ECE 250 / CS 250
Computer Architecture

Procedures

Benjamin Lee

Slides based on those from Alvin Lebeck, Daniel Sorin, Andrew Hilton, Amir Roth, Gershon Kedem

Today's Lecture

Admin

- HW #2 is up, due Sunday Feb 16 11:55pm
- No late submissions!
  - We will post solutions on Monday so they are available prior to the midterm.

Outline

- Review
- MIPS Assembly Procedures, calling conventions

Reading

- An overview of SPIM and the MIPS32 instruction set @ http://spimsimulator.sourceforge.net/HP_AppA.pdf
- MIPS Quick Ref (PDF in docs section of web site)

Recap: Details of the MIPS instruction set

- Register zero always has the value zero
- Jump-and-Link instruction place return address (PC+4) into link register
- All instructions change all 32 bits of the destination register (lui, lb, lh) and read all 32 bits of sources (add, sub, and, or, …)
- Immediate arithmetic and logical instructions are extended as follows:
  - logical immediates are zero extended to 32 bits
  - arithmetic immediates are sign extended to 32 bits
- lb and lh extend data as follows:
  - lbu, lhu are zero extended
  - lb, lh are sign extended

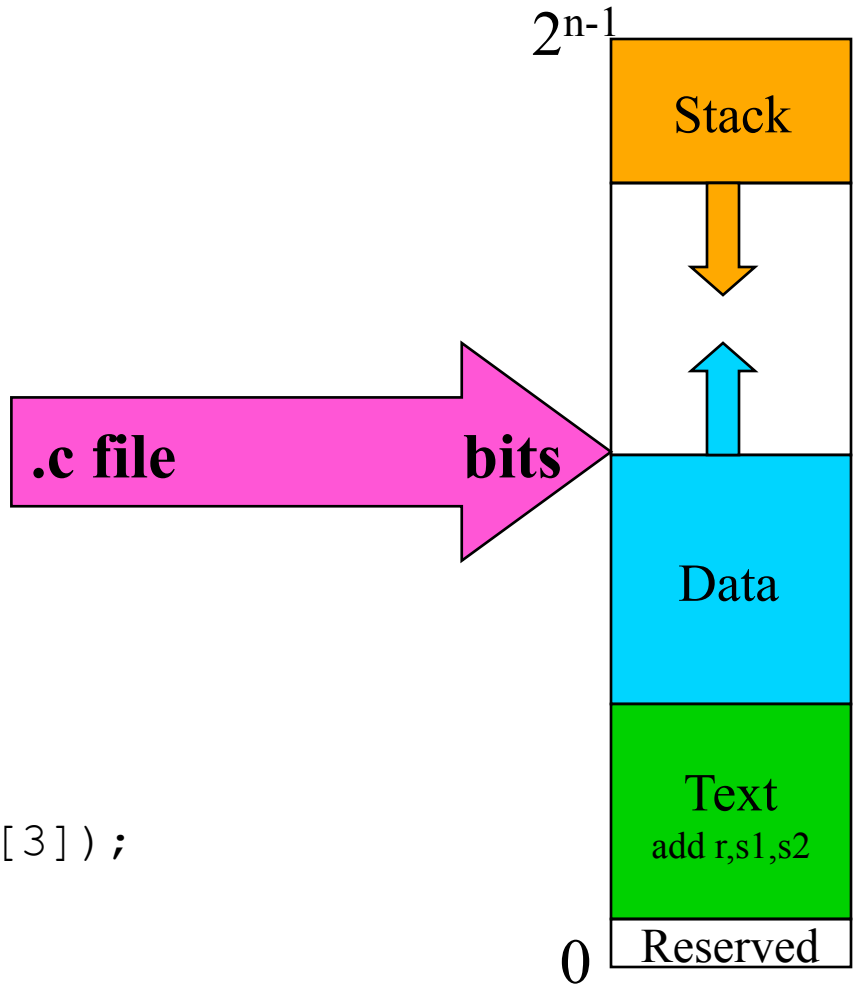
Review: A Simple Program

Add two numbers x & y together

```
   .text                # declare text segment
   .align 2             # align it on 4-byte boundary
main:                   # label for main
   la   $3, x           # load address of x into R3 (pseudo-inst)
   lw   $4, 0($3)       # load value of x into R4
   la   $3, y           # load address of y into R3 (pseudo-inst)
   lw   $5, 0($3)       # load value of y into R5
   add  $6, $4, $5      # compute x+y
