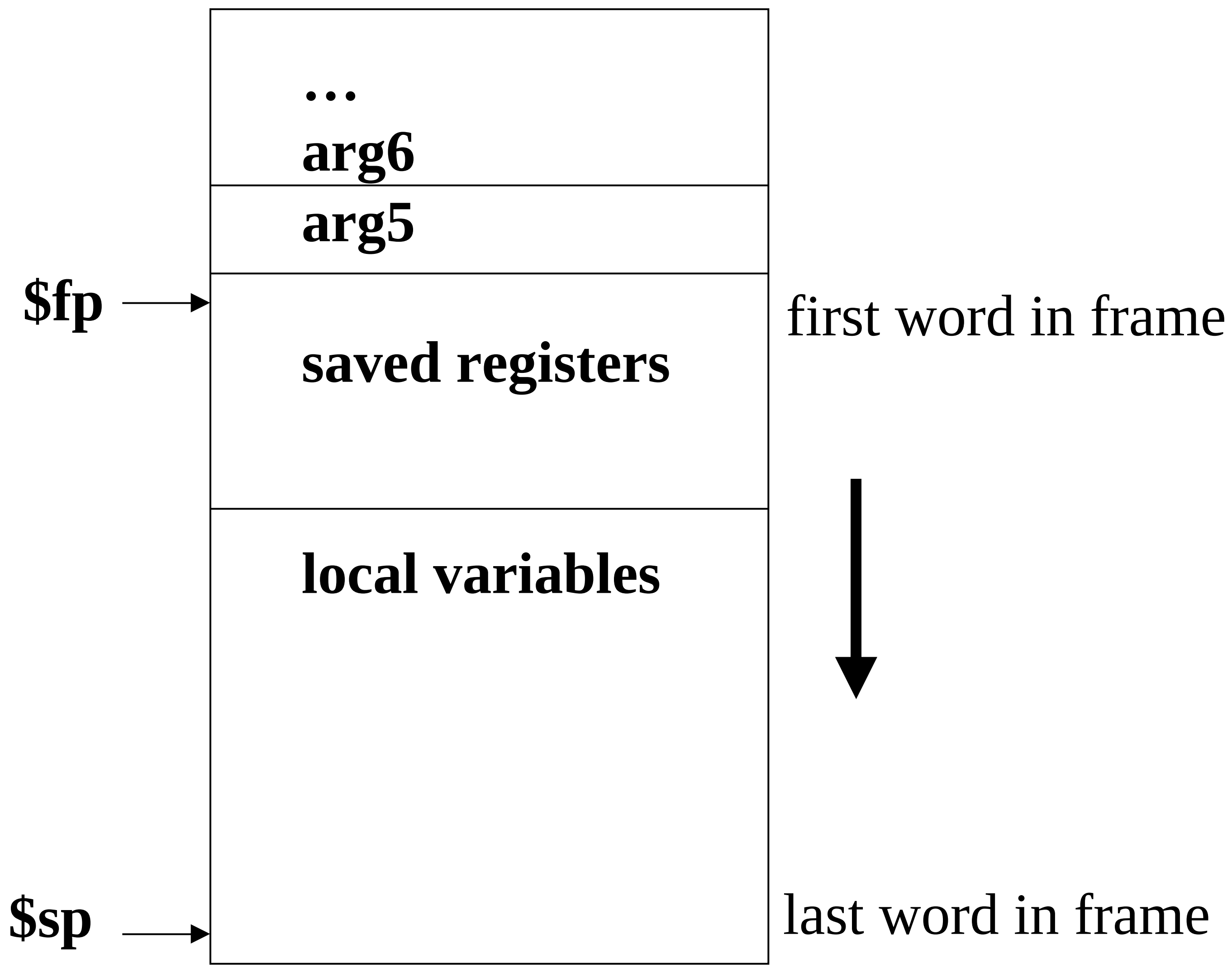
MIPS Calling Context Switch-2

- **Callee, before it starts running**
  - **Allocate memory for frame (subtract frame size from stack pointer)**
  - **Save callee-save registers in frame ($s0-$s7, $fp, and $ra if callee makes a call)**
  - **Establish frame pointer in $fp by adding stack frame's size minus 4 to $sp**

MIPS Calling Context Switch-3

- **Before returning to the caller, the callee**
  - **Places return value in $v0**
  - **Restores all callee-save registers**
  - **Pops stack frame by adding the frame size to $sp**
  - **Return by jumping to address in $ra**

MIPS Frame



...
arg6
arg5
$fp
first word in frame
saved registers
local variables
$sp
last word in frame

Factorial Example (SPIM manual, p A-26ff))

C code for computing 10! :

```
main(){
    printf("the factorial of 10 is %d\n",fact(10));
}
int fact (int n){
    if (n < 1) return(1);
    else return (n*fact(n-1));
}
```

Example

Minimum frame size=24 bytes. Prologue code from main:

```
    .text
    .globl main
main:
    subu    $sp,$sp,32      #stack frame 32 bytes long
    sw      $ra,20($sp)     #save return address in $ra
    sw      $fp,16($sp)     #save old frame pointer
    addu    $fp,$sp,28      #setup frame pointer
```

Example

main calls factorial routine, passing it a single argument 10. after factorial returns, main calls library print routine printf and passes it both a format string and the result from factorial.

```
li      $a0,10      #put argument into $a0
jal     fact        #call factorial routine

la      $a0,$LC     #put format string into $a0
move    $a1,$v0     #move factorial result into $a1
jal     printf      #call print function
```

Example

main's epilogue restores saved registers, pop its stack frame and return.

```
    lw      $ra,20($sp)     #restore return address
    lw      $fp,16($sp)     #restore frame pointer
    addu    $sp,$sp,32      #pop stack frame
    jr      $ra             #return to caller

    .rdata
$LC:
    .ascii  "The factorial of 10 is %d\n\000"
```

Example

factorial routine structured similarly to main. factorial prologue:

```
    .text
fact:
    subu    $sp,$sp,32      #stack frame is 32 bytes
    sw      $ra,20($sp)     #save return address
    sw      $fp,16($sp)     #save frame pointer
    addu    $fp,$sp,28      #setup frame pointer
    sw      $a0,0($fp)      #save argument n
```

Example

computation within factorial routine:

```
    lw      $v0,0($fp)      #load n
    bgtz    $v0,$L2         #branch if n>0
    li      $v0,1           #return 1
    j       $L1             #jump to return code
$L2:
    lw      $v1,0($fp)      #load n
    subu    $v0,$v1,1       #compute n-1
    move    $a0, $v0        #move argument to $a0
    jal     fact            #recursive call of factorial
    lw      $v1,0($fp)      #load n
    mul     $v0,$v0,$v1     #compute factorial(n-1) * n
                            #result in $v0
```

Example

factorial routine epilogue:

$L1:

```
lw      $ra, 20($sp)    # restore $ra
lw      $fp, 16($sp)    # restore frame pointer $fp
addu    $sp,$sp,32      # pop stack pointer $sp
j       $ra             #return to caller
```