   jr   $31             # return to calling routine

   .data                # declare data segment
   .align 2             # align it on 4-byte boundary
x: .word 10             # initialize x to 10
y: .word 3              # initialize y to 3
```

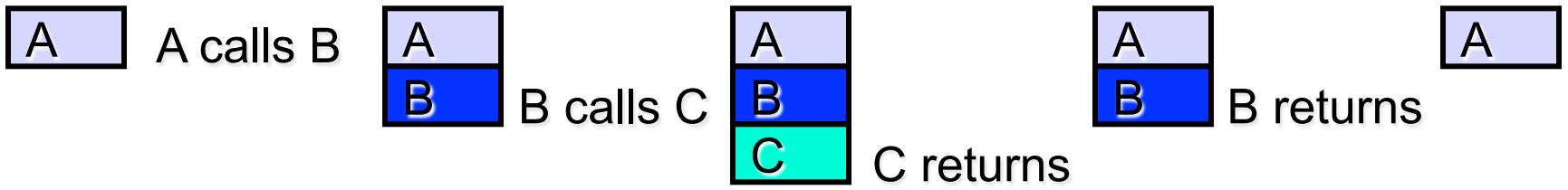
Review: A Program

```
#include <stdio.h>

main()
{
   int *a = new int[100];
   int *p = a;
   int k;

   for (k = 0; k < 100; k++)
      {
        *p = k;
        p++;
      }

   printf ("entry 3 = %d\n",a[3]);
}
```



Control Idiom: Procedure Call

- Procedure calls obey stack discipline
  - Local procedure state contained in stack frame
  - When a procedure is called, a new frame opens
  - When a procedure returns, the frame collapses
- Procedure stack is in memory
  - Distinct from operand stack (e.g., stack-based ISA) which is not addressable
- Procedure linkage implemented by convention
  - Called procedure ("callee") expects frame to look a certain way
    - Input arguments and return address are in certain places
  - Caller "knows" this

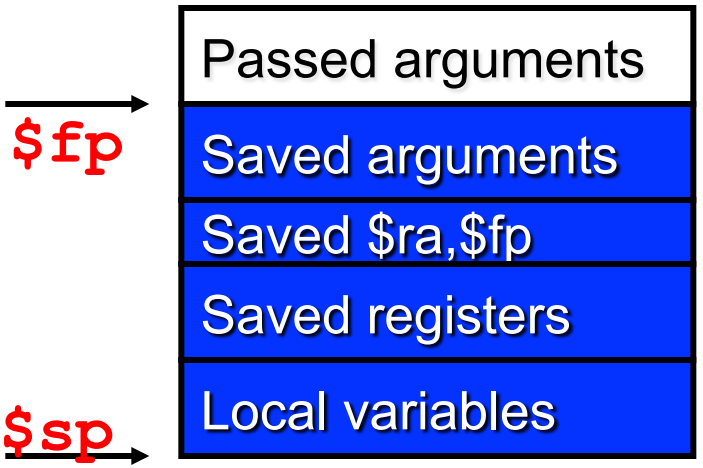


MIPS Procedure Calls

- Procedure stack implemented in software
  - No ISA support for frames: set them up with conventional stores
  - Stack is linear in memory and grows down (popular convention)
  - One register reserved for stack management
    - Stack pointer (`$sp`=`$29`): points to bottom of current frame
    - Sometimes also use frame pointer (`$fp`=`$30`): top of frame
      - Why? For dynamically variable sized frames: e.g., long expression evaluation with many temporary locations
- Frame layout
  - Contents accessed using $sp

    ```
    sw $ra,24($sp)
    ```

    Note: `$ra`=`$r31`
  - Displacement addressing



MIPS Calls and Register Convention

- Some inefficiencies with basic frame mechanism
  - Registers: do all need to be saved/restored on every call/return?
  - Arguments: must all be passed on stack?
  - Returned values: are these also communicated via stack?
  - No! Specify a register convention

    `$2,$3($v0,$v1)`: expression evaluation and return values

    `$4-$7($a0-$a3)`: function arguments

    `$8-$15,$24,$25($t0-$t9)`: caller saved temporaries

      - A saves before calling B only if needed after B returns

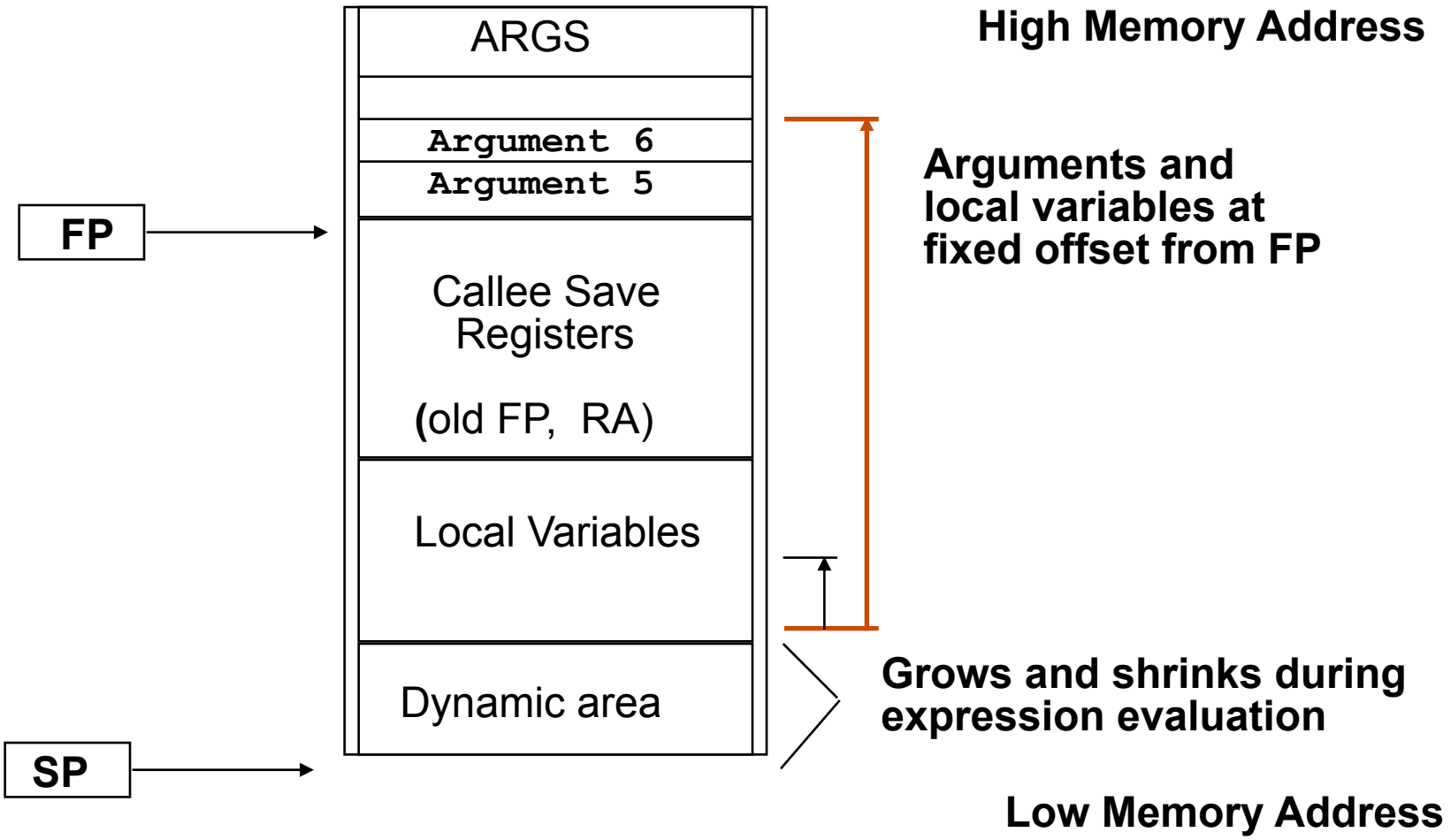
    `$16-$23($s0-$s7)`: callee saved

      - A needs after B returns, B saves if it uses also

MIPS Register Naming Conventions

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
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	Pointer to global area
29	sp	Stack pointer
30	fp	frame pointer
31	ra	Return Address (HW)

Call-Return Linkage: Stack Frames



MIPS Procedure Calling Conventions

Calling Procedure

- Step-1: Set up the arguments:
  - The first four arguments (arg0-arg3) are passed in registers $a0-$a3
  - Remaining arguments are pushed onto the stack
    (in reverse order arg5 is at the top of the stack).
- Step-2: Save caller-saved registers
  - Save registers $t0-$t9 if they contain live values at the call site.
- Step-3: Execute a jal instruction.

MIPS Procedure Calling Conventions

Called Routine

- Step-1: Establish stack frame.
  - Subtract the frame size from the stack pointer.
    subiu $sp, $sp, <frame-size>
  - Typically, minimum frame size is 32 bytes (8 words).
- Step-2: Save callee saved registers in the frame.
  - Register $fp is always saved.
  - Register $ra is saved if routine makes a call.
  - Registers $s0-$s7 are saved if they are used.
- Step-3: Establish frame pointer
  - Add the stack <frame size> - 4 to the address in $sp
    addiu $fp, $sp, <frame-size> - 4

MIPS Procedure Calling Conventions

On return from a call

- Step-1: Put returned values in registers $v0, [$v1].
  (if values are returned)
- Step-2: Restore callee-saved registers.
  - Restore $fp and other saved registers. [$ra, $s0 - $s7]
- Step-3: Pop the stack
  - Add the frame size to $sp.
    addiu $sp, $sp, <frame-size>
- Step-4: Return
  - Jump to the address in $ra.
    jr $ra

Example 2 Solution w/ Conventions

```
# Example for CS/ECE 250
# Program to add together list of 9 numbers.
        .text                       # Code
        .align  2
        .globl  main
main:                               # MAIN procedure Entrance
        subu    $sp, 40             #\ Push the stack
        sw      $ra, 36($sp)        # \ Save return address
        sw      $s3, 32($sp)        #  \
        sw      $s2, 28($sp)        #   > Entry Housekeeping
        sw      $s1, 24($sp)        #  /  save registers on stack
        sw      $s0, 20($sp)        # /
        move    $v0, $0             #/ initialize exit code to 0
        move    $s1, $0             #\
        la      $s0, list           # \ Initialization
        la      $s2, msg            # /
        la      $s3, list+36        #/
```

Example 2 Solution w/ Conventions

```
#                          Main code segment
Again:  slt     $t1, $s0, $s3     #   Begin main loop
        beqz    $t1, loop_end     # all done exit loop
        lw      $t6, 0($s0)      #\
        addu    $s1, $s1, $t6    #/  Actual "work"
                                  #    SPIM I/O
         li      $v0, 4           #\
         move    $a0, $s2         # >  Print a string
         syscall                  #/
         li      $v0, 1           #\
         move    $a0, $s1         # >  Print a number
         syscall                  #/
         li      $v0, 4           #\
         la      $a0, nln         # > Print a string (eol)
         syscall                  #/

         addu    $s0, $s0, 4      #\ index update and
         b       again            #/  end of loop
```

Example 2 Solution w/ Conventions

```
#                       Exit Code
loop_end:
        move    $v0, $0          #\
        lw      $s0, 20($sp)     # \
        lw      $s1, 24($sp)     #  \
        lw      $s2, 28($sp)     #   \ Closing Housekeeping
        lw      $s3, 32($sp)     #   /   restore registers
        lw      $ra, 36($sp)     #  / load return address
        addu    $sp, 40          # / Pop the stack
        jr      $ra              #/    exit(0) ;
        .end    main             #  end of program

#               Data Segment

        .data                    # Start of data segment
list:   .word   35, 16, 42, 19, 55, 91, 24, 61, 53
msg:    .asciiz "The sum is "
nln:    .asciiz "\n"
```

- Code is in docs section of course web site

System Call Instruction

System call is used to communicate with the operating system, and request services (memory allocation, I/O)

Basic sequence is:

1. Load system call code (value) into Register $v0
2. Load arguments (if any) into registers $a0, $a1 or $f12 (for floating point).
3. do: syscall
4. Results returned in registers $v0 or $f0.

- Note: $v0 = $2, $a0=$4, $a1 = $5 per naming conventions

SPIM System Calls (Useful Subset)

Service	System call code	Arguments	Result
print_int	01	$a0 = integer	
print_float	02	$f12 = float	
print_double	03	$f12 = double	
print_string	04	$a0 = string	
read_int	05		integer (in $v0)
read_float	06		float (in $f0)
read_double	07		double (in $f0)
read_string	08	$a0 = buffer, $a1 = length	
sbrk	09	$a0 = amount	address (in $v0)
exit	10		

Echo Example

Note: Does not follow calling conventions for main

```
.text
main:
   li   $v0, 5          # code to read an integer
   syscall              # do the read (invokes the OS)
   move $a0, $v0        # copy result from v0 to a0

   li   $v0, 1          # code to print an integer
   syscall              # print the integer

   li   $v0, 4          # code to print string
   la   $a0, nln        # address of string (newline)
   syscall
```

Echo Example

```
    li   $v0, 8          # code to read a string
    la   $a0, name       # address of buffer (name)
    li   $a1, 8          # size of buffer (8 bytes)
    syscall

    la   $a0, name       # address of string to print
    li   $v0, 4          # code to print a string
    syscall

    jr   $ra             # return

    .data
    .align 2
name:  .word 0,0
nln:   .asciiz "\n"
```

SPIM demo

- SPIM is a program that simulates the behavior of MIPS32 computers
  - Can run MIPS32 assembly language programs
  - You will use QtSPIM to run/test the assembly language programs you write for homework in this class
- QtSPIM
  - Download appropriate version for your machine from http://spimsimulator.sourceforge.net/
  - Quick demo

How to use QtSPIM

1. Download and install QtSPIM per web site directions
2. Use text editor to create a .s file on your local hard
   - Must have a .globl main (so main is externally visible)
   - Textedit: set preferences to plaintext, disable *.txt suffix and autocorrect
3. Start QtSPIM
4. IMPORTANT! QtSPIM must always be initialized/reset before you run your program.
5. Click the "reinitialize and load file" icon
   - Better than just load file since it guarantees reinitialize ☺
   - Sometimes you will get syntax, parse errors here. Fix them…
6. When the program loads correctly (without errors)
   - Click the green "run" icon

- But what about a program with bugs? No printf debugging!

Debugging with QtSPIM

1. Single step execution (the icon with numbers…)
   - This lets you run one instruction at a time
2. Breakpoints (stop execution at a specific point)
   - Right click on the instruction where you want execution to stop
3. Examine Register and Memory contents
   - Left panel has registers
   - Text and Data Segment Tabs
   - Can display in binary/hex/decimal (drop down menu)
   - Can disable display of kernel text and data for now (drop down)

- Sometimes the console window doesn't come to front

Control Idiom: Call by Reference

- Passing arguments
  - By value: pass contents [$sp+4] in $a0

```
int n;                          // n at memory location 4($sp)
foo(n);
        lw $a0,4(sp)
        jal foo
```

  - By reference: pass address $sp+4 in $a0

```
int n;                          // n at memory location 4($sp)
bar(&n);
        add $a0,$sp,4
        jal bar
```

Swap function

- Write a MIPS Assembly function to swap two values in memory x & y

```
.data
x: .word 12
y: .word 23
```

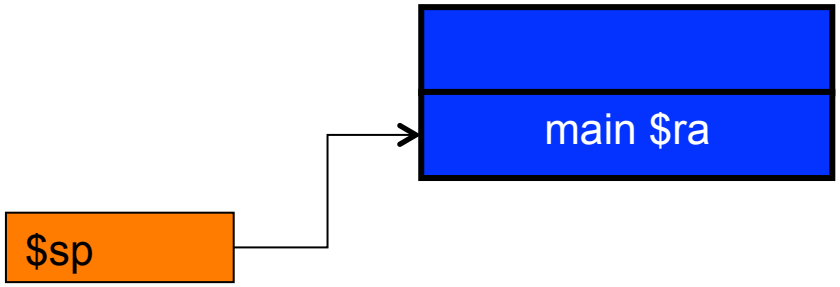
Procedure Calls

```
#include<stdio.h>

int x=4,y=5,z=0;     /* global variables */

int bar(int a, int b) {
 int result;
 result = a+b;
 return (result);
}

int foo(int x, int y) {
    int tmp1, tmp2, res;
    tmp1 = x+x;
    tmp2 = y-3;
    res = bar(tmp1, tmp2);
  return(res);
}
```

```
int main() {
   z = foo(x,y);
   printf("%d\n",z);
}
```

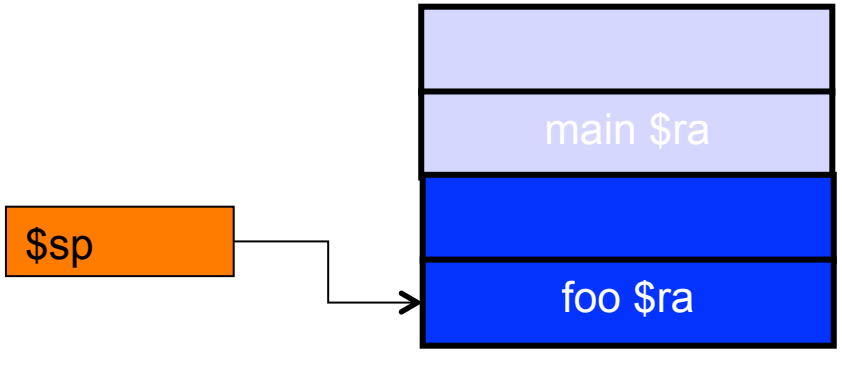
Stack for main

```
int main() {
   z = foo(x,y);
   printf("%d\n",z);
}
```



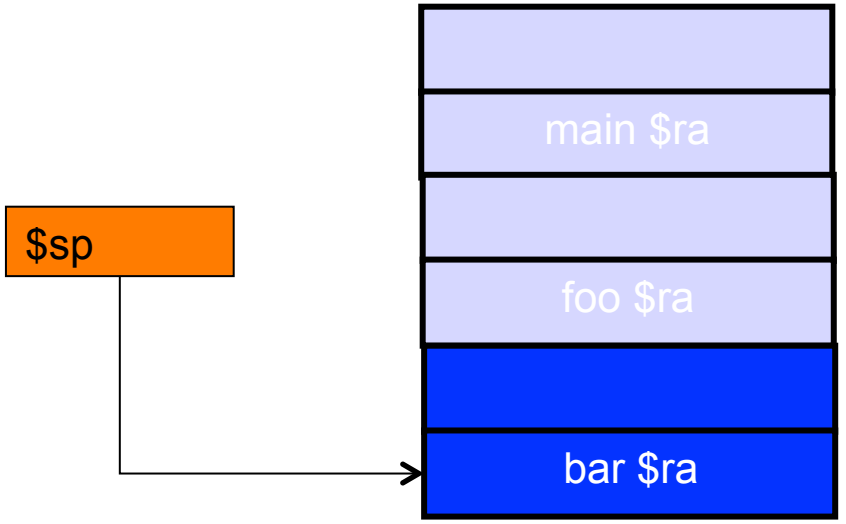
Stack for main -> foo

```
int main() {
   z = foo(x,y);
   printf("%d\n",z);
}

int foo(int x, int y) {
    int tmp1, tmp2, res;
    tmp1 = x+x;
    tmp2 = y-3;
    res = bar(tmp1, tmp2);
  return(res);
}
```



Stack for main -> foo -> bar

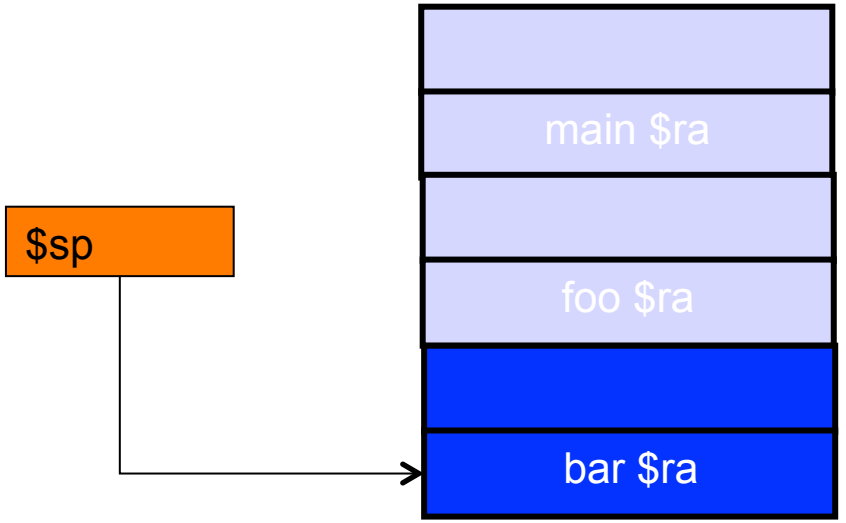
```
int main() {
   z = foo(x,y);
   printf("%d\n",z);
}

int foo(int x, int y) {
    int tmp1, tmp2, res;
    tmp1 = x+x;
    tmp2 = y-3;
    res = bar(tmp1, tmp2);
  return(res);
}
```




```
int bar(int a, int b) {
 int result;
 result = a+b;
 return (result);
}
```

Stack for main -> foo -> bar

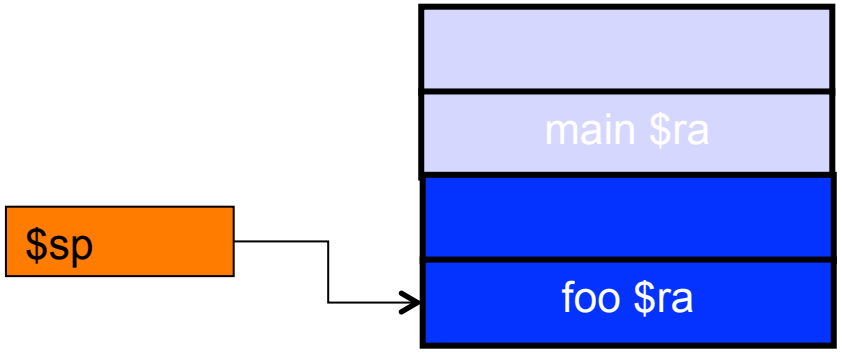
```
int main() {
   z = foo(x,y);
   printf("%d\n",z);
}

int foo(int x, int y) {
    int tmp1, tmp2, res;
    tmp1 = x+x;
    tmp2 = y-3;
    res = bar(tmp1, tmp2);
  return(res);
}
```




```
int bar(int a, int b) {
 int result;
 result = a+b;
 return (result);
}
```

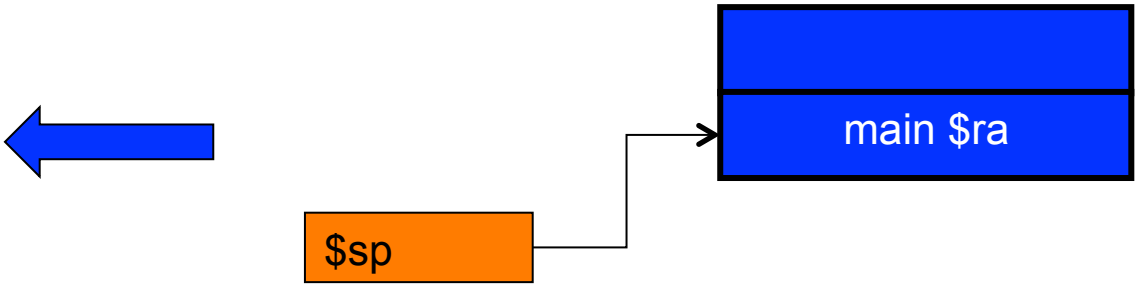
Stack for main -> foo

```
int main() {
   z = foo(x,y);
   printf("%d\n",z);
}

int foo(int x, int y) {
    int tmp1, tmp2, res;
    tmp1 = x+x;
    tmp2 = y-3;
    res = bar(tmp1, tmp2);
    return(res);
}
```



Stack for main

```
int main() {
   z = foo(x,y);
   printf("%d\n",z);
}
```



Recursion

- Nothing special is needed!
- Procedure call convention w/ stack paradigm
- Every procedure has its own local storage for variables
  - Storage is on stack or in registers
- If A calls B, and B calls C, then C returns before B returns
  - Follows a Last-in First-out (LIFO) pattern

Factorial

- Write a MIPS assembly program that computes fact(n) = n*fact(n-1) for n >= 0; where fact(0) = 1

C Code

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

Let's write MIPS…

MIPS Factorial (Using Conventions)

```
fact: addi $sp,$sp,-8       // open frame (2 words)
      sw $ra,4($sp)         // save return address
      sw $s0,0($sp)         // save $s0
      …
      add $s0,$a0,$0        // copy $a0 to $s0
      subi $a0,$a0,1        // pass arg via $a0
      jal fact              // recursive call
      mul $v0,$s0,$v0       // value returned via $v0
      …
      lw $s0,0($sp)         // restore $s0
      lw $ra,4($sp)         // restore $ra
      addi $sp,$sp,8        // collapse frame
      jr $ra                // return, value in $v0
```

+ Pass/return values via `$a0-$a3` and `$v0-$v1` rather than stack
+ Save/restore 2 registers (`$s0, $ra`) rather than 31 (excl. $0)

Miscellaneous MIPS Instructions

break A breakpoint trap occurs, transfers control to exception handler

syscall A system trap occurs, transfers control to exception handler

coprocessor instrs Support for floating point.

TLB instructions Support for virtual memory: discussed later

restore from exception Restores previous interrupt mask & kernel/user mode bits into status register

load word left/right Supports unaligned word loads

store word left/right Supports unaligned word stores

MIPS Factorial (Naïve version)

```
fact:   addi $sp,$sp,-128      // open frame (32 words of storage)
        sw $ra,124($sp)        // save all 32 registers
        sw $1,120($sp)
        sw $2,116($sp)
        …
        lw $s0,128($sp)        // read argument from caller's frame
        subi $s1,$s0,1
        sw $s1,0($sp)          // store (argument-1) to frame
        jal fact               // recursive call
        lw $s1,-4($sp)         // read return value from frame
        mul $s1,$s1,$s0        // multiply
        …
        lw $2,116($sp)         // restore all 32 registers
        lw $1,120($sp)
        lw $ra,124($sp)
        sw $s1,124($sp)        // return value below caller's frame
        addi $sp,$sp,128       // collapse frame
        jr $ra                 // return
```

Note: code ignores base case of recursion (should return 1 if arg==1)

Summary

- System Call
- Procedure Calls

Next Time

- Recursion, Other Instruction Sets

Reading

- An overview of SPIM and the MIPS32 instruction set @ http://spimsimulator.sourceforge.net/HP_AppA.pdf
- MIPS Quick Ref (PDF in docs section of web site